UBL: *U*nsupervised *B*ehavior *L*earning for Predicting Performance Anomalies in Virtualized Cloud Systems

Daniel J. Dean, Hiep Nguyen, Xiaohui Gu
Department of Computer Science
North Carolina State University
{djdean2,hcnguye3}@ncsu.edu, gu@csc.ncsu.edu

ABSTRACT

Infrastructure-as-a-Service (IaaS) clouds are prone to performance anomalies due to their complex nature. Although previous work has shown the effectiveness of using statistical learning to detect performance anomalies, existing schemes often assume labelled training data, which requires significant human effort and can only handle previously known anomalies. We present an *U*nsupervised *B*ehavior *L*earning (UBL) system for IaaS cloud computing infrastructures. UBL leverages Self-Organizing Maps to capture emergent system behaviors and predict unknown anomalies. For scalability, UBL uses residual resources in the cloud infrastructure for behavior learning and anomaly prediction with little add-on cost. We have implemented a prototype of the UBL system on top of the Xen platform and conducted extensive experiments using a range of distributed systems. Our results show that UBL can predict performance anomalies with high accuracy and achieve sufficient lead time for automatic anomaly prevention. UBL supports large-scale infrastructure-wide behavior learning with negligible overhead.

Categories and Subject Descriptors

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

General Terms

Reliability, Management, Experimentation

Keywords

Unsupervised System Behavior Learning, Cloud Computing, Anomaly Prediction

1. INTRODUCTION

Infrastructure-as-a-Service (IaaS) cloud infrastructures [1] allow users to lease resources in a pay-as-you-go fashion. Due to its inherent complexity and sharing nature, the cloud system is prone to performance anomalies due to various reasons such as resource contentions, software bugs, or hardware failures. It is a daunting task for system administrators to manually keep track of the execution status of tens of thousands of virtual machines (VMs) all the time. Moreover, delayed anomaly detection can cause long service level objective (SLO) violation time, which is often associated with a large financial penalty. Thus, it is highly desirable to provide automatic anomaly prediction techniques that can forecast whether a system will enter an anomalous state and trigger proper preventive actions to steer the system away from the anomalous state.

It is challenging to achieve efficient anomaly management for large-scale IaaS cloud infrastructures. First, applications running inside the cloud often appear as black-box to the cloud service provider. Therefore, it is impractical to apply previous white-box or grey-box anomaly detection techniques (e.g., [7]) which require application instrumentation. Second, a large-scale cloud infrastructure often runs thousands of applications concurrently. The anomaly management scheme itself must be light-weight and should operate in an online fashion. Third, it is difficult, if not totally impossible, to obtain *labelled* training data (i.e., measurement samples associated with normal or abnormal labels) from production cloud systems. As a result, it is hard to apply previous supervised learning techniques [15, 17, 33] for monitoring production cloud systems. More importantly, supervised learning techniques can only detect previously known anomalies.

In this paper, we present the design and implementation of an *U*nsupervised *B*ehavior *L*earning (UBL) system for virtualized cloud computing infrastructures. UBL does not require any labelled training data, allowing it capture *emergent* system behaviors. This makes it possible for UBL to predict both known anomalies and *unknown* anomalies. UBL employs a set of continuous VM behavior learning modules to capture the patterns of normal operations of all application VMs. To avoid manual data labeling and capture emergent system behaviors, UBL leverages an unsupervised learning method called the Self Organizing Map (SOM) [24]. We chose the SOM because it is capable of capturing complex system behaviors while being computationally less expensive than comparable approaches such as k-nearest neighbor [32]. To predict anomalies, UBL looks for early deviations from normal system behaviors. UBL only relies on system-level metrics that can be easily acquired via the hypervisor or guest OS to achieve black-box anomaly prediction.

For scalability, UBL takes a *decentralized* and *virtualized* learning approach that leverages *residual* resources in the cloud infrastructure for behavior learning and anomaly prediction. It encapsulates the behavior analysis program within a set of special *learning VMs*. We then use the Xen credit scheduler [8] to enforce the learning VM to only use residual resources without affecting other co-located application VMs. We can also easily migrate the learning VM between different hosts using live VM migrations [14] to utilize time-varying residual resources on different hosts.

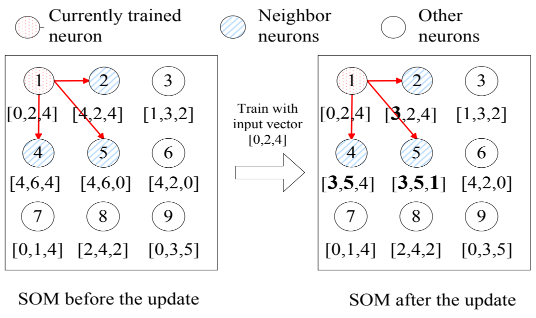


Figure 1: SOM training process.

Specially, this paper makes the following contributions:

- We show how to use the SOM learning technique to achieve efficient unsupervised system behavior learning.
- We describe how to leverage the system behavior model along with the node neighborhood area size analysis to predict emergent system anomalies and infer anomaly causes.
- We present a virtualized system behavior learning scheme that leverages the virtualization technology to efficiently and safely harvest residual resources in the cloud to achieve scalable online system behavior learning and anomaly prediction with little add-on cost.

We have implemented a prototype of UBL on top of the Xen platform [8]. We have deployed and tested UBL on the NCSU's virtual computing lab (VCL) [6] that operates in a similar way as Amazon EC2 [1]. We conducted extensive experiments using a range of real distributed systems: 1) RUBiS, an online auction benchmark [4], 2) IBM System S, a commercial stream processing system [18], and 3) Hadoop, an open source implementation of MapReduce framework [2]. Our experimental results show that UBL can predict a range of performance anomalies with 5.9-87.7% higher true positive rates and 3.3-84.5% lower false alarm rates than other alternative schemes. UBL can achieve sufficient lead time in most cases for the system to take just-in-time preventative actions [34]. Our prototype implementation shows that UBL is feasible and imposes negligible overhead for the cloud system.

The remainder of the paper is organized as follows. Section 2 presents the design details of UBL. Section 3 presents the experimental evaluation. Section 4 compares our work with related work. Finally, Section 5 concludes this paper.

2. SYSTEM DESIGN

In this section, we present the design details of the UBL system. We first describe our continuous runtime system behavior learning scheme. We then present our unsupervised anomaly prediction algorithm that can raise advance alerts about both known and unknown anomalies. Next, we present our decentralized learning framework to achieve scalable and low-cost cloud infrastructure behavior learning.

2.1 Online System Behavior Learning

It is a challenging task to achieve efficient online system behavior learning for large-scale cloud computing infrastructures. The learning scheme first needs to achieve scalability, which can induce behavior models for a large number of application components on-the-fly without imposing excessive learning overhead. Furthermore, system metric measurements for real world distributed applications are often fluctuating due to dynamic workloads or measurement noises, which requires a robust learning scheme. We chose to use the SOM learning technique in this work to achieve scalable and efficient system behavior learning.

The SOM maps a high dimensional input space into a low dimensional map space (usually two dimensions) while preserving the topological properties of the original input space (i.e., two similar samples will be projected to close positions in the map). Thus, the SOM can handle multi-variant system behavior learning well without missing any representative behaviors. Specially, we collect a vector of measurements $D(t) = [x_1, x_2, ..., x_n]$ continuously for each VM, where x_i denotes one system-level metrics (e.g., CPU, memory, disk I/O, or network traffic), and use the measurement vectors as inputs to train SOMs. UBL can dynamically induce a SOM for each VM to capture the VM's behaviors.

A SOM is composed of a set of neurons arranged in a lattice, illustrated by Figure 1. Each neuron is associated with a weight vector and a coordinate in the map. Weight vectors should be the same length as the measurement vectors (i.e., $D(t)$), which are dynamically updated based on the values of the measurement vectors in the training data. UBL uses SOMs to model system behaviors in two different phases: learning and mapping. We first describe the learning phase. We will present the mapping phase in detail in the next subsection.

During learning, the SOM uses a competitive learning process to adjust the weight vectors of different neurons. The competitive learning process works by comparing the Euclidean distance of the input measurement vector to each neuron's weight vector in the map. The neuron with the smallest Euclidian distance is selected as the currently trained neuron. For example, Figure 1 shows a map consisting of 9 neurons being trained with an input measurement vector of [0,2,4]. We first calculate the Euclidean distance to every neuron. Neuron 1 is selected as the currently trained neuron because it has the smallest Euclidean distance to the measurement vector. That neuron's values along with its neighbor neurons are then updated. In this example, we define our neighborhood to be the neurons in a radius of $r = 1$. Striped neurons (neurons 2, 4, and 5) are the neurons in neuron 1's neighborhood. The general formula for updating the weight vector of a given neuron at time *t* is given in Equation 1. We use *W(t)* and *D(t)* to define the weight vector and the input vector at time instance *t*, respectively. $N(v, t)$ denotes the neighborhood function (e.g., a Gaussian function) which depends on the lattice distance to a neighbor neuron v. $L(t)$ denotes a learning coefficient that can be applied to modify how much each weight vector is changed as learning proceeds.

$$W(t + 1) = W(t) + N(v, t)L(t)(D(t) - W(t)) \qquad (1)$$

Figure 1 illustrates the learning process using Equation 1 with a learning coefficient of 1 and a neighborhood function of $\frac{1}{4}$. We use a simple function here to illustrate the learning process, but more complex neighborhood functions are used in non-trivial applications, which we discuss further in Section 3. For example, neuron 2 has a weight vector of [4,2,4] and the input vector is [0,2,4]. Taking the difference between the input vector and the weight vector gives a value of [-4,0,0] which is then multiplied by 1 and $\frac{1}{4}$. This gives value of [-1,0,0] which is then added to the initial weight of [4,2,4] to give a final updated value of [3,2,4] to neuron 2. All updated values are shown in bold. The intuition behind this approach is to make the currently trained neuron and the neurons in its neighborhood converge to the input space.

When each input vector has been used to update the map multiple times (e.g., 10 in our experiments), learning is complete. At this point, the weight vectors of neurons represent a generalization of the whole measurement vector space. Thus, the SOM can capture the *normal* system behaviors under different workloads. We define this phase to be the *bootstrap* learning phase. UBL also supports incremental updates which can continuously adjust the SOM with new measurement vectors. However, too many incremental updates may degrade the quality of the SOM as all weight vectors may converge to a small number of vector values. This can happen when the system starts to process a completely different new workload. In this case, we can re-bootstrap the SOM with new measurement data to maintain the quality of the SOM.

When applying the SOM to learning real system behaviors, we found that UBL needs to address several metric pre-precessing problems in order to achieve efficiency. First, different system metric values can have very different ranges in their raw form. For example, the MEM_USAGE metric ranges from 0 to 2048, while the CPU_USAGE metric expressed as a utilization percentage from 0 to 100. This is problematic for our map as large data ranges would require a large number of neurons. To address this problem, we normalize all metric values to the range [0,100] by looking at the maximum value of each metric in the learning data. We chose to normalize our values this way because we found using the absolute maximum possible value sometimes produced distorted normalized values that distribute within a small range. For example, during normal operation, the observed network traffic should be much less than the maximum traffic possible. Normalizing to the maximum possible value would mean the network traffic value would only cover a small range.

During online operation, some measurement values might exceed the maximum value in the training data. This will cause some normalized metric values to be greater than 100. However, we found this does not cause an unexpected result. By doing this, we can significantly reduce the number of neurons needed for covering the whole measurement space while still capturing the patterns of the system behavior. We also filter constant metric values which have no effect on our system to further decrease the memory footprint for storing the training data. Second, some real system metric values (e.g., memory usage in Hadoop) are highly fluctuating. We might induce a map with poor quality using the raw monitoring data. To address the problem, we apply k-point moving average filter to smooth the raw monitoring data. The length of k represents the degree of smoothing, which computes an average value for the current value with the k metric values before the current value.

Determining how to properly configure and initialize the map is critical for the performance of SOM. We first need to decide the size of the map we should use for modeling a VM's behavior. We found a matrix topology based map with dimensions 32x32 consisting of 1024 total neurons works well for all the applications we tested. As values have been normalized to [0,100], we initialize each weight vector element to a random value between 0 and 100. We found random initialization to be necessary because initializing the weight vectors to a set of known values causes the produced map to be heavily biased towards the known values. This decreases the ability of the map to predict unknown values.

Due to the randomness used in weight vector initialization, we found the random vectors generated in some maps would only represent a subset of the training data values. This caused only a small portion of neurons to be trained, which in turn led to a poor quality map. To address this problem, we use K-fold cross validation as part of our learning phase, which works as follows. The training data is first partitioned into K parts denoted by $D_1, \cdots, D_K$. The validation process takes K rounds to complete. In round i, $1 \leq i \leq K$, D_i is selected to be the testing data while the other $(K-1)$ parts $D_1, \cdots, D_{i-1}, D_{i+1}, \cdots, D_K$ are used as the training data.

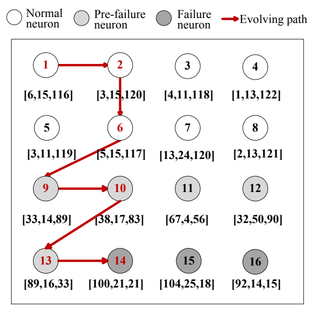


Figure 2: An example path showing the system evolution from normal to failure.

We collect various correct and incorrect classification statistics to compute the accuracy of each map. Since UBL is designed to be unsupervised, we only use unlabeled *normal* data to train the map. UBL relies on the SLO feedback from the application or some external SLO monitoring tool [11] to select normal data. Suppose N_{fp} is the number of false positives, when UBL raised an alarm yet no anomaly was found. N_{fn} is the number of false negatives, when UBL failed to raise an alarm but the current sample was an anomaly. N_{tp} is the number of true positives, when UBL raised an alarm and there was an anomaly. N_{tn} is the number of true negatives, when UBL did not raise an alarm and the current sample was normal. Since our training data are all normal data, $N_{fn} = N_{tp} = 0$. The accuracy metric for each map is calculated using the standard way as follows:

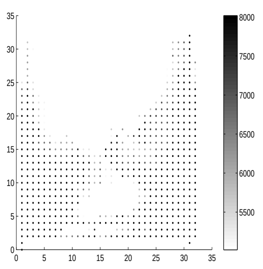
$$A = \frac{N_{tn} + N_{tp}}{N_{tn} + N_{fp} + N_{fn} + N_{tp}} \quad (2)$$

The cross validation module selects the map with the best accuracy as the final trained map. We use the same Gaussian neighborhood function and the same constant learning coefficient among all datasets. We also conducted sensitivity experiments to show how those parameter values affect the performance of UBL. We will present those results in Section 3.

2.2 Unsupervised Anomaly Prediction

Performance anomalies, such as SLO violations, in distributed systems often manifest as anomalous changes in system-level metrics. Faults do not always cause a SLO failure immediately. Instead there is a time window from when the fault occurs to the actual time of failure. Therefore, at any given time, a system can be thought to be operating in one of three states: normal, pre-failure, or failure. Additionally, the system typically first enters the pre-failure state before entering the failure state. Since the SOM is able to maintain the topological properties of the measurement samples, we can observe when the system enters the pre-failure state and moves to the failure state. Figure 2 shows an example using a real system failure where the failing system follows a path through the SOM over time. UBL can raise an advanced alarm when the system leaves the normal state but has not yet entered the failure state.

To decide the system state represented by each neuron, UBL calculates a neighborhood area size for each neuron in the SOM. As mentioned in Section 2.1, when neurons in the SOM are updated with training data, we also adjust the weight vectors of their neighboring neurons. After learning, frequently trained neurons will have modified the weight vector values of their neighboring neurons with the same input measurement vectors. As a result, the weight vectors of the neurons that are frequently trained will look similar to the weight vectors of their neighboring neurons.


(a) Network Hog (RUBiS)

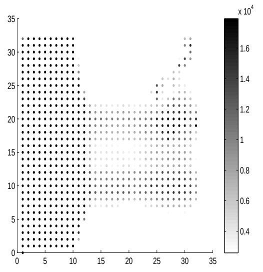

(b) MemLeak (SystemS)

Figure 3: Grey-scale visualization of the SOM models for the RUBiS with the Network Hog fault and System S with the MemLeak fault. Darker neurons have larger neighborhood area sizes while lighter neurons have smaller neighborhood area sizes.

Since systems are usually in the normal state, neurons representing the normal state will be more frequently trained than the neurons representing the pre-failure or failure states. Thus, we will have clusters of neurons representing different normal system behaviors. We calculate a neighborhood area size value for each neuron by examining the immediate neighbors of each neuron. As our lattice topology is a two-dimensional grid, this means we examine the top, left, right, and bottom neighbors. We calculate the Manhattan distance between two neurons N_i, N_j with weight vectors $W_i = [w_{1,i}, \cdots, w_{k,i}]$, $W_j = [w_{1,j}, \cdots, w_{k,j}]$ respectively, as follows:

$$M(N_i, N_j) = \sum_{l=1}^{k} |w_{l,i} - w_{l,j}| \tag{3}$$

We define the neighborhood area size for a neuron N_i as the sum of Manhattan distance between the neuron N_i and its top, left, right and bottom immediate neighbors denoted by N_T, N_L, N_R and N_B, as follows:

$$S(N_i) = \sum_{X \in \{N_T, N_L, N_R, N_B\}} M(N_i, X) \tag{4}$$

UBL determines if a neuron is normal or anomalous by looking at the neighborhood area size of that neuron. If the neighborhood area size is small, we know that the neuron we have mapped to is in a tight cluster of neurons, meaning the neuron is normal. On the other hand, if a neuron maps to a neuron with a large neighborhood area value, we know that the neuron is not close to other neurons, and thus, probably anomalous. For example, in Figure 2, the calculated neighborhood area size for neuron 6 (a normal neuron) of would be the sum of the differences to neighbors 2,5,7, and 10, which is 102. The neighborhood area size of neuron 10 (a pre-failure neuron), on the other hand, is the sum to neighbors 6, 9, 14, and 11, which is 280.

Figure 3 shows two maps after bootstrap learning has completed: one is for the RUBiS web server with a network hog bug and the other is for one faulty component in System S including a memory leak bug. We use gray-scale visualization to illustrate the behavior patterns. Darker neurons represent anomalous behaviors while lighter neurons represent normal behaviors. Once learning is complete, we can clearly see different systems present distinct behavior patterns that can be captured by the SOM.

During application runtime, we map each measurement vector to a neuron using the same Euclidean distance metric as the learning phase. We look at the neighborhood area size of the mapped neuron. If the neighborhood area size is below the threshold for the map, that means the sample has mapped to a neuron which is close to many other neurons. We consider this sample to be a normal sample and do not raise an alarm. However, if the sample maps to a neuron with an area value greater than or equal to our threshold value, this sample represents something we rarely see during learning. We consider this type of sample to be anomalous. Transient fluctuations in system metrics due to noise can still be present even after data smoothing. Those momentary fluctuations may be mapped to anomalous neurons, although it would be incorrect to raise an alarm in this case. As a result, we raise an alarm only when the system identifies three consecutive anomalous samples.

Determining a neighborhood area size threshold to differentiate normal and anomalous neurons is integral to the accuracy of the UBL system. If the threshold is set too high, we cannot raise an alarm early enough and may miss some anomalies. Alternatively, if we set the threshold too low, we might raise too many alarms, including false alarms. Additionally, neighborhood area size values vary from map to map depending on the range of values in the dataset. To address this issue, we set the threshold value based on a percentile instead of a fixed value. We sort all calculated neighborhood area size values and set the threshold value to be the value at a selected percentile. We found a percentile value of 85% is able to achieve good results across all datasets in our experiments. We further examine the effect of the threshold on accuracy in Section 3.

2.3 Anomaly Cause Inference

Determining the root cause of an anomaly is a highly non-trivial task. UBL is able to ameliorate this task by giving a hint as to what metrics are the top contributors to an anomaly. While this does not directly identify the root cause of the anomaly, it provides a clue of where to start looking. As the SOM preserves the topological properties of the measurement space, UBL can use this information to identify the faulty metric causing an anomaly. The basic idea is to look at the difference between anomalous neurons and normal neurons, and output the metrics that differ most as faulty metrics. Specifically, when we map a measurement sample to an anomalous neuron, we calculate the Euclidean distance from the mapped anomalous neuron to a set of nearby normal neurons. Here, it is necessary to avoid comparing with anomalous neighbor neurons as they represent unknown states and therefore may give incorrect anomaly cause hints. We examine the neighborhood area value for each neuron first. If it is above our threshold, we ignore it and move on to the next neuron in our neighborhood. If no normal neuron is found in the anomalous neuron's neighborhood, we expand our distance calculation to include more neurons in the map. In order to ensure we get a good representation of normal metrics, we select Q normal nearby neurons (e.g., Q = 5 in our experiments).

Once a set of normal neurons has been found, we calculate difference between the individual metric values of each normal neuron and those of the anomalous neuron. As the change can be positive or negative, we take the absolute value of the calculated difference. We then sort the metric differences from the highest to the lowest to determine a ranking order. After this process completes, we will have Q metric ranking lists. Finally, we examine the ranking orders of each of the Q rankings to determine a final order. To do this, we use majority voting. Each list votes for which metric it had identified as having the largest difference in values. We then output the metric with the most votes as the first ranked metric, the metric that has the 2nd most is the second ranked metric, and so on. Ties indicate no consensus could be reached and we output the metric that happens to come first in the output list construction. While we have found ties to be rare, a potential refinement of this approach would be to use the total difference of each metric to break ties. As an example, suppose three ranking lists rank CPU usage as the top anomaly cause but two other ranking lists rank Memory usage as the top cause. We will output CPU usage as the top anomaly cause as it has been ranked the top anomaly cause by a majority.

2.4 Decentralized Behavior Learning

Based on the monitoring results of a production cloud infrastructure, we observe that many hosts have less than 100% resource utilization. UBL leverages these *residual* resources to perform behavior learning as background tasks that are co-located with different application VMs (foreground tasks) on distributed hosts. Through this, we can achieve scalable infrastructure-wide behavior learning with minimum add-on cost. Our approach is particularly amenable for energy saving since a large portion of energy consumption is wasted in machine's idle state. To avoid affecting the foreground tasks, UBL takes advantage of the isolation provided by *Xen* to encapsulate itself within a special *learning VM*. We then use weight-based priority scheduling provided by the Xen platform to ensure the learning VM has a minimal effect on the foreground workload. Specifically, we assign a very low weight (e.g., 8) to all learning VMs which causes them to yield resources to the foreground application VMs.

UBL monitors the residual resources on each host by aggregating the resource consumption of all the VMs running on the host. If we find the available residual resources are insufficient, we employ live migration to move the learning VM to a host with sufficient residual resources. UBL maintains a resource demand signature for each learning VM and the residual resource signature for each host [19]. UBL finds a suitable host for migrating the learning VM by matching the resource demand signature of the learning VM with the residual resource signature of the host. We define a host to be overloaded when the total resource consumption of the host exceeds a certain threshold (e.g., $>$ 90%). In this case, we relocate all the learning VMs running on that host to the hosts with suitable residual resources.

3. EXPERIMENTAL EVALUATION

We have implemented a prototype of UBL on top of the Xen platform and conducted extensive experiments using three benchmark systems: the RUBiS multi-tier online auction web application (EJB version) [4], IBM System S data stream processing system [18], and the Hadoop MapReduce framework [2]. We begin by describing our evaluation methodology. We then present our results.

3.1 Evaluation Methodology

Our experiments were conducted on the Virtual Computing Lab (VCL) infrastructure [6] which operates in a similar way as Amazon EC2 [1]. Each VCL host has a dual-core Xeon 3.0GHz CPU and 4GB memory, and runs 64bit CentOS 5.2 with Xen 3.0.3. The guest VMs also run 64bit CentOS 5.2 .

UBL monitors VMs' resource demands from domain 0, using the `libxenstat` and `libvirt` libraries to collect resource usage information (e.g., CPU usage, memory allocation, network I/O, disk I/O) for both domain 0 and guest VMs. UBL also uses a small memory monitoring daemon within each VM to get memory usage statistics (through the /proc interface in Linux). The sampling interval is 1 second.

We have chosen three benchmark systems to evaluate UBL in order to demonstrate the agnosticism necessary for such a system to be used in the real world. Moreover, UBL can handle dynamic applications processing time-varying workloads. To demonstrate this, we drive all the benchmark applications using dynamic workload intensity observed in real world online services. We injected faults at different times while the system was under dynamic workload. Each experiment duration varies slightly but all last about one hour. Fault injections also vary slightly depending on the fault type but all last between 1 and 5 minutes. For each fault injection, we repeated the experiment 30 to 40 times. We now describe all the systems and fault injections in detail as follows.

RUBiS online auction benchmark: We used the three-tier online auction benchmark system RUBiS (EJB version) with one web server, two application servers, and one database server. In order to evaluate our system under workloads with realistic time variations, we used a client workload generator that emulates the workload intensity observed in the NASA web server trace beginning at 00:00:00 July 1, 1995 from the IRCache Internet traffic archive [5] to modulate the request rate of our RUBiS benchmark. The client workload generator also tracks the response time of the HTTP requests it made. A SLO violation is marked if the average request response time is larger than a pre-defined threshold (e.g., 100ms).

We injected the following faults in RUBiS: 1) *Memleak*: we start a memory-intensive program in the VM running the database server; 2) *CpuLeak*: a CPU-bound program with gradually increasing CPU consumptions competes CPU with the database server inside the same VM; and 3) *NetHog*: we use httperf [3] tool to send a large number of http requests to the web server.

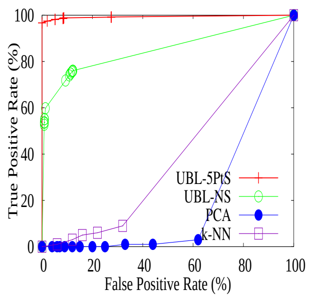
IBM System S: We used the IBM System S that is a commercial high-performance data stream processing system. Each System S application consists of a set of inter-connected processing elements (PEs). We measured the average per-tuple processing time. A SLO violation is marked if the average processing time is larger than a pre-defined threshold (e.g., 20ms). In order to evaluate our system under dynamic workloads with realistic time variations, we used the workload intensity observed in the ClarkNet web server trace beginning at 1995-08-28:00.00 from the IRCache Internet traffic archive [5] to modulate the data arrival rate.

For System S, we injected the following faults: 1)*MemLeak*: we start a memory-intensive program in one randomly selected PE; 2) *CpuHog*: a CPU-bound program competes CPU with one randomly selected PE within the same VM; and 3) *Bottleneck*: we make one PE the bottleneck in the application by setting a low CPU cap for the VM running the PE.

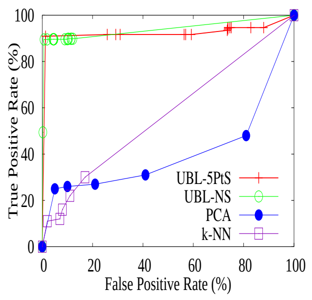
Hadoop: We run Hadoop sorting application that is one of the sample applications provided by the Hadoop distribution. We measure the progress score of the job through Hadoop API. A SLO violation is marked when the job does not make any progress (i.e., 0 progress score increase). We use 3 VMs for Map tasks and 6 VMs for Reduce tasks. The number of map slots on each VM running map tasks is set to 2, and the number of Reduce slots on each VM running reduce tasks is set to 1. We use this configuration because the reduce task requires much more disk and memory space than the map task in the sorting application. Since this is a small Hadoop cluster, the JobTracker and NameNode are very light-weight. We colocate them together with the first reduce VM. The data size we process is 12GB, which is generated using the RandomWriter application.

For Hadoop, we injected two types of faults into all the VMs running the map tasks: 1) *MemLeak*: we injected a memory leak bug into all the map tasks, which repeatedly allocates certain memory from the heap without releasing; and 2) *CpuHog*: we injected an infinite loop bug into all the map tasks.

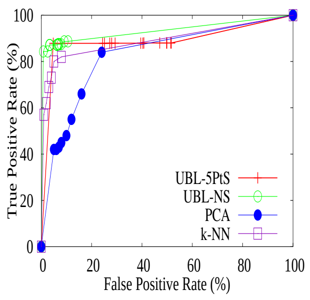
We evaluate the anomaly prediction accuracy using the standard *receiver operating characteristic* (ROC) curves. ROC curves can effectively show the tradeoff between the true positive rate (A_T)



(a) MemLeak

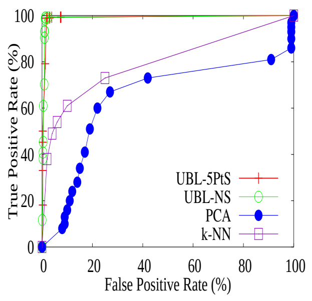


(b) CpuLeak

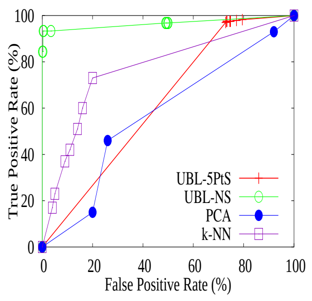


(c) NetHog

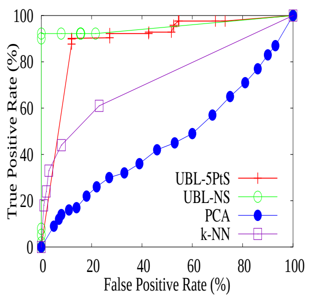
Figure 4: Performance anomaly prediction accuracy comparison for RUBiS under different faults.



(a) MemLeak



(b) CpuHog



(c) Bottleneck

Figure 5: Performance anomaly prediction accuracy comparison for IBM System S under different faults.

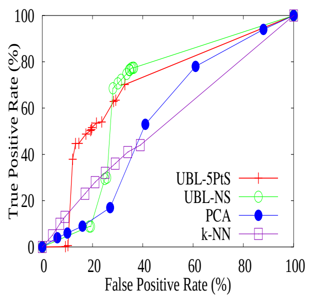
and the false positive rate (A_F) for a prediction model. We use standard *true positive rate* A_T and *false positive rate* A_F metrics given in equation 5. The N_{tp}, N_{fp}, N_{tn}, and N_{fn} values are the same as those described in Section 2.

$$A_T = \frac{N_{tp}}{N_{tp} + N_{fn}}, A_F = \frac{N_{fp}}{N_{fp} + N_{tn}} \tag{5}$$

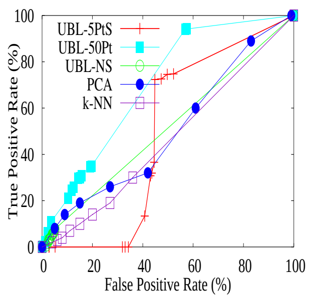
We say the prediction model makes a true positive prediction if it raises an anomaly alert at time t_1 and the anomaly indeed happens at time t_2, $t_1 < t_2 < t_1 + W$, where W denotes the upper-bound of the anomaly pending time.[1]. Otherwise, we say the prediction model fails to make a correct prediction. If the predictor raises an alert and the predicted anomaly does not happen within the $t_1 + W$, we say that the prediction model raises a false alarm. We further evaluate the prediction capability of UBL using *achieved lead time*, which we define to be the amount of lead time we give prior to a SLO violation occurring. For example, if we raise an alarm at time *t* and the actual SLO violation occurs at time *t*+20 seconds, we have achieved a lead time of 20 seconds.

For comparison, we also implemented a set of commonly used unsupervised learning schemes: 1) the *PCA* scheme uses principle component analysis to identify normal and anomalous samples [26]; and 2) the *k-NN* scheme calculates a k-nearest neighbor distance for each measurement sample to identify normal and anomalous samples [32]. Different from UBL, both PCA and k-NN models need to be trained with both normal and anomalous data. In contrast, UBL does not require the training data to contain anomalous data. We use *UBL-NS* to denote the UBL scheme without applying any data smoothing. We use *UBL-kPtS* to represent the UBL scheme using the k-point moving average smoothing. Through experimentation, we have defined our map to be 32x32 nodes, the neighborhood of each node to have a radius of 4, the learning factor to be a constant 0.7, and the neighborhood function to be a Gaussian function. We use 3-fold cross validation to select the best map among three randomly initialized map. We have also conducted sensitivity study experiments on those parameters, which will be presented in the next subsection.

[1]We have determined an appropriate anomaly pending time upper-bound W for each dataset by manually examining the fault injection time to the SLO violation time. For example, if a fault is injected at time $t = 20$ and a SLO violation is observed at time $t = 30$, our window size would be 10.



(a) CpuHog



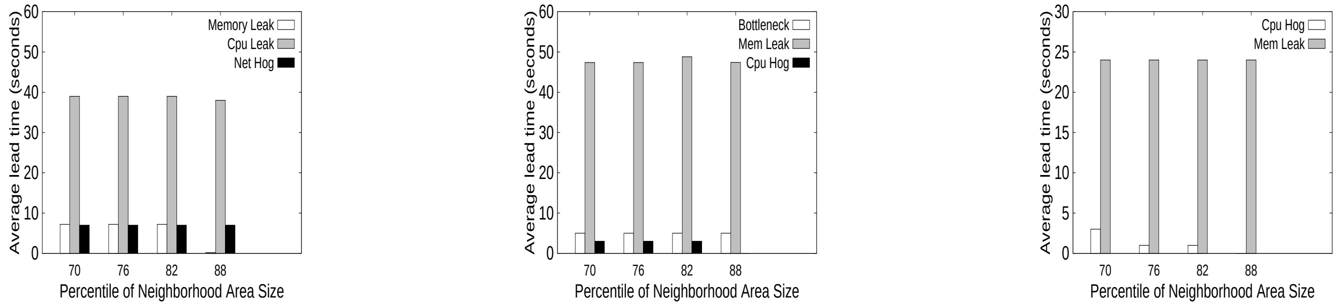
(b) MemLeak

Figure 6: Performance anomaly prediction accuracy comparison for Hadoop under different faults.

3.2 Results and Analysis

3.2.1 Prediction Accuracy Results

We now present the anomaly prediction accuracy comparison results. We acquire the ROC curves for the UBL schemes by ad-



(a) RUBiS (b) System S (c) Hadoop

Figure 7: The achieved lead time by the UBL anomaly prediction model.

justing the neighborhood area size percentile threshold (i.e., 70'th percentile to 98'th percentile). For PCA, we obtain the ROC curves by adjusting the variance threshold. The ROC curves of k-NN is calculated by adjusting the $k'th$ nearest neighbor distance threshold.

We begin with the results of our RUBiS experiments. Figure 4 shows the ROC curves for the RUBiS systems under three different faults. The memory leak dataset was our best RUBiS dataset, we were able to achieve a high true positive rate of 97% with a very low false positive rate of 2%. This is consistent with what we expected as the memory leak manifests gradually and slowly. Conversely, we see our worst results from our NetHog dataset, achieving a 87% maximum true positive rate with a corresponding 4.7% false positive rate. This is also consistent with what we expected since the NetHog fault manifests more quickly than the other two faults. In all cases, UBL consistently outperforms PCA and k-NN with higher true positive rates and lower false positive rates.

We can see the positive effect of smoothing by looking at the Memleak dataset. Due to the gradual nature of this fault, smoothing allows us to achieve approximately 20% higher true positive rates with corresponding false positive rates. This is expected as the RUBiS dataset contains quite some transient noises. Additionally, due to the gradual manifestation time of the fault, we do not smooth out any pre-failure symptoms. Therefore, we see a marked improvement between the smoothed and non-smoothed data.

Figure 5 shows the prediction accuracy results for the IBM System S application under different faults. The results show that UBL is able to achieve higher prediction accuracy than the other schemes in all cases. The best result we were able to achieve was a 98% true positive rate along with a 1.7% false positive rate in the memory leak dataset. The worst results we achieved were in the CPU Hog dataset, with a 93% true positive rate and a 0.5% false positive rate. This is expected as CPU spikes are more difficult to predict due to the rapid onset of the fault. Similarly, the Bottleneck fault is also hard to predict as the time from fault to failure is also short. The System S dataset has relatively less noise than the RUBiS datasets, so the high accuracy results are expected. Additionally, the Bottleneck and CPU Hog datasets are harder to predict than the Memleak dataset, while our results for these datasets are good, they are lower than Memleak as expected.

It is interesting to observe smoothing does not always help achieve better accuracy. In the Bottleneck and CPU Hog datasets, the best results we achieve are those without any smoothing. This is due to two reasons. First, both faults manifest very quickly. Second, System S datasets are inherently not very noisy. When we apply smoothing, even 5-point smoothing, we sometimes smooth out those critical pre-anomaly symptoms. When this happens, our model is unable to raise an alarm appropriately, leading to lower accuracy than without smoothing.

We now present the results of our Hadoop experiments shown by Figure 6. The MemLeak dataset was able to achieve the highest overall true positive rate due to the gradual nature of the fault. Hadoop is our noisiest dataset, which explains for the high false positive rates observed in these datasets. As expected, the rapid onset time of the CpuHog fault means the overall true positive rate we could achieve here was lower than the gradual memory leak dataset. As we can see, smoothing helps the MemLeak dataset, reducing the overall noise of the dataset while preserving the pre-failure symptoms. We show an additional curve to illustrate this point. Conversely, while smoothing reduced the noise of the CpuHog dataset, reducing the overall false positive rate, it also smoothed out pre-failure symptoms leading to a lower true positive rate as well. In both cases, UBL still can achieve better prediction accuracy than PCA and k-NN.

3.2.2 *Lead Time Results*

Figure 7 shows the average lead times achieved by UBL for RUBiS, System S, and Hadoop, respectively. The results shown only consider the lead time achieved for cases determined to be true positive results. We first discuss the RUBiS lead time results. We were closest to the maximum achievable lead time in the CpuLeak dataset. We achieved an average lead time of 38 seconds, with a maximum lead time of 40 seconds. The memory leak results for this dataset were the worst results we saw. We achieved an average lead time of only 7 seconds, with a maximum lead time of 50 seconds. This can be explained by variations in the data. The workload and background noise of the system caused the metrics to approach unknown levels only when the system was close to the anomaly state.

We next discuss the lead time we were able to achieve for the System S datasets. Here, we were able to achieve an average lead time of 47 seconds for the memory leak dataset with a maximum lead time possible of 50 seconds. While the lead time is lower for the CpuHog dataset, we achieved an average lead time of 3 seconds, with a maximum possible lead time of 4 seconds. Similarly, we achieved a lead time of 5 seconds in the Bottleneck dataset, with a maximum lead time of 6 seconds possible. The memory leak dataset had the best lead time because it was a gradual change with little memory fluctuation. The CpuHog and BottleNeck datasets had much shorter manifestation durations, and thus our system had little time to predict the anomaly, however we still are able to achieve results close to the maximum possible lead time.

Finally, we present the average lead time we are able to achieve in the Hadoop experiments in. The average lead time we were able to achieve in the memory leak dataset was 24 seconds. Here the

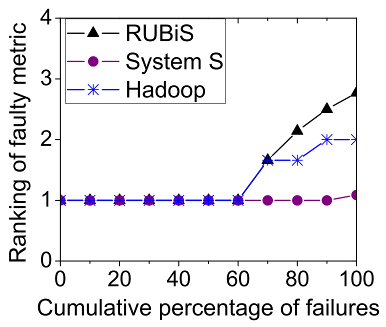


Figure 8: Ranking results of the faulty metrics in different failure instances. The Y axis is the faulty metric rank as determined by UBL while the X axis represents the total number of faults observed.

maximum lead time possible was 25 seconds. In this case, UBL is able to quickly determine the pattern is not normal and raise an early alarm. In contrast, the CpuHog lead time is lower, but as before this is due to the rapid onset time. We achieved an average lead time of 3 seconds with a maximum possible lead time of 4 seconds.

All in all, UBL can achieve close to maximum possible lead time for different faults tested in our experiments. Our previous study [34] shows that we can take local anomaly prevention actions such as VM resource scaling within one second and more costly anomaly preventions such as live VM migration within 10 to 30 seconds. Thus, the lead time achieved by UBL is sufficient in most cases for the cloud system to provide automatic anomaly preventions.

3.2.3 *Anomaly Cause Inference Results*

We now present our anomaly cause inference results shown by Figure 8. We consider the faulty metric to be the metrics most closely associated with a given failure. For example, for the MemLeak datasets, we consider the memory metric as the faulty metric. The figure shows the ranking of the faulty metric in the rank list output by UBL. As the Figure shows, UBL can correctly rank the faulty metric as top ranked metric in most failure cases. These results indicate UBL is able to preserve the topological properties of the input measurement space and is useful for diagnosis as well as prediction. In datasets where noise is less of an issue, such as System S, UBL achieves near perfect ranking results.

3.2.4 *Scalability Results*

To demonstrate the benefit of using the decentralized approach, we first measure the CPU load and power consumption of centralized system behavior learning approach. We run 25 learning VMs on five physical hosts, each of which runs five VMs. This experiment is conducted on a small cluster in our lab since VCL hosts are not equipped with power meters. Each host has a quad-core Xeon 2.53GHz processor, 8GB memory and 1Gbps network bandwidth, and runs CentOS 5.5 64 bit with Xen 3.4.3. In each physical host, we pin down Domain 0 to one core and run all learning VMs on three other cores, each VM is configured with one virtual core.

Figure 9 shows the total CPU consumption and energy consumption of the 25 VMs using 6000 data samples. The X-axis shows different numbers of learning VMs in the training state (the rest are in the prediction state). The left Y-axis shows the total CPU consumption and the right Y-axis shows the total energy consumption. We find that in all three schemes (PCA, k-NN, UBL), the learning VMs in the training state are CPU-greedy. The total CPU

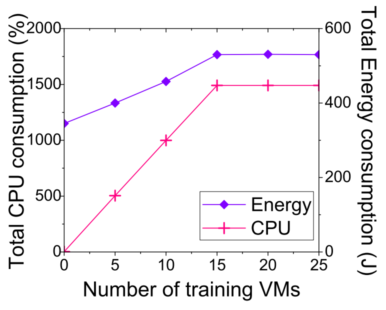


Figure 9: CPU load of 25 learning VMs running on 15 cores.

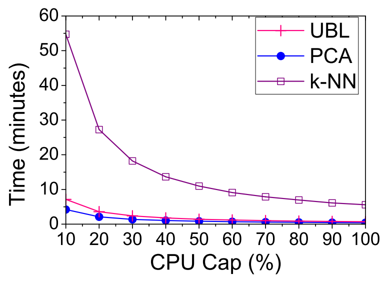


Figure 10: Training time comparison.

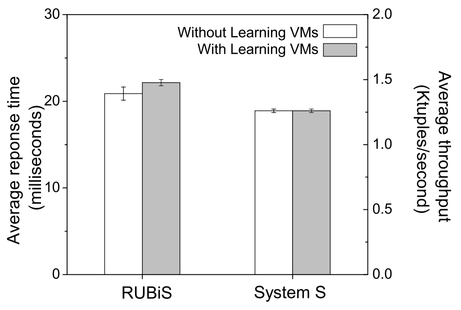
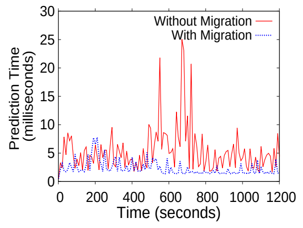


Figure 11: Learning VM impact to co-located application VMs.

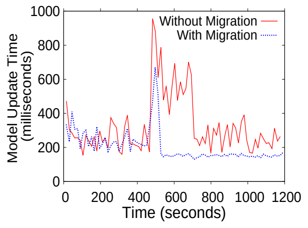
consumption and energy consumption exhibits linear growth according to the number of learning VMs in the training state. Since there are a total of 15 cores on 5 hosts, the total CPU consumption cannot exceed 1500%. This experiment shows that the centralized learning approach will not be scalable.

Figure 10 shows the training time using 6000 data samples when varying the CPU cap of the learning VM on the HGCC host. We see that the training time of UBL is similar to PCA and much faster compared to k-NN. This is expected since k-NN has higher computation complexity than PCA and UBL. We also find that the training time of UBL decreases linearly as the CPU cap increases, from 7 minutes to 42 seconds. This motivates our idea of leveraging residual resources with migration to achieve fast system learning with low cost. Additionally, this demonstrates that even with few resources available, UBL training time is reasonable.

We now examine the impact of learning VMs to the performance of co-located application VMs. We set up a small cluster in which we run both RUBiS and System S. We use the Xen credit scheduler to set the scheduling weight of the application VMs to 256 and the weight of the learning VM to 8. Figure 11 shows the performance of the RUBiS and System S with and without the presence of learning VMs. We observe that with the presence of learning VMs, the average response time of RUBiS and the throughput of System S



(a) Prediction time



(b) Model update time

Figure 12: Prediction time and model update time of UBL.

System Modules	CPU cost
VM monitoring (8 attributes)	1.33±0.09 ms
3-fold cross validation (6000 samples)	42 ± 1 sec
SOM model updating	245±54.9 ms
Anomaly prediction	2.4±2.6 ms

Table 1: UBL System overhead measurements.

	Accuracy (NetHog)	Accuracy (Memleak)
Map 25x25	97%	93%
Map 32x32	**98%**	**92.8%**
Map 40x40	97.1%	93.3%
Neighborhood size 3	98.6%	93.6%
Neighborhood size 4	**98.5%**	**92.8%**
Neighborhood size 5	97.6%	92.5%
Gaussian function height 7	98.2%	92.9%
Gaussian function height 10	**98.5%**	**92.8%**
Gaussian function height 13	98.9%	93.8%

Table 2: Sensitivity experiment results for the NetHog fault in RUBiS and MemLeak fault in System S.

have little difference compared to the case that the learning VMs are not present. The results show that the learning VMs have little impact to the performance of the co-located application VMs.

Figure 12 shows the benefit of using learning VM migration to maintain the prediction time and online model update time. Since we require the learning VMs to always yield to foreground applications, the performance of the learning VMs will be affected when the foreground applications use up the resources. The results show that, from time 450 to 700, the prediction time and the model update time increase because residual resources are low. By migrating the learning VMs to another physical host with more resources, the performance of UBL is preserved.

Finally, we evaluate the overhead of the UBL system. Table 1 lists the CPU cost of each key module in our system. The VM monitoring module runs within Domain 0 of each host and collects eight resource attributes per second. Each collection takes about 1.3 milliseconds. 3-fold cross validation is the most time-consuming operation, taking about 42 seconds. However, this step is only used during bootstrap learning phase. Incremental SOM updates take about 245 milliseconds for every 30 new data samples. Anomaly prediction takes about 2.4 milliseconds. During the normal execution, the learning VM imposes less than 1% CPU load and UBL consumes less than 16MB of memory. Overall, the overhead measurements show that UBL is light-weight, which makes it practical for online system anomaly management.

3.2.5 *Sensitivity Study*

We have conducted sensitivity experiments to study how UBL performs under different key parameter settings. Due to space limitation, we only show a subset of our results in Table 2. The accuracy values are calculated using Equation 2. We observe that UBL is not very sensitive to different parameter values and is able to achieve accuracy values which differ by less than 1% in most cases. The map size parameter has the potential to affect the accuracy of the system if it is set too low. For example, a 5x5 map is too small to effectively capture the overall pattern of the system. Additionally, if the map size is too large, the learning time becomes long. We have found map sizes in the range we list are able to give good results for all datasets we tested.

4. RELATED WORK

The idea of using machine learning methods to detect and predict anomalies, faults, and failures has been of great interest to the research community in recent years. Broadly, these approaches can be classified into supervised approaches and unsupervised approaches. Supervised approaches rely on labelled training data to accurately identify previously known anomalies. Unsupervised approaches do not require labelled training data to find problems, but generally are less accurate than supervised approaches, looking for a broader range of problems. These approaches can be further divided into detection schemes and prediction schemes. Detection schemes identify failures at the moment of failure, while prediction schemes try to predict a failure before it happens.

Supervised anomaly prediction. The most closely related work to ours is Tiresias [36], which also addresses the black-box failure prediction problem in distributed systems. Tiresias relies on external anomaly detectors to create anomaly vectors. The system then applies Dispersion Frame Technique (DFT) prediction heuristics on the anomaly vectors for anomaly prediction. Gu et al. [20] integrate Markov feature value prediction with naive Bayesian classification to predict performance anomalies. Tan et al. [33] use a hierarchical clustering technique to discover different execution contexts of a dynamic system but build context-aware prediction models to improve prediction accuracy. Different from UBL, the above works need labelled normal and failure data in the training data and do not provide anomaly cause inference. In contrast, UBL does not require any data labeling, which allows UBL to predict both known and unknown performance anomalies.

Supervised anomaly detection. Cohen et al. [16] use clustering over labelled failure data to extract failure signatures, which can be used to detect recurrent problems. Powers et al. [27] study different statistical learning methods to find the approaches that can detect performance violations in an enterprise system. Bhatia et al. [9] develop sketches of system events, which are then visualized for diagnosis by an expert. The Fa system [17] uses anomaly-based clustering to achieve automatic failure diagnosis for query processing systems. Cha et al. [12] use a signature based approach along with a bloom filter for malware detection. Bodik et al. [10] use signatures along with feature selection and a regression model to detect performance anomalies. In contrast, our approach focuses on predicting unknown anomalies and does not require prior knowledge about different failure instances.

Unsupervised anomaly detection. Previous work has proposed model-driven approach to performance anomaly detection. For example, Stewart et al. [31] instrument the OS to gather data and profile system performance using queuing models. Shen et al. [29] use a reference based approach to detect performance anomalies by looking at how metrics differ from the ideal case. Stewart et al. [30] use a transaction mix model to predict the performance given a certain workload, and hence can detect the anomaly if the observed performance is different with the predicted performance. Compared to UBL, those model-driven approaches typically require extensive model calibration using offline profiling and need to make certain assumptions about the workload type (e.g, transactions) and user request arrival patterns. In contrast, UBL is application-agnostic and does not require extensive application profiling. Cherkasova et al. [13] build regression-based transaction models and application performance signatures to provide a solution for anomaly detection considering system changes. Different from UBL, this model is designed to consider a single metric. Wang et al. [35] have used entropy based approaches to quantify the metric distribution and detect anomalies using signal processing and spike detection. Similarly, Jiang et al. [22] detect failures by looking at the entropy of clustered system metric relationships. Makanju et al. [25] assign entropy scores to event log data to detect anomalies. WAP5 [28] detects the bottleneck in distributed systems by analyzing message traces to infer the causal structure and timing of communication within these systems. Kasick et al. [23] use peer comparison to determine the root cause of a problem in a distributed environment. Jiang et al. [21] employ linear regression models to extract invariants and then track their changes to detect the anomaly in transaction systems. In contrast to the above approaches, UBL can predict future anomalies as opposed to detecting anomalies at the moment of failure. Moreover, UBL is broader in scope, designed to learn system behavior for a variety of uses. We show anomaly prediction as one of the uses of UBL.

5. CONCLUSION

In this paper, we have presented UBL, a novel black-box unsupervised behavior learning and anomaly prediction system for IaaS clouds. UBL leverages the Self-Organizing Map (SOM) learning technique to capture dynamic system behaviors without any human intervention. Based on the induced behavior model, UBL can predict previously unknown performance anomalies and provides hints for anomaly causes. UBL achieves scalable behavior learning by virtualizing and distributing the learning tasks among distributed hosts. We have implemented a prototype of UBL on top of the Xen platform and conducted extensive experiments using real world distributed systems running inside a production cloud infrastructure. Our results show that UBL can achieve high prediction accuracy with up to 98% true positive rate and 1.7% false positive rate, and raise advance alarms with up to 47 seconds lead time. UBL is lightweight, which makes it practical for large-scale cloud computing infrastructures.

6. ACKNOWLEDGMENT

This work was sponsored in part by NSF CNS0915567 grant, NSF CNS0915861 grant, NSF CAREER Award CNS1149445, U.S. Army Research Office (ARO) under grant W911NF-10-1-0273, IBM Faculty Awards and Google Research Awards. Any opinions expressed in this paper are those of the authors and do not necessarily reflect the views of NSF, ARO, or U.S. Government. The authors would like to thank the anonymous reviewers for their insightful comments as well as Zhiming Shen, Yongmin Tan, and Kamal Kc for their help.

7. REFERENCES

[1] Amazon elastic compute cloud. http://aws.amazon.com/ec2/.

[2] Apache Hadoop System. http://hadoop.apache.org/core/.

[3] Httperf. http://code.google.com/p/httperf/.

[4] RUBiS: Rice University Bidding System. http://rubis.ow2.org.

[5] The IRCache Project. http://www.ircache.net/.

[6] Virtual computing lab. http://vcl.ncsu.edu/.

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

[8] P. Barham and et al. Xen and the Art of Virtualization. In *Proc. of SOSP*, 2003.

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

[10] P. Bodik, M. Goldszmidt, and A. Fox. Hilighter: Automatically building robust signatures of performance behavior for small- and large-scale systems. In *Proc. of SysML*, 2008.

[11] D. Breitgand, M. B.-Yehuda, M. Factor, H. Kolodner, V. Kravtsov, and D. Pelleg. NAP: a building block for remediating performance bottlenecks via black box network analysis. In *Proc. ICAC*, 2009.

[12] S. K. Cha, I. Moraru, J. Jang, J. Truelove, D. Brumley, and D. G. Andersen. Splitscreen: enabling efficient, distributed malware detection. In *Proc. of NSDI*, 2010.

[13] L. Cherkasova, K. Ozonat, N. Mi, J. Symons, and E. Smirni. Anomaly? application change? or workload change? In *Proc. of DSN*, 2008.

[14] C. Clark, K. Fraser, S. Hand, J. G. Hansen, E. Jul, C. Limpach, I. Pratt, and A. Warfield. Live migration of virtual machines. In *Proc. of NSDI*, 2005.

[15] 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.

[16] 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.

[17] S. Duan, S. Babu, and K. Munagala. Fa: A system for automating failure diagnosis. In *Proc. of ICDE*, 2009.

[18] 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.

[19] Z. Gong and X. Gu. PAC: Pattern-driven Application Consolidation for Efficient Cloud Computing. In *Proc. of MASCOTS*, 2010.

[20] X. Gu and H. Wang. Online anomaly prediction for robust cluster systems. In *Proc. of ICDE*, 2009.

[21] G. Jiang, H. Chen, and K. Yoshihira. Discovering likely invariants of distributed transaction systems for autonomic system management. In *Proc. of ICAC*, 2006.

[22] M. Jiang, M. Munawar, T. Reidemeister, and P. A. S. Ward. Automatic fault detection and diagnosis in complex software systems by information-theoretic monitoring. In *Proc. of DSN*, 2009.

[23] M. P. Kasick, J. Tan, R. Gandhi, and P. Narasimhan. Black-box problem diagnosis in parallel file systems. In *Proc. of FAST*, 2010.

[24] T. Kohonen, M. R. Schroeder, and T. S. Huang, editors. *Self-Organizing Maps*. Springer, 3rd edition, 2001.

[25] A. Makanju, A. N. Zincir-Heywood, and E. E. Milios. Fast entropy based alert detection in super computer. In *Proc. of DSN*, 2010.

[26] I. T. Olliffe. *Principal Component Analysis*. Springer-Verlag, 2002.

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

[28] P. Reynolds, J. Wiener, J. Mogul, M. Aguilera, and A. Vahdat. Wap5: black-box performance debugging for wide-area systems. In *Proc. of WWW*, 2006.

[29] K. Shen, C. Stewart, C. Li, and X. Li. Reference-driven performance anomaly identification. In *Proc. of SIGMETRICS*, 2009.

[30] C. Stewart, T. Kelly, and A. Zhang. Exploiting nonstationarity for performance prediction. In *Proc. of Eurosys*, 2007.

[31] C. Stewart and K. Shen. Performance modeling and system management for multi-component online service. In *Proc. of NSDI*, 2005.

[32] P.-N. Tan, M. Steinbach, and V. Kumar. *Introduction to Data Mining*. Addison Wesley, 2005.

[33] Y. Tan, X. Gu, and H. Wang. Adaptive system anomaly prediction for large-scale hosting infrastructures. In *Proc. of PODC*, 2010.

[34] Y. Tan, H. Nguyen, Z. Shen, X. Gu, C. Venkatramani, and D. Rajan. PREPARE: Predictive Performance Anomaly Prevention for Virtualized Cloud Systems. In *Proc. of ICDCS*, 2012.

[35] C. Wang, V. Talwar, K. Schwan, and P. Ranganathan. Online detection of utility cloud anomalies using metric distributions. In *Proc. of NOMS*, 2010.

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