vPerfGuard: an Automated Model-Driven Framework for Application Performance Diagnosis in Consolidated Cloud Environments

Pengcheng Xiong and Calton Pu
Georgia Institute of Technology
{pxiong3,calton}@cc.gatech.edu

Xiaoyun Zhu and Rean Griffith
VMware Inc.
{xzhu,rean}@vmware.com

ABSTRACT

Many business customers hesitate to move all their applications to the cloud due to performance concerns. White-box diagnosis relies on human expert experience or performance troubleshooting "cookbooks" to find potential performance bottlenecks. Despite wide adoption, the *scalability* and *adaptivity* of such approaches remain severely constrained, especially in a highly-dynamic, consolidated cloud environment. Leveraging the rich telemetry collected from applications and systems in the cloud, and the power of statistical learning, *vPerfGuard* complements the existing approaches with a model-driven framework by: (1) *automatically* identifying system metrics that are most predictive of application performance, and (2) *adaptively* detecting changes in the performance and potential shifts in the predictive metrics that may accompany such a change. Although correlation does not imply causation, the predictive system metrics point to potential causes that can guide a cloud service provider to zero in on the root cause.

We have implemented vPerfGuard as a combination of three modules: a sensor module, a model building module, and a model updating module. We evaluate its effectiveness using different benchmarks and different workload types, specifically focusing on various resource (CPU, memory, disk I/O) contention scenarios that are caused by workload surges or "noisy neighbors". The results show that vPerfGuard automatically points to the correct performance bottleneck in each scenario, including the type of the contended resource and the host where the contention occurred.

1. INTRODUCTION

Diagnosing and resolving application performance problems in consolidated cloud environments is hard. For example, consider the following scenario. Alice, a cloud service provider, hosts Bob's *vSlashdot* news aggregation site on her virtualized infrastructure as shown in Fig. 1. Since Bob's site composes of Web, App and DB servers, Alice deploys each server in a *virtual machine* (VM) because such a distributed deployment enables consolidation and flexible resource sharing.

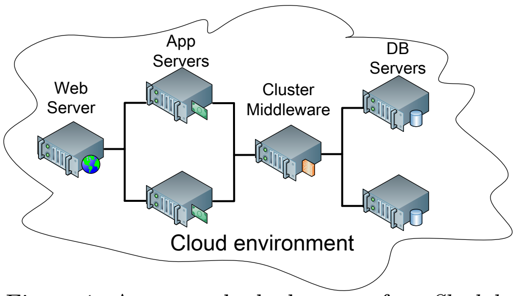


Figure 1: An example deployment for *vSlashdot*

After the *vSlashdot* service goes online, Bob notices that the performance of the web site degrades over time – its throughput drops and the response times increase. Bob immediately issues a support call with Alice. Traditionally, an IT service provider either relies on human experts with deep technical knowledge to identify performance bottlenecks, with the help of performance monitoring tools and system logs, or follows standard procedures in performance troubleshooting "cookbooks" [26] for problem localization and root cause analysis.

However, such *white-box* performance analysis methods [24] can no longer meet the demand for fast and accurate diagnosis in a highly-dynamic, large-scale, complex cloud environment, for the following reasons: (1) these methods have highly variable resolution times (from minutes to weeks); (2) they are not easily *scalable* to analyzing the behavior of many hosts and VMs in consolidated environments and many heterogeneous and distributed applications; (3) performance "cookbooks" only provide guidelines for problems that were seen before, whereas a dynamic cloud environment is likely to see emergent behavior or new interactions. For example, the performance of one application may suffer due to demand spikes in other applications (i.e., *noisy neighbors*) sharing the same physical infrastructure.

To overcome these limitations, we believe a data-driven approach [17] to performance diagnosis should be adopted to leverage the rich telemetry collected from applications and systems in the cloud. However, any performance analysis tool may be overwhelmed by the sheer amount of performance data and statistics collected at different levels of the stack. A recent report by TRAC Research [7], cited in [27], showed that 42% of the surveyed IT organizations reported challenges regarding usability of performance data. These include amount of time spent correlating performance data (63%), amount of performance data that is irrelevant (61%), issues they are not able to see (false negatives) (42%), and getting invalid alerts (false positives) (32%).

In order to address the above challenges, we propose *vPerfGuard*[1], a model-driven framework for achieving *automated*, *scalable*, and *adaptive* performance diagnosis in consolidated cloud environments. vPerfGuard accomplishes this by learning a performance model for an application using the system metrics that are most predictive of the application performance, and by adapting the model online by automatically detecting changes in the performance and the potential shifts in the predictive metrics that may accompany such a change. More specifically, the vPerfGuard architecture, as shown in Figure 2, consists of three modules - a sensor module, a model building module, and a model updating module.

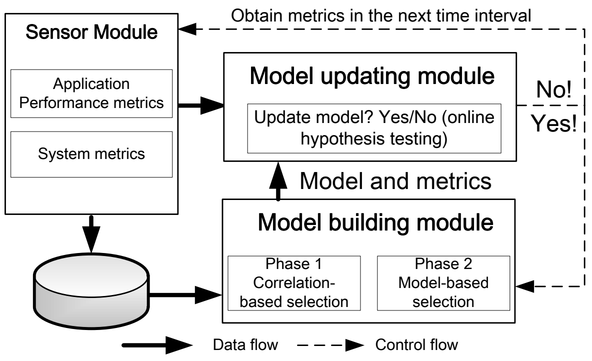

Obtain metrics in the next time interval
Sensor Module
Application Performance metrics
System metrics
Model updating module
Update model? Yes/No (online hypothesis testing)
No!
Yes!
Model and metrics
Model building module
Phase 1 Correlation-based selection
Phase 2 Model-based selection
Data flow
Control flow

Figure 2: vPerfGuard framework

Whenever a performance degradation is observed and a performance troubleshooting request is received, vPerfGuard presents the top predictive system metrics in the current performance model to the cloud service provider such as Alice. These metrics can provide hints to Alice regarding potential causes for the observed performance problem, including the critical component within a complex, distributed application as well as the suspected resource bottleneck in the associated host or VM. Alice can then use this information to determine the real root cause and take remediation steps such as VM re-sizing or migration. Moreover, the ability of these models to predict application performance using system metrics can enable the development of performance control systems that further automate the process of remediation. The last goal is the focus of our ongoing work and will not be discussed further in this paper.

This paper makes two main contributions, using both theoretical reasoning and experimental validation:

1. We apply appropriate statistical learning techniques to construct performance models that capture the relationship between application performance and system resources. We implement the solution in the model building module of vPerfGuard. The statistical learning techniques (1) filter thousands of system metrics and select those that are most strongly correlated with observed application performance, eliminating a large number of irrelevant metrics, and (2) further reduce redundancy in the selected metrics and build a performance model using a small set of metrics that give the best prediction accuracy. The automatic model generation process successfully overcomes the *scalability* challenge.

2. We apply appropriate statistical hypothesis tests to detect the need to update the performance model when it no longer accurately captures the relationship between performance and system resources. We implement the solution in the model updating module of vPerfGuard. The statistical hypothesis tests (1) detect the change-point due to variations in workloads (such as demand spikes) or system conditions (such as resource contention), and (2) trigger the model building module to update the set of predictive metrics and rebuild the model at runtime. The automatic model updating process effectively overcomes the *adaptivity* challenge.

[1]vPerfGuard stands for: virtual Performance Guard.

The remainder of the paper is organized as follows. We describe the design of vPerfGuard in Section 2. In Section 3, we introduce our testbed setup. We then present the results of our experimental studies in Sections 4 and 5, and describe the visualization of the results in Section 6. In Section 7, we review related work. Finally, we conclude and discuss future work in Section 8.

2. SYSTEM DESIGN

In this section, we introduce the design of the three modules - the sensor module, the model building module, and the model updating module in vPerfGuard.

2.1 Sensor module

The objective of this module is to continuously collect system metrics and application performance metrics. More specifically, it collects two categories of system metrics - *VM metrics* from the operating systems within individual VMs and *host metrics* from the physical hosts running the hypervisors. In our experimental evaluations, the sensor can collect thousands of system metrics, which we refer to as *raw metrics*. It also collects the application performance metrics of interest, e.g., throughput, response times, etc. Workload metrics such as offered load and transaction mix have been required by others for performance predictions [33]. Our framework does not preclude these metrics but we do not *require* their inclusion because: (1) we prefer vPerfGuard to be application-agnostic to free cloud service providers from needing detailed knowledge about the inner operations of their customers' applications, and (2) results from our experimental studies show that they are filtered out and do not appear in any of our final performance models.

2.2 Model building module

The objective of this module is to automatically utilize the thousands of *raw metrics* from the sensor module to derive a performance model that captures the relationship between application performance and system resources.

However, a performance model that is built using all the raw metrics can be computationally expensive to construct and can lead to model over-fitting. First, since the size of the search space with *thousands* of raw metrics is huge, machine learning algorithms operate slowly. Second, many raw metrics are irrelevant or redundant, e.g., a VM's CPU utilization observed from its host is closely related to the CPU utilization observed from within the VM itself. Such dependencies among metrics increase the amount of redundant information in the model and can degrade model quality.

This necessitates the selection of a small number of highly predictive metrics. After removing as many of the irrelevant and redundant metrics as possible, the model accuracy can be improved in some cases while the model can be more easily interpreted in other cases. We leverage two categories of algorithms for feature selection [23] to our metric selection: *filters* [25] evaluate features according to heuristics based on general characteristics of the data while *wrappers* [25] use the learning algorithm itself to evaluate the usefulness of features.

We achieve the objectives of metric selection and model building using a two-phase algorithm which is a combination of filters and wrappers: first (in phase 1) selecting a small number of *candidate metrics* that are most strongly correlated with the application performance from among the raw system metrics, and then (in phase 2) identifying even fewer *predictor metrics* that can give the best prediction accuracy for a specific model from among the candidate metrics.

A. Phase 1: Correlation-based selection.

In phase 1 (see Algorithm 1), we aggressively reduce the number of raw metrics considered by filtering out the raw metrics that are not highly correlated with the observed application performance. We denote the application performance metric (e.g., mean response time) as $perf$, and the time series of the $perf$ metric ending at time interval t as a vector $\overrightarrow{perf(t)} = [perf(t), perf(t-1), ...]$. We denote a raw system metric (e.g., CPU consumption of a VM) as m, and the set of all the raw metrics as M. We then denote the time series of each metric ending at time interval t as a vector $\overrightarrow{m(t)} = [m(t), m(t-1), ...]$. For each metric $m \in M$, we use r_m to denote the absolute value of the correlation coefficient between $perf$ and m, and p_m to denote the associated p-value for testing the hypothesis of no correlation. Each p-value is the probability of getting a correlation coefficient as large as the observed value by random chance, when the true correlation is zero. If the p-value is small, say less than 0.05, then the observed correlation is significant.

Algorithm 1: Phase 1

```
1 procedure Metrics selection by correlation coefficient
2 Input: performance metric $perf$ and a set of raw metrics $M$;
3 Output: a set of candidate metrics $M_{can} \subset M$;
4 Tunable Parameter: number of candidate metrics $N_{can}$;
5 $\forall m \in M$, $r_m = |\text{corrcoef}(\overrightarrow{perf}, \overrightarrow{m})|$,
  $p_m = \text{p-value}(\overrightarrow{perf}, \overrightarrow{m})$;
6 Select top $N_{can}$ metrics with highest $r_m$ and lowest $p_m$;
7 Sort $N_{can}$ metrics in descending $r_m$ and ascending $p_m$;
8 Return the set of $N_{can}$ metrics, denoted as $M_{can}$..
```

To limit the number of candidate metrics for our model, we select N_{can} top metrics as the candidate metrics for the phase 2 algorithm from all the N_{raw} raw metrics based on the absolute correlation coefficient value and the p-value. We also sort N_{can} metrics for visualization purpose. The time complexity is $O(N_{raw}) + O(N_{can} \log N_{can})$ when the BFPRT algorithm [5] is used for selection and the quicksort algorithm is used for sorting, respectively. N_{can} is a configurable parameter for managing the tradeoff between better model accuracy and lower overhead in the second phase.

B. Phase 2: Model-based selection.

In phase 2 (see Algorithm 2), we explore various combinations of the candidate metrics generated in phase 1, and choose a combination that gives the best prediction accuracy measured by the average R^2 (coefficient of determination) value [19] of the performance model using a 10-fold cross validation [29]. We evaluate and compare the predictive capability of the following four specific types of performance models [20] — linear regression model, k-nearest neighbor (k-NN), regression tree, and boosting approach.

Although the number of metric combinations has been reduced from $2^{N_{raw}}$ to $2^{N_{can}}$ after phase 1, the exploration process is still clearly prohibitive for all but a small number of metrics. We use a heuristic, hill climbing [30] search strategy, i.e., given a set of selected metrics, we choose the additional metric from the remaining set that can give the best improvement in the R^2 value. The algorithm ends when the improvement is smaller than a given threshold. For N_{can} candidate metrics, the computation complexity of the phase 2 model-based selection is $O(N_{pred} * N_{can})$, where N_{pred} is the final number of predictor metrics selected.

Algorithm 2: Phase 2

```
1  procedure Metrics selection by a specific model
2  Input: a performance metric $perf$ and $M_{can}$ from phase 1;
3  Output: a set of predictor metrics $M_{pred} \subset M_{can}$ and the
   associated model $F(M_{pred})$ with learned parameter values;
4  Tunable Parameter: type of model $F$ (e.g., "linear"), $R^2_{inc}$
   for the minimum incremental $R^2$ improvement;
5  $selected = \emptyset$, $left = M_{can}$, $R^2_{old} = 0$, $R^2_{best} = 0$;
6  while $true$ do
7      for $m \in left$ do
8          $metrics = selected \cup \{m\}$;
9          Use $\overrightarrow{perf}$ and all $\overrightarrow{m}$ in $metrics$, obtain $R^2_{new}$
           following a 10-fold cross validation;
10         if $R^2_{new} > R^2_{best}$ then
11             $R^2_{best} = R^2_{new}$;
12         end
13     end
14     if $R^2_{best} - R^2_{old} > R^2_{inc}$ then
15         move $m$ from $left$ to $selected$;
16         $R^2_{old} = R^2_{best}$;
17     else
18         break;
19     end
20 end
21 Build the final model $F$ using the metric set $selected$;
   Return $M_{pred} = selected$ and the model $F(M_{pred})$.
```

Hall [23] proposes a method to select a subset of metrics based on a heuristic "merit" of the subset. The motivation is that phase 1 (i.e., filters [25]) may pick many metrics which individually have high correlation with the output metric, but that when combined together in a model do not provide much additional useful information. We have implemented his method and compared it with our two-phase algorithm. We find that (1) his method has comparable overhead with ours; (2) our phase 2 algorithm can also overcome the limitation of phase 1; (3) the final metrics and models are very similar if the final number of predictor metrics (N_{pred}) is a small number.

2.3 Model updating module

The objective of this module is to automatically detect the change-point when the performance model derived from the model building module no longer accurately captures the relationship between application performance and system resources.

In a highly-dynamic, consolidated cloud environment, the relationship between application performance and system resources could be altered due to time-varying workload patterns, aggravated resource contention, different VM-to-host mappings, or other changes. We define such a relationship change as a *change-point*. This is different from detecting changes in performance alone. For example, if a 20% increase in the workload leads to degraded performance, our module should not flag this as a change-point if the relationship between performance and system resources still holds.

We assume that the distribution of the model's prediction errors (residuals) is stationary across adjacent time intervals when there is no change. Motivated by this, we use an online change-point detection technique [14] to determine whether a change occurs by performing hypothesis testing on the model's prediction errors across adjacent time intervals.

More specifically, given an existing performance model constructed in time window $W(t')$, its prediction errors in $W(t')$ and those in an adjacent time window $W(t)$ as shown in Fig. 3, we adopt an unpaired 2-sample t-test [19] to determine whether the prediction errors observed in $W(t')$ and $W(t)$ come from the same distribution, (could have the same statistical mean), i.e., the null hypothesis is that "there is no significant difference in the statistical mean between $W(t')$ and $W(t)$." If the result of the hypothesis test suggests that to be the case, then our performance model is likely as (in)accurate as when we accepted it for use in production and, absent other information, we have no reason to discontinue using the model. A significant difference in the statistical mean is a sufficient but not necessary condition for a significant difference in the distribution and our t-testing does not assume that the variances of the prediction errors in $W(t')$ and $W(t)$ are equal.

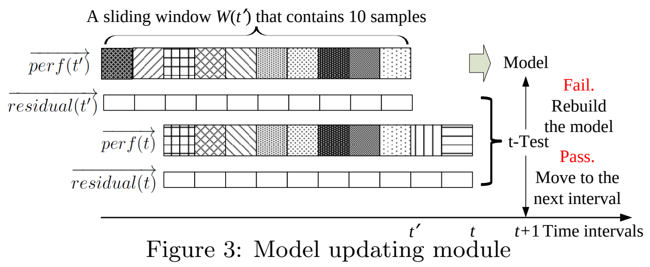

A sliding window W(t') that contains 10 samples
$\overrightarrow{perf(t')}$
$\overrightarrow{residual(t')}$
$\overrightarrow{perf(t)}$
$\overrightarrow{residual(t)}$
Model
Fail.
Rebuild the model
t-Test
Pass.
Move to the next interval
t'
t
t+1 Time intervals

Figure 3: Model updating module

3. EXPERIMENTAL SETUP

3.1 Hypervisor and sensor module

The vPerfGuard framework is generic and can work with different virtualized platforms and monitoring tools.

For the system metrics, we run VMware ESX 4.1 [9] as the hypervisor on each host. The sensor module of vPerfGuard can collect the host metrics (∼1800 metrics per host) from the "esxtop" [2] interface on ESX systems. Whereas our evaluation is done using VMware's hypervisor and tools, our framework generalizes to other virtualized platforms where similar tools exist to gather system-level metrics, e.g., "xentop" for Xen-based systems [13] and "Hyper-V Monitor Gadget" for Hyper-V-based systems [10]. For the VM metrics, we run "dstat" [1] and "iostat" [3] tools within the guest VMs so that the sensor module of vPerfGuard can collect the VM metrics (48 metrics per VM) from them.

For the application performance metrics, we collect the throughput and response times per sampling interval directly from the benchmark workload generator. In future work, we plan to leverage monitoring tools that can measure application metrics from the hosting platform. One example of such tools is the VMware vFabric Hyperic [8], which offers out-of-the-box performance monitoring for a suite of Web applications.

3.2 Benchmarks and workloads

Although we run various benchmarks [2] on our virtualized testbed, due to space limitations, we focus on the results from the RUBBoS [4] and the TPC-H [6] benchmarks.

In our experiments, we deploy the RUBBoS application with the browsing-only transaction mix in a 4-tier setup, including one Apache server, two Tomcat severs, one CJDBC cluster server and two MySQL servers as shown in Figure 4(a). The sampling interval is 1 minute. We deploy the TPC-H benchmark with a scaling factor 3 using PostgresSQL. The total database size is 4571MB including all the data files and index files. The original benchmark contains 22 queries, i.e., Q1 to Q22. We choose Q6, Q7, Q12 and Q14 for our experiments because these are IO-intensive queries and they can be completed within a sampling interval of 6 minutes. For both benchmarks, we modify the original workload generator to dynamically vary the number of concurrent users in the system.

3.3 Testbed setup and configurations

We run the RUBBoS benchmark on eight hosts, as shown in Fig. 4(a). Four hosts, *ESX1* through *ESX4*, are used to run the six VMs hosting the individual application tiers, labeled as *Web, App1, App2, CJDBC, DB1*, and *DB2*, respectively. We also run some *co-hosted* VMs on *ESX1* and *ESX4* to induce resource contention on the respective host. The four client VMs run on the other four hosts.

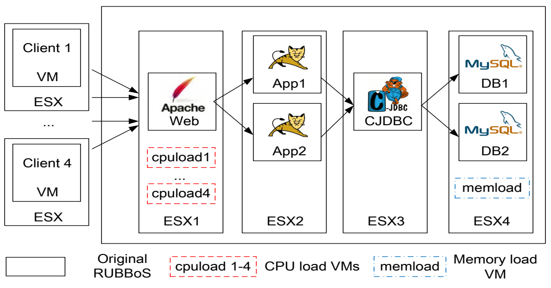
We run the TPC-H benchmark on three hosts shown in Fig. 4(b). Two virtual machines, $TPCH_F$ (foreground) and $TPCH_B$ (background), are deployed on one host, *ESX5*, running two instances of the TPC-H DB server. The two client VMs run on the other two ESX hosts.

The host and VM configurations are shown in Tables 1 and 2, respectively. All the VMs run Linux kernel 2.6.32. vPerfGuard runs on a separate host.

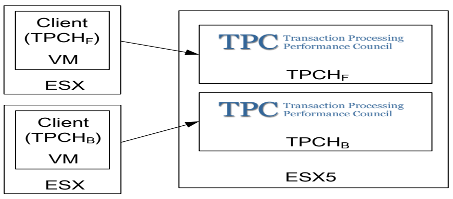
3.4 Naming convention

We describe the naming convention for all the metrics we collect in Table 3. For example, THR, MRT and RT_{95p} are application performance metrics, denoting throughput, mean response time and 95th percentile response time, respectively. The other metrics that begin with H and V are host and VM metrics, respectively. For instance, metric $H\_ESX1\_Web\_CPU\_System$ represents the CPU "System" counter for the "Web" VM running on the "ESX1" host, and metric V_CJDBC_Int represents the "Interrupt" counter for the "CJDBC" VM.

[2]RUBiS, RUBBoS with the browsing-only and the read-write transaction mixes, TPC-W and TPC-H.



(a) Test bed for RUBBoS benchmark


Client (TPCH_F) VM
ESX
Client (TPCH_B) VM
ESX
TPC Transaction Processing Performance Council
TPCH_F
TPC Transaction Processing Performance Council
TPCH_B
ESX5

(b) Test bed for TPC-H benchmark

Figure 4: Setup of two experimental testbeds

Table 1: Configuration of hosts

Testbed	RUBBoS	TPC-H, vPerfGuard
Model	Dell Power Edge 1950	Dell OptiPlex 780
CPU	2 Intel Xeon E5420 2.5 GHz Quad-Core	1 Intel Core2 Q9650 3.0 GHz Quad-Core
Memory	32 GB	16 GB
Storage	Clariion CX-40 SAN	7200 RPM local disk

Table 2: Configuration of VMs

Testbed	RUBBoS	TPC-H
Application VM vCPU	2	4
Application VM vRAM	1 GB	2 GB
Client VM vCPU	2	4
Client VM vRAM	8 GB	4 GB

Table 3: Metrics naming convention

Application perf. metrics	$THR, MRT, RT_{std}, RT_{50p}, RT_{75p}, RT_{90p}, RT_{95p}, RT_{99p}$
Host metrics	$H\_\{ESX\}\_\{VM\}\_\{Metric\}\_\{Details\}$
VM metrics	$V\_\{VM\}\_\{Metric\}\_\{Details\}$

4. EVALUATION OF MODEL BUILDING MODULE

In order to evaluate the model building module and compare the predictive capabilities of different performance models, we use the RUBBoS benchmark in the setup shown in Fig. 4(a), without the co-hosted VMs. We first run a calibration experiment where we vary the number of users from 400 to 4000 with a step size of 400, and observe that the application reaches a performance bottleneck at around 3000 concurrent users. This can be seen in Fig. 5(a) in the saturation of the application throughput and the near 100% CPU utilization of the "Web" VM. We then run a "random" workload for 400 minutes, where the number of users changes randomly between 400 to 4000. A sampling interval of 1 minute is used in the sensor module, resulting in 400 measurement samples that are used for the evaluation in this section. Each sample is a high-dimensional vector, consisting of the following 7522 metrics: 8 application performance metrics as shown in Table 3, 7226 host metrics from the four ESX hosts, and 288 VM metrics from the six VMs.

4.1 Evaluation of phase 1

For evaluation purposes, instead of limiting the number of candidate metrics from phase 1 as described in Algorithm 1, we report the number of candidate metrics selected by the phase 1 algorithm as a function of two threshold values — a lower bound, r_{LB}, on the absolute value of the correlation coefficient, and an upper bound, p_{UB}, on the p-value of the observed correlation.

We use throughput as the $perf$ metric and the 7226 ESX host metrics as the raw metrics. The number of selected host metrics is shown in Fig. 5(b). For example, for $r_{LB} = 0.8$ and $p_{UB} = 0.1$, a total of 132 metrics are selected out of the 7226 raw metrics. That means these 132 metrics (or 2% of the raw metrics) are correlated with the observed throughput with $r_m \geq 0.8$ and $p_m \leq 0.1$. We can also infer that 98% of the raw metrics are not highly correlated with the application throughput. The number of selected metrics is reduced as the minimum correlation level increases or as the maximum p-value decreases. The latter indicates an increased level of confidence in the observed correlation.

We observe similar trends when MRT or RT_{95p} is chosen as the $perf$ metric. We also observe similar trends when we use throughput or MRT as the $perf$ metric and the 288 VM metrics as the raw metrics.

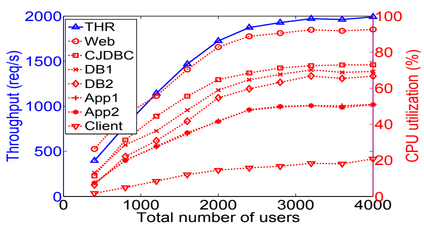
4.2 Evaluation of phase 2

To evaluate the phase 2 algorithm, we set the number of candidate metrics from phase 1 to be 100 and the minimum incremental R^2 improvement in phase 2 to be 0.01. For illustration, we provide an example of building a linear regression model for MRT in Figure 5(c), which shows the R^2 value for the model when one, two and three predictor metrics are selected sequentially. The phase 2 algorithm first chooses the network UDP active status of the Web VM on the $ESX1$ host ($V\_Web\_UDP\_Act$), resulting in an R^2 value of 0.668 for the single-metric linear MRT model. The algorithm then adds the second metric, the percentage of CPU Used of the Web VM on the $ESX1$ host ($H\_ESX1\_Web\_vCPU\_Used$), increasing the R^2 value of the model to 0.731. After adding the third metric, the total CPU Used on the $ESX1$ host ($H\_ESX1\_CPU\_TotalUtil$), the algorithm stops searching because the model quality improvement falls below the minimum incremental R^2 improvement threshold (0.01) when a 4th metric is added.

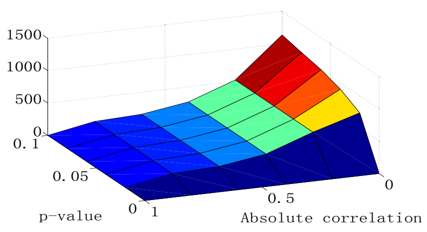
To evaluate the impact of the phase 2 metric selection algorithm, Fig. 5(d) reports the R^2 values for different model types in two scenarios: (1) using the 100 candidate metrics from phase 1 directly as the predictor metrics (without phase 2 selection), and (2) using the smaller number of predictor metrics selected from phase 2. For the first three model types (linear regression, k-NN, regression tree), the additional metric selection in phase 2 helps improve the accuracy of the final model.

4.3 Sensitivity analysis

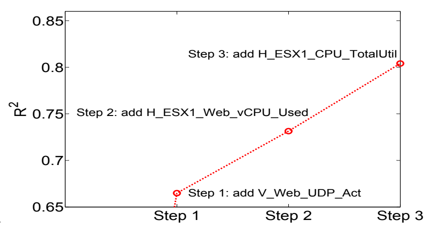
The effectiveness of the two-phase algorithm depends on the values of the tunable parameters, including (1) the number of selected candidate metrics from phase 1 (N_{can}), (2) the type of model F chosen in phase 2, and (3) the minimum incremental R^2 improvement in phase 2 (R^2_{inc}). We evaluate the impact of these parameters in two aspects — model accuracy and computation overhead. We assume that MRT is chosen as the $perf$ metric.



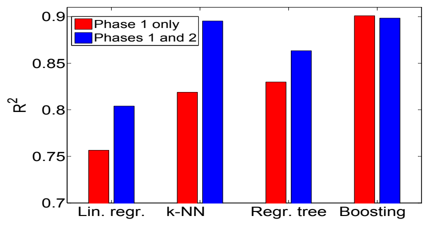
(a) Application throughput and CPU utilization of various VMs



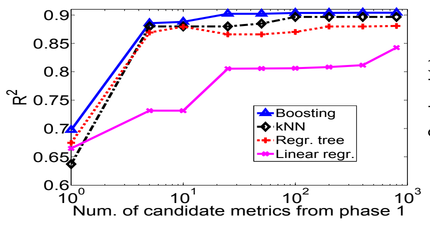
(b) Number of selected host metrics



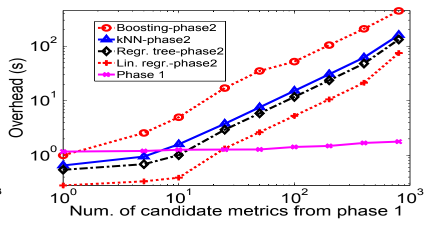
(c) Selected predictor metrics in a linear MRT model



(d) Improvement by phase 2 selection



(e) R^2 of different models



(f) Two phase algorithm overhead

Figure 5: Evaluation results for 2-phase metric selection and model building algorithm

Firstly, we fix the minimum incremental R^2 improvement in phase 2 at 0.01, and vary the other two tunable parameters. More specifically, for each of the four types of models, we limit the number of candidate metrics from phase 1 at [5, 10, 25, 50, 100, 200, 400, 800], and run the phase 2 algorithm to build a model for the MRT. In both experiments, a 10-fold cross validation [29] is used to compute the R^2 value for each model type.

Figure 5(e) shows the R^2 value of the final model as a function of the number of candidate metrics from phase 1 (in *log* scale). Different lines represent different model types. We make the following observations: (1) As more metrics are selected in phase 1, the model accuracy from phase 2 is generally improved for all four model types; (2) All the models achieve reasonably good accuracy ($R^2 > 0.8$) with 25 or more candidate metrics from phase 1, although the linear model's R^2 value is slightly lower than those from the nonlinear models.

Figure 5(f) shows the computation time of both phase 1 and phase 2 as a function of the number of candidate metrics from phase 1 (in *log* scale). Different lines for phase 2 represent different model types. We make the following observations: (1) The phase 1 overhead increases slowly with the number of candidate metrics; (2) For all four model types, the overhead in the phase 2 algorithm grows as we increase the number of candidate metrics from phase 1; (3) The boosting model has the most overhead, and the linear regression model has the least.

For demonstration purposes, we also run the phase 2 algorithm directly on all the raw metrics, without initial metric selection in phase 1. The result shows that, for all the model types, the metric selection in phase 1 helps achieve better accuracy in the final model as well as reducing the overhead in model building in phase 2.

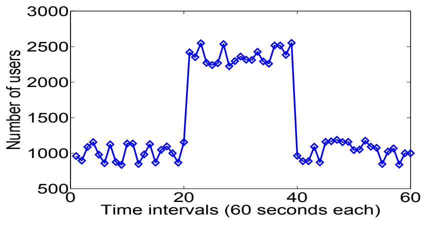
Secondly, we run similar experiments to evaluate the effect of the minimum incremental R^2 improvement in phase 2 (R^2_{inc}). We omit detailed results due to space constraints. As the threshold value becomes smaller, we have better model accuracy and more computational overhead in phase 2. We find that the threshold value of 0.01 for R^2 improvement strikes a good balance between better accuracy and lower overhead. Larger values require larger model-accuracy improvements to add metrics to the model, pre-empting the consideration of additional metrics.

Finally, we choose the linear regression model as our default model because it has the best *human-interpretability* and lowest overhead with only slightly lower accuracy relative to the nonlinear models. In spite of better accuracy, the regression tree is not a good candidate because (1) if we use shallow trees, the marginal ratio between performance and predictor metric is zero at most points, making it unable to infer which system metric is the bottleneck, and (2) if we use deep trees, the over-fitting issue prevents the model from generalizing faithfully. The *k*-NN and boosting approaches are also not preferred because they are harder to interpret directly due to model complexity.

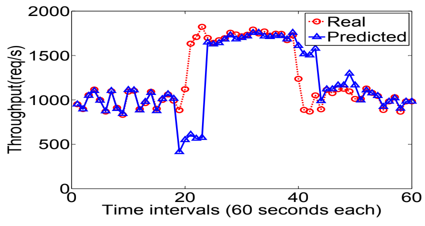
5. EVALUATION OF MODEL UPDATING MODULE

To evaluate the model updating module of vPerfGuard, we run the tool against four typical, dynamic workload scenarios a cloud service provider such as Alice may experience, including an application performance bottleneck caused by a surge in the workload intensity, and performance degradation in one application due to the CPU, memory, or disk I/O contention from co-hosted VMs (*aka. noisy neighbors*).

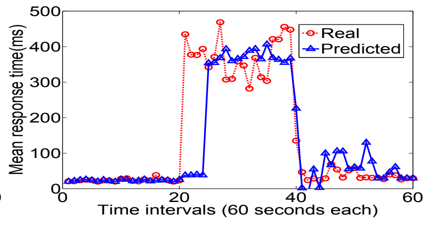
Our evaluation criteria focus on three aspects of the performance models generated: (1) **prediction:** whether a model provides an accurate prediction of the application performance using the selected system metrics; (2) **diagnosis:** whether the selected system metrics point to the correct performance bottlenecks, including the critical application component, the resource under contention, or the host where the contention occurred; (3) **adaptivity:** whether the model is adaptive to the changes in relationship between application performance and system resources. Note that we do not expect the human analyst to interpret the models directly. We will show a graphical user interface in the next section to il-



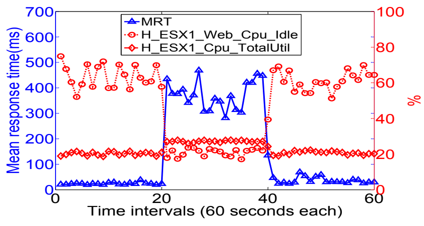
(a) Workload under test



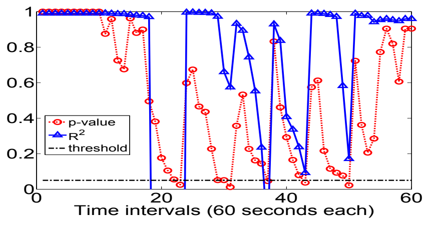
(b) Real and predicted throughput



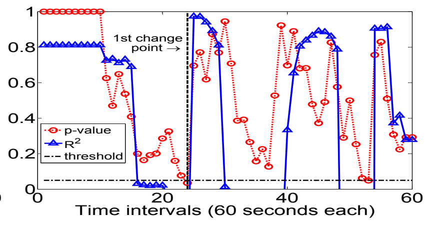
(c) Real and predicted mean response time



(d) MRT and top selected metrics

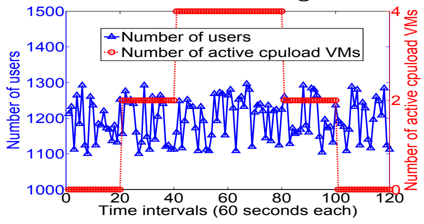


(e) Model accuracy for throughput

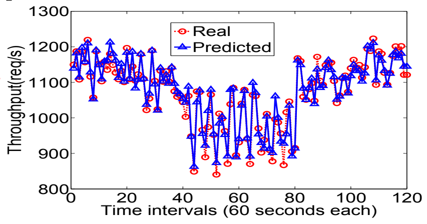


(f) Model accuracy for mean response time

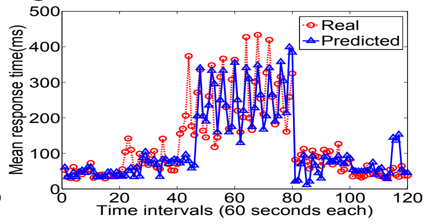
Figure 6: Experimental results for the workload surge scenario



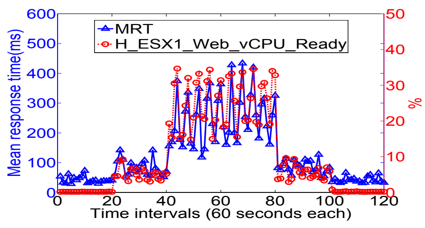
(a) Workload under test



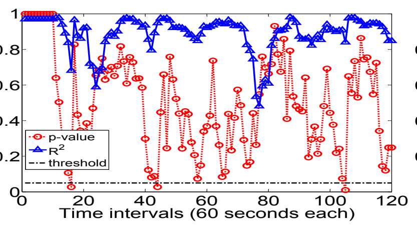
(b) Real and predicted throughput



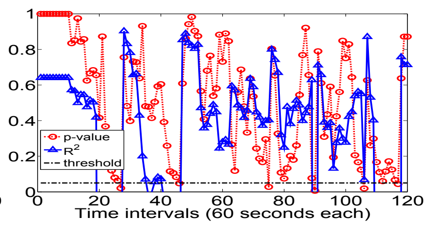
(c) Real and predicted mean response time



(d) MRT and top selected metric



(e) Model accuracy for throughput



(f) Model accuracy for mean response time

Figure 7: Experimental results for the CPU contention scenario

lustrate how the top suspicious metrics and the associated coefficients can be presented to the user for inspection.

The following subsections describe the experimental settings of the four scenarios, present the detailed results in Figures 6-9, and summarize the evaluation in Tables 4 and 5. In particular, each figure is organized as follows. Fig. (a) shows the workload(s) used, Fig. (b) and Fig. (c) compare the real and the model-predicted throughput and mean response time (MRT), respectively, Fig. (d) shows the MRT and the top selected metrics in the MRT model, and finally, the model accuracy measures including the p-value and the R^2 value for the throughput and the MRT models are shown in Fig. (e) and Fig. (f), respectively.

5.1 Workload surge

A. Experimental settings.

In this scenario, we use the testbed in Fig. 4(a) with the RUBBoS application only (i.e., no co-hosted VMs). The browsing-only workload mix is run for an hour (60 time intervals), with a surge in the workload intensity that goes from 1000 to 2300 users (with small, random variation) and lasts from the 21st to the 40th intervals (Fig. 6(a)).

B. Evaluation.

According to the experimental setting and our previous knowledge from Fig. 5(a), the mean response time increases during the workload surge period due to the CPU resource bottleneck in the Web tier of the RUBBoS application. As shown in Fig. 6(e) and Fig. 6(f), the online module detects a change-point multiple times, resulting in 6 throughput models and 3 MRT models for the duration of the experiment, such that we maintain a high confidence in the learned models (p-value ≥ 0.05). The throughput models starting from the 1st, 24th, 32nd, 38th, 44th and 51st intervals are:

$$THR = 0.19 \times V\_CJDBC\_Int - 519.60,$$

$$THR = 0.05 \times H\_ESX1\_Web\_vSwitch\_PcksTrans/s + 36.25$$

$$THR = 28.53 \times H\_ESX2\_App2\_vSwitch\_MBitsRec/s + 22.53 \times V\_DB2\_CPU\_Sys + 358.79$$

$$THR = 16.16 \times H\_ESX2\_CPU\_Idle\_Overlap + 1277$$

$$THR = 0.08 \times V\_Web\_ContextSwitch - 202.37$$

$$THR = 28.53 \times H\_ESX2\_App2\_vSwitch\_MBitsRec/s + 113.75$$

The MRT models for the mean response time starting from the 1st, 25th and 54th intervals are:

$$MRT = 2.17 \times H\_ESX1\_CPU\_TotalUtil - 20.91$$
$$MRT = -8.10 \times H\_\mathbf{ESX1\_Web\_CPU\_Idle} + 545.10$$
$$MRT = 0.49 \times V\_CJDBC\_UDP\_Act - 162.75$$

We make the following observations. (1) The models are predictive of the application performance most of the time according to Figs. 6(b), 6(c) and Table 4. The signs of the coefficients in each THR or MRT model make sense, e.g., when throughput increases, interrupts, CPU utilization, context switches and network packets transmitted or received also increase. (2) The selected system metrics in all the THR models do not point to the correct cause of the performance degradation. Three of the six models choose network attributes as the top metrics, and two other models choose system interrupts or context switches as the top metrics. The selected system metric $H\_ESX1\_Web\_CPU\_Idle$ in the second MRT model as shown in Fig. 6(d) directly points to not only the bottleneck host (ESX1), but also the bottleneck VM (Web) and the critical resource (CPU).

Following the above observations, we conclude that, (1) both THR and MRT models have good *performance-prediction* capability, and THR models have better prediction accuracy due to its linear relationship with many system metrics, and (2) THR models are not suitable for diagnosis, and MRT models have good *diagnosis* capability during the periods of performance bottlenecks. This observation is also validated in the next three scenarios and in [22]. When the application experiences a performance bottleneck, the THR remains almost constant, making it harder for our correlation-based selection to identify critical system metrics. At the same time, a small change in a critical system metric may lead to a large change in the MRT, making it easier for our algorithm to identify the correlation. For conciseness, we do not show the throughput models in the following scenarios, and we show only the MRT models during the periods of performance degradation to illustrate their diagnosis capability.

The MRT models are also *adaptive* to the surge in the workload, with only 3 intervals of delay in response. As Fig. 6(f) shows, after the workload surge at the 21th interval, the application MRT increases dramatically. It takes 3 intervals for the p-value of the first MRT model to drop below the threshold value of 0.05. The vertical line in the figure (24th interval) indicates where the first change-point is detected, after which the Web VM's CPU idle time is chosen as the key metric for learning a new MRT model starting from the next interval.

5.2 CPU contention

A. Experimental settings.

In this experiment, we run the RUBBoS benchmark with a workload intensity randomly varying between 1100 and 1300 users, as shown in Fig. 7(a). To create a CPU contention scenario, we use the four co-hosted VMs, *cpuload*1 to *cpuload*4, as noisy neighbors, to share the CPUs on the ESX1 host with the RUBBoS *Web* VM (see Fig. 4(a)). Each *cpuload* VM is configured with 2vCPUs, 1GB vRAM, and runs a "CPU eater" workload that consumes the vCPUs at a specified utilization level for a specified time period. We let the CPU utilization of each VM vary periodically between 10% and 40% with a period of 2 minutes. All the four co-hosted VMs are idle for the initial 20 time intervals. We then start the workload in *cpuload*1 and *cpuload*2 at the 21st interval, and start *cpuload*3 and *cpuload*4 at the 41st interval. The workloads in these VMs are idle again at the 81st and the 101st intervals (Fig. 7(a)).

B. Evaluation.

According to the experimental setting, the application's mean response time starts to increase after the 21st interval due to the CPU resource bottleneck on the ESX1 host, caused by the active workloads in the four *cpuload* VMs. The MRT models starting from the 28th, 47th, 76th and 91st intervals are:

$$MRT = 1.97 \times H\_ESX4\_DB1\_Mem\_Active + 1.13 \times H\_\mathbf{ESX1\_CPU\_Util} - 89.7$$
$$MRT = 752.76 \times H\_\mathbf{ESX1\_CPULoad}\_1MinuteAvg - 562.87$$
$$MRT = 12.48 \times H\_\mathbf{ESX1\_Web\_vCPU\_Ready} - 25.01$$
$$MRT = 6.38 \times H\_\mathbf{ESX1\_Web\_vCPU\_Ready} + 48.72$$

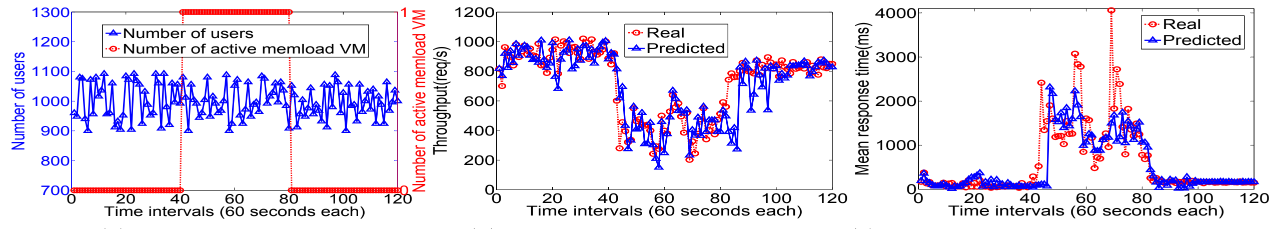
We make the following observations. (1) These MRT models have good prediction capability as shown in Figs. 7(b), 7(c) and Table 4. (2) These models have good diagnosis capability because they all point to the correct performance bottleneck. For example, one of the top system metrics, $H\_ESX1\_Web\_vCPU\_Ready$, as shown in Fig. 7(d), specifies not only the bottleneck host (ESX1), but also the bottleneck VM (Web) and the critical resource (CPU). (3) The models are adaptive to the increased CPU load from the noisy neighbors as shown in Fig. 7(f). After two of the cpuload VMs become active in the 21th interval, it takes the model updating module 6 intervals to detect the change and build a new model.

5.3 Memory contention

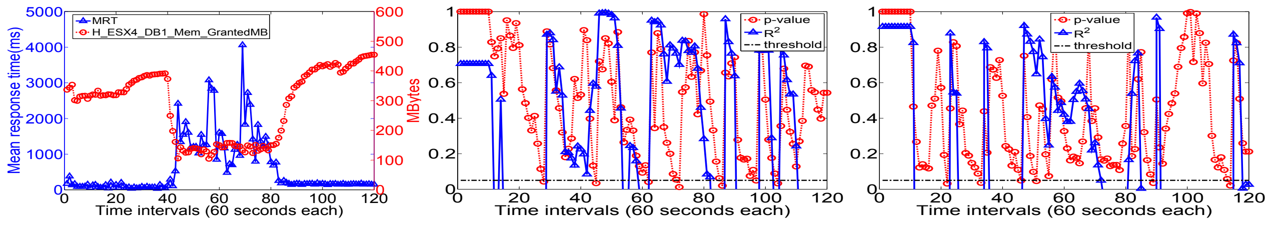
A. Experimental settings.

In this experiment, we run the RUBBoS benchmark with a workload intensity randomly varying between 900 and 1100 users, as shown in Fig. 8(a). Because the size of the MySQL database is 498.88MB, the total size of database files on the two database VMs, i.e., $DB1$ and $DB2$, are approximately 1GB. We use a co-hosted VM, *memload* (configured with 4vCPUs, 1GB vRAM), as the noisy neighbor, to run on the $ESX4$ host along with $DB1$ and $DB2$ (see Fig. 4(a)). To create memory contention, we configure the $ESX4$ host with 4GB of physical memory. Since about 3GB of memory is reserved by the hypervisor, only 1GB of memory is available for the three VMs ($DB1$, $DB2$, and *memload*) to share. As a result, the total memory commitment during these 40 intervals is much more than the shared 1GB memory.

For the initial 40 intervals, the *memload* VM remained idle, so the total memory commitment on $ESX4$ is close to 1GB. Between the 41st and the 80th intervals, a four-thread "memory eater" application is started inside the *memload* VM. Each thread in the application allocates 120-180MB memory and randomly touches the allocated pages to keep them actively used. As a result, the total memory commitment during these 40 intervals is much more than the shared 1GB memory. When the RUBBoS DB servers cannot access enough physical memory, more requests require

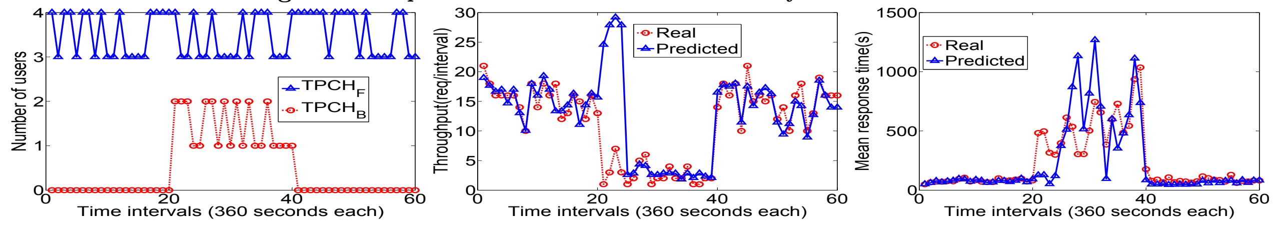


(a) Workload under test (b) Real and predicted throughput (c) Real and predicted mean response time

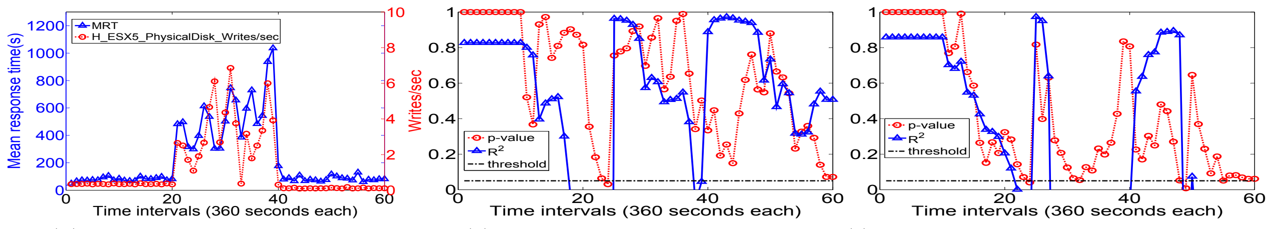


(d) MRT and top selected metric (e) Model accuracy for throughput (f) Model accuracy for mean response time

Figure 8: Experimental results for the memory contention scenario



(a) Workload under test (b) Real and predicted throughput (c) Real and predicted mean response time



(d) MRT and top selected metric (e) Model accuracy for throughput (f) Model accuracy for mean response time

Figure 9: Experimental results for the disk I/O contention scenario

disk accesses, reducing throughput (Fig. 8(b)) and increasing response times (Fig. 8(c)).

B. Evaluation.

The MRT models starting from the 47th and 52th intervals are:

$$MRT = -0.75 \times H\_ESX4\_Network\_PksReceived/sec + 3072.86$$

$$MRT = -25.38 \times H\_\mathbf{ESX4\_DB1\_Mem\_GrantedMB} + 4876.55$$

We make the following observations. (1) The models have reasonably good prediction capability as shown in Figs. 8(b), 8(c) and Table 4. (2) The second model for mean response time has good diagnosis capability, since the top system metric, $H\_ESX4\_DB1\_Mem\_GrantedMB$ as shown in Fig. 8(d), indicates not only the bottleneck host (ESX4), but also to the critical resource (Memory). (3) The models are adaptive to the increased memory load in the system with 10 intervals of delay as shown in Fig. 8(f).

5.4 Disk I/O contention

A. Experimental settings.

In this experiment, we run two instances of the TPC-H benchmark in parallel. A foreground database VM ($TPCH_F$) and a background database VM ($TPCH_B$) are co-located on the $ESX5$ host, as shown in Fig. 4(b). Since the total database size is 4571MB, which cannot fit into the VM's 2GB vRAM, some of the queries much involve disk I/O. Fig. 9(a) shows the workloads used in both VMs with a 6 minute sampling interval. The workload for $TPCH_F$ has the number of users randomly varying between 3 and 4 throughout the experiment. The background VM, $TPCH_B$, is idle for the initial 20 intervals. Between the 21st and the 40th intervals, the TPC-H workload is activated inside the $TPCH_B$ VM, a noisy neighbor, with the number of users randomly varying between 1 and 2.

B. Evaluation.

The MRT model between the 25th and 49th intervals for the $TPCH_F$ application is:

$$MRT = 180.62 \times H\_\mathbf{ESX5\_PhysicalDisk\_Writes/s} + 29.24$$

We make the following observations. (1) The models can capture the main trend in the application MRT as shown in Figs. 9(b), 9(c) and Table 4. (2) The top system metric $H\_ESX5\_PhysicalDisk\_Writes/s$ selected, as shown in Fig. 9(d), points not only to the bottleneck host (ESX5), but also the critical resource (disk I/O). Note that database servers often write temporary files (e.g., sorting files) to the disk when the physical memory is scarce. (3) The models are adaptive to the occurrance of the disk I/O bottleneck with only 3 intervals of delay as shown in Fig. 9(f).

5.5 Evaluation summary

For the four dynamic workload scenarios that we test, we summarize the evaluation results for the model prediction and the model diagnosis accuracies of vPerfGuard.

A. Prediction.

Besides showing the model prediction accuracy in R^2 over a sliding window of samples in the previous figures, we also report in Tab. 4 the statistics (mean, standard deviation, 50th percentile, 90th percentile) of the relative error for the individual THR and MRT samples, i.e., $(|(predicted - real)/real|)$. In the last column, we also show the *overall* relative error as (sum of $|predicted - real|$)/(sum of $real$). We can see that vPerfGuard achieves reasonably good prediction accuracy. We also notice that the relative error for THR is much smaller than that for MRT in all four scenarios except the disk I/O contention scenario. This validates our earlier observation in Sec. 5.1 that linear models capture the relationship between application throughput and system metrics well in most cases.

Table 4: Relative error for THR and MRT

Scenarios(perf)	Mean(std)	50p	90p	overall
Workload(THR)	0.10(0.20)	0.01	0.53	0.10
Workload(MRT)	0.41(0.64)	0.12	0.96	0.35
CPU(THR)	0.01(0.01)	0.01	0.03	0.01
CPU(MRT)	0.29(0.26)	0.23	0.64	0.27
Memory (THR)	0.12(0.16)	0.06	0.33	0.10
Memory (MRT)	0.51(0.45)	0.25	0.95	0.37
Disk I/O(THR)	0.95(3.32)	0.11	1.58	0.24
Disk I/O(MRT)	0.29(0.39)	0.16	0.70	0.39

B. Diagnosis and adaptivity.

Table 5 shows the diagnosis accuracy of the MRT models using *precision* and *recall* measures from pattern recognition literature, computed only for the performance bottleneck period. In our context, we define precision to be the fraction of all the selected metrics that are relevant (i.e., point to the correct bottleneck); and if a metric appears in n intervals, it's counted n times. We define recall to be the fraction of all the intervals in which the selected metrics are relevant; and for intervals with multiple selected metrics, that interval is counted using only the fraction of the relevant metrics. We report precision and recall for the detection of bottleneck resource and bottleneck host, separately. We use the CPU contention scenario as an example where the length of the bottleneck period is 80 intervals In intervals 21-27, the model contains an irrelevant metric; in intervals 28-46, the models contains two metrics with one being relevant; in the remaining intervals the model contains one relevant metric. Hence, precision$=(19+54)/(7+19\times2+54) = 74\%$, and recall$=(19/2 + 54)/80 = 79\%$. In the last column, we also report the delay in change-point detection in number of intervals. We can see that models built by vPerfGuard achieve good diagnosis accuracy in terms of precision and recall, with short delays in model updates.

Table 5: Diagnosis summary

Scenarios	Precision		Recall		Delay
	resource	host	resource	host	
Workload	100%	100%	100%	100%	3
CPU	74%	74%	79%	79%	6
Memory	73%	85%	73%	85%	10
Disk I/O	80%	100%	80%	100%	3

5.6 Discussion

A. Adaptation overhead.

In Table 6, we show the mean and standard deviation of the overhead in running vPerfGuard online for four dynamic scenarios, using 10 consecutive samples for both model building and change-point detection. The average overhead is 67ms (standard deviation = 37ms) for the sensor module to pull the application performance metrics and the system metrics from the hosts and the VMs. The metric selection and model building module takes an average of 221ms, the longest among all. The average model testing time is 54ms, and the average hypothesis testing time is 5ms.

Table 6: Online model adaptation overhead(mean(std))

Metrics collection(ms)	Building time(ms)	Testing time(ms)	Hypothesis testing(ms)
67(37)	221(148)	54(23)	5(18)

Table 7: Sensitivity analysis

Method (# of samples)	average R^2	# of models (correct)	# of metrics (correct)
p-value (10)	0.62	4 (4)	5(4)
p-value (20)	0.62	3 (2)	4 (2)
p-value (30)	0.76	3 (0)	3 (0)
R^2 (10)	0.74	49 (32)	56 (32)
R^2 (20)	0.73	40 (18)	45 (18)
R^2 (30)	0.80	38 (12)	38 (12)

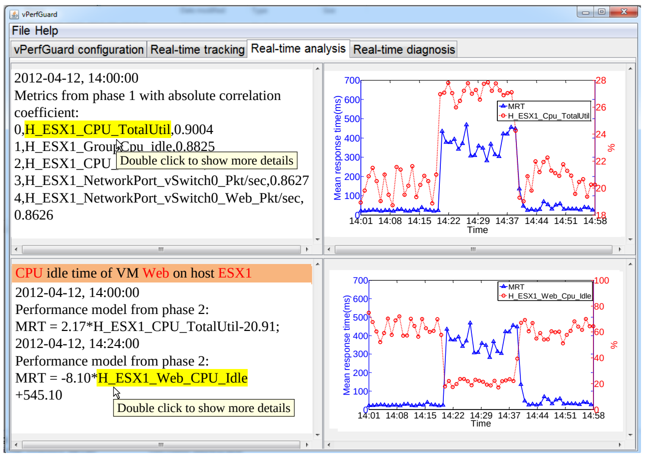
B. Sensitivity analysis.

We perform a sensitivity analysis using different change-point detection criteria (p-value < 0.05 or $R^2 < 0.8$) or a different number of samples to explore the tradeoff between prediction accuracy and diagnosis accuracy. In Table 7, we summarize the average positive R^2 value, the number of models generated, the number of models indicating the correct bottleneck, the total number of selected metrics in all the models, and the number of metrics identifying the correct bottleneck for the CPU contention scenario, during the contention period. As we increase the number of samples, the average R^2 increases (as one would expect), but the percentage of correct models or metrics decreases. For change-point detection, if a criterion of $R^2 < 0.8$ is used instead of using hypothesis testing with p-value < 0.05, we may generate too many models (due to over-fitting) for the human analyst to reason about. This result indicates that using hypothesis testing is a more robust method for change-point detection than using the R^2 value directly.

6. VISUALIZATION OF RESULTS

We introduce a primitive graphic user interface (GUI) for visualization of the results from vPerfGuard. The GUI consists of four panels: a configuration tab, a real-time tracking tab, a real-time analysis tab, and a real-time diagnosis tab. The configuration tab is used to configure the necessary parameters for vPerfGuard to operate. The real-time tracking tab composes of two windows: one shows the real and the predicted performance metric values for comparison; the other shows the corresponding p-value and R^2 value. Next we describe the other two tabs using the workload surge scenario as an example.

The real-time analysis tab (shown in Figure 10) presents the selected metrics and models in real time. The top-left window shows the selected metrics from phase 1 with absolute correlation coefficient in descending order. If a highlighted metric is double-clicked, the time series of the metric and the application performance metric will be shown in the top-right window. In the same tab, the bottom-left window shows the series of models built in phase 2 with the top system metrics, Because an abstract model may be hard to interpret by a cloud service provider, when a highlighted metric is double-clicked, the GUI translates the abstract metric name into human readable description at the top of the window. At the same time, the time series of the metric and the application performance metric are displayed in the bottom-right window.



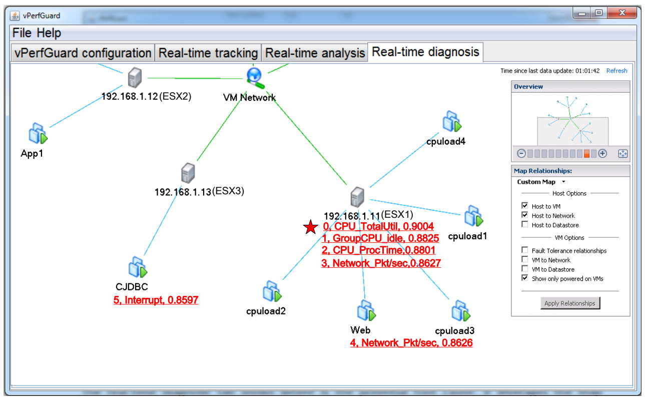
2012-04-12, 14:00:00
Metrics from phase 1 with absolute correlation coefficient:
0,H_ESX1_CPU_TotalUtil,0.9004
1,H_ESX1_GroupCpu_idle,0.8825
2,H_ESX1_CPU_
3,H_ESX1_NetworkPort_vSwitch0_Pkt/sec,0.8627
4,H_ESX1_NetworkPort_vSwitch0_Web_Pkt/sec, 0.8626

Double click to show more details

CPU idle time of VM Web on host ESX1
2012-04-12, 14:00:00
Performance model from phase 2:
MRT = 2.17*H_ESX1_CPU_TotalUtil-20.91;
2012-04-12, 14:24:00
Performance model from phase 2:
MRT = -8.10*H_ESX1_Web_CPU_Idle +545.10

Double click to show more details

Figure 10: GUI for real-time analysis



Time since last data update: 01:01:42

0, CPU_TotalUtil, 0.9004
1, GroupCPU_idle, 0.8825
2, CPU_ProcTime,0.8801
3, Network_Pkt/sec,0.8627
4, Network_Pkt/sec, 0.8626
5, Interrupt, 0.8597

Map Relationships:
Custom Map
Host Options
☑ Host to VM
☑ Host to Network
☐ Host to Datastore
VM Options
☐ Fault Tolerance relationships
☐ VM to Network
☐ VM to Datastore
☑ Show only powered on VMs

Figure 11: GUI for real-time diagnosis

The real-time diagnosis tab (shown in Figure 11) points to possible performance bottleneck locations. We utilize vCenter map [9], a visual representation of the vCenter Server topology that captures the relationships between the virtual and physical resources managed by the vCenter Server. The selected system metrics are overlayed on top of the associated components in the system. We use red fonts to locate the metrics from phase 1 and red star icon to locate the metrics from phase 2, respectively. As a result, this tab offers a cloud service provider better visibility into the potential bottlenecks in a complex and distributed environment.

One potentially useful feature we'd like to implement is allowing a user to manually add a metric to the model, by right clicking on a specific metric. This offers a cloud service provider an interface to provide inputs to the diagnosis process by applying domain knowledge. While model-driven performance diagnosis is useful in highlighting potential bottlenecks, incorporating domain knowledge from an experienced human analyst may lead to even better results in the timely determination of the real root causes of the performance problems.

7. RELATED WORK

In this section we survey prior work in performance diagnosis in two categories: knowledge-lean analysis of passive measurements and model-driven analysis.

Several projects build on low-overhead end-to-end tracing (e.g., [11, 12, 16, 31, 21]), which captures the flow (i.e., path and timing) of individual requests within and across the components of a distributed system. For example, Aguilera *et al.* [11] develop two different algorithms, i.e., RPC messages based and signal-processing based ones for inferring the dominant causal paths through a distributed system. Magpie [12] extracts the resource usage and control path of individual requests in a distributed system and tags incoming requests with a unique identifier and associating resource usage throughout the system with that identifier. Chen *et al.* [16] describe Pinpoint, a system for locating the components in a distributed system most likely to be the cause of a fault. vPerfGuard is similar to these systems in that it relates application performance to VMs and hypervisors that host application components. The key difference is that we consider metrics collected within VMs and hypervisors rather than communication patterns among components. Therefore, our approach is complementary to theirs.

Other recent research seeks to adopt a performance model that connects application performance with system metrics. A performance model may be constructed using a *white-box* approach based on human expert experience and domain knowledge [28, 18, 15, 32] or a *data-driven* approach [17] where a model is built automatically using statistical learning techniques. Compared with a *a white-box* performance model, a *data-driven* model assumes little or no domain knowledge; it is therefore generic and has potential to apply to a wide range of systems and to adapt to changes in the system and the workload. Shen *et al.* [32] present a performance debugging approach based on a whole-system performance model according to the design protocol/algorithms of the target system. Bodík *et al.* [15] present a methodology for automating the identification of performance crises using a fingerprint to represent the state of a datacenter. Tan *et al.* [34] present *PREPARE* which can predict recurrent performance anomalies by combining attribute value prediction with supervised anomaly classification methods. However, the approach in [32] requires deep understanding of I/O systems while the approach in [15] and [34] currently only work with recurrent anomalies. Compared with these prior approaches, vPerfGuard is more flexible to capture emergent behavior or new relationships inside a dynamic cloud environment that have not been seen before due to its data-driven nature. Our work complements theirs by helping identify performance bottlenecks using models. Probably the work closest to ours is [17] where Cohen *et al.* also use a data-driven approach to build a tree-augmented naive (TAN) Bayesian network model to learn the probabilistic relationship between the SLO state and system metrics. However, their models are built offline after an SLO violation has occurred to identify performance bottlenecks. Instead, we use an online approach for continuous adaption of our performance models.

8. CONCLUSION AND FUTURE WORK

In this paper, we propose vPerfGuard, an automated model-driven framework for application performance diagnosis in consolidated cloud environments. The key difference of vPerfGuard from existing approaches is the application of a model-driven approach. First, compared with existing *white-box* performance diagnosis [28, 32, 15, 24] that relies heavily on human expert experience and domain knowledge, vPerfGuard automatically construct models to bridge the application performance and system resources. Second, compared with the offline diagnosis approach in [17], vPerfGuard adaptively updates the models to accommodate changes in highly dynamic consolidated cloud environments. The model-driven approach not only helps a cloud service provider track the application performance using models, but also presents her with suspicious system metrics that can lead to the root cause of the performance problem.

Recall Bob's performance issue (Section 1). Running vPerfGuard with his application constructs a model mapping system metrics to application performance. Alice can track Bob's application performance with models and differentiate between performance problems that may be resolved by the cloud user and those that may have to be resolved by the cloud provider (e.g., contention due to noisy neighbors). The models also allow for richer automated diagnostics to be built on top of them and provide a solid foundation for constructing (and understanding) automatic performance control systems, two active areas of research for our future work.

9. REFERENCES

[1] dstat. `http://dag.wieers.com/home-made/dstat`.

[2] Interpreting esxtop Statistics. `http://communities.vmware.com/docs/DOC-9279`.

[3] iostat. `http://linux.die.net/man/1/iostat`.

[4] RUBBoS. `http://jmob.ow2.org/rubbos.html`.

[5] Selection algorithm: en.wikipedia.org/wiki/Selection_algorithm.

[6] TPC-H. `http://www.tpc.org/tpch`.

[7] TRAC Research. `www.trac-research.com/application-performance-management`.

[8] VMware vFabric Hyperic. `http://www.vmware.com/products/application-platform/vfabric-hyperic/`.

[9] VMware vSphere `www.vmware.com/products/vsphere/overview.html`.

[10] Windows Hyper-V Server. `www.microsoft.com/hyper-v-server/`.

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

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

[13] Barham, P., Dragovic, B., Fraser, K., H, S., Harris, T., Ho, A., Neugebauer, R., Pratt, I., and Warfield, A. Xen and the art of virtualization. In *Proc. of SOSP* (2003).

[14] Basseville, M., and Nikiforov, I. V. *Detection of Abrupt Changes*. Prentice Hall, 1993.

[15] Bodik, P., Goldszmidt, M., Fox, A., Woodard, D. B., and Andersen, H. Fingerprinting the datacenter: automated classification of performance crises. In *Proc. of EuroSys* (2010).

[16] Chen, M. Y., Accardi, A., Kiciman, E., Lloyd, J., Patterson, D., Fox, A., and Brewer, E. Path-based faliure and evolution management. In *Proc. of NSDI* (2004).

[17] Cohen, I., Goldszmidt, M., Kelly, T., Symons, J., and Chase, J. S. Correlating instrumentation data to system states: A building block for automated diagnosis and control. In *Proc. of OSDI* (2004).

[18] Doyle, R. P., Chase, J. S., Asad, O. M., Jin, W., and Vahdat, A. M. Model-based resource provisioning in a web service utility. In *Proc. of USITS* (2003).

[19] Draper, N. R., and Smith, H. *Applied regression analysis*. Wiley-Interscience, 1998.

[20] Duda, R. O., Hart, P. E., and Stork, D. G. *Pattern Classification*, 2nd ed. Wiley, New York, 2001.

[21] Erlingsson, U., Peinado, M., Peter, S., and Budiu, M. Fay: extensible distributed tracing from kernels to clusters. In *Proc. of SOSP* (2011).

[22] Georges, A., and Eeckhout, L. Performance metrics for consolidated servers. In *Proc. of HPCVirt* (2010).

[23] Hall, M. A. Feature selection for discrete and numeric class machine learning. Morgan Kaufmann, pp. 359–366.

[24] Jain, R. *The Art of Computer Systems Performance Analysis*. Wiley-Interscience, New York, 1991.

[25] John, G. H., Kohavi, R., and Pfleger, K. Irrelevant features and the subset selection problem. In *Proc. of ICML* (1994).

[26] Kumar, C., and Rosenberg, H. Troubleshooting Performance Related Problems in vSphere 4.1 Environments. `communities.vmware.com/docs/DOC-14905`.

[27] Livni, N. Blog: Six Application Performance Challenges facing IT. `www.correlsense.com/blog/six-application-performance-challenges-facing-it/`.

[28] Padala, P., Hou, K., Zhu, X., Uysal, M., Wang, Z., Singhal, S., Merchant, A., and Shin, K. Adaptive control of virtualized resources in utility computing environments. In *Proc. of EuroSys* (2007).

[29] Refaeilzadeh, P., Tang, L., and Liun, H. Cross validation. *Encyclopedia of Database Systems* (2009).

[30] Rich, E., and Knight, K. *Artificial intelligence*. McGraw-Hill, Inc., New York, NY, USA, 1991.

[31] Sambasivan, R. R., Zheng, A. X., De Rosa, M., Krevat, E., Whitman, S., Stroucken, M., Wang, W., Xu, L., and Ganger, G. R. Diagnosing performance changes by comparing request flows. In *Proc. of NSDI* (2011).

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

[33] Stewart, C., Kelly, T., and Zhang, A. Exploiting nonstationarity for performance prediction. In *Proc. of EuroSys* (2007).

[34] Tan, Y., Nguyen, H., Shen, Z., Gu, X., Venkatramani, C., and Rajan, D. Prepare: Predictive performance anomaly prevention for virtualized cloud systems. In *Proc. of ICDCS* (2012).