A Resource Contention Analysis Framework for Diagnosis of Application Performance Anomalies in Consolidated Cloud Environments

Tatsuma Matsuki
Software Laboratory
FUJITSU LABORATORIES LTD.
Kawasaki, Japan
matsuki.tatsuma@jp.fujitsu.com

Naoki Matsuoka
Software Laboratory
FUJITSU LABORATORIES LTD.
Kawasaki, Japan
matsuoka.naoki@jp.fujitsu.com

ABSTRACT

Cloud services have made large contributions to the agile developments and rapid revisions of various applications. However, the performance of these applications is still one of the largest concerns for developers. Although it has created many performance analysis frameworks, most of them have not been efficient for the rapid application revisions because they have required performance models, which may have had to be remodeled whenever application revisions occurred.

We propose an analysis framework for diagnosis of application performance anomalies. We designed our framework so that it did not require any performance models to be efficient in rapid application revisions. That investigates the Pearson correlation and association rules between system metrics and application performance. The association rules are widely used in data-mining areas to find relations between variables in databases.

We demonstrated through an experiment and testing on a real data set that our framework could select causal metrics even when the metrics were temporally correlated, which reduced the false negatives obtained from cause diagnosis. We evaluated our framework from the perspective of the expected remaining diagnostic costs of framework users. The results indicated that it was expected to reduce the diagnostic costs by 84.8% at most, compared with a method that only used the Pearson correlation.

Keywords

Cloud computing, Performance diagnosis, Correlation analysis, Association rule

1. INTRODUCTION

Cloud services have created many benefits to application and service developers. One of the main benefits of the cloud has been brought about by its agility. Cloud services can quickly provide application developers on-demand computational resources (e.g., virtual machines (VMs)), which allows them to agilely develop, easily revise, and scale out their applications, which accelerates their business growth. The advantage of the cloud has recently been changing the software industry, which is shifting from traditional plan-based developments to agile developments. That has also created a new term called *lean startup* [21]. Agilely developed applications in lean startups are repeatedly delivered to users for earlier feedbacks. Such agile features of applications in the cloud is leading to an era with more rapid revisions of businesses as well as applications.

While cloud services enable the agile developments of applications, the performance of applications in the cloud is still one of the prime concern for application developers. A physical server in the cloud typically hosts several VMs and these share resources (e.g., CPUs, memories, disks, and networks) on the same server. This resource sharing enables more efficient and flexible resource management, which, however, can lead to problems with the performance of applications [15]. When resource contentions between VMs occur, applications on the VMs can suffer from performance degradation due to a lack of resources. Application developers (cloud service users) and cloud service providers have to address these performance issues, which includes having to detect and diagnose them. However, these processes often involve excessive costs because of the complexity and large scale of applications and clouds.

Many researchers have proposed performance analysis tools or frameworks [25] in cloud environments and datacenters to reduce the costs. Most of them [2, 5, 14, 22, 24] have used the model-based approach, which models the performance of applications or problems using techniques such as machine learning with a training data set. However, they are inefficient for the rapid application revisions because agile revisions may cause frequent remodeling or outputting many adverse results.

This paper proposes a framework for the diagnosis of application performance anomalies caused by resource contentions in the consolidated cloud. Our framework was designed so that it did not require any performance models to work efficiently for the agile application revisions. Our framework diagnoses the performance anomalies of applications by correlating the application performance (e.g., response times) with performance metrics (e.g., CPU usage, network throughput, and disk I/O) obtained from the cloud

ICPE'16, March 12 - 18, 2016, Delft, Netherlands

ISBN 978-1-4503-4080-9/16/03…$15.00

DOI: http://dx.doi.org/10.1145/2851553.2851554

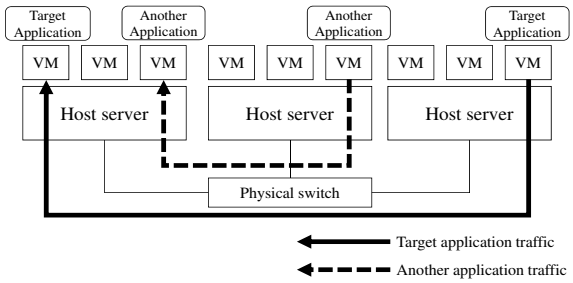


Figure 1: Target scenario for application performance anomalies in this paper: Performance degradation of *target application* occurs due to anomalous network traffic of *another application*

infrastructure. The correlated performance metrics could provide useful information for pinpointing the root causes of the application performance anomalies as some previous studies [9, 17, 29] indicated.

Our three main contributions in this paper are:

1. We propose a model-less framework for the diagnosis of application performance anomalies caused by the resource contentions (bottlenecks) in the cloud. Our diagnosis not only includes the diagnosis of the application performance bottlenecks but also the causes of these bottlenecks.

2. We also propose a method to mine association rules [1] between application performance and individual metrics to complement the drawbacks of the standard Pearson correlation. Association rules are widely used in data-mining areas to find relations between variables in databases. The approach using association rules reduces false negatives when there are temporal correlations between a metric and application performance.

3. We further investigated the effectiveness of the proposed technique of diagnosis using the data obtained from an experimental environment and a real data set. We also evaluated our framework from the perspective of not only the accuracy and coverage but also the expected remaining diagnostic costs to our framework users.

The rest of this paper is organized as follows. We explain our performance diagnosis framework in Section 2. Section 3 details how we evaluated it in an experimental environment. Section 4 clarifies how we tested our framework on a real data set and confirmed its effectiveness. Section 5 discusses some of its limitations. Section 6 introduces some related work. Finally, we conclude the paper in Section 7.

2. SYSTEM DESIGN

2.1 Diagnosis approach

The main objective of our framework is to select (1) the subset of metrics and their associated VMs that may have caused bottlenecks (application performance anomalies) and (2) the bottleneck attributes of performance anomalies (e.g., *CPU*, *network*, and *disk*). We expect that these two outputs will easily motivate cloud administrators to deal with performance anomalies by performing operations such as VM migration [4]. Metrics are selected by finding the correlation between individual metrics and application performance (e.g., response times). In the scenario in Figure 1, where a network resource contention occurs between a *target application* and *another application*, the causal VMs are those that host *another application* and the causal attribute is *network*. These outputs can motivate the administrator to migrate the VM to another host or set a limit for its usage of network bandwidth.

Let us now estimate the remaining diagnosis costs to our framework users and determine the main direction of our approach. It is almost impossible for diagnosis frameworks to achieve an optimal result (i.e., no false positives and no false negatives). They leave higher or lower costs for framework users. We assumed that framework users have the ability to correctly determine whether each metric is related to a performance anomaly or not, and once a related metric is found, they can address the performance issue. The cost of diagnosing a cause is then proportional to the number of metrics that is obtained from the cloud infrastructure. Denote the total number of metrics as $\aleph$. The cost, X, is represented as $X = \alpha\aleph$, where α is determined by depending on the skills of users. Assume that a cause diagnosis framework has *precision* p $(0 \leq p \leq 1)$ and *recall* r $(0 \leq r \leq 1)$, which are defined as

$$p = \frac{\#\text{ of true positives}}{\#\text{ of true positives} + \#\text{ of false positives}} \quad (1)$$

and

$$r = \frac{\#\text{ of true positives}}{\#\text{ of true positives} + \#\text{ of false negatives}}, \quad (2)$$

where p represents accuracy and r represents the comprehensiveness of diagnosis. We assumed that the diagnosis framework selected a subset of metrics whose size is $\aleph^*$. We then calculated the expected remaining cost of the diagnosis when a user utilizes the framework as:

$$X^* = \alpha\aleph^*(1-p) + \alpha(\aleph - \aleph^*)(1-r) \quad (3)$$

Note that we here assumed that p and r have probabilistic meanings. If the selected metrics include all correct metrics, the expected cost is $\alpha\aleph^*(1-p)$ because the selected metrics include false positives with the probability, $1-p$. If the selected metrics do not include the correct metrics, the cost expands by $\alpha(\aleph - \aleph^*)$ because the user has to additionally investigate the other $\aleph - \aleph^*$ metrics. The total expected remaining cost is calculated with Eq. (3) because the probability that the selected metrics will include the correct metrics is r. Even though p and r, in fact, mean the ratio in Eq. (1) and (2), the cost function (3) is intuitively accurate. When $p = 1$ and $r = 1$, the cost is $X^* = 0$, which means that an optimal diagnosis framework does not leave any diagnostic costs for users. When $r = 1$ (the results include all correct metrics), the cost is $\alpha\aleph^*(1-p)$, which means that the cost is proportional to $\aleph^*$ and the user have to investigate the cause within $\aleph^*$ metrics that have been selected by the framework. When $r = 0, p = 0$ (the results do not include any correct causes), the cost is $X^* = \alpha\aleph = X$, which means that the user has to investigate within all metrics.

We assumed $\aleph = d\aleph^*(d \geq 1)$ and transform Eq. (3) as:

$$\begin{aligned} X^* &= \alpha\aleph^*(1-p) + \alpha(\aleph - \aleph^*)(1-r) \\ &= \alpha\aleph - \alpha(\aleph^* p + \aleph r - \aleph^* r) \\ &= X - \alpha\aleph(\frac{d-1}{d}r + \frac{1}{d}p) \end{aligned} \quad (4)$$

The second term in Eq. (4) represents *gain* of the diagnosis framework. We obtained an important observation from the gain function, i.e., *recall* is more important than *precision*. As d increases, the contribution of p decreases with the rate, $1/d$. When $d \to \infty$, the gain is approaching $\alpha\aleph r$ and p makes no contributions. The intuitive meaning is that when d is large, $\aleph^*$ decreases and the diagnostic costs within $\aleph^*$ metrics also decrease even when p is small.

We designed our framework from this observation to achieve better recall (fewer false negatives), while achieving acceptable precision and large d (small $\aleph^*$).

2.2 System overview

Figure 2 overviews our framework, which is configured with the Metric Collector, Application Performance Input Module, Bottleneck Diagnosis, Cause Diagnosis and Database. Our framework uses three kinds of information for the diagnosis, i.e., performance metrics in the cloud, application performance data and classification information on the metrics, which is used to limit the analyzing metrics. We will now explain the metric classification in Section 2.3.

The Metric Collector collects the metrics data from the cloud infrastructure with a fixed sampling interval (e.g., 1 min) and timestamps the data that are thus obtained. It also attaches labels to the collected metrics data. The attached labels include information from which the metrics have been collected. For example, the *PercentProcessorTime* metric collected from VM1 in host A has a label, *PercentProcessorTime;VM1;hostA*. These labels are used to associate a correlated metric with a VM or a host. The Application Performance Input Module receives the application performance data from application administrators (cloud service users). They can transfer time-stamped numerical data (i.e., time series data) to this module. This input module architecture is motivated by a cloud environmental feature. The application and cloud infrastructure are usually managed by different administrators, which may make it difficult to continuously collect application performance data especially in a cloud provided as an infrastructure-as-a-service (IaaS). Our framework has also been designed to work even when application data have temporally been obtained.

The Bottleneck Diagnosis and Cause Diagnosis in Figure 2 are modules that infer the bottleneck attributes of the application performance anomalies for the former and causal VMs for the latter. We will explain the technical details of these two modules in Sections 2.4 and 2.5. The Bottleneck Diagnosis outputs the analysis results to both the Application and cloud administrator. The results include the bottleneck attributes and associated metrics. The Cause Diagnosis, on the other hand, only outputs the analysis results (causal VMs and the associated metrics) to the cloud administrator because the application administrator should be agnostic about causal VMs.

2.3 Preliminary

This section explains two preliminaries: the classification of metrics and the creation of pair-wise data. We classified

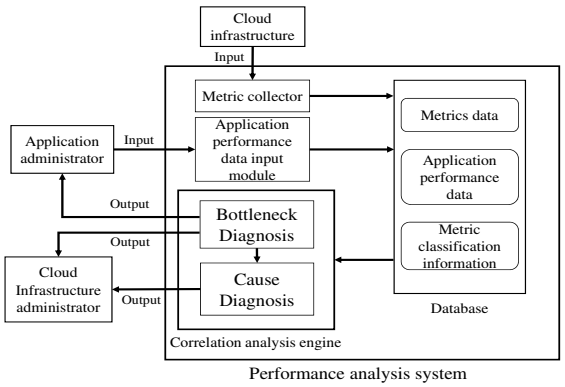


Figure 2: System overview

all the collected metrics into both of the location groups and attribute groups in advance, as summarized in Table 1, which lists the groups and the metrics examples belonging to each group. The Application VM group contains the metrics associated with VMs that the application administrator uses. The VM groups contains the metrics that do not belong to the Application VM group but that are associated with VMs. The Host group contains the metrics that is not associated with any VMs. This classification is done once in advance when our framework is installed because the kinds of collected metrics are rarely changed during the operational phase. We used this classification information for bottleneck and cause diagnosis, which will be described in Subsections 2.4 and 2.5.

Our framework evaluates the correlations between application performance and individual metrics. However, the timestamps of application performance data are different from those of metrics data. We define time units with a specific time interval $\bar{t}$ (e.g., 300 s) to adjust the timestamps. An analysis time period is determined based on input application performance data. We assume that t_s denotes the oldest timestamps of the application performance data and t_f denotes the newest. Here, we define time as absolute time such as UNIX time. Assume that the analysis start time is denoted as T_s and the end time is denoted as T_f. We then define them as $T_s = t_s - (t_s \bmod \bar{t})$ and $T_f = t_f - (t_f \bmod \bar{t}) + \bar{t}$. When T_s and T_f are represented as $T_f = T_s + N\bar{t}$, N is the number of time units in the analysis period. We denote time unit u_n as $u_n = [T_s + n\bar{t}, T_s + (n+1)\bar{t})$. A set of application performance data that have a timestamp within time unit u_n is aggregated (e.g., averaged), and we obtain a new time series as $\boldsymbol{A} = (a_0, a_1, ..., a_{N-1})$, where a_n represents the average application performance data within the time unit, u_n. We similarly obtain the new time series of metric i as $\boldsymbol{M}_i = (m_0^i, m_1^i, ..., m_{N-1}^i)$, where m_n^i represents the average metric i data within the time unit, u_n. The time series $\boldsymbol{A}$ and $\boldsymbol{M}_i$ are pair-wised, and we evaluate the correlation between them.

2.4 Bottleneck diagnosis

This section explains the technical details of how we select the bottleneck attribute, which is done by finding the metrics that are highly correlated with the application performance data. The port of *physical switch* in the scenario in Figure 1 is the bottleneck point and it is associated with a metric such as the *physical switch port packet counter*. When

Table 1: Metric classification

Location groups	Metrics	Attribute groups	Metrics
Application VM	Virtual CPUs, disks, NICs, and memories	CPU	CPUs and virtual CPUs
VM	Virtual CPUs, disks, NICs, and memories	Disk	Disks, storage, and virtual disks
Host	CPUs, disks, memories, NICs, virtual switches, physical switches and storage	Memory	Memory and virtual memories
		Network	Physical switches, NICs, virtual switches, and virtual NICs

performance degradation occurs on the target application, the *packet counter* should correlate with application performance. In order to find the correlated metrics (we call them *bottleneck metrics*), we simply used the Pearson correlation coefficient (we simply call it *correlation*). Although the correlation evaluates *linear* relationships between application performance and individual metrics, the previous work [29] has indicated that it achieves good accuracy in the bottleneck estimations. Here, we denote the set of all analysis time units as $\boldsymbol{T}$, and calculate the correlation between $\boldsymbol{A}$ and $\boldsymbol{M}_i$ as:

$$c_i = \frac{\sum_{n\in\boldsymbol{T}}(a_n-\bar{a})(m_n^i-\bar{m_i})}{\sqrt{\sum_{n\in\boldsymbol{T}}(a_n-\bar{a})^2}\sqrt{\sum_{n\in\boldsymbol{T}}(m_n^i-\bar{m_i})^2}}$$

where $\bar{a}$ and $\bar{m_i}$ are calculated as $\bar{a} = 1/N \cdot \sum_{n\in\boldsymbol{T}} a_n$ and $\bar{m_i} = 1/N \cdot \sum_{n\in\boldsymbol{T}} m_n^i$.

We then select a set of correlated metrics as bottleneck metrics and diagnose the bottleneck attribute. We explain the procedure for diagnosing the bottleneck attributes in Algorithm 1. All metrics belonging to the *application VM* or *host* group (summarized in Table 1) are analysis target data, which are input to Algorithm 1. All input metrics in the Algorithm 1 are sorted in decreasing order of $|c_i|$. The metrics are checked as to whether the correlation is over the threshold, T_{bot}, in order of $|c_i|$, and if so, the metrics are stored in *metr*. We use hypothesis testing of Pearson correlation with a specific significance level (e.g., 0.05) to determine threshold T_{bot}. Given the number of sample data (i.e., the number of analysis time units, N), T_{bot} is set as the critical value for testing [12]. We set a limit, l_m, for the number of selected bottleneck metrics to limit the number of selected attributes. The attributes of the correlated metrics are fetched by *attribute*(i) using the *attribute group* information in Table 1 and stored in *attr*. We finally obtain the bottleneck attributes, *attr*, and its associated metrics, *metr*.

Algorithm 1 Bottleneck attribute diagnosis

```
Input: metrics in application VM group or host group.
Output: bottleneck metrics metr ⇐ φ and selected attributes attr ⇐ φ
Threshold of correlation: T_bot
Maximum number of selected bottleneck metrics: l_m
Number of selected bottleneck metrics: count = 0
Set of input metrics: G

Calculate c_i for all metrics in G.
Sort i ∈ G in decreasing order of |c_i|.
for all i ∈ G do
    if |c_i| > T_bot then
        metr ⇐ i
        if attribute(i) ∉ attr then
            attr ⇐ attribute(i)
        end if
        count++
        if count > l_m then
            Break
        end if
    end if
end for
return metr, attr
```

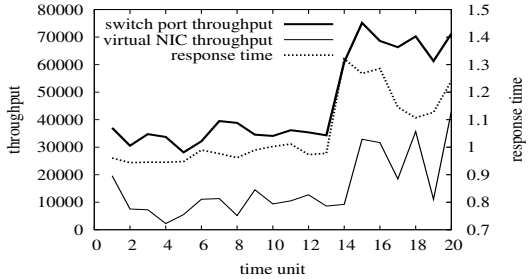


Figure 3: Examples of bottleneck and cause metrics

2.5 Cause diagnosis

This section explains the technical details of cause diagnosis. We also use the correlation-based approach in causal VM diagnosis as well as bottleneck diagnosis. However, there are some difficulties in selecting metrics that are associated with causal VMs (we call them *cause metrics*). Figure 3 plots examples of bottleneck and cause metrics. We obtained the behaviors of these metrics in the experimental environment that is explained in Subsection 3.1, which is somewhat similar to the scenario in Figure 1. Figure 3 indicates the behavior of the bottleneck metric (*switch port throughput* obtained from a bottleneck point) highly correlates with the application performance (*response time*). The correlation coefficient is 0.904. The behavior of a cause metric (*virtual NIC throughput* of the causal VM), on the other hand, is less correlated with it because the cause metric has opposite behavior at some time units (e.g., time unit 14 and 19). The correlation coefficient is 0.667. This metric is in fact one of the cause metrics because we experimentally generated a performance bottleneck using the VM. Therefore, we have to capture the cause metric by applying an additional method. Otherwise, no low-correlated cause metrics can be captured, which would lead to false negatives.

The main reason for low correlation of cause metrics is that there are multiple causes. For example, there are multiple VMs that bring about the performance bottlenecks. This means that the bottleneck metrics behave following the aggregated behavior (e.g., sum) of cause metrics. That is, each cause metric is temporally correlated with application performance. We evaluate the *association rules* [1], which has often been used in the field of data mining and previous work [9], between each metric and application performance to capture *temporal* correlations. To do this, we first have to discretize application performance and individual metrics.

We simply use a threshold in discretizing application performance and we assume that the threshold is specified by the application administrator. Given an application performance threshold, a, we discretize the application performance data as: $S_n = 0$ if $a_n \leq a$ and $S_n = 1$ otherwise. We then obtain $\boldsymbol{S} = \{S_n, n \in \boldsymbol{T}\}$. We here assume that time unit u_n is anomalous when $S_n = 1$ without losing generality.

We propose an algorithm that investigates the optimal threshold for discretizing metrics. We define the optimal threshold that maximizes *support* within a constraint of its *confidence* (denoted as T_{conf} and fixed with $T_{\text{conf}} = 0.8$). Given a threshold, m^i, for metric i, we discretize metric i as: $d_n^i = 0$ if $m_n^i \leq m^i$ and $d_n^i = 1$ otherwise. We obtain the discretized data for metric i as $\boldsymbol{D}_i = \{d_n^i, n \in \boldsymbol{T}\}$. We next calculate the support and its confidence for the direction from metric i to application performance (denoted as $\text{supp}(\boldsymbol{D}_i \rightarrow \boldsymbol{S}), \text{conf}(\boldsymbol{D}_i \rightarrow \boldsymbol{S})$) as follows. If $|\boldsymbol{D}_i^1 \cap \boldsymbol{S}^1|/|\boldsymbol{D}_i^1| \geq |\boldsymbol{D}_i^0 \cap \boldsymbol{S}^1|/|\boldsymbol{D}_i^0|$,

$$\text{supp}(\boldsymbol{D}_i \rightarrow \boldsymbol{S}) = \frac{\left|\boldsymbol{D}_i^1 \cap \boldsymbol{S}^1\right|}{\left|\boldsymbol{S}^1\right|}, \tag{5}$$

$$\text{conf}(\boldsymbol{D}_i \rightarrow \boldsymbol{S}) = \frac{\left|\boldsymbol{D}_i^1 \cap \boldsymbol{S}^1\right|}{\left|\boldsymbol{D}_i^1\right|}, \tag{6}$$

where $\boldsymbol{D}_i^1 = \{n | d_n^i = 1, n \in \boldsymbol{T}\}, \boldsymbol{S}^1 = \{n | S_n = 1, n \in \boldsymbol{T}\}$, and if not,

$$\text{supp}(\boldsymbol{D}_i \rightarrow \boldsymbol{S}) = \frac{\left|\boldsymbol{D}_i^0 \cap \boldsymbol{S}^1\right|}{\left|\boldsymbol{S}^1\right|}, \tag{7}$$

$$\text{conf}(\boldsymbol{D}_i \rightarrow \boldsymbol{S}) = \frac{\left|\boldsymbol{D}_i^0 \cap \boldsymbol{S}^1\right|}{\left|\boldsymbol{D}_i^0\right|}. \tag{8}$$

Equations (7) and (8) are the support and confidence for negatively correlated metrics. We set m^i as m_n^i for each $n \in \boldsymbol{T}$ and examine the optimal threshold, m^{i*}, as shown in Algorithm 2. We obtain the optimal threshold as the value of *opt*, which achieves the largest support within the constraint, $\text{conf}(\boldsymbol{D}_i \rightarrow \boldsymbol{S}) \geq T_{\text{conf}}$. We denote the discretized metric, i, with the optimal threshold as: $d_n^{i*} = 0$ if $m_n^i \leq m^{i*}$ and $d_n^{i*} = 1$ otherwise. We then obtain $\boldsymbol{D}_i^* = \{d_n^{i*}, n \in \boldsymbol{T}\}$. When the *opt* has *null* value, we assume that there is no valid association rule between $\boldsymbol{M}_i$ and $\boldsymbol{A}$.

The support measure defined in Equations (5) and (7) indicates the degree of association between application performance and metric i. A large support means a strong association and a small support means a temporal association. For example, when $\text{supp}(\boldsymbol{D}_i \rightarrow \boldsymbol{S}) = 0.5$, the half of anomalous time units ($S_n = 1$) is associated with (supported by) the metric i. Equations (6) and (8) indicate that the confidence measure is used to evaluate whether application performance is anomalous or not when the value of metric i increases (or decreases), and therefore, the confidence measure is close to one even when the correlation is temporal. The example of the temporally correlated metric in Figure 3 (*virtual NIC throughput*) is one of the metrics that has a valid association rule. When we set $a = 1.1$, its optimal discretizing threshold is set to 14,517 by Algorithm 2 and its confidence and support are calculated as 0.833 (5/6) for the former and 0.714 (5/7) for the latter.

The constraint of $T_{\text{conf}} = 0.8$ indicates that 80% of the time units with $d_n^{i*} = 1$ is associated with the application performance anomaly ($S_n = 1$). We have to set T_{conf} relatively large (e.g., ≥ 0.8) because if it is set too small, no valid association exists in the association rules. Therefore, we set $T_{\text{conf}} = 0.8$, which is also the default value of the *arules* package in R language [10].

We also have to confirm *lift* measures for association rules to make the association rules valid. The lift measure for association rule $\boldsymbol{D}_i^* \rightarrow S$ is defined as:

$$\text{lift}(\boldsymbol{D}_i^* \rightarrow \boldsymbol{S}) = \frac{\text{conf}(\boldsymbol{D}_i^* \rightarrow \boldsymbol{S})}{|\boldsymbol{S}^1|/N} \tag{9}$$

The lift measures generally have to be larger than one to validate association rules. The fraction of analysis time units that are anomalous ($|\boldsymbol{S}^1|/N$) therefore have to be less than 0.8 to make the lift larger than one, because we set $T_{\text{conf}} = 0.8$. This means our framework requires that input application performance data have to include more than 20% of normal ($S_n = 0$) time units.

Algorithm 2 Investigate optimal discretizing threshold

```
Input: Discretized application performance S and M_i
Output: Optimal threshold opt = null
Current max support: max = 0

for all m_n^i ∈ M_i do
    Calculate D_i with threshold m_n^i.
    Calculate supp(D_i → S) and conf(D_i → S).
    if conf(D_i → S) ≥ 0.8 then
        if supp(D_i → S) > max then
            max = supp(D_i → S), opt = m_n^i
        end if
    end if
end for
return opt
```

Let us next select a set of metrics that are associated with the causal VMs. In addition to the correlation, we use the association rules previously examined to capture the temporally correlated cause metrics. We explain the procedure for cause diagnosis in Algorithm 3. The analysis target metrics in our cause diagnosis are the metrics in the *VM* group listed in Table 1. We first remove the metrics that have no correlation with $\boldsymbol{A}$ by setting a threshold, T_{nocor}. We determine T_{nocor} as the critical value of hypothesis testing with a high significance level of $l = 0.1$, which is generally the highest. All selected cause metrics are satisfied under two conditions: i) *attr* obtained from Algorithm 1 contains its attribute and ii) its correlation is larger than T_{cor} or its support is $T_{\text{supp}} (0 \leq T_{\text{supp}} \leq 1)$ or above. We also determine T_{cor} by using hypothesis testing with a specific significance level of $l = 0.01$, which is set larger than T_{nocor}. We investigate the effect of the settings of T_{supp} in Section 3. The function, $extractVM$, in Algorithm 3 extracts the name of the associated VM from the index of metrics. Algorithm 3 finally outputs the cause metrics as *cause* and their associated names of VM as $causeVM$.

3. EXPERIMENTAL EVALUATION

3.1 Experimental setup

We built two types of applications on our experimental environment, i.e., a Web and virtual desktop infrastructure

Algorithm 3 Cause diagnosis

Input: Metrics in VM group, bottleneck attribute $attr$ from Algorithm 1, and application performance data $\boldsymbol{S}$
Output: Cause metrics $cause \Leftarrow \phi$ and $causeVM \Leftarrow \phi$
Set of input metrics: $\boldsymbol{G}$

for all $i \in \boldsymbol{G}$ **do**
  Calculate c_i.
  if $|c_i| > T_{\text{nocor}}$ and $attribute(i) \in attr$ **then**
    Calculate $\boldsymbol{D}_i^*$ and $\text{supp}(\boldsymbol{D}_i^* \rightarrow \boldsymbol{S})$.
    if ($|c_i| > T_{\text{cor}}$ or $\text{supp}(\boldsymbol{D}_i^* \rightarrow \boldsymbol{S}) \geq T_{\text{supp}}$) **then**
      $cause \Leftarrow i$
      if $extactVM(i) \notin causeVM$ **then**
        $causeVM \Leftarrow extractVM(i)$
      end if
    end if
  end if
end for
return $cause, causeVM$

Table 2: Configurations of VMs

# of VMs	Usage	vCPU	MEM	vDisk
10	Virtual desktop	2	2 GB	20 GB
1	VDI benchmark	2	2 GB	20 GB
1	Web, AP server	2	2 GB	20 GB
1	Load balancer	4	8 GB	40 GB
3	DB server	4	8 GB	40 GB
3	Load generator on Host-2	8	2 GB	10 GB
1	Load generator on Host-1	2	4 GB	10 GB

(VDI), which are typical applications that require stable performance [11]. Figure 4 illustrates our experimental environment. There are two physical host servers (*Host-1* and *Host-2*) and a physical switch that connects these two servers with a local network that has 1-Gbps bandwidth. Hosts-1 and -2 have the same configuration, i.e., Intel Xeon X5680, 3.33-GHz 6 core x 2 processors, 48 GB of RAM, 278-GB SAS x 4 (RAID1 x 2, one for the host and the other for the VMs), and Windows server 2012 R2 datacenter. We installed Hyper-V [18] on the two hosts to manage the VMs. Host-1 hosts 14 VMs: 10 VMs for the VDI application, a VM for benchmarking VDI performance, a VM for the Web and application (AP) server, a VM for the load balancer of the Web application, and a VM for generating load. Host-2 hosts six VMs: three VMs for database (DB) server of the Web application and the others for generating load. Table 2 summarizes the configurations of these VMs. These 10 VDI VMs and one VDI benchmark VM were running with Windows 7 (64 bits) and the other VMs were running with CentOS-7 (64 bits).

The Web application built on our experimental environment was configured with Nginx, Kibana and ElasticSearch, which have recently been widely used for visualizing logging data [8]. We used Nginx as the Web server, Kibana as the AP server and ElasticSearch as the load balancer and DB server. We assumed that *response times* between the Web server and DB servers would provide the application performance data for the Web application, which were obtained from the access logs that were output from Nginx. We built five VMs to generate the Web access workload on another environment, each of which accessed the same Kibana dashboard with a fixed polling interval (10 s).

The 10 VDI VMs simulated a VDI service, which provided

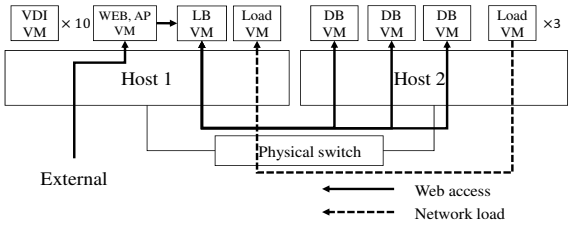


Figure 4: Experimental environment

desktop environments on Windows 7 operating system (OS). We simulated a VDI user workload that included a local Excel file open and over-write, an Excel file on an external file server open and over-write and external Web page access on each VDI VM. We wrote a script that randomly ran the above actions at random time intervals. The random intervals followed the an exponential distribution with a specific average. We set the averages as listed in Table 3. We prepared the application performance data for the VDI by measuring the experience of a VDI user. We then measured the response time of an Excel file (1 MB) open with a fixed time interval (30 s) on the VDI benchmark VM. These measures enable the disk performance of the VDI to be captured. (The disk performance is also a problem on our testing on a real data set, which will be explained in Section 4.)

Table 3: VDI workload simulation

# of VMs	Avg. interval	Actions
1	150 s	Local/remote file access, Web access
6	300 s	Local/remote file access, Web access
3	600 s	Local/remote file access, Web access

Table 4: VDI anomalous workload simulation

# of VMs	Avg. interval	Actions
1	150 s	Local/remote file access, Web access
5	300 s	Local/remote file access, Web access
4	10 s	Local file access

We used data on *Windows performance monitor* [20], which included the metrics of the host OS processor, disk, memory, network adapter, and those of VMs [28]. We obtained the data through the Windows management instrumentation (called *WMI*) [26] by running the *wmic* [27] command on the *Metric Collector* in Figure 2 with a fixed interval (60 s). We also used the simple network management protocol (SNMP) to obtain the management information base (MIB) counters of the physical switch port interfaces, which included I/O packets, bytes, and dropped packets. We obtained the MIB counters with a fixed interval (60 s). Some metrics have an cumulative feature such as a packet *counter* of a network interface and a read/write operation *counter* on a disk. Those metrics have to be calculated difference between successive data and the difference is divided by the sampling interval. These calculations are done on the *Metric Collector*. We obtained a total of 12,410 metrics from our experimental environment, which included 6,110 metrics that were associated with VMs.

3.2 Bottleneck injection

We evaluated the proposed framework with the data that contained injected performance anomalies. This section explains how performance anomalies were injected. We injected three types of resource contentions to create application performance anomalies: CPU, network, and disk.

CPU resource contentions were injected by the three load VMs on the host-2. This results in CPU resource shortages on DB servers, and it led to degraded performance in the Web application. We intermittently ran the *stress* [23] command on each load VM with a specific duration and the number of stressed CPU cores. The duration was randomly determined following an exponential distribution with a fixed average (10 s). The interval between two successive runs was also randomly determined following the exponential distribution with a fixed average (2 s). When the application performance was not anomalous, the number of stressed cores is set to four. When it was anomalous, we increased the number of stressed core to eight.

Network resource contentions were injected by the three load VMs on the host-2 and one load VM on host-1. We intermittently ran *iperf* [13] on all load VMs on Host-1 to generate network traffic load. The network traffic was transmitted from individual load VMs (i.e., iperf clients) on host-1 to a load VM (i.e., the iperf server) on host-2. The interval between two successive runs was also randomly determined following the exponential distribution with a fixed average (5 s). We ran *iperf* with the user datagram protocol (UDP) specifying the duration and bandwidth. The duration was randomly determined following the exponential distribution with a fixed average (10 s). When application performance was not anomalous, the bandwidth was randomly set following a uniform distribution $U(100\text{Mbps}, 300\text{Mbps})$. When it was anomalous, we switched the distribution of the determined bandwidth to $U(500\text{Mbps}, 700\text{Mbps})$ on two of the load VMs on Host-2. The increased bandwidth created response delays between the load balancer on Host-1 and DB servers on Host-2, which led to degraded performance in the Web application.

Disk resource contentions were injected by four of the ten VDI VMs. When VDI performance was anomalous, we switched the actions of four VDI VMs to only a local Excel file open/write with the small average intervals listed in Table 4. Frequent local file access caused disk I/O contentions between VDI VMs and the benchmark VM. That led to degraded VDI performance.

3.3 Experimental results

This section presents the results we obtained from the experimental evaluation that was previously explained. We injected 10 resource contentions for each CPU, network, and disk resource, which degraded application performance. We set $\bar{t} = 300$ (s) and input 10 times 18 time units (1.5 h) of application performance data, each of which included the performance degradation period.

3.3.1 Bottleneck diagnosis

Here, we present the results from bottleneck diagnosis that were obtained with Algorithm 1. The algorithm has a changeable parameter, l_m, which indicates the maximum number of bottleneck metrics that is selected by Algorithm 1. We investigated the effects of the setting of l_m as well as the evaluation of bottleneck diagnosis. We used *precision* and *recall* measures for the evaluation. Precision was defined as the fraction of selected attributes that was correct. Recall was defined that if the selected attributes contained the correct one, recall was one; if not, recall was zero.

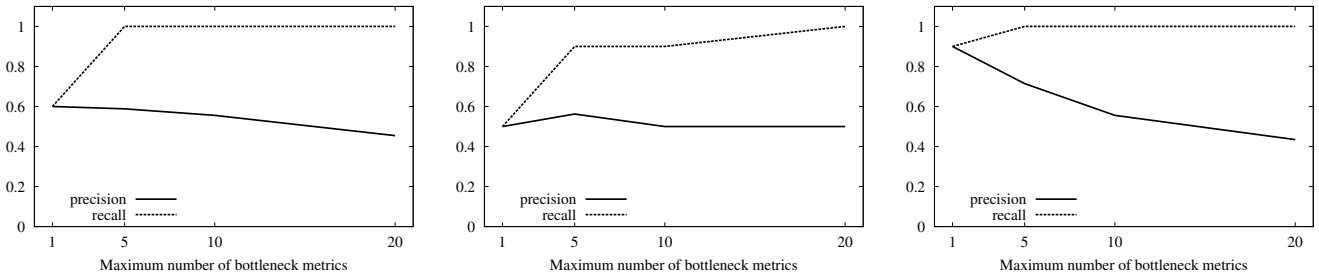
Figure 5 plots the results obtained from evaluating the selected attributes. Note that the results are the total (i.e., average) results for 10 anomaly injections for each type of resource contention. When $l_m = 1$, there are some false negatives, as shown in Figure 5; the results do not include the correct attribute. Setting $l_m = 1$ means that the attribute is determined by the most correlated metric. The reason for these false negatives is that the bottleneck injections not only affect the bottleneck attribute but also others. For example, running *iperf* not only affects network metrics but also CPU and memory metrics because generating network traffic uses some CPU and memory resources. Figure 6 plots the behavior of a false positive bottleneck metric that is most correlated with application performance whose attribute is memory. As we can see from the figure, the false positive metric is highly correlated with the response time (the correlation is 0.895). If we want to remove the false positive, we have to use another method that is not correlation-based. The similar phenomena can occur in a real environment because almost all real applications use several kinds of resources. Therefore, l_m should be set relatively large. Although, the false negatives decreased and false positives increased as l_m is increased, it is important for diagnosis frameworks to reduce the number of false negatives, as was explained in Subsection 2.1.

3.3.2 Cause diagnosis

We will next discuss our evaluation of the results obtained from cause diagnosis. Note that here we assumed that the bottleneck diagnosis was accurately done (i.e., *attr* in Algorithm 3 only included an accurate attribute), so that we could evaluate our cause diagnosis without the effects of the results from the bottleneck diagnosis. We used precision and recall measures for the evaluation. Here, we define precision as the fraction of all selected cause metrics (i.e., those in *cause* in Algorithm 3) whose attributes and associated VMs are accurate. However, it is difficult to identify the exact number of metrics that have to be correlated with the application performance when resource contention is injected. We therefore used VM-level recall instead of metrics -level. We defined recall as the fraction of all causal VMs that were accurately selected (i.e., those in $causeVM$ in Algorithm 3). We compared our framework with an approach using only the Pearson correlation, which selected cause metrics by hypothesis testing of the Pearson correlation with variations in significance levels of $l = 0.01, 0.05,$ and 0.1.

Figures 7 through 9 present the results in CPU, network, and disk contention cases. Each contains three graphs: (a) indicates precision, recall, and the number of selected metrics (denoted as $\aleph^*$), (b) indicates the effect of the T_{supp} setting ($T_{\text{supp}} = 0.2, 0.4, 0.6,$ and 0.8), and (c) indicates the expected remaining diagnosis cost of a framework user, which was calculated from Eq. (3) and normalized by $\alpha\aleph$. The results labeled $l = 0.01, 0.05,$ and 0.1 indicate those obtained by using the Pearson correlation with different significance levels l, and those labeled *proposal* are the results obtained from our framework with $T_{\text{supp}} = 0.2$. Note that here we have presented the total results obtained from 10 anomaly injections.

Figures 7(a), 8(a), and 9(a) indicate precision for different significance levels l has a tendency that as l increases, precision decreases and $\aleph^*$ increases. The reason for this is that the thresholds for the correlation are set large as l decreases. (When $N = 18$ and $l = 0.01, 0.05,$ and 0.1, the respective thresholds of the correlations are set to 0.589,



(a) CPU contention cases (b) Network contention cases (c) Disk contention cases

Figure 5: Total results from bottleneck diagnosis

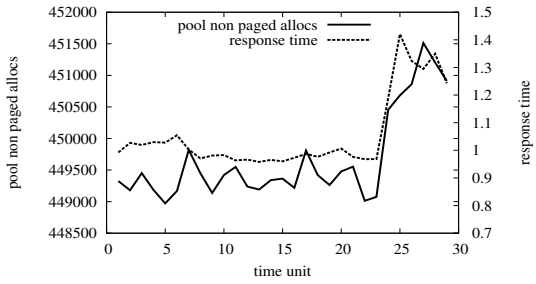


Figure 6: Example of false positive bottleneck metric

0.468, 0.400.) Higher thresholds also decrease recall because some cause metrics that have temporal correlating behaviors are not captured. As was previously explained, drops in recall may incur relatively large diagnosis costs for framework users. The framework we propose, on the other hand, achieves stable recall as well as the Pearson correlation with $l = 0.1$, which indicates that it accurately captures the metrics that have temporal and low correlation. The proposed framework achieves better precisions and smaller $\aleph^*$ for all cases of contention than that with the results using $l = 0.1$, which was used for the T_{nocor} setting. This means that even though setting $l = 0.1$ also enabled to capture temporal correlations by using low correlation thresholds, it created more false positives.

The results for the proposed framework were affected by the T_{supp} setting. As T_{supp} was set large, precision improved and $\aleph^*$ decreased as can be seen from Figures 7(b), 8(b), and 9(b). However, when $T_{\text{supp}} = 0.8$, recall decreased in Fig. 8(b) and 9(b). These results indicate that some cause metrics have association rules with support of < 0.8, which means that these metrics have more temporal correlations with application performance. Therefore, T_{supp} should not be set large (e.g., ≤ 0.6) so that the temporal correlation can be captured.

Although there were tradeoffs between precision and recall in these results, we could evaluate the total performance of our framework by calculating the remaining diagnosis costs from Eq. (3). Our framework achieved better performance than those of $l = 0.01$ and 0.05 in Figs. 7(c), 8(c), and 9(c). It respectively reduced the costs, especially with $l = 0.01$, by 77.4%, 84.8%, and 79.5% for CPU, network, and disk contention case. That was caused by stable recall as was previously explained. Our framework achieves better performance in Fig. 7(c) than that in $l = 0.1$ cases. However, the cost reductions in Figs. 8(c) and 9(c) are quite small despite the better precision of our framework. This is because $\aleph^*$ in the network and disk contention in Figs. 8(a) and 9(a) is smaller than that in the CPU contention in Fig. 7(a). That reduces the contribution of precision as was explained in Subsection 2.1.

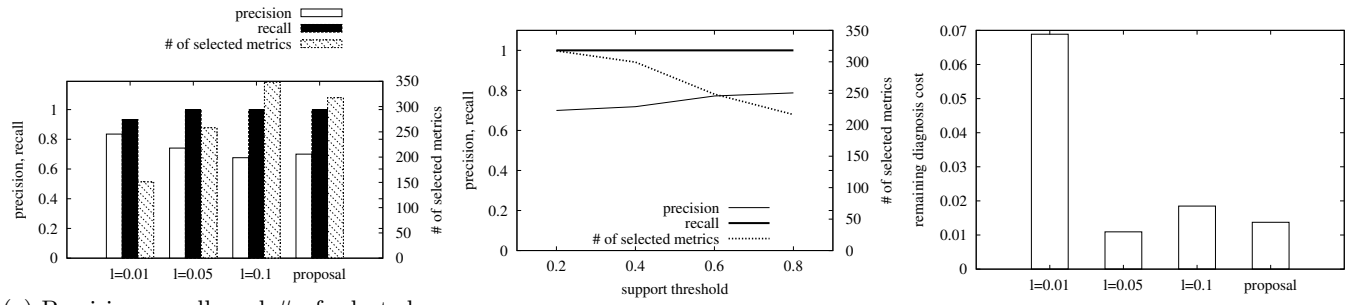
The threshold of the Pearson correlation is also affected by the length of the analysis time period (denoted as N) because the critical value of hypothesis testing is calculated from the number of sample data. As the number of sample data increases, the critical value decreases and the correlation threshold is then set small.

Figure 10 presents the results for the remaining diagnosis costs where the analysis time period length, N, is 24 and 30. We have omitted the results for precision, recall, and $\aleph^*$ because of space limitations. The results in Fig. 10(b) have a similar tendency with the results for $N = 18$ because there are still some false negatives and recall decreases even when the correlation threshold is set small. However, the results in Figs. 10(a) and 10(c) demonstrate a different tendency. When N is large, the low correlation threshold removes false negatives and the cost is only affected by precision and $\aleph^*$. In that case, our framework achieves better performance than that of $l = 0.1$. This is because when N is large, $\aleph^*$ also increases and precision makes a larger contribution to the remaining diagnosis costs. Even though $l = 0.01$ and 0.05 achieve lower costs, there are more risks of recall-decreases, as can be seen from Fig. 10(b).

We concluded from these observations that our framework worked stably both when N was large and small. This conclusion is suitable for our framework because i) some application performance data may not be able to obtained during long periods from application administrators and ii) long periods of performance data have a greater probability of containing totally different types of causes such as disk contention and workload surge of applications. These may not be captured by the Pearson correlation because individual cause metrics should correlate with a type of anomalies but not correlate with the others, which bring about temporal correlations. Even in that situation, our framework should work well if we set T_{nocor} smaller because there can be valid association rules between individual cause metrics and application performance.

4. TESTING ON REAL DATA SET

This section explains how we adapted a prototype of our framework to a real data set and investigated the efficiency of our framework.

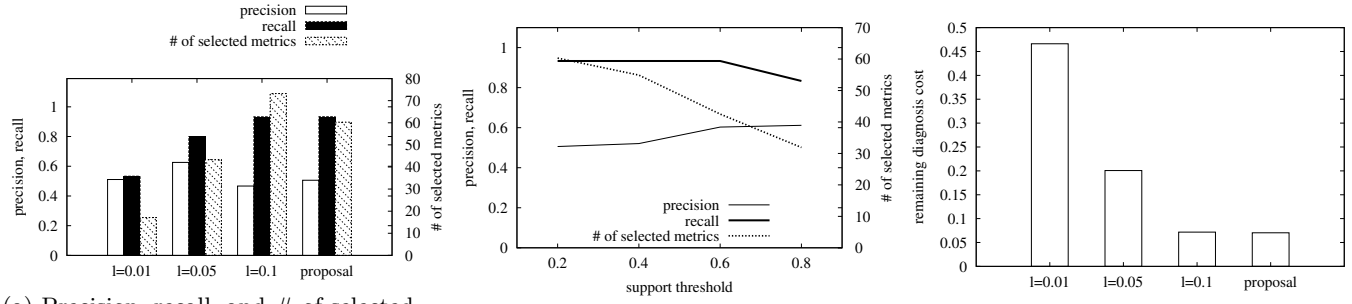


(a) Precision, recall, and # of selected metrics

(b) Effect of support threshold

(c) Remaining diagnosis cost

Figure 7: Results from cases of CPU contention



(a) Precision, recall, and # of selected metrics

(b) Effect of support threshold

(c) Remaining diagnosis cost

Figure 8: Results from cases of network contention

A. Setups

The real data set we used in this section was data that were obtained from a real VDI service, which hosted a total of 290 users (VMs). The VDI service was configured with five host servers, two FCoE 10-G physical switches, and physical storage. The five host servers had the same configuration: 256 GB of RAM, Intel Xeon E5-2695v2 2.4 GHz 12 core x 2 processors, and Windows server 2008 (Hyper-V was used for the hypervisor). Each host server used an 8-TB RAID6 volume on the physical storage, which was used for virtual disks of VMs on the host.

The metrics data were obtained by using WMI and SNMP with a fixed interval (60 s), similarly to our experimental environment explained in Subsection 3.1. We also obtained metrics from storage, which supported various kinds of private MIBs, which included metrics related to storage such as read/write response times, I/O per second, read/write throughput, and cache hits for each volume. The total number of metrics we obtained from the cloud infrastructure was approximately 49,000 (fluctuating by time), which include approximately 35,000 metrics that were associated with a specific VM. The performance of the VDI service was measured in the same way that we used in the experiment in Subsection 3.1. We ran the benchmark script on a virtual desktop and measured user-experienced performance.

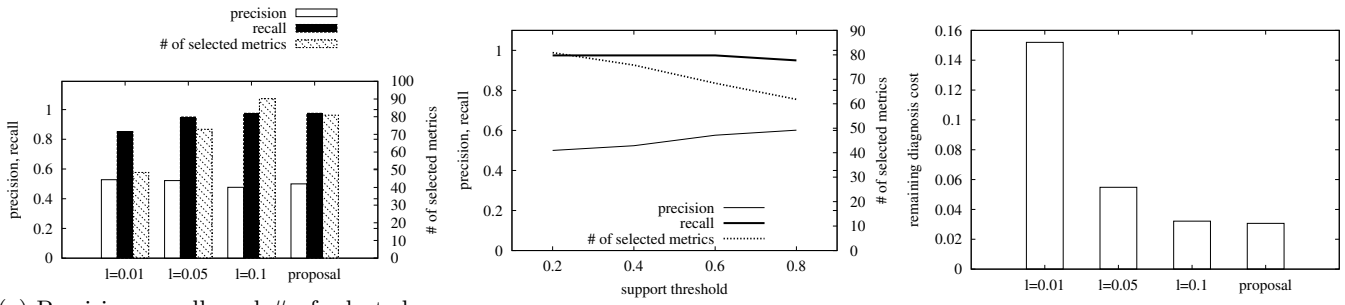
When we ran the benchmark script for one week, we encountered 13 times performance anomalies. We found from additional investigations that the root cause of these performance anomalies was disk resource contentions brought about by synchronous virus scans on a large number of VMs. Therefore, the correct bottleneck attribute was the disk. However, the correct causal VMs were not identical because in order to investigate these, we have to investigate the logs obtained from virus scan software on all VMs, which was impossible for us to do. We therefore have only presented a summary of the metrics that were selected in cause diagnosis. We input application performance data 24 time units in length. When sample data size was 24 units, the critical value of hypothesis testing was 0.344, 0.404, and 0.515 with significance levels that corresponded to 0.1, 0.05, 0.01.

B. Results

Figure 11(a) plots the results obtained from bottleneck diagnosis, where the precision (recall) has the same definition as that provided in Subsection 3.3.1. As we can see from the figure, there is the same tendency as that in the experimental results in Fig. 5(c), i.e., when l_m is set small (e.g., $l_m = 1$), there are some false negatives. Therefore, using only the most correlated metric has to be avoided in bottleneck diagnosis, which is the same conclusion that we reached in our experimental evaluation.

Figure 11(b) shows the number of selected metrics. Note its similarity with the experimental evaluation, in which we assumed that bottlenecks had been accurately diagnosed. Our framework can capture larger numbers of cause metrics than the correlation approaches ($l = 0.01$ and 0.05). Figure 11(c) shows the effect of setting T_{supp}. As can be seen from in Fig. 11(c), larger numbers of metrics are selected when T_{supp} is set small. That indicates there are large numbers of metrics that have temporal correlations. In this case, there are many temporally correlated cause metrics because the degradation in application performance is brought about by a large number of VMs.

Figure 12 plots three examples of cause metrics selected by cause diagnosis on a real data set and compares application performance. The figure indicates the *read bytes/sec* metrics of three virtual disks on VMs. These metrics (denoted *vm1*, *vm2*, and *vm3*) have temporally correlated behaviors;

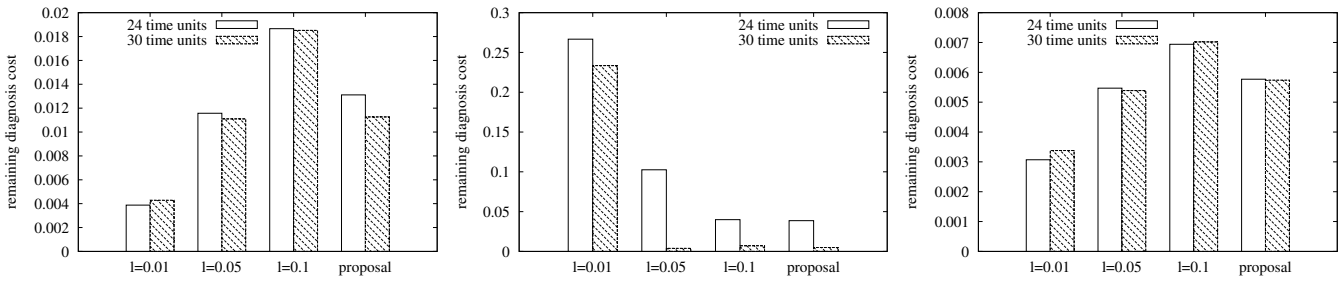


(a) Precision, recall, and # of selected metrics

(b) Effect of support threshold

(c) Remaining diagnosis cost

Figure 9: Results from cases of disk contention



(a) CPU contention cases

(b) Network contention cases

(c) Disk contention cases

Figure 10: Effect of analysis time period length

when application performance increases at 11 through 24 time units, these three metrics increase at the subset of these time units, and they have respective correlations of 0.358, 0.478, and 0.577. Then, *vm1* is not captured by the Pearson correlation with significance levels 0.01 and 0.05, and *vm2* is also not captured by that with significance level of 0.01. Our framework, on the other hand, can capture these metrics because they have valid association rules. *vm1*, *vm2*, and *vm3* have confidence measures of 1.000, 1.000, 0.889 and support measures that correspond to 0.214, 0.357, and 0.571. Their discretizing thresholds have the same value 0.0 because that maximizes the support measure.

Our framework, however, captures some obvious false positives. Figure 13 plots the examples of false positive metrics. These metrics have negative correlations with the application performance. For example, the *vm4* metric has a confidence of 0.818, a support of 0.643, and a discretizing threshold of $4.19e^{+06}$. They do not indicate the cause of performance bottlenecks because if they indicate the cause, the *read bytes/sec* metrics of virtual disks should increase during the performance anomaly period. We inferred that these metrics, which had negative correlations, indicated VMs that suffered from degraded performance. If so, these metrics also bring to light quite useful information for cloud administrators.

5. DISCUSSION

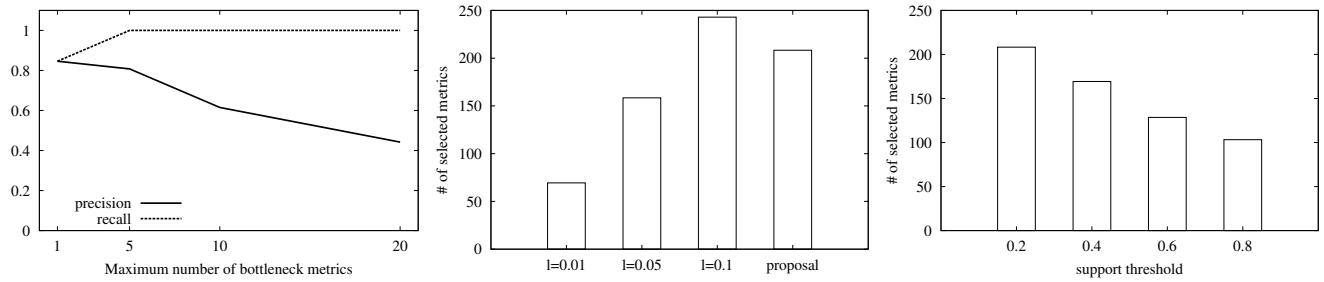
We evaluated our framework and found that it worked well even when short periods of application performance data were obtained, which we discussed in Section 3. This feature and its model-less design will provide excellent adaptation for dynamic application revisions in the cloud. However, there were some limitations in our framework.

First, our framework only works when the cause of a performance anomaly is a resource bottleneck in the cloud and the bottleneck is created by VMs. Some performance problems are brought about by anomalous host servers and hypervisors or faults on physical equipment (e.g., switches and storage). These kinds of causes can be inferred from text log data (e.g., error and warning messages) obtained from the hypervisor logs or SNMP trap data, which are managed by the cloud administrator. Those kinds of causes can be inferred with higher accuracy and coverage by using the text log data. However, no performance bottlenecks caused by VMs remain in any text logs. Therefore, our framework only focused on resolving that issue.

Second, our framework did not pinpoint the causes of performance anomalies. It required some supports from users because the results still contained some false positives. These false positives are often unavoidable because some metrics are correlated with application performance by chance. Users have to investigate the results to remove these false positives. Even though our framework was expected to reduces the cost of investigations as was explained in Section 3, the verification of the cost estimation still remains for future work.

Third, our framework needs to obtain application performance data from application administrators, who have to include performance anomaly periods. Therefore, performance anomalies have to be detected by them. They should detect performance anomalies by directly monitoring application performance such as response times or throughput because model-based or training-based approaches are not efficient for the detections in highly dynamic applications.

Finally, we will discuss the computational cost of our framework. Our framework has scalable features because the computation of correlations and association rules are independent between metrics. This means that the computation



(a) Results from bottleneck diagnosis (b) # of selected metrics (c) Effect of support threshold

Figure 11: Results from testing on real data set

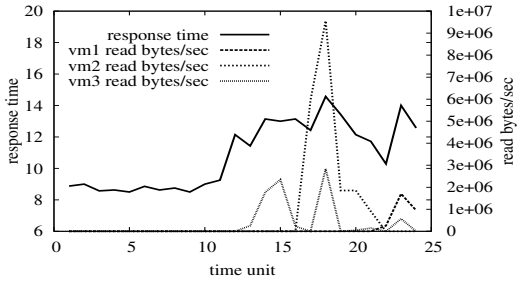


Figure 12: Examples of cause metrics on real data set

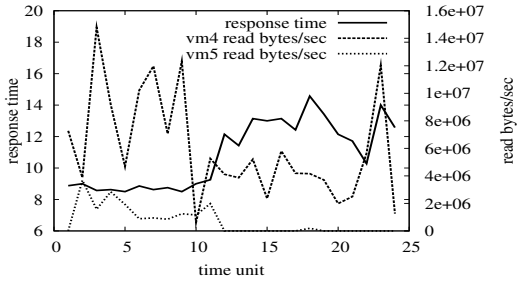


Figure 13: Examples of false positives on real data set

can be parallelized. The computational cost of our framework mainly depends on the length of analysis time periods. The computational order, especially in Algorithm 2, is two powers of the analysis period length. That may cause long computation times when excessive application performance data are input.

6. RELATED WORK

There is much existing work that has investigated the performance problems in the cloud. This section introduces some of these that are related to our work.

Several researches [3, 9, 16, 24, 30] have diagnosed the cause of performance problems by identifying which performance issues that occurred in the past were the same or closest to current performance issues. Fu et al.'s approach [9] characterizes the past performance issues based on class association rules (CARs) and identifies the metrics that will help to pinpoint the root cause of a newly encountered issue by using the CARs. These approaches require training data set and only work for recurring performance problems.

Some applications in the cloud bring about stable correlating behavior between metrics when the applications are sound. Some researchers [2, 14, 22] diagnosed the cause of performance problems by finding breaks in the stable correlating behavior between metrics. Breaks in correlations provides a chance to predict severe performance issues and information on cause diagnosis. These approaches focus on the issues with applications, and not on resource contentions between applications in the cloud.

Xiong et al. [29] proposed a model-driven framework that builds regression models to diagnose performance bottlenecks in applications. Their approach infers the bottleneck point (e.g., a VM or a host) and its attribute (e.g., CPU or memory) from the regression models, which is similar to our bottleneck diagnosis. However, this framework does not infer VMs that created performance bottlenecks.

Resource contentions in the cloud have been addressed as problems in VM placements or scheduling in some researches [6, 7, 19]. Do et al. [7] used canonical correlation analysis (CCA) for application profiling which enables to identify dominant factors in application performance and predict resource usage of applications. The profiling helps the decisions of VM placements. These approaches assume the applications are sound and do not address application performance anomalies that are caused by anomalous resource usage in some applications.

7. CONCLUSION

We proposed an analysis framework for the cause diagnosis of application performance anomalies. Our framework captured cause metrics that had temporal correlations with application performance, which is difficult for methods that use the standard Pearson correlation, by finding association rules between each metric and application performance.

We confirmed the efficiency of our framework through our experimental evaluation. The experimental results revealed that it achieved better performance even with few application performance data. In addition to the evaluation of accuracy and coverage, we also evaluated our framework from the perspective of the remaining diagnosis cost of our framework users. Our framework was expected to reduce the cost by 84.8% at most because it reduced many false negatives. We also adapted our framework to a real data set obtained from a VDI service. The results from the real data set indicated that some cause metrics have temporal correlations, which are difficult to be captured by the Pearson correlation.

Our framework should be efficient for agile application developments and revisions because it does not involve any performance models. It works as soon as the performance data are obtained even when an application is newly developed and revised, and even when few data are obtained.

8. REFERENCES

[1] R. Agrawal, T. Imieliński, and A. Swami. Mining association rules between sets of items in large databases. In *ACM SIGMOD Record*, volume 22, pages 207–216. ACM, 1993.

[2] A.-F. Antonescu and T. Braun. Improving management of distributed services using correlations and predictions in SLA-driven cloud computing systems. In *Network Operations and Management Symposium (NOMS), 2014 IEEE*, pages 1–8. IEEE, 2014.

[3] P. Bodik, M. Goldszmidt, A. Fox, D. B. Woodard, and H. Andersen. Fingerprinting the datacenter: automated classification of performance crises. In *Proceedings of the 5th European conference on Computer systems*, pages 111–124. ACM, 2010.

[4] C. Clark, K. Fraser, S. Hand, J. G. Hansen, E. Jul, C. Limpach, I. Pratt, and A. Warfield. Live migration of virtual machines. In *Proceedings of the 2nd conference on Symposium on Networked Systems Design & Implementation-Volume 2*, pages 273–286. USENIX Association, 2005.

[5] D. J. Dean, H. Nguyen, and X. Gu. Ubl: unsupervised behavior learning for predicting performance anomalies in virtualized cloud systems. In *Proceedings of the 9th international conference on Autonomic computing*, pages 191–200. ACM, 2012.

[6] C. Delimitrou and C. Kozyrakis. Paragon: QoS-aware scheduling for heterogeneous datacenters. *ACM SIGARCH Computer Architecture News*, 41:77–88, 2013.

[7] A. V. Do, J. Chen, C. Wang, Y. C. Lee, A. Y. Zomaya, and B. B. Zhou. Profiling applications for virtual machine placement in clouds. In *Cloud Computing (CLOUD), 2011 IEEE International Conference on*, pages 660–667. IEEE, 2011.

[8] elastic. https://www.elastic.co/.

[9] Q. Fu, J.-G. Lou, Q.-W. Lin, R. Ding, D. Zhang, Z. Ye, and T. Xie. Performance issue diagnosis for online service systems. In *Reliable Distributed Systems (SRDS), 2012 IEEE 31st Symposium on*, pages 273–278. IEEE, 2012.

[10] M. Hahsler, B. Grün, and K. Hornik. arules-A computational environment for mining association rules and frequent item sets. *Journal of Statistical Software*, 14(i15), 2005.

[11] T. Hobfeld, R. Schatz, M. Varela, and C. Timmerer. Challenges of QoE management for cloud applications. *Communications Magazine, IEEE*, 50(4):28–36, 2012.

[12] Hypothesis Testing (Critical value approach). https://onlinecourses.science.psu.edu/statprogram/node/137.

[13] Iperf. http://sourceforge.net/projects/iperf/.

[14] H. Kang, H. Chen, and G. Jiang. Peerwatch: a fault detection and diagnosis tool for virtualized consolidation systems. In *Proceedings of the 7th international conference on Autonomic computing*, pages 119–128. ACM, 2010.

[15] Y. Koh, R. Knauerhase, P. Brett, M. Bowman, Z. Wen, and C. Pu. An analysis of performance interference effects in virtual environments. In *Performance Analysis of Systems and Software, 2007. ISPASS 2007. IEEE International Symposium on* pages 200–209. IEEE, 2007.

[16] M.-H. Lim, J.-G. Lou, H. Zhang, Q. Fu, A. B. J. Teoh, Q. Lin, R. Ding, and D. Zhang. Identifying recurrent and unknown performance issues. In *Data Mining (ICDM), 2014 IEEE International Conference on*, pages 320–329. IEEE, 2014.

[17] C. Luo, J.-G. Lou, Q. Lin, Q. Fu, R. Ding, D. Zhang, and Z. Wang. Correlating events with time series for incident diagnosis. In *Proceedings of the 20th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 1583–1592. ACM, 2014.

[18] Microsoft Hyper-V Server 2012 R2. https://technet.microsoft.com/en-us/library/hh833684.aspx.

[19] R. Nathuji, A. Kansal, and A. Ghaffarkhah. Q-clouds: managing performance interference effects for QoS-aware clouds. In *Proceedings of the 5th European conference on Computer systems*, pages 237–250. ACM, 2010.

[20] Overview of Windows Performance Monitor. https://technet.microsoft.com/en-us/library/cc749154.aspx.

[21] E. Ries. *The lean startup: How today's entrepreneurs use continuous innovation to create radically successful businesses*. Random House LLC, 2011.

[22] B. P. Sharma, P. Jayachandran, A. Verma, and C. R. Das. CloudPD: problem determination and diagnosis in shared dynamic clouds. In *Dependable Systems and Networks (DSN), 2013 43rd Annual IEEE/IFIP International Conference on*, pages 1–12. IEEE, 2013.

[23] Stress. http://people.seas.harvard.edu/~apw/stress/.

[24] Y. Tan, H. Nguyen, Z. Shen, X. Gu, C. Venkatramani, and D. Rajan. Prepare: Predictive performance anomaly prevention for virtualized cloud systems. In *Distributed Computing Systems (ICDCS), 2012 IEEE 32nd International Conference on*, pages 285–294. IEEE, 2012.

[25] C. Wang, S. P. Kavulya, J. Tan, L. Hu, M. Kutare, M. Kasick, K. Schwan, P. Narasimhan, and R. Gandhi. Performance troubleshooting in data centers: an annotated bibliography? *ACM SIGOPS Operating Systems Review*, 47(3):50–62, 2013.

[26] Windows Management Instrumentation Overview. https://technet.microsoft.com/en-us/library/dn265977.aspx.

[27] WMI client. http://pkgs.org/download/wmi.

[28] WMI Performance. http://wutils.com/wmi/root/cimv2/win32_perfrawdata/#Childs.

[29] P. Xiong, C. Pu, X. Zhu, and R. Griffith. vPerfGuard: an automated model-driven framework for application performance diagnosis in consolidated cloud environments. In *Proceedings of the 4th ACM/SPEC International Conference on Performance Engineering*, pages 271–282. ACM, 2013.

[30] Q. Zhu, T. Tung, and Q. Xie. Automatic fault diagnosis in cloud infrastructure. In *Cloud Computing Technology and Science (CloudCom), 2013 IEEE 5th International Conference on*, volume 1, pages 467–474. IEEE, 2013.

,