Performance Issue Diagnosis for Online Service Systems

Qiang Fu[1], Jian-Guang Lou[1], Qing-Wei Lin[1], Rui Ding[1], Dongmei Zhang[1]
Zihao Ye[2], Tao Xie[3]

[1]Microsoft Research Asia
Beijing, China
{qifu, jlou, qlin, juding, dongmeiz}@microsoft.com

[2]Beihang University
Beijing, China
yzhvictor@gmail.com

[3]North Carolina State University
Raleigh, NC, USA
xie@csc.ncsu.edu

Abstract -- Monitoring and diagnosing performance issues of an online service system are critical to assure satisfactory performance of the system. Given a detected performance issue and collected system metrics for an online service system, engineers usually need to make great efforts to conduct diagnosis by first identifying performance issue beacons, which are metrics that pinpoint to the root causes. In order to reduce the manual efforts, in this paper, we propose a new approach to effectively detecting performance issue beacons to help with performance issue diagnosis. Our approach includes techniques for mining system metric data to address limitations when applying previous classification-based approaches. Our evaluations on both a controlled environment and a real production environment show that our approach can more effectively identify performance issue beacons from system metric data than previous approaches.

Keywords-performance issue diagnosis; class association rule; monitoring data analysis

I. INTRODUCTION

In online service systems, performance issue diagnosis typically starts with hunting for a small subset of monitoring data that are symptoms to represent the cause(s) of the performance issues. We name such kind of metrics *performance issue beacons.* Performance issue beacons could be the same as the root causes or could be intermediate useful information that pinpoints to the root causes. For example, the cause of a performance issue is a blocking SQL query, whose execution blocks the execution of other queries accessing the same table. The corresponding observable symptoms on monitoring data can be considered as the issue beacons: the metric on SQL-inducing intensive I/O bytes and the metric of service critical events "SQL query timeout failure". Among a large number of system metrics (e.g., more than 1200 metrics in real production systems under our investigation), identifying these performance issue beacons forms a critical step towards identifying the cause of the performance issues illustrated by the SQL query example. Because the propagation path from root causes to the final performance issue may involve multiple components, there may be multiple performance issue beacons corresponding to different intermediate factors. All of them can provide rich contextual information for diagnosing the root cause.

However, identifying performance issue beacons still remains as a time-consuming and challenging task. System performance issues may be caused by various causes. The huge investigation scope brings a lot of uncertainties and makes the diagnosis time consuming. Systems usually record a large amount of monitoring data. In practice, however, quite often only a small subset of monitoring data is actually related to a given performance issue [1, 2, 3]. The overwhelming amount of irrelevant monitoring data brings challenges for identifying performance issue beacons. Therefore, it is important to create automated tools to improve the effectiveness and efficiency of the identification.

In this paper, we propose a novel approach that effectively and efficiently identifies performance issue beacons for helping engineers diagnose performance issues. This paper makes the following main contributions:

- In order to mine performance issue beacons out of system metrics, we propose a novel approach that consists of metric-outlier detection, class-association-rule (CAR) mining, and log-likelihood ranking.
- We have implemented the proposed approach and applied it to a real production environment in a large software company. The results show that the proposed approach can outperform previous related approaches, and provide useful performance issue beacons to help engineers with their daily tasks of performance diagnosis.

The rest of the paper is organized as follows. Section 2 discusses related work. Section 3 presents a problem statement and preliminaries. Section 4 presents the approach of mining system metrics to identify performance issue beacons. Section 5 presents the evaluation results, and Section 6 concludes the paper.

II. RELATED WORK

Previous approaches [1-5][8, 9] apply statistical analysis to tackle the challenges of scale and complexity in performance monitoring and diagnosis. These approaches statistically analyze traces, console logs, or system metrics. Given the data of system Service-Level-Objective (SLO) states (violation or compliance) and system metrics, Cohen et al. [2, 3] propose an approach to deduce a classification model based on Tree-Augmented-Network (TAN), which uses a few system metrics to predict system SLO states. Their approach identifies the metrics used by the deduced TAN classification model as performance issue signatures for clustering, indexing, and retrieving performance issues.

Bodik et al. [1] adapted their approach by adopting a different model, called L1-Logistic Regression, for identifying highly correlated metrics more accurately.

A straightforward idea of identifying performance issue beacons is to treat the performance issue signatures (constructed by previous classification-based approaches) as performance issue beacons. However, the primary focus of those approaches is discriminating different types of issues instead of providing comprehensive diagnosis information. In addition, such classification-based approaches have three main limitations in practice. First, since they usually learn a model for each individual performance issue, they would suffer from the *over-fitting* problem when learning a classifier for a performance issue occurring in a short interval. Second, since they use only one model representing performance issues, they tend to identify only *general symptoms* as performance issue signatures and miss those *minor symptoms*, which are usually more helpful for deeper diagnosis. Third, they do not fully leverage the contextual information (e.g., adjacent violations are usually caused by the same cause), which can improve the diagnosis accuracy.

III. PRELIMINARIES

A. Metric Preprocessing

For monitoring and diagnosing performance issues, a huge amount of performance data is collected during system executions. The monitoring data is aggregated and measured once per time epoch (e.g., 5 minutes in the online service systems investigated by us) for calculating the values of Key Performance Indicators (KPIs) and system metrics. The most common KPIs include latency and availability. During system operation, if a KPI's value (e.g., average latency) violates the SLO, a *KPI violation* is detected, and the system is said to be in the SLO-violation state. Otherwise, the system is in the SLO-compliance state. System metrics include system-resource usage information (e.g., the CPU utilization, disk-queue lengths, and I/O-operation rate) and the counts of each kind of critical events (e.g., Windows-kernel events and important service events).

B. Basic Concepts

We next formally define some concepts that we use in the problem statement.

Definition 1. System state S_i for the i^{th} time epoch is a binary flag to indicate whether any system KPI value violates an SLO on the i^{th} time epoch.

S_i =1 indicates that the system is in the *SLO-violation* state on the i^{th} time epoch, and S_i=0 indicates that the system is in the *SLO-compliance* state on the i^{th} time epoch.

Definition 2. System metrics M are a vector of metrics' names $< M_1, ... M_k >$. An example is M = <"Processor Usage", "Inactive Sessions",…> , where the first two metrics M_1 and M_2 are "Processor Usage" and "Inactive Sessions", respectively.

Definition 3. System metric record MR_i for the i^{th} time epoch is a vector of metric values $< m_{i,1}, ... m_{i,k} >$ measured on the i^{th} time epoch.

The inputs to our approach are (1) the system metrics M, (2) the sequence of system states S: $<S_1, S_2, \ldots, S_n>$ for n time epochs, and (3) the sequence of system metric records MR: $<MR_1, MR_2, \ldots, MR_n>$ for the same n time epochs. The objective of our approach is to find the performance issue beacons (a subset of metrics $\subset M$) for each performance issue.

IV. APPROACH

Before going into the detail of our approach, we first introduce engineers' experience on manually identifying performance issue beacons. In fact, the basic ideas of our approach come from these experiences in practice, including three key points: (1) *performance issue beacons often have extraordinarily high or low values (or significantly different from their normal value range) during the period when the investigated issue occurs*; (2) *performance issue beacons usually remain in the normal value range when the system is in the SLO-compliance state*; (3) *the adjacent violations within one performance issue may usually be caused by the same cause and would have the same metric values as those of performance issue beacons*.

As the scale of a system increases, manual analysis based on rough qualitative experience becomes inefficient or even infeasible. In this section, we present our approach that applies machine learning to turn such qualitative experience into a quantitative model. Our approach consists of two stages: the training stage and diagnosis stage. During the training stage, we first use an outlier-detection algorithm to detect abnormal values of each system metric (Section 4.1). Then, with the discretized values (indicating normal or abnormal values) of system metrics and the SLO states (violation states or compliance states) of a KPI, we mine Class Association Rules (CARs) [4] from historical data to discover all possible associations between metrics' anomalies and SLO-violation states. These mined CARs are stored as beacon candidates for later processing (Section 4.2). In the diagnosis stage, given a newly detected performance issue, we use log likelihood to calculate a matching score for each stored CAR candidate. We return the CARs with the highest match score as results to engineers (Section 4.3).

A. Metric Discretization

Detecting whether the value of a metric is lower or higher than its normal value range is quite useful for performance diagnosis (we call this detection process as metric discretization). In this subsection, we use outlier detection to discretize system metric values. Such technique reflects the above-mentioned experience in practice, and the technique' effectiveness has also been demonstrated in previous work [1]. Similar to the previous work, we determine the

discretization threshold for a metric by the *p*-percentage of past values of the metric during normal system operation. In our approach, the default value of p is selected as 2, which is similar to that of previous work [1].

After discretization, each value of the metric has been discretized as one of *High*, *Low*, or *Normal*. For each epoch, we have a KPI state and a set of discretized metric values. In this paper, we call the KPI state and discretized metric values in an epoch as a *sample*. Based on all historical samples, we perform the CAR-mining algorithm to discover associations between metric outliers and performance issues, as described in the next subsection.

B. Discovery of Associations Between Metric Outliers and Performance Issues

Performance diagnosis aims to find out the causation between metric outliers and performance issues. For example, its output can be that "*the CPU resource contention causes this performance issue*". Such an expression can be expressed by an association rule such as "{*metric A is Low*, *metric B is High*} => *SLO violation*". Here, the left part "{*metric A is Low*, *metric B is High*}" is an *antecedent* (including multiple metric names and their values, abbreviated as *MC*), and the right part "*SLO violation"* is a *consequence* (denoted as *S*). We aim to mine a set of CARs whose *antecedents* are "*combination of metric anomalies* (*High or Low*)" and the *consequence* is "*SLO violation*".

We apply the algorithm from previous work [4] to identify rules whose support (*sup*), confidence (*conf*), and lift values are above a set of given thresholds, respectively. Then, we further prune every rule $MC \rightarrow S$ that satisfies one of the following conditions:

- There exists a rule $MC_x \rightarrow S$ in R that satisfies $MC_x \subset MC$ and $conf(MC_x \rightarrow S) > conf(MC \rightarrow S)$.
- There exists a rule $MC_y \rightarrow S$ in R that satisfies $MC_y \supset MC$ and $sup(MC_y) = sup(MC)$.

The intuition behind the first condition is that if a metric is related to a cause of performance issues, adding it to the conditions of a rule should increase the rule's confidence. Otherwise, the metric may not be related to the issues. The intuition behind the second condition is that if MC is highly correlated to the performance issues, and all the metrics in the set $(MC_y - MC)$ always occur together with MC, then these co-occurring metrics are also potentially related to the performance issues. Using the rule-pruning strategy, we can reduce redundancies in the mined CARs.

C. Ranking of Relevant Metric Sets

After the training stage (Sections 4.1-4.2), we obtain a set of candidate CARs from historical monitoring data. Each candidate CAR is a combination of metrics and their values that may cause performance degradation. This section illustrates the diagnosis stage where we select the CARs that can best fit the newly encountered performance issue. Intuitively, if a CAR represents the real cause of the issue under investigation, the metrics of its antecedent should be abnormal (High or Low) in the SLO-violation epochs of the issue. On the contrary, the metrics of its antecedent should be normal in the SLO-compliance epochs around the issue. We use log likelihood as the evaluation algorithm for realizing the preceding intuitions.

First, for a given performance issue under investigation, we expand (usually double) the investigation period to include the nearby SLO-compliance epochs. By including the nearby SLO-compliance epochs, we can fully leverage the contrast information to reduce false positives (i.e., non-helpful metrics detected as performance beacons).

Next, for a candidate rule $MC \rightarrow S$ (e.g., {*metric CPU is High*} => *SLO violation*), we calculate the following statistics (i.e., conditional probabilities) from all historical monitoring data. These statistics are used in the log-likelihood computation. Among these statistics, $sup\ (\neg MC)$ is the number of samples that do not satisfy the antecedent MC (i.e., *metric CPU is High*), $\neg S$ denotes the SLO-compliance state.

- $P(S|MC) = sup\ (MC \rightarrow S)/sup\ (MC)$
- $P(\neg S|MC) = 1 - P(S|MC)$
- $P(S|\neg MC) = sup\ ((\neg MC) \rightarrow S)/sup\ (\neg MC)$
- $P(\neg S|\neg MC) = 1 - P(S|\neg MC)$

After that, for evaluating the rule $MC \rightarrow S$, the likelihood probability P_i of the i^{th} epoch in the investigated period can be one of the preceding probability values according to the values of metric vector MR_i and system state S_i of the sample. For example, if MR_i satisfies the antecedent *MC* (i.e., CPU value in MR_i is higher than Th^2_{CPU}), and S_i is 1 (i.e., the SLO-violation state), then the likelihood of epoch *i* is $P(S|MC)$. If MR_i does not satisfy the antecedent *MC*, and S_i is 1, the epoch's likelihood is $P(S|\neg MC)$. Based on the likelihood values of the epochs within the investigated period, we calculate the log likelihood L of the CAR $MC \rightarrow S$ by the equation $L = \sum \log (P_i)$.

With the log-likelihood values for all CAR candidates, we then rank these candidates, and return the top candidates as our analysis results to engineers.

D. Algorithm Discussion

Previous classification-based approaches learn a model from each individual performance issue. These approaches suffer from the over-fitting problem when they deal with short-period performance issues due to insufficient data. If we compose the data of multiple performance issues together to enlarge the data to avoid the over-fitting problem, doing so also introduces two other problems that the previous classification-based approaches fail to address. First, since multiple performance issues may be caused by different causes that correspond to very different system-metric symptoms, learning one classifier from the data caused by *mixed causes* may degrade the diagnosis accuracy. Second, there exists the *coupling-effect* phenomenon among system metrics. For example, many different causes (e.g., a blocking SQL query or a database server with high CPU usage)

may lead to frequent occurrences of the service critical event "SQL query timeout". Therefore, the metric value of "number of blocking query" and the metric value of "number of SQL query timeout event" usually increase together. Similarly, the metric value of "CPU usage" and "number of SQL query timeout event" may also increase together. However, the previous classification-based approaches tend to select only the dominant metric of "number of SQL query timeout event" because it has the best prediction accuracy on SLO violations caused by both of the two causes. These approaches fail to detect the metrics of "CPU usage" and "blocking query", which contain diagnosis information in fine granularity.

Our approach addresses the over-fitting problem by mining candidate models from the whole historical data first, and then selects the best one from the candidate models by matching them with the performance issue under investigation. In addition, CAR mining can discover all rules that satisfy some basic requirements including above the support and confidence thresholds. It can mine not only dominant metrics but also other metrics of interest as rules. For the preceding example, we select all of the three metrics. For the metric of "CPU usage", we can select it out because it can associate some of SLO violations (caused by high CPU usage) well enough even when it does not associate with SLO violations caused by blocking queries. Similarly, we can also select out the metric of "blocking query".

E. Adaptation in Practice

We have applied our approach to a real-world web-based multi-tier online system that consists of IIS servers, application servers, and SQL servers. Each tier contains several hosts that share similar hardware and software configurations. For example, the web front end is served by a web server farm where a set of IIS servers serve the incoming user requests behind a load balancer.

Service-layer-based analysis. We identify performance issue beacons for each service layer of the system because each tier often has the same types of metrics. In particular, we first discretize the monitored metrics host by host; then we merge the resulting data from different hosts of the same service tier to form the historical training set. In the next step, we run the CAR miner to obtain a set of performance issue beacons. Note that the historical training set allows the CAR miner to aggregate the support of CARs from similar hosts, thus yielding more accurate estimates.

Given a newly detected performance issue, we discretize the metrics for each host of the first tier (i.e., IIS servers) on each epoch during the time period of the issue, and then select the hosts that have abnormal (i.e., high/low) metric values. For each of these selected hosts, we use the step in Section 4.3 to calculate the score of each mined CAR. Similar procedures are also conducted on the hosts of the second tier and the third tier to evaluate CARs, respectively. At last, the *k* CARs with the highest scores are produced as our final results.

Incremental mining. The real-world system under investigation continuously generates a large amount of monitored metric data. In general, a CAR algorithm needs to scan all historical data to produce CARs. However, due to the storage and computational constraints, we may not be able to archive all historical data. In addition, running the CAR miner over a huge data set is computationally expensive. We propose two strategies to address these practical challenges. First, we reduce the number of samples by exploiting a property of the performance data. In a real-world scenario, the performance of the service system is in the SLA-compliance state most of the time, and the corresponding metric patterns are of no interest. In contrast, the metric patterns from the time intervals where the service is in the SLA-violation state are of interest. Therefore, we can largely reduce the number of archived training samples by removing redundant samples with the SLA-compliance state, and assign each compliance epoch a weight to count the number of removed epochs. For example, we randomly keep a moderate-size set of samples with the SLA-compliance state (e.g., 1000 samples). Second, we also apply an incremental miner [6, 7] to further reduce the computational cost of CAR mining.

V. EVALUATIONS

To evaluate our approach, we conducted evaluations with two environments (TPC-W and a production system called SystemX), respectively. TPC-W is to serve synthetic workloads in a controlled laboratory environment, and SystemX is a real production environment that serves real users. We aggregate monitoring data in every time epoch (i.e., 5 minutes) to calculate the values of KPIs and system metrics. The calculated values of KPIs and metrics on each time epoch form a value vector as a data sample. In order to compare the effectiveness of different approaches, we also implemented the TAN classifier [2, 3] and the L1-Logistic Regression [1]. We measure different approaches' identification results by the *accuracy* and the *coverage*. The *accuracy* is the ratio between the number of identified *real* issue beacons and the total number of identified issue beacons. The *coverage* is the ratio between the number of identified *real* issue beacons and the total number of real issue beacons. For both accuracy and coverage, the higher the better.

A. Evaluations on TPC-W

Cause injection. Many performance issues are caused by exhaustion of specific system resources. To synthesize performance causes, we use a standalone program to exhaust specific system resources, including CPU exhaustion, disk IO exhaustion, or their combinations, according to our specified configuration. By running such resource-eating program, we inject root causes to produce system KPI violations.

Transaction workload. In the evaluations, we use two workload patterns: (1) we use a periodical workload by

changing the number of concurrent clients from 50 to 150 for each hour, and (2) each client sends a request to trigger a system-processing transaction, waits for 10 milliseconds thinking time, and then sends a request again, and so on.

Transaction types. In the evaluations, we intend to construct different types of transactions in terms of their different extents of consuming specific resources.

- CPU-intensive transactions: transactions that execute a CPU-intensive servlet "A" that executes one loop with a randomly chosen number of iterations (from the range of 1000 to 2000), each of which prints a number to the screen.
- Disk-IO-intensive transactions: transactions that execute a disk-IO-intensive servlet "B" that opens and reads the content of a randomly selected file from 1000 files (with the file size randomly chosen from the range of 2MB to 4MB and the file content as constant characters).
- Mixed-intensive transactions (i.e., both CPU-intensive and disk-IO-intensive ones): transactions that execute a servlet "C" that (1) executes loops; (2) opens and reads the content of randomly selected files.

System metrics. During the time period of executing the transactions, we collect about 90 system metric data related to Cache, LogicalDisk, Memory, Network Interface, PhysicalDisk, Process, Processor, and so on.

Latency KPI. By parsing the server logs, we obtain each request's latency. We calculate the requests' 50 percentile latency in each epoch (5 minutes) as the latency KPI.

In the evaluations, we use three different patterns of mixing different root causes with different proportions. In Time Pattern A, the lengths of the periods with the effect of CPU exhaustion, disk-IO exhaustion, and both CPU and disk-IO exhaustion are *24 hours, 3 hour,* and *3 hour*, respectively. In Time Pattern B, they are *24 hour, 24 hour,* and *24 hour*, respectively. In Time Pattern C, they are *24 hours, 12 hours, and 6 hours,* respectively. Table 1 shows the comparison results.

TABLE 1. THE ACCURACIES OF DIFFERENT APPROACHES

Approach	Accuracy on beacon identification			
	Pattern A	Pattern B	Pattern C	Avg.
TAN	0%	0%	0%	0%
L1-LR	0%	14%	0%	5%
Ours	30%	50%	30%	36%

TAN identifies only one metric "Committed Memory Bytes In Use" as issue beacons. We investigate the detailed results to figure out why TAN does not work. In our evaluations, the values of "Committed Memory Bytes In Use" remain as a stable range of 20%~25% when we do not run the resource-eating program. However, the values increase to the range of 30%~55% when we run the resource-eating program to produce SLA violations. TAN automatically constructs the prediction model as follows: if the value of "Committed Memory Bytes In Use" is larger than 30%, then the system is in the SLA-violation state; otherwise, the system is in the SLA-compliance state. Because such a prediction model can achieve 100% accuracy, TAN identifies only this metric; such identification is in fact resulted from a coupling effect. However, such identified metric may not be able to provide explicit and direct help for quick diagnosis.

For our approach and L1-Logistic Regression, we observe that our approach can outperform L1-Logistic regression under such complex situations (i.e., multiple root causes taking effect within the period). Specifically, for Pattern A, L1-Logistic Regression identifies only two metrics "Server\Sessions Timed Out" and "Terminal Services\Active Sessions" as issue beacons; for Pattern C, it identifies seven metrics as issue beacons: "Memory\Cache Bytes", "Memory\% Committed Bytes In Use", "System\System Calls/sec", "Terminal Services\Active Sessions", "Server\Sessions Timed Out", "Terminal Services\Total Sessions", and "System\Threads". All of these seven identified issue beacons are false positives. L1-Logistic Regression can detect only a real issue beacon as "PhysicalDisk(_Total)\Avg. Disk sec/Transfer" among the identified seven metrics for Pattern B. In contrast, our approach can detect real issue beacons for all of the three Time Patterns. For example, for Time Pattern A, our identified issue beacons are listed in Table 2.

TABLE 2. OUR IDENTIFIED ISSUE BEACONS FOR TIME PATTERN A

Metric index	Metrics identified by our approach
1	Memory\% Committed Bytes In Use
2	Terminal Services\Inactive Sessions
3	System\Processor Queue Length
4	Processor(_Total)\% Processor Time
5	Cache\Data Flush Pages/sec
6	System\File Write Bytes/sec
7	Cache\Data Flushes/sec
8	Memory\Pages/sec
9	Cache\Read Aheads/sec
10	Cache\Pin Read Hits %

B. Empirical Study on a Real Production System

We have deployed our tools (implementation of our approach) in a real production service system to help engineers on their daily tasks of performance diagnosis. In this subsection, due to company-confidentiality policies, we show only the empirical results based on data of recent two months from the internal environments where there are 13 deployed web front-end servers. The environments collect more than 1129 system metrics, which include 549 performance counters and 580 service critical events. The KPI used in the evaluations is Percentile-95 Latency. The KPI-violation threshold is 1500ms. These settings are defined by the product team for managing the service system.

We use 36 performance issues occurring within two months to conduct the evaluations. Because the evaluations need to involve great manual efforts of engineers in the product team, and the approach of L1-Logistic Regression is the most recent related work and has been shown to per-

form better than a TAN-based approach [2], we evaluate only the results of our approach and L1-Logistic Regression. We ask production engineers' help for labeling out real issue beacons among only the union set of identified issue beacons by our approach and by L1-Logistic Regression. The labeled results are used to calculate the accuracy and the coverage of each approach. A metric is labeled as an issue beacon if it provides explicit helpful information for identifying the cause of performance issues in diagnosis practice of engineers.

Figures 1 and 2 show the coverage and accuracy results, respectively, as we change the threshold of the number of selected metrics from 1 to 10. We can observe that our approach can achieve better coverage and accuracy in all cases than the approach of L1-Logistic Regression.

We next illustrate one detailed case. Because plans of SQL stored procedures are not all cached, the SQL server needs to recompile execution plans frequently. The frequent recompilation occupies the SQL server's processor intensively and causes compilation locks. Such factor can cause SQL queries' waiting time to be much longer than their normal values. It can finally lead to long latency for serving user requests. At the same time, the count of critical service events of "Slow Query Duration" becomes large.

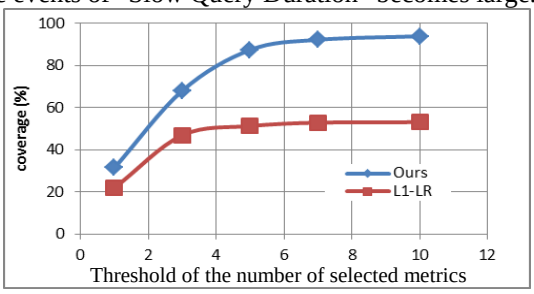

Figure 1. The coverage of our approach and L1-Logsictic Regression

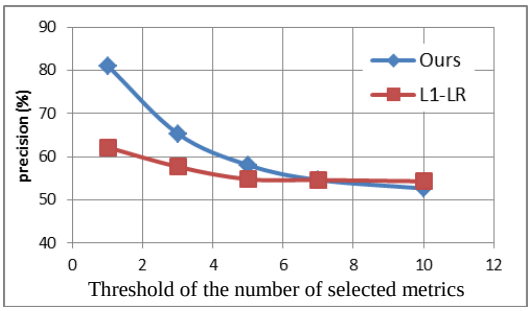

Figure 2. The accuracy of our approach and L1-Logsictic Regression

Our identified top 10 metrics contain 7 correct metrics that identify the following performance issue beacons: *Service critical event "Slow query duration"*, *Performance counter "SQL CPU usage"*, and *Performance counter "SQL Re-Compilations"*. We consider these metrics as helpful, and thus as real issue beacons because they can quickly guide engineers to move their focus to problems related to CPU/Re-compilation of a SQL server. In a complex multi-tier system, such hints are very helpful to reduce the problem-investigation effort and to speed up the diagnosis process. In contrast, the approach of L1-Logistic Regression identifies only two metrics as issue beacons with one being a real issue beacon *"SQL query duration"* and the other being a false positive.

VI.CONCLUSION

We have proposed an approach of performance issue diagnosis for online service systems. In our approach, we analyze system metrics to identify comprehensive performance issue beacons for engineers to diagnose performance issues. In particular, from system metric data, our approach includes three steps based on class-association-rule mining to identify performance issue beacons. Compared with a previous classification-based approaches [1, 2, 3], our approach achieves better results. The evaluation results on both a controlled environment and a real production environment show the benefits of our approach: our approach can sift through a large amount of system metric data and identify helpful performance issue beacons for engineer to conduct performance issue diagnosis.

REFERENCES

[1] P. Bodik, M. Goldszmidt, A. Fox, D. B. Woodard, and H. Andersen, "Fingerprinting the datacenter: automated classification of performance crises", In *Proc. of EuroSys*, pp.111-124, 2010.

[2] I. Cohen, M. Goldszmidt, T. Kelly, J. Symons, and J. S. Chase, "Correlating instrumentation data to system states: A building block for automated diagnosis and control", In *Proc. of OSDI*, pp.231-244, 2004.

[3] I. Cohen, S. Zhang, M. Goldszmidt, J. Symons, T. Kelly, and A. Fox, "Capturing, indexing, clustering, and retrieving system history", In *Proc. of SOSP*, pp. 105-118, 2005.

[4] J. Li, H. Shen, and R. W. Topor, "Mining optimal class association rule set", In *Proc. of PAKDD*, pp. 364-375, 2001.

[5] J. Lou, Q Fu, S. Yang, Y. Xu, and J. Li, "Mining invariants from console logs for system problem detection", In *Proc. of USENIX ATC*, pp. 231-244, 2010.

[6] N. Jiang, L. Gruenwald, "Research issues in data stream association rule mining", *ACM SIGMOD Record*, vol.35, issue 1, March 2006.

[7] P. S.M. Tsai, C.-C. Lee, and A. L.P. Chen, "An efficient approach for incremental association rule mining", In *Proc. of PAKDD*, pp. 74-83, 1999.

[8] W. Xu, L. Huang, A. Fox, D. Patterson, and Mi. I. Jordan, "Detecting large-scale system problems by mining console logs", In *Proc. of SOSP*, pp. 117-132, 2009.

[9] C. Yuan, N. Lao, J.R. Wen, J. Li, Z. Zhang, Y.M. Wang, and W. Y. Ma, "Automated known problem diagnosis with event traces", In *Proc. of EuroSys*, pp. 375-388, 2006.

[10] http://www.tpc.org/tpcw/