2013 IEEE 33rd International Conference on Distributed Computing Systems

Detecting Transient Bottlenecks in n-Tier Applications through Fine-Grained Analysis

Qingyang Wang [1], Yasuhiko Kanemasa [2], Jack Li [1], Deepal Jayasinghe [1]
Toshihiro Shimizu [2], Masazumi Matsubara [2], Motoyuki Kawaba [2], Calton Pu [1]

[1] *College of Computing, Georgia Institute of Technology*
[2] *Cloud Computing Research Center, FUJITSU LABORATORIES LTD.*

***Abstract*—Identifying the location of performance bottlenecks is a non-trivial challenge when scaling n-tier applications in computing clouds. Specifically, we observed that an n-tier application may experience significant performance loss when there are transient bottlenecks in component servers. Such transient bottlenecks arise frequently at high resource utilization and often result from transient events (e.g., JVM garbage collection) in an n-tier system and bursty workloads. Because of their short lifespan (e.g., milliseconds), these transient bottlenecks are difficult to detect using current system monitoring tools with sampling at intervals of seconds or minutes. We describe a novel transient bottleneck detection method that correlates throughput (i.e., request service rate) and load (i.e., number of concurrent requests) of each server in an n-tier system at fine time granularity. Both throughput and load can be measured through passive network tracing at millisecond-level time granularity. Using correlation analysis, we can identify the transient bottlenecks at time granularities as short as 50ms. We validate our method experimentally through two case studies on transient bottlenecks caused by factors at the system software layer (e.g., JVM garbage collection) and architecture layer (e.g., Intel SpeedStep).**

I. Introduction

Achieving both good performance and high resource utilization is an important goal for enterprise cloud environments. High utilization is essential for high return on investment for cloud providers and low sharing cost for cloud users. Good performance is essential for mission-critical applications (e.g., web-facing e-commerce applications) with Service Level Agreement (SLA) guarantees such as bounded response time. Unfortunately, achieving both objectives for mission-critical applications has remained an elusive goal. Concretely, both practitioners and researchers have experienced wide-range response time variations in clouds during periods of high utilization. A practical consequence is that enterprise cloud environments have adopted conservative (low) average utilization (e.g., 18% in [19]).

In this paper, we describe clear experimental evidence that shows transient bottlenecks being an important contributing factor to the wide response time variations. Using extensive measurements of an n-tier benchmark (RUBBoS [1]), we demonstrate the presence of transient bottlenecks with a short lifespan on the order of tens of milliseconds. Transient bottlenecks can arise from several factors at different system layers such as Java Virtual machine garbage collection (JVM GC) at the software layer and Intel SpeedStep at the architecture layer. These factors interact with normal bursty workloads [14] from clients, often leading to transient bottlenecks that cause overall performance degradation. The discovery of these transient bottlenecks is important as they will cause wide-range response time variations and limit the overall system performance while all the system resources are less than 100% utilized. Specifically, we have found that frequent transient bottlenecks can cause a long-tail response time distribution that spans a spectrum of 2 to 3 orders of magnitude, which can lead to severe violations of strict Service Level Agreements (SLAs) required by web-facing e-commerce applications (see Section II-B).

The study of transient bottlenecks has been hampered due to many transient bottlenecks being short-lived (on the order of tens of milliseconds). From Sampling Theory, these transient bottlenecks would not be detectable by normal monitoring tools that sample at time intervals measured in seconds or minutes. These monitoring tools incur very high overhead at sub-second sampling intervals (about 6% CPU utilization overhead at 100ms interval and 12% at 20ms interval). By combining fine-grained monitoring tools and a sophisticated analytical method to generate and analyze monitoring data, we are able to find and study transient bottlenecks.

The first contribution of this paper is a novel transient bottleneck detection method, which is sensitive enough to detect transient bottlenecks at millisecond level. Our method uses passive network packet tracing, which monitors the arrival and departure time of each request of each server at microsecond granularity with negligible impact on the servers. This data supports the counting of concurrent requests and completed requests at fine time granularity (e.g., 50ms). For sufficiently short time intervals, we can use the server request completion rate as throughput, and concurrent requests as server load, to identify transient performance bottlenecks (Utilization Law [9]) at time granularity as short as 50ms (See Section III).

The second contribution of the paper is a detailed study of various system factors that cause the transient bottlenecks in the system. In this paper we focus on two representative factors: one at the system software layer and the other at the architecture layer. At the system software layer, JVM garbage

1063-6927/13 $26.00 © 2013 IEEE
DOI 10.1109/ICDCS.2013.17

Software Stack

Web Server	Apache 2.0.54
Application Server	Apache Tomcat 5.5.17
Cluster middleware	C-JDBC 2.0.2
Database server	MySQL 5.0.51a
Sun JDK	jdk1.5.0_07, jdk1.6.0_14
Operating system	RHEL 6.2 (kernel 2.6.32)
Hypervisor	VMware ESXi v5.0
System monitor	esxtop 5.0, Sysstat 10.0.0

(a) Software setup

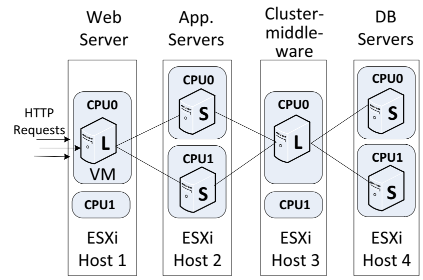
ESXi Host Configuration

Model	Dell Power Edge T410
CPU	2* Intel Xeon E5607, 2.26GHz Quad-Core
Memory	16GB
Storage	7200rpm SATA local disk

VM Configuration

Type	# vCPU	CPU limit	CPU shares	vRAM	vDisk
Large (L)	2	4.52GHz	Normal	2GB	20GB
Small (S)	1	2.26GHz	Normal	2GB	20GB

(b) ESXi host and VM setup



(c) 1L/2S/1L/2S sample topology

Fig. 1: Details of the experimental setup.

collections in a Java-based server happen frequently especially when the server is at high resource utilization and cause frequent transient bottlenecks for the server (see Section IV-A). At the architecture layer, the Intel SpeedStep technology unintentionally creates frequent transient bottlenecks due to the mismatch between the current CPU clock speed and the bursty real-time workload on the server (See Section IV-C).

The rest of the paper is organized as follows. Section II shows the wide-range response time variations using a concrete example. Section III introduce our transient bottleneck detection method. Section IV shows two case studies of applying our method to transient bottlenecks. Section V summarizes the related work and Section VI concludes the paper.

II. Background and Motivation

A. Experimental Setup

We adopt the RUBBoS standard n-tier benchmark, based on bulletin board applications such as Slashdot [1]. RUBBoS can be configured as a three-tier (web server, application server, and database server) or four-tier (addition of clustering middleware such as C-JDBC [11]) system. The workload consists of 24 different interactions. The benchmark includes two kinds of workload modes: browse-only and read/write mixes. We use browse-only workload in this paper.

We run the RUBBoS benchmark on our virtualized testbed. Figure 1 outlines the software components, ESXi host and virtual machine (VM) configuration, and a sample topology used in the experiments. We use a four-digit notation $\#W/\#A/\#C/\#D$ to denote the number of web servers, application servers, clustering middleware servers, and database servers. Each server runs on top of one VM. We have two types of VMs: "L" and "S", each of which represents a different size of processing power. Figure 1(c) shows a sample 1L/2S/1L/2S topology. The VMs from the same tier of the application run in the same ESXi host. Each VM from the same tier is pinned to separate CPU cores to minimize the interference between VMs. Hardware resource utilization measurements (e.g., CPU) are taken during the runtime period using Sysstat at one second granularity and VMware esxtop at two second granularity.

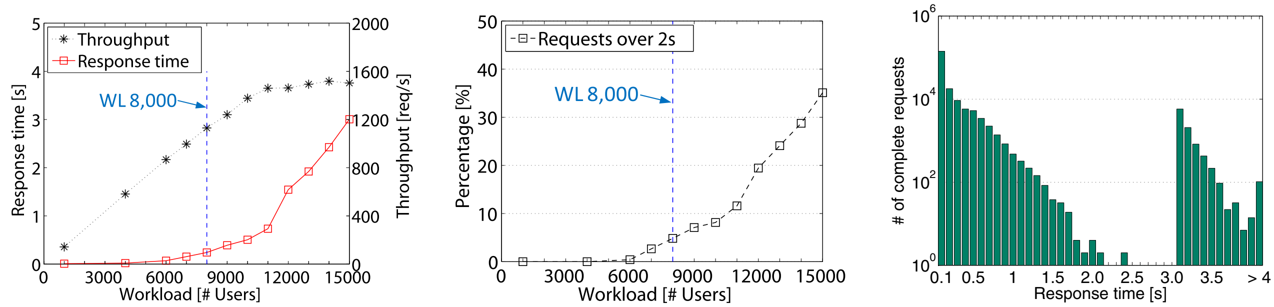
B. Why Are Transient Bottlenecks a Problem?

We use an example where the response time of an n-tier system presents wide-range variations while the system is far from saturation. The example was derived from a three-minute experiment of RUBBoS running on a four-tier configuration (1L/2S/1L/2S, see Figure 1(c)).

Figure 2(a) shows the system throughput increases linearly from a workload of 1,000 concurrent users to 11,000, but after 11,000, the throughput becomes flat and the average response time increases dramatically. The interesting observation is that before the throughput reaches the maximum, for example, from WL 6,000 to 11,000, the average response time already starts increasing. In particular, Figure 2(b) shows that the percentage of requests with response time over 2s starts increasing after WL 6,000, which means that the system performance starts deteriorating far before the system reaches the maximum throughput. Figure 2(c) further shows the response time distribution of the system at WL 8,000, which presents a clear long-tail and bi-modal distribution. In real business situations, there are often cases when web-facing applications have strict service level agreements (SLAs) in terms of end-to-end response time; for example, experiments at Amazon show that every 100ms increase in the page load decreases sales by 1% [12]. In such cases, wide-range variations in response time can lead to severe SLA violations.

In order to diagnose the causes for the wide-range response time variations, we measured the utilization of various resources in each component server of the system. Since the browse-only workload of RUBBoS is CPU intensive, we show the timeline graphs (with one second granularity) of CPU utilization in Figure 3. During the execution of the WL 8,000, both Tomcat and MySQL show less than full CPU utilization, with an average of 79.9% (Tomcat) and 78.1% (MySQL). We also summarize the average usage of other main hardware resources of each server in Table I. This table shows that except for Tomcat and MySQL CPU, the other system resources are far from saturation.

This example shows that monitoring hardware resource utilization at one second granularity is insufficient at identifying the cause of wide-range response time variations, since there is no single saturated resource. Later in Section IV-C we explain that the problem is due to the frequent transient bottlenecks unintentionally caused by Intel SpeedStep technology in MySQL. SpeedStep is designed to adjust CPU clock speed to meet instantaneous performance needs while minimizing



(a) Average end-to-end response time and throughput at each workload

(b) Percentage of requests with response time over two seconds at each workload

(c) Long-tail and bi-modal end-to-end response time distribution at WL 8,000

Fig. 2: A case where the system response time shows wide-range variation far before the system reaches the maximum throughput. Figure 2(c) shows the long-tail and bi-modal end-to-end response time distribution at WL 8,000, which indicates the unstable system performance.

Server/Resource	CPU util. (%)	Disk I/O (%)	Network receive/send (MB/s)
Apache	34.6	0.1	14.3/24.1
Tomcat	**79.9**	0.0	3.8/6.5
CJDBC	26.7	0.1	6.3/7.9
MySQL	**78.1**	0.1	0.5/2.8

TABLE I: Average resource utilization in each tier at WL 8,000. Except Tomcat and MySQL CPU, the other system resources are far from saturation.

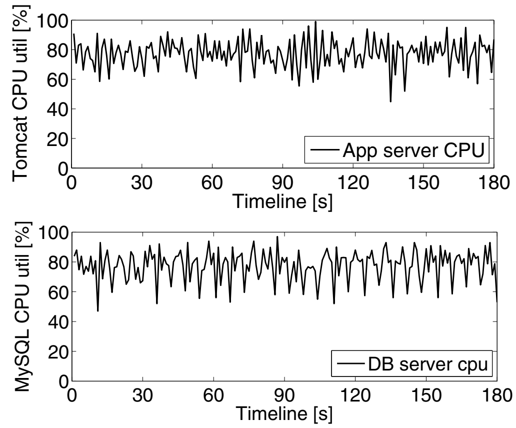


Fig. 3: Tomcat and MySQL CPU utilization at WL 8,000; the average is 79.9% and 78.1% respectively.

the power consumption of CPUs; however, the Dell's BIOS-level SpeedStep control algorithm is unable to adjust the CPU clock speed quickly enough to match the bursty real-time workload; the mismatch between CPU clock speed and real-time workload causes frequent transient bottlenecks in MySQL and leads to wide-range variations of system response time [1].

C. Trace Monitoring Tool

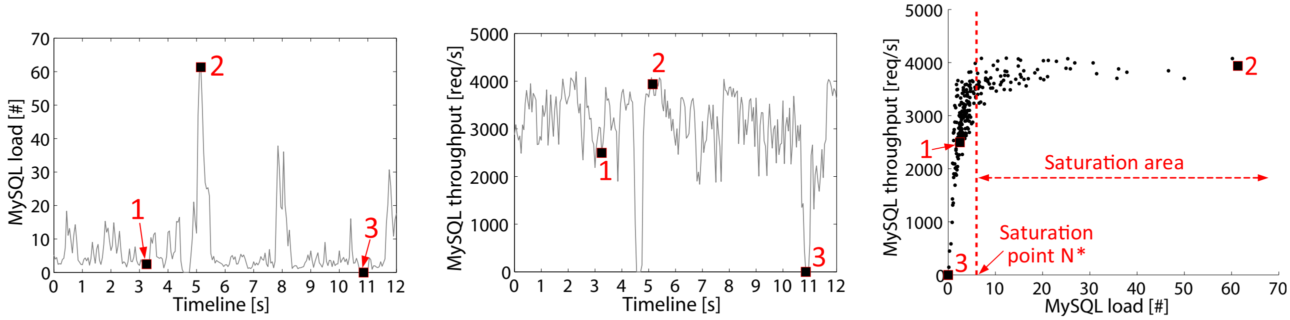
The previous example shows the necessity of detecting transient bottlenecks in the system. Our approach is based on passive network tracing, which can mitigate the monitoring overhead while achieve high precision of detecting transient bottlenecks in the system. In this section, we introduce our monitoring tool, which we use in our transient bottleneck detection method presented in the next section.

We use Fujitsu SysViz [2] to monitor the trace of transaction executions in our experiments. Figure 4 shows an example of such a trace (numbered arrows) of a client transaction execution in a three-tier system. A client transaction services an entire web page requested by a client and may consist of multiple interactions between different tiers. SysViz is able to reconstruct the entire trace of each transaction executed in the system based on the interaction messages (odd-numbered arrows) collected through network taps or network switches which support passive network tracing. Since the timestamp of each interaction message is recorded on one dedicated SysViz machine and independent of clock errors caused by limited accuracy of NTP, the intra-node delay (small boxes with even-numbered arrows) of every request in any server in the system can be precisely recorded.

In fact the end-to-end transaction tracing has been studied for many years and there are mainly two classes of implementations: *annotation-based* and *black box*. Most annotation-based implementations [7] [8] [10] [18] rely on applications or middleware to explicitly associate each interaction message with a global identifier that stitches the messages within a transaction together. Black-box solutions [3] [6] assume there is no additional information other than the interaction messages, and use statistical regression analysis to reconstruct each transaction execution trace. SysViz belongs to the *black-box* class. Experiments in our environment shows that SysViz is able to achieve more than 99% accuracy of transaction trace reconstruction for a 4-tier application even when the application is under a high concurrent workload.

End-to-end transaction tracing in distributed systems has passed the research stage. Reasearch continues on how to best use the information provided by such tracing to diagnose performance issues in the system.

[1]Transient bottlenecks cause instantaneous high concurrency in an n-tier system; once the concurrency exceeds the thread limit in the web tier of the system, new incoming requests will encounter TCP retransmissions, which cause over 3s response times [22].



(a) MySQL load measured at every 50ms time interval in a 12-second time period. Frequent high peaks suggest that MySQL presents short-term congestions from time to time.

(b) MySQL throughput measured at every 50ms time interval in the same 12-second time period as in Figure 5(a).

(c) MySQL load vs. MySQL throughput in the same 12-second time period as in Figure 5(a) and 5(b); MySQL is temporarily congested once the load exceeds $N^{\star}$.

Fig. 5: Performance analysis of MySQL using fine-grained load and throughput at WL 7,000. Figure 5(a) and 5(b) show the MySQL load and throughput measured at the every 50ms time interval. Figure 5(c) is derived from 5(a) and 5(b); each point in Figure 5(c) represents the MySQL load and throughput measured at the same 50ms time interval in the 12-second experimental time period.

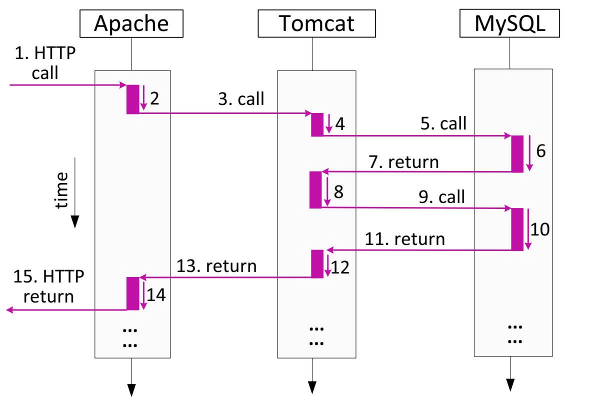


Fig. 4: Illustration of a transaction execution trace captured by SysViz

III. Fine-Grained Load/Throughput Analysis

In this section, we first briefly show how our method detects transient bottlenecks in an n-tier system using a simple example. The details of each part of our method are in the following subsections.

Since a bottleneck in an n-tier system is the place where requests start to congest in the system, a key point of detecting transient bottlenecks is to find component servers that frequently present short-term congestions. To achieve this goal, the first step of our method is to measure a server's load and throughput in continuous fine-grained time intervals. The throughput of a server can be calculated by counting the number of completed requests in the server in a fixed time interval, which can be 50ms, 100ms, or 1s. Load is the average number of concurrent requests over the same time interval [2]. Figure 5(a) and 5(b) shows the MySQL load and throughput measured using a 50ms time interval over a 12-second time period for the 1L/2S/1L/2S configuration case at WL 7,000 (See the case in Figure 2). These two figures show that both the MySQL load and throughput fluctuate significantly, which indicates that MySQL frequently presents short-term congestions.

To diagnose in which time intervals a server presents short-term congestion, we need to correlate the server's load and throughput as shown in Figure 5(c). This figure is derived from Figure 5(a) and Figure 5(b); each point in Figure 5(c) represents the MySQL load and throughput measured at the same 50ms time interval during the 12-second experimental time period (i.e., in total 240 points). This figure shows a clear trend of load/throughput correlation (*main sequence curve*), which is consistent with Denning et al.'s [9] operational analysis result for the relationship between a server's load and throughput. Specifically, a server's throughput increases as the load on the server increases until it reaches the *maximum throughput* TP_{max}, which is determined by the average demand for the bottleneck resource per job according to the Utilization Law. The *congestion point* $N^{\star}$ is the minimum load beyond which the server starts to congest.

Once $N^{\star}$ is determined, we can judge in which time intervals the MySQL tier is congested based on the measured load. For example, Figure 5(c) highlights three points labeled 1, 2, and 3, each of which represents the load/throughput in a time interval that can match back to Figure 5(a) and 5(b). Point 2 shows that the MySQL tier is congested in the corresponding time interval because the load far exceeds $N^{\star}$. Point 3 shows that MySQL is not congested due to the zero load. Point 1 also shows that the MySQL tier is not congested because the corresponding load is less than $N^{\star}$ though it generates high throughput.

After we apply the above analysis to each component server of an n-tier system, we can detect which servers have encountered frequent transient bottlenecks and cause the wide-range response time variations of the system.

[2]Given the precise arrival and departure timestamps of each request for a server monitored through passive network tracing, the load and throughput of the server can be calculated at any given time interval, more details are in Section III-A and III-B

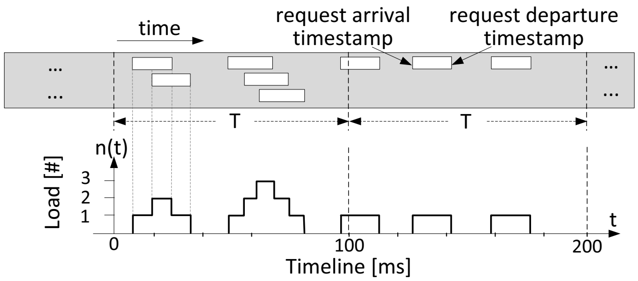


Fig. 6: Load calculation for a server based on the arrival/departure timestamps of requests for the server

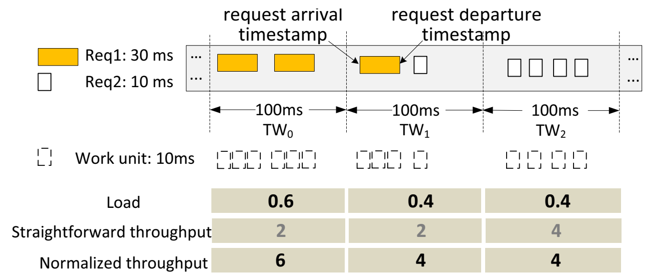


Fig. 7: Load/throughput calculation with mix-class workload

A. Load Calculation

For each server, our direct observables are the arriving (input) requests and departing (output) responses with timestamps generated at microsecond ticks. At each tick, we know how many requests have arrived, but not yet departed. This is the number of concurrent requests being processed by the server. We define the server load as the average number of concurrent requests over a time interval.

Figure 6 shows an example of load calculation for a server in two consecutive 100ms time intervals. The upper part of this figure shows the arrival/departure timestamps of the requests received by the server, which are collected through passive network tracing. Due to the multi-threaded architecture, requests received by a server can be processed concurrently as shown by the interleaved arrival/departure timestamps of different requests. The bottom part of this figure shows the number of concurrent requests being processed by the server at each moment; thus the average in each time interval can be calculated and used as the server load over the time interval.

B. Throughput Calculation

A straightforward approach to calculate throughput of a server in each time interval is to count the number of finished requests during each time interval. This approach is reasonable if a server processes only one class of requests because the same class of requests can be assumed to have a similar amount of demand for the bottleneck resource of the server. Thus, the throughput calculated in each time interval is comparable.

In typical applications including RUBBoS, the workload on a server is mixed with multiple classes of requests each having a different demand for the bottleneck resource of the server. As the time interval length decreases (e.g. 50ms), the request-mix distribution among time intervals becomes significantly different. Thus throughput values calculated (using the straightforward way) in different time intervals are not directly comparable because the requests that comprise the throughput may have different demands for the bottleneck resource.

To calculate the throughput of a server under a mix-class workload, we apply a throughput normalization technique which transforms different classes of completed requests into a certain number of comparable work units. [3]. We define a work unit as the greatest common divisor among the service times from different classes of requests. Requests with a longer service time can transform into a greater number of work units while those with shorter service times only transform into a smaller number. Since the normalized throughput in each time interval only takes into account the transformed work units, throughputs from different time intervals become comparable. This throughput normalization technique is motivated by the request canonicalization and clustering as introduced in Barham et al.'s Magpie [7].

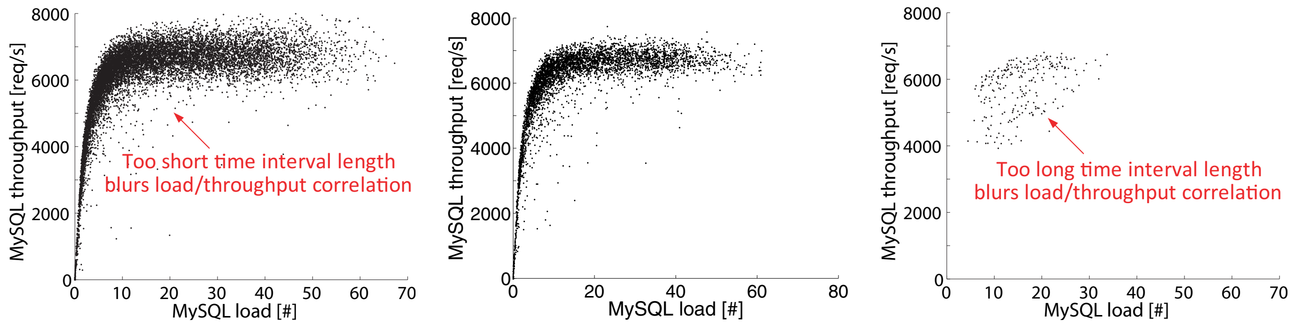
Figure 7 shows an example of the load and throughput calculation under a mix with two classes of requests: Req_1 and Req_2 with service time 30ms and 10ms respectively. The time interval length is 100ms. We set the work unit size as 10ms, so then Req_1 transforms into 3 work units and Req_2 transforms into 1 work unit. Thus, the server processes 6 work units in TW_0 and 4 in both TW_1 and TW_2. We can see that in these three time intervals the normalized throughput has a strong positive correlation with the load, which means the server is not saturated based on Utilization Law. On the other hand, the number of completed requests (the straightforward throughput) has no correlation with the load in this case.

Service time approximation: The service time approximation for each class of requests is obtained using passive network tracing. Figure 4 shows the intra-node delay (small boxes in the figure) of each individual request in each server, which can be treated as the service time if there is no queueing effect. Thus, service time approximation for each class of requests can be conducted online when the production system is under low workload in order to mask out the queuing effects inside a server [20]. Since the service time of each class of requests may drift over time (e.g., due to changes in the data selectivity) in real applications, such service time approximations have to be recomputed accordingly.

C. Congestion Point $N^\star$ Determination

In our method $N^\star$ is used to classify a server's performance state in each time interval; however, the $N^\star$ of a server is not known a priori because the value depends on many factors

[3]For mix-class workload, we assume the demand for the bottleneck resource of a server is proportional to the service time of a request. This assumption is reasonable if a mix-class workload is one specific resource intensive in a server (e.g., CPU). Then the service time can be approximated as CPU time.



(a) Monitoring time interval length 20ms (b) Monitoring time interval length 50ms (c) Monitoring time interval length 1s

Fig. 8: The impact of time interval length on load/throughput correlation analysis for MySQL at WL 14,000. Subfigure (a) (b), and (c) are derived from the same 3-minute experimental data; thus there are 9,000 points with 20ms time interval, 3,600 points with 50ms time interval, and 180 points with 1s time interval.

such as the server's hardware/software configuration and also the workload characteristics [23].

In practice we use a simple statistical intervention analysis [13] to approximate $N^{\star}$, where the main idea of this analysis is to find the minimum load ($N^{\star}$) beyond which the increments of throughput becomes negligible with further increment of load. Suppose the load in a server varies between $[N_{min}, N_{max}]$; then we divide $[N_{min}, N_{max}]$ into k even intervals (e.g., $k = 100$) and calculate the average throughput in each load interval based on the load/throughput samples we collected during the experimental period. Each load interval and the corresponding average throughput is recorded as $\{\langle \overline{ld}_1, \overline{tp}_1\rangle, \langle \overline{ld}_2, \overline{tp}_2\rangle, ..., \langle \overline{ld}_k, \overline{tp}_k\rangle\}$, where $\overline{ld}_1 < \overline{ld}_2 < ... < \overline{ld}_k$. Then the slope δ_i between every two consecutive load intervals can be calculated as Equation 1:

$$\delta_i = \begin{cases} \overline{tp}_1 / \ \overline{ld}_1 & : i = 1 \\ \dfrac{\overline{tp}_i - \overline{tp}_{i-1}}{\overline{ld}_i - \overline{ld}_{i-1}} & : 1 < i \leq k \end{cases} \tag{1}$$

$$tol \leq \overline{\delta} - t_{(0.95, n_0 - 1)} * s.d.\{\delta\} \tag{2}$$

δ_i should be nearly constant (e.g., δ_0) when the server is not saturated and starts to lose stability once the load exceeds $N^{\star}$. The right side of Equation 2 shows a simple heuristic approximation for the lower bound of a ninety percent confidence interval of the sequence $\{\delta_1, \delta_2, ..., \delta_{n_0}\}$ [4], where $1 < n_0 \leq k$. We approximate $N^{\star}$ as $\overline{ld}_{n_0}$ when the lower bound of the variation of the sequence $\{\delta_1, \delta_2, ..., \delta_{n_0}\}$ is below the pre-defined threshold tol (e.g., $0.2\delta_0$).

D. Impact of Monitoring Time Interval Length

Both too short and too long a time interval length have side-effects in detecting transient bottlenecks of a server. Though a short time interval length can better capture the transient variation of the load of a server, it decreases the precision of the throughput calculation due to factors such as requests with a lifespan crossing consecutive time intervals or the errors caused by throughput normalization. For example, the service time even for the same class of requests varies in real applications (e.g., data selectivity changes). The average service time for the same class of requests may not be representative during throughput normalization due to too few requests completed in a small time interval. On the other hand, though a longer time interval length can average out the service time variation for the same class of requests, it may lose the ability to capture the short-term congestions of a server.

Figure 8(a), 8(b), and 8(c) show the load/throughput correlation results of MySQL at workload 14,000 with 20ms, 50ms, and 1s time interval length, respectively. Comparing these three figures we can see that too long a time interval length cannot capture the load/throughput variations, thus losing the ability to detect transient bottlenecks (Figure 8(c)); too short a time interval length blurs the shape of the expected main sequence curve due to the increased errors of normalized throughput (Figure 8(a)).

Note a proper time interval length for a server is workload dependent (e.g., depends on the service time variation of each class of requests for the server). In general a proper length should be small enough to capture the short-term congestions of a server. In the evaluation section we choose the time interval length to be 50ms. An automatic way to choose a proper time interval length is part of our future research.

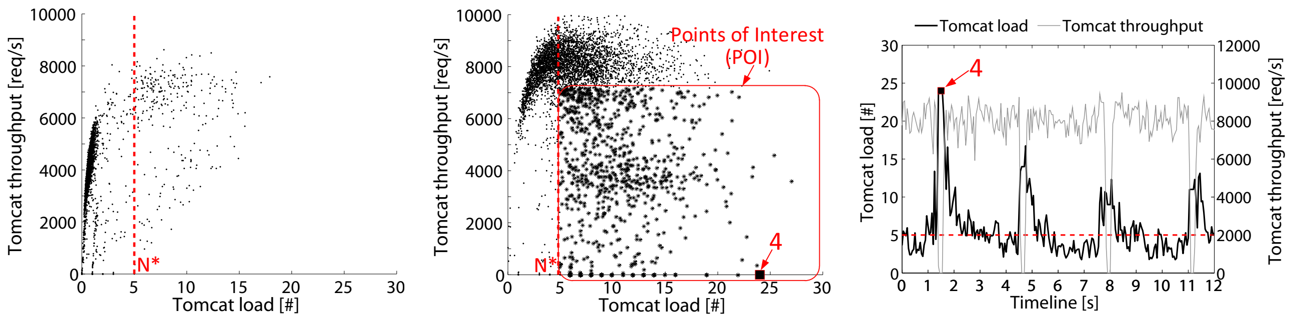
IV. Evaluation

In this section we show two case studies of applying our method to detect transient bottlenecks caused by factors at different levels (e.g., JVM GC at software level and Intel SpeedStep at architecture level). For each case we also show a solution to resolve the transient bottlenecks in the system.

A. Transient bottlenecks caused by JVM GC

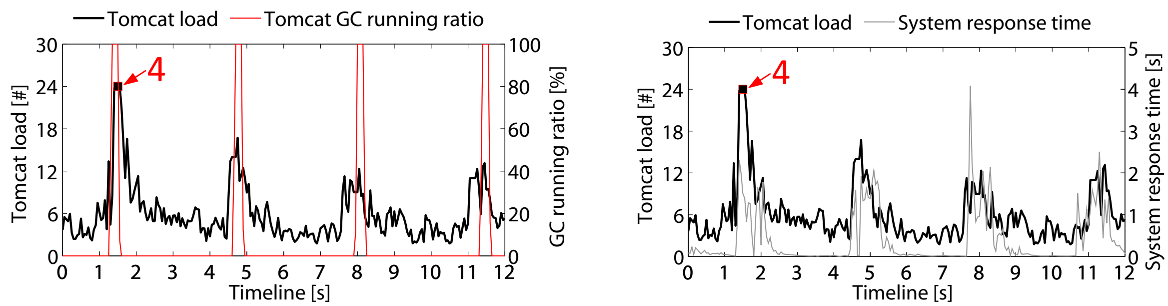
The first case is the transient bottlenecks caused by frequent JVM GCs in Tomcat. In the experiments of this subsection,

[4] $t_{(0.95, n_0-1)}$ is the coefficient for a 90 percent confidence interval when a variable follows a t-distribution; $\overline{\delta} = \frac{1}{n_0}\sum_{i=1}^{n_0} \delta_i$ and $s.d.\{\delta\} = \sqrt{\sum_{i=1}^{n_0}(\delta_i - \overline{\delta})^2}$, which are the mean and the standard deviation of the sequence $\{\delta_1, \delta_2, ..., \delta_{n_0}\}$, respectively.



(a) Tomcat tier at WL 7,000 (b) Tomcat tier at WL 14,000 (c) Tomcat tier at WL 14,000

Fig. 9: Fine-grained load/throughput(50ms) analysis for Tomcat as workload increases. Subfigure 9(b) is derived from Subfigure 9(c), but with 3-minute experimental data. Subfigure 9(b) shows that Tomcat frequently presents short-term congestion at WL 14,000.



(a) Tomcat load vs. Tomcat GC running ratio at WL 14,000; high GC running ratio causes requests to congest in Tomcat.

(b) Tomcat load and system response time in the same time period as in (a); long queue in Tomcat leads to high peak in response time.

Fig. 10: Fine-grained analysis for the large response time fluctuations of the system at WL 14,000. Figure 10(a) shows that frequent JVM GCs cause transient bottlenecks (long queue) in Tomcat, which lead to large response time fluctuations as shown in Figure 10(b).

we use JDK 1.5 in Tomcat which has a synchronous garbage collector; the inefficiency of this garbage collector frequently causes transient bottlenecks in Tomcat and results in significant fluctuations of system response time as we will show in Figure 11(c).

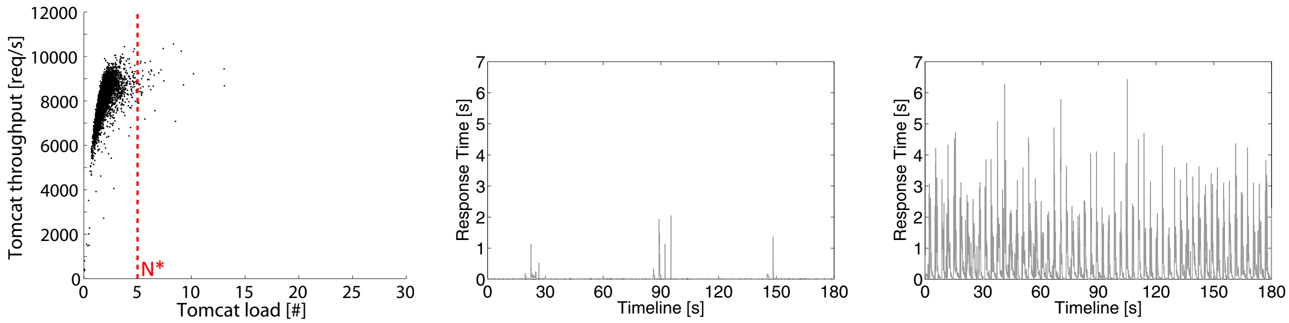
Figure 9 shows the fine-grained load/throughput (50ms) analysis for Tomcat at WL 7,000 and 14,000 with the hardware configuration 1L/2S/1L/2S. Figure 9(a) shows that Tomcat is not bottlenecked in most of the time intervals at WL 7,000 since only a few points are right after $N^{\star}$ derived from Figure 9(b). The interesting figure is Figure 9(b), which shows that at WL 14,000 Tomcat frequently presents transient bottlenecks. In particular, this figure shows there are many points when Tomcat has a high load but low or even zero throughput (POI inside the rectangular area), which contradicts our expectation of the main sequence curve followed by a server's load and throughput.

To illustrate when these POIs happen, Figure 9(c) shows the fine-grained timeline analysis of Tomcat load and throughput in a 10s experimental period at WL 14,000. This figure clearly shows in some time intervals the Tomcat load is high (e.g., the point labeled 4) but the corresponding throughput is zero, which means that many requests are congested in Tomcat but there are no output responses (throughput). In such time intervals, the load/throughput pairs fall into the POI area as shown in Figure 9(b).

Our further analysis shows that the POIs are caused by JVM GCs that frequently stop Tomcat. In this set of experiments, the JVM in Tomcat (JDK 1.5) uses a synchronous garbage collector; it waits during the GC period and only starts processing requests after the GC is finished. To confirm that JVM GCs cause the frequent transient bottlenecks in Tomcat, Figure 10(a) shows the timeline graph which correlates the Java GC running ratio [5] with the Tomcat load. This figure shows that the occurrence of Tomcat JVM GCs have a strong positive correlation with the high peaks of load.

Figure 10(b) shows the correlation between the Tomcat load and the system response time over the same 12-second time period as in Figure 10(a). This figure shows that these two metrics positively correlate with each other, which suggests that the short-term congestions (high load) in Tomcat cause the high peaks of system response time. Figure 10(a) and 10(b) together show that frequent JVM GCs in Tomcat causes frequent short-term congestions in Tomcat, which in turn cause the significant variations on system response time.

[5]Java GC running ratio means the percentage of time spent on Java GC in each monitoring time interval. JVM provides a logging function which records the starting and ending timestamp of every GC activity.



(a) Tomcat tier at WL 14,000 (JDK 1.6 in Tocmat) (b) System response time averaged in every 50ms (JDK 1.6 in Tocmat) (c) System response time averaged in every 50ms (JDK 1.5 in Tocmat)

Fig. 11: Resolving transient bottlenecks by upgrading Tomcat JDK version from 1.5 to 1.6. Figure 11(a) shows that the frequent transient bottlenecks in Tomcat as shown in Figure 9(b) are resolved. Thus, comparing Figure 11(b) and 11(c), the system response time presents much less fluctuations.

B. Solution: upgrade JDK version in Tomcat

Once we detect the frequent transient bottlenecks in Tomcat, we can resolve such bottlenecks by simply scaling-out/up the Tomcat tier since low utilization of Tomcat can reduce the negative impact of JVM GC [22]. Here we illustrate a more economical way to solve the problem by just upgrading the Tomcat JDK version from 1.5 to 1.6, which has more efficient garbage collectors[6]. The experimental configurations are kept the same as before except the Tomcat JDK version.

Figure 11(a) shows the fine-grained load/throughput correlation analysis of Tomcat at workload 14,000 after upgrading the Tomcat JDK version. This figure shows that Tomcat no longer presents frequent transient bottlenecks compared to Figure 9(b). Specifically, the POIs in Figure 9(b) do not appear in Figure 11(a), which means the Tomcat JVM does not have long "freezing" periods after we upgrade the Tomcat JDK.

Figure 11(b) and 11(c) show the average system response time measured at every 50ms time intervals in the 3-minute experimental period before and after we upgrade Tomcat JDK version. These two figures show that the large response time fluctuations disappear after the JDK version upgrade, which shows that the system performance becomes more stable after we resolve the frequent transient bottlenecks in Tomcat.

C. Transient bottlenecks caused by Intel SpeedStep

The second case is the use of Intel SpeedStep technology which unintentionally causes transient bottlenecks, leading to the wide-range response time variations as we showed in Section II-B. Intel SpeedStep allows the clock speed of a CPU to be dynamically adjusted (to different P-states) based on the real-time computing demands on a server in order to achieve a good balance between power usage and server performance; however, we found that the Dell's BIOS-level SpeedStep control algorithm cannot adjust the CPU clock speed quickly enough to match the real-time workload once the workload becomes bursty; the mismatch between CPU clock speed and real-time workload causes frequent transient bottlenecks that lead to the long-tail response time distribution as shown in Figure 2(c).

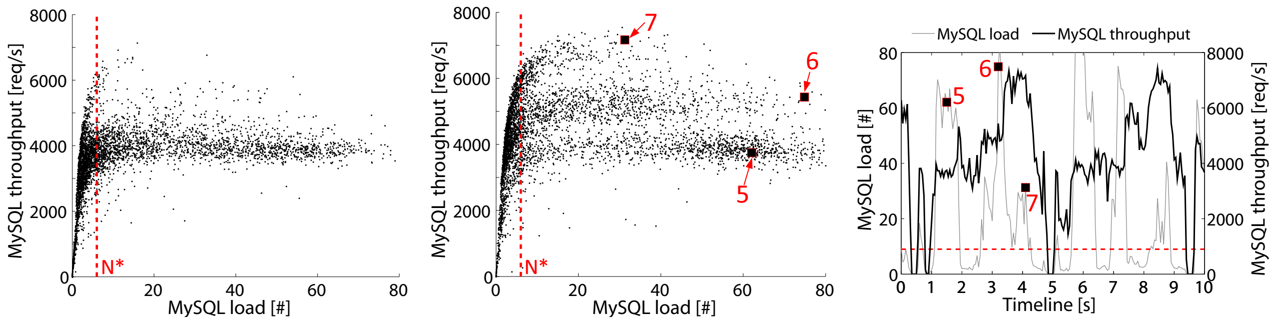
[6]JDK 1.6 uses garbage collection algorithms which support both parallel and concurrent garbage collection while JDK 1.5 by default uses a serial, stop-the-world collector.

P-state	P0	P1	P4	P5	P8
CPU clock [MHz]	2261	2128	1729	1596	1197

TABLE II: Partial P-states supported by the Xeon CPU of our machines

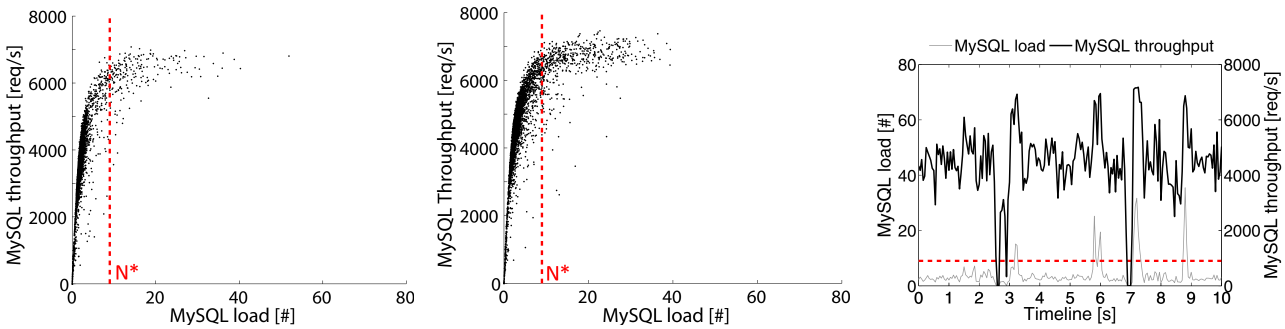
We enable the Intel SpeedStep support for MySQL in the BIOS settings to illustrate the mismatch problem. Table II shows a part of the P-states supported by our experimental machine CPU. This table shows that the CPU clock speed of the lowest P-state (P8) is nearly half of the highest P-state (P0). The experiments described here still keep the same 1L/2S/1L/2S configuration as in the previous sections with the only difference being the change in BIOS settings. We note that in all of the previous experiments, we disable the SpeedStep support in the BIOS settings of all our machines to simplify our analyses.

Figure 12 shows the fine-grained load/throughput analysis for MySQL at WL 8,000 and 10,000. As illustrated in Figure 2(c), the system already presents wide-range response time variations at WL 8,000. Such variations are caused by the frequent transient bottlenecks in MySQL as shown in Figure 12(a). The interesting observation in Figure 12(a) is that though MySQL presents one main throughput trend (about 3700 req/s) when the load exceeds $N^{\star}$, there are many points above the main throughput trend, which contradicts our expectation of the shape of the main sequence curve. The comparison between Figure 12(a) and 12(b) reveals the cause. Since workload 8000 is relatively low, MySQL prefers to stay in P8-state in order to save power; however, MySQL is not responsive enough to scale-up to higher P-states to handle peak request rates from the upstream tiers in the system and thus presents short-term congestions as shown in Figure 12(a). As workload increases to 10,000, Figure 12(b) shows that MySQL throughput presents three clear trends (about 3700 req/s, 5000



(a) MySQL tier at WL 8,000 (b) MySQL tier at WL 10,000 (c) MySQL tier at WL 10,000

Fig. 12: Fine-grained load/throughput(50ms) analysis for MySQL when CPU SpeedStep is enabled in MySQL. Figure 12(b) is derived from Figure 12(c), with 3-minute experimental data. Figure 12(a) shows one throughput trend when MySQL is temporarily bottlenecked, which indicates that MySQL chooses the lowest CPU clock speed when the workload is low. Figure 12(b) shows three throughput trends, which indicates that MySQL alternates among three CPU frequencies supported by Intel CPU SpeedStep as workload increases to 10,000.



(a) MySQL at WL 8,000 (b) MySQL at WL 10,000 (c) MySQL at WL 10,000

Fig. 13: Fine-grained load/throughput(50ms) analysis for MySQL when CPU SpeedStep is disabled in MySQL. Since MySQL always chooses to stay in the maximum CPU clock speed, the frequency of transient bottlenecks is significantly reduced by comparing Figure 13(a) and 13(b) with Figure 12(a) and 12(b).

req/s, and 7000 req/s) when the corresponding load exceeds $N^{\star}$, which indicates that MySQL CPU alternates among three different P-states. For instance, the points labeled 5, 6, 7 show three time intervals when MySQL is temporarily congested but produces different throughputs. Point 5 indicates that MySQL stays in the lowest P8-state, point 6 indicates that MySQL stays in either P4- or P5-state, and point 7 indicates that MySQL stays in P0-state.

To illustrate when the mismatch of CPU clock speed and the real-time load on MySQL happens, Figure 12(c) shows the fine-grained MySQL load and throughput in a 10s experimental period at WL 10,000. The points labeled 5, 6, 7 correspond to the highlighted points in Figure 12(b), and show that in these three time intervals MySQL is temporarily congested but generates different throughputs. This figure illustrates the time lag of MySQL scaling-up to higher P-states, which causes frequent transient bottlenecks in MySQL.

D. Solution: Disable Intel SpeedStep in BIOS

Once detecting the frequent transient bottlenecks caused by the mismatch between CPU clock speed and bursty workload, we can resolve such bottlenecks by disabling the SpeedStep support in MySQL and let MySQL always stay in P0-state.

Figure 13 shows the fine-grained load/throughput analysis for MySQL at WL 8,000 and 10,000 after we disable the SpeedStep support in MySQL. Figure 13(a), 13(b) and 13(c) match back to Figure 12(a), 12(b) and 12(c), respectively. Since MySQL CPU always stays in P0-state, both Figure 13(a) and 13(b) show that MySQL only presents one throughput trend when load exceeds $N^{\star}$. More importantly, Figure 13(a) and 13(b) show that MySQL presents much less transient bottlenecks compared to the case shown in Figure 12(a) and 12(b) at WL 8,000 and 10,000. Figure 13(c) also shows that MySQL load is below $N^{\star}$ most of the time at WL 10,000, which suggests more stable performance of the system compared to Figure 12(c).

Further reduction of the transient bottlenecks in MySQL needs to either scale-out the MySQL tier (add more nodes to the MySQL tier) or scale-up MySQL (switch to a more powerful CPU).

V. Related Work

Techniques based on end-to-end request-flow tracing have been proposed in previous research for performance anomaly diagnosis. Magpie [7] and Pinpoint [8] focus on identifying anomalous requests that either have long response times or mutations of request-flow path by finding rare paths that differ greatly from others. Pip [16] identifies anomalous requests by comparing request-flows from actual behaviors and developer-expected behaviors. Spectroscope [17] proposes a similar monitoring infrastructure as Pip, but instead of comparing request-flows between actual behaviors and developer-expected behaviors, it compares request-flows between "problem" periods and "non-problem" periods. Though detecting anomalous requests gives very useful hints to diagnose performance problem, they may fail to diagnose the root cause of anomalous requests in an n-tier system. A "anomalous" request may be slow not because of its own behavior, but because other requests were queued ahead of it [18], [22].

Analytical models have been proposed for bottleneck detection and performance prediction of n-tier systems. Urgaonkar [21] present a flexible queueing model for an n-tier application that determines how much resources to allocate to each tier of the application for the target system response time; however, this model is based on Mean Value Analysis (MVA), which has difficulties dealing with wide-range response time variations caused by bursty workloads and transient bottlenecks in the system. Mi et al. [14] propose a more sophisticated analytical model that predicts system performance based on bursty workloads. One challenge of this work is to precisely map the bursty characteristics of a workload to the queueing model with multiple service rates for each queue in the system. As shown in this paper, without fine-grained monitoring (sub-second level) granularity, the bursty characteristics of a workload and the potential transient bottlenecks as a result can be largely masked.

Software mis-configuration and failure detection of distributed system have been studied in [4], [5], [15]. Attariyan et al. [4], [5] present a tool that locates the root cause of configuration errors by applying dynamic information flow analysis within a process (mainly) during runtime. Oliveira et al. [15] propose a mistake-aware management framework for protecting n-tier systems against operator mistakes by using the previous correct operations. All these works differ from our work in that they focus on faulty/anomalous behavior of system components rather than the performance problem.

VI. Conclusion

We observed that the performance of an n-tier system may degrade significantly due to transient bottlenecks in component servers in the system. We proposed a novel bottleneck detection method to detect these transient bottlenecks (Section III), where the effectiveness of our approach is validated through the two case studies in Section IV. We found that transient bottlenecks can be caused by various factors at different levels of an n-tier application; for instance, JVM GC at the software level (Section IV-A) and Intel SpeedStep at the architecture level (Section IV-C). Solving these transient bottlenecks leads to significant performance improvements (Section IV-B and IV-D). More generally, our work is an important contribution towards scaling complex n-tier applications under elastic workloads in cloud environments.

VII. Acknowledgement

This research has been partially funded by National Science Foundation by IUCRC/FRP (1127904) , CISE/CNS (1138666), RAPID (1138666), CISE/CRI (0855180), NetSE (0905493) programs, and gifts, grants, or contracts from DARPA/I2O, Singapore Government, Fujitsu Labs, Wipro Applied Research, and Georgia Tech Foundation through the John P. Imlay, Jr. Chair endowment. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation or other funding agencies and companies mentioned above.

References

[1] *RUBBoS: Bulletin board benchmark*. "http://jmob.ow2.org/rubbos.html", 2004.

[2] *Fujitsu SysViz: Visualization in the Design and Operation of Efficient Data Centers*. "http://globalsp.ts.fujitsu.com/dmsp/Publications/public/E4_Schnelling_Visualization%20in%20the%20Design%20and%20Operation%20of%20Efficient%20Data%20Centers.pdf", 2010.

[3] M. K. Aguilera, J. C. Mogul, J. L. Wiener, P. Reynolds, and A. Muthitacharoen. Performance debugging for distributed systems of black boxes. In *SOSP '03*.

[4] M. Attariyan, M. Chow, and J. Flinn. X-ray: Automating root-cause diagnosis of performance anomalies in production software. In *OSDI'12*.

[5] M. Attariyan and J. Flinn. Automating configuration troubleshooting with dynamic information flow analysis. In *OSDI'10*.

[6] P. Bahl, R. Chandra, A. Greenberg, S. Kandula, D. A. Maltz, and M. Zhang. Towards highly reliable enterprise network services via inference of multi-level dependencies. In *SIGCOMM'07*.

[7] P. Barham, A. Donnelly, R. Isaacs, and R. Mortier. Using magpie for request extraction and workload modelling. In *OSDI'04*.

[8] M. Chen, E. Kiciman, E. Fratkin, A. Fox, and E. Brewer. Pinpoint: problem determination in large, dynamic internet services. In *DSN'02*.

[9] P. J. Denning and J. P. Buzen. The operational analysis of queueing network models. *ACM Comput. Surv.*, 1978.

[10] R. Fonseca, G. Porter, R. H. Katz, S. Shenker, and I. Stoica. X-trace: a pervasive network tracing framework. In *NSDI'07*.

[11] E. C. Julie, J. Marguerite, and W. Zwaenepoel. *C-JDBC: Flexible Database Clustering Middleware*. 2004.

[12] R. Kohavi and R. Longbotham. Online experiments: Lessons learned. IEEE Computer'07.

[13] S. Malkowski, M. Hedwig, J. Parekh, and C. Pu. Bottleneck detection using statistical intervention analysis. In *DSOM'07*.

[14] N. Mi, G. Casale, L. Cherkasova, and E. Smirni. Burstiness in multi-tier applications: symptoms, causes, and new models. In *Middleware '08*.

[15] F. Oliveira, A. Tjang, R. Bianchini, R. P. Martin, and T. D. Nguyen. Barricade: defending systems against operator mistakes. EuroSys '10.

[16] P. Reynolds, C. Killian, J. L. Wiener, J. C. Mogul, M. A. Shah, and A. Vahdat. Pip: Detecting the unexpected in distributed systems. In *NSDI'06*.

[17] R. Sambasivan, A. Zheng, M. De Rosa, E. Krevat, S. Whitman, M. Stroucken, W. Wang, L. Xu, and G. Ganger. Diagnosing performance changes by comparing request flows. In *NSDI'10*.

[18] B. Sigelman, L. Barroso, M. Burrows, P. Stephenson, M. Plakal, D. Beaver, S. Jaspan, and C. Shanbhag. Dapper, a large-scale distributed systems tracing infrastructure. In *Google Technical report'10*.

[19] B. Snyder. Server virtualization has stalled, despite the hype. *InfoWorld*, 2010.

[20] B. Urgaonkar, G. Pacifici, P. Shenoy, M. Spreitzer, and A. Tantawi. An analytical model for multi-tier internet services and its applications. In *SIGMETRICS'05*.

[21] B. Urgaonkar, P. Shenoy, A. Chandra, and P. Goyal. Dynamic provisioning of multi-tier internet applications. In *ICAC'05*.

[22] Q. Wang, Y. Kanemasa, M. Kawaba, and C. Pu. When average is not average: Large response time fluctuations in n-tier systems. In *ICAC'12*.

[23] Q. Wang, S. Malkowski, Y. Kanemasa, D. Jayasinghe, P. Xiong, M. Kawaba, L. Harada, and C. Pu. The impact of soft resource allocation on n-tier application scalability. In *IPDPS'11*.