2013 IEEE Sixth International Conference on Cloud Computing

An Experimental Study of Rapidly Alternating Bottlenecks in n-Tier Applications

Qingyang Wang [1], Yasuhiko Kanemasa [2], Jack Li [1], Deepal Jayasinghe [1]
Toshihiro Shimizu [2], Masazumi Matsubara [2], Motoyuki Kawaba [3], Calton Pu [1]

[1]*College of Computing, Georgia Institute of Technology*
[2]*Cloud Computing Research Center, FUJITSU LABORATORIES LTD.*
[3]*IT Systems Laboratories, 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 bottlenecks alternate rapidly between component servers. Such rapidly alternating bottlenecks arise naturally and often from resource dependencies in an n-tier system and bursty workloads. These rapidly alternating bottlenecks are difficult to detect because the saturation in each participating server may have a very short lifespan (e.g., milliseconds) compared to current system monitoring tools and practices with sampling at intervals of seconds or minutes. Using passive network tracing at fine-granularity (e.g., aggregate at every 50ms), we are able to correlate throughput (i.e., request service rate) and load (i.e., number of concurrent requests) in each server of an n-tier system. Our experimental results show conclusive evidence of rapidly alternating bottlenecks caused by system software (JVM garbage collection) and middleware (VM collocation).**

I. Introduction

Web-facing enterprise applications such as electronic commerce are not embarrassingly parallel (e.g., web indexing and data analytics). They are typically implemented using an n-tier architecture with web server, application server, and database server tiers. Such n-tier applications have implicit dependencies among their components, which create *alternating bottlenecks* [4], [6], [12], [14]. These alternating bottlenecks are both interesting and challenging. They are interesting because they cause the entire n-tier system to reach its performance limit (i.e., flat throughput) even though all system resources are measurably below 100% utilization. They are challenging because classic queuing models that assume independent jobs predict single resource saturation bottlenecks, so they are inapplicable to alternating bottlenecks.

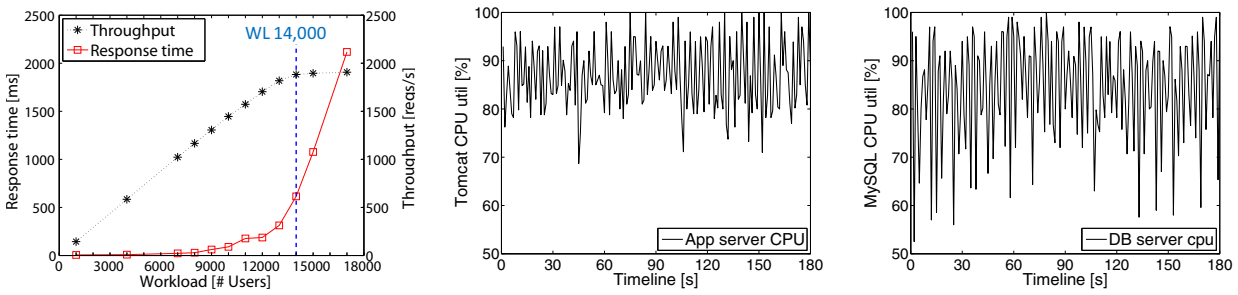
The main hypothesis of this paper is that (contrary to previously common belief) alternating bottlenecks occur naturally in real application scenarios and they can be found by experimental measurements using appropriate tools. Alternating bottlenecks constitute an important problem because there is lingering skepticism about their prevalence (and even existence) in the real world, despite early theoretical predictions [4], [6], [14]. In the past, observed evidence of alternating bottlenecks was rare and it was not easy to reproduce them reliably in experiments. We report consistent experimental results which suggest that alternating bottlenecks may be far more common than previously believed. The perception of rarity is simply due to many alternating bottlenecks being short-lived (on the order of tens of milliseconds). Consequently, these interesting phenomena have been (and still are) completely invisible to normal monitoring tools that sample at time intervals measured in seconds or minutes.

The main contribution of the paper is an unequivocal confirmation of our hypothesis through reproducible experimental observation of two rapidly alternating bottlenecks when running the standard n-tier RUBBoS benchmark [1]. Specifically, we found that bottlenecks alternate between the Tomcat tier and the MySQL tier at time interval of tens of milliseconds. Our study further shows that alternating bottlenecks can be caused by factors at the software level (e.g., JVM garbage collection (GC), see Section III-B) and middleware level (e.g., VM collocation, see Section III-C). Despite its relatively short duration, the impact of this alternating bottleneck becomes significant when the frequency and intensity of the alternating pattern increase.

The detection of alternating bottlenecks required a novel method that differs from traditional bottleneck detection in two main aspects. First, since alternating bottlenecks may arise without any single resource saturation, our method is completely independent of any single resource saturation measurements. Concretely, Section II-B shows an example in which the throughput of a four-tier system stops increasing even though the highest resource utilization in the system (MySQL CPU) is only 86.9%. Second, our method works at an unprecedented fine time granularity (milliseconds), which is more precise than normal sampling tools (e.g., dstat consumes 12% of CPU at 20ms intervals). Our method uses passive network packet tracing, which captures the arrival and departure time of each request of each server at microsecond granularity with negligible impact on the servers. By correlating the load and throughput of each server at millisecond granularity, our method is able to find short-lived alternating bottlenecks (lifetime of tens of milliseconds) that have been invisible to state-of-the-art sampling tools.

The rest of the paper is organized as follows. Section II

978-0-7695-5028-2/13 $26.00 © 2013 IEEE
DOI 10.1109/CLOUD.2013.32



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

(b) Tomcat CPU utilization at WL 14,000; the average is 86.9%.

(c) MySQL CPU utilization at WL 14,000; the average is 84.3%.

Fig. 1: A non-single bottleneck case. The system achieves the highest throughput at WL 14,000 while no hardware resources are saturated.

Server/Resource	CPU util. (%)	Disk I/O (%)	Network receive/send (MB/s)
Apache	45.9	0.5	23.8/39.9
Tomcat	**86.9**	0.3	7.6/13.1
CJDBC	36.23	0.2	11.2/14.3
MySQL	**84.3**	0.4	0.8/4.6

TABLE I: Average resource utilization in each tier at WL 14,000. Except Tomcat/MySQL CPU, other resources are far from saturation.

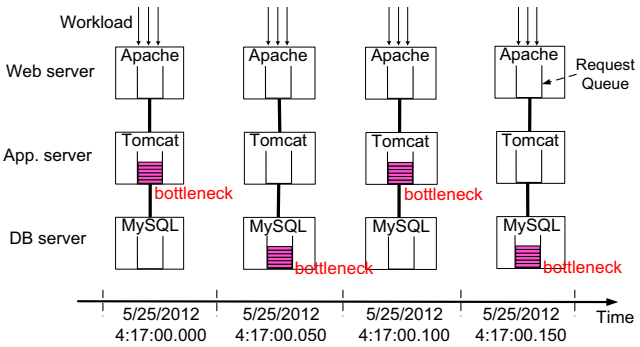


Fig. 2: A rapidly alternating bottleneck in a 3-tier web application.

introduces various kinds of bottlenecks. Section III shows our experimental observations of rapidly alternating bottlenecks of an n-tier application. Section IV shows solutions to resolve the observed rapidly alternating bottlenecks. Section V summarizes the related work and Section VI concludes the paper.

II. Various Kinds of Bottlenecks

A. Single Bottlenecks

A system bottleneck in an n-tier system is the place where requests start to queue (or congest) and throughput is limited in the system. Classic queuing models assume independent jobs and predict single resource bottleneck in an n-tier system, in which the system achieves the maximum throughput when the single bottleneck resource is 100% utilized. Due to its simplicity and intuitiveness, classic queuing models have provided the foundation for system administrators to manage and predict system performance [18], [22], [11]. Despite their popularity, classic queuing models are based on assumptions (e.g., independent jobs among component servers in a system) that do not necessarily hold in an n-tier system in practice.

B. Saturation without Single Bottlenecks

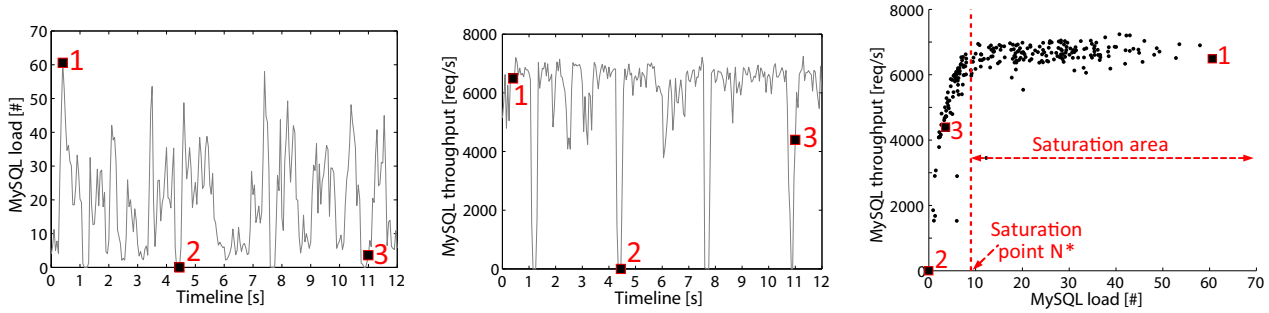
We use an example where an n-tier system is saturated while no hardware resources are fully utilized. The example was derived from a three-minute experiment of RUBBoS [1] benchmark running on a four-tier configuration 1L/2S/1L/2S (see Figure 12(c)), which means one web server (apache), two application servers (tomcat), one database clustering middleware (C-JDBC), and two database servers (MySQL). The details of the experimental setup is in Appendix A.

Figure 1(a) shows the system works well from a workload of 1,000 concurrent users to 13,000. At 14,000, the average response time increases significantly and the throughput reaches a maximum. The interesting observation is that the saturated system does not have any single resource bottleneck. Since CPU is the critical resource in a browse-only workload, we show the timeline graphs (with one second granularity) of CPU utilization. During the execution of the WL 14,000, both Tomcat (Figure 1(b)) and MySQL (Figure 1(c)) show less than full CPU utilization, with an average of 86.9% (Tomcat) and 84.3% (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 unable to identify the system bottleneck, since there is no single saturated resource. Later in Section III-B we explain the bottleneck alternates rapidly between MySQL and Tomcat. During normal processing, MySQL CPU is the primary system bottleneck, being fully utilized for processing bursty requests from Tomcat. However, the JVM GC in Tomcat freezes the server and consumes the server CPU (at the granularity of milliseconds). Thus the Tomcat becomes the system bottleneck during JVM GC periods. In either case, the system throughput is limited.

C. Rapidly Alternating Bottlenecks

Alternating bottleneck describes a phenomenon that the bottleneck alternates among multiple system resources while



(a) MySQL load average at each 50ms time interval in a 12-second period. Large fluctuation suggests MySQL frequently becomes the bottleneck.

(b) MySQL throughput average at each 50ms time interval in the same 12-second period as in Figure 4(a).

(c) MySQL load vs. throughput in the same 12-second period as in Figure 4(a), 4(b); MySQL is temporarily saturated once the load exceeds N^*.

Fig. 4: Performance analysis of MySQL using fine-grained load and throughput at WL 14,000. Figure 4(a) and 4(b) show the MySQL load and throughput measured at the every 50ms time interval. Figure 4(c) is derived from 4(a) and 4(b); each point in Figure 4(c) represents the MySQL load and throughput measured at the same 50ms time interval in the 12-second experimental time period.

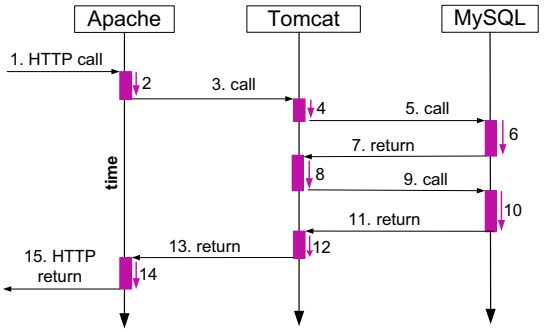


Fig. 3: A transaction execution trace captured by SysViz.

at any moment one system resource becomes the main bottleneck. Alternating bottleneck arises due to the implicit dependencies among servers in an n-tier system. For example, requests that originate from a client arrive at the web server, which distributes them among the application servers, which in turn ask the database servers to carry out the query. The dependencies among the servers are in the long invocation chain of transaction processing in the system and maintained by soft resources (threads, TCP connections [21]). The dependencies, combined with many system events including database locks, JVM GC, memory contention, and/or characteristics of the scheduling algorithms and many others, may cause requests to congest in different servers at different time periods. Figure 2 illustrates a rapidly alternating bottleneck case in a 3-tier web application. In this case, Tomcat and MySQL face congestions in a rapidly alternating pattern.

III. Experimental Observation of Rapidly Alternating Bottlenecks

In this section, we first explain our fine-grained analysis to detect rapidly alternating bottlenecks. Then we show two case studies of applying our method to detect rapidly alternating bottlenecks of an n-tier application, which are caused by JVM GC and VM collocation respectively.

A. Fine-Grained Load/Throughput Analysis

1) Trace Monitoring Tool: Our fine-grained analysis is based on the tracing of client transaction executions of an n-tier system. We first briefly explain our tool (Fujitsu SysViz [2]) for the tracing of transaction executions.

Figure 3 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 can reconstruct the entire trace of each transaction executed in the system based on the traffic messages (odd-numbered arrows) collected through a network switch which supports passive network tracing. Thus, the arrival/departure timestamps of each request (small boxes with even-numbered arrows) for any server can be recorded.

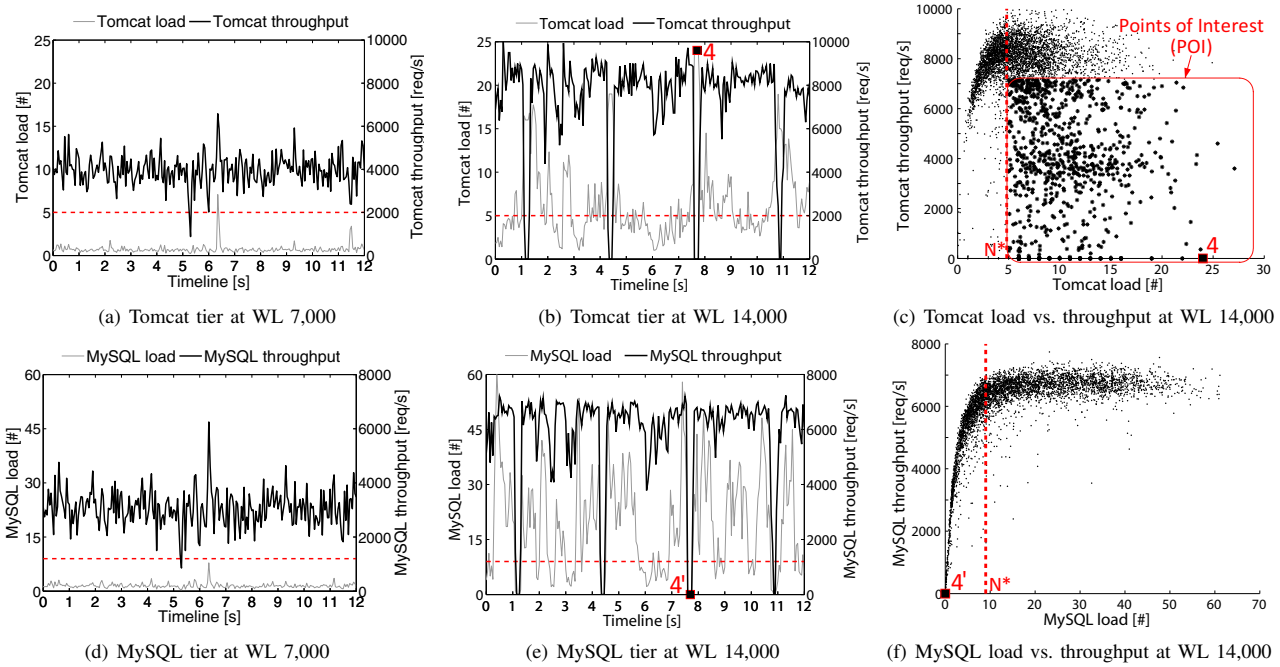
In fact transaction tracing technology has been studied intensively in previous research [5], [8], [16], [17]; how to utilize the captured tracing information to diagnose system performance problem is the ongoing research trend.

2) Detecting Rapidly Alternating Bottleneck: Since each participating server in a rapidly alternating bottleneck case only presents short-term saturations, a key point of detecting the rapidly alternating bottleneck is to find the participating short-term saturated servers. Instead of monitoring hardware resource utilizations, our approach measures a server 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 [1]. Both these two metrics for each server in the system can be easily derived from the trace captured by SysViz [2].

Figure 4(a) shows the MySQL load average at every 50ms time interval over a 12-second time period at WL 14,000 (See

[1] At each time tick, we know how many requests for a server have arrived, but not yet departed. This is the number of concurrent requests being processed by the server. Concurrent requests can also be thought as "queued" requests.

[2] The detailed fine-grained load/throughput calculation can be found in [20].



(a) Tomcat tier at WL 7,000

(b) Tomcat tier at WL 14,000

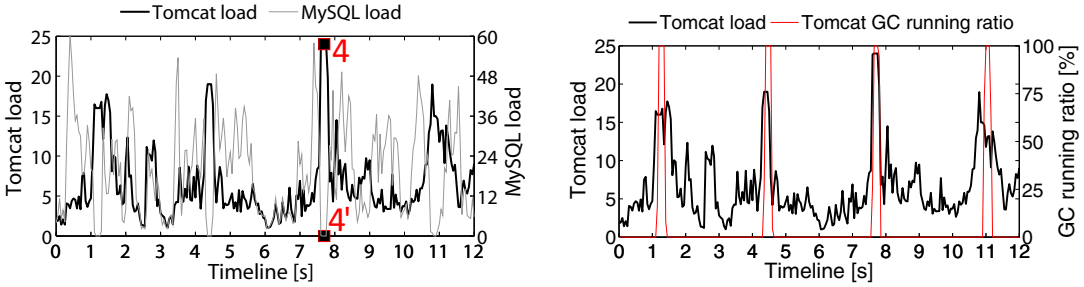
(c) Tomcat load vs. throughput at WL 14,000

(d) MySQL tier at WL 7,000

(e) MySQL tier at WL 14,000

(f) MySQL load vs. throughput at WL 14,000

Fig. 5: Fine-grained load/throughput(50ms) analysis for Tomcat and MySQL. Figure 5(c), 5(f) are derived from Figure 5(b), 5(e) respectively, with 3-minute experimental data. Figure 5(b), 5(e) show that both Tomcat and MySQL frequently present short-term saturations at WL 14,000.



(a) Correlation between Tomcat load and MySQL load; negative correlation suggests rapidly alternating bottleneck.

(b) Tomcat load and GC running ratio in the same time period as in (a); high GC running ratio causes long queue in Tomcat.

Fig. 6: Correlation analysis of the rapidly alternating bottleneck between Tomcat and MySQL at WL 14,000. Figure 6(b) shows that Tomcat temporarily becomes the bottleneck due to frequent JVM GC.

the case in Figure 1). This figure shows frequent high load (large number of queued requests) in MySQL, which suggests MySQL frequently presents short-term saturation. Figure 4(b) shows the fine-grained MySQL throughput over the same 12-second time period as in Figure 4(a). This figure shows that in some time intervals MySQL even produces zero throughput, which suggests MySQL is frequently under-utilized.

To precisely diagnose in which time intervals a server is temporarily saturated, we correlate the server's load and throughput as shown in Figure 4(c). This figure is derived from Figure 4(a) and 4(b); each point in Figure 4(c) represents the MySQL load/throughput measured at the same 50ms time interval during the 12-second time period. This figure shows the clear trend of load/throughput correlation (*main sequence curve*), which is consistent with Denning et al.'s [7] 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 TP_{max} [3], the *maximum throughput*. The *saturation point* $N^{\star}$ is the minimum load beyond which the server starts to saturate.

Given the $N^{\star}$ of a server, we can decide the time intervals in which the server is saturated based on the measured load. For example, Figure 4(c) highlights three points labeled 1, 2, and 3; each point represents the load/throughput in a time interval that can match back to Figure 4(a) and 4(b). Point 1 shows that MySQL is saturated during the time interval because the high load far exceeds $N^{\star}$. Point 2 shows that MySQL is not saturated due to the zero load and throughput. Point 3 also shows MySQL is not saturated because the corresponding load

[3]Due to the Utilization Law, the maximum throughput TP_{max} of a server is fixed by the bottleneck resource in terms of $1/d$, where d is the service demand for the bottleneck resource per job [7].

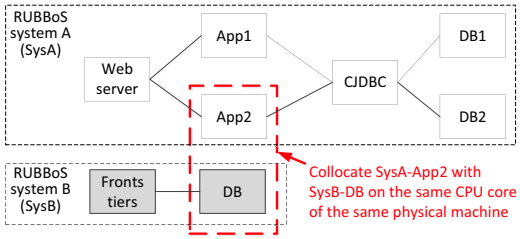


Fig. 7: Collocation strategy between *SysA* and *SysB*; *SysA-App2* is collocated with *SysB-DB*.

TABLE II: Workload of *SysA* and *SysB* during collocation. *SysA* is at constant stable WL 14,000 with $I = 1$ and *SysB* is in constant workload but with different burstiness levels.

#	SysA-App2			SysB-DB		
	WL (# users)	Burstiness level	CPU (%)	WL (# users)	Burstiness level	CPU (%)
1	14,000	I=1	**74.1**	0		0
2	14,000	I=1	**74.9**	400	I=1	**10.2**
3	14,000	I=1	**74.7**	400	I=100	**10.6**
4	14,000	I=1	**75.5**	400	I=200	**10.5**
5	14,000	I=1	**75.2**	400	I=400	**10.8**

is less than $N^\star$ though it generates high throughput.

After we detect all the short-term saturated servers, the next step is to analyze whether the short-term saturation of each participating server occurs in an alternating pattern. We will illustrate this point in the following case studies.

B. Rapidly Alternating Bottleneck Caused by JVM GC

In this section we explain the rapidly alternating bottleneck mentioned in Section II-B. In that example, the poor system performance is caused by the frequent short-term saturations of both Tomcat and MySQL. Our further correlation analysis shows that the frequent JVM GC in Tomcat cause the bottleneck to alternate between Tomcat and MySQL.

Figure 5 shows the fine-grained load/throughput(50ms) analysis for Tomcat and MySQL at WL 7,000 and 14,000 with the same system configuration as in Section II-B. Figure 5(a) and 5(d) show that both Tomcat and MySQL are not saturated at WL 7,000 since the load of each tier is below the corresponding $N^\star$, which is derived from Figure 5(c) and Figure 5(f) respectively.

The interesting figures are Figure 5(b) and 5(e), which show that at WL 14,000 both the Tomcat tier and the MySQL tier frequently present short-term saturations. Specially, Figure 5(b) shows that 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 queued in Tomcat but no output responses (throughput). Figure 5(c), which is derived from Figure 5(b) but based on the 3-minute runtime experiments, shows that there are many time intervals when Tomcat has a high load but low or even zero throughput (POI inside the rectangular area). Since Tomcat is the upstream tier of MySQL, the output responses of Tomcat feeds the input requests of MySQL; thus having fewer output responses from Tomcat means there will be fewer input requests sent

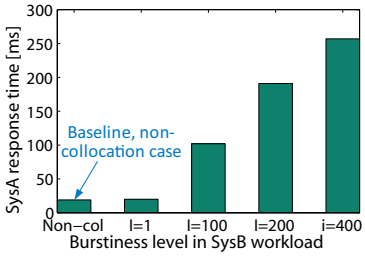


Fig. 8: *SysA* response time (at WL 14,000) when collocated with SysB (at WL 400 but with increased burstiness level).

to MySQL, which leads to the under-utilization of MySQL as shown in Figure 5(e). For instance, the point labeled 4 in Figure 5(b) illustrates zero throughput in Tomcat, which leads to the zero throughput and load of MySQL (see the point labeled $4'$ in Figure 5(e)).

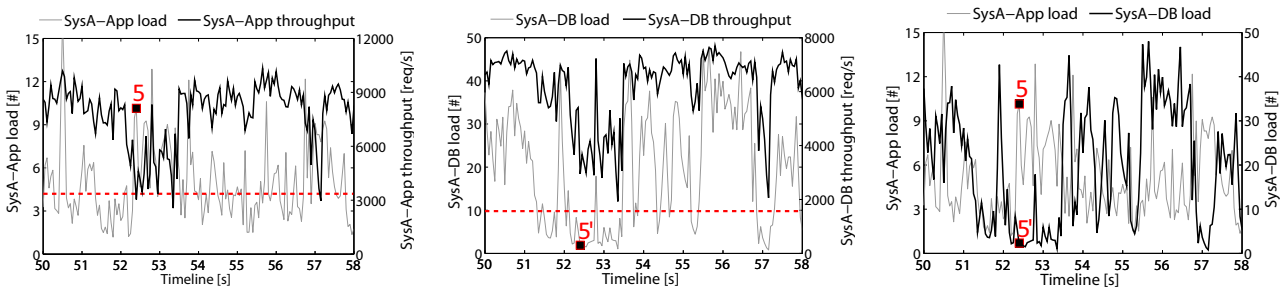
To illustrate the rapidly alternating bottleneck between Tomcat and MySQL, Figure 6(a) shows the correlation between the Tomcat load and the MySQL load over the same 12-second time period. This figure shows that these two metrics have a negative correlation (the Pearson correlation is -0.42), which suggests that the short-term saturation alternates between Tomcat and MySQL. Thus, the reason for the limited system throughput is clear: at any moment either Tomcat or MySQL becomes the bottleneck in the system.

Our further analysis shows that the short-term saturations of Tomcat are caused by frequent JVM GC. In this set of experiments, the JVM in Tomcat (with JDK 1.5) uses a synchronous garbage collector; it waits during the garbage collection period and only starts processing requests after the garbage collection is finished. To confirm that JVM GC causes the bottleneck in Tomcat, Figure 6(b) shows the timeline graph which correlates the Java GC running ratio [4] with the Tomcat load (50ms). This figure shows the occurrence of Tomcat JVM GC has a strong positive-correlation with the high load in Tomcat. The high peaks of JVM GC in Figure 6(b) stops Tomcat and makes requests queued in Tomcat dramatically. We note that such long freeze times in Tomcat do not happen frequently when the system is under low workload as shown in Figure 5(a). This is because JVM GC has a non-linear relationship with workload [19].

C. Rapidly Alternating Bottleneck Caused by VM Collocation

In this section we show the second rapidly alternating bottleneck case which is caused by VM collocation, i.e., collocating multiple under-utilized VMs into the same physical host so that VMs can share hardware resources such as CPU. Although VM collocation can reduce infrastructure and maintenance costs [9], it may significantly hamper the performance of the collocated applications in a non-trivial way, especially when the workload for the collocated applications becomes bursty [13], [14].

[4]Java GC running ratio means the total time spent on Java GC during each monitoring time interval to the total monitoring time interval length. JVM provides a tool recording the starting/ending timestamp of every GC activity.



(a) Tomcat tier of *SysA* (*SysA-App*) when the burstiness level of *SysB* workload is $I = 400$

(b) MySQL tier of *SysA* (*SysA-DB*) when the burstiness level of *SysB* workload is $I = 400$

(c) Negative correlation between *SysA-App*/*SysA-DB* load suggests rapidly alternating bottleneck.

Fig. 9: Fine-grained load/throughput analysis for the Tomcat and MySQL of *SysA* in the collocation experiments (see Figure 7 and Table II). Figure 9(a) and 9(b) show that both the Tomcat and MySQL of *SysA* frequently present short-term saturations.

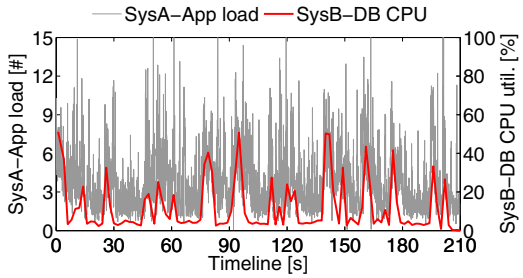


Fig. 10: *SysA-App* load vs. *SysB-DB* CPU; burst of *SysB-DB* CPU utilization causes frequent high load (long queue) in the *SysA-App* tier, which indicates frequent transient bottlenecks in *SysA-App*.

We illustrate this problem by collocating two VMs, each of which is from a separate n-tier application, into the same host and with each VM sharing the same CPU core. Figure 7 shows our collocation strategy of the two applications; *SysA* with 1L/2S/1L/2S configuration and *SysB* with 1S/1S/1S configuration. *SysA* keeps the same hardware configuration as in the previous sections but with JDK1.6 in Tomcat [5]. The VM of *SysA-App2* is collocated with the VM of *SysB-DB* on the same *ESXi* host and they share the same CPU core; the VMs of the front tiers of *SysB* are deployed in separate *ESXi* hosts from *SysA* to simplify the analysis. Table II shows the workload conditions for both systems and the average CPU utilization for the collocated VMs. *SysA* is at a constant stable workload of 14,000 in all five experiments. Except for the first experiment (the non-collocation case), *SysB* is at constant WL 400 but with varying burstiness levels, which is represented by I [6]. The average CPU utilization of both the collocated VMs *SysA-App2* and *SysB-DB* are almost constant and the total CPU utilization is less than 90%, which justifies the collocation strategy based on traditional bin packing practices.

Figure 8 shows the average response time of *SysA* in all the five cases. This figure shows that the *SysA* response time only increases significantly when the burstiness level of the workload for *SysB* becomes high (e.g, I = 400). The significant increase in response time for *SysA* may seem strange since the average CPU utilization remains constant as seen in Table II.

Figure 9(a) shows a similar interesting phenomenon as in the previous sections that in some time intervals (e.g, the point labeled 5) the *SysA-App* has a high load but low throughput; during these time intervals, *SysA-App* presents short-term saturations and *SysA-DB* is under-utilized due to fewer input requests fed from *SysA-App* (see Figure 9(b)).

Figure 9(c) shows the correlation of the load between *SysA-App* and *SysA-DB* over the same 8-second time period. This figure shows that *SysA-App* load has a negative correlation with *SysA-DB* load ($\rho_{X1,Y2}$ is -0.46), which suggests the bottleneck alternates rapidly between *SysA-App* and *SysA-DB*.

Our further analysis shows that the short-term saturation of *SysA-App* is caused by the burst of *SysB-DB* CPU utilization. Figure 10 shows the timeline graph of the CPU utilization of *SysB-DB* (measured using VMware esxtop with 2s granularity) and the *SysA-App* load (measured at every 50ms time interval). This figure shows that the *SysA-App* load increases significantly when there is a spike in the *SysB-DB* CPU utilization [7],which indicates that Tomcat of *SysA* temporarily becomes the system bottleneck due to the interference of *SysB-DB*. More detailed research about the CPU contention between collocated VMs has been studied by Malkowski et al. [13].

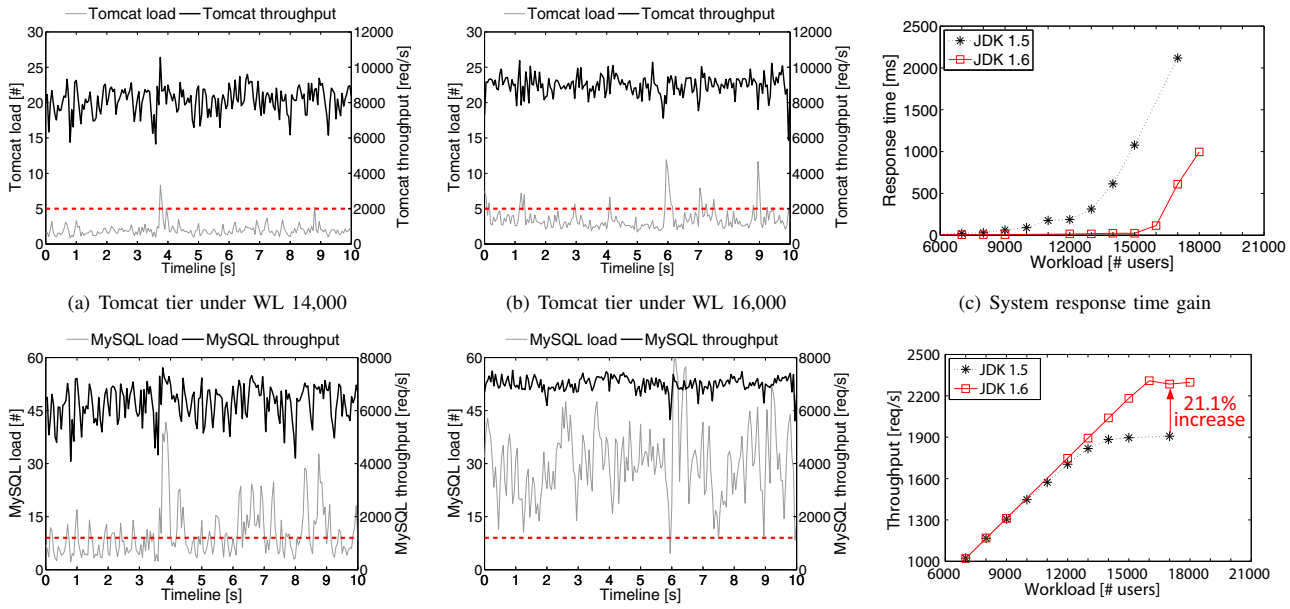
IV. Resolving Rapidly Alternating Bottlenecks

Once we detect a rapidly alternating bottleneck case, we can resolve the bottleneck through various ways, depending on whether we can find the exact cause for the rapidly alternating bottleneck. Specifically, we can simply scale-out/up the participating servers if we cannot find the exact cause, or we can resolve the bottleneck by addressing the exact cause. For instance, we can resolve the rapidly alternating bottleneck caused by frequent JVM GC in Tomcat as described in Section III-B through upgrading the JDK version from 1.5 to 1.6, which has more efficient garbage collectors.

[5]The upgrade of JDK version in Tomcat solves the rapidly alternating bottleneck caused by frequent JVM GC; see Section IV for more details.

[6]Mi et al. [15] introduced *index of dispersion* (abbreviated as I) to characterize the intensity of the traffic surges. The larger the I is, the longer the duration of the traffic surge. The burstiness level of the by default RUBBoS workload is $I = 1$.

[7]We cannot show the fine-grained CPU utilization of *SysB-DB* because 2s is the finest granularity the lastest esxtop supports.



(a) Tomcat tier under WL 14,000 (b) Tomcat tier under WL 16,000 (c) System response time gain

(d) MySQL tier under WL 14,000 (e) MySQL tier under WL 16,000 (f) System throughput gain

Fig. 11: Fine-grained load/throughput(50ms) analysis for Tomcat (with JDK 1.6) and MySQL. Figure 11(b) and 11(e) show that the rapidly alternating bottleneck is resolved and the MySQL tier becomes the single bottleneck.

Figure 11 shows the fine-grained load/throughput analysis for Tomcat and MySQL at WL 14,000 and 16,000 after we upgrade the Tomcat JDK version from 1.5 to 1.6. The experiments here have the same hardware/software configuration as in Section III-B except for the Tomcat JDK version. Recall from Section III-B the system throughput reaches the maximum at WL 14,000 due to the rapidly alternating bottleneck between Tomcat and MySQL. After the JDK version upgrade, Figure 11(a) and 11(d) show that Tomcat does not have long "freezing" periods (high load but low throughput) and only MySQL presents frequent short-term saturations at WL 14,000; further workload increase to 16,000 leads to the full saturation of MySQL as shown in Figure 11(e) (the load is above the $N^\star$ most of the time). Thus, the rapidly alternating bottleneck is resolved.

Figure 11(c) and 11(f) show the system response time and throughput gain after we resolve the rapidly alternating bottleneck. At WL 17,000, the system with JDK 1.6 outperforms the system with JDK 1.5 by a 21.1% higher throughput while achieving an average response time that is about 71% lower.

We note we can resolve the rapidly alternating bottleneck caused by VM collocation described in Section III-C through migrating the collocated VM to a different ESXi host. We omit such analysis due to space constraints.

V. Related Work

Shifting/Alternating bottlenecks have been studied before in either multiclass queueing networks or n-tier enterprise systems. Balbo et al. [4] and Casale et al. [6] use analytical approaches to illustrate that bottlenecks in a multiclass queueing network with load independent servers can switch to different servers, depending on the current workload mix. Malkowski et al. [12] showed an alternating bottleneck case with slow alternating pattern among eight MySQL databases in the system. As shown in this paper, alternating bottlenecks can be far more common than previously believed.

Analytical models have been proposed for bottleneck detection and performance prediction of n-tier systems. Xiong et al. [22] present a multi-level control approach for n-tier systems that employs a flexible queuing model to determine the optimal resources to each tier of the application under both total resource constrains and SLA constrains. However, such model uses Mean Value Analysis (MVA), which may not handle the alternating bottleneck cases in the system. Mi et al. [14] propose an analytical model that considers shifting bottleneck in an n-tier system due to bursty workloads; however, the accuracy of their model is unclear once the frequency of shifting becomes high.

Perhaps the work closest to ours is Aguilera et al.'s performance debugging based on Black boxes [3]. The authors propose a statistical method to derive causal paths (the trace of a transaction) in a distributed system from the communication messages between different nodes. By measuring the delay of each request in each node, they detect the "bottleneck server" as the node where a request has the longest delay. Though this approach can be effective to detect the single bottleneck case, it may fail to detect rapidly alternating bottlenecks.

VI. Conclusions

We observed a significant performance loss of an n-tier system due to rapidly alternating bottlenecks between multiple tiers (Section II-B). We proposed a novel bottleneck

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

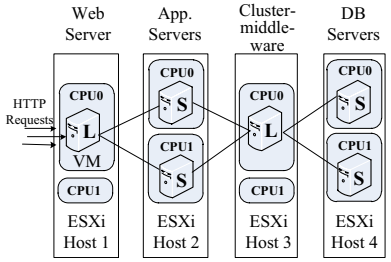
ESXi Host Configuration

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. 12: Details of the experimental setup.

detection method to detect these rapidly alternating bottlenecks (Section III-A). We found that rapidly alternating bottlenecks can be caused by factors at different levels of a system; for instance, JVM GC at the software level (Section III-B), VM collocation at the middleware level (Section III-C). Solving those rapidly alternating bottlenecks leads to significant performance improvement (Section IV). 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] G. Balbo and G. Serazzi. Asymptotic analysis of multiclass closed queueing networks: multiple bottlenecks. *Perform. Eval.*, 1997.

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

[6] G. Casale and G. Serazzi. Bottlenecks identification in multiclass queueing networks using convex polytopes. In *MASCOTS'04*.

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

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

[9] S. Govindan, J. Liu, A. Kansal, and A. Sivasubramaniam. Cuanta: quantifying effects of shared on-chip resource interference for consolidated virtual machines. SOCC '11.

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

[11] G. Jung, K. R. Joshi, M. A. Hiltunen, R. D. Schlichting, and C. Pu. A cost-sensitive adaptation engine for server consolidation of multitier applications. In *Middleware'09*.

[12] S. Malkowski, M. Hedwig, and C. Pu. Experimental evaluation of n-tier systems: Observation and analysis of multi-bottlenecks. In *IISWC '09*.

[13] S. Malkowski, Y. Kanemasa, H. Chen, M. Yamamoto, Q. Wang, D. Jayasinghe, M. Yamamoto, M. Kawaba, and C. Pu. Challenges and opportunities in consolidation at high resource utilization: Non-monotonic response time variations in n-tier applications. In *Cloud'12*.

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

[15] N. Mi, G. Casale, L. Cherkasova, and E. Smirni. Injecting realistic burstiness to a traditional client-server benchmark. ICAC '09.

[16] 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*.

[17] 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*.

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

[19] 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*.

[20] Q. Wang, Y. Kanemasa, J. Li, D. Jayasinghe, T. Shimizu, M. Matsubara, M. Kawaba, and C. Pu. Detecting transient bottlenecks in n-tier applications through fine-grained analysis. In *ICDCS'13*.

[21] 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*.

[22] P. Xiong, Z. Wang, S. Malkowski, Q. Wang, D. Jayasinghe, and C. Pu. Economical and robust provisioning of n-tier cloud workloads: A multi-level control approach. In *ICDCS'11*.

Appendix

In our experiments we adopt the RUBBoS [1] standard n-tier benchmark, based on bulletin board applications such as Slashdot. 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 [10]) system. The benchmark includes two kinds of workload modes: browse-only and read/write interaction mixes. We use browse-only workload in this paper.

We run the RUBBoS benchmark on our virtualized testbed. Figure 12 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 12(c) shows a sample 1L/2S/1L/2S topology. Each ESXi host runs the VMs from the same tier of the application. The VMs from the same tier are 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.