When Average is Not Average: Large Response Time Fluctuations in n-Tier Systems

Qingyang Wang[1], Yasuhiko Kanemasa[2], Motoyuki Kawaba[2], Calton Pu[1]
[1]College of Computing, Georgia Institute of Technology
[2]Cloud Computing Research Center, FUJITSU LABORATORIES LTD.
[1]{qywang, calton}@cc.gatech.edu, [2]{kanemasa, kawaba}@jp.fujitsu.com

ABSTRACT

Simultaneously achieving both good performance and high resource utilization is an important goal for production cloud environments. Through extensive measurements of an n-tier application benchmark (RUBBoS), we show that the response time of an n-tier system frequently presents large scale fluctuations (e.g., ranging from tens of milliseconds up to tens of seconds) during periods of high resource utilization.

Except the factor of bursty workload from clients, we found that the large scale response time fluctuations can be caused by some system environmental conditions (e.g., L2 cache miss, JVM garbage collection, inefficient scheduling policies) that commonly exist in n-tier applications. The impact of these system environmental conditions can largely amplify the end-to-end response time fluctuations because of the complex resource dependencies in the system. For instance, a 50ms response time increase in the database tier can be amplified to 500ms end-to-end response time increase. We evaluate three heuristics to stabilize response time fluctuations while still achieving high resource utilization in the system. Our results show that large scale response time fluctuations should be taken into account when designing effective autonomous self-scaling n-tier systems in cloud environments.

Categories and Subject Descriptors

C.2.4 [**Computer-Communication Networks**]: Distributed Systems-Distributed Applications; C.4 [**Performance of Systems**]: Reliability, availability, and serviceability; H.3.4 [**Performance evaluation (efficiency and effectiveness)**]: Metrics—*complexity measures, performance measures*

Keywords

N-tier system, Web-facing applications, Performance evaluation, scalability, Soft resources, burstiness.

1. INTRODUCTION

Simultaneously achieving good performance and high resource utilization is an important goal for production cloud environments. High utilization is essential for high return on investment for cloud providers and low sharing cost for cloud users [9]. 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, simultaneously achieving both objectives for applications that are *not* embarrassingly parallel has remained an elusive goal. Consequently, both practitioners and researchers have encountered serious difficulties in predicting response time in clouds during periods of high utilization. A practical consequence of this problem is that enterprise cloud environments have been reported to have disappointingly low average utilization (e.g., 18% in [17]).

In this paper, we describe concrete experimental evidence that shows an important contributing factor to the apparent unpredictability of cloud-based application response time when under high utilization conditions. Using extensive measurements of an n-tier benchmark (RUBBoS [1]), we found the presence of large scale response time fluctuations. These fluctuations, ranging from tens of milliseconds up to tens of seconds, appear when workloads become bursty [13], as expected of web-facing applications. The discovery of these large scale response time fluctuations is important as it will have significant impact on the autonomous performance prediction and tuning of n-tier application performance, even for moderately bursty workloads. Specifically, a distinctly bi-modal distribution with two modes (that span a spectrum of 2 to 3 orders of magnitude) can cause significant distortions on traditional statistical analyses and models of performance that assume uni-modal distributions.

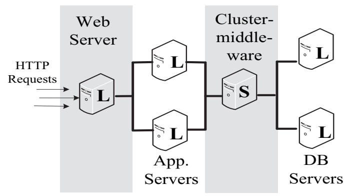
One of the interesting facts that made this research challenging is that the long queries (that last several seconds) are not inherently complex in their nature, i.e., they are normal queries that would finish within tens of milliseconds when run by themselves. Under a specific (and not-so-rare) set of system environmental conditions, these queries take several seconds. The detailed analysis to reveal these system environmental conditions in an n-tier system is non-trivial considering that classical performance analysis techniques that assume uni-modal distributions are inapplicable. Our approach recorded both application level and system level metrics (e.g., response time, throughput, CPU, and disk I/O) of each tier in an n-tier system at fine-grained time granularity (e.g., 100ms). Then we analyzed the relationship

Function	Software
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	jdk1.6.0_14
Operating system	RHEL Server 5.7 (Tikanga)
System monitor	Sysstat 10.0.02, Collectl 3.5.1
Transaction monitor	Fujitsu SysViz

(a) Software setup

Hardware	Processor			Memory	Disk	Network
	# cores	Freq.	L2 Cache			
Large (L)	2	2.27GHz	2M	2GB	200GB	1Gbps
Medium (M)	1	2.4 GHz	4M	2GB	200GB	1Gbps
Small (S)	1	2.26GHz	512k	1GB	80GB	1Gbps

(b) Hardware node setup



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

Figure 1: Details of the experimental setup.

of these metrics among each tier to identify the often shifting and sometimes mutually dependent bottlenecks. The complexity of this phenomenon is illustrated by a sensitivity study of soft resource allocation (e.g., number of threads in the web and application servers and DB connection pool) on system performance and resource utilization.

The first contribution of the paper is an experimental illustration of the large scale response time fluctuations of systems under high resource utilization conditions using the n-tier RUBBoS benchmark. Due to the large fluctuations, the average system response time is not representative of the actual system performance. For instance, when the system is under a moderately bursty workload and the average utilization of the bottleneck resource (e.g., MySQL CPU) is around 90%, the end-to-end response time shows a distinctly bi-modal distribution (Section 2.2).

The second contribution of the paper is a detailed analysis of several system environmental conditions that cause the large scale response time fluctuations. For instance, some transient events (e.g., CPU overhead caused by L2 cache miss or Java GC, see Section 4.1) in the tier under high resource utilization conditions significantly impact the response time fluctuations of the tier. Then the in-tier response time fluctuations is amplified to the end-to end response time due to the complex resource dependencies across tiers in the system (Section 4.2). We also found that the operating system (OS) level "best" scheduling policy in each individual tier of an n-tier system may not achieve the best overall application level response time (Section 4.3).

The third contribution of the paper is a practical solution for stabilizing the large scale response time fluctuations of systems under high resource utilization conditions (Section 5). For instance, our experimental results show that the CPU overhead caused by transient events can be reduced by limiting the concurrency of request processing in the bottleneck tier (heuristic ii) while the limitations of OS level scheduling policies can be overcome through application level transaction scheduling (heuristic i).

The rest of the paper is organized as follows. Section 2 shows the large scale response time fluctuations using a concrete example. Section 3 illustrates our fine-grained monitoring analysis. Section 4 shows some system environmental conditions for the large scale response time fluctuations. Section 5 explains three heuristics in detail. Section 6 summarizes the related work and Section 7 concludes the paper.

2. BACKGROUND AND MOTIVATION

2.1 Background Information

In our experiments we adopt the RUBBoS 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 includes 24 different interactions such as "register user" or "view story". The benchmark includes two kinds of workload modes: browse-only and read/write interaction mixes. We use browse-only workload in this paper.

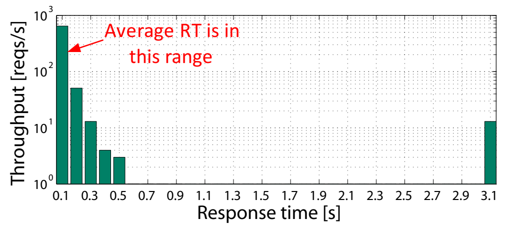
Mi et al. [13] proposed a *bursty workload generator* which takes into account the Slashdot effect, where a web page linked by a popular blog or media site suddenly experiences a huge increase in web traffic. Unlike the original workload generator which generates a request rate that follows a Poisson distribution parameterized by a number of emulated browsers and a fixed user think time E[Z], the bursty workload generator generates request rates in two modes: a fast mode with short user think time and a slow mode with long user think time. The fast mode simulates the Slashdot effect where the workload generator generates traffic surges for the system. The bursty workload generator uses one parameter to characterize the intensity of the traffic surges: *index of dispersion*, which is abbreviated as I. The larger the I is, the longer the duration of the traffic surge. In this paper, we use both the original workload generator (with $I = 1$) and the bursty workload generator (with $I = 100$, 400, and 1000) to evaluate the system performance.

Figure 1 outlines the details of the experimental setup. We carry out the experiments by allocating a dedicated physical node to each server. A four-digit notation $\#W/\#A/\#C/\#D$ is used to denote the number of web servers, application servers, clustering middleware servers, and database servers. We have three types of hardware nodes: "L", "M", and "S", each of which represents a different level of processing power. Figure 1(c) shows a sample 1L/2L/1S/2L topology. Hardware resource utilization measurements are taken during the runtime period using collectl at different time granularity. We use Fujitsu SysViz [3], a prototype tool developed by Fujitsu laboratories, as a transaction monitor to precisely measure the response time and the number of concurrent requests in each short time window (e.g., every 100ms) with respect to each tier of an n-tier system.

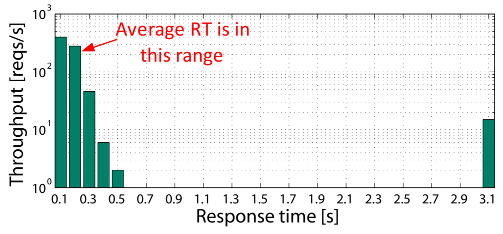
2.2 Motivation

In this section, we give one example to show that the average of measured performance metrics may not be representative of the actual system performance perceived by clients when the system is under high utilization conditions. The results shown here are based on 10-minute runtime experiments of RUBBoS benchmark running in a four-tier system (see Figure 1(c)) with different burstiness levels of workload.

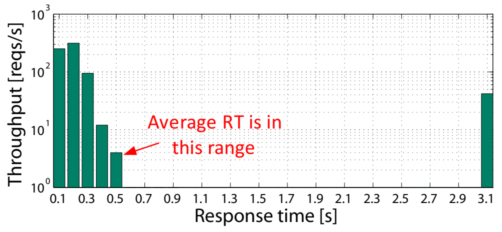
Figure 2 shows the system response time distribution with four different burstiness levels of workload. The sum of the value of each bar in a subfigure is the total system throughput. We note that in all these four cases, the CPU utilization



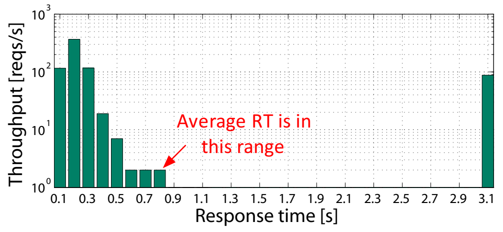
(a) $I = 1$ (original workload generator); average RT = 0.068s



(b) $I = 100$; average RT = 0.189s



(c) $I = 400$; average RT = 0.439s



(d) $I = 1000$; average RT = 0.776s

Figure 2: End-to-end response time distribution of the system in workload 5200 with different burstiness levels; the average CPU utilization of the bottleneck server is 90% in 10 minutes runtime experiments for all the four cases.

of the bottleneck server (the CJDBC server) of the system is 90%. This figure shows that the response time distribution in each of these four cases has a distinctly bi-modal characteristic; while majority of requests from clients finish within a few hundreds of milliseconds, a few percentage finish longer than three seconds. Furthermore, this figure shows the more bursty the workload, the more requests there will be with response time longer than 3 seconds.

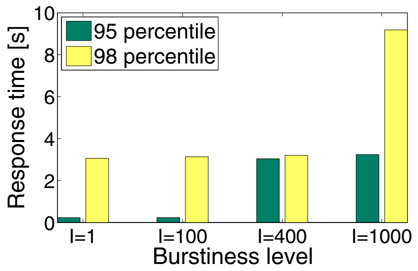


Figure 3: The percentiles of system response time in workload 5200 with different burstiness levels.

Large scale response time fluctuations have significant negative impact on the performance of a system requiring strict Service Level Agreement (SLA) guarantees such as bounded response time. Figure 3 shows the 95- and 98-percentiles of the end-to-end response time under different levels of bursty workload. For the original workload ($I = 1$) case and the bursty workload ($I = 100$) case, the 95th percentile is very low (less than 200ms) while the 98th percentile is over 3 seconds. As the burstiness level of workload increases, even the 95-percentile's response time is beyond 3 seconds, and the 98-percentile's for bursty workload ($I = 1000$) case exceeds 9 seconds. Some web-facing applications have strict response time requirement, for example, Google requires clients' requests to be processed within one second [2]. Thus, response time with large scale fluctuations may lead to severe SLA violations though the average response time is small.

3. FINE-GRAINED ANALYSIS FOR LARGE RESPONSE TIME FLUCTUATIONS

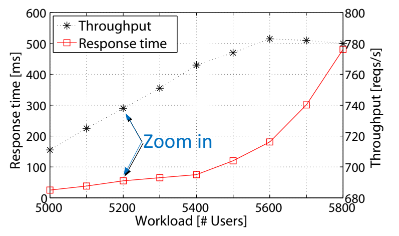
In this section we show the cause of the distinctly bi-modal response time distribution as introduced in the motivation case through fine-grained analysis. The results here are based on the same configuration as shown in the motivation case. We use the original workload generator ($I = 1$), which is an extension analysis for the case as shown in Figure 2(a).

Figure 4(a) shows the average throughput and response time of the system from workload 5000 to 5800. The response time distribution shown in Figure 2(a) is based on the result of workload 5200, where the average response time is 0.068s and the average CPU utilization of CJDBC server is about 90% (see Figure 4(d)). Next, we zoom in the highly aggregated average of the application/system metrics measured in workload 5200 through fine-grained analysis.

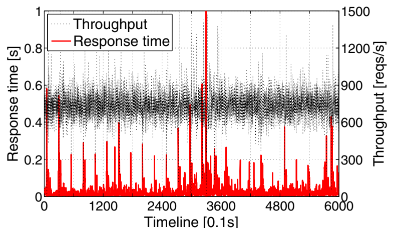
Figure 4(b) and 4(c) show the average system response time and throughput aggregated at 100ms and 10s time granularities respectively. Figure 4(b) shows both the system response time and throughput present large fluctuations while such fluctuations are highly blurred when 10 second time granularity is used (Figure 4(c)). Figure 4(e) and 4(f) show the similar graphs for the CJDBC (the bottleneck server) CPU utilization. Figure 4(e) shows the CJDBC CPU frequently reaches 100% utilization if monitored at 100ms granularity while such CPU saturation disappears if 10s time granularity is used [1].

Figure 4(h) and 4(i) show the number of concurrent requests on the Apache web server aggregated at 100ms and 10s time granularity in workload 5200. Concurrent requests on a server refer to the requests that have arrived, but have not departed from the server; these requests are being processed concurrently by the server due to the multi-threading architecture adopted by most modern internet server designs (e.g., Apache, Tomcat, and MySQL). We note that the

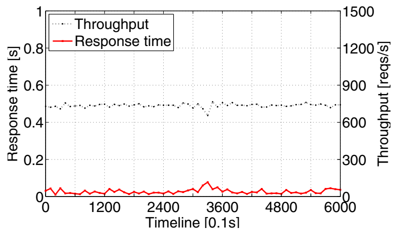
[1]10 seconds or even longer control interval is frequently used in automatic self-scaling systems [5, 12, 15, 20].



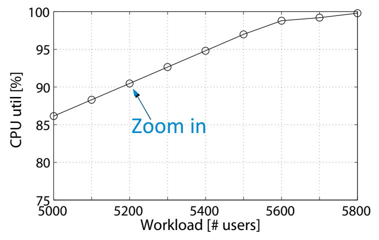
(a) End-to-end RT and TP; subfigures on the right show the "zoom in" results.



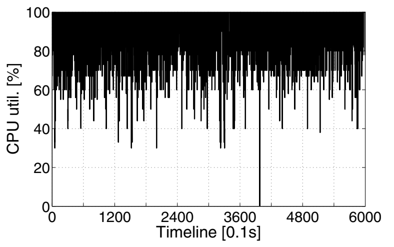
(b) Large fluctuations of RT and TP (average in each 100ms).



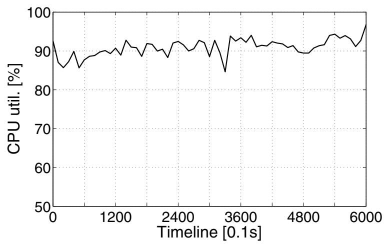
(c) Relatively stable RT and TP (average in each 10s).



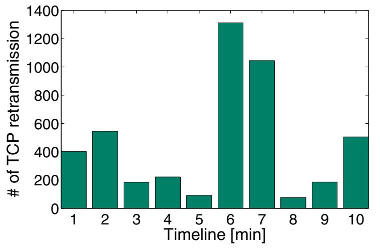
(d) Bottleneck server CPU usage; subfigures on the right show the "zoom in" results.



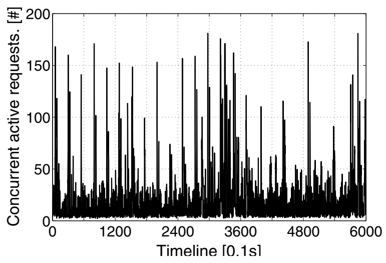
(e) Large fluctuations of CPU usage (average in each 100ms).



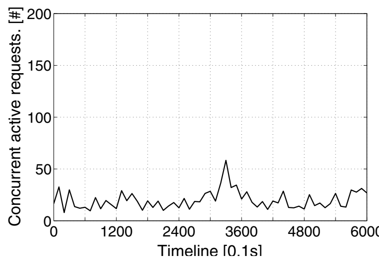
(f) Relatively stable CPU usage (average in each 10s).



(g) # of TCP retransmission in each minute.



(h) Large concurrent request fluctuations in Apache (average in each 100ms).



(i) Relatively stable concurrent requests in Apache (average in each 10s).

Figure 4: Analysis of system/application level metrics for the large response time fluctuations of the system (1L/2L/1S/2L config.). Requests with long response time are caused by TCP transmissions as shown in subfigure (g), which are caused by the large fluctuations of concurrent requests in Apache web tier as shown in subfigure (h).

thread pool size we set for the Apache web server in this set of experiments is 50; considering the underlying operating system has a buffer (TCP backlog, the default size is 128) for incoming TCP connection requests from clients, the maximum number of concurrent requests the Apache web server can handle is 178. Once the server reaches the limit, the new incoming requests will be dropped and TCP retransmission happens, which causes the long response time perceived by a client [2]. Figure 4(h) shows that the concurrent requests, if aggregated at 100ms time granularity, frequently present high peaks which are close to the limit. Such high peaks cause large number of TCP retransmissions as shown in Figure 4(g), which counts the number of TCP retransmissions in every minute during the 10-minute runtime experiment.

3.1 Sensitivity Analysis of Large Fluctuations with Different Bursty Workloads

System administrators may want to know under which workload(s) the large scale response time fluctuations happen. Table 1 shows the minimum workload (with different burstiness levels) under which the system has at least 1% requests that encounter TCP retransmissions. This table shows that both the threshold workload and the corresponding average CPU utilization of the bottleneck server decrease as the burstiness level of workload increases. This further justifies that the evaluation of the large scale response time fluctuations using fine-grained monitoring is an important and necessary step in autonomic system design.

Burstiness level	Threshold WL	Bottleneck server CPU util.
I =1	5000	88.1%
I =100	4800	86.3%
I =400	4400	80.4%
I =1000	3800	74.6%

Table 1: Workload (with different burstiness levels) beyond which more than 1% TCP retransmission happens.

4. SYSTEM CONDITIONS FOR LARGE RESPONSE TIME FLUCTUATIONS

Understanding the exact causes of large scale response time fluctuations of an n-tier system under high utilization conditions is important to efficiently utilize the system resources while achieving good performance. In this section

[2]TCP retransmission is transparent to clients; the waiting time is three seconds for the first time and is exponentially increased for the consecutive retransmissions (RFC 2988).

we will discuss some system environmental conditions that cause large scale response time fluctuations even under the moderately bursty workload from clients. We note that all the experimental results in this section are based on the original RUBBoS browse-only workload ($I = 1$).

4.1 Impact of Transient Events

Transient events are events that are pervasive but only happen from time to time in computer systems, such as L2 cache miss, JVM GC, page fault, etc. In this section we will show two types of transient events, L2 cache miss (the last level cache) and JVM GC, that cause significant overhead to the bottleneck resource in the system, especially when the bottleneck tier is in high concurrency of request processing.

4.1.1 CPU overhead caused by L2 cache misses

For modern computer architectures, caching effectiveness is one of the key factors for system performance [8, 14]. We found that the number of L2 cache misses of the bottleneck server in an n-tier system increases *nonlinearly* as workload increases, especially when the system is under high utilization conditions. Thus the CPU overhead caused by L2 cache misses significantly impacts the large scale response time fluctuations of the system.

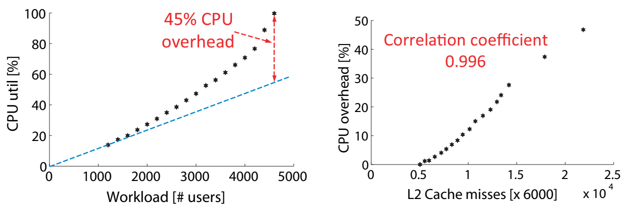
The hardware configuration of the experiments in this section is 1L/2L/1M (one Apache and two Tomcats on the type "L" machine, and one MySQL on the type "M" machine). Under this configuration, the MySQL server CPU is the bottleneck of the system. We choose the "M" type machine for MySQL as the corresponding Intel $Core^{TM}2$ CPU has two CPU performance counters which allow us to monitor the L2 cache misses during the experiment.

Figure 5(a) shows the MySQL CPU utilization as workload increases from 1200 to 4600 at a 200 increment per step. Ideally the MySQL CPU should increase linearly as workload increases until saturation if there is no CPU overhead. However, this figure clearly shows that the CPU overhead increases nonlinearly as workload increases, especially in high workload range. In order to quantify the CPU overhead and simplify our analysis, we make one assumption here: MySQL has no CPU overhead for request processing from workload 0 to workload 1200 (our starting workload). Under this assumption, we can quantify the CPU overhead for the following increasing workloads by measuring the distance between the actual CPU utilization and the ideal CPU utilization. For instance, under workload 4600, the MySQL CPU overhead reaches 45%.

Figure 5(b) shows the correlation between the number of L2 cache misses of MySQL and the corresponding CPU overhead from workload 1200 to 4600. The CPU overhead is calculated as shown in Figure 5(a) and the number of L2 cache misses in MySQL is recorded using the CPU performance counter [3] during the runtime experiments. This figure shows that the L2 cache misses and the corresponding CPU overhead are almost linearly correlated; thus higher L2 cache misses indicate higher CPU overhead.

One more interesting phenomenon we found is that the CPU overhead caused by L2 cache misses can be effectively reduced by limiting the concurrency level of request processing in the bottleneck server. Table 2 shows the comparison of CPU utilization and L2 cache misses under two different DB

[3]The CPU performance counter increases by 1 for 6000 L2 cache misses in our environmental settings.



(a) MySQL CPU overhead (b) L2 misses vs. CPU overhead

Figure 5: CPU overhead caused by L2 cache misses.

WL	DBconn12 TP (req/s)	DBconn12 CPU util (%)	DBconn12 L2 miss (×6000)	DBconn2 TP (req/s)	DBconn2 CPU util (%)	DBconn2 L2 miss (×6000)
1200	168	13.8	5036	169	13.6	4704
2400	340	34.9	8320	340	34.6	8153
3600	510	61.0	12304	510	60.2	11233
3800	538	**66.1**	**12963**	536	**64.5**	**11968**
4200	595	**76.6**	**14204**	595	**74.8**	**13053**
4600	642	**99.6**	**21868**	650	**86.2**	**14133**

Table 2: Comparison of MySQL CPU utilization and L2 cache misses between DBconn12 and DBconn2 with 1L/2L/1M configuration; higher concurrency leads to more L2 cache misses in the bottleneck tier (MySQL).

connection pool sizes in Tomcat: DBconn12 and DBconn2. In the current RUBBoS implementation, each Servlet has its own local DB connection pool; DBconn12 means the DB connection pool size for each Servlet is 12 while DBconn2 means 2. This table shows that although the throughputs of these two cases are similar under different workloads, the DBconn2 case has less CPU utilization and less L2 cache misses in MySQL than the DBconn12 case, especially in the high workload range. We note that the DB connection pools in Tomcat controls the number of active threads in MySQL. In the DBconn12 case under high workload more concurrent requests are sent to the MySQL server, thus more concurrently active threads are created in MySQL and contend for the limited space of L2 cache causing more cache misses and CPU overhead than those in the DBconn2 case.

4.1.2 CPU overhead caused by Java GC

For Java-based servers like Tomcat and CJDBC, the JVM garbage collection process impacts the system response time fluctuations in two ways: first, the CPU time used by the garbage collector cannot be used for request processing; second, the JVM uses a synchronous garbage collector and it waits during the garbage collection period, only starting to process requests after the garbage collection is finished [4]. This delay significantly lengthens the pending requests and causes fluctuations in system response time.

Our measurements show that when a Java-based server is highly utilized, the JVM GCs of the server increase *nonlinearly* as workload increases. The hardware configuration of the experiments in this section is 1L/2L/1S/2L (see Figure 1(c)). Under this configuration, the CJDBC CPU is the bottleneck of the system. We note that the CJDBC server is a Java-based DB clustering middleware; each time a Tomcat server establishes a connection to the CJDBC server, which balances the load among the DB servers, a thread is created by CJDBC to route the SQL query to a DB server.

Table 3 compares the CPU utilization and the total GC time of the CJDBC server during the runtime experiments

WL	DBconn24 TP (req/s)	DBconn24 CPU util (%)	DBconn24 GC (s)	DBconn2 TP (req/s)	DBconn2 CPU util (%)	DBconn2 GC (s)
3000	428	49.6	0.05	428	49.2	0.05
4000	572	69.0	0.07	571	68.8	0.07
5000	721	**86.1**	**1.06**	719	**84.8**	**0.19**
5200	738	**91.2**	**1.51**	737	**87.4**	**0.37**
5400	759	**94.3**	**1.72**	767	**91.1**	**0.40**
5600	779	**98.8**	**2.15**	795	**96.6**	**0.45**

Table 3: Comparison of CJDBC CPU utilization and JVM GC time between DBconn24 and DBconn2 with 1L/2L/1S/2L configuration; higher concurrency leads to longer JVM GC time in the bottleneck tier (CJDBC).

between the cases DBconn24 and DBconn2 from workload 3000 to 5600. This table shows that the total GC time for both the two cases increases nonlinearly as workload increases, especially when the CJDBC CPU approaches saturation. One reason is that when the CJDBC CPU approaches saturation, the available CPU for GC shrinks; thus cleaning the same amount of garbage takes longer time than in the non-saturation situation. Accordingly, the impact of JVM GC on system response time fluctuations is more significant when CJDBC approaches saturation.

Table 3 also shows that the total GC time of the CJDBC server in the DBconn24 case is longer than that in the DBconn2 case from workload 5000 to 5600. The reason is similar to the L2 cache miss case as introduced in Section 4.1.1. Compared to the DBconn2 case, the Tomcat App tier in the DBconn24 case is able to send more concurrent requests to the CJDBC server under high workload, which in turn creates more concurrent threads for query routing and consumes more memory. Thus the CJDBC server performs more GCs for cleaning garbage in memory in the DBconn24 case than that in the DBconn2 case.

4.2 Fluctuation Amplification Effect in n-Tier Systems

Unlike some embarrassingly parallel "web indexing" applications using MapReduce and Hadoop, an n-tier application is unique in its amplification effect among different tiers due to the complex resource dependencies in the system. For instance, small request rate fluctuations from clients can be amplified to a bottom tier (e.g., DB tier), which causes significant response time fluctuation in the bottom tier; on the other hand, response time fluctuations in the bottom tier can be amplified to the front tiers.

4.2.1 *Top-down request rate fluctuation amplification*

The traffic for an n-tier system is, by nature, bursty [13]. One interesting phenomenon we found is that the bursty request rate from clients can be amplified to the bottom tier of the system. Except for the impact of transient events such as JVM GC, the complexity of inter-tier interactions of an n-tier system contributes most to the amplification effect. For example, a client's HTTP request may trigger multiple interactions between the application server tier and the DB tier to retrieve all the dynamic content to construct the web page requested by the client (We define the entire process as a client *transaction*).

Figure 6 shows the approximately instant request rate (aggregate at every 100ms) received by the Apache web tier and the MySQL DB tier of a three tier system (1L/2L/1L) in

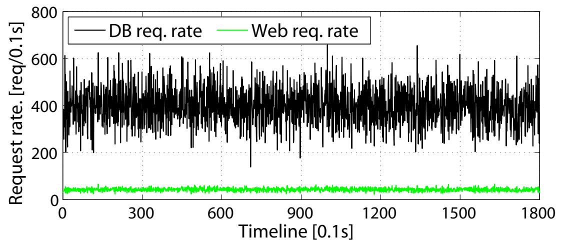


Figure 6: Amplified request rate flucutation from the web tier to the DB tier with 1L/2L/1L configuration in WL 3000.

Req. Rate (req/0.1s)	Web	App	DB
Mean	42.88	41.12	397.40
Std. Deviation	6.71	6.53	77.70
Coefficient of Variance.	**0.16**	0.16	**0.20**

Table 4: Statistic analysis of top-down request rate fluctuation amplification (corresponds to Figure 6).

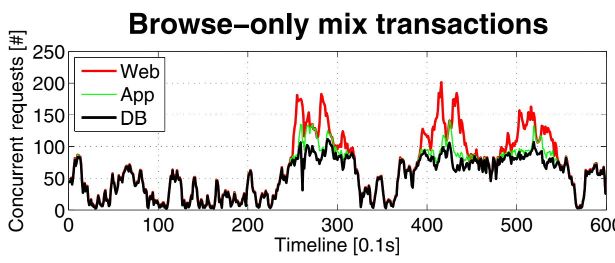
workload 3000. This figure shows that the request rate fluctuation in the MySQL tier is significantly larger than that in the Apache web tier. Table 4 shows the statistical analysis result of the amplification effect corresponding to Figure 6. This table shows three values related to the request rate for each tier: mean, standard deviation, and coefficient of variation (CV) [4]. Comparing the mean request rate between the web tier and the DB tier, one HTTP request can trigger 9.3 database accesses on average, which explains why the instant DB request rate is much higher than the instant Web request rate; second, the CV of the request rate in the DB tier (0.20) is larger than that in the web tier (0.16), which shows the effect of request rate fluctuation amplification from the web tier to the DB tier.

4.2.2 *Bottom-up response time fluctuation amplification*

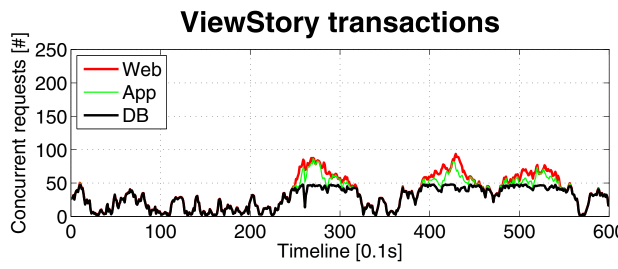
Due to the top-down request rate fluctuation amplification and also the interference of transient events, the response time of the bottom tier in an n-tier system naturally fluctuates. We found that even small response time fluctuations in the bottom tier can be amplified to the front tiers due to the following two reasons.

First, the complex soft resource dependencies among tiers may cause requests to queue in front tiers before they reach the bottom tier, which increases the waiting time of transaction execution. Soft resources refer to system software components such as threads, TCP connections, and DB connections [19]. In an n-tier system, every two consecutive tiers in an n-tier system are connected through soft resources during the long invocation chain of transaction execution in the system. For example, the Tomcat App tier connects to the MySQL tier through DB connections. Such connections are usually limited soft resources; once soft resources in a tier run out, the new requests coming to the tier have to queue in the tier until they get the released soft resources by other finished requests in the same tier. We note that for a RPC-style n-tier system, a request in a front tier releases soft resources (e.g., a processing thread) in the tier until

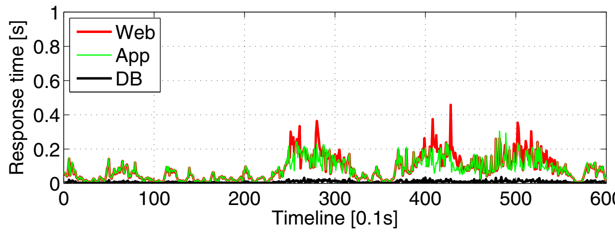
[4]Coefficient of variation means normalized standard deviation, which is standard deviation divided by mean.



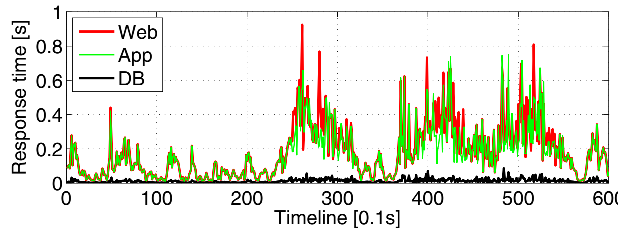
(a) Approximately instant # of concurrent requests in each tier



(b) Approximately instant # of concurrent ViewStory requests in each tier



(c) Approximately instant response time in each tier



(d) Approximately instant response time for ViewStory requests in each tier

Figure 7: Amplified response time fluctuations from the DB tier to the web tier with 1L/2L/1L (DBconn24) configuration in WL 5400.

the downstream tiers finish all the processing for the corresponding transaction. Accordingly, long response times in the bottom tier may lead to the saturation of soft resources (and thus a large number of queued requests) in front tiers.

Figure 7(a) shows the approximately instant number of concurrent requests (aggregated every 100ms) in each tier of a three-tier system (1L/2L/1L, MySQL is the bottleneck tier) under workload 5400. This figure shows that when the number of concurrent requests in MySQL reaches about 90, requests start to queue in the front tiers due to the scarcity of DB connections in Tomcat. Figure 7(c) shows the approximately instant response time in each tier. This figure shows that very small response time fluctuations (within 50ms) in MySQL lead to large response time fluctuations in Tomcat and Apache; the high peaks of response time in Figure 7(c) match well with the high peaks of queued requests in front tiers as shown in Figure 7(a). This indicates the waiting time of requests in front tiers largely contributes to the long response time of transaction execution.

Second, multi-interactions between tiers of an n-tier system amplify the bottom-up response time fluctuations. In an n-tier system it is natural that some transactions involve more interactions between different tiers than the other transactions. For example, in the RUBBoS benchmark, a ViewStory request triggers an average of twelve interactions between Tomcat and MySQL; a small response time increment in MySQL leads to a largely amplified response time in Tomcat and thus longer occupation time of soft resources in Tomcat. In such case, soft resources such as DB connections in Tomcat are more likely to run out, which leads to longer waiting time of the queued requests in Tomcat.

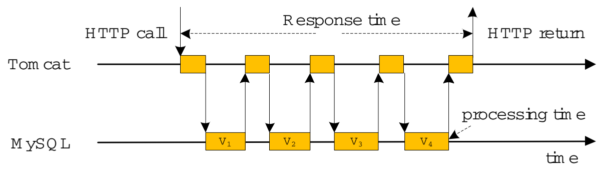
Figure 7(b) and 7(d) show the similar graphs as shown in Figure 7(a) and 7(c), but only for ViewStory transactions. Compared to Figure 7(c), Figure 7(d) shows that the response time of ViewStory requests in the Apache tier fluctuates more significantly. This is because ViewStory requests involve more interactions between Tomcat and MySQL than the average and run out their local DB connections earlier than the other types of requests; thus new incoming ViewStory requests have to wait longer in the Tomcat App tier (or in the Apache web tier if the connection resources between Apache and Tomcat also run out).

4.3 Impact of Mix-Transactions Scheduling in n-Tier Systems

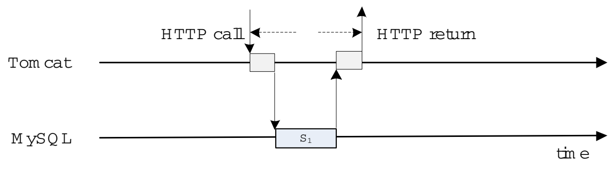
Scheduling polices impacting web server performance have been widely studied [10, 16]. These previous works mainly focus on a single web server and show that the performance can be dramatically improved via a kernel-level modification by changing the scheduling policy from the standard FAIR (processor-sharing) scheduling to SJF (shortest-job-first) scheduling. However, for more complex n-tier systems where a completion of a client transaction involves complex interactions among tiers, the best OS level scheduling policy may increase the overall transaction response time.

The main reason for this is because the operating system of each individual server in an n-tier system cannot distinguish heavy transactions from light transactions without application level knowledge. A transaction being heavier than a light transaction can be caused by the heavy transaction having more interactions between different tiers than the light one. However, in each individual interaction the processing time of the involved tiers for a heavy transaction can be even smaller than that for a light transaction. Since the operating system of a tier can only schedule a job based on the processing time of the current interaction, applying SJF scheduling policy to the operating system of each tier may actually delay the application level light transactions.

Figure 8 shows sample interactions between a Tomcat App tier and a MySQL tier for a ViewStory transaction (heavy) and a StoryOfTheDay transaction (light) specified in the RUBBoS benchmark. A ViewStory transaction involves multiple interactions between Tomcat and MySQL (see Figure 8(a)) while a StoryOfTheDay transaction in-



(a) A sample ViewStory (heavy) transaction processing



(b) A sample StoryOfTheDay (light) transaction processing

Figure 8: ViewStory vs. StoryofTheDay, different interaction pattern between Tomcat and MySQL.

volves only one interaction (see Figure 8(b)). Suppose MySQL is the bottleneck tier. Our measurements show that a single query from a ViewStory transaction has similar execution time in MySQL as a query from a StoryOfTheDay transaction. During each interaction, a thread in the MySQL tier receives a query from Tomcat and returns a response after the query processing, regardless of which servlet sends the query. From MySQL's perspective, MySQL cannot distinguish which transaction is heavy and which transaction is light. Thus either FAIR or SJF scheduling in the MySQL tier can delay the processing of the light transactions.

We note that once the waiting time of queries from light transactions increases in MySQL, the total number of queued light requests in upper tiers also increases. Since each queued request (regardless if entailing heavy or light transactions) in upper tiers occupies soft resources such as threads and connections, soft resources in upper tiers are more likely to run out under high workload. In this case, the response time fluctuations in a bottom tier are more likely to amplified to upper tiers (see Section 4.2.2).

5. SOLUTION AND EVALUATION

So far we have discussed some system environmental conditions causing the large scale response time fluctuations under high utilization conditions and explained the unique amplification effect inside an n-tier system. In this section we will evaluate three heuristics to help stabilizing the large scale response time fluctuations.

Heuristic (i): *We need to give higher priority to light transactions than heavy transactions to minimize the total amount of waiting time in the whole n-tier system. We need to schedule transactions in an upper tier which can distinguish light transactions from heavy transactions.*

Heuristic (i) is essentially an extension of applying the SJF scheduling policy in the context of n-tier systems. Suppose the MySQL tier is the bottleneck tier; as explained in section 4.3, applying SJF scheduling policy to MySQL through the kernel-level modification may not reduce the overall system response time because MySQL cannot distinguish application level heavy transactions and light transactions. Thus we need to schedule transactions in an upper tier that can make such distinction in order to apply SJF scheduling policy properly in an n-tier system. We define such scheduling as cross-tier-priority (CTP) based scheduling.

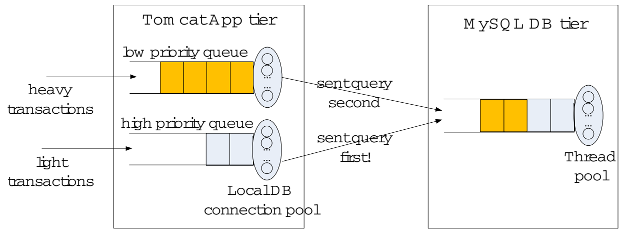


Figure 9: Illustration of applying CTP scheduling policy across tiers (only 2 servlets shown).

Figure 9 illustrates how to apply the CTP scheduling to a simple two-tier system. This figure shows only requests for two servlets (the RUBBoS browse-only workload consists of requests for eight servlets): ViewStory (heavy) and StoryOfTheDay (light). Once ViewStory requests and StoryOfTheDay requests reach the Tomcat App tier at the same time, we give StoryOfTheDay requests higher priority to send queries to MySQL. In this case the waiting time of the light StoryOfTheDay transactions can be reduced and the overall waiting time for all transactions is reduced [5].

Figure 10 shows the response time stabilization by applying the CTP scheduling to a three-tier system (1L/2L/1L with DBconn2) in workload 5800. Under this configuration, the MySQL CPU is the bottleneck in the system. Figure 10(a) and 10(c) show the results of the original RUBBoS implementation (using the default OS level scheduling) and Figure 10(b) and 10(d) show the results after the CTP scheduling is applied to the Tomcat App tier and the MySQL DB tier (see Figure 9).

Figure 10(a) and Figure 10(b) show the number of concurrent requests in each tier of the three-tier system for these two cases. Although in both cases the number of concurrent requests in the MySQL tier is very small (around eight), the fluctuations of the number of concurrent requests in the Tomcat App tier and the Apache web tier are much higher in the original case than those in the CTP scheduling case. This is because in the original case more light requests are queued in the upper tiers due to the increased waiting time of light requests in the MySQL tier.

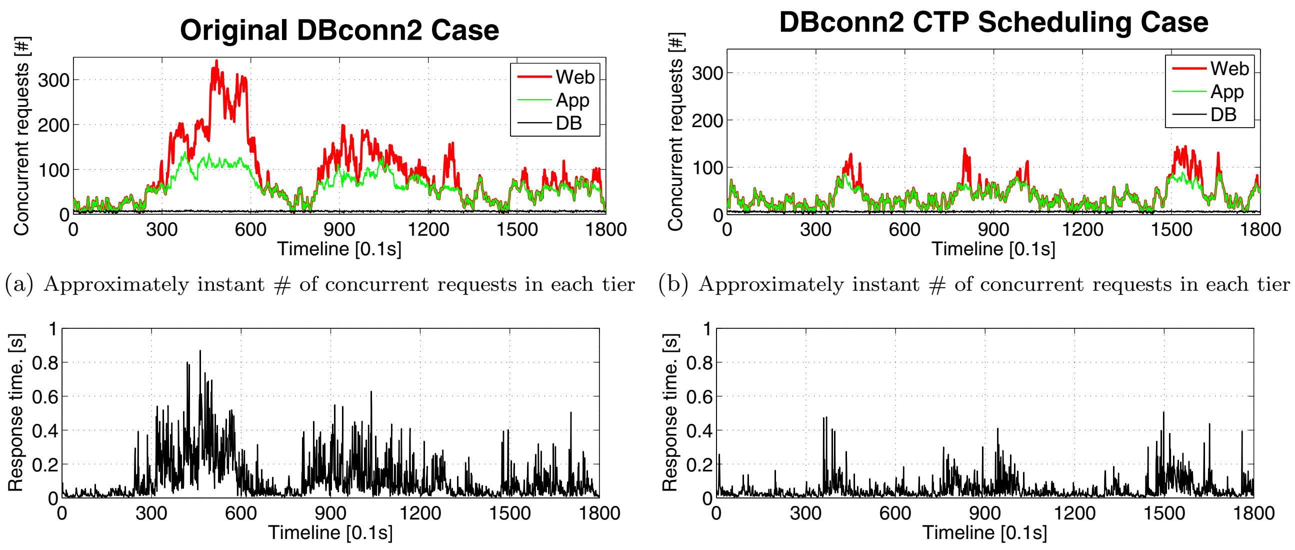
Figure 10(c) and Figure 10(d) show that the approximately instant response time in the Apache web tier in the original case has much larger fluctuations than that in the CTP scheduling case, which validates that CTP scheduling actually reduces the overall waiting time of all transactions in the system. In fact the high peaks of response time in these two figures perfectly matches the high peaks of the number of queued requests in upper tiers as shown in Figure 10(a) and Figure 10(b).

Heuristic (ii): *We need to restrict the number of concurrent requests to avoid overhead caused by high concurrency in the bottleneck tier.*

Heuristic (ii) is illustrated by Figure 11. The hardware configuration is 1L/2L/1S/2L where the CJDBC server CPU is the bottleneck of the system. We choose DBconn24 and DBconn2 for each servlet in Tomcat; the CPU utilization and JVM GC time of the CJDBC server under different workloads are shown in Table 3.

Figure 11(a) and 11(b) show the approximately instant

[5]Heavy transactions are only negligibly penalized or not penalized at all as a result of SJF-based scheduling [10].



(a) Approximately instant # of concurrent requests in each tier (b) Approximately instant # of concurrent requests in each tier

(c) Approximately instant response time in Apache web tier (d) Approximately instant response time in Apache web tier

Figure 10: Response time stabilization by applying CTP scheduling in 1L/2L/1L configuration in WL 5800.

response time in the Apache web tier for the DBconn24 case and the DBconn2 case in workload 5600 respectively. These two figures show that the response time in the DBconn24 case present much larger fluctuations than that in the DBconn2 case. As shown in Table 3, the DBconn24 case in workload 5600 has significantly longer JVM GC and high CPU utilization in CJDBC than those in the DBconn2 case. This set of results clearly shows that higher concurrency in the bottleneck tier causes many more transient events such as JVM GC, which in turn cause more CPU overhead in the tier and lead to large end-to-end response time fluctuations.

We note that lower concurrency in the bottleneck tier is not always better; too low concurrency in the bottleneck tier may under-utilize the hardware resource in the tier and degrade the overall system performance. Interested readers can refer to [19] for more information.

Heuristic (iii): *We need to allocate enough amount of soft resources in front tiers (e.g., web tier) to buffer large fluctuations of concurrent requests and avoid TCP retransmission.*

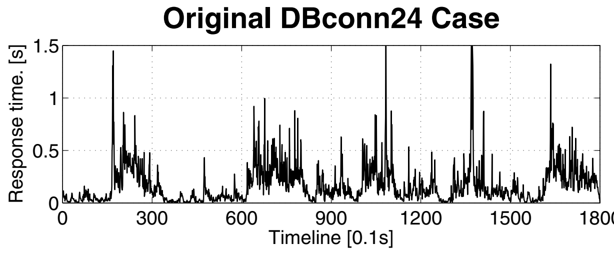
This heuristic is illustrated in the motivation case. Though the average concurrent requests over a long time window is low (see Figure 4(i)), the approximately instant concurrent requests may present high peaks that can be 10 times higher than the average (see Figure 4(h)) due to the impact of system environmental conditions discussed in this paper. Thus, allocating a large number of soft resources in front tiers is necessary to buffer such high peaks of concurrent requests and avoid TCP retransmission. A reasonable allocation should also be hardware dependent since soft resources consume hardware resources such as memory.

6. RELATED WORK

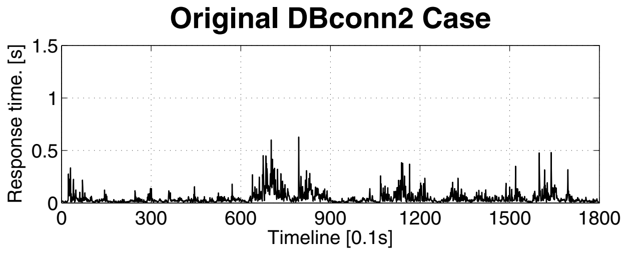
Autonomic self-scaling n-tier systems based on elastic workload in cloud for both good performance and resource efficiency has been studied intensively before [12, 15, 20, 21]. The main idea of these previous works is to propose adaptive control to manage application performance in cloud by combining service providers' SLA specifications (e.g., bounded response time) and virtual resource utilization thresholds. Based on the average of the monitored metrics (e.g., response time, CPU) over a period of time (a control interval), the controller of the system allocates necessary hardware resources to the bottleneck tier of the system once the target threshold is violated. However, how long a proper control interval should be is an open question and sometimes difficult to determine. As shown in this paper, the average of monitored metrics based on inappropriately long control intervals may blur the large performance fluctuations caused by factors such as bursty workload or JVM GC.

The performance impact of bursty workloads for the target n-tier system has been studied before. The authors in [6, 13] observed that while the system CPU utilization may be low at a coarse time granularity, it fluctuates significantly if observed at a finer time granularity, and such large fluctuation significantly impacts the n-tier system response time. Different from the previous works which mainly focus on bursty workload, we focus more on system aspects such as JVM GC, scheduling policy, and fluctuation amplification effects in n-tier systems. As shown in this paper, system response time presents large scale fluctuations due to these factors even under the moderately bursty workload.

Analytical models have been proposed for performance prediction and capacity planning of n-tier systems. Chen et al. [7] present a multi-station queuing network model with regression analysis to translate the service providers' SLA specifications to lower-level policies with the purpose of optimizing resource usage of an n-tier system. Thereska et al. [18] propose a queuing modeling architecture for clustered storage systems which constructs the model during the system design and continuously refines the model during operation for better accuracy due to the changes of system. Though these models have been shown to work well for particular domains, they are constrained by rigid assumptions such as normal/exponential distributed service times, disregard of some important factors inside the system which can cause significant fluctuations of both application level and system level metrics.



(a) Approximately instant response time in Apache web tier



(b) Approximately instant response time in Apache web tier

Figure 11: Response time stabilization by limiting bottleneck tier concurrency in 1L/2L/1S/2L config. in WL 5600.

7. CONCLUSIONS

We studied the large scale response time fluctuations of n-tier systems in high resource utilization using the n-tier benchmark RUBBoS. We found that the large scale response time fluctuations can be caused by some system environmental conditions such as L2 cache miss, JVM GC, and limitations of OS level scheduling policies in the system, in addition to the bursty workload from clients. We showed that because of the complex resource dependencies across tiers, a small response time fluctuation in a bottom tier can be amplified to front tiers and eventually to clients. To mitigate the large scale response time fluctuations, we evaluated three heuristics to stabilize the response time fluctuations while still achieving efficient resource utilization. Our work is an important contribution to design more effective autonomous self-scaling n-tier systems in cloud to achieve both good performance and resource efficiency under elastic workloads.

8. ACKNOWLEDGMENTS

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.

9. REFERENCES

[1] *Rice University Bulletin Board System.* "`http://jmob.ow2.org/rubbos.html`", 2004.

[2] *What is the average response time for displaying results.* "`http://support.google.com/mini/bin/answer.py?hl=en&answer=15796`", 2004.

[3] *Fujitsu SysViz: System Visualization.* "`http://www.google.com/patents?id=0pGRAAAAEBAJ&zoom=4&pg=PA1#v=onepage&q&f=false`", 2010.

[4] *Java SE 6 Performance White Paper.* `http://java.sun.com/performance/reference/whitepapers/6_performance.html`, 2010.

[5] T. Abdelzaher, Y. Diao, J. Hellerstein, C. Lu, and X. Zhu. Introduction to control theory and its application to computing systems. SIGMETRICS Tutorial, 2008.

[6] P. Bodik, A. Fox, M. J. Franklin, M. I. Jordan, and D. A. Patterson. Characterizing, modeling, and generating workload spikes for stateful services. SoCC'10.

[7] Y. Chen, S. Iyer, X. Liu, D. Milojicic, and A. Sahai. Translating service level objectives to lower level policies for multi-tier services. *Cluster Computing*, 2008.

[8] S. Cho and L. Jin. Managing distributed, shared l2 caches through os-level page allocation. MICRO'06.

[9] T. Forell, D. Milojicic, and V. Talwar. Cloud management: Challenges and opportunities. In *IPDPSW'11.*

[10] M. Harchol-Balter, B. Schroeder, N. Bansal, and M. Agrawal. Size-based scheduling to improve web performance. *ACM Trans. Comput. Syst.*, 2003.

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

[12] H. C. Lim, S. Babu, and J. S. Chase. Automated control for elastic storage. ICAC'10.

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

[14] K. S. Min Lee. Region scheduling: Efficiently using the cache architectures via page-level affinity. ASPLOS'12.

[15] P. Padala, K.-Y. Hou, K. G. Shin, X. Zhu, M. Uysal, Z. Wang, S. Singhal, and A. Merchant. Automated control of multiple virtualized resources. EuroSys'09.

[16] B. Schroeder, A. Wierman, and M. Harchol-Balter. Open versus closed: a cautionary tale. NSDI'06.

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

[18] E. Thereska and G. R. Ganger. Ironmodel: robust performance models in the wild. SIGMETRICS'08.

[19] 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. IPDPS'11.

[20] 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. ICDCS'11.

[21] X. Zhu, M. Uysal, Z. Wang, S. Singhal, A. Merchant, P. Padala, and K. Shin. What does control theory bring to systems research? *SIGOPS Oper. Syst. Rev.*, 2009.