The Impact of Soft Resource Allocation on n-Tier Application Scalability

Qingyang Wang, Simon Malkowski, Deepal Jayasinghe, Pengcheng Xiong, Calton Pu
College of Computing
Georgia Institute of Technology
Atlanta, USA
{qywang, zmon, deepal, pxiong3, calton}@cc.gatech.edu

Yasuhiko Kanemasa, Motoyuki Kawaba, Lilian Harada
Research Center for Cloud Computing
Fujitsu Laboratories Ltd
Kawasaki, Japan
{kanemasa, kawaba, harada.lilian}@jp.fujitsu.com

***Abstract*—Good performance and efficiency, in terms of high quality of service and resource utilization for example, are important goals in a cloud environment. Through extensive measurements of an n-tier application benchmark (RUBBoS), we show that overall system performance is surprisingly sensitive to appropriate allocation of soft resources (e.g., server thread pool size). Inappropriate soft resource allocation can quickly degrade overall application performance significantly. Concretely, both under-allocation and over-allocation of thread pool can lead to bottlenecks in other resources because of non-trivial dependencies. We have observed some non-obvious phenomena due to these correlated bottlenecks. For instance, the number of threads in the Apache web server can limit the total useful throughput, causing the CPU utilization of the C-JDBC clustering middleware to decrease as the workload increases. We provide a practical iterative solution approach to this challenge through an algorithmic combination of operational queuing laws and measurement data. Our results show that soft resource allocation plays a central role in the performance scalability of complex systems such as n-tier applications in cloud environments.**

***Keywords*-bottleneck, configuration, n-tier, parallel processing, scalability, and soft resource**

I. INTRODUCTION

One of the major advantages of consolidated data centers and cloud environments is scalability: the ability to allocate more resources to an application if needed. For occasional users that run an application only a few times, scalability often compensates for lack of efficiency since achieving high utilization rates is unimportant for them. However, for long term users who run their applications often or continuously (e.g., in electronic commerce), a high level of utilization and low operational costs are important requirements. Otherwise, high operational costs due to inefficiencies in a cloud quickly outweigh the savings from avoiding the capital expenditure of setting up a dedicated cluster. Therefore, a shared cloud environment needs both scalability and efficient utilization.

Efficient utilization of cloud resources requires intelligent mapping of system resources to applications that need them. This is a significant challenge since internet-scale applications often have elastic workloads with peak load several times the steady state. In addition, such applications also have Quality of Service (QoS) requirements, often stated in terms of Service Level Agreement (SLA) guarantees such as bounded response time. At the hardware level, intelligent mappings of resources have been previously studied as offline configuration plans [1] and dynamic adaptation policies for run-time adjustments [2].

In this paper, we show that achieving efficient utilization of cloud resources when running large scale n-tier applications requires a unified exploration of each system layer, including both hardware and software. This is primarily due to the complex dependencies among software and hardware resources from each tier. Concretely, we developed tools to monitor the usage of soft resources such as application server thread pool and its database (DB) connection pool. Then we analyzed the relationship among throughput, response time, hardware resource usage (CPU, memory, I/O), and soft resource usage (thread/DB connection pool) by changing the size of each thread/DB connection pool in web and application servers.

The first contribution of the paper is a quantitative evaluation (based on measurements of the RUBBoS benchmark [3]) analyzing the impact of soft resource allocation on application performance for given hardware configurations. For instance, we show that sub-optimal allocation of soft resources such as application server thread pool and its database connection pool can easily degrade the application performance between 28% and 90%, depending on the SLA specifications (see Figure 2).

The second contribution of the paper is an impact analysis of two naive soft resource allocation strategies. On one side, a conservative (low) allocation of soft resources (e.g., too small thread pool) often creates software bottlenecks that limit overall system throughput, even though no hardware resources are saturated. On the other side, a liberal (high) allocation of soft resources often wastes hardware resources such as CPU and memory. This degrades application performance at high utilization levels. For example, in our experiments Java Virtual Machine (JVM) garbage collection can consume up to 9% of CPU when the number of threads reaches several hundreds, reducing the achievable throughput (and lengthening the response time) when CPU is near saturation.

The third contribution of the paper is a practical algorithm for the allocation of soft resources. Our measurements show that an optimal allocation (within our data set) for one hardware configuration is often sub-optimal for other hardware configurations. Consequently, static rule-of-thumb

allocations will be almost always sub-optimal in the presence of volatile workloads. Our adaptive algorithm for the proper soft resource allocation is analogous and complementary to adaptive hardware resource configuration algorithms [4] [5].

In general, our results strongly suggest that in studies on the efficient utilization of clouds, soft resources should be considered integral components in determining overall application performance due to the dependencies linking hardware and software resources. In fact, complex applications can only become truly scalable if soft resources are treated as first class citizens (analogous to hardware resources) during the principal component analysis of the overall system performance.

The rest of the paper is organized as follows. Section II describes the impact of soft resource allocation on application performance. Section III summarizes the utilization achieved by various allocation strategies through measured results with explanations. Section IV describes our proposed allocation algorithm in detail. Section V summarizes related work and Section VI concludes the paper.

II. Background and Motivation

A. Background Information

1) Soft Resources in n-Tier Systems: Hardware resources such as CPU, memory, disk, and network are well defined components in performance evaluation studies. We use the term *soft resources* to refer to system software components that use hardware or synchronize the use of hardware. For example, *threads* use CPU and *TCP connections* use network I/O. Expanding this definition, we also use the term soft resources to refer to components that use (or synchronize the use of) soft resources as well as a combination of hardware and soft resources. For example, various locks synchronize the access to shared data structures or resources. Usually, soft resources are created to facilitate the sharing of hardware resources in order to increase hardware utilization. For example, threads facilitate multiprogramming to achieve a higher utilization of the CPU. Consequently, soft resources are critical path components that contribute to determining the level of hardware resource utilization achievable in the system.

In this paper, we study the role of soft resources in determining n-tier application performance. The identification of soft resources as explicit components in the critical path of system execution is due to the long invocation chain of requests in an n-tier system. Requests that originate from a client machine arrive at the web server, which distributes it among the application servers, which in turn ask the database servers to carry out the query. The dependencies among the servers are in the critical path and maintained by soft resources.

2) Experimental Environment: We run an n-tier benchmark (RUBBoS) on our private cluster testbed. We summarize the benchmark and experimental testbed in this section.

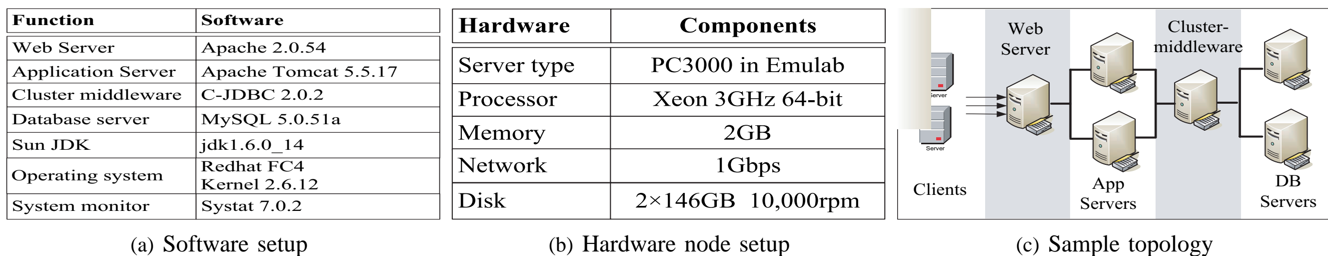
RUBBoS is a standard n-tier benchmark based on bulletin board applications such as Slashdot. RUBBoS has been widely used in numerous research efforts due to its real production system significance. Figure 1 outlines the choices of software components, hardware node, and a sample network topology used in our experiments.

The RUBBoS benchmark application can be implemented as three-tier (web server, application server, and database server) or four-tier (addition of clustering middleware such as C-JDBC [6]) system. The workload consists of 24 different interactions such as "view story". The benchmark includes two kinds of workload modes: browsing-only and read/write interaction mixes. In our experiments each experiment trial consists of an 8 minute ramp-up, a 12-minute runtime, and a 30-second ramp-down. Performance measurements (e.g., CPU utilization) are taken during the runtime period using SysStat at one second granularity. We use the functionality provided by JVM to monitor thread status in Java applications. To conveniently monitor the utilization of the DB connection pool, we made slight changes to the RUBBoS benchmark: all servlets share a global DB connection pool instead of using an individual connection pool for each servlet. We also modified Apache server source code to record its detailed internal processing time.

The experiments used in this paper were run in the Emulab testbed [7]. Figure 1(b) contains a summary of the hardware used in our experiments. The experiments were carried out by allocating a dedicated physical node to each server. 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. A sample topology of experiments with two clients, one web server, two application servers, one clustering middleware server, two database servers (i.e., 1/2/1/2) is shown in Figure 1(c). In our experiments, we focus on how the allocation of soft resources such as threads and DB connections affects n-tier system performance. Thus, we change the allocation of those soft resources by changing thread pool size in Apache servers, the thread pool and DB connection pool size in Tomcat servers. For each hardware provisioning $\#W/\#A/\#C/\#D$, we use $\#W_T$-$\#A_T$-$\#A_C$ to represent the thread pool size in web server, the thread pool size in application server, and the DB connection pool size in application server. For example, the hardware provisioning can be 1/2/1/2. The corresponding soft resource allocation $\#W_T$-$\#A_T$-$\#A_C$ can be 200-100-100, which means the thread pool size in a web server, the thread pool size and the DB connection pool size in each application server is 200, 100, 100, respectively. The allocation of other soft resources are fixed in order to limit the exponential experiment space. Regarding the configuration of Apache, we chose worker MPM as its multi processing module. It should be mentioned that we turned off the keepAlive function because RUBBoS workload only has a few consecutive http requests.

B. Performance Requirements Specified by SLA

In applications such as e-commerce, response time of requests is critical for users. According to Aberdeen's June 2008 report [8], response time longer than 5 seconds would likely make 10% of potential customers navigate away. In this case, only requests with a fast response time have positive impact to



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	Redhat FC4 Kernel 2.6.12
System monitor	Systat 7.0.2

Hardware	Components
Server type	PC3000 in Emulab
Processor	Xeon 3GHz 64-bit
Memory	2GB
Network	1Gbps
Disk	2×146GB 10,000rpm

(a) Software setup (b) Hardware node setup (c) Sample topology

Fig. 1: Details of the experimental setup on the Emulab cluster

service providers' business. In shared infrastructures such as cloud environments, service level agreements (SLAs) are commonly used for specifying desirable response times, typically in the one to two seconds range. The SLA document usually contains the service provider's revenue model, determining the earnings of the provider for SLA compliance (when request response times are within the limit) as well as the penalties in case of failure (when response times exceed the limit). The provider's revenue is the sum of all earnings minus all penalties.

Different service providers may have different SLA models, which would result in different conclusions from a performance evaluation based on the system resource monitoring data. A generic SLA model has been studied in our previous work [1]. In this paper we use a simplified SLA model to illustrate the revenue tradeoffs between throughput and response time (i.e., the system performance). Even with this simplified model, our results clearly show that increasing throughput (and utilization) without other considerations leads to significant drops in provider revenue through high response times. A detailed study of the influence of different revenue-based SLA models is beyond the scope of this paper and a the subject of our future research.

For our simplified SLA model we set a single threshold for the response time of requests (e.g., 1 second). Requests with response time equal or below the threshold satisfy the SLA. We call the throughput of these requests **goodput**. Requests with response time higher than the threshold violate the SLA, and the throughput of these requests is called badput. We note that the sum of goodput and badput amounts to the traditional definition of throughput.

A classic performance model that only considers the throughput as a whole may be appropriate for a batch-oriented workloads. However, for interactive applications such as e-commerce the situation is more complex because the response time increases significantly when a system resource reaches full utilization. By refining our throughput model to consider both goodput and badput, we are able to quantify user-perceived response time, which yields a more realistic provider revenue analysis.

C. Degradation with Simplified SLA Model

In this section, we apply the simplified SLA model to show the magnitude of performance variations when soft resource allocations are inappropriate. The goal of this section is to illustrate the importance of the problem. The explanations will be described in Section III.

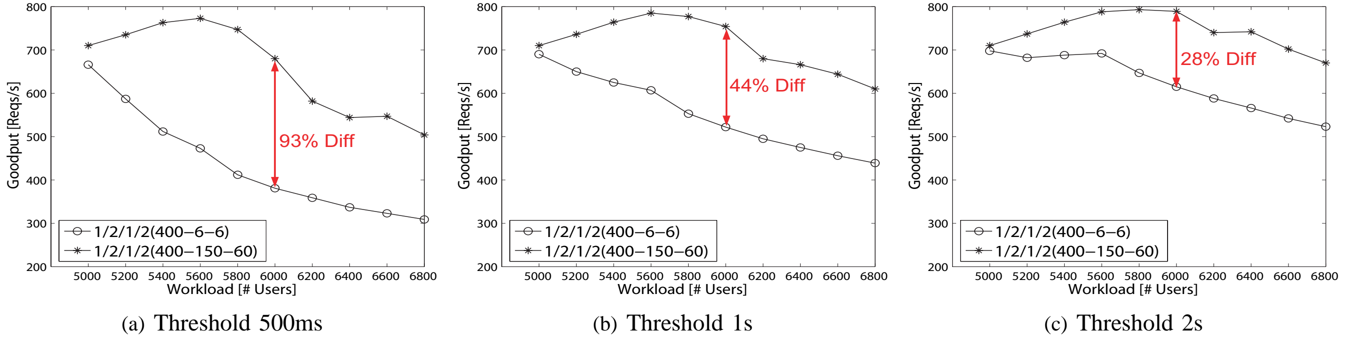
1) Impact of Under-Allocation: Figures 2(a), 2(b), and 2(c) compare the goodput of the same hardware configuration (1/2/1/2) and two different soft resource allocations (400-150-60 and 400-6-6). The range of workload chosen (5000 to 6800) captures the knee when the overall throughput curve stops growing. All three figures show that the goodput of 400-6-6 starts to decrease (before 5000) much earlier than the case 400-150-60 (after 5600). An analysis of hardware resource utilization (omitted due to space constraints) shows that no hardware resources are saturated in the 400-6-6 case. This confirms that the soft resource allocation of application server cause the bottleneck (6-6 thread pool and DB connection pool).

An experienced reader may immediately question the wisdom of attempting the 400-6-6 allocation, which appears to be an "obviously low" number compared to the 400-150-60 "intuitive" choice of practitioners. As it happens, the situation is more complex than it appears. Section II-C2 will show that 400-6-6 allocation is near optimal for hardware configuration (1/4/1/4). The reasons for these differences will be explained in Section III.

Applying the simplified SLA model, the three graphs show the magnitude of the goodput difference between the two soft resource allocations. At the workload of 6000 users, the goodput of 400-150-60 allocation is about 28% higher than the 400-6-6 allocation under the threshold of 2 seconds, 44% higher under the 1-second threshold, and 93% higher under the half-second threshold. Therefore, for the same overall throughput there may be a significant difference in goodput, depending on the SLA requirements.

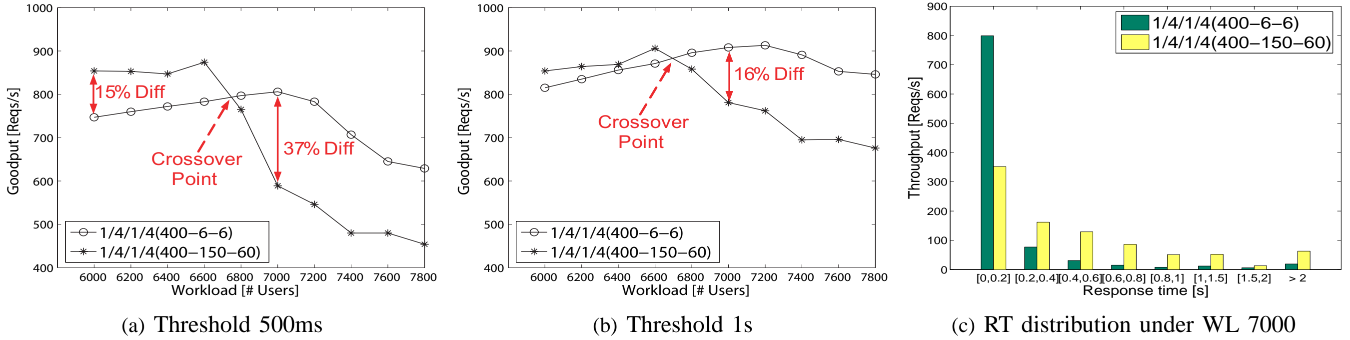
2) Impact of Over-Allocation: Figures 3(a) and 3(b) show the performance degradation of the same thread pool allocations of Section II-C1 for the 1/4/1/4 hardware configuration. The figures show a crossover point. Before the crossover, 400-150-60 has better performance due to better hardware resource utilization achieved. After the crossover, 400-6-6 is better due to smaller CPU consumption of the smaller thread pool. This will be explained better in Section III. It may be unexpected that Figure 3 shows the non-intuitive choice of 400-6-6 is better when nearing saturation, which is just the opposite of Figure 2.

In fact the performance difference between 400-6-6 and 400-150-60 showed in Figures 3(a) and 3(b) can be much



(a) Threshold 500ms (b) Threshold 1s (c) Threshold 2s

Fig. 2: The goodput comparison of the same software source allocations. In these figures, each line represents the performance of a specific combination of hardware configuration and soft resource allocation. The first set of four numbers separated by dash, denotes hardware configuration. For example, 1/2/1/2 means one web server (apache), two application servers (tomcat), one database clustering middleware (C-JDBC), and two database servers (MySQL). The following set of 3 numbers, separated by hyphens, denotes software configuration. For example, 400-150-60 refers to the size of thread pool (400) in one web server, the size of thread pool (150) in one application server, and the size of DB connection pool (60) in the same application server. This setting (400-150-60) is considered a good choice by practitioners from industry



(a) Threshold 500ms (b) Threshold 1s (c) RT distribution under WL 7000

Fig. 3: Performance degradation due to over-allocation of thread pool with 1/4/1/4 hardware configuration

more significant when a detailed response time analysis is applied. For example, as shown in 3(c), the 400-6-6 allocation has 130% higher goodput than the case 400-150-60 using 0.2 seconds as a threshold.

In general, SLA models correlate economic goals with technical performance data to derive intuitive domain insights at the real quality of services provided by service providers. In order to facilitate our discussion with our experimental results, unless stated otherwise, we apply the simplified SLA model with 2-second threshold (a commonly used value for many web applications) in the application performance evaluation.

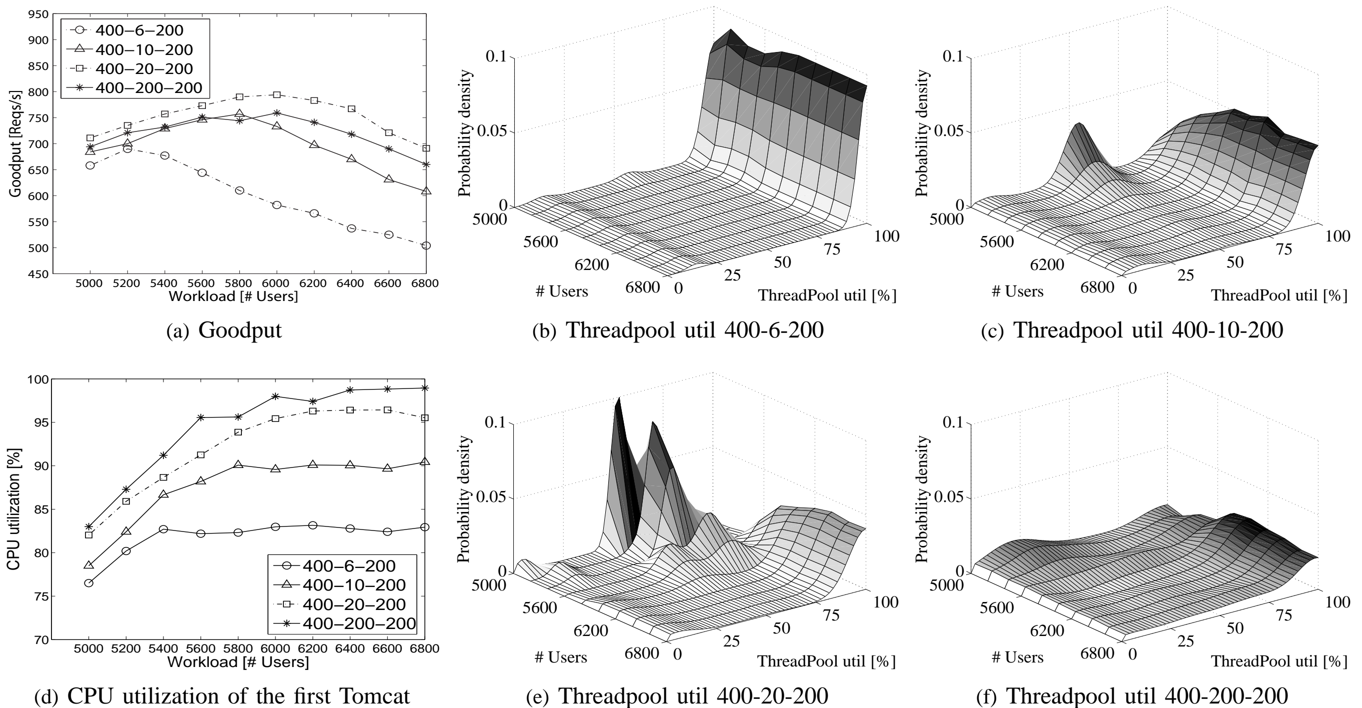
In the following Section III we will explain the reasons for the different performance results shown so far.

III. Sensitivity Analysis of Soft Resource Allocation Strategies

In this section we analyze the influence of soft resource allocation on the overall application performance in terms of measured goodput. We observe some similarities and differences between soft resources and hardware resources. Section III-A describes an example of similarities, when scarce soft resources become the system bottleneck. This is a problem solvable by adding more soft resource capacity, analogous to scarce hardware resources. Section III-B describes an example of differences, when unused soft resources consume too much of another critical resource (e.g., CPU), causing the overall performance to decay. This is a problem solvable by reducing the unused soft resource capacity, which reduces the consumption of the critical resource and improves overall goodput. Section III-C shows another interesting example of similarities; hardware resources can be under-utilized by non-obvious low soft resource allocation in front tiers. In fact, we show that high allocation of soft resources in front tiers acts as a buffer that stabilizes bursty request flows and provides a more even workload distribution to the back-end.

A. Analysis of Soft Resource Under-Allocation

The first naive soft resource allocation strategy we consider is straight-forward resource minimization, i.e., choose a small capacity to not overload the system. Although this strategy minimizes the overhead incurred by soft resources, it may significantly hamper workload propagation. When there are too few units of a soft resource, they become a bottleneck and the situation is analogous to the saturation of hardware resources. A soft resource bottleneck may cause the rest of the system (e.g., all hardware resources) to become under-utilized. Consequently, adding more hardware does not improve overall performance (e.g., throughput).



(a) Goodput (b) Threadpool util 400-6-200 (c) Threadpool util 400-10-200

(d) CPU utilization of the first Tomcat (e) Threadpool util 400-20-200 (f) Threadpool util 400-200-200

Fig. 4: Performance degradation due to under-allocation of thread pool with 1/2/1/2 hardware configuration

We use the thread pool in Tomcat to illustrate the problem of under-allocation of soft resources. The experiments described here have fixed numbers of threads in the Apache server (400), and the number of DB connections is fixed to 200 in Tomcat servers. Therefore, these soft resources never become the system bottlenecks. The only degree of freedom in the experiments is the size of the Tomcat thread pool, ranging between 6 to 200 threads. The hardware configuration is chosen as 1/2/1/2. Figure 4(a) shows the goodput of increasing allocations of threads in Tomcat. The goodput increases as the size of the thread pool increases from 6 to 20. For example, at a workload of 6000, the goodput of the 400-20-200 configuration is about 40% higher than the goodput of the 400-6-200 configuration. This is due to the better overall hardware resource utilization achieved by 20 threads.

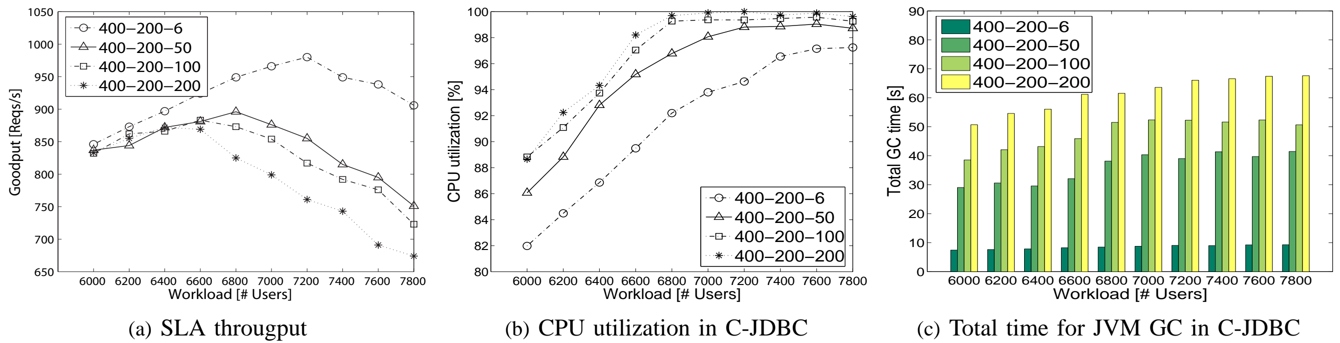
Figure 4(d) shows the average CPU usage of the first Tomcat server. Note that given the load balancing provided by the Apache web server, all the Tomcat servers have similar behavior. Thus, it suffices to show a single representative graph for this tier. The small thread pool size shows a corresponding lower CPU utilization. For example, at workload 6000, the Tomcat CPU utilization of configuration 400-6-200 is about 82% while the utilization exceeds 95% for configurations 400-20-200 and 400-200-200. It is clear that in small thread pool allocations all available threads in Tomcat are either busy processing the existing requests or busy waiting for response from the lower tier (i.e., C-JDBC server). Consequently, the Tomcat servers become idle and the throughput cannot increase even when all hardware resources are idle.

We use a resource utilization density graph to analyze this resource utilization scenario more precisely [9]. Figures 4(b), 4(c), 4(e), and 4(f) show for which thread pool sizes this soft resource becomes the system bottleneck. The analysis of the figures shows that thread pool size 6 saturates before the workload of 5000. In contrast, pool size 10 saturates at about 5600, and pool size 20 saturates at about 6000. We note that the system performance symptoms of thread pool under-allocation are similar to the symptoms of hardware resource saturation. Hence, it is necessary to monitor the thread pool utilization along with hardware resource utilization in order to find this particular bottleneck. If only hardware utilizations were monitored, this bottleneck would remain hidden beneath the underutilized hardware resources.

Figure 4(a) and 4(d) show another interesting phenomenon: the highest goodput achieved by the thread pool size of 200 is lower than the goodput achieved by thread pool size 20. This observation shows that monotonically increasing the thread pool size eventually leads to sub-optimal allocations. This is an example of differences between soft resources and hardware resources, which are the subject of the next section.

B. Over-Allocation of Soft Resources

The second naive soft resource allocation strategy we consider is straight-forward resource maximization, i.e., choose a large capacity to enable full hardware utilization system. This strategy clearly illustrates some of the differences between soft and hardware resources. Unlike hardware resources, which cannot consume other resources when idle, soft resources actually may consume other system resources (e.g., CPU and memory), regardless of whether they are being used or not. Usually, the maintenance costs of soft resources are considered



(a) SLA througput (b) CPU utilization in C-JDBC (c) Total time for JVM GC in C-JDBC

Fig. 5: Performance degradation due to over-allocation of DB connection pool with 1/4/1/4 hardware configuration

small, so the maximization strategy is considered reasonable as long as there are no hardware bottlenecks in the system.

Our measurements show that when the system approaches saturation, over-allocation of soft resources can degrade system performance significantly. For a thread pool size of 200, Figures 4(a) and 4(d) illustrate an example of such a degradation. We keep the other parameters fixed and solely vary the size of the database connection pool in the Tomcat server (from 6 to 200) to illustrate the phenomenon. We note that in these experiments, each time the Tomcat server establishes a connection to the C-JDBC server, which balances the load among the database servers, a thread is created by C-JDBC to route the SQL query to a MySQL database server. The MySQL server also creates a thread for actual processing. Consequently, one database connection implies one thread in the C-JDBC and one thread in MySQL server.

Figure 5(a) shows the system goodput for database connection pool sizes of 10, 50, 100, and 200 for a workload between 6000 to 7800. This figure show that the lowest allocation 400-200-6 achieves the best performance. Under workload 7800, the goodput of the 400-200-6 configuration is about 34% higher than the throughput of the 400-200-200 configuration. Figure 5(b) shows the average CPU utilization of the C-JDBC server from workload 6000 to 7800. The highest goodput achieved by each configuration is shown in Figure 5(a). The goodput corresponds to about 95% of C-JDBC server CPU utilization in Figure 5(b), which suggest that C-JDBC server CPU is the system bottleneck.

A careful analysis of C-JDBC server CPU utilization shows a super-linear increase for higher numbers of DB connections in Tomcat as workload increases from 6000 to 7800. At workload 7800, the C-JDBC server CPU utilization of the 400-200-6 configuration is the lowest while its goodput is the highest. After a wide range search of candidate system software components, we found that JVM garbage collection played a major role degrading the C-JDBC server efficiency as the number of DB connections in Tomcat servers increased.

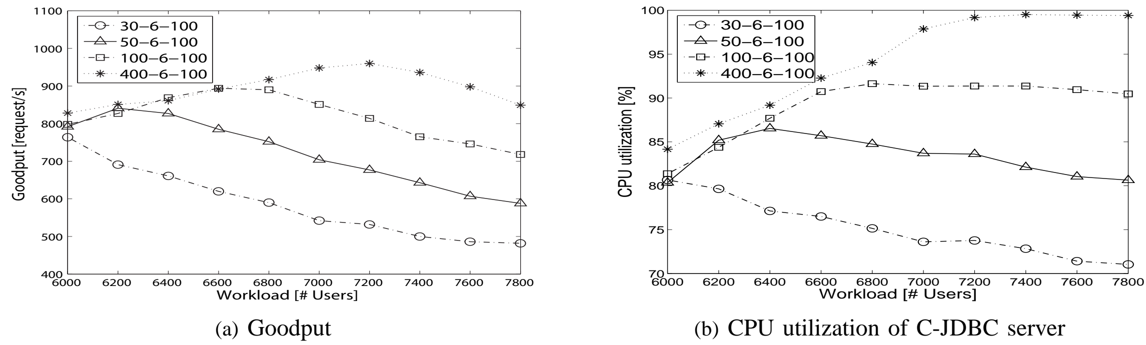
The JVM garbage collection process may affect the system goodput in two ways. First, since the C-JDBC server CPU is the primary bottleneck, the CPU time used by the garbage collector cannot be used for request processing. We measured the time used by the JVM garbage collector directly. Figure 5(c) shows the total time for JVM garbage collection on the C-JDBC server. During a 12-minute experiment, at workload 7800, the C-JDBC server's JVM garbage collector consumes nearly 70 seconds (9% of total) for the 400-200-200 configuration, compared to less than 10 seconds (about 1% of total) for the 400-200-6 configuration. Second, the JVM uses a synchronous garbage collector and it waits during the garbage collection period, only starting the processing requests after the garbage collection is finished [10]. This delay significantly lengthens the pending query response time—an important component of the goodput.

C. *Buffering Effect of Soft Resource*

In this subsection we show that high allocation of soft resources in front tiers of an n-tier system sometimes is necessary to achieve good performance. Unlike the over-utilization case as introduced in Section III-B, high allocation of soft resources in front tiers (e.g., Apache server) may function as a request buffer, which stabilizes the requests sent to the back-end tiers under high workload and improves the system performance.

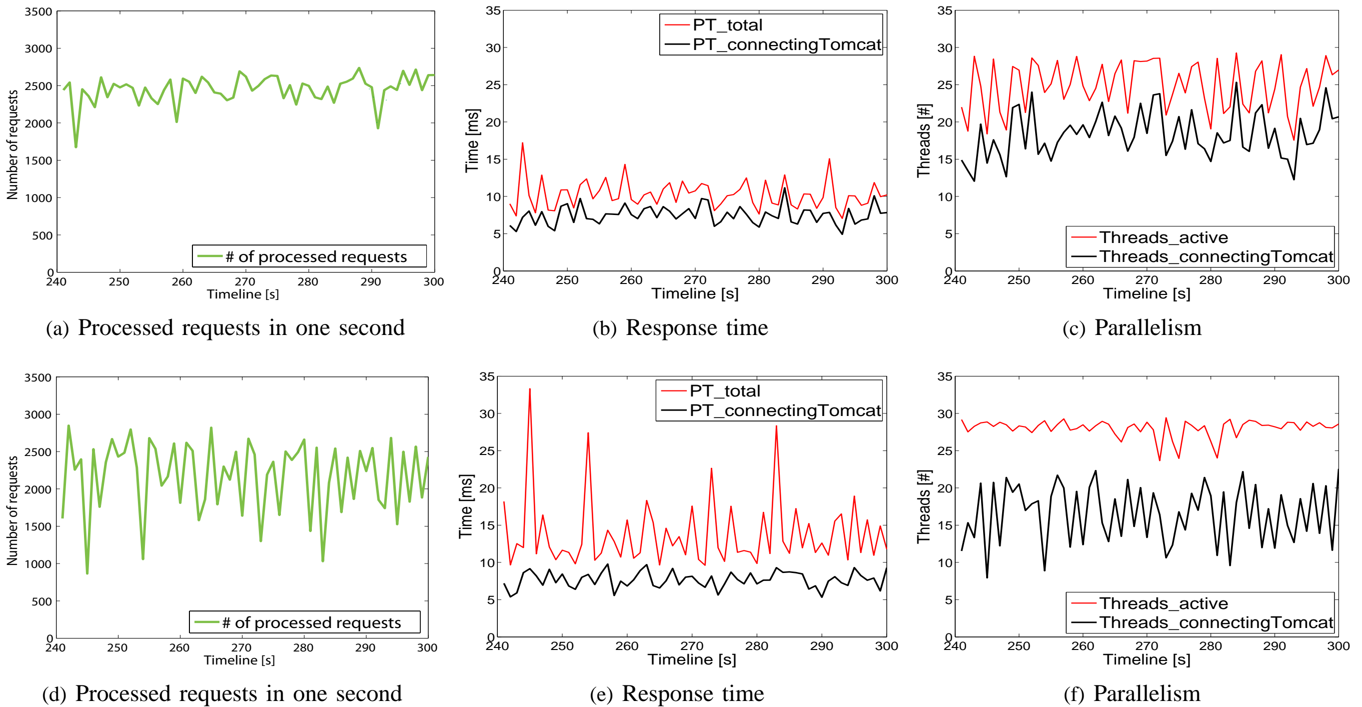
We use the thread pool in Apache server to illustrate the buffering effect phenomenon. The experiments described here have a fixed number of threads (6) and DB connections (200) in Tomcat server. The size of the Apache thread pool is increased from 30 to 400. The hardware configuration is 1/4/1/4. Figure 6(a) shows the system goodput for Apache thread pool sizes of 30, 50, 100, 400 for a workload between 6000 and 7800. The goodput increases as the allocation of threads in the Apache server increases. Under workload 7800, the goodput of the 400-6-100 configuration is 76% higher than the goodput of the 30-6-100 configuration. Intuitively, one may be misled to believe that the latter example is an under-allocation scenario analogous to the scenario in Section III-A. In other words, it appears that a small thread pool in the Apache server limits the number of concurrent requests going through the back-end tiers, which leads to hardware resource under-utilization. However, the following two phenomena show that this scenario is, in fact, significantly more complex.

First, Figure 6(b) shows that the C-JDBC CPU utilization continuously decreases, in both the 30-6-100 and the 50-6-100 configurations after the workload exceeds a certain



(a) Goodput (b) CPU utilization of C-JDBC server

Fig. 6: Performance degradation due to small buffer of threads in Apache server with 1/4/1/4 hardware configuration



(a) Processed requests in one second (b) Response time (c) Parallelism

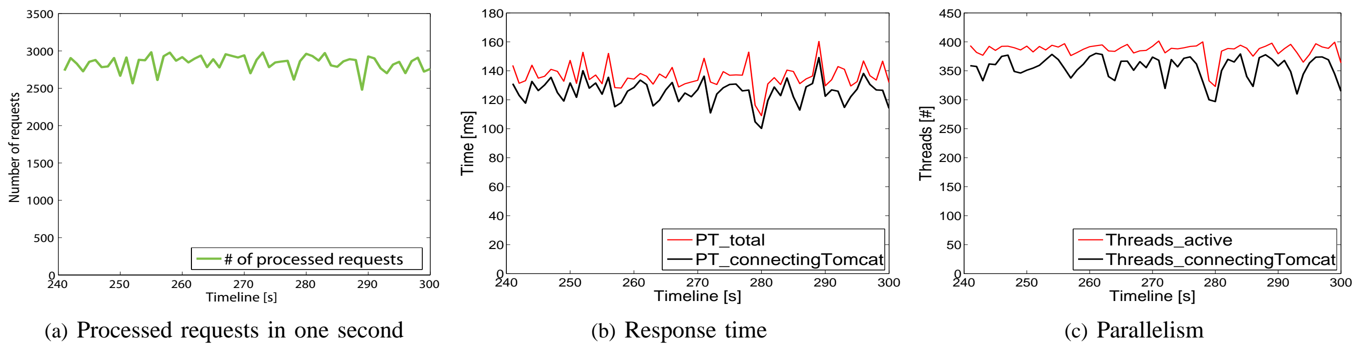
(d) Processed requests in one second (e) Response time (f) Parallelism

Fig. 7: Performance degradation under high workload due to small buffer of threads in Apache server in 30-6-100 allocation. Subfigures (a)–(c) show the results under workload 6000 while Subfigures (d)–(f) under workload 7400.

point. In the case of 30-6-100, the CPU utilization of the C-JDBC server under workload 7800 is about 10% lower than under workload 6000. This observation does not match the intuitively expected characteristic that growing workload implies monotonic growth of hardware resource utilization.

Second, Figure 6(a) shows that 30 threads in the Apache server cause worse performance than 400 threads. We have to point out that the size of the thread pool in a Tomcat server is fixed to 6 in all configurations, which means that the Apache server solely requires 24 threads for sending concurrent requests to the four Tomcat servers. Although each request sent to a RUBBoS Servlet is followed by two consecutive requests for static content (e.g., image file RUBBoS_logo.jpg), such requests can be handled by a small number of threads because all small files with static content are cached in memory. Therefore, it is hard to believe that the thread pool in Apache server is too small, even in the 30-6-100 configuration.

A careful analysis of the Apache server runtime status shows that both aforementioned phenomena are the result of a too small request buffer (i.e., too low allocation of soft resources in the Apache server). Figure 7 shows the characteristics of the Apache server with 30 threads under workload 6000 and 7400 during one minute of the experiment runtime. Figures 7(a) and 7(d) show the timeline graphs of the number of requests that the Apache server processed during each second. Figures 7(b) and 7(e) show the average busy time of a worker thread in Apache, and the average processing time of HTTP contents. During processing time an Apache worker thread is either occupying a Tomcat connection or waiting to obtain a Tomcat connection from the connection pool in order to interact with the lower tier. Figures 7(c) and 7(f) show the total number of active worker threads and the number



(a) Processed requests in one second

(b) Response time

(c) Parallelism

Fig. 8: Performance improvement due to large thread buffer in Apache server in 400-6-100 allocation under workload 7400

of worker threads interacting or waiting to interact with the Tomcat tier. The cause of the performance difference between the two workloads becomes evident through the comparison of Figures 7(b) and 7(e). Under workload 7400, the average processing time of a worker thread has many significant peaks even though the actual average time spend in an interaction state that requires a Tomcat connection is largely similar to the 6000 workload scenario. Although an Apache worker thread needs to perform some additional tasks apart from processing HTTP contents (e.g., maintain TCP connections), these additional task generally only take a short time under low workload. Under workload 7400, the main contributor of the high busy time peaks is the wait time for a FIN reply from a client closing a TCP connection. Under high workload, this wait time is much larger, which reduces the throughput of the Apache server dramatically. This is shown in Figure 7(d). In addition, long wait times for FIN replies result in the number of active worker threads almost reaching the upper boundary of 30 while the number of worker threads that actually interact with the Tomcat tier is far less than 30. This is shown in Figure 7(f). The drop in the worker threads interacting with the Tomcat tier results in the reduced number of requests pushed to the lower tiers; therefore the CPU utilization of the C-JDBC server reduces under higher workload.

In order to explain why a similar characteristic does not appear in the 400-6-100 configuration scenario, the graphs in Figure 8 show a similar analysis as in Figure 7 for the 400-6-100 configuration under workload 7400. While the previously introduced increase of wait time for a FIN reply also occurs in this scenario (see Figure 8(b)), the average processing time in this case is much larger than in the 30-6-100 case. This is due to the high number of active threads, which results in long waiting times for obtaining one of 24 Tomcat connections. Therefore, the relative waiting time for a FIN reply is much lower compared to the previous scenario. This results in a relatively stable high number of worker threads interacting (i.e., occupying or waiting a connection) with the Tomcat tier. As shown in Figure 8(c), the number of worker threads interacting with the Tomcat tier is always much higher than 24. Therefore, the Apache server can push a much larger amount of workload to lower tiers, which leads to high throughput as shown in Figure 8(a).

IV. Determining Soft Resource Allocations

In the previous section, we showed that both under-allocation and over-allocation of soft-resources may cause performance degradation. In this section we will outline a practical algorithm to choose a "Goldilocks" soft resource allocation for each hardware configuration.

A. *Challenges of Soft Resource Allocation*

The discussion so far can be summarized as following. Too-low soft resource allocations may cause under-utilization of other resources. Too-high soft resource allocations may waste critical system resources. Therefore, a proper allocation range needs to be found that maximizes the utilization of the system resources in the critical application execution path. However, finding a good allocation range is non-trivial due to the following three challenges:

1) The good soft resource allocation depends on location of the critical hardware resource, which can be quite different under different hardware configurations. The critical system resources are the Tomcat CPU in 1/2/1/2 configuration (Section III-A) and the C-JDBC CPU in 1/4/1/4 configuration (Section III-B), for example.
2) Typically, the system performance is not sensitive to over-allocation of soft resources until some critical hardware resource is close to saturation.
3) The state space for a soft resource allocation is usually very large (often unlimited). Thus, finding a good allocation by running experiments exhaustively is impractical.

B. *Soft Resource Allocation Algorithm*

Our algorithm is based on the following three assumptions:

- The system has a single hardware resource bottleneck.
- Our monitoring tools are able to monitor the bottlenecked hardware resource when the system is saturated.
- Each individual server response time for every request is logged.

The first assumption excludes the complex multi-bottleneck cases. In a multi-bottleneck scenario the saturation of hardware resources may oscillate among multiple servers locating in different tiers [9]. In these complex cases, our algorithm may fail to determine the critical hardware resource since no single hardware resource is fully utilized when the system

is saturated. The algorithmic resource allocation in multi-bottleneck cases is highly challenging and will require additional research. The second and third assumptions assume that we have properly instrumented the system with monitoring tools such as SysStat and Log4j.

The algorithm consists of the following three steps:

1) *Exposing the critical hardware resource in the system.* This step examines the hardware resource that will saturate earliest in the system. Such hardware resources are critical because their saturation limits the entire system performance.
2) *Inferring a good allocation of local soft resources.* This step allocates soft resources in the server that utilize the critical hardware resource directly. The critical hardware resource will be heavily utilized based on the associated soft resources if they are over-allocated.
3) *Calculating a good allocation of other soft resources.* This step allocates soft resources to tiers other than the tier where the critical hardware resource resides. The front tiers (e.g., Apache server) should provide enough soft resources to act as a buffer that stabilizes the request flow to the critical tier (see Section III-C). The back-end tiers need to provide enough soft resources to avoid request congestion in the critical tier.

Algorithm 1 shows the pseudo-code for the three steps, which are explained in more detail in the following.

1) Exposing the critical hardware resource: This part of the algorithm exposes the critical hardware resource by increasing the workload step by step until throughput saturates. The initial soft resource allocation and hardware provisioning are S_0 and H_0, respectively. Function $RunExperiment(H, S, workload)$ initiates an experiment with input hardware/software configuration at specific $workload$, and at the same time, all hardware resource and soft resource utilizations are monitored. During this process saturated hardware and soft resources are recorded in B_h and S_h, respectively. If B_h is not empty, the critical hardware resource has been successfully exposed and returned. If S_h is not empty, the system hardware resources are under-utilized due to the scarcity of some soft resource(s). In this case, the soft resource allocations are double, and the experiment is repeated. If both B_h and S_h are empty, the workload offered to the system is insufficient and needs to be increased by one workload step before a new iteration of the experiment.

It is possible that none of the hardware resources we observed are fully utilized when system is saturated. This complex case may be due to multi-bottlenecks [9] in the system or limitation of our monitoring tools. Dealing with such complex case is one of our on-going research.

2) Inferring a good allocation of local soft resources: The InferMinConncurentJobs procedure infers a good allocation of soft resources in the critical server (the server with the critical hardware resource) based on the minimum number of concurrent jobs inside the server that can saturate the critical hardware resource. Intuitively, all hardware resources have their limitation; the critical hardware resource will saturate the earliest if the number of the concurrent jobs inside the server is too high. Each job inside a server needs to consume certain hardware resources such as CPU, memory, or disk I/O. In practice each job in the server is delegated to one processing thread. The optimal number of threads in the critical server should be equal to the minimum number of concurrent jobs that can saturate the critical hardware resource. In this case, the critical hardware resource will neither be under-utilized nor over-utilized.

The average number of jobs inside a server can be inferred from the average throughput and response time of the server by applying Little's law. Throughput and response time of a sever can be obtained by checking the server log. Therefore, solely the minimum workload saturating the system (caused by the critical hardware resource saturation) needs to be determined.

In order to approximate the exact workload that can saturate the critical hardware resource, the procedure uses a statistical intervention analysis [11] on the SLO-satisfaction of a system. The main idea of such analysis is to evaluate the stability

Algorithm 1: Pseudo-code for the allocation algorithm

```
1  procedure FindCriticalResource
2  workload = step, TP_curr = 0, TP_max = -1;
3  S = S_0, H = H_0;
4  while TP_curr > TP_max do
5      TP_max = TP_curr;
6      (B_h,B_s, TP) = RunExperiment(H, S, workload);
7      if (B_h ≠ φ) then
8          /* hardware resource saturation */
9          S_reserve = S;
10         return B_h;
11     else if (B_s ≠ φ) then
12         /* soft resource saturation */
13         workload = step, TP = 0, TP_max = -1;
14         S = 2S;
15     else
16         workload = workload + step;
17     end
18 end
19 procedure InferMinConncurentJobs
20 workload = smallStep, i = 0, TP_curr = 0, TP_max = -1;
21 S = S_reserve;
22 while TP_curr > TP_max do
23     TP_max = TP_curr;
24     WL[i] = workload;
25     (RTT[i], TP[i], TP_curr) = RunExperiment(H, S, workload);
26     workload = workload + smallStep;
27     i++;
28 end
29 WL_min = InterventionAnalysis(WL, RTT);
30 minJobs = TP[WL_min] * RS[WL_min];
31 criServer.threadpool = minJobs;
32 criServer.Connpool = minJobs;
33 procedure CalculateMinAllocation
34 for server in front tiers do
35     server.threadpool= minJobs * RTT_ratio/Req_ratio;
36     server.Connpool = minJobs * RTT_ratio/Req_ratio;
37 end
38 for server in end tiers do
39     server.threadpool= minJobs;
40     server.Connpool = minJobs;
41 end
```

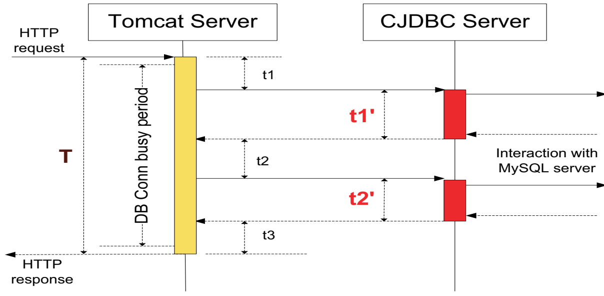


Fig. 9: A sample request process between a Tomcat server and a C-JDBC server

of the SLO-satisfaction of the system as workload increases; the SLO-satisfaction should be nearly constant under low workload and start to deteriorate significantly once the workload saturates the critical hardware resource. Readers who are interested in more details can refer to our previous paper [11].

After figuring out the minimum number of concurrent jobs ($minJobs$) that saturate the critical hardware resource, line 31 and 32 of the pseudo-code assign the value to the soft resource pools of the critical server.

3) Calculating a good allocation of other soft resources: Soft resource pools in each tier of an n-tier system control the maximum concurrent requests flow through the system. Even if the optimal soft resource allocation is set in the critical tier, the system performance may still be sub-optimal due to the improper soft resource allocation in other tiers. Too small soft resources in front of the critical tier may limit the number of concurrent requests flowing through the critical resource too much. Too small soft resources in tiers behind the critical tier may cause request congestion in front tiers. Both cases lead to critical hardware resource under-utilization.

The CalculateMinAllocation procedure calculates a good allocation of soft resources in other tiers based on the allocation of soft resources in the critical server.

Figure 9 shows a sample HTTP request processing between a Tomcat server and a C-JDBC server. The RTT of the HTTP request in the Tomcat server is T while the RTT of the following two interactions between the Tomcat server and C-JDBC server are t'_1 and t'_2. Thus, there is a job in the Tomcat server during the entire period T while a job only resides in the C-JDBC server during periods t'_1 and t'_2. Because each job in the Tomcat server needs to occupy one DB connection to communicate with the C-JDBC server, the Tomcat server needs to obtain at least $N_0 * T/(t'_1 + t'_2)$ DB connections in order to keep N_0 jobs active the C-JDBC server.

In fact, the ratio of soft resource allocation between different tiers can be calculated by combining Little's law and the Forced Flow Law [12].

$$L = TP * RTT \quad (1)$$

$$TP_{tomcat} = TP_{cjdbc}/Req_{ratio} \quad (2)$$

The Req_{ratio} denotes the average number of SQL queries issued by one servlet request for each Tomcat server. Such a ratio between different tiers depends on the characteristics of the workload. Suppose RTT_{ratio} means RTT_{tomcat}/RTT_{cjdbc}, by combining Formula(1) and Formula (2), one can obtain the following Formula:

$$L_{tomcat} = L_{cjdbc} * (RTT_{ratio}/Req_{ratio}) \quad (3)$$

Formula 3 shows that the soft resource allocation in the Tomcat tier should be no less than L_{tomcat}. The soft resource allocation in tiers behind the C-JDBC server should be no less than $minJobs$. The reason is to avoid request congestion in front tiers, which leads to the critical hardware resource under-utilization.

C. Validation of Algorithm

So far we have discussed our algorithm primarily in qualitative terms. Here we will show two case studies of applying the algorithm to find a good allocation of soft resources for all tiers in the system.

Table I shows the result of applying the algorithm to our previous two cases 1/2/1/2 and 1/4/1/4. We can see that the critical hardware resource is Tomcat CPU under configuration 1/2/1/2 while it is C-JDBC CPU under 1/4/1/4 configuration. In the second procedure of the algorithm, we get the response time, throughput, average # of jobs for each individual server under the saturation workload. The third procedure of the algorithm calculates the minimum size of thread/conn pools for tiers other than the critical tier, which is shown in the last two rows of the table. We have to point out that we turned on the logging function of each server in order to get the corresponding average response time and throughput, which degrades the system maximum throughput to about 10%-20%.

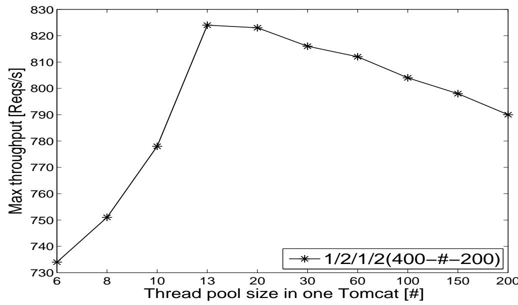
1) Validation of 1/2/1/2 case: Table I shows that the optimal size of one Tomcat thread pool under 1/2/1/2 configuration is 13. This value matches the result in Figure 10(a), which shows the maximum throughput the system can achieve with increasing Tomcat thread pool size from 6 to 200. In this set of experiments we fixed the size of Apache thread pool and Tomcat DB connecting pool to a relatively large value. The purpose is to keep enough concurrent requests flowing through the Tomcat tier when the size of Tomcat thread pool increases.

2) Validation of 1/4/1/4 case: Table I shows that the optimal size of one C-JDBC thread pool under 1/2/1/2 configuration is 8. However, there is no explicit thread pool in the C-JDBC server in practice; the one-to-one mapping between Tomcat database connection pool and C-JDBC request handling thread means that we control the maximum number of C-JDBC threads by setting the Tomcat database connection pool size.

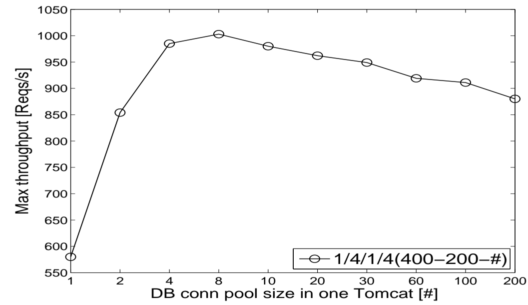
Table I shows that the minimum size of each Tomcat DB connection pool is 8 (the total size is 32 since there are 4 Tomcat servers). In fact 8 should be the optimal allocation because we don't want to over-utilize C-JDBC CPU. Figure 10(b) shows the maximum throughput the system can achieve with increasing size of each Tomcat DB connection pool from 1 to 200. We can see that the system can achieve the highest throughput when the Tomcat DB pool size is 8, which matches the output of our algorithm.

Hardware Configuration	1/2/1/2				1/4/1/4			
	Apache	**Tomcat**	CJDBC	MySQL	Apache	Tomcat	**CJDBC**	MySQL
Critical hardware resource		**CPU**					**CPU**	
Saturation WL [# Users]		5,800					6,200	
RTT [s]	0.137	0.0351			0.147	0.0378	0.0082	
TP [Reqs/s]	740	740			847	847	2728	
Average # of jobs inside	101.3	**26**			124.5	32	**22.4**	
Req_{ratio}	1	1	3.22	3.22	1	1	3.22	3.22
Size of total threads/conns	101	26	26	26	125	32	22	22
Size of individual thread/conn pool	101	**13** (×2)	26	13 (×2)	125	8 (×4)	**22**	6 (×4)

TABLE I: Output of the algorithm for hardware configuration 1/2/1/2 and 1/4/1/4



(a) Max TP vs. size of thread pool in Tomcat



(b) Max TP vs. size of DBconn pool in Tomcat

Fig. 10: Validation of the optimal soft resource allocation for hardware configuration 1/2/1/2 and 1/4/1/4

V. Related Work

Multi-threaded architecture is widely adopted as standard design for internet servers such as Apache, Tomcat, and Knot [13] due to its simple and natural programming style. Some other works [14]–[16] advocate that a good design for high concurrency servers should follow a Staged Event-Driven Architecture (SEDA), which integrates both threads and events. No matter which design an internet server adopts, the allocation of soft resources such as threads plays an important role for the control of concurrency in the request processing and has a significant impact on overall performance [13], [16], [17]. To the best of our knowledge, previous works mainly focus on the performance impact of thread allocation for a single web server, which serves requests only for static content. More complex web applications such as n-tier applications, which involve web servers, application servers, and DB servers are not discussed. As shown in our paper, soft resource allocation impacts the performance of complex n-tier systems in a more intricate and subtle way.

Previous work on soft resource allocation can be classified based on three main approaches—analytical model-based approach, feedback control-based approach, and experimental-based approach.

Analytical model based approaches have been used to find the optimal soft resource allocation for Internet servers [4], [18]. These approaches employ an analytical queuing model to simulate different scenarios. The model is meant to capture significant performance differences based on the variation of workload, concurrency limits, or QoS requirements. Though these analytical models have been shown to work well for particular domains, they are typically hard to generalize. Moreover, analytical models are constrained by rigid assumptions such as normal service times and disregard of multi-threading overhead such as context switching or JVM GC.

Feedback-control approaches [19]–[21] have been applied to optimize soft resource allocation or hardware provisioning during run-time. The feedback is generated based on a given SLA specification such as response time limit or resource utilization boundaries. However, feedback-control approaches are crucially dependent on system operators choosing correct control parameters and defining correct reconciliation actions. As shown in our paper, determining suitable parameters of control is a highly challenging task. Applying correct actions when system performance starts to deteriorate is similarly complicated due to the threat of over-allocation and under-allocation of soft resources.

Experimental based approaches for optimal soft resource allocation are evidently closest to our study. Sopitkamol et al. provide an ad-hoc experimental methodology to identify key configuration parameters that impact application server performance [22]. Raghavachari et al. present an experimental methodology, which explores the configuration space of an application server and finds the optimal configuration of the application server based on linear regression techniques [23]. Osogami et al. present an algorithm to reduce the measurement time for exploring a large configuration space and find a near optimal configuration of a web server [24]. Zheng et al. introduce an interesting framework to automate the generation of configurations of clusters of web servers based on a parameter dependency graph [25]. However, most of these experimental based approaches lack a deep understanding of the dependencies among soft and hardware resources and their implications for system performance. In contrast, our work clearly shows that optimal soft resource allocation is closely related to the weakest hardware resource in the system.

VI. CONCLUSIONS

We studied the performance impact as a function of soft resources allocation. Impact was measured and quantified using the n-tier benchmark RUBBoS. We found several situations where an inappropriate allocation of soft resources can lead to significant degradation in goodput (requests within SLA bound) at high concurrency levels. Close to system saturation, choosing an appropriate soft resource allocation to achieve good performance is a non-trivial problem. Too low and too high allocations may both cause performance degradation. On the one side, if the thread pool allocation is too low, goodput may drop by several tens of percent (Section III-A and Section III-C). On the other side, too high allocations also cause the goodput to drop by several tens of percent (Section III-B). To avoid such performance penalties, we described a soft resource allocation algorithm.

In this study, we found similarities and differences between soft resources and hardware resources when the system approaches saturation. Similar to hardware resources, the system needs sufficient soft resources to reach top performance. If a soft resource allocation is too low, it may become a bottleneck due to scarcity. This kind of soft resource saturation can be detected analogously to hardware resource saturation. In other words, the soft resource bottleneck reaches full utilization while other resources are only partially utilized.

Unlike hardware resources, which do not consume other resources when idle, soft resources may compete for shared resources such as CPU whether they are being used or not. In our experiments, when the size of thread pool reaches hundreds, it increases JVM garbage collection overhead significantly (up to 9% of total CPU). This overhead reduces the total CPU available for application processing, and too-high thread pool size reduces the maximum achievable goodput as mentioned above when the CPU approaches full utilization. More generally, for complex application scale-up soft resources have to become first class citizens (analogous to hardware) during the performance evaluation phase.

ACKNOWLEDGMENT

This research has been partially funded by National Science Foundation by IUCRC, CyberTrust, CISE/CRI, and NetSE programs, National Institutes of Health grant U54 RR 024380-01, PHS Grant (UL1 RR025008, KL2 RR025009 or TL1 RR025010) from the Clinical and Translational Science Award program, National Center for Research Resources, and gifts, grants, or contracts from Wipro Technologies, Fujitsu Labs, Amazon Web Services in Education program, 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] S. Malkowski, M. Hedwig, D. Jayasinghe, C. Pu, and D. Neumann, "Cloudxplor: a tool for configuration planning in clouds based on empirical data," in *SAC '10: Proc. of the 2010 ACM Symposium on Applied Computing*, New York, USA, 2010.

[2] 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*, New York, USA, 2009.

[3] *RUBBoS: Bulletin board benchmark*. http://jmob.ow2.org/rubbos.html.

[4] K. Aberer, T. Risse, and A. Wombacher, "Configuration of distributed message converter systems using performance modeling," in *IPCCC '01: Proc. of 20th International Performance, Computation and Communication Conference*, 2001.

[5] A. Deshmeh, J. Machina, and A. Sodan, "Adept scalability predictor in support of adaptive resource allocation," in *IPDPS '10: IEEE International Parallel and Distributed Processing Symposium*, apr. 2010.

[6] E. C. Julie, J. Marguerite, and W. Zwaenepoel, "C-jdbc: Flexible database clustering middleware," in *In Proc. of the USENIX 2004 Annual Technical Conference*, 2004.

[7] *Emulab–Network Emulation Testbed*. http://www.emulab.net/.

[8] A. R. Library, *The Performance of Web Applications: Customers are Won or Lost in One Second*. http://www.aberdeen.com/.

[9] S. Malkowski, M. Hedwig, and C. Pu, "Experimental evaluation of n-tier systems: Observation and analysis of multi-bottlenecks," in *IISWC '09*, Washington, DC, USA, 2009.

[10] *Tuning Garbage Collection with the Java[tm] Virtual Machine*. http://java.sun.com/docs/hotspot/gc5.0/gc_tun-ing_5.

[11] S. Malkowski, M. Hedwig, J. Parekh, C. Pu, and A. Sahai, "Bottleneck detection using statistical intervention analysis," in *DSOM'07: Proc. of the Distributed systems: operations and management*, Berlin, Heidelberg, 2007.

[12] P. J. Denning and J. P. Buzen, "The operational analysis of queueing network models," *ACM Comput. Surv.*, vol. 10, no. 3, 1978.

[13] R. von Behren, J. Condit, F. Zhou, G. C. Necula, and E. Brewer, "Capriccio: scalable threads for internet services," in *SOSP '03*, New York, USA, 2003.

[14] M. Welsh, D. Culler, and E. Brewer, "SEDA: an architecture for well-conditioned, scalable internet services," *SOSP '01*, pp. 230–243, 2001.

[15] D. Pariag, T. Brecht, A. Harji, P. Buhr, A. Shukla, and D. R. Cheriton, "Comparing the performance of web server architectures," *In Proc. EuroSys '07*, 2007.

[16] V. Beltran, J. Torres, and E. Ayguade, "Understanding tuning complexity in multithreaded and hybrid web servers," in *IPDPS'08: IEEE International Parallel and Distributed Processing Symposium*, 2008.

[17] H. Jamjoom, C.-T. Chou, and K. Shin, "The impact of concurrency gains on the analysis and control of multi-threaded internet services," in *INFOCOM '04*, 2004.

[18] G. Franks, D. Petriu, M. Woodside, J. Xu, and P. Tregunno, "Layered bottlenecks and their mitigation," in *QEST '06: Proc. of the 3rd international conference on the Quantitative Evaluation of Systems*, Washington, DC, USA, 2006.

[19] Y. Diao, J. Hellerstein, A. Storm, M. Surendra, S. Lightstone, S. Parekh, and C. Garcia-Arellano, "Using mimo linear control for load balancing in computing systems," in *American Control Conference '04*, 2004.

[20] X. Liu, L. Sha, Y. Diao, S. Froehlich, J. L. Hellerstein, and S. Parekh, "Online response time optimization of apache web server," in *IWQoS'03: Proc. of the 11th international conference on Quality of service*, Berlin, Heidelberg, 2003.

[21] D. Olshefski and J. Nieh, "Understanding the management of client perceived response time," in *SIGMETRICS '06/Performance '06*, New York, NY, USA, 2006.

[22] M. Sopitkamol and D. Menascé, "A method for evaluating the impact of software configuration parameters on e-commerce sites," in *Proc. of the 5th international workshop on Software and performance*, 2005.

[23] M. Raghavachari, D. Reimer, and R. Johnson, "The Deployer's Problem: Configuring Application Servers for Performance and Reliability," 2003.

[24] T. Osogami and S. Kato, "Optimizing system configurations quickly by guessing at the performance," in *SIGMETRICS '07*, NY, USA, 2007.

[25] W. Zheng, R. Bianchini, and T. D. Nguyen, "Automatic configuration of internet services," in *EuroSys '07*, Lisbon, Portugal, 2007.