# Optimizing N-Tier Application Scalability in the Cloud: A Study of Soft Resource Allocation

QINGYANG WANG and SHUNGENG ZHANG, Louisiana State University
YASUHIKO KANEMASA, FUJITSU LABORATORIES LTD., Japan
CALTON PU, Georgia Institute of Technology
BALAJI PALANISAMY, University of Pittsburgh
LILIAN HARADA and MOTOYUKI KAWABA, FUJITSU LABORATORIES LTD., Japan

An effective cloud computing environment requires both good performance and high efficiency of computing resources. Through extensive experiments using a representative n-tier benchmark application (Rice University Bulletin Board System (RUBBoS)), we show that the soft resource allocation (e.g., thread pool size and database connection pool size) in component servers has a significant impact on the overall system performance, especially at high system utilization scenarios. Concretely, the same software resource allocation can be a good setting in one hardware configuration and then becomes an either under- or over-allocation in a slightly different hardware configuration, causing a significant performance drop. We have also observed some interesting phenomena that were caused by the non-trivial dependencies between the soft resources of servers in different tiers. For instance, the thread pool size in an Apache web server can limit the total number of concurrent requests to the downstream servers, which surprisingly decreases the Central Processing Unit (CPU) utilization of the Clustered Java Database Connectivity (C-JDBC) clustering middleware as the workload increases. To provide a globally optimal (or near-optimal) soft resource allocation of each tier in the system, we propose a practical iterative solution approach by combining a soft resource aware queuing network model and the fine-grained measurement data of every component server. Our results show that to truly scale complex distributed systems such as n-tier web applications with expected performance in the cloud, we need to carefully manage soft resource allocation in the system.

CCS Concepts: • **General and reference** → **Performance**; *Measurement*; *Experimentation*; • **Software and its engineering** → **Software configuration management and version control systems**; • **Theory of computation** → *Parallel computing models*;

Additional Key Words and Phrases: Soft resource, configuration, web application, parallel processing, scalability, cloud computing

10

Q. Wang and S. Zhang contributed equally to this work.

This research has been partially funded by National Science Foundation by CISE's CNS (1566443), Louisiana Board of Regents under grant LEQSF(2015-18)-RD-A-11, and gifts or grants from Fujitsu. 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.

Authors' addresses: Q. Wang and S. Zhang, School of Electrical Engineering and Computer Science, Louisiana State University, Baton Rouge, 3325 Patrick F. Taylor Hall, LA 70803, USA; emails: {qywang, shungeng}@csc.lsu.edu; Y. Kanemasa, L. Harada, and M. Kawaba, FUJITSU LABORATORIES LTD., 1-1, Kamikodanaka 4-chome, Nakahara-ku, Kawasaki 211-8588, Japan; email: {kanemasa, harada.lilian, kawaba}@jp.fujitsu.com; C. Pu, College of Computing, Georgia Institute of Technology, 266 Ferst Dr, Atlanta, GA 30332-0765, USA; email: calton.pu@cc.gatech.edu; B. Palanisamy, School of Computing and Information, University of Pittsburgh, 135 N. Bellefield Avenue, Pittsburgh, PA 15260, USA; email: bpalan@pitt.edu.



2376-3639/2019/06-ART10 $15.00
https://doi.org/10.1145/3326120

**ACM Reference format:**
Qingyang Wang, Shungeng Zhang, Yasuhiko Kanemasa, Calton Pu, Balaji Palanisamy, Lilian Harada, and Motoyuki Kawaba. 2019. Optimizing N-Tier Application Scalability in the Cloud: A Study of Soft Resource Allocation. *ACM Trans. Model. Perform. Eval. Comput. Syst.* 4, 2, Article 10 (June 2019), 27 pages.
https://doi.org/10.1145/3326120

## 1 INTRODUCTION

An important advantage of cloud computing environment is the scalability: an application can scale the underlying computing resources (e.g., number of virtual machines) to always meet the demand of the fluctuation workload for both good performance and high resource efficiency [13, 39]. For long-term cloud users, scalability is especially important because the low operational cost brought by efficient resource utilization can justify the savings of avoiding building a dedicated cluster. For cloud vendors, scalability is also very important because they can save infrastructure cost by using less amount of computing resources and power through efficient resource utilization. So scalability and the associate efficient resource utilization are important requirements for shared cloud computing environment.

A key challenge of achieving efficient resource utilization through scalability is the intelligent mapping of cloud resources to the real-time resource demands of running applications. This is because typical cloud applications such as e-commerce usually have large fluctuating and sometimes unpredictable workload (e.g., peak load 10 times higher than average) [9]. In addition, these applications may have strict quality of service (QoS) requirements such as bounded response time. To achieve such an intelligent mapping, significant previous work has been done in hardware resource scaling, for example, scaling hardware resources such as virtual machines or CPU cores based on offline configuration plans [36] or pre-defined online adaptation policies (e.g., CPU utilization larger than 80%) [20, 23, 27, 43].

In this article, we show that an intelligent mapping of cloud resources to real-time resource demands of n-tier applications needs to take both hardware and software resources into account. This is because of the complex dependencies between hardware and software resources (e.g., thread pool, connection pool, which we refer to as soft resources in this article) from each tier of the system. For example, the number of threads in an upstream tier of an n-tier system can control the level of concurrent requests flowing into downstream tiers, which may lead to either under-utilization or over-utilization of the critical hardware resource in the system. To study the impact of soft resource allocation on n-tier application performance, we developed tools to conduct fine-grained monitoring of both hardware (CPU, memory, I/O) and soft resources (thread/DB connection pool) of each tier of the system. Then, we analyze the relationship between application performance metrics (e.g., throughput, response time) and the fine-grained measurement of hardware and soft resource usage by varying the soft resource allocation in each tier of the system.

The first contribution is the quantitative evaluation of the impact of soft resource allocation on the performance of n-tier applications with different hardware configurations. Using the Rice University Bulletin Board System (RUBBoS) benchmark [44], we show that a sub-optimal soft resource allocation (e.g., thread/DB connection pool) can easily degrade the performance of a 4-tier system from 31% to 110%, depending on the Service Level Agreement (SLA) specifications (see Figure 2). We also show that a set of near-optimal soft resource allocation for one hardware configuration can become a very bad choice when the system scales out to a different size (hardware resource scaling) (comparing Figures 2 and 3).

The second contribution is the sensitivity analysis of two policies of soft resource allocation: either conservative or liberal allocation leads to inferior performance. We show that a conservative soft resource allocation (e.g., small thread pool size) may lead to under-utilization of the critical

hardware resource because of not enough workload concurrency in the system. On the other hand, a liberal allocation of soft resources (e.g., large Database (DB) connection pool size) may degrade the efficiency of the critical hardware resource in the system because of the increased overhead in CPU and memory. For example, our experiments show that allocating a few hundreds of threads in a Tomcat server can cause frequent Java garbage collection activities that waste the critical CPU resource of the bottleneck server up to 9%.

The third contribution is a practical solution that recommends a near-optimal soft resource allocation for each tier of an n-tier system. Our solution combines a soft resource-aware queuing network model and fine-grained measurement data of each tier in the system. The model abstracts the request processing in an n-tier system composed of typical thread-based servers and characterizes the relationship of proper allocation of soft resources in each tier of the system. The parameters of the model can be obtained through fine-grained measurement of the system, which enables the prediction of optimal (or near-optimal) allocation of soft resources in each tier of the system.

In general, our results suggest that soft resources should be treated as essential components when scaling n-tier applications in the cloud. This is due to the strong dependencies between soft and hardware resources within each component server and among different servers in the system [41]. For example, scaling out/in the servers of one tier not only affects the workload concurrency in itself, but also affects the workload concurrency in both the upstream and downstream tiers (see Section 3). In fact, complex systems like n-tier applications can be truly scalable only if hardware and soft resources are treated as a whole.

The rest of the article is organized as follows. Section 2 demonstrates the impact of soft resource allocation on n-tier application performance with different hardware configurations. Section 3 conducts a sensitivity analysis of two soft resource allocation policies: liberal and conservative allocation. Section 4 describes our soft resource-aware queuing network model and a practical algorithm for near-optimal soft resource allocation in the system. Section 5 summarizes the related work and Section 6 concludes the article.

## 2 BACKGROUND AND MOTIVATION

### 2.1 Background Information

*2.1.1 Soft Resources in n-Tier Systems.* When conducting the performance evaluation of computer systems, hardware resources (e.g., CPU, disk, memory, network) are usually well-defined monitoring components for performance analysis and reasoning. We use soft resources to refer the software components such as threads and Transmission Control Protocol (TCP) connections that use hardware resources. For example, threads use CPU and memory, and TCP connections use the network. We also expand the definition of soft resources to refer the software components that use soft resources or synchronize the use of both soft and hardware resources. For example, a lock is a soft resource that synchronizes the use of data structures and CPU. In general, the function of soft resources is to facilitate the sharing of hardware resources. For example, threads facilitate the sharing of the CPU resources through concurrency and parallelism. As a result, soft resources are indispensable components of the critical job execution path in the system.

More importantly, soft resources create dependencies among components in the system because of the classic synchronous Remote Procedure Call (RPC) style inter-server communication. For example, a request arrives in an Apache server, which dispatches the request to the downstream application server such as Tomcat, which sends queries to a downstream database server such as MySQL to fetch persistent state of the system. Soft resources such as processing threads in an

**Software Stack**

| | |
|---|---|
| Web Server | Apache 2.2.31 |
| Application Server | Apache Tomcat 7.0.56 |
| Cluster middleware | C-JDBC 2.0.2 |
| Database server | MySQL 5.6.35 |
| Sun JDK | jdk1.7.0_06 |
| Operating system | RHEL 6.4 (kernel 2.6.32) |
| Hypervisor | VMware ESXi v6.0 |
| System monitor | Collectl v4.0, Sysstat 10.0.0 |

(a) Software setup

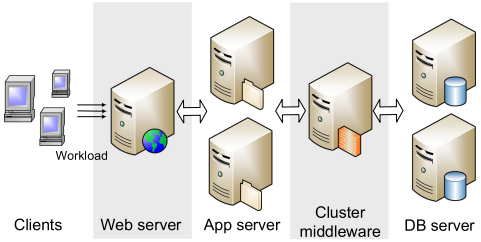
**ESXi Host Configuration**

| | |
|---|---|
| Model | Dell Power Edge R430 |
| CPU | 2 * Intel Xeon E5-2603, 1.60GHz Hexa-Core |
| Memory | 16GB |
| Storage | 7200rpm SATA local disk |

**VM Configuration**

| Type | # vCPU | CPU limit | CPU shares | vRAM | vDisk |
|---|---|---|---|---|---|
| Small (S) | 1 | 1.60GHz | Normal | 2GB | 20GB |

(b) Hardware node setup




(c) Sample topology

Fig. 1. Experimental setup of the VMware ESXi cluster.

upstream server will not be released until a downstream server finishes all the processing, creating non-trivial dependencies in the long invocation chain.

*2.1.2 Experimental Environment.* We use RUBBoS, a representative n-tier application benchmark to conduct our experiments in our VMware vSphere cluster. RUBBoS benchmark application is modeled after the famous tech news website Slashdot [2]. It can be configured as 3-tier (web server, application server, and database server) or 4-tier (add Cluster Java Database Connectivity (C-JDBC) database clustering middleware [17]). Figure 1 shows the software stack, hardware specification, and a sample configuration that we have used in our experiments. The RUBBoS workload consists of 24 web transactions such as ViewStory and StoryOfTheDay. The workload generator of RUBBoS simulates a certain number of users sending real HTTP requests to interact with the benchmark application; each user navigates between different web transactions by following a Markov chain model abstracted from the trace of real user behaviors [16]. For example, the average think time between every two consecutive web transactions follows a normal distribution with 7 seconds as its mean. Such a user behavior model has been widely used by other typical n-tier benchmark applications such as Rice University Biding System (RUBiS) [45], Transaction Performance Council benchmark W (TPC-W), and CloudStone [48].

We ran our experiments in our VMware vSphere cluster. We use $\#Web/\#App/\#CM/\#DB$ to represent the hardware topology of a 4-tier system, which means the number of web servers, application servers, database clustering middleware, and database servers. Each server (including Apache, Tomcat, C-JDBC, and MySQL) is running in a VM deployed on a dedicated physical machine (the specification of each machine is similar to the EC2 g2 dedicated host [7]), in this case, we are able to eliminate complicated factors like VM interference and focus on the study of the impact of soft resource allocation on the n-tier system performance. For example, in our 1/4/1/4 configuration case, the 4-tier system uses 10 VMs, each of which is deployed on a dedicated Dell Power Edge R430 as shown in Figure 1(b). Except for database servers, every other server has a

thread pool to handle requests from the upstream tier and a connection pool to communicate with the downstream tier. For each hardware topology, we choose three representative soft resources for our evaluation, which we denote as $\#W_{threads}$-$\#A_{threads}$-$\#A_{DBConn}$, meaning the thread pool size in a web server, the thread pool size in an application server, and the DB connection pool size in an application server. So for a hardware configuration like 1/2/1/2 (Figure 1(c)), the associate soft resource allocation can be 400-150-60, which means 400 Apache threads, 150 Tomcat threads, and 60 Tomcat DB connections. Since there are two Tomcat servers, the total number of threads and DB connections doubles in the application server tier. Other soft resource allocations are fixed to limit the exponential experiment space.

### 2.2 SLA-based Performance Requirements

Web applications such as e-commerce are sensitive to response time variation. Amazon reported that every 100ms increase in page load is positively correlated with 1% decrease in sales [29]. Google requires all the queries to return within 500ms [19]. Thus, only those requests returning within certain response time limits can generate a positive impact on service providers' business. In a cloud computing environment, SLA are typically used to specify desired response time, usually in one or two seconds, depending on the sensitivities of the target application for response time.

Our previous research shows a general SLA model that evaluates the impact of different response time range on a target service provider's revenue [37]. In this article, we adopt a simplified SLA model to integrate throughput and response time together for system performance evaluation. In our simplified SLA model, we use a simple response time threshold. We only consider the requests with response time lower than the threshold, which satisfy our SLA and count as *goodput*. Requests with response time above the threshold are counted as *badput*. Goodput and badput put together equal the classic definition of throughput. By considering both goodput and badput, we can refine our traditional throughput model by taking user-perceived response time into account, leading to a more realistic system performance analysis.

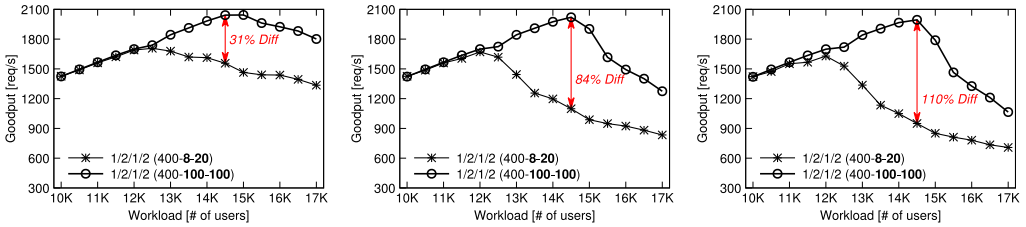
### 2.3 Performance Decrease with Simplified SLA Model

In this section, we use concrete measurements to show the significance of soft resource allocation on n-tier application performance by applying our simplified SLA model. The goal is to illustrate the importance of the problem; a more detailed explanation is in Section 3.

*2.3.1 Impact of Under-Allocation.* Figure 2 shows the goodput comparison between two soft resource allocations 400-100-100 and 400-8-20 under the same hardware topology 1/2/1/2. 400-100-100 is based on rule-of-thumb practice from industry while 400-8-20 is a more conservative configuration. We choose the workload range from 10,000 to 17,000 because it well captures the keen of system goodput as workload increases. The three subfigures show that the goodput of the 400-8-20 case starts to degrade much earlier than the 400-100-100 case, indicating the significant impact of soft resource allocation on n-tier application performance.

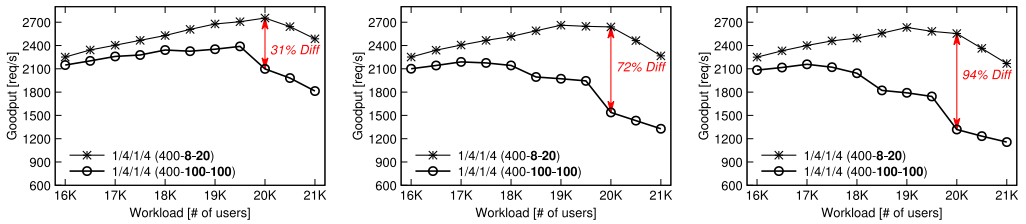
Readers may immediately question the choice of the conservative 400-8-20 allocation, since it is "obviously" too low. However, as we will see in Section 2.3.2, the conservative allocation 400-8-20 will significantly outperform the original rule-of-thumb allocation 400-100-100 once the system scales from 1/2/1/2 to 1/4/1/4, suggesting that a good soft resource allocation may not always be good as the system scales to a different size.

Figure 2 also shows that the impact of soft resource allocation on system goodput is sensitive to the response time threshold. For example, at workload 14,500, we show the goodput gap between the 400-100-100 case and 400-8-20 increases from 31% to 110% when the target response time threshold decreases from 3s to 500ms as shown in Figure 2(a)–(c). We note that we choose the




(a) Threshold 3s (b) Threshold 1s (c) Threshold 500ms

Fig. 2. The goodput comparison between two different soft resource allocations under the same hardware configuration. 1/2/1/2 represents the hardware configuration, meaning one Apache (web server), two Tomcat (application servers), one C-JDBC (database clustering middleware), and two MySQL (database servers). The three numbers separated by hyphens refer to soft resource allocation, for example, 400-100-100 means thread pool size (400) in one Apache server, the thread pool size (100) in one Tomcat server, and the DB connection pool size (100) in the same Tomcat server. 400-100-100 is considered as a reasonable soft resource allocation by practitioners from industry.




(a) Threshold 3s (b) Threshold 1s (c) Threshold 500ms

Fig. 3. Performance degradation of the previous reasonable allocation 400-100-100 after the system hardware configuration scales from 1/2/1/2 with 1/4/1/4.

allocation 400-100-100 as the baseline because it is considered as a good choice by practitioners from industry. Such results indicate that even if the overall throughput may be the same, the goodput can be very different, depending on the SLA specification.

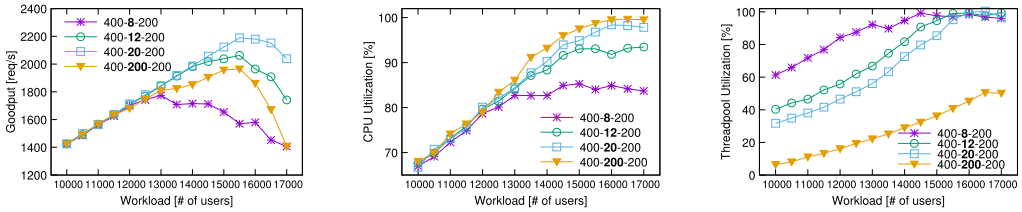
*2.3.2 Impact of Over-Allocation.* Figure 3 shows the performance comparison of the same soft resource allocations as in Section 2.3.1 after we scale the underlying hardware configuration from 1/2/1/2 to 1/4/1/4. As we can see, the previously inferior allocation 400-8-20 now outperforms significantly the "rule-of-thumb" configuration 400-100-100. As we decrease the response time threshold from 3s to 500ms, the goodput gap between these two cases becomes wider. The main reason is that of the unintentional over-allocation of soft resources after system scales out, causing significant overhead to the critical resource in the downstream tiers. More details are in Section 3.2.

Overall, SLA models connect economic goals with technical performance measurements, enabling a more realistic view of the impact of soft resource allocations on the service providers' business. For the rest of the article, we will use 3 seconds as response time threshold for goodput calculation of different combinations of soft resource allocation and hardware resource configuration.

In the following section, we will explain the reasons for performance difference under different combinations of soft resource allocation and hardware configuration we have seen so far.

## 3 EVALUATION OF DIFFERENT SOFT RESOURCE ALLOCATION STRATEGIES

In this section, we conduct a sensitivity analysis of the impact of different soft resource allocation strategies on n-tier application performance. We found some similarities and differences between




(a) Goodput (b) CPU utilization of the first Tomcat (c) Thread pool utilization in Tomcat tier

Fig. 4. Performance degradation caused by Tomcat thread pool under-allocation. The hardware configuration is 1/2/1/2.

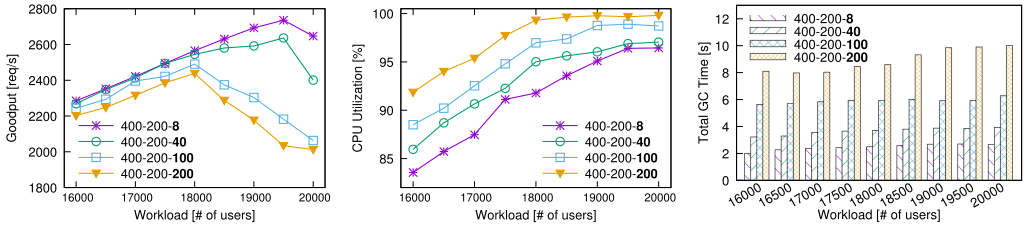
hardware and soft resources. Section 3.1 shows a case of similarity that too low allocation of soft resources in one tier under-utilizes the hardware resources of the tier, and becomes the bottleneck of the system. Section 3.2 shows one case of difference; unlike hardware resources, too high allocation of soft resources degrades the system performance by causing the high overhead of the critical hardware resource in the system. For example, high allocation of threads can cause more frequent Java garbage collection activities in C-JDBC, causing high overhead of the C-JDBC CPU, which is the critical hardware resource of the system. Section 3.3 shows a more interesting case of similarity. Since soft resources create dependencies between different tiers, we show that a non-obvious "low" allocation of soft resources in the frontmost tier (Apache) leads to the under-utilization of the critical hardware resource in the downstream C-JDBC server.

### 3.1 Conservative Soft Resource Allocation

The first strategy of soft resource allocation is the straight-forward conservative allocation. The purpose of conservative allocation is to avoid unnecessary overhead caused by abundant soft resources. Since a virtual machine in a cloud computing platform typically has a small number of CPU cores (e.g., 1 to 4), a conservative allocation (e.g., 10) of threads or connections should be able to fully utilize the underlying hardware resources and avoid high concurrency overhead caused by scheduling and context switches. Our results show that a reasonable conservative allocation of soft resources is non-trivial to gain; the traditional wisdom of one or two threads per CPU core does not apply to component servers in the context of n-tier systems.

We use Tomcat thread pool as an example to illustrate the impact of conservative soft resource allocation. The hardware configuration is 1/2/1/2, where the Tomcat CPU is the critical hardware resource of the system (we will show later). The number of threads in Apache and database connections in Tomcat are fixed at 400 and 200, respectively. The liberal allocation of these soft resources are supposed to be abundant and never become the bottleneck in the system. We only change Tomcat thread pool size, from 8 to 200. We note that each Tomcat VM only has one vCPU core, thus eight threads in Tomcat should be sufficient to utilize the one vCPU core in Tomcat.

The system goodput under the increasing number of threads in Tomcat is shown in Figure 4(a). This figure shows that the system goodput increases as the Tomcat threads increase from 8 to 20. For example, the goodput of the 400-20-200 case is 47% higher than that of the 400-8-200 case, indicating that eight threads in Tomcat are not enough to fully utilize the Tomcat CPU. This hypothesis is confirmed in Figure 4(b), which shows the CPU utilization of the first Tomcat server in the system. We omit the second Tomcat CPU utilization since it is similar as the first one because of the well-functioned load-balancer in the upstream tier (the Apache web server). This figure shows that eight threads in Tomcat are not able to saturate Tomcat CPU. For example, at workload 16,000, the Tomcat CPU utilization is only 83% in the 400-8-200 case while the number




(a) Goodput (b) C-JDBC CPU utilization (c) Total JVM GC time in C-JDBC

Fig. 5. Performance degradation caused by Tomcat DB connection pool over-allocation. The hardware configuration is 1/4/1/4.

is about 100% in the 400-20-200 case. This result indicates that all the threads in the small thread pool allocation cases (i.e., the 8 and 12 threads case) are either processing the current requests or busy waiting for the response from the downstream C-JDBC. As a result, there is no available thread in Tomcat to process new requests, leading to the idle state of Tomcat CPU.

A detailed analysis of Tomcat thread pool utilization is shown in Figure 4(c). This figure shows that the smaller the Tomcat thread pool is, the earlier the pool becomes saturated. For example, thread pool size 8 saturates at the workload of 14,500 while the pool size 12 saturates at about 15,500, and pool size 20 saturates at about 16,000. On the other hand, the pool size 200 is far from saturation (e.g., only 50% utilized) even under high workload range. This is because Tomcat CPU becomes a bottleneck (see Figure 4(b)) before the thread pool is exhausted for request processing. This result suggests that we need to monitor both soft and hardware resources to get a full picture of performance analysis. Only monitoring hardware resources (e.g., using vmstat, sar, collectl) will miss the real performance bottleneck as shown in the cases 400-8-200 and 400-12-200.

Figure 4(a) and 4(b) show another interesting phenomenon: the highest goodput achieved in the 200 threads case is significantly lower than that of the 20 threads case, suggesting that monolithically increasing threads allocation could lead to sub-optimal system performance. This is due to the non-trivial overhead caused by liberal soft resource allocation, as we will discuss in the next section.

### 3.2 Liberal Soft Resource Allocation

The second soft resource allocation strategy we want to evaluate is the straight-forward liberal allocation, with the purpose of fully utilizing the underlying hardware resources. This allocation strategy shows the differences between soft and hardware resources. Unlike hardware resources, soft resources such as threads and database connections consume other hardware resources even if they are idle. Traditional wisdom assumes that the cost for maintaining soft resources is low (e.g., a small amount of memory or CPU cycles). So liberal allocation of soft resources is considered reasonable as long as there is enough memory.

Our experimental results show that liberal allocation of soft resources can cause significant overhead to the critical hardware resource when the system is approaching saturation. In this set of experiments, we scale the hardware configuration from 1/2/1/2 in Section 3.1 to 1/4/1/4 in order to resolve the Tomcat bottleneck. We fix the number of threads in Apache and Tomcat to be 400 and 200 to avoid any bottleneck of these soft resources. Then we change the database connection pool size in Tomcat from 8 to 200 and see the impact of such an increase on system performance. Figure 5(a) and (b) show the system performance degradation as we increase the database connections in Tomcat. We note that each database connection in Tomcat corresponds to one thread in C-JDBC, which in turn corresponds to one thread in the database server MySQL due

to the synchronous RPC-style communication between consecutive tiers. So a higher number of database connections means a higher number of threads in C-JDBC and MySQL for query routing and actual query processing.

Figure 5(a) shows the significant system goodput degradation as we increase the Tomcat database connection pool size from 8 to 200. For example, at workload 19,500, the goodput of 400-200-8 is 34% higher than that of the 400-200-200 case. Figure 5(b) shows the average CPU utilization of C-JDBC server at the same workload range. This figure shows the C-JDBC CPU utilization is above 95% at high workload range (after 19,000) under all the four configurations, suggesting that C-JDBC CPU is the critical hardware resource in the system.

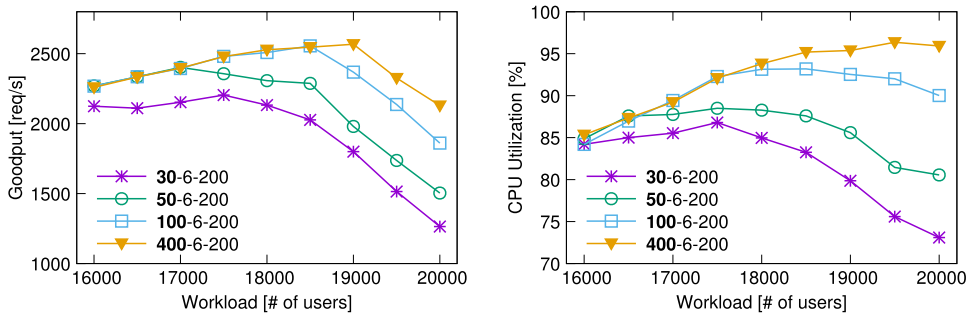
A further analysis of C-JDBC CPU utilization shows an opposite trend of system goodput as the number of database connections increases in Tomcat. For example, at workload 19,500, the C-JDBC CPU utilization of the 400-200-8 case is the lowest while the corresponding system goodput is the highest (see Figure 5(a)), suggesting significant CPU overhead in C-JDBC as we increase the database connections in Tomcat from 8 to 200. Remember each database connection corresponds to one thread in C-JDBC, so the significant CPU overhead is caused by the high number of threads in C-JDBC. The well-known multithreading overhead includes context switches and scheduling. Here, we show another major source of high overhead for C-JDBC CPU: the Java Virtual Machine (JVM) garbage collection as the number of threads increases in C-JDBC.

The JVM garbage collection affects system goodput in two ways. First, the JVM garbage collection consumes C-JDBC CPU, which is the critical hardware resource of the system. The CPU time used for JVM garbage collection cannot be used for request processing, thus reducing the maximum achievable throughput of the system. Figure 5(c) compares the accumulated JVM garbage collection time of the C-JDBC server during a 3-minute runtime experiment. This figure shows that at workload 19,500, the total JVM garbage collection time is above 16 seconds (9% of total) in the 400-200-200 case while about 4 seconds (2% of total) in the 400-200-8 case. Second, during the JVM garbage collection period, the JVM will freeze for cleaning garbage (unreferenced objects) in memory, which lengthens the waiting time of the queued requests and further decreases the system goodput.

### 3.3 Buffering Effect of Soft Resources

In the previous two sections, we always keep the Apache thread pool size to be 400. Considering that the Apache VM only has one vCPU core, 400 threads appear to be more than enough. In this section, we show that allocating a high number of soft resources in the front-most tier (Apache in this case) of the system is important to achieve good performance. Unlike the significant overhead caused by the over-allocation of soft resources as introduced in Section 3.2, liberal allocation of soft resources in the front-most tier (e.g., Apache server) functions as a buffer for clients' requests, stabilizing the requests flowing to the downstream tiers and improving the overall system performance.

The experiments here still use the 1/4/1/4 hardware configuration, where the critical hardware resource in the system is C-JDBC CPU as shown in the previous section. We keep a fixed number of threads (6) and database connections (200) in each of the four Tomcat servers. Then, we vary the thread pool size of the front-most Apache server from 30 to 400 and see its performance impact. Figure 6(a) shows that the system goodput keeps increasing as we increase the Apache thread pool size from 30 to 400. For example, under workload 19,500, the 400 threads case outperforms the 30 threads case by 53% in system goodput. Intuitively, readers may believe that the 30 threads case is just a simple soft resource under-allocation case as we have seen in Section 3.1, where the scarcity of soft resources becomes the system bottleneck, limiting the full utilization of the critical hardware resource in the system.




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

Fig. 6. Performance degradation caused by under-allocation of Apache threads. The hardware configuration is 1/4/1/4.

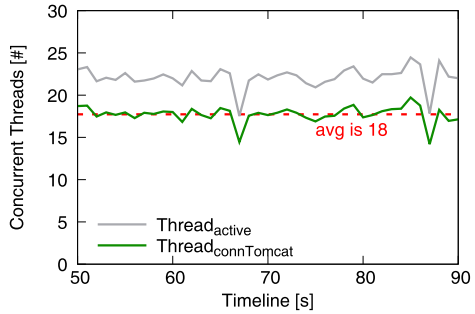
We have observed two interesting phenomena that make the 30 threads case a special soft resource under-allocation case. First, it is not clear why 30 threads in Apache is the bottleneck since the total number of threads in the Tomcat server tier is just 24 (four Tomcat servers and six threads each). Given that soft resources control the concurrent requests flowing to the downstream tiers, it should be the 24 threads in the Tomcat tier, not the 30 threads in Apache that limit the concurrent requests flowing to the downstream bottleneck server C-JDBC. Second, the utilization of the C-JDBC CPU (the critical hardware resource of the system) keeps decreasing as the workload increases from 16,000 and 20,000 in the 30 threads case, which is counter-intuitive to our normal understanding that higher workload should lead to higher hardware resource utilization before saturation. For example, the CPU utilization of C-JDBC at workload 19,500 is about 10% lower than that at workload 16,000.

Our detailed analysis of the thread status in Apache reveals that both the above interesting phenomena are due to the small request buffer size (30 threads case) in the front-most tier of the system. Figure 7 shows a 40-second runtime status of Apache threads in the 30 threads case at workload 16,000 and 19,500, respectively. We consider two periods of an Apache thread: $Thread_{active}$ and $Thread_{connTomcat}$. The former period means that the Apache thread receives a request from a client but is not released to process the next request, meaning the thread is active (or occupied); the latter period is a sub-period of the active period, which means the Apache thread just routes the request to the downstream Tomcat but has not received the response from Tomcat.

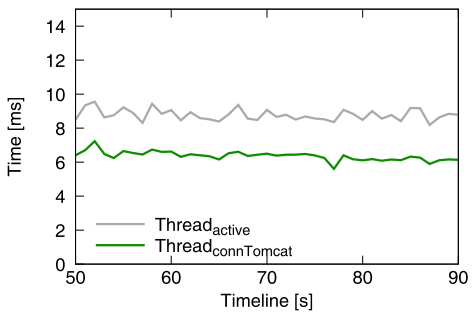
The first phenomenon (30 threads is not enough) can be explained by counting the number of Apache threads in $Thread_{connTomcat}$ period in the 30-6-200 at workload 19,500. Figure 7(b) shows that about 25 threads in the Apache thread pool are active; however, the actual number of threads connecting to Tomcat is only 15 in average, less than the total number of threads (24) in the Tomcat tier. Thus, it is the Apache tier, not the Tomcat tier that limits the concurrent requests flowing to the downstream bottleneck server C-JDBC, causing the C-JDBC CPU under-utilization.

The second phenomenon (CPU utilization in CJDBC decreases as workload increases) can be explained by comparing Figures 7(a) and (b), which shows that the number of threads in $Thread_{connTomcat}$ period at workload 19,500 is even lower than that at workload 16,000. For example, the average number of $Thread_{connTomcat}$ is about 15 in the former case while 18 in the latter case. Fewer worker threads interacting with Tomcat means a fewer number of concurrent requests been pushed to the downstream tiers including C-JDBC, resulting in reduced the CPU utilization of the C-JDBC server at workload 19,500.

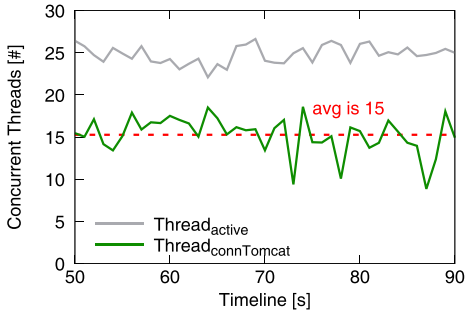
Readers may wonder why $Thread_{connTomcat}$ at workload 19,500 is lower than that at 16,000. Figures 7(c) and (d) show the average time that a worker thread in $Thread_{active}$ period (gray line) and



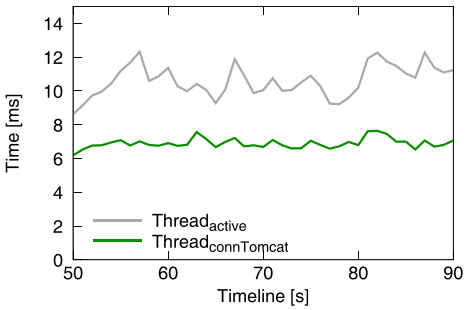

(a) # of Apache threads in connecting Tomcat among the total active Apache threads at WL 16000




(b) # of Apache threads in connecting Tomcat among the total active Apache threads at WL 19500




(c) Apache thread time spent in connecting Tomcat over total active time at WL 16000. The average percentage is 73%.




(d) Apache thread time spent in connecting Tomcat over total active time at WL 19500. The average percentage is 60%.

Fig. 7. Analysis of performance degradation of the 30-6-200 case as workload increases from 16,000 to 19,500. Comparing (a) and (b), although more Apache threads are busy at WL 19,500, the number of threads connecting to Tomcat is lower, limiting the number of concurrent requests flowing to downstream tiers. This is because the percentage of time of Apache threads in connecting Tomcat $Thread_{connTomcat}$ over total active $Thread_{active}$ at WL 19,500 is significantly lower than that at WL 16,000 by comparing (c) and (d).

also in $Thread_{connTomcat}$ period (red line) when the system is at 16,000 and 19,500, respectively. It is clear that the thread time in $Thread_{active}$ period at workload 19,500 is averagely higher than that at workload 16,000, while the thread time in $Thread_{connTomcat}$ period at the two workloads are similar. In this case, the percentage of time that a worker thread in Apache interacting with the Tomcat tier over the total active time ($\frac{Thread_{connTomcat}}{Thread_{active}}$) at workload 19,500 is less than that at workload 16,000. As a result, a lower number of Apache threads are connecting to Tomcat at workload 19,500. We note that the main contributor of the high thread time in $Thread_{active}$ period at workload 19,500 is that Apache waits for Finish flag (FIN) replies from clients that close their corresponding TCP connection. We observed that at a high workload, this wait time becomes longer than that at higher workload (higher workload means more congestion in the network), which delays the release of the corresponding Apache worker thread.

The above two interesting phenomena do not happen in the 400-6-200 case because 400 threads in Apache provide a large buffer that stabilizes the number of concurrent requests flowing to downstream tiers. Figure 8 shows the Apache threads status in 400-6-200 at workload 19,500. While in this case, the Apache threads still need to wait for FIN reply from clients, the number of Apache

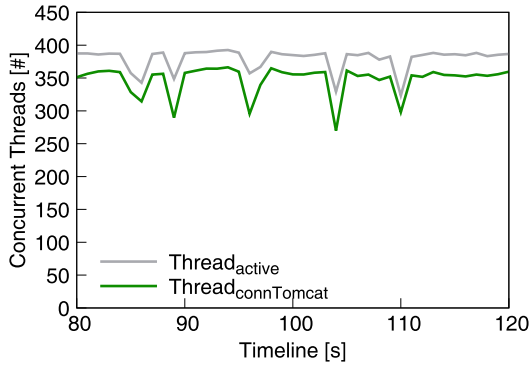



Fig. 8. # of Apache threads in connecting Tomcat among the total active Apache threads in the 400-6-200 case at workload 19,500. Four hundred threads in the Apache server enable a high number of Apache threads in connecting Tomcat, allowing a high number of concurrent requests to downstream tiers.

threads connecting to the downstream Tomcat tier is much more than the concurrency limit in the Tomcat tier (24). Thus, Apache can always push a stable amount of workload to downstream tiers, saturating the critical hardware resource of the system—the CJDBC CPU (see Figure 6(b)).

## 4 DETERMINING SOFT RESOURCE ALLOCATIONS

So far, we evaluated the impact of two soft resource allocation strategies on n-tier system performance. In this section, we first summarize the challenges and opportunities of optimal soft resource allocations. We then introduce a model of n-tier systems with explicit soft resource allocations in each tier. Based on the model, we design an empirical algorithm of choosing a "Goldilocks" soft resource allocation for each hardware configuration, followed by experimental validation.

### 4.1 Challenges and Opportunities of Good Soft Resource Allocation

Our previous experimental results can be summarized as follows. Too low soft resource allocation may under-utilize the critical hardware resource in the system (Sections 3.1 and 3.3); too high allocation may cause high overhead on the critical system resource (Section 3.2). Thus, a key principle of a good soft resource allocation is to maximize the utilization of the critical hardware resource in the system while avoiding unnecessary overhead on it. However, searching for a globally good soft resource allocation is challenging due to the following three reasons:

(1) The optimal soft resource allocation is highly related to the location of the critical hardware resource in an n-tier system; however, the location may shift when the system scales to a different size due to workload variation. For example, the critical hardware resource in the 1/2/1/2 configuration is the Tomcat CPU (Section 3.1) while it is C-JDBC CPU in 1/4/1/4 (Section 3.2).
(2) The system performance (both throughput and response time) in general is not sensitive to over- or under-allocation of soft resources until some critical hardware resource approaches saturation. The impact of a bad soft resource allocation may not be revealed when the system is at low utilization.
(3) The state space for the allocation of each soft resource is usually very large (e.g., from one to unlimited). The complexity increases exponentially to find good combinations for multiple soft resources. Brute-force search for optimal allocation of soft resources by running experiments exhaustively is impractical.

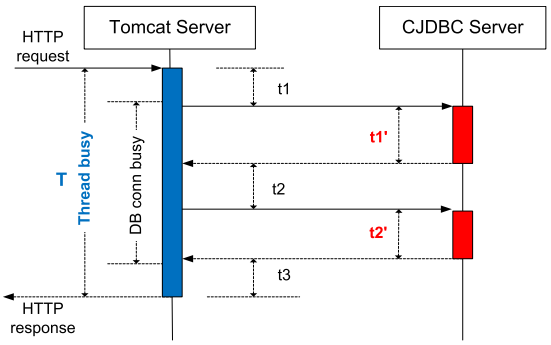



Fig. 9. A sample HTTP request processing between Tomcat and C-JDBC.

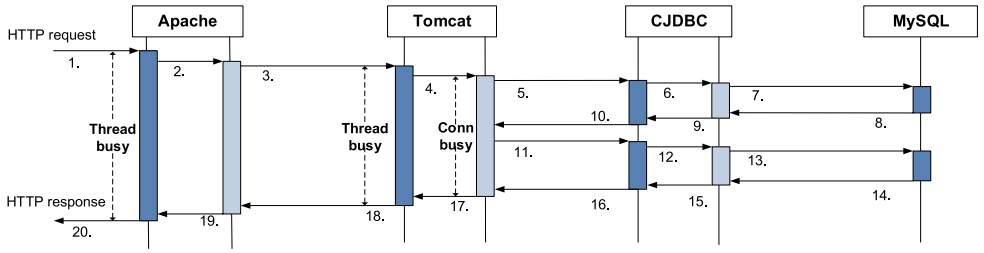



Fig. 10. Illustration of a sample request processing in a 4-tier application.

Although challenging, there are still opportunities to reduce the complexity of good soft resource allocation given the characteristics of n-tier systems. Our previous analysis (Section 3.2 and 3.3) shows that the soft resource allocations in different tiers are correlated with each other. So given a good allocation of soft resources in one tier, it may be possible to infer good soft resource allocations in other tiers given such "hidden" correlation. For example, Figure 9 shows a sample interaction between a Tomcat and a C-JDBC server when they process an HTTP request. The HTTP request arrives in Tomcat triggers two subsequent DB queries to the downstream C-JDBC. The response time of the HTTP request in Tomcat is $T$, while $t_1'$ and $t_2'$ for the following two queries to C-JDBC. Thus, a worker thread in Tomcat is busy during the entire period $T$ while the corresponding worker thread in C-JDBC is busy only during $t_1'$ and $t_2'$. In this case, the Tomcat server needs to obtain at least $N_0 * T/(t_1' + t_2')$ threads in order to keep $N_0$ threads busy in C-JDBC.

The above example only shows the relationship between processing threads in Tomcat and CJDBC. Other important soft resources in the system such as database connections are not shown. A more detailed picture of the request processing in a 4-tier application is shown in Figure 10. This figure shows that when a processing thread in any server (except database) receives a request, it pre-processes the request first and then fetches a connection to communicate with the downstream tier. The boxes under each server show the busy time of a processing thread and the corresponding connection for downstream communication. To achieve high performance of such a 4-tier system, we need to choose a good allocation of soft resources in each tier in a coordinated manner.

In the following section, we generalize the relationship of soft resources between different tiers in an n-tier system through an analytical model.

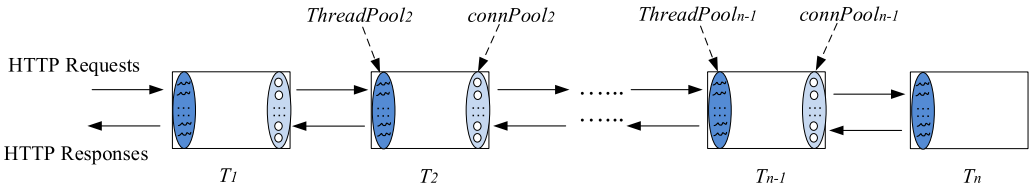



Fig. 11. A simple model of n-tier applications with soft resources. Except the last tier (database tier), every server has a thread pool to process incoming requests and a connection pool to communicate with the consecutive downstream tier.

Table 1. Descriptions of Parameters in Our Model

| Symbol | Description |
|---|---|
| $N$ | Number of tiers |
| $ThreadPool_i$ | Thread pool size in tier $i$ |
| $ConnPool_i$ | Connection pool size in tier $i$ |
| $TP_i$ | Average throughput of tier $i$ ($1 \leq i \leq N$) |
| $RT_i$ | Average response time of tier $i$ |
| $RT_i^{conn}$ | Average connection busy time in tier $i$ |
| $Q_i$ | Average queued requests of tier $i$ |
| $V_{ij}$ | Request visit ratio between tier $i$ and tier $j$ |
| $RT_{ratio}^{ij}$ | Average response time ratio between tier $i$ and tier $j$ |

### 4.2 Soft Resource Aware Modeling of n-Tier Systems

Consider that a web application with $n$ tiers is denoted by $T_1, \ldots, T_n$ as shown in Figure 11. For simplicity, we assume each tier only has one server.[1] We only consider the soft resources that are responsible to communicate with other tiers. Concretely, each server has a thread pool to receive/process the incoming requests and a connection pool to communicate with the downstream tier. The size of thread pool and connection pool in tier $T_i$ is $ThreadPool_i$ and $ConnPool_i$. We omit other potential soft resources inside a server since they are implementation specific. Table 1 shows the description of all the parameters in our model.

Based on the Little's Law and Forced Flow Law, we can get the following equations for each tier:

$$Q_i = TP_i * RT_i \tag{1}$$

$$TP_i = TP_j * V_{ij}, \tag{2}$$

where $Q_i$, $TP_i$, and $RT_i$ are the number of queued requests, the average throughput, and the average response time in tier $T_i$. $V_{ij}$ is the visit ratio between the $T_i$ and $T_j$.

Combining Equations (1) and (2), we have the following:

$$\begin{aligned} Q_j &= Q_i * \frac{TP_j}{TP_i} * \frac{RT_j}{RT_i} \\ &= Q_i * V_{ji} * \frac{RT_j}{RT_i} \end{aligned} \tag{3}$$

[1]Multiple servers in one tier can be viewed as one big server with all the soft resources of the same type added together.

Let $RT_{ratio}^{ij}$ denote the ratio between $RT_i$ and $RT_j$, the above equation can turn into the following equation:

$$Q_j = Q_i * V_{ji} * RT_{ratio}^{ji} \tag{4}$$

Equation (4) shows that in an n-tier system, once we determine the number of queued requests in tier $T_i$, we are able to infer the number of queued requests in any other tier $T_j$, given the visit ratio $V_{ji}$ and response time ratio $RT_{ratio}^{ji}$. We know that $V_{ji}$ depends on the workload characteristics. Both $V_{ji}$ and $RT_{ratio}^{ji}$ can be measured in the runtime based on each server's processing log, which usually records the start time and the response time of each processed request. For example, the visit ratio $V_{ji}$ is approximated as the ratio of the number of requests between tier $T_i$ and $T_j$.

Assuming that each queued request in a server requires a dedicated thread, Equation (4) can be transformed to:

$$ThreadPool_j = ThreadPool_i * V_{ji} * RT_{ratio}^{ji} \tag{5}$$

Other than the thread pool size, the connection pool size in tier $T_j$ can also be derived from Equation (3). In this case, we need to record the timestamps when a connection is fetched by a worker thread and when the same connection is released back to the connection pool. We denote the time gap as the connection busy time $RT_i^{conn}$. By replacing $RT_i$ with $RT_i^{conn}$ in Equation (3), we can get the following:

$$\begin{aligned} ConnPool_j &= Q_i * V_{ji} * \frac{RT_j^{conn}}{RT_i} \\ &= ThreadPool_i * V_{ji} * \frac{RT_j^{conn}}{RT_i} \end{aligned} \tag{6}$$

Combining Equations (5) and (6), we are able to determine the relationship between thread pool size and connection pool size in different tiers, given the proper measurement of request visit ratio and response time of each tier.

We note that the above general soft resource aware model does not make any assumptions on the implementation of the benchmark application as long as the application uses thread-based servers to process business logic and connection pools for inter-tier communication. In fact, the model can be applied to any multi-stage execution pipelined systems with synchronous/blocking remote function invocation. Concretely, the parameters of the general model can be derived from detailed measurements from any specific n-tier benchmark applications (e.g., RUBBoS, RUBiS [45], CloudStone [48]). This general model characterizes the relationship of appropriate soft resource allocations between any two tiers in an arbitrary n-tier system, providing a solid foundation for our soft resource allocation algorithm to derive the optimal soft resource allocation in each tier of the target system, which is the topic of the following section.

### 4.3 Soft Resource Allocation Algorithm

The model described in the previous section only quantifies the relationship of soft resource allocations in different tiers; however, how to determine the "optimal" soft resource allocation in each tier is still undetermined. In this section, we describe a soft resource allocation algorithm based on the previous model. Our algorithm is based on the following three assumptions:

(1) Only a single critical hardware resource is in the system.
(2) When the system is saturated, our monitoring tools are able to identify the critical hardware resource (e.g., C-JDBC CPU).
(3) Response time of every request is logged in every component server.

The first assumption is to avoid complex multi-bottleneck scenarios. In a multi-bottleneck scenario, multiple hardware resources may saturate in a fast alternating pattern (e.g., millisecond lifespan) due to the inter-resource dependencies [38, 55]. In such a case, the average utilization of each involved hardware resource can be far from saturation (e.g., 60%) while the system already achieves the maximum throughput. Thus, the key challenge in a multi-bottleneck scenario is to identify the involved critical hardware resources that participate in the fast alternating pattern. However, normal monitoring tools with coarse monitoring granularity (e.g., seconds or minutes) may fail to detect any hardware resources that present transient saturation. Designing low-overhead fine-grained monitoring tools to detect transient resource saturation in an n-tier system is the key to expend our solution to a multi-bottleneck scenario, which is beyond the scope of this article. The latter two assumptions assume that we have proper monitoring tools (e.g., collectl and Log4j) to record both the system and the application level events during experiments.

The algorithm to find a good soft resource allocation has the following three steps (pseudo-code in Algorithm 1):

(1) **Identify the critical hardware resource.** This step identifies the hardware resource that saturates first as the workload increases. Such a hardware resource is critical because it limits the entire system throughput.
(2) **Infer the "optimal" soft resource allocation of the server that uses the critical hardware resource.** This step is to find the just-right allocation inside the bottleneck server to avoid either under- or over-utilization of the critical hardware resource in the system.
(3) **Infer a good allocation of other soft resources.** This step allocates soft resources of the tiers other than the bottleneck tier. We use the soft resource aware model (Equations (5) and (6)) that we derived in Section 4.2 to infer the appropriate soft resource allocations in other tiers based on detailed measurements and the soft resource allocation in the bottleneck tier.

Currently, the above three steps are implemented in an offline style during the system profiling phase by exploiting our automated experimental infrastructure [25] (the overhead analysis is in Section 4.5). In the following, we will explain each of the three steps in more detail.

*4.3.1 Identifying the Critical Hardware Resource.* This step is to identify the critical hardware resource of the system by increasing the workload gradually until the system reaches the highest throughput. $H_0$ and $S_0$ represent the initial hardware configuration and soft resource allocation, respectively. Function $RunExperiment(H, S, workload)$ runs an experiment with the given hardware/software configuration at specific $workload$. During the experiment period, our monitoring infrastructure monitors all the hardware and soft resources and the saturated ones are recorded in $H_b$ and $S_b$, respectively. We increase the workload one $step$ each time until the system reaches the highest throughput. At this moment, either some hardware or soft resource limits the continual increase of the system throughput. If $H_b$ is not empty, this step of the algorithm successfully identifies the critical hardware resource and returns. Otherwise, if $S_b$ is not empty, the system encounters the soft resource bottleneck, causing the under-utilization of the critical hardware resource. In this case, we double all the soft resource allocations and repeat the experiment.

*4.3.2 Inferring an "Optimal" Allocation of Soft Resources in the Bottleneck Tier.* This step infers a good allocation of soft resources (e.g., threads) in the bottleneck tier that can saturate the critical hardware resource without causing additional overhead. In a thread-based server, each job requires one dedicated thread to process it; the optimal threads number should equal the minimum

**ALGORITHM 1:** Pseudo-code for soft resource allocation

```
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          /* critical hardware resource found */
9          S_reserve = S;
10         return B_h;
11     else if (B_s ≠ φ) then
12         /* soft resource bottleneck */
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 /* find the minimum workload */
30 WL_min = i − 1;
31 minJobs = RTT[WL_min] * TP[WL_min];
32 criServer.threadpool = minJobs;
33 criServer.Connpool = minJobs;
34 procedure CalculateMinAllocation
35 for server in front tiers do
36     /* apply soft resource allocation model */
37     server.threadpool= minJobs * V_visitRatio * RTT_ratio;
38     server.Connpool = minJobs * V_visitRatio * RTT_ratio;
39 end
40 for server in end tiers do
41     server.threadpool= minJobs;
42     server.Connpool = minJobs;
43 end
```

concurrent jobs that saturate the critical hardware resource in the server. In this case, the critical hardware resource will neither be under- nor over-utilized.

Applying Little's law (see Equation (1)), we can infer the average number of jobs inside a server based on the average server throughput and response time. Both the two metrics can be easily obtained from the server's log. Therefore, to find the minimum concurrent jobs (*minJobs*) that

saturate the bottleneck server, we need to determine the minimum workload ($WL_{min}$) that can just achieve the highest system throughput. This can be achieved by gradually increasing the workload until the system reaches the highest throughput. Line 31 of the pseudo-code shows that *minJobs* can be calculated given the throughput and response time of the critical server at workload $WL_{min}$.

*4.3.3 Calculating a Good Allocation of Other Soft Resources.* This step of the algorithm applies Equations (5) and (6) to calculate appropriate soft resource allocations in other tiers based on the "optimal" soft resource allocations in the critical tier (determined in the second step). We note that the calculated soft resource allocations in other tiers are just the minimum that allows the critical hardware resource in the bottleneck tier to be saturated by the concurrent jobs (*minJobs*). In practice, there are two optimization techniques to further improve the algorithm effectiveness.

First, to handle the naturally bursty workload from clients, the front-most tier (e.g., Apache web server) should provide more soft resources than the minimum to act as a buffer that stabilizes bursty workload to the downstream tiers (see Section 3.3). An appropriate value depends on the burstiness level of the workload. Our experiments show that allocating 3 to 4 times of the calculated minimum threads in the front-most Apache is able to achieve a good buffering effect for the default RUBBoS workload, the request rate of which follows a normal distribution with the mean value related to the number of clients. We note that randomly allocating a high number of threads (beyond 3 to 4 times) may again degrade the efficiency of the Apache server because threads will consume resources such as the main memory and CPU cache even if they are idle, which may cause side effects such as memory or cache thrashing.

Second, the soft resource allocations in the tiers behind the bottleneck tier (e.g., MySQL in 1/4/1/4) can directly assign the value of *minJobs*. This is because according to Equations (5) and (6), the minimum soft resource allocation in these downstream tiers are definitely less than *minJobs*.[2] Since these downstream tiers are not the bottleneck, slightly over-allocation of soft resources does not hurt the overall system performance while such optimization can speed up the soft resource allocation process for the whole system.

### 4.4 Validation of the Algorithm

In this section, we apply our iterative algorithm to find the "optimal" (or near-optimal) soft resource allocation under different hardware configurations and also validate the recommended allocation through extensive experiments.

Table 2 summarizes the output of the three procedures of the algorithm for the two hardware configurations that we have evaluated before: 1/2/1/2 and 1/4/1/4. The first procedure of the algorithm reports that the critical hardware resource of the 1/2/1/2 configuration is Tomcat CPU while C-JDBC CPU for the 1/4/1/4 configuration. The second procedure reports the average response time and throughput for each individual server under the minimum saturation workload, and infers the "optimal" allocation of soft resources in the bottleneck tier. In this procedure, we turned on the logging function of each server in order to record the start time and response time of each request, which is used to calculate the average response time and throughput of each server. The third procedure calculates the minimum thread/conn pool size for other tiers based on Equations (5) and (6).

*4.4.1 Validation of 1/2/1/2 Case.* Since the combination of soft resource allocation space is too large in a 4-tier system, we validate the recommended allocation of each individual soft resource one by one. Concretely, we vary the allocation of each individual soft resource and check whether

[2]Given that tier $j$ is behind tier $i$, Figure 10 shows that both ($V_{ji} * RT_{ratio}^{ji}$) and ($V_{ji} * \frac{RT_j^{conn}}{RT_i}$) are less than 1.

Table 2. Algorithm Output for the Hardware Configuration 1/2/1/2 and 1/4/1/4

| 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] | | 15400 | | | | | 19200 | |
| RT [s] | 0.050 | 0.028 | | | 0.045 | 0.020 | 0.005 | |
| $RT^{conn}$ [s] | 0.034 | 0.013 | | | 0.023 | 0.009 | 0.003 | |
| TP [Reqs/s] | 2101 | 2101 | | | 2625 | 2625 | 8426 | |
| Visit ratio $V$ | 1 | 1 | 3.21 | 3.21 | 1 | 1 | 3.21 | 3.21 |
| $minJobs$ in the bottleneck tier | | **59** | | | | | **42** | |
| Size of total threads | 105 | 60 | 60 | 60 | 118 | 52 | 42 | 42 |
| **Size of individual thread pool** | **105** | **30** (×2) | **60** | **30** (×2) | **118** | **13** (×4) | **42** | **11** (×4) |
| Size of total connections | 73 | 26 | 60 | \ | 60 | 32 | 17 | \ |
| **Size of individual connection pool** | **73** | **13** (×2) | **30** | \ | **60** | **8** (×4) | **17** | \ |

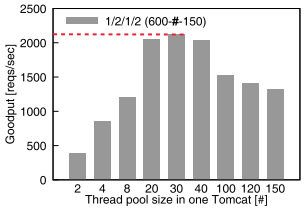
The results show that as the hardware configuration changes, the "optimal" allocation of soft resources in each tier also changes.

the algorithm recommended allocation is able to achieve the best performance under the same workload. When validating one specific soft resource, we carefully choose the allocation for the other soft resources in order to avoid being the main parameter affecting the system performance. More comprehensive evaluation such as exploring larger allocation space or varying the allocation of multiple soft resources simultaneously will be our future work.

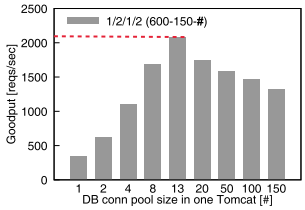
Figure 12(a)–(c) shows the validation results for three individual soft resources: the Tomcat threads, Tomcat database connections, and Apache threads. The 4-tier system is always at a high workload 16,000 to make sure that the critical hardware resource (Tomcat CPU) can be fully utilized given that soft resources are sufficient. Figure 12(a) and (b) show that the recommended allocation given by our algorithm indeed outperforms all the other allocation cases. For example, Figure 12(a) shows the goodput comparison when we increase the thread pool size in Tomcat (the bottleneck tier) from 4 to 100. The workload is 16,000, which is able to saturate the system when there is no soft resource bottleneck. This figure shows that the optimal size of one Tomcat thread pool is 30, matching the output given by our algorithm as shown in Table 2. Figure 12(c) shows that the system goodput achieves the highest when Apache threads reach 300, three times the recommended value (105). This is because we need to allocate an adequatly high number of threads in the front-most Apache to achieve a good buffering effect and provide stable concurrent requests to downstream tiers, as long as the Apache web server does not become the system bottleneck.

We also demonstrate the varying impact of soft resource allocations on n-tier application performance when the response time threshold (the SLA objective) changes in Figure 13. For example, the system with the recommended threads pool size in Tomcat (30) outperforms 60% in goodput than the over-allocation case (150) in Figure 12(a); such a performance difference is enlarged to 214% when the response time threshold changes to 500 milliseconds in Figure 13(a). Such a result shows that while the recommended soft resource allocation still outperforms the other testing allocation cases, the performance gap tends to be wider when the target response time threshold is small (e.g., 500 milliseconds in Figure 13) on the same hardware configuration.

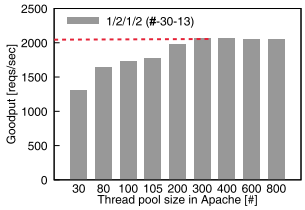
*4.4.2 Validation of 1/4/1/4 Case.* The critical hardware resource under 1/4/1/4 hardware configuration is C-JDBC CPU, however, we can not directly validate the recommended threads number



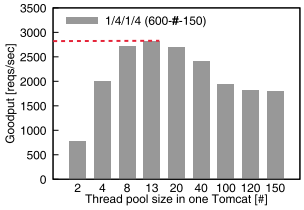

(a) Validation of thread pool size in Tomcat. 30 is recommended under the 1/2/1/2 hardware configuration.



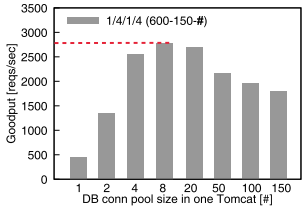

(b) Validation of DB connection pool size in Tomcat. 13 is recommended.



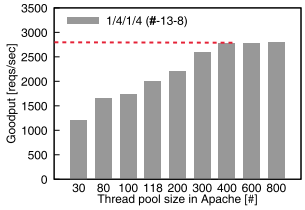

(c) Validation of buffering effect of Apache threads. 105 is recommended. But 300 is needed for highest goodput.




(d) Validation of thread pool size in Tomcat. 13 is recommended under the 1/4/1/4 hardware configuration.




(e) Validation of DB connection pool size in Tomcat. 8 is recommended.



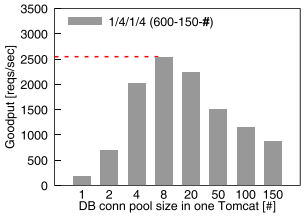

(f) Validation of buffering effect of Apache threads. 118 is recommended. But 400 is needed for highest goodput.

Fig. 12. Validation of each individual soft resource allocation recommended by our algorithm for the hardware configuration 1/2/1/2 (the first row subfigures) and 1/4/1/4 (the second row subfigures) with a 3-second target response time threshold. Each subfigure validates one parameter; the other two parameters are chosen to be abundant in order to avoid being the main parameter limiting the system performance. (a), (b), (d), and (e) show that the recommended allocation of the corresponding soft resource is able to achieve the highest system goodput; either under-allocation or over-allocation leads to degraded system goodput (see Section 3.1 and 3.2). In addition, (c) and (f) show that 3 to 4 times of the recommended allocation of threads in Apache (the front-most tier) are needed to achieve a good buffering effect due to the natural burstiness of n-tier application workload (see Section 3.3).

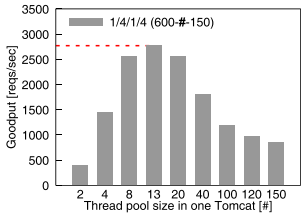
in C-JDBC because there is no explicit thread pool in the current C-JDBC server implementation. [3] So, for 1/4/1/4, we also show the validation for Tomcat threads, Tomcat database connections, and Apache threads as we did in Section 4.4.1. Figure 12(d) and (e) show that the recommended soft resource allocations (13 threads and 8 database connections in each Tomcat) by our algorithm actually achieve the highest goodput comparing to other allocation cases. Figure 12(f) shows that, compared to the 1/2/1/2 case, 400 Apache threads are needed to achieve a good buffering effect, almost 4 times as the recommended value (116). This is because the 1/4/1/4 case under validation is at a much higher workload (20,000) compared to the 1/2/1/2 case, thus larger buffer size is needed in the front-most tier.

*4.4.3 Global Optimal Allocation.* Astute readers may question the wisdom of validating each recommended allocation of soft resources (i.e., Apache threads, Tomcat threads, and database connections) one by one as shown in Figure 12. What about the global optimal soft resource allocation? In fact, Algorithm 1 already gives the answer. The principle of a global optimal soft resource allocation is to most efficiently utilize the critical (bottleneck) hardware resource in the system, neither

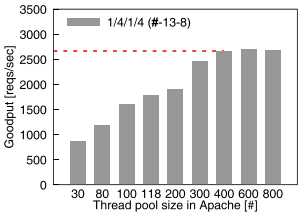
[3] A C-JDBC server launches a new request handling thread for each new database connection from the upstream Tomcat.



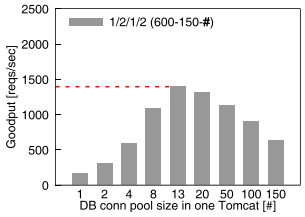

(a) Validation of thread pool size in Tomcat. 30 is recommended under the 1/2/1/2 hardware configuration.



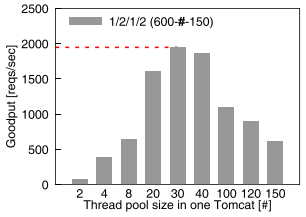

(b) Validation of DB connection pool size in Tomcat. 13 is recommended.



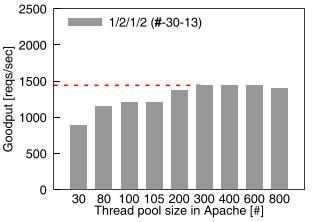

(c) Validation of buffering effect of Apache threads. 105 is recommended. But 300 is needed for highest goodput.




(d) Validation of thread pool size in Tomcat. 13 is recommended under the 1/4/1/4 hardware configuration.




(e) Validation of DB connection pool size in Tomcat. 8 is recommended.

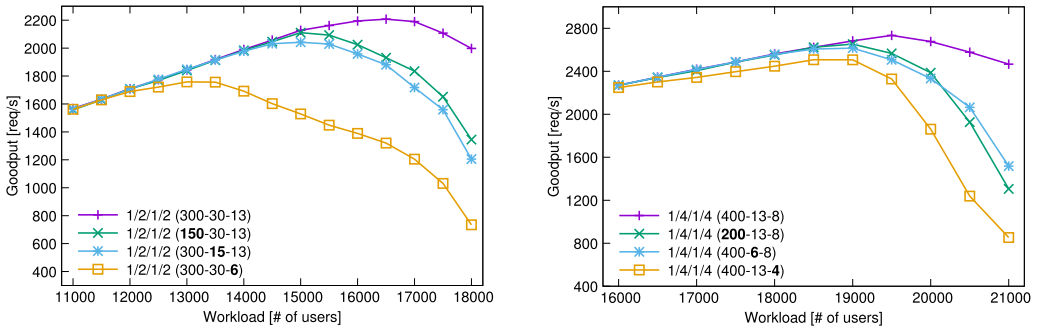



(f) Validation of buffering effect of Apache threads. 118 is recommended. But 400 is needed for highest goodput.

Fig. 13. Validation results using a 500-millisecond target response time threshold. Compared to the 3-second response time threshold case (Figure 12), this set of results show that smaller target response time threshold (500-millisecond in this case) tends to lead to higher performance impact of the recommended soft resource allocation (e.g., Figure 12(a) vs. 13(a)) on the same hardware configuration (either 1/2/1/2 or 1/4/1/4).

under- nor over-utilize it, in order to achieve the highest system goodput. In the 1/2/1/2 case, the critical hardware resource is Tomcat CPU, while it is CJDBC CPU in the 1/4/1/4 case. The algorithm recommended allocation in each tier as shown in Table 2 is the minimum that allows the concurrent jobs to saturate the critical hardware resource in the system. Thus, controlling either the Tomcat thread pool or the database connection pool is able to achieve the highest system goodput. For example, Figure 12(a) and (b) show that either allocating 30 Tomcat threads or 13 database connections allows the 1/2/1/2 system to achieve the highest goodput (about 2,200reqs/sec), as long as the allocation of other soft resources is abundant. We note that the allocation of threads in the front-most Apache is special since it also serves as a buffer to stabilize the bursty workload from clients; thus, more than the recommended minimum is needed to serve as an effective buffer, as shown in Figure 12(c). The same analysis also applies to the 1/4/1/4 case as shown in Figure 12(d)–(f).

On the other hand, allocating less than the algorithm-recommended minimum of any soft resource will lead to degraded system goodput. This is because such an under-allocation will prevent the necessary number of concurrent jobs that saturate the critical hardware resource in the system. Figure 14 validates this point for both 1/2/1/2 and 1/4/1/4. In both cases, we half the size of the recommended value of each of the three parameters (Apache threads, Tomcat threads, and DB connections) one by one. The achieved system goodput is significantly lower than that of the algorithm-recommended global optimal case (the top purple line) under the high workload range when the system is approaching saturation or slightly overloaded.




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

Fig. 14. Validation of the global soft resource allocation recommended by our algorithm for the hardware configuration 1/2/1/2 (the left) and 1/4/1/4 (the right). The soft resource allocation recommended by our algorithm (the top purple line) achieves the highest system goodput under different hardware configurations. For both (a) and (b), we only compare with the cases of under-allocation (half size of the recommended value) of each parameter at a time. This is because the abundant allocation (more than the recommended value) of any one of three parameters will still allow the system to achieve the highest system goodput, as shown in Figure 12.

### 4.5 Discussion

**Overhead analysis of soft resource allocation algorithm.** Since our algorithm is implemented in an offline style, the performance overhead of applying the algorithm to the runtime system is minimum. Concretely, during the offline training phase, our proposed algorithm is able to quickly generate a near-optimal soft resource allocation policy for each hardware configuration (e.g., 1/2/1, 1/2/1/2, 1/4/1/4) by exploiting our automated Elba [25] experimental infrastructure, which is able to generate scripts to automate the system deployment/configuration, experiments execution, data collection, data analysis, and visualization. For example, to generate a near-optimal soft resource allocation for the 1/2/1/2 RUBBoS configuration in Table 2, the algorithm runs 10 workload steps to identify the critical hardware resource, each of which runs about 10 minutes. So, in total, 100 minutes is needed to generate a near-optimal soft resource allocation for this specific hardware configuration (1/2/1/2).

These offline-generated soft resource allocation policies will be adopted for online system reconfiguration during the system scaling phase in the cloud. Concretely, when the system scaling out/in is triggered during runtime, the application scaling controller just needs to dynamically change the soft resource allocation of the involved servers based on the offline-generated soft resource allocation policies. We note that these offline-generated soft resource allocation policies need to be re-trained when the workload characteristic is detected to change significantly over time (e.g., from CPU-intensive to I/O-intensive). Although web application workload is naturally bursty (e.g., number of users), the workload characteristic (transaction mix) is relatively stable. So we expect the regeneration of soft resource allocation policies would not happen frequently for real production n-tier systems.

**Soft resource allocation algorithm for other types of soft resources.** Our definition of soft resources in this article is to refer to the software components such as threads and TCP connections that use hardware resources or synchronize the use of both soft and hardware resources. For example, a lock is one of the soft resources that synchronizes the use of data structures and CPU. In general, the function of soft resources is to facilitate the sharing of the hardware resources through concurrency. Regardless of the types of soft resources, the optimal soft resource allocation/

configuration is to use the underlying hardware resources most efficiently with minimum overhead. This is the key principle of the second and the third steps of our proposed soft resource allocation algorithm described in Sections 4.3.2 and 4.3.3.

**Multi-core scenario.** The essential problem of soft resource allocation/configuration is how to optimize the request processing concurrency (controlled by soft resource allocation) in each server/tier of an n-tier system to most efficiently utilize the underlying hardware resources with minimum overhead. Thus, the difference between VMs with single-core and multi-core relies on the number of soft resources (e.g., server threads) needed to most efficiently utilize the capacity of the CPU resources (assume CPU is the bottleneck resource). We note that a multi-core CPU may not be able to be fully utilized by simply increasing the number of threads due to the inter-thread dependency and synchronization issue, especially when the number of cores is large (e.g., more than eight [14]), but that is beyond the scope of this article.

**Dealing with asynchronous event-driven servers.** We note that some recent web applications started to use event-driven asynchronous component servers (e.g., Nginx [42] and Node.js [33]), the performance of which are less affected by the request processing concurrency; however, thread-based servers are still widely used in today's production web systems and sometimes difficult to be replaced by their asynchronous counterparts (e.g., database servers) due to the complexity of the asynchronous programming model. We believe our conclusions on soft resource allocations and the general soft resources aware model are still important contributions for building truly scalable n-tier systems in the cloud.

## 5 RELATED WORK

In this section, we classify previous work on system performance optimization into four main categories—server design, analytical models, feedback control, and experimental-based software engineering approach.

**Asynchronous event-driven server design** has been explored for high performance servers [30, 46], with Nginx [42] and Node.js [33] as a few emerging asynchronous web servers. Some previous work [10, 57] even advocates a hybrid design of combining both threads and events for high-performance servers. Nevertheless, many mainstream Internet servers such as Apache and Tomcat still adopt the classic thread-based architecture because of its simple and natural programming style. Regardless of which design that an Internet server (either thread-based, asynchronous event-driven, or the hybrid design server) adopts, soft resource allocation controls the request processing concurrency and has a significant impact on server/system performance. While previous research typically focuses on the performance impact of single soft resource allocation (e.g., threads) on a single web server [10, 26, 46, 54], this article focuses on highly distributed n-tier applications with inter-dependent servers.

**Analytical models** have been proposed for system performance prediction and the optimal resource allocation [1, 3, 8, 12, 15, 18, 21, 40, 52]. For example, Franks et al. [21] propose a layered queuing network model that characterizes the dependencies of software and hardware resources across nodes in a distributed system. Urganonkar et al. [52] propose a queue-based model to capture the performance characteristics of each tier and application idiosyncrasies. Bhimani et al. [12] propose a performance approximation approach to model the computing performance of iterative and multi-stage applications using the Stochastic Markov Model and Machine Learning Model. These analytical models generally extend the classic queuing network model and are meant to capture some key features that affect the target system performance such as the dynamic changes of workload characteristics, concurrency bound, or server replicas. These models, although they have been shown to work very well in some specific scenarios, may not apply to realistic n-tier applications because of certain strict assumptions. For example, these models do not consider the

practical factors such as non-linear multithreading overhead or JVM GC activities, which are very related to soft resource allocation and can significantly degrade server efficiency as shown in this article.

**Feedback-control** have been applied to adapt system resource provisioning based on run-time workload variation [4, 11, 22–24, 28, 31, 32, 34, 51, 58]. The feedback signal is usually generated based on certain resource utilization boundaries or a pre-defined SLA specification such as response time threshold. Previous work mainly focuses on how (e.g., live migration [43]) and when (e.g., pro- [22, 24] and re-active scaling [23]) to add or remove hardware resources such as virtual machines/Docker containers to change the system capacity. For example, Amazon provides Amazon Web Services (AWS) Auto Scaling [5] in its cloud platform, which adopts a re-active scaling strategy to enable the dynamic scaling of virtual machines based on the average CPU utilization measured by its monitoring tool—Amazon CloudWatch [6]. Gandhi et al. [22, 24] dynamically allocate system capacity (number of servers) by taking advantage of both pro- and re-active scaling strategies based on an offline-trained workload forecasting model. Alsarhan et al. [4] use reinforcement learning (RL) to derive scaling policies (number of VMs) that can adapt to system changes in order to guarantee the QoS for all client classes. Nevertheless, how to re-adapt request processing concurrency to match the hardware provisioning changes is usually neglected. As shown in this article, request processing concurrency controlled by soft resources such as server threads and connections have a significant impact on n-tier web application performance. Thus, when a scaling action is triggered, reallocating soft resources is necessary to maximize the efficiency of the underlying hardware resources.

**Software performance engineering approaches** for optimal system configuration (both hardware and software) are closest to our study [35, 47, 49, 50, 53, 59, 60]. For example, Zheng et al. [59] design an automation framework to generate configurations automatically for a cluster of servers in a web system by using a parameter dependency graph derived from runtime measurement data. Zhu et al. [60] propose an automated approach for optimal system configuration given the conditions of limited computing resources and a fixed application workload. Tang et al. [50] present Facebook's holistic configuration management stack for managing applications' dynamic runtime configuration, for example, gating product rollouts, managing application-level traffic, and running A/B testing experiments. However, these previous approaches, in general, do not seriously consider the non-trivial dependencies among soft and hardware resources provisioning.

## 6 CONCLUSIONS

In this article, we studied the impact of soft resource allocation on n-tier application performance. We found that the system goodput (requests within SLA bound) is sensitive to soft resource allocations at high concurrency levels; given a good soft resource allocation in one hardware configuration, it may become inappropriate (either under or over-allocation) when the hardware configuration scales (Section 2.3). Concretely, we showed that too low allocation of Tomcat threads (Section 3.1) or Apache threads (Section 3.3) degrades system goodput several tens of percent by underutilizing the critical hardware resource in different manners. On the other hand, over-allocation of Tomcat DB connections causes significant overhead on the downstream C-JDBC CPU and also degrades system goodput several tens of percent (Section 3.2). We note that we have conducted similar experiments on Emulab with much older hardware configuration and software versions [56], and the same conclusion has been reached as in our new VMware ESXi cluster environment, given all the hardware and software upgrades since 2011.

To achieve a good soft resource allocation, we provide a novel model of n-tier systems with soft resources as explicit components, upon which we described a practical soft resource allocation algorithm followed by extensive validation experiments (Section 4). More generally, to truly scale

complex systems such as n-tier applications, soft resources have to be treated as first-class citizens (analogous to hardware) during the system scaling management.

## REFERENCES


[1] K. Aberer, T. Risse, and A. Wombacher. 2001. Configuration of distributed message converter systems using performance modeling. In *Conference Proceedings of the 2001 IEEE International Performance, Computing, and Communications Conference (Cat. No.01CH37210)*. 57–67. DOI: https://doi.org/10.1109/IPCCC.2001.918636

[2] Stephen Adler. 1999. The slashdot effect: An analysis of three Internet publications. *Linux Gazette* 38, 2 (1999).

[3] Ahsan Ali, Riccardo Pinciroli, Feng Yan, and Evgenia Smirni. 2018. CEDULE: A scheduling framework for burstable performance in cloud computing. In *2018 IEEE International Conference on Autonomic Computing (ICAC)*. IEEE, 141–150.

[4] Ayoub Alsarhan, Awni Itradat, Ahmed Y. Al-Dubai, Albert Y. Zomaya, and Geyong Min. 2018. Adaptive resource allocation and provisioning in multi-service cloud environments. *IEEE Trans. Parallel Distrib. Syst.* 29, 1 (2018), 31–42.

[5] Amazon Web Services. 2017. Amazon Auto Scaling. Retrieved August 8, 2018 from https://aws.amazon.com/documentation/autoscaling.

[6] Amazon Web Services. 2018. Amazon CloudWatch. Retrieved August 8, 2018 from https://aws.amazon.com/cloudwatch/.

[7] Amazon Web Services. 2019. Amazon EC2 Dedicated Hosts Pricing. Retrieved February 8, 2019 from https://aws.amazon.com/ec2/dedicated-hosts/pricing/.

[8] Danilo Ardagna, Giuliano Casale, Michele Ciavotta, Juan F. Pérez, and Weikun Wang. 2014. Quality-of-service in cloud computing: Modeling techniques and their applications. *J. Internet Serv. Appl.* 5, 1 (2014), 11.

[9] Berk Atikoglu, Yuehai Xu, Eitan Frachtenberg, Song Jiang, and Mike Paleczny. 2012. Workload analysis of a large-scale key-value store. In *ACM SIGMETRICS Performance Evaluation Review*, Vol. 40. ACM, 53–64.

[10] Vicenç Beltran, Jordi Torres, and Eduard Ayguadé. 2008. Understanding tuning complexity in multithreaded and hybrid web servers. In *IEEE International Symposium on Parallel and Distributed Processing (IPDPS'08)*. IEEE, 1–12.

[11] Daniel S. Berger, Benjamin Berg, Timothy Zhu, Siddhartha Sen, and Mor Harchol-Balter. 2018. RobinHood: Tail latency aware caching–dynamic reallocation from cache-rich to cache-poor. In *13th {USENIX} Symposium on Operating Systems Design and Implementation (OSDI'18)*. 195–212.

[12] Janki Bhimani, Ningfang Mi, Miriam Leeser, and Zhengyu Yang. 2017. FIM: Performance prediction for parallel computation in iterative data processing applications. In *IEEE 10th International Conference on Cloud Computing (CLOUD'17)*. IEEE, 359–366.

[13] Eric Boutin, Jaliya Ekanayake, Wei Lin, Bing Shi, Jingren Zhou, Zhengping Qian, Ming Wu, and Lidong Zhou. 2014. Apollo: Scalable and coordinated scheduling for cloud-scale computing. In *OSDI*, Vol. 14. 285–300.

[14] Silas Boyd-Wickizer, Austin T. Clements, Yandong Mao, Aleksey Pesterev, M. Frans Kaashoek, Robert Morris, Nickolai Zeldovich, et al. 2010. An analysis of Linux scalability to many cores. In *OSDI*, Vol. 10. 86–93.

[15] Rodrigo N. Calheiros, Enayat Masoumi, Rajiv Ranjan, and Rajkumar Buyya. 2015. Workload prediction using ARIMA model and its impact on cloud applications' QoS. *IEEE Trans. Cloud Comput.* 3, 4 (2015), 449–458.

[16] Emmanuel Cecchet, Anupam Chanda, Sameh Elnikety, Julie Marguerite, and Willy Zwaenepoel. 2003. Performance comparison of middleware architectures for generating dynamic web content. In *ACM/IFIP/USENIX International Conference on Distributed Systems Platforms and Open Distributed Processing*. Springer, 242–261.

[17] Emmanuel Cecchet, Julie Marguerite, and Willy Zwaenepole. 2004. C-JDBC: Flexible database clustering middleware. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference (ATEC'04)*. USENIX Association, Berkeley, CA, 26–26.

[18] Eli Cortez, Anand Bonde, Alexandre Muzio, Mark Russinovich, Marcus Fontoura, and Ricardo Bianchini. 2017. Resource central: Understanding and predicting workloads for improved resource management in large cloud platforms. In *Proceedings of the 26th Symposium on Operating Systems Principles (SOSP'17)*. ACM Press, New York, New York, 153–167. DOI: https://doi.org/10.1145/3132747.3132772

[19] Kristal Curtis, Peter Bodík, Michael Armbrust, Armando Fox, Mike Franklin, Michael Jordan, and David Patterson. 2010. *Determining SLO Violations at Compile Time.*

[20] Christina Delimitrou and Christos Kozyrakis. 2014. Quasar: Resource-efficient and QoS-aware cluster management. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'14)*. ACM, New York, NY, 127–144.

[21] Greg Franks, Tariq Al-Omari, Murray Woodside, Olivia Das, and Salem Derisavi. 2009. Enhanced modeling and solution of layered queueing networks. *IEEE Trans. Software Eng.* 35, 2 (2009), 148–161.

[22] Anshul Gandhi, Yuan Chen, Daniel Gmach, Martin Arlitt, and Manish Marwah. 2012. Hybrid resource provisioning for minimizing data center SLA violations and power consumption. *Sustainable computing: Informatics and systems* 2, 2 (Jun 2012), 91–104. DOI: https://doi.org/10.1016/J.SUSCOM.2012.01.005

[23] Anshul Gandhi, Mor Harchol-Balter, Ram Raghunathan, and Michael A. Kozuch. 2012. AutoScale: Dynamic, robust capacity management for multi-tier data centers. *ACM Trans. Comput. Syst.* 30, 4, Article 14 (Nov. 2012), 26 pages.

[24] Anshul Gandhi, Yuan Chen, Daniel Gmach, Martin Arlitt, and Manish Marwah. 2011. Minimizing data center SLA violations and power consumption via hybrid resource provisioning. In *2011 International Green Computing Conference and Workshops*. IEEE, Orlando, FL, 1–8. DOI: https://doi.org/10.1109/IGCC.2011.6008611

[25] Gatech. 2008. Elba: Automated Design/Configuration Evaluation and Tuning. Retrieved February 5, 2019 from https://www.cc.gatech.edu/systems/projects/Elba/index.html.

[26] Ashif S. Harji, Peter A. Buhr, and Tim Brecht. 2012. Comparing high-performance multi-core web-server architectures. In *Proceedings of the 5th Annual International Systems and Storage Conference (SYSTOR'12)*. ACM, New York, NY, Article 1, 12 pages.

[27] Gueyoung Jung, Kaustubh R. Joshi, Matti A. Hiltunen, Richard D. Schlichting, and Calton Pu. 2009. A cost-sensitive adaptation engine for server consolidation of multitier applications. In *Proceedings of the ACM/IFIP/USENIX 10th International Conference on Middleware (Middleware'09)*. Springer-Verlag, Berlin, 163–183.

[28] Jalal Khamse-Ashari, Ioannis Lambadaris, George Kesidis, Bhuvan Urgaonkar, and Yiqiang Zhao. 2018. An efficient and fair multi-resource allocation mechanism for heterogeneous servers. *IEEE Transactions on Parallel and Distributed Systems* 29, 12 (2018), 2686–2699.

[29] R. Kohavi and R. Longbotham. 2007. Online experiments: Lessons learned. *Comput.* 40, 9 (Sept. 2007), 103–105.

[30] Maxwell N. Krohn, Eddie Kohler, and M. Frans Kaashoek. 2007. Events can make sense. In *USENIX Annual Technical Conference*. 87–100.

[31] Gil Jae Lee and José A. B. Fortes. 2017. Hierarchical self-tuning of concurrency and resource units in data-analytics frameworks. In *IEEE International Conference on Autonomic Computing (ICAC'17)*. IEEE, 49–58.

[32] Zhao Lucis Li, Chieh-Jan Mike Liang, Wenjia He, Lianjie Zhu, Wenjun Dai, Jin Jiang, and Guangzhong Sun. 2018. Metis: Robustly tuning tail latencies of cloud systems. In *2018 {USENIX} Annual Technical Conference ({USENIX}ATC'18)*. 981–992.

[33] Linux Foundation. 2017. node.js. Retrieved August 8, 2018 from https://nodejs.org/en/.

[34] Richard T. B. Ma. 2016. Efficient resource allocation and consolidation with selfish agents: An adaptive auction approach. In *IEEE 36th International Conference on Distributed Computing Systems (ICDCS'16)*. IEEE, 497–508.

[35] Amiya K. Maji, Subrata Mitra, Bowen Zhou, Saurabh Bagchi, and Akshat Verma. 2014. Mitigating interference in cloud services by middleware reconfiguration. In *Proceedings of the 15th International Middleware Conference (Middleware'14)*. ACM, New York, NY, 277–288.

[36] Simon Malkowski, Markus Hedwig, Deepal Jayasinghe, Junhee Park, Yasuhiko Kanemasa, and Calton Pu. 2009. A new perspective on experimental analysis of n-tier systems: Evaluating database scalability, multi-bottlenecks, and economical operation. In *5th International Conference on Collaborative Computing: Networking, Applications and Worksharing, CollaborateCom 2009*. IEEE, 1–10.

[37] Simon Malkowski, Markus Hedwig, Deepal Jayasinghe, Calton Pu, and Dirk Neumann. 2010. CloudXplor: A tool for configuration planning in clouds based on empirical data. In *Proceedings of the 2010 ACM Symposium on Applied Computing (SAC'10)*. ACM, New York, NY, 391–398.

[38] Simon Malkowski, Markus Hedwig, and Calton Pu. 2009. Experimental evaluation of N-tier systems: Observation and analysis of multi-bottlenecks. In *Proceedings of the 2009 IEEE International Symposium on Workload Characterization (IISWC) (IISWC'09)*. IEEE Computer Society, Washington, DC, 118–127.

[39] Xiaoqiao Meng, Vasileios Pappas, and Li Zhang. 2010. Improving the scalability of data center networks with traffic-aware virtual machine placement. In *Proceedings of the 29th Conference on Information Communications (INFOCOM'10)*. IEEE Press, Piscataway, NJ, 1154–1162.

[40] Ningfang Mi, Giuliano Casale, Ludmila Cherkasova, and Evgenia Smirni. 2008. Burstiness in multi-tier applications: Symptoms, causes, and new models. In *Proceedings of the 9th ACM/IFIP/USENIX International Conference on Middleware (Middleware'08)*. Springer-Verlag New York, Inc., New York, NY, 265–286.

[41] Jeffrey C. Mogul. 2006. Emergent (mis) behavior vs. complex software systems. *ACM SIGOPS Operating Systems Review* 40, 4 (2006), 293–304.

[42] NGINX. 2017. nginx. Retrieved August 8, 2018 from http://nginx.org/.

[43] Hiep Nguyen, Zhiming Shen, Xiaohui Gu, Sethuraman Subbiah, and John Wilkes. 2013. AGILE: Elastic distributed resource scaling for infrastructure-as-a-service. In *Proceedings of the 10th International Conference on Autonomic Computing (ICAC'13)*. USENIX, San Jose, CA, 69–82.

[44] ObjectWeb Consortium. 2005. RUBBoS: Bulletin board benchmark. Retrieved June 11, 2018 from http://jmob.ow2.org/rubbos.html.

[45] ObjectWeb Consortium. 2009. RUBiS: Bidding System. Retrieved June 11, 2018 http://jmob.ow2.org/rubbos.html.

[46] David Pariag, Tim Brecht, Ashif Harji, Peter Buhr, Amol Shukla, and David R. Cheriton. 2007. Comparing the performance of web server architectures. In *Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007 (EuroSys'07)*. ACM, New York, NY, 231–243.

[47] Mukund Raghavachari, Darrell Reimer, and Robert D. Johnson. 2003. The deployer's problem: Configuring application servers for performance and reliability. In *Proceedings of the 25th International Conference on Software Engineering (ICSE'03)*. IEEE Computer Society, Washington, 484–489.

[48] Will Sobel, Shanti Subramanyam, Akara Sucharitakul, Jimmy Nguyen, Hubert Wong, Arthur Klepchukov, Sheetal Patil, Armando Fox, and David Patterson. 2008. Cloudstone: Multi-platform, multi-language benchmark and measurement tools for web 2.0. In *Proc. of CCA*, Vol. 8. 228–234.

[49] Monchai Sopitkamol and Daniel A. Menascé. 2005. A method for evaluating the impact of software configuration parameters on e-commerce sites. In *Proceedings of the 5th International Workshop on Software and Performance (WOSP'05)*. ACM, New York, NY, 53–64.

[50] Chunqiang Tang, Thawan Kooburat, Pradeep Venkatachalam, Akshay Chander, Zhe Wen, Aravind Narayanan, Patrick Dowell, and Robert Karl. 2015. Holistic configuration management at Facebook. In *Proceedings of the 25th Symposium on Operating Systems Principles (SOSP'15)*. ACM, New York, NY, 328–343.

[51] L. TomÃąs, E. B. Lakew, and E. Elmroth. 2016. Service level and performance aware dynamic resource allocation in overbooked data centers. In *2016 16th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*. 42–51.

[52] Bhuvan Urgaonkar, Giovanni Pacifici, Prashant Shenoy, Mike Spreitzer, and Asser Tantawi. 2005. An analytical model for multi-tier internet services and its applications. In *Proceedings of the 2005 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS'05)*. ACM, New York, NY, 291–302.

[53] Nedeljko Vasić, Dejan Novaković, Svetozar Miučin, Dejan Kostić, and Ricardo Bianchini. 2012. DejaVu: Accelerating resource allocation in virtualized environments. *SIGARCH Comput. Archit. News* 40, 1, 423–436.

[54] Rob von Behren, Jeremy Condit, Feng Zhou, George C. Necula, and Eric Brewer. 2003. Capriccio: Scalable threads for internet services. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP'03)*. ACM, New York, NY, 268–281.

[55] Qingyang Wang, Yasuhiko Kanemasa, Jack Li, Deepal Jayasinghe, Toshihiro Shimizu, Masazumi Matsubara, Motoyuki Kawaba, and Calton Pu. 2013. Detecting transient bottlenecks in n-tier applications through fine-grained analysis. In *Proceedings of the 2013 IEEE 33rd International Conference on Distributed Computing Systems (ICDCS'13)*. IEEE Computer Society, Washington, DC, 31–40.

[56] Qingyang Wang, Simon Malkowski, Yasuhiko Kanemasa, Deepal Jayasinghe, Pengcheng Xiong, Calton Pu, Motoyuki Kawaba, and Lilian Harada. 2011. The impact of soft resource allocation on n-tier application scalability. In *2011 IEEE International Parallel & Distributed Processing Symposium*. IEEE, 1034–1045.

[57] Matt Welsh, David Culler, and Eric Brewer. 2001. SEDA: An architecture for well-conditioned, scalable Internet services. In *Proceedings of the Eighteenth ACM Symposium on Operating Systems Principles (SOSP'01)*. ACM, New York, NY, USA, 230–243.

[58] Pengcheng Xiong, Zhikui Wang, Simon Malkowski, Qingyang Wang, Deepal Jayasinghe, and Calton Pu. 2011. Economical and robust provisioning of N-tier cloud workloads: A multi-level control approach. In *Proceedings of the 2011 31st International Conference on Distributed Computing Systems (ICDCS'11)*. IEEE Computer Society, Washington, DC, 571–580.

[59] Wei Zheng, Ricardo Bianchini, and Thu D. Nguyen. 2007. Automatic configuration of Internet services. In *Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007 (EuroSys'07)*. ACM, New York, NY, 219–229.

[60] Yuqing Zhu, Jianxun Liu, Mengying Guo, Yungang Bao, Wenlong Ma, Zhuoyue Liu, Kunpeng Song, and Yingchun Yang. 2017. Bestconfig: Tapping the performance potential of systems via automatic configuration tuning. In *Proceedings of the 2017 Symposium on Cloud Computing*. ACM, 338–350.

Received August 2018; revised February 2019; accepted April 2019