On Load Balancing: a Mix-aware Algorithm for Heterogenous Systems

Short Research Paper

Sebastiano Spicuglia
IBM Research Zurich Lab
Switzerland
seb@zurich.ibm.com

Mathias Björkqvist
IBM Research Zurich Lab
Switzerland
mbj@zurich.ibm.com

Lydia Y. Chen
IBM Research Zurich Lab
Switzerland
yic@zurich.ibm.com

Giuseppe Serazzi
Polytechnic Milano
Milano, Italy
serazzi@elet.polimi.it

Walter Binder
University of Lugano
Lugano, Switzerland
walter.binder@usi.ch

Evgenia Smirni
College of William and Mary
Virginia, US
esmirni@cs.wm.edu

ABSTRACT

Today's web services are commonly hosted on clusters of heterogenous servers, such as in computing clouds, whose computational and storage resources are based on different technologies. The workload is typically composed of different applications exhibiting disparate computation patterns (e.g., CPU-intensive or IO-intensive) that fluctuates both in terms of the volume and of their mix. The system complexity together with workload diversity further exacerbates the challenge of load balancing. This paper presents a novel mix-aware load-balancing algorithm, which aims at distributing requests of multiple applications in heterogenous servers such that the response times are minimized and system resources (e.g., CPU and disk) are equally utilized. To such an end, the presented algorithm tries not only to balance the total number of requests on each server, but also to shape requests received by each server into certain mixes, which are shown analytically optimal to minimize response times for individual servers. Our experimental results—based both on simulation and on a prototype system implementation in a computing cloud—show that our new algorithm achieves robust performance in most workload mixes as well as a consistent performance improvement in comparison with one of the most robust load-balancing schemes of the Apache server, i.e, bybusyness policy.

1. INTRODUCTION

To ensure scalability, today's web services are replicated and hosted on distributed systems that experience regular resource upgrades and are thus comprised of heterogenous servers. The employment of virtualization technology further amplifies the server heterogeneity [5], especially on hosting platforms shared with different service providers such as computing clouds. Web service applications are characterized not only by disparate resource requirements (e.g., CPU-intensive browsing service vs. I/O-intensive transaction service), but also by time-varying request workloads [4, 16]. Consequently, the overall system workloads fluctuate in terms of mixes of applications and the volume of requests [15]. The heterogeneity of servers, together with diversified applications with different workload characteristics, further exacerbate the challenges of load balancing. An example highlighting the importance of distributing the load across distributed services is the launch of the elastic load balancer in Amazon EC2 [1].

There is a large body of load balancing studies [3, 6, 7, 16] that mainly focus on homogenous systems and consider a single bottleneck resource where queues build up. Dispatching requests to the servers with the least number of outstanding requests, the so-called Join the Shortest Queue (JSQ) policy, has been shown very robust theoretically [9] and practically [11], to distribute the entire load across distributed servers. In a heterogenous system experiencing time-varying application mixes, such a policy can potentially lead to the situation where servers receive similar amounts of requests but servers with powerful CPU (resp. disk) process a lot of IO- (resp. CPU-) intensive requests. Clearly, depending on the received application mix, the use of servers with different bottleneck resources can result in very different performance, such as response times. Therefore, it is imperative for the load-balancing policy to distribute the server load evenly as well as the resource load, which is influenced by the application mix received at individual servers.

In this paper, we focus on the following questions: Is there an optimal application mix for each server which can lead to a low response time and all resources equally utilized? Moreover, is there a load-dispatching policy to control the received application mix for each server such that an optimal value is reached? Last, as JSQ has been shown very effective in balancing the overall loads on servers, can one leverage JSQ when designing a mix-aware load dispatching policy to further balance resource loads on heterogenous servers?

To this end, we develop a novel mix-aware load-balancing algorithm, which aims at balancing the server loads as well as their resource loads such that the global response time averaged over all applications is minimized. We explore the JSQ policy and the analytical results [14], which show that there is an optimal application mix for a single server, using a closed queueing network model. Our algorithm distributes requests based on two criteria: the number of outstanding requests and the difference between the optimal mix and the received application mix at each server. A goodness value, measuring the variability of queue length of outstanding requests across servers, is used to select the criteria. To further guide

ICPE September 16–20, 2013, Prague, Czech Republic

the choice of the threshold value, we provide a bounding analysis of queueing length based on the goodness. Essentially, our algorithm tries to balance the total number of outstanding requests on different servers using JSQ and the resource load by reaching the optimal application mix. We use an event-driven simulator to evaluate the scalability of the proposed algorithm, and a prototype to support the applicability of the proposed algorithm on a real system.

The original, scientific contributions of this paper are both analytical and practical. Our proposed load-balancing algorithm is designed for distributing time-varying requests from multiple applications, hosted on a set of heterogenous systems with multiple resources. Being aware of the queueing length and of the application mix, our algorithm is able to achieve a constant performance improvement, compared to one of load-balancing policies available on Apache web server, i.e., bybusyness policy.

This paper is organized as follows: The system architecture is explained in Section 2. The proposed mix-aware load-balancing algorithm is explained in Section 3. Section 4 contains the experimental results. Related studies are summarized in Section 5. Section 6 concludes this paper.

2. SYSTEM MODEL

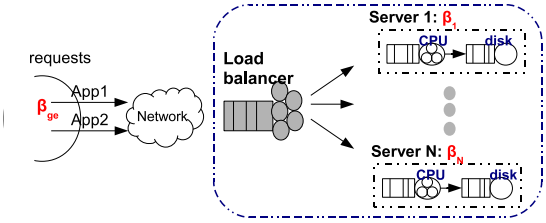


Figure 1: System model.

We consider a service hosting system consisting of K heterogenous servers and one load balancer, as well as *two* types of applications. In particular, we are interested in a CPU-intensive and an IO-intensive application. Motivated by the accuracy of queueing models capturing the performance of complicated systems and workloads [2], we model a server as a two-station queueing model shown in Fig. 1. The first station represents the aggregate computational capacity of the available cores in the server, and the second station corresponds to the disk of the server. The available computational and IO capacity is different across servers. As a result, the execution times on each server vary. The requests are processed in a first-come, first-served manner in both stations. A load balancer has T threads to concurrently dispatch requests to servers, based on the load balancing algorithm. Moreover, performance statistics (e.g., response times and the received workload mix) are collected at the load balancer. Herein, we consider the network delay of dispatching requests from the load balancer to services negligible.

Clients generate requests, which are sent to the load balancer. We assume the interarrival times of requests follow an exponential distribution with parameter λ, and the application mix generated is β_i^{ge}, denoting the percentage of application i in the mix ($i \in \{1,2\}$ and $\beta_1^{ge} + \beta_2^{ge} = 1$). Both types of request can be executed on any server; the execution of a request demands certain CPU time and disk time. Once requests complete execution of both stations, they leave the system. This is a so-called open queueing system in traditional queueing terminology.

The cumulative time an application spends on the CPU station includes the memory and cache accessing times. Yet, although the memory and caches are not explicitly modeled, their performance effect is captured via the queueing delays at the CPU station. We call the CPU and disk execution times *resource demands*, which are assumed to follow exponential distributions with mean R_{ij} for application $j = \{1,2\}$ on resource $i = \{\text{CPU}, \text{disk}\}$. Note that resource demands do not reflect any waiting nor queueing times that are due to other requests.

Resource demands vary on different servers, because of the server heterogeneity. Consequently, to capture the variability of CPU and disk demands on different servers, we introduce the scaling factors, $\gamma_{i,k}$, corresponding to each resource on all servers $k = \{1 \ldots K\}$. In a homogeneous system, the scaling factors equal to one for all server resources, meaning the average resource demands are the same on all servers. Essentially, the average resource demands of application j on server k are $R_{i,j} \cdot \gamma_{i,k}$. For each application i, each server receives an application mix, β_i, which is the percentage of requests that are for application i. The application mix received by each server depends on the load-balancing policy employed, and the mix can be very different from the original workload mix generated, β_i^{ge}. As there are two application considered here, $\beta_1 + \beta_2 = 1$.

In summary, the abstract queueing system we study is multi-server, multi-resource, and multi-application, encountering fluctuating workloads with varying volume of requests and application mix.

3. LOAD BALANCING SCHEME

In this section, we first introduce some theoretical background regarding optimization of application mixes for single server systems and load-balancing schemes for multiple servers. Motivated by the advantages of balancing server loads and resource loads on a single server, we propose new mix-aware load-balancing algorithm.

3.1 Background

3.1.1 *Optimal Application Mix in a Single Server*

Rosti et. al [14] showed that there is an optimal application mix, which can minimize response time while maximizing throughput. They focus on a single server system with multiple resources, where a constant number of requests circulates, i.e., they consider a so-called closed system. They provide a closed form formula on the application mix, which is derived from the equal utilization point for all system resources. Intuitively, to minimize response times in a system with multiple applications that compete for multiple resources, the utilization levels of the various resources should be equal or close to equal. As a result, none of the resource becomes a bottleneck where queues build up.

Applying their methodology to our system, i.e., two-station queueing model catering two applications, the optimal received mix of application j on server k, $\beta_{j,k}^*$, can be computed as

$$\begin{aligned} \beta_{1,k}^* &= \frac{\log \frac{R_{d,2}\gamma_{d,k}}{R_{c,2}\gamma_{c,k}}}{\log \frac{R_{c,1}\gamma_{c,k}R_{d,2}\gamma_{d,k}}{R_{c,2}\gamma_{c,k}R_{d,1}\gamma_{d,k}}}, \\ \beta_{2,k}^* &= 1 - \beta_{1,k}^*. \end{aligned} \tag{1}$$

However, the aforementioned optimal mix is derived from the closed queueing system, assuming a fixed number of requests. Consequently, the optimality of the mix provided in Eq. 1 may no longer be valid when encountering a varying number of requests, such as ours. For example, a system has two servers, whose current

application mixes are both at their optimal values, but the queue lengths of outstanding requests are very different. Clearly, joining the server with shorter queue leads to a shorter response time for any arriving requests. Consequently, to minimize response times on our open queueing systems consisting of multiple servers, one needs to reach not only the optimal mix of Eq. 1 for each server, but also minimal queue lengths of outstanding requests for all servers.

3.1.2 Load Balancing on Multiple Servers

There is a large body of studies on how to balance loads on multiple servers, especially for web systems. The Apache web server provides three default policies, namely bybusyness, byrequest, and bytraffic. The byrequest policy is very similar to round-robin policy. The bybusyness policy is almost identical to JSQ, except when handling the situation that multiple servers have the same number of outstanding requests. Indeed, JSQ is a simple yet powerful policy to balancing server loads, proven by theoretical analysis [9] and wide usage, especially for workloads of a single type. The bytraffic policy tries to balance number of bytes transmitted by each server.

3.2 Mix-aware Algorithm

We now develop a mix-aware policy, leveraging the advantages of the optimal application mix on a single server and the JSQ policy on multiple servers. We first illustrate the algorithm and then provide a bounding analysis, which provides us guidance to tune the algorithm parameters.

3.2.1 Overview

The proposed mix-aware algorithm tries to balance the queue length of outstanding requests on servers by JSQ and resource loads on a server by achieving the optimal application mix. When servers have very different queue lengths, it is imperative to balance the outstanding requests on servers. On the contrary, it is more essential to reach the optimal mix and balance resource loads for individual servers when queue lengths on servers are similar. The proposed algorithm chooses a server based on either of the criteria, using a *goodness threshold*. The goodness is a metric to quantify the variability of queueing lengths of outstanding requests on all servers.

In addition to the goodness value, our algorithm is composed of two load-balancing schemes, i.e., JSQ and Join Minimum Mix Slack (JMMS). When the goodness is greater than a certain threshold, JSQ is used to choose a server; otherwise, JMMS is applied. For an arriving request, JMMS checks the current application mixes on all servers and chooses a server, whose mix can be best improved by the request, i.e., the resulting mix has the minimum slack to the optimal value in Eq. 1. The required inputs of the algorithm are the class of the current request, current queue lengths on all servers, and current application mixes on all servers. The load balancer computes these inputs by keeping counters of outstanding requests for both applications on each server. When a request arrives or leaves the cluster, the load balancer retrieves its application type from the request header and increments or decrements the corresponding counter. Moreover, the load balancer is assumed to know the optimal application mix listed in Eq. 1 on all servers. The output of the algorithm is the server to which the arriving request shall be dispatched. We assume the resource demands and scaling factors required in Eq. 1 can be obtained via statistical profiling methods, and the development of such profilers is out of the scope of this paper. We summarize the algorithm pseudocode in Alg. 1.

In the following subsections, we first explain how to reach the optimal application mix using JMMS and then detail the derivation and the analysis of the threshold value of goodness.

Algorithm 1 Mix-aware load-balancing algorithm.

```
//j is the class of the arriving request
function MIX-AWARE(j)
    Q ← {Q_i, ..., Q_k}
    S_j ← {(β_{j,1} − β*_{j,1}), ..., (β_{j,K} − β*_{j,K})}
    res ← k s.t. Q_i = min(Q), k ∈ {1...K}
    goodness ← 1 − Q_res / Σ_{k=1}^{K} Q_k × K
    if goodness < G then
        res ← k s.t. (β_{j,k} − β*_{j,k}) = min{S_j}, k ∈ {1...K}
    end if
    return res
end function
```

3.2.2 Join Minimum Mix Slack (JMMS)

When dispatching a particular request, the aim of JMMS is to search for a server, whose optimal application mix can be reached with the best effort. When a requests for application j arrives, the load balancer checks the difference between the current application mix, $\beta_{j,k}$, including this request, and the optimal mix on server k, $\beta^*_{j,k}$, for all servers. We call this value the slackness,

$$S_{j,k} = \beta_{j,k} - \beta^*_{j,k}.$$

JMMS chooses the server with the minimum slack. When the minimum slack is negative, it implies that there is not a sufficient number of requests from application j on server k. The resource loads on server k can be better balanced by processing an extra application j request. On the other hand, when all servers have positive slacks, it implies that all servers have more application j requests than the optimal values, at the time instant the new request arrives. In such a case, choosing a server with minimum slack can minimize the deviation from the optimal values.

3.2.3 Goodness Value and Threshold

The queue length of outstanding requests is an indicator of the server load. The higher the variability of queue length, the higher the performance improvement that can be achieved with JSQ. To capture the best opportunity for applying JSQ, we propose a metric, termed *goodness*, defined as one minus the minimum queue length divided by the average queue length at the time instant when a new request arrives,

$$goodness = 1 - \frac{Q^{min}}{\sum_{k=1}^{K} Q_k} \times K, \tag{2}$$

where Q_k denotes the queue length of server k and Q^{min} is the minimum queue length across all servers, i.e., $Q^{min} = \min\{Q_k, \forall k\}$. The intuition behind is that when comparing Q^{min} to the average queue length, one can obtain not only a coarse estimate of the variability but also the potential performance improvement using JSQ. When all servers have the same queue lengths, Q^{min} is equal to the average value and thus the *goodness* value is 0. In contrast, when Q^{min} is significantly lower than the average queue length, the *goodness* is approaching to one and thus using JSQ can improve the overall system performance. A higher *goodness* value indicates greater queue length variability as well as the advantage of applying JSQ. In summary, the mix-aware algorithm dispatches requests using JSQ when the *goodness* value is high and using JMMS when the *goodness* value is low.

However, it is not clear how the *goodness* exactly reflects the queue length variability so that one can choose a good threshold value to decide between the employment of JSQ or JMMS, i.e.,

to balance outstanding requests or resource loads. Therefore, we derive the following bounding analysis, in which the upper bound of the queueing length variability is shown as a function of *goodness*. Based on the analysis, we then further explain our choice of *goodness* threshold, $\bar{G}$.

We use the variance of queue length, $Var[Q]$, to present the variability of queue length across servers.

COROLLARY 1. *The upper bound of $Var[Q]$ is a function of goodness:*

$$Var[Q] \leq \frac{(K-(1-goodness)(K-1))^2-1}{K^2}\left(\sum_{k=1}^{K} Q_k\right)^2 \quad (3)$$

PROOF. Thus, based on the definition of variance, one can write

$$Var[Q] = \frac{\sum_{k=1}^{K} Q_k^2}{K} - \left(\frac{\sum_{k=1}^{K} Q_k}{K}\right)^2. \quad (4)$$

Moreover, following simple algebraic manipulation, one can write the upper bound of queue length of server k,

$$Q_k \leq \sum_{h=1}^{K} Q_h - (K-1)Q^{min}, k \in \{1 \ldots K\}, \quad (5)$$

As Q_{min} is bounded by the average queue length, one knows $Q_{min} \in \{0 \ldots \frac{\sum_{k=1}^{K} Q_k}{K}\}$. We further relax the discrete property of Q^{min} and express Q_{min} as a continuous variable using $\alpha \in [0,1]$,

$$Q^{min} = \alpha \times \frac{\sum_{k=1}^{K} Q_k}{K}, \alpha \in [0,1]. \quad (6)$$

Substituting Eq. 6 in Eq. 5, we obtain Eq. 7

$$Q_k \leq \left(\frac{K-\alpha(K-1)}{K}\right)\sum_{k=1}^{K} Q_k \quad (7)$$

Substituting Eq. 7 in Eq. 4 we obtain Eq. 8

$$Var[Q] \leq \frac{(K-\alpha(K-1))^2-1}{K^2}\left(\sum_{k=1}^{K} Q_k\right)^2 \quad (8)$$

Combining Eq. 6 and Eq. 2, we straightforwardly obtain

$$goodness = 1-\alpha, \alpha \in [0,1] \quad (9)$$

Substituting Eq. 9 in Eq. 8, we thus can express the upper bound of $Var[Q]$ as a function of *goodness* □

To quantitatively gauge the relationship between the queue length variability and the *goodness*, we plot the upper bound of $Var[Q]$ in Figure 2, assuming a scenario of three servers, i.e., $K = 3$. Clearly, the upper bound of queue length variability increases in *goodness* values. On one hand, we can observe that when *goodness* values are small, e.g. less than 0.1, such upper bounds are constantly low independent of the values of $(\sum_{k=1}^{K} Q_k)^2$, as shown by the flat area on the left corner of Figure 2. On the other hand, when the *goodness* value is greater than 0.2, those upper bounds vary a lot due to the multiplication of $(\sum_{k=1}^{K} Q_k)^2$ in Eq. 3. We conjecture that a small goodness value ensures the similarity of queue lengths across server. Our extensive evaluation of the numerical results of Eq. 3 leads us to set the goodness threshold, $\bar{G}$, very low, i.e.between 0.05 to 0.1, so as to capture the scenarios where the queue lengths of all servers are very similar and balancing resource loads on individual servers by JMMS is more imperative.

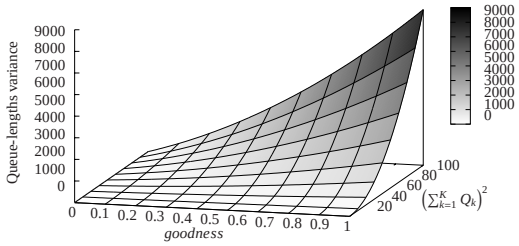


Figure 2: The upper bound of queue length variability of a three server case

4. EVALUATION

In this section, we evaluate the proposed mix-aware algorithm for service systems, using event driven simulation and a prototype in the cloud. The performance metrics evaluated are the response times, and difference between CPU and disk utilization of servers. We benchmark the mix-aware load balancing algorithm against purely JSQ and bybusyness, under different system sizes and workload mixes.

4.1 Simulation

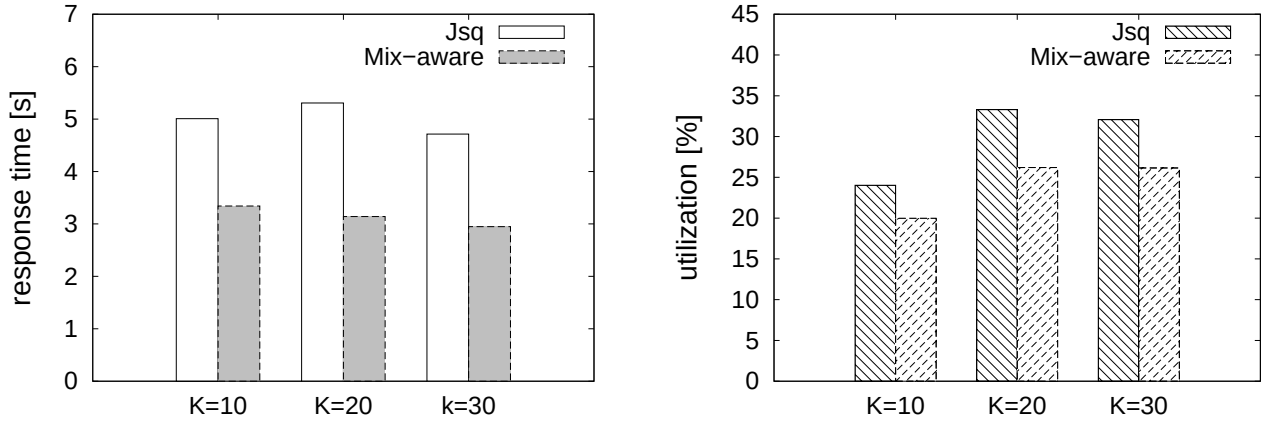
To evaluate the scalability of the mix-aware algorithm on different system sizes, we built an event driven simulator, as shown in Figure 1.

We simulate the systems with 10, 20 and 30 replicated servers. The request workload is generated using the following parameters: The inter-arrival rates are exponentially distributed with $\lambda = 9, 18$ and 27 per seond, for the scenarios with 10, 20 and 30 replicas, respectively. The generated workload mix $\beta_{ge} = 0.55$, meaning 55% of the requests are from application one, and the remaining requeststs are from application two. The CPU and disk demands for both applications are also exponentially distributed with averages being $R_{11} = 0.75s$, $R_{12} = 0.64s$, $R_{21} = 0.48s$, and $R_{22} = 1.25s$. To determine the heterogeneity of servers, the scaling factors, $\gamma_{i,k}$, are randomly chosen from the range of $[0.8, 1.2]$.

Our mix-aware algorithm computes the optimal application $\beta^*_{j,k}$ in Eq. 1, which ranges from $[0.37, 0.89]$. Note that we assume that the resource demands are known, via certain profiling methodology, which is out of the scope for this paper. The threshold value of goodness, $\bar{G}$, is set to 0.1.

We summarize the evaluation results in Figure 3. Compared to JSQ, our mix-aware algorithm can achieve a significantly lower (roughly 35%) average response time. Moreover, the average response times of mix-aware decreases with an increasing number of servers, whereas JSQ seems to not scale well with the cluster size, as shown by the increase in the response times from the scenario of 10 servers to 20 servers. JMMS is expected to be more effective when there is a larger number of servers, because of a higher chance to attain the optimal mix.

Such a performance improvement can be explained by the fact that JMMS can reach a better resource load distribution. To validate our reasoning, we compute the difference of average CPU and disk utilization per server and plot the maximum values among all servers in Figure 3(b). These values indicate the worst case of unbalance resource loads, which often result in server bottlenecks, higher queue length, and thus higher response times. Clearly, the proposed mix-aware algorithm can achieve a more balanced resource load than JSQ. Consequently, from our simulation results, we conclude that the proposed mix-aware algorithm can scale well with the cluster size, and even achieve a better performance gain.



(a) Response times

(b) Maximum difference of cpu and disk utilization

Figure 3: Simulation results: comparison of JSQ and mix-aware algorithm on systems with different number of servers.

4.2 Prototype

We built a prototype system, consisting of three servers and one load balancer, in the IBM Research Cloud. Each component is based on a virtual instance, equipped with 4GB of RAM, 4 x86_64 CPUs, 55GB of disk, and Fedora 14 the operating system distribution. Requests are generated from the client servers, located at the IBM Zurich Research Lab, using the httperf load generation tool [12], and forwarded to the load-balancer, which is based on the Apache web server version 2.4.2. Application one is `fop` in the Dacapo suite [10], a CPU intensive benchmark producing PDF documents from XSL-FO files, and application two is `luindex`, a disk intensive benchmark using lucene to index a set of documents. The average inter-arrival rate is $\lambda = 2$ per second. We consider two workload scenarios: (1) the generated workload mix is *stationary* and kept at 0.5, and the total number of requests is 2000; (2) the workload mix is *non-stationary*, i.e., changes over the time as shown in Figure 4(a).

The challenge of applying the proposed mix-aware algorithm on a real system lies in deciding the optimal application mix for each server. However, the premise of using Eq. 1 is the knowledge of the resource demands, which requires sophisticated profiling pointed out by methodology [2]. As developing profiling methodology is out of the scope of this study, we resort to offline and exhaustive search to identify the optimal application mix. Thus, we obtain the optimal mix of 0.4, meaning 40% of `fop` and 60% of `luindex` for all servers. The threshold value of goodness is set to 0.1.

4.2.1 Stationary Request Workload

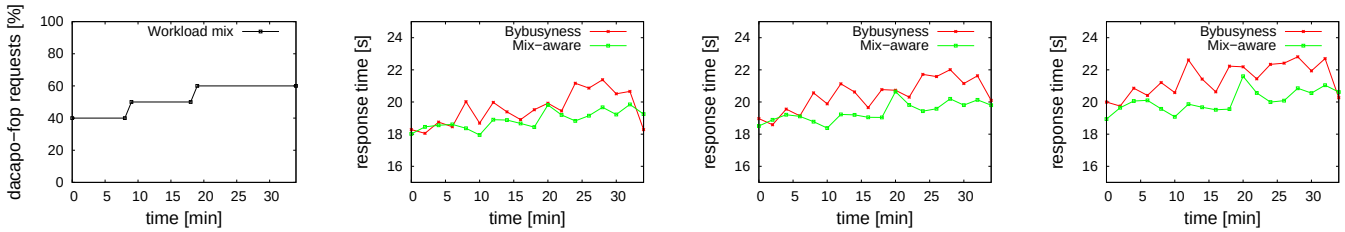
In Table 1, we summarize the average, 90^{th}, 95^{th}, 99^{th} percentile and maximum response times for both bybusyness and mix-aware algorithm. To accommodate high performance variability in the computing cloud [5], we present the results of two iterations. Indeed, one can see that the statistics of high percentile for a given policy have higher differences between two iterations. Moreover, one can observe that average response times of mix-aware are lower than bybusyness by roughly 10% and above, for all statistics presented. The performance improvement of mix-aware is particularly visible for higher percentile of response times. For example, in iteration one, the maximum response time of mix-aware is lower than bybusyness by roughly 24%, whereas the performance gain of $90^{th}-$percentile of mix-aware algorithm is only 10%. Therefore, we believe that mix-aware algorithm is able to achieve lower response times, especially for higher percentile, meaning that the worse performance is better mitigated.

4.2.2 Non-stationary Request Workload

In this subsection, we evaluate our proposed algorithm on non-stationary workload mix, as shown in Figure 4(a). Motivated by the mitigation of worse performance in the stationary workload, here we focus on presenting higher percentile of response times computed from 2 minutes window in Figure 4 (b), (c), and (d). One can observe that mix-aware is able to achieve lower response times, especially for the 99^{th} percentile. Another observation worth of mentioning is that the performance gain of mix-aware algorithm, compared to bybusyness, is lower in our cloud prototype than in the simulation. This can be attributed to applying inaccurate values of optimal application mix on virtual servers, which are known to be highly variable [5], i.e., the resource demands may change over the time. In our future work, we will address the profiling methodology, that is able to computing the optimal mix on the fly. Nonetheless, given the inherent inaccuracy of optimal values applied in mix-aware algorithm in the cloud environment, we are still able to mitigate the worse performance, compared to simple bybusyness algorithm.

5. RELATED WORK

There is a large body of related studies of load balancing for various conventional service systems [6, 7, 13, 16] and modern cloud systems [8]. As a detailed survey of the extensive related work is not possible here, we only outline some particularly relevant work. Cardellini et. al [6] qualitatively classified existing load balancing schemes at web server systems into four approaches, namely server-based, client-based, DNS-based, and dispatcher-based. They quantitatively compared the maximum utilization of clusters, via simulation. Cherkasova and Ponnekanti [7] developed FLEX, a locality-aware load balancing solution, especially for efficient memory usage. Zhang et. al [13, 16] proposed load balancing schemes, LARD and ADAPTLOAD, for rapidly fluctuating workloads that are characterized in terms of arrival rates and document popularity. Via simulation, they showed that ADAPTLOAD can achieve a low slowdown and resource utilization on a cluster of homogenous web servers. Björkqvist et. al [3] used a lottery bal-



(a) workload mix, β_{ge} (b) 90^{th} percentile (c) 95^{th} percentile (d) 99^{th} percentile

Figure 4: Non-stationary workload mix: comparison of bybusyness and mix-aware algorithm on 90^{th}, 95^{th} and 99^{th} response times over the time.

Table 1: Response times statistics of stationary workload mix: comparison of bybusyness and mix-aware load balancing.

Statitstics	mean[s]		90^{th}[s]		95^{th}[s]		99^{th}[s]		Max[s]	
Algorithm	bybusy.	m-w	bybusy.	m-w	bybusy.	m-w	bybusy.	m-w	bybusy.	m-w
iter 1	16.40	15.01	19.70	17.30	20.60	17.90	21.80	18.60	24.10	19.54
iter 2	16.46	15.08	19.10	17.30	19.90	17.90	21.10	18.80	22.93	19.96

ancing algorithm for distributed stateful service systems. Dejun et. al [8] focused on balancing requests among a set of heterogenous machine instances in the cloud, based on the profile of response times and request rates for each server.

Singh et. al [15] leveraged the idea of application mixes and proposed a mix-aware resource allocation for data centers. They used a k-means clustering algorithm to automatically determine the workload mix and allocate servers. However, they did not explore the workload mix in their load balancing scheme.

Most of the aforementioned studies focus on balancing loads on homogenous servers with a single resource type, i.e., CPU. In contrast, our mix-aware algorithm considers multiple resources on a set of heterogenous servers, and aims at balancing the overall server loads as well as their resource load.

6. CONCLUSION

In this paper, we propose a mix-aware load-balancing algorithm for web service systems, which are hosted on a set of heterogenous servers and cater requests from a dynamically varying application mix. Our algorithm aims at minimizing the response times by dispatching requests in a way that optimizes the application mix for each server. Depending on a goodness value, a metric used to evaluate the variability of outstanding requests, our mix-aware algorithm either balances the outstanding requests using JSQ, or balances the resource loads using JMMS. Our results obtained with simulation and with a cloud prototype show that our proposed algorithm can scale with an increasing system size, and attenuate the worst response times compared to the bybusyness policy, one of the most robust default load-balancing algorithms in the Apache web server.

Regarding ongoing research, we are exploring efficient on-line profiling methodologies to obtain the optimal application mix, as well as extending our algorithm to a larger number of applications.

7. REFERENCES

[1] Amazon EC2. http://www.amazon.com/.

[2] D. Ansaloni, L. Y. Chen, E. Smirni, and W. Binder. Model-driven Consolidation of Java Workloads on Multicores. In *Proceedings of IEEE/IFIP DSN*, 2012.

[3] M. Björkqvist, L. Y. Chen, and W. Binder. Load-balancing dynamic service binding in composition execution engines. In *Proceedings of APSCC*, pages 67–74, 2010.

[4] M. Björkqvist, L. Y. Chen, and W. Binder. Dynamic replication in service-oriented systems. In *IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCgrid)*, 2012.

[5] M. Björkqvist, L. Y. Chen, and W. Binder. Opportunistic service provisioning in the cloud. In *Proceedings of IEEE Cloud*, 2012.

[6] V. Cardellini, M. Colajanni, and P. S. Yu. Dynamic load balancing on web-server systems. *IEEE Internet Computing*, 3(3):28–39, 1999.

[7] L. Cherkasova and S. Ponnekanti. Optimizing a 'content-aware' load balancing strategy for shared web hosting service. In *Proceedings of MASCOTS*, pages 492–, 2000.

[8] J. Dejun, G. Pierre, and C.-H. Chi. Resource provisioning of web applications in heterogeneous clouds. In *Proceedings of the 2nd USENIX conference on Web application development*, pages 5–5, 2011.

[9] V. Gupta, K. Sigman, M. Harchol-Balter, and W. Whitt. Insensitivity for ps server farms with jsq routing. *SIGMETRICS Performance Evaluation Review*, 35(2):24–26, 2007.

[10] http://dacapobench.org/. `Dacapo suite`.

[11] http://httpd.apache.org/. `Apache`.

[12] http://www.hpl.hp.com/research/linux/httperf/. `httperf`.

[13] A. Riska, W. Sun, E. Smirni, and G. Ciardo. Adaptload: Effective balancing in custered web servers under transient load conditions. In *Proceedings of ICDCS*, pages 104–111, 2002.

[14] E. Rosti, F. Schiavoni, and G. Serazzi. Queueing Network Models with Two Classes of Customers. In *Proceedings MASCOTS*, pages 229–234, 1997.

[15] R. Singh, U. Sharma, E. Cecchet, and P. Shenoy. Autonomic mix-aware provisioning for non-stationary data center workloads. In *Proceedings of ICAC*, pages 21–30, 2010.

[16] Q. Zhang, A. Riska, W. Sun, E. Smirni, and G. Ciardo. Workload-aware load balancing for clustered web servers. *IEEE Trans. Parallel Distrib. Syst.*, 16(3):219–233, 2005.