On Load Balancing: A Mix-Aware Algorithm for Heterogeneous Systems

Sebastiano Spicuglia
University of Lugano
Lugano, Switzerland
sebastiano.spicuglia@usi.ch

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
Politecnico di 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 servers that are often located within computing clouds, whose computational and storage resources can be highly heterogeneous. The workload served typically exhibits disparate computation patterns (e.g., CPU-intensive or IO-intensive), that fluctuate both in terms of volume and mix. The system heterogeneity together with workload diversity further exacerbates the challenge of effective distribution of load within a computing cloud. This paper presents a novel, mix-aware load-balancing algorithm, which aims to distribute requests sent by multiple applications in heterogeneous servers such that the application response times are minimized and system resources (e.g., CPU and IO) are equally utilized. To this end, the presented algorithm tries to not only balance the total number of requests seen by each server, but also to shape the requests received by each server into a certain "mix", that is analytically shown to be optimal for response time minimization. Our experimental results—based both on simulation and on a prototype implementation—show that the mix-aware 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.

Categories and Subject Descriptors

H.3.4 [**Systems and Software**]: Performance evaluation—*Algorithms; Experimentation; Measurement*

Keywords

Load balancing algorithm; Heterogeneous system; Cloud computing; Simulation; Prototype

ICPE'13, March 21–24, 2013, Prague, Czech Republic.

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 heterogeneous components. Web service applications are characterized not only by disparate resource requirements (e.g., a CPU-intensive browsing mix versus an I/O-intensive transaction mix), but also by time-varying request workloads [4, 17]. Consequently, the overall system workloads fluctuate in terms of mixes of dissimilar applications and their volume of requests [16]. The heterogeneity of servers, together with the workload heterogeneity, further exacerbate the challenges of load balancing. Load balancing within a compute cloud is critical for performance and effectiveness of the compute cloud paradigm, see for example the elastic load balancer in Amazon EC2 [1].

There is a large body of load balancing studies [3, 6, 8, 17] that focus on mainly homogeneous 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 to be theoretically robust [10] and is also used in practice [12] for distributing the entire load across distributed servers. In a heterogeneous system experiencing a time-varying workload mix, such a policy can potentially lead to the situation where servers receive similar amounts of requests but servers with powerful CPU (resp. IO) process a lot of IO- (resp. CPU-) intensive requests. Such unanticipated behavior can be detrimental for the overall system performance as well as for the individual application response times. It is therefore imperative for the load-balancing scheduler to evolve such that it becomes aware of both of the heterogeneity of the workload (e.g., is it IO- or CPU-intensive) but also on the heterogeneity of the architecture of the back-end servers.

In this paper, we aim to provide a robust load balancing solution in a heterogeneous compute cloud. We first focus on the following questions: Is there an optimal application mix for each server which can lead to a low response time such that all resource types 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 to be very effective in balancing the overall loads on homogeneous servers, can one leverage JSQ when designing a mix-aware load dispatching policy to further balance resource loads on heterogeneous servers?

To this end, we develop a novel mix-aware load-balancing algorithm, which aims to balance the server loads as well as balance bottleneck across the various server resources such that the

global response time averaged over all applications is minimized. We explore the JSQ policy and the analytical results in [15], which illustrate via a closed queueing network model that there is an optimal application mix for each server. Our algorithm distributes requests based on two criteria: the number of outstanding requests per server and the desire to balance the load across each resource of each server, by eliminating the existence of a single bottleneck, e.g., CPU or IO. A *balance* metric, that measures the variability of queue lengths for outstanding requests across servers, is used to decide whether it is more beneficial to balance the number of outstanding requests per server using JSQ or to slightly violate the JSQ conditions by aiming to eliminate single resource bottlenecks in each server by directing requests such that each server serves an "optimal" load. The threshold value of the *balance* metric that triggers the above activity is based on a bound analysis of the potential queue lengths at the various servers. We use an event-driven simulator to evaluate the scalability of the proposed algorithm and a prototype implementation to support the applicability of the proposed algorithm on a real system.

The 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 heterogeneous systems with multiple resources. Being aware of the queue length of the servers and of the application mix, our algorithm is able to achieve consistent performance improvements when compared to the standard load-balancing policy available on Apache web server, i.e., the `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 presents experimental results. Related studies are summarized in Section 5. Section 6 concludes this paper.

2. SYSTEM MODEL

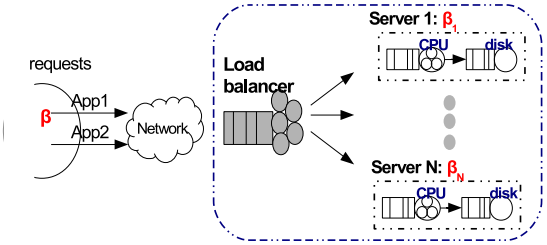


Figure 1: System model

We consider a service hosting system consisting of K heterogeneous servers and one load balancer, as well as *two* types of applications, that we classify as *CPU-intensive* and *IO-intensive*. These two classes of applications are referred with the index $j = \{1,2\}$. We model each server as a two-station queueing model shown in Fig. 1, see also [2,7] for the effectiveness of such queuing network models in capturing the performance of non-trivial application co-location. The first station marked "CPU" represents the aggregate computational capacity of the available cores in the server, and the second station marked "IO" corresponds to the disk of the server. These two stations are referred with the index $i = \{CPU, IO\}$. The available computational and IO capacity is different across servers. As a result, the execution times of the same application on each server may vary. The threads at the load balancer are concurrently dispatched to the servers, based on a load balancing scheme. Performance statistics (e.g., response times and the characteristics of the workload mix) are collected at the load balancer. We consider that the network delay of dispatching requests from the load balancer to servers to be negligible.

Clients generate requests, which are sent to the load balancer. We assume that the interarrival times of requests follow an exponential distribution with parameter λ. The percentage of the two application classes in the arriving flow of requests is $\{\beta_1, \beta_2\}$ with $\beta_1 + \beta_2 = 1$. Both types of requests can be executed on any server; the execution of a request needs certain CPU time and IO time. Once requests complete execution in both stations, they leave the system.

The cumulative time an application spends on the CPU station includes any memory and cache accessing times. Yet, as shown in [2,7], although the memory and caches are not explicitly modeled, if the CPU execution times without any queueing are correctly measured, then the performance effects of memory and cache are captured via the queueing delays at the CPU station. The cumulative time spent on the CPU or on the IO resources by an application in the absence of any queueing is referred to as *resource demand*. CPU and IO demands are assumed to be exponentially distributed with mean R_{ij}, with the subscript i referring to the resource and the subscript j referring to the application. Note that the resource demands do not reflect any waiting nor queueing times due to resource contention with other applications.

Resource demands vary on different servers, because of the server heterogeneity. Consequently, to capture the variability of CPU and IO demands on different servers, we introduce the scaling factors $\gamma_{i,k}$, corresponding to each resource i on server $k = \{1 \ldots K\}$. In a homogeneous system, the scaling factors equal to one for all servers, implying that the average resource demands of an application are the same across all servers. Based on this convention, the average resource demands of application j on server k are $R_{ij} \cdot \gamma_{i,k}$. Because services are replicated, i.e., each server may be able to execute *any* application, we assume that it serves a percentage of requests β_j of application j. The application mix received by each server depends on the load-balancing policy, and this mix may be very different from the one that is observed by the load balancer (or alternatively, the one generated by the clients).

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 a new, mix-aware, load-balancing algorithm.

3.1 Background

3.1.1 Optimal Application Mix in a Single Server

Rosti et. al [15] showed that there is an optimal application mix, which can minimize response time while maximizing throughput. The authors of [15] focus on a single server closed system with multiple resources and provide a closed form formula on the application mix, which is derived from the equal utilization point for all system resources.

Intuitively, a system operates at the best performance when the global utilization of the resources is maximized. According to their methodology, in a system with two stations and two classes of applications, the optimal mix of application class j on server k, is:

$$\beta^*_{1,k} = \frac{\log \frac{R_{IO,2}\gamma_{IO,k}}{R_{CPU,2}\gamma_{CPU,k}}}{\log \frac{R_{CPU,1}\gamma_{CPU,k}\,R_{IO,2}\gamma_{IO,k}}{R_{CPU,2}\gamma_{CPU,k}\,R_{IO,1}\gamma_{IO,k}}}, \quad (1)$$

$$\beta^*_{2,k} = 1-\beta^*_{1,k}.$$

This optimal mix refers to a single server k. Here the problem we are addressing is significantly more complex because to minimize response times in a system consisting of multiple heterogeneous servers, one needs to reach not only the optimal mix of Eq. 1 for each server, but also minimal queue lengths of outstanding requests across all servers. Clearly, joining the server with the shortest queue of outstanding requests is not a sufficient condition to optimize performance.

3.1.2 Load Balancing on Multiple Servers

There is a large body of literature 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 the round-robin policy. The `bybusyness` policy is almost identical to JSQ, except when handling the situation where multiple servers have the same number of outstanding requests. Indeed, JSQ is a simple yet powerful policy for balancing homogeneous server loads, its optimality has been shown theoretically [10]. 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 some bound analysis of the servers which provides us guidance to tune the algorithm parameters and we then present the mix-aware load balancing algorithm.

3.2.1 Overview

The proposed mix-aware algorithm tries to balance the queue length of outstanding requests on servers using JSQ and the resource loads on each server by achieving the optimal application mix that equalizes the utilization IO and CPU resources. When servers have very different queue lengths, it is imperative to balance the outstanding requests across servers. When queues of servers are balanced, then the algorithm slightly "unbalances" the queues by aiming to reach the optimal mix as shown by Eq. (1) on each server and achieve equiutilization across its resources. Straddling between the above two competing targets, the mix-aware algorithm aims to achieve overall better performance.

We definite a metric called *balance* that quantifies the degree of (un)balancing across the K servers. We start by defining this metric and discuss its usefulness.

3.2.2 Balance 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 identify the best opportunity for applying JSQ, we propose a metric, called *balance*, defined as follows:

$$balance = 1 - \frac{Q^{min}}{\sum_{k=1}^{K} Q_k} \times K. \quad (2)$$

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 reasoning behind this is that when comparing Q^{min} to the average queue length, one can obtain not only a rough estimate of the variability but also of 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 *balance* value is 0. In contrast, when Q^{min} is significantly lower than the average queue length, *balance* is approaching one and thus using JSQ can improve the overall system performance. A higher *balance* value indicates greater variability across queues as well as the advantage of applying JSQ. The mix-aware algorithm dispatches requests primarily using JSQ but when the *balance* value is low, as a second step it attempts to unbalance the queues at the servers by aiming to equally utilized the CPU and IO resources at the servers. This unbalancing action is bringing higher performance gains on a per-server basis, which is also positively reflected in the overall system performance.

Since *balance* is a metric between zero and one, it is not clear what is its value that should trigger using JSQ or instead the unbalancing the queue lengths while aiming equiutilization of individual server resources. To calculate this threshold $\bar{B}$, we use the upper bound of the queue length variability across servers. Let $Var[Q]$ present the variance of queue lengths.

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

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

PROOF. Based on the definition of variance, one can write

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

Following simple algebraic manipulations, one can write the upper bound of the queue length of server k as

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

As Q^{min} is bounded by the average queue length, then $Q^{min} \in \{0\ldots\frac{\sum_{k=1}^{K} Q_k}{K}\}$. 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

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

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

To quantitatively gauge the relationship between the queue length variability and *balance*, 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 the queue length variability increases as *balance* increases. We can observe that when *balance* values are

small, e.g., less than 0.1, such upper bounds are constantly low, independently 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 *balance* is greater than 0.2, the upper bound varies a lot due to the multiplication of $(\sum_{k=1}^{K} Q_k)^2$ in Eq. 3. We conjecture that a small *balance* value ensures the similarity of queue lengths across servers. Our extensive evaluation of the numerical results of Eq. 3 leads us to set the *balance threshold*, $\bar{B}$, very low, i.e., between 0.05 to 0.1, so as to capture the cases where the queue lengths of all servers are very similar.

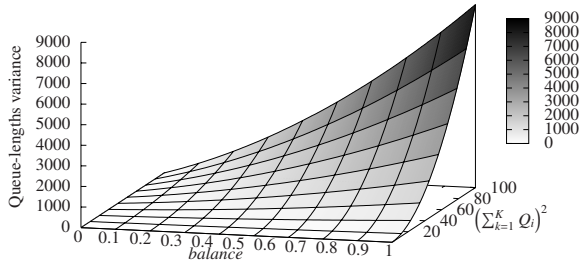


Figure 2: The upper bound of queue length variability of a three server case

3.2.3 Unbalance by Equiutilizing

When dispatching a particular request, the aim is to search for a server, whose optimal application mix (i.e., server equiutilization, as expressed by Eq. 1) can be reached as quickly as possible. When a request from application j arrives, we check 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:

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

We chose the server with the minimum $S_{j,k}$ value. When this value is negative, it implies that there is not a sufficient number of requests from application j on server k. The resource utilizations (loads) on server k can be better balanced by processing an extra request from application j. When all servers have positive S values, it implies that all servers have more application j requests than their optimal values, at the time instant the new request arrives. In such a case, choosing a server with the minimum S value can minimize the deviation from the optimal one.

3.2.4 Combining JSQ and Unbalance by Equiutilization

Combining together JSQ and Unbalance by Equiutilizing, we obtain the Mix-Aware Load Balancing Policy. As a first step, the algorithm calculates the *balance* metric and compares it to the $\bar{B}$ threshold. If the value is larger than $\bar{B}$, then the incoming request from application class j is scheduled using JSQ. Otherwise, the $S_{j,k}$ values are computed for all k servers and the server than can best improve its optimal value in Eq. 1 is selected. The required inputs of the algorithm are the class of the current request, the current queue lengths on all servers, and the current application mixes on all servers. The load balancer computes these inputs by keeping counters of outstanding requests for both application classes 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. The load balancer is assumed to know the optimal application mix listed in Eq. 1 for all servers. The output of the algorithm is the server to which the arriving request should be dispatched. We assume that the resource demands and scaling factors required in Eq. 1 can be obtained via statistical profiling methods, although the development of such profilers is out of the scope of this paper. We summarize all this in Alg. 1.

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})}
    server ← k such that Q_k = min(Q), k ∈ {1...K}      //JSQ
    balance ← 1 − Q_res / (Σ_{k=1}^{K} Q_k) × K
    if balance < B̄ then    //Unbalance by Equiutilizing
        server ← k such that (β_{j,k} − β*_{j,k}) = min{S_{j,k}}, ∀k
    end if
    return server      //send request to the selected server
end function
```

4. EVALUATION

In this section, we evaluate the proposed mix-aware algorithm for service systems, using event driven simulation and a prototype. 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.

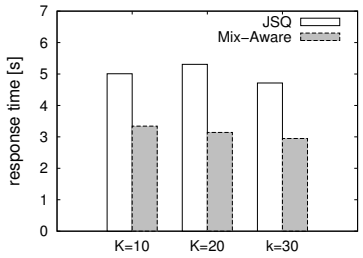
4.1 Simulation Results

To evaluate the scalability of the mix-aware algorithm on different system sizes, we built an event driven simulator of the model shown in Figure 1 and also a discrete event simulator for JSQ. We simulate the systems with 10, 20 and 30 replicated servers. The workload is generated using the following traffic intensities: inter-arrival rates are exponentially distributed with $\lambda = 9$, 18 and 27 per second, for the scenarios with 10, 20 and 30 replicas, respectively. We assumed a two-class workload, with 55% of the requests are from class one, and the remaining requestst are from class two. The CPU and disk demands for both applications are also exponentially distributed with averages being $R_{CPU,1} = 0.75s$, $R_{CPU,2} = 0.64s$, $R_{IO,1} = 0.48s$, and $R_{IO,2} = 1.25s$. To enforce server heterogeneity, the scaling factors $\gamma_{i,k}$, are uniformly distributed in $[0.8, 1.2]$.

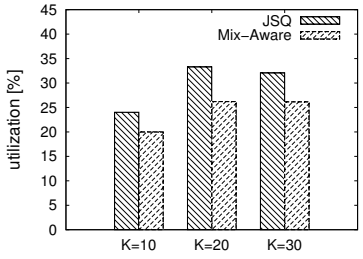
Our mix-aware algorithm computes the optimal $\beta^*_{j,k}$ for application class j in server k, see Eq. 1. With the above inputs, this value is in $[0.37, 0.89]$. Note that we assume that the resource demands are known and are obtained via a profiling methodology, see [**?**, **?**]. The threshold value of *balance*, $\bar{B}$, is set to 0.1.

We summarize the simulation results in Figure 3. Compared to JSQ, the mix-aware algorithm can achieve a significantly lower (roughly 35%) average response time. As expected, 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. Focusing on equiutilizing the loads on the server resources is expected to be more effective when there is a larger number of servers, because of a higher chance to attain the optimal mix on individual servers. To validate this reasoning, we compute the difference of average CPU and disk utilization per server and plot the maximum values among all servers, see Figure 3(b). The plotted values show the worst case of unbalance resource utilizations, which results in resource bottlenecks, overall

higher queue lengths, and thus higher response times. Clearly, the mix-aware algorithm can achieve more “balanced "resource utilizations than JSQ.



(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 numbers 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 Laboratory, using the httperf load generation tool [13], and forwarded to the load-balancer, which is based on the Apache web server version 2.4.2. Naturally, we present here experiments with `bybusyness`, the standard Apache load balancing policy that resembles JSQ and our mix-aware policy.

We consider a two-class workload. Class one consists of the `fop` benchmark in the Dacapo suite [11], a CPU intensive benchmark producing PDF documents from XSL-FO files. Class two consists of `luindex`, a disk intensive benchmark that indexes a set of documents. The average inter-arrival rate of arriving requests (`fop` or `luindex`) is $\lambda = 2$ applications per second. We consider two workload scenarios: (1) the two-class mix is *stationary* and kept at 0.5, and the total number of requests (i.e., applications sent for execution) is 2000; (2) the workload mix is *non-stationary*, i.e., changes across 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. Within a server, the optimal mix is 0.4, i.e., 40% of `fop` and 60% of `luindex`, see [2] for the methodology to compute ths mix. As in the simulation experiments, the threshold value of *balance*, $\bar{B}$, is set to 0.1.

4.2.1 Stationary 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 algorithms. To accommodate high performance variability in the computing cloud [5], we present the results of two separate experimental runs. One can see that the statistics of the higher percentiles for a given policy have higher differences from one experimental run to the other. Moreover, one can observe that the average response times of mix-aware are lower than `bybusyness` by roughly 10%, for all presented statistics. The performance improvement of mix-aware is particularly visible for the higher percentiles of response times. For example, in run one, the maximum response time of mix-aware is lower than `bybusyness` by roughly 24%, whereas the performance gain of the $90^{th}-$percentile of the mix-aware algorithm is only 10%. This experiment illustrates that the mix-aware algorithm is able to achieve lower response times, especially for the higher percentiles, meaning that the worse performance is better mitigated.

4.2.2 Non-stationary Workload

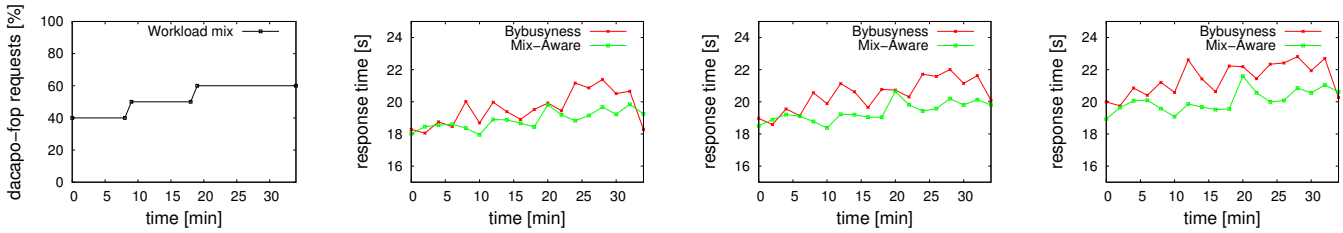
In this subsection, we evaluate our proposed algorithm on a non-stationary workload mix, as shown in Figure 4(a). Here we focus on presenting the higher percentile of response times computed from 2 minute windows, see Figure 4 (b), (c), and (d). One can observe that mix-aware is able to achieve lower response times, especially for the 99^{th} percentile. Another observation worth mentioning is that the performance gain of the mix-aware algorithm compared to `bybusyness`, is lower in our cloud prototype than in 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., resource demands may change across time. In our future work, we will address the profiling methodology that is able to compute the optimal mix on the fly. Nonetheless, given the inherent inaccuracy of optimal values applied in the mix-aware algorithm in the cloud environment, we are still able to mitigate the worse performance, compared to the `bybusyness` algorithm.

5. RELATED WORK

There is a large body of related studies of load balancing for various conventional service systems [6, 8, 14, 17] and modern cloud systems [9]. 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 for 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 [8] developed FLEX, a locality-aware load balancing solution, especially for efficient memory usage. Zhang et. al [14, 17] proposed 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 homogeneous web servers. Björkqvist et. al [3] used a lottery balancing algorithm for distributed stateful service systems. Dejun et. al [9] focused on balancing requests among a set of heterogeneous machine instances in the cloud, based on the profile of response times and request rates for each server.

Singh et. al [16] 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 homogeneous servers with a single resource type, i.e., CPU. In contrast, our mix-aware algorithm considers multiple resources on a set of heterogeneous servers, and aims at balancing the overall server loads as well as their resource load.



(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.

Statistics	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
run 1	16.40	15.01	19.70	17.30	20.60	17.90	21.80	18.60	24.10	19.54
run 2	16.46	15.08	19.10	17.30	19.90	17.90	21.10	18.80	22.93	19.96

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

6. CONCLUSION

In this paper, we propose a mix-aware load-balancing algorithm for web services that are hosted on heterogeneous servers and cater for 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. We evaluate the how well the variability of outstanding requests across servers and based on this metric we balance the outstanding requests using JSQ or unbalance the server queues by aiming to equally utilize CPU and disk resources on the servers. Our results obtained with simulation and with a cloud prototype show that our proposed algorithm can scale with an increasing system size, and improve on 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. ACKNOWLEDGMENTS

The research presented in this paper has been supported by the Swiss National Science Foundation (project 200021_141002) and by the European Commission (Seventh Framework Programme grant 287746). The research presented in this paper was conducted while the first author was with IBM Research Zurich Lab. Evgenia Smirni is partially supported by NSF grants CCF-0937925 and CCF-1218758.

8. 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*, pages 1–12, 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)*, pages 531–538, 2012.

[5] M. Björkqvist, L. Y. Chen, and W. Binder. Opportunistic service provisioning in the cloud. In *Proceedings of IEEE Cloud*, pages 237–244, 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. Y. Chen, D. Ansaloni, E. Smirni, A. Yokokawa, and W. Binder. Achieving Application-Centric Performance Targets via Consolidation on Multicores: Myth or Reality? In *Proceedings of HPDC*, pages 37–48, 2012.

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

[9] 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.

[10] 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.

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

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

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

[14] 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.

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

[16] 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.

[17] 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.