This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TPDS.2016.2639009, IEEE Transactions on Parallel and Distributed Systems

Stochastic Resource Provisioning for Containerized Multi-Tier Web Services in Clouds

Omer Adam*, Young Choon Lee**, and Albert Y. Zomaya, *Fellow, IEEE*§

*§School of Information Technologies, J12, University of Sydney, NSW 2006, Australia.
**Department of Computing, Faculty of Science and Engineering, Macquarie University, NSW 2109, Australia.

Abstract—Under today's bursty web traffic, the fine-grained per-container control promises more efficient resource provisioning for web services and better resource utilization in cloud datacenters. In this paper, we present *Two*-stage *S*tochastic *P*rogramming *R*esource *A*llocator (2SPRA). It optimizes resource provisioning for containerized n-tier web services in accordance with fluctuations of incoming workload to accommodate predefined SLOs on response latency. In particular, 2SPRA is capable of minimizing resource over-provisioning by addressing dynamics of web traffic as workload uncertainty in a native stochastic optimization model. Using special-purpose OpenOpt optimization framework, we fully implement 2SPRA in Python and evaluate it against three other existing allocation schemes, in a Docker-based CoreOS Linux VMs on Amazon EC2. We generate workloads based on four real-world web traces of various traffic variations: AOL, WorldCup98, ClarkNet, and NASA. Our experimental results demonstrate that 2SPRA achieves the minimum resource over-provisioning outperforming other schemes. In particular, 2SPRA allocates only 6.16% more than application's actual demand on average and at most 7.75% in the worst case. It achieves 3× further reduction in total resources provisioned compared to other schemes delivering overall cost-savings of 53.6% on average and up to 66.8%. Furthermore, 2SPRA demonstrates consistency in its provisioning decisions and robust responsiveness against workload fluctuations.

Index Terms—Cloud Resources Provisioning, Workload Uncertainty, Stochastic Optimization, Multi-tier Web Applications, Containers.

1 Introduction

Dynamically adjusting resources provisioned for web services in clouds in a timely response to cope up with load changes require developing awareness of dynamics in workloads as well as resource deployment platforms with a short lead time. In particular, uncertainty introduced in application's workloads due to the stochastic nature of web traffic has made it a challenge to accurately provision the exact amount of resources for a web application [1]. As a result, applications have to be over-provisioned typically for peak loads, or jeopardize their performance service-level objectives (SLOs) with penalties, otherwise. Either of which is costly to adopt by cloud providers and users.

For example, failure to efficiently scaling cloud resources provisioned for a web application under uncertainty in time-varying workloads is a key reason to pose expensive delays in responses and risk overall user experience and the quality of service (QoS) [1] [2]. On Mar 11 2015, major Apple online stores suffered 12 hours global outage two days after an event to promote the new Apple-Watch product was held [3]. Web services outage or even short delays may incur substantial loss in revenue. For instance, an additional 100ms of response delay costs Amazon.com 1% of sales. Likewise, Google loses 20% of its traffic upon an additional 500ms in response latency [4].

As for cloud providers, over-provisioning clearly poses a chronic problem in datacenters that has led to inefficient resource utilization and higher operating cost for resource wastage. [1] [5]. A typical datacenter runs at only 8-15% efficiency on average [6], whilst application owners are still being charged for unused deployed resources. Another plausible reason for such underutilization could be attributed to the coarse granularity of resources acquisition with virtual machines (VMs) [7]. In addition, VMs are not tailored enough to effectively respond to the uncertain demands of web services in particular [5] [8]. However, with the advent of container-based model, a finer-grained resource slicing is feasible, specifically for web services with varying warkloads, so as only needed resources can be deployed and hence actively utilized. Besides, this model exhibits a low startup overhead, offering a short lead time to deploy containers to readily serve workload changing as it occurs [9] [10]. As such, the model becomes highly attractive particularly for web services. For example, Google declares that everything from Web Search to Gmail services are packaged and run as Linux containers in its cloud [11].

In this paper, we address the problem of resource over-provisioning with the consideration of uncertainty in web workloads in a native stochastic provisioning model. To this end, we design Two-stage Stochastic Programming Resource Allocator (2SPRA) that minimizes resource over-provisioning for n-tier web applications in clouds. 2SPRA captures the complex relationship between workload fluctuations, resources provisioning, and response latency in a two-stage stochastic programming optimization model. This model computes the optimal number of application containers in response to the fluctuations of incoming workload to fulfill SLOs on response latency with minimal over-provisioning. Specifically, in the first stage an initial decision is made before the realization of traffic workload is revealed/known. The system is then subject to a stochastic process affecting the outcome of the first-stage decision. A

Email addresses: omoh7417@uni.sydney.edu.au (Omer Adam), young.lee@mq.edu.au (Young Choon Lee), albert.zomaya@sydney.edu.au (Albert Y. Zomaya).

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TPDS.2016.2639009, IEEE Transactions on Parallel and Distributed Systems

recourse decision can then be made in the second stage to rectify any bad effects resulted from the first-stage decision.

Specific contributions in this paper are as follows:

- We model time-varying fluctuations in web traffic as a stochastic process that realistically captures uncertainty in web workloads.
- We devise a realistic decision-making mechanism by implementing it using a powerful stochastic programming model that drives decisions on resources to be allocated in two stages. The optimal policy in such model is to finalize allocation decisions that will perform well *on average* against uncertainty in workloads.
- We fully implement 2SPRA in Python using the Python-written OpenOpt optimization framework [12] [13] supported by a number of global nonlinear problem (GLP) solvers.
- We thoroughly evaluate 2SPRA against three existing resource allocation schemes, on Docker-based CoreOS VMs on Amazon EC2: (**1**) CPU Threshold-based Predictive [14], (**2**) Cost-Latency tradeoff Optimization [15], and (**3**) PEAK-based [16].
- We further verify the results using workloads from 4 real-world web traces of different load intensities, namely: AOL, WorldCup98, ClarkNet, and NASA.

Our experimental results show that 2SPRA outperforms others in terms of: amount of allocated resources, resource over-provisioning relative to application demands, and allocation costs. In particular, since it allocates resources closely match what application actually needs, 2SPRA achieves minimal over-provisioning of 6.16% on average (and 7.75% on maximum) more than actual demand observed. It achieves 3× further reduction in total resources provisioned while SLO-compliant latency is consistently fulfilled. The overall savings obtained in allocation costs are 53.6% on average and up to 66.8%. Moreover, 2SPRA demonstrates robustness in its provisioning decisions against workload variability.

The rest of the paper is organized as follows: Section 2 describes our system infrastructure, applications targeted, operational architecture, and workload traffic modulation. Section 3 defines performance modeling of workload and container utilization. Section 4 models application containers tier and then proposes 2SPRA. Section 5 presents our experimental results, performance analysis, and comparisons with existing schemes. Section 6 discusses related works, and Section 7 concludes the paper with future work.

2 System Model

We address multi-tier web applications containerized in clouds. This section models the physical layout of the underlying infrastructure, the applications targeted, the operational architecture, and workload modulation.

2.1 Infrastructural Layout of the System

This section lays out our system architecture along with the targeted applications.

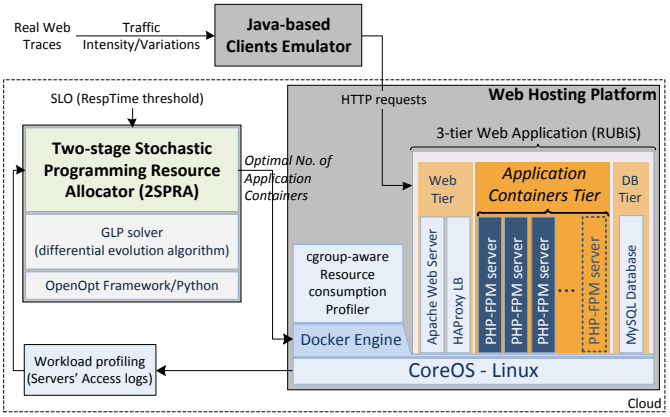

Fig. 1: The overview of our testbed.

2.1.1 Abstract n-tier Web Applications

Majority of services in clouds are classified into one of three broader categories, multi-tier web services (i.e. long-running services) represent the most common workloads in clouds nowadays [17]. As a representative benchmark for web services, we use the PHP-version of RUBiS v1.4.2 – an online auction system modeled after eBay. It is designed to evaluate application servers performance scalability [18]. It implements the core functionalities of an auction site: selling, browsing and bidding, with a distinction of three kinds of user sessions: visitor, buyer, and seller. As shown in Figure 1, RUBiS system is modeled as three-tier web application and hence comprised of: a frontend web servers tier, a middleware tier, and a backend databases (DB) tier. The middleware runs a series of PHP FastCGI Process Managers (PHP-FPM) [19]. The PHP-FPM is robust, fast, and 'independent' application server, enabling the capability to maintain a resizable pool of *PHP Application servers Tier* for processing php web requests to render dynamic pages/web resources. This tier is typically where major computations related to request processing are carried out.

2.1.2 Testbed Architecture

We build a web hosting testbed using container-enabled system as a representative use-case of reality (Figure 1) where each application server runs as a container. The time elapsed in the application containers tier typically represents large portion of the total delay in the processing lifecycle of each HTTP dynamic request [2] [20] [7]. To capture application containers tier's behavior, we model it using G/G/m queuing system for which there is no exact analysis (i.e. there is no exact formula for queuing performance measures such as queuing waiting time) [21]. However, approximate analysis of an exact model can be used for the system as it operates in an efficiency-driven status [21] [22] [23]. In efficiency-driven system, the majority of requests essentially are delayed in a waiting queue prior to processing when application servers are highly utilized under heavy traffic [22].

Our testbed setup consists of three components: (**1**) the clients emulator, (**2**) the web hosting platform where RUBiS benchmark is deployed, and (**3**) the resource allocator (2SPRA). More specifically, the first component runs a Java-based Clients Emulator (Oracle Java v1.7.0_80) to create varying number of concurrent users sessions each of which

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TPDS.2016.2639009, IEEE Transactions on Parallel and Distributed Systems

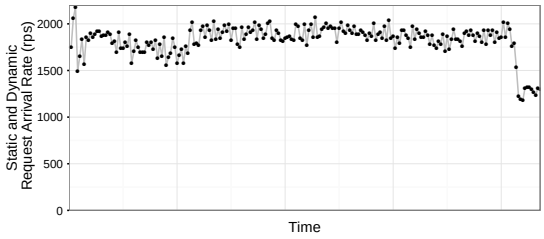

Fig. 2: A steady load observed for the default RUBiS clients emulator.

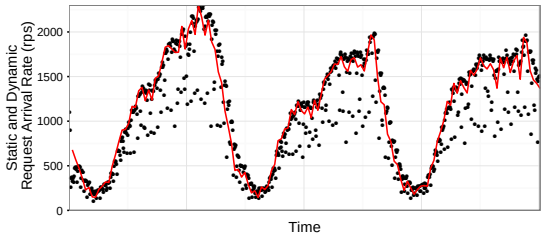

Fig. 3: Traffic variations in workload under influence of AOL real web traces.

generates real traces-based workload HTTP requests against the hosted application.

The second component hosts the targeted 3-tier web application (RUBiS) in VM with CoreOS (stable r717.3.0) Linux operating system. CoreOS [24] is a Docker-centric minimal Linux operation system primarily engineered with Linux kernel (v4.0.5) and Docker engine (v1.6.2) [25] running as a daemon to readily manage 'Docker' Linux containers. CoreOS uses `systemd` as its init system and configuration platform. Since CoreOS only allows programs to run as containers, we deploy the entire RUBiS benchmark's tiers [26] as containers[1]. All Docker container images are built from Ubuntu 14.04 default public image. As such, an Apache HTTP web server (v2.4.7) container is deployed as a frontend tier. While directly responding to requests for static pages, it is configured to forward php web requests (for dynamic pages) to a HAProxy (v1.4.24) container serving as a load balancer (LB) with a round-robin distribution fashion. A dynamic-sized pool of PHP-FPM application-servers containers (or simply, *application containers*) are seated behind the load balancer ready to process requests upon their arrival. Each application container runs PHP v5.5.9-1 with Zend Engine v2.5.0. The LB container maintains a dynamic-list of IP addresses of application containers available. Via Docker daemon, this tier of application containers is under control of 2SPRA whenever optimization is carried out in response to the variations in workload. A MySQL database (v5.6.1) container is also setup with relevant data. Our testbed can be easily extended to a multi-VMs setup via exploiting container management tools like Kubernetes [27] or Docker Swarm [28]. Such tools abstract away the underlying VMs provisioned enabling application tier to expand its containers across multiple VMs in the cloud.

The third component runs a Python-based piece of code that implements 2SPRA in a VM with AMD64 Ubuntu-trusty OS 14.04 (Linux kernel 3.13.0-48-generic). Subject to SLO requirement, it retrieves workload profiles and accordingly optimizes number of application containers in the application containers tier. 2SPRA requires this VM to install a scientific Python software stack which is comprised of (bottom up): Python v2.7.1 (r271:86832), scientific numerical computing Python Libs (numpy-v1.9.2), and more importantly, Python-written OpenOpt v0.5501 Optimization Framework [12] [13]. The OpenOpt framework introduces a Python module named 'FuncDesigner' for stochastic programming problems, along with a number of global nonlinear problem solvers (GLP). They can be used to formulate, express, and solve systems of nonlinear equations that involve stochastic variables of various distributions representing uncertainties in optimization problems. More specifically, we use a nonlinear GLP solver named 'de' that implements a Differential Evolution Algorithm capable of handling nonlinear objective functions and constraints.

Using real web traces, the clients emulator generates time-varying workload of HTTP requests against the hosted RUBiS application. Arrival requests for static pages are served by the web tier while those for dynamic pages are forwarded to the application containers tier for processing. Requests processed are logged into workload profiling datastore which is designated to be used for aggregation by the 2SPRA to make allocation decisions. Given SLO threshold for the response time, 2SPRA retrieves workload profiles and accordingly optimizes number of application containers needed in the application containers tier.

All VMs in our testbed are 64-bit machines and equipped with an Intel Xeon Processor E5-2670 v2 speeding at 2.5GHz, with 7.5GB RAM, and 30GB SSD. All VMs are situated in us-east-1c zone on Amazon EC2 [29].

2.2 Modulating Traffic Workload

RUBiS comes with its own default workload generator that emulates users behavior using 26 different transaction types including browse items, buy, sell, bid, among others. The clients emulator initiates a number of concurrent Java threads representing clients' sessions in each of which a sequence of HTTP requests is generated. The type of the next request is determined by a state transition matrix that specifies probability to go from one request type to another, with think-time (in ms) to wait for between requests. Each session lasts for a fixed duration of 15 min [18]. Having said that, this default workload generator apparently lacks important features we were seeking. More specifically, session length is fixed and the number of concurrent clients is fixed too, and there is no sense of realistic traffic variations/burstiness usually observed in real web services. As result, we unsurprisingly witnessed steady workloads generated against our testbed (Figure 2).

To introduce long-scale traffic intensity, we modify the source code of original RUBiS clients emulator to accept as input files traffic-variation distributions derived from the AOL real webserver traces [30] [31]. We used per-30-minutes workload intensity observed over 72 hours in the traces to modulate workload intensity[2] of the modified emulator.

1. As of this writing, an access to our built containers are publicly available for download from our account: '*omera*', on Docker Hub Registry [26].

2. It is similar in essence to modulation approaches for RUBiS emulator [32] [33] [34], and for SPECweb2009 Client [14].

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TPDS.2016.2639009, IEEE Transactions on Parallel and Distributed Systems

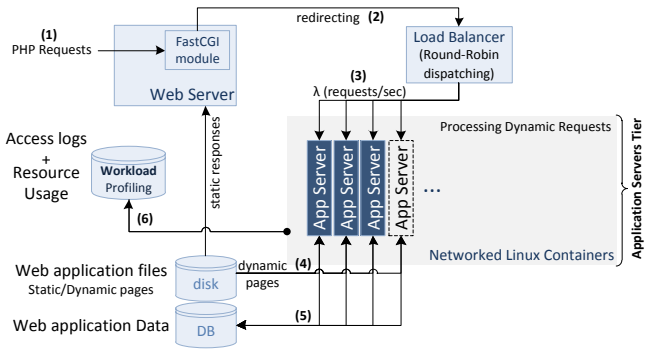

Fig. 4: A logical view of processing lifecycle of incoming web requests.

Each trace-based traffic variation file consists of fraction values ($\in [0, 1]$) representing portions of the maximum possible number of 1200 simultaneous clients (e.g. 0.5 indicates that the current number of clients should be adjusted to 600 clients). These fraction values are generated by normalizing all load-variation values (in the AOL traces) with respect to the maximum load-variation found. The key idea for our modified client emulator is to dynamically change number of simultaneous clients as per those fraction values. To allow the entire trace to be replayed, the client emulator divides the runtime of the experiment into equal intervals based on the number of fractions in the trace. The emulator iterates over each fraction value to compute the corresponding number of simultaneous clients to adjust to; by adding more new Java threads or removing existing ones. The emulator sleeps after each iteration for fixed duration to allow the variation to take place. Each client session created (a Java thread) may last for 15 min (i.e. the default) or may be *terminated* at anytime as more load-variation fraction values are being applied (i.e. causing sessions to vary in length). Similar to the default emulator, our modified emulator uses the same state transition matrix to determine next request types, as well as the same think-time to wait for between requests [18].

As a result, throughout any experiment, we would have continuously varying number of concurrent clients, indeterministic session lengths, and more importantly, realistic workload fluctuations resembling the input real web traces. Figure 2 shows the default steady load, while Figure 3 demonstrates the workload observed from the modified RUBiS clients emulator where traffic variations (red solid line) imposed by AOL traces. Notice the two figures depict arrival rates in requests/second (rps) of all requests regardless of being for static or dynamic pages. Dynamic page requests represent around 63% of total number of requests.

2.3 Operational Architecture

In Figure 4, we demonstrate the processing lifecycle of requests arriving the web hosting platform. Upon arrival of php requests, the web server redirects them through its enabled FastCGI module to a load balancer (steps 1-2). A load balancer with round-robin dispatching policy is sufficient in the sense that application containers are of similar capacity. As such, the load balancer evenly distributes the incoming workload (measured as request arrival rate λ) across the application tier (step 3). Each application container processes assigned requests along with any involved necessary accesses to both php scripts files and databases to ultimately compose a response (steps 4-5). In addition, each container is enabled to log its processed workloads (i.e. each request is logged as soon as it is completed) along with its own resources usage (every second) into a workload profiling datastore which is designated to be used for aggregation and analysis by 2SPRA to make allocation decisions (step 6). It is worth mentioning that the web server is enabled to tag each incoming request with a unique id with which each request can be tracked until its completion. 2SPRA relies on fairly-accurate measured processing capacities of the containers. As a result, it requires all completed requests to be aggregated in each second to measure the processing rates (i.e. request service rate μ) of application containers in requests per second.

3 Performance Modeling

Web services are of long running nature as they are typically designed to run indefinitely. Hence, they persistently consume cloud resources in the long term representing important workloads in clouds to deal with [17]. In multi-tier web services, web resources (e.g. webpages) have to be rendered and composed dynamically upon request. *Application containers Tier* is where such heavy duty job takes place. As such, controlling resources and performance of this tier is the key to comply with the QoS proclaimed [2] [35].

3.1 Web Services Performance Metric

The latency, as the time from when a client issues a transaction (a request) until the response is received, indeed involves two time components; namely a) network latency, and b) processing delay (time elapsed within the web hosting platform where requests are processed). Although network latency represents an important factor, it has been studied extensively in previous works [36] [37] from different perspectives including network switches design, packets routing, and transport protocols/medium in datacenters [4]. Hence, it goes beyond the scope of our study. In addition, with unprecedented ever-growing number of web-services providers that are moving to cloud-based datacenters, controlling delays incurred in network is no way under their control. That is because web application providers typically have no option on co-location decisions which make them to share the network with other co-located applications that could be of totally different nature (e.g. parallel data processing, distributed graph-processing) and indeed have different requirements for network resources consumption (which is not the case with the legacy web-hosting datacenters where only web applications share network resources). As a result, they ultimately resort to optimize their web hosting platform upon which they have full control in attempt to tame processing delays to maintain QoS. This paper focuses on processing delay of requests, and it may be perceived as the time elapsed from when a request arrives at the web hosting platform until a response is queued [38].

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TPDS.2016.2639009, IEEE Transactions on Parallel and Distributed Systems

3.2 Service-Level Objective

For web services, SLOs typically set out performance objectives to maintain certain QoS. In our experiments where we record timing in microseconds precision (and is converted to milliseconds as needed), we noticed that processing delay of requests for static html pages is way far lower than that of dynamic php requests. That is obviously due to the special processing php requests have to go through in the application containers tier. Moreover, the time a php request spends in the application containers tier represents large portion of the total delay in its processing lifecycle, which also includes access delays to database tier. It is also a common practice to over-provision database tiers in real-world deployments [39]. Because databases typically make a unique and challenging set of demands on resources allocated. Due to these observations, we center our work and modeling on the application containers tier with the assumption that it remains the bottleneck tier throughout our experiments. We provision sufficient resources to DB tier to prevent it from being bottlenecked [2] [7]. As such, we define the response time for a request to be the time from when a request is forwarded to the application containers tier until a response departs the tier. The performance objective is to maintain the average response time below a predefined SLO threshold.

In this work, we consider average response time rather than tail latency (e.g. 99th percentile of response time) because the G/G/m queuing model, upon which 2SPRA is designed, uses the mean response time by default. Nevertheless, our work can be easily extended to consider tail latency by fitting a latency distribution whose 99th percentile fulfills the SLO threshold, and then use the mean of that distribution [16].

3.3 Workload and Container Utilization Modeling

Although the time-varying relationship between resource utilization and the effective workload is dynamic and nonlinear, it can be approximately characterized around the current operating point in every time interval [32]. Given a set of identical application containers $\mathbb{S}$, the amount of workload λ_i imposed on a container i ($i \in \mathbb{S}$) to generate average CPU utilization u_i over a time interval t can be approximated in the following form [14]:

$$\lambda_{i,t} = \Lambda \cdot u_{i,t} \tag{1}$$

The quantity Λ is defined as the ratio of the peak workload to the corresponding peak CPU utilization measured for the only one container i (in a singular set $\mathbb{S}' = \{i\}$) when it is saturated. As such, we can rewrite equation 1 as:

$$\lambda_{i,t} = \frac{\lambda^{peak}}{u^{peak}} \cdot u_{i,t} \tag{2}$$

To quantify Λ, we need to map workload to container utilization. So, inspired by approaches [2] [38], we run an experiment where we provision only one application container in the application containers tier in our testbed. We flood it with an overwhelming workload and measure its CPU consumption. During this saturation, the measured workload (in rps) and CPU utilization (in percentage of CPU time) are considered peak values. CPU utilization is computed as the difference between two readings of the cumulative CPU time for a container (taken at two points in time displaced by a fixed time interval) divided by that interval length.

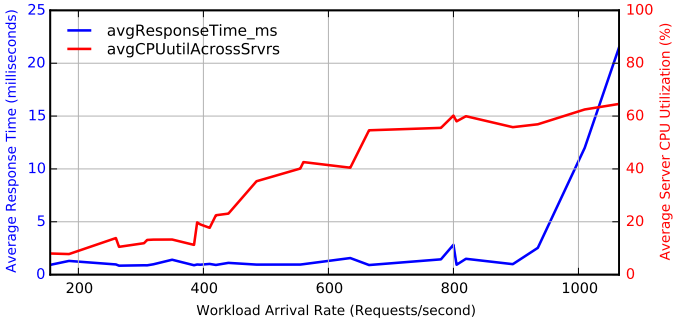

Fig. 5: Average response time (ms) and containers utilization versus workload (rps) measured for dynamic PHP web requests.

3.4 Container Provisioning and Resource Utilization

Related works [1] [38] identified that resource utilization is strongly correlated to the QoS as higher workload intensity would lead to QoS violations due to queuing delay. Workload transaction types also have direct impact on utilization levels at which affected resources are occupied [20] [14]. We emphasize that workload is a key factor on determining the required capacities of the underlying resources that are translated to the needed processing rates and service times. This correlation is also observed in Figure 5 of an experiment where we provision our hosting testbed (Section 2.1.2) with 9 PHP application containers in the application containers tier.

As shown in the figure, the average response time is in nearly steady range with a mean around ∼1.2ms until the workload arrival rate approaches 1000 requests per second (rps), after which the response time dramatically increases by a factor of ∼10, and it further spikes reaching beyond ∼21ms occasionally. Similar observation is reported for containers' CPU utilization. A steady expected growth in CPU consumption is observed as workload increases, reaching beyond ∼64% when the workload exceeds 1000 rps. Consequently, monitoring incoming workloads and accordingly adjusting the underlying resource capacities to accommodate certain level of QoS is essential to maintain healthier overall system performance with improved web user experience. The model (Section 4) upon which 2SPRA is designed stands on these facts.

4 Two-stage Stochastic Programming Resource Allocator

In this section, we first model application containers tier (Figure 1), define its resource allocation problem as Two-stage stochastic programming optimization model, and finally solve the model (i.e., 2SPRA).

4.1 Modeling Application Containers Tier

Web requests arriving a hosting platform represents workload volumes inherently fluctuate in a random fashion over time. The volume of requests is usually represented in terms

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TPDS.2016.2639009, IEEE Transactions on Parallel and Distributed Systems

of their arrival rates. In order to present a statistical setting for describing its characteristics, we represent web requests data as a time series [40]. We assume a time series can be defined as a collection of random variables indexed according to the order they are obtained in time [41]. Let X_t be a random variable representing the total number of web requests that arrive in time fragment t of a fixed length l. As such, a sequence $\{\lambda_t : t \in [1, T]\}$ represents arrival rates (w.r.t. l) of requests arriving at a web application for a time epoch of length $T \cdot l$. We would refer to such set as a stochastic process, and to any of its observed value λ_t at time fragment t as a realization of the stochastic process.

In the application containers tier, a number of containers are provisioned to process arriving requests for which response latency is incurred. To capture tier's behavior, we need to model the relationship between web requests volumes applied, resource capacities provisioned, and resultant response times. To this end, we employ G/G/m queuing system with m representing number of statistically identical and independent containers to provision. The G/G/m queue defines a queue formed by arrival requests awaiting to be processed with a number of servers m (i.e. application containers) where interarrival times and service times have general (i.e. arbitrary) distributions (hence, the 'G' in the queue name). All servers are attending to this single queue from which requests are dispatched for processing on first-come first-served (FIFO) service discipline [23] [42].

Requests arrive at the system with arrival rate λ (estimated as expected value $\mathbb{E}[\mathrm{X}_t]$) and get processed by a container at service rate μ. This G/G/m system is of good fit to our setup as it is sufficiently general to capture *arbitrary* arrival distributions and service time distributions [16]. As such, interarrival times of any consecutive arrival requests follow a generally-distributed (not necessarily exponential) random variable, and can be estimated as $1/\lambda$. Upon dispatching an arrival request, an application container processes the request incurring a service time S to elapse. The service time S is generally-distributed (not necessarily exponential) random variable as well and may be estimated as $\mathbb{E}[S] = 1/\mu$. At any point in time, the overall completion rate of requests depends on the number of application containers m allocated in the system. And since each application container processes incoming requests at rate μ, the overall processing rate for the system is then $m\mu$. The system is considered to be stable if and only if $\lambda < m\mu$, or equivalently (via Little's law):

$$\rho = \frac{R}{m} < 1, \quad \text{s.t.} \quad R = \frac{\lambda}{\mu} \tag{3}$$

where ρ is the utilization factor, and R is the offered load, and the system is said to be in steady-state [23] [42].

Since there is no exact analysis for G/G/m system model (i.e. there is no exact formula for queuing performance measures such as queuing waiting time), one thus resorts to approximate analysis of an exact model in an operational system that is efficiency-driven. In an efficiency-driven system, the majority of requests essentially are delayed in a waiting queue prior to processing when heavy traffic is imposed. It is more likely to prevail as ρ is closely approaching 1 ($\rho \to 1$) when containers are highly utilized [21] [22] [23]. Furthermore, according to Kingman [43], the waiting time W_q is exponential in heavy traffic if there is a probability that an incoming request has to wait (i.e. with probability $\Pr[Wait > 0]$). To represent this effect, the following Equation 4, formally lays out the so called Kingman's exponential Law of Congestion [23]:

$$\begin{aligned}\Gamma &= \frac{W_q}{\mathbb{E}[S]} \\ &\approx \begin{cases} \exp\!\left(mean = \frac{1}{m} \cdot \frac{1}{1-\rho} \cdot \frac{C_a^2 + C_s^2}{2}\right) & ,w.p.\ \Pr[Wait{>}0] \\ 0 & ,w.p.\ \Pr[Wait{=}0] \end{cases}\end{aligned} \tag{4}$$

This measure of performance may be perceived as a *unitless congestion index* (Γ) to approximate G/G/m system under heavy traffic as $\rho \to 1$ [22] [23]. Equation 4 decomposes the congestion into three multiplicative components: (**1**) m: number of containers with which the system is operating, (**2**) ρ: the utilization factor, and (**3**) Stochastic Variability: arises from both squared coefficient of variation C_a^2 and C_s^2 that quantify the effect of arrivals distribution and service time distribution on system performance, respectively. For example, the squared coefficient of variation of interarrival times is evaluated as: variance of interarrival times/(mean interarrival time)^2.

To approximate the probability that requests have to wait ($\Pr[Wait{>}0]$) upon arrival at G/G/m system, a reasonable approximation [23] [21] is to simply use Erlang's C formula $E_{2,m}$ (i.e. probability of delay) as given in the exact analysis of M/M/m queuing [42]:

$$\Pr[Wait > 0] = \frac{\frac{R^m}{m!} \cdot \frac{1}{1-\rho}}{\sum_{k=0}^{m-1} \frac{R^k}{k!} + \frac{R^m}{m!} \cdot \frac{1}{1-\rho}} = E_{2,m} \tag{5}$$

Given Equations 4 and 5, the Allen-Cunneen approximation [23] [42] often gives a good approximation for average congestion measures for G/G/m, such as expected waiting time $\mathbb{E}[W_q]$ [21]:

$$\begin{aligned}\mathbb{E}[W_{q,G/G/m}] &\approx \mathbb{E}[W_{q,M/M/m}] \cdot \frac{C_a^2 + C_s^2}{2} \\ &\approx \frac{1}{m} \cdot \mathbb{E}[S] \cdot \frac{E_{2,m}}{1-\rho} \cdot \frac{C_a^2 + C_s^2}{2} \\ &\approx \frac{1}{m} \cdot \mathbb{E}[S] \cdot \frac{\rho}{1-\rho} \cdot \frac{C_a^2 + C_s^2}{2}\end{aligned} \tag{6}$$

As such, the response time L is defined as a random variable describing the total time a request spends in the system from when it arrives until a response departs the system. The expected response time can be calculated as follows [23]:

$$\mathbb{E}[L] = \mathbb{E}[W_{system}] = \mathbb{E}[W_q] + \mathbb{E}[S] \tag{7}$$

That is, the response time is the sum (W_{system}) of the time the request spends waiting in queue and the processing time incurred while serving it by some application container in the tier. Approximations in Equations 4-7 improves further as traffic gets heavier and the system becomes more congested ($\rho \to 1$) [22] [23] [21].

4.2 Two-stage Stochastic Optimization Model: A Formulation for Resource Allocation Problem

Resource allocation problem. We present our resource allocation mechanism (2SPRA) as a two-stage stochastic programming optimization model, in Equation 8. We model uncertainty of workload traffic in such stochastic model

where it fits naturally to deal with the relationship that relates together: workload fluctuation, resource provisioning, and response time (as QoS measure). 2SPRA optimizes number of containers provisioned m in the application containers tier, in accordance with the fluctuations of incoming workload λ_t to accommodate a given response latency L_{thres}. Consider a problem when a decision on number of application containers to be provisioned should be made *before* a realization of uncertain workload that yet to come. To model such situation is to regard the incoming workload as a random 'stochastic' process X representing requests arrival rates. We assume that the probability distribution P of X can be estimated from prior historical data collected over time (e.g. containers' logs) and therefore is known. Suppose that the set of all realizations of X is finite and given by X = $\{\lambda_1, \lambda_2, \ldots, \lambda_T\}$ over a history window of size T, where λ_t is the arrival rate of requests in time interval t of length l (e.g. a second). Each realization of X is called a scenario. Thus, we construct the probability space[3] (X,P) to define a finite set of scenarios, each of which is associated with probability p_t where $\sum_t p_t = 1$.

Mechanism of two-stage model. The workflow of the two-stage stochastic model is as follows: a decision on number of containers m_o to provision is made in the *first stage*. After which, the system is subject to a random process described by (X,P) affecting the outcome of the first-stage decision. A recourse decision m_r can then be made in the *second stage* that compensates for any bad effects (described by functions Γ and L) that might have been experienced as a result of the first-stage decision. The ultimate objective is to come up with a final decision m of number of containers to provision which is defined as the sum of the two stages' decisions m_o and m_r. The optimal policy in such model is to finalize a decision that will perform well *on average* against the uncertainty in workload volumes represented by (X,P), and we say that the system is optimized on average.

$$
\begin{aligned}
&\textbf{Min} \quad \Gamma(m_o, \lambda_o) + \mathbb{E}_{\mathrm{X}}[Q(m_o, \mathrm{X})] \\
&1.\ \textbf{s.t.} \quad L(m_o, \lambda_o) \leq L_{thres} \\
&2. \quad \lceil \lambda_o/\mu \rceil + 1 < m_o \\
&3. \quad m_o \geq 1, \quad m_o \text{ is real} \\
&\text{where} \quad Q(m_o, \mathrm{X}) = \textbf{Min} \quad \Gamma(m, \mathrm{X}) \\
&4. \quad \textbf{s.t.} \quad m_o + m_r = m \\
&5. \quad \mathbb{E}_{\mathrm{X}}[L(m, \mathrm{X})] \leq L_{thres} \\
&6. \quad \lceil \mathbb{E}_{\mathrm{X}}[\mathrm{X}/\mu + 1] \rceil < m \\
&7. \quad m_r \in \mathbb{R} \\
&8. \quad m \geq 1 \\
&9. \quad m \text{ is integer}
\end{aligned}
\tag{8}
$$

Description of 2SPRA. In Equation 8, we present 2SPRA algorithm. It takes as inputs: λ_o, μ, (X,P), C_a^2, C_s^2, and L_{three}. It produces as output the optimal size (m) for the application containers tier. $\mathbb{E}_{\mathrm{X}}$ is the expectation of possible scenario or outcome with respect to the probability space (X,P). The objective functions constitute the congestion index function Γ (given by Equation 4). In the first stage, it accepts as input a given workload requests volume λ_o for which the first-stage decision variable of number of application containers m_o has to be made before the outcome of the stochastic variable X is revealed/known. In constraint 1 ($L(m_o, \lambda_o) \leq L_{thres}$), the average response latency function L (given by Equations 7 and 6) works to maintain the observed response times below a certain given threshold value L_{thres}. Constraint 2, however, helps to decide for a value for m_o by imposing a boundary where the system would be closer to a steady state.

Similarly, in the second stage, the function Γ accepts as input the first-stage decision *outcome* m_o along with stochastic process described by (X,P). For this stage, constraints 5 and 6 work in similar fashion to constraints 1 and 2, but this time under the influence of probabilistic conditions imposed by X. Notice that constraint 3 defines a positive range of real values for the first-stage variable m_o. Constraint 7, however, declares the second-stage variable m_r to be a real number ($\in \mathbb{R}$) accepting positive or negative values, allowing it to fix (tune) whatever outcome was decided for the first-stage variable m_o. It works in conjunction with constraint 4 such that the final decision m of number of application containers is always positive, hence, constraint 8 follows. Constraint 9 ensures that m is a whole number. Notice that both functions Γ and L have other input quantities, namely: μ, C_a^2, and C_s^2. They are dropped from Equation 8 to improve its readability.

4.3 Optimizing Applications with 2SPRA

When the system approaches a busier state as heavier traffic load emerges, optimizing resources in the application tier in such periods of time is critical to maintain system health and avoid degradation in user experience (e.g. longer latency). We need to identify these periods which would be referred to as eligibility periods for optimization. To this end, 2SPRA continuously monitors containers' logs, parses and aggregates logged requests in each time interval t of length $l = 1$ sec to measure arrival rates λ_t and processing rates μ_t in rps, over a history window of size T= 60 sec. Monitoring how much workload is being processed (μ) with respect to the incoming load (λ) can be expressed[4] by:

$$Z = \frac{\rho}{1-\rho} \tag{9}$$

Equation 9 approximates the ratio of system utilization to system availability, by virtue of Little's law [23].

From our experiments, we observed clear variability in Z as the system undergoes through heavier traffic over time. To capture this variability, we employ the 'coefficient of variation' (C_Z) of Z as a measure to detect eligibility periods:

$$C_Z = \frac{\mathrm{sdev}(Z)}{\mathrm{mean}(Z)} \tag{10}$$

The C_Z is a standardized measure of dispersion of the frequency distribution of Z. We identify a range of 5-10% for C_Z values above which the system is considered entering periods in time where traffic load is getting heavier, and

3. A probability space is typically a mathematical triplet (Ω,$\mathcal{F}$,P), where Ω is a set of all realizations of X, and $\mathcal{F}$ is a set of events where each event is a set containing zero or more realizations of X. In this work, each event contains one realization of X, therefore, the probability space can be simply expressed as (X,P).

4. Monitoring arrival rate alone is insufficient as it must be correlated to a consumption measure such as processing rate μ as in our work and others [38], or end-to-end response time [20] [44] to identify when to trigger resource-scaling.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TPDS.2016.2639009, IEEE Transactions on Parallel and Distributed Systems

thus the 2SPRA may be triggered to optimize the system. We observed dramatic increases in C_Z variations by a factor of 11 from <5% reaching ~73% on average. We use a history window of 60 sec to measure C_Z.

When 2SPRA is triggered, all required input data are aggregated. 2SPRA requires a value for the initial arrival rate (λ_o) in the first stage to decide on the first-stage variable (m_o). We set it to be the average arrival rate aggregated over the history window. Using other techniques (e.g. time-series statistical analysis) to specify this initial rate may improve the first-stage outcome which may facilitate approaching the optimal result eventually. This would make an interesting future extension of our work. Other input quantities such as μ, C_a^2, and C_s^2 are aggregated too over the history window. In addition, the probability distribution of X is computed by aggregating each arrival rate observed in the history window to estimate its associated probability. 2SPRA is implemented in Python and it uses the nonlinear GLP solver 'de' of the OpenOpt optimization framework to search for the optimal number (m) of containers for the application tier. The GLP solver 'de' is efficient to seek feasible solutions in few seconds. For 2SPRA, it took 0.22-1.44 seconds.

Once m is found, the 2SPRA issues the necessary Docker commands instructing the Docker Engine to initiate/terminate containers to adjust to the required optimal tier size, and consequently, updates the LB's list of active containers to put them in effect and to redirect their shares of upcoming workload. To support different types of containers with different capacities, 2SPRA can be adjusted to consider allocate larger container whose processing capacity is multiple(s) of that of a smaller container. For example, replacing three containers with one container whose capacity is three times larger. It is analogous to multiplicity in granularity of resources offered in clouds. Amazon EC2 typically offers resource types in multiple capacity of one another (e.g. xlarge instance is 2x in its computing capacity than a large instance, and so on). It is worth mentioning that initiating containers generally incurs startup time until its actual availability for operating. However, such time is not addressed for the scope of our study due to being sufficiently small for the applications considered. Because we observed extreme repaid launching of containers (almost instantly) on CoreOS VMs on Amazon EC2.

5 Evaluation

In this section, in comparison with three existing schemes, we evaluate 2SPRA in terms of amount of allocated resources, resource over-provisioning w.r.t. application demands, and allocation costs, using AOL traces in addition to 3 other real web traces.

5.1 Experimental Setup

Our testbed comprises containerized web hosting platform which is built with Dockers. The entire testbed is fully deployed on VMs running CoreOS operating system and hosted on Amazon EC2. The exhaustive technical information is presented in Section 2.1.2, and Figure 1 depicts its detailed physical structure. In our experiments, we initially set up our testbed with 2 PHP application containers in the application containers tier to serve requests hitting RUBiS-based 3-tier web application. We assume that the application tier remains the bottleneck tier during all the experiments. Therefore, we ensure sufficient resources are provisioned for the database tier to prevent it from being bottlenecked.

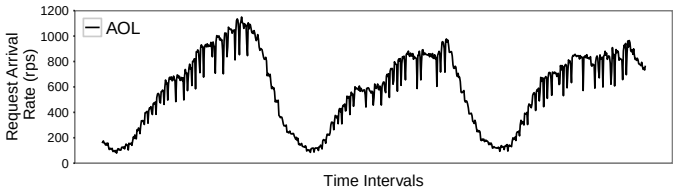

Fig. 6: Workload generated based on AOL traces.

As for the workload, RUBiS provides two workload types, namely, browse-only, and bidding-mix workloads. We use the bidding-mix workload which includes 15% read-write transactions, and is meant to be the most representative of an auction site workload [18]. However, we integrate this workload with long-scale traffic variations extracted from AOL real web traces datasets [30]. AOL traces consist of ~20M web queries collected from ~650k real users over 3 months, from Mar 1 to May 31, 2006. In our experiments, we select a time span of 3 days from Mar 2, 2006 01:00:12 to Mar 4, 2006 23:59:59 for RUBiS clients emulator, to determine the distribution of concurrent threads with a maximum of ~ 600 to generate modulated load traffic (Figure 6) using per-30 minutes workload intensity aggregated over the 72 hours time span.

5.2 Alternative Resource Provisioning Schemes

We implemented three existing schemes in Python. One at a time, a scheme is deployed in the place of 2SPRA, and is subject to the same workloads and experimental conditions for fairness.

- *PEAK-based* [16]: This scheme uses the tail of workload distribution to estimate worst-case peak demand for the next optimization period to provision resources. A histogram is generated for each period seen in a history window from which a probability distribution of arrival rates is yielded. The peak workload is then estimated as a high percentile of the arrival rate distribution.
- *Cost-Latency tradeoff* [15]: This scheme uses an optimization model whose objective function is a trade-off between cost of resources and response latency: $\arg_m \min \ \ \alpha \cdot L(m) + (1-\alpha) \cdot C(m)$, where $\alpha \in [0,1]$ is the importance ratio of cost and latency, and m is the number of containers to allocate. The latency function L is from M/M/m queuing system. The cost function C calculates container normalized cost per time unit. Since this objective function is nonlinear, we use the same GLP solver 'de' to solve it, with α = 0.8 [15].
- *CPU Threshold-based Predictive* [14]: For a preset threshold for CPU utilization (= 70%), this scheme computes the corresponding maximum workload arrival rate λ_i^{max} a single container i can handle, using workload-utilization model similar to Equations 1-2. It also defines a capacity constraint: that the total workload

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TPDS.2016.2639009, IEEE Transactions on Parallel and Distributed Systems

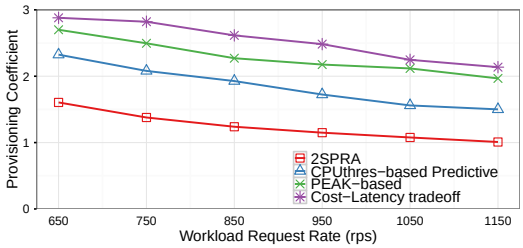

Fig. 7: Provisioning Coefficient of different allocation schemes.

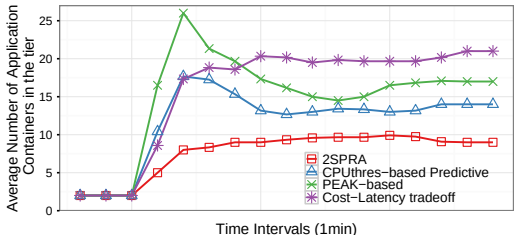

Fig. 8: Provisioning rate of application containers in 1-min intervals.

should not exceed the maximum workload affordable by containers set $\mathbb{S}$ in time interval t:

$$\sum_{i \in \mathbb{S}} \lambda_i^{max} \geq \lambda_t^{total}, \text{ s.t. } \lambda_i^{max} = \frac{u_i^{thres}}{c_i}$$

where c_i (= 0.6) is the average utilization a unit request rate imposes on a container i [14]. The average value for total workload $\bar{\lambda}_t^{total}$ is then predicted using Kalman filter, and its peak value $\hat{\lambda}_t^{total}$ is estimated as follows:

$$\hat{\lambda}_t^{total} = \bar{\lambda}_t^{total} + \gamma \cdot \sigma(\lambda_{t-1}^{total}) \cdot \frac{\bar{\lambda}_t^{total}}{\bar{\lambda}_{t-1}^{total}}$$

where γ is scaling factor (=7), and σ is the standard deviation of prior workload in time interval $t-1$. This scheme is susceptible to a number parameters that constitute essential parts of its design such as u^{three}, γ, and c_i. Determining their right combination requires offline measuring, most likely for different workload characteristics and servers' computing capacities. It is generally common practice for provisioned servers not to be overloaded beyond a certain threshold to ensure SLO conformance [14]. We measure CPU utilization for a container by parsing `/sys/fs/cgroup/cpu` hierarchy to read cumulative CPU time metrics [45] twice at two consecutive points in time displaced by a fixed interval. Then, we compute the difference between the two values divided by the interval length[5].

5.3 Experimental Results

In our experiments, we predefine a SLO threshold value of 10ms for mean response time, and hence, resources are auto-scaled to fulfill this objective. As we justified in Section 3.2, since G/G/m queuing system uses mean response time, 2SPRA considers average response time as performance metric. Considering tail latency instead is left as future work. Starting with 2 application containers in the application containers tier in our testbed (Figure 1), our 2SPRA aims at efficiently provisioning resources in accordance with varying workloads. We relate the efficient provisioning to the fact that only the necessary amount of resources are to be duly provisioned in such a way that it is adequate to accommodate incoming workload and yet SLO latency is fulfilled. Over-provisioning is measured by relating resources allocated by a scheme to the estimated application demands [5]. Application demands are estimated based on the peak processing capacity at which a single application server can fulfill the SLO requirement [2]. As such, over-provisioning measurements give a decent indication of SLO conformance and resource utilization [2] [5].

Amount of Resource Allocation. To study the relationship between the workload and the amount of resources provisioned in the application tier, and by using AOL traces, we use an offline profiling procedure to measure application's demands (in terms of number of application containers needed in the tier) to maintain application performance (i.e. average response time) below the SLO threshold of 10ms [38]. We measured peak throughput at 117 rps a container can achieve while fulfilling the SLO threshold. The number of application containers demanded (D) is then estimated as: current request arrival rate / peak throughput [2]. In other words, D is the estimated amount of resources required to maintain response time below its SLO threshold. As such, we define the provisioning coefficient as: $\delta = S_i/D$, where S_i is the amount of resources allocated by scheme i [5]. Recall that we support our earlier assumption by ensuring the application tier remains the bottleneck tier in all experiments. Figure 7 shows results of a set of experiments in which workload starts with 650 rps and assuming there is no sufficient history data initially. As the figure implies, notice that since all schemes provisioned resources S_i that are larger than application demand D (i.e. over-provisioning), they all maintained response time below the SLO ($<$ 10ms). Nevertheless, 2SPRA achieves the best performance with very little over-provisioning at the beginning due to the limited historical data. However, it provisions resources that are steadily approaching the actual demand (i.e. $\delta \sim 1$) over time as the application receives more requests. That is because 2SPRA uses both the throughput achieved by current containers in the tier and the actual workload (i.e. measured arrival rates) when it calculates resources needed for the next time interval. This makes it reactive to load variability as it occurs in the system.

Figures 8 and 9 present results of another set of experiments where the application tier is initialized with 2 containers and the AOL-driven workload starts at 150 rps. Since AOL-driven load is periodic with 3 similar repeating patterns (Figure 6), we depict in Figure 8 the result of the first pattern because similar results are observed for the other two patterns. However, we depict in Figure 9 the complete results for the whole experiments.

Figure 8 shows provisioning rates at which application containers are provisioned in the tier as the application executes over time. We have 3 observations. First, all schemes (but 2SPRA) extravagantly allocate excess containers when workload rapidly rises from 150 to 1000 rps. For instance,

5. This measurement method runs as a background Bash process and reports readings in a good match to the '`systemd-cgtop`'; a cgroup-aware monitoring tool built-in as an interactive-mode in CoreOS Linux.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TPDS.2016.2639009, IEEE Transactions on Parallel and Distributed Systems

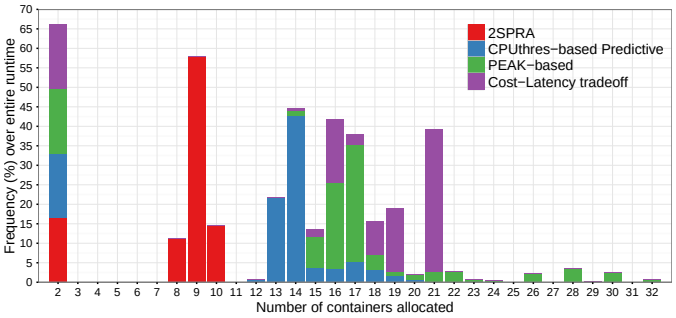

Fig. 9: Distribution of optimal numbers of Docker application containers allocated by various schemes, over entire runtime.

Allocation Scheme	Mean Latency
2SPRA	8.6
CPUthres-based Predictive	7.4
PEAK-based	6.4
Cost-Latency tradeoff	5.8

TABLE 1: Mean response time for different schemes.

PEAK-based scheme allocates 26 containers on average when workload spikes at 1000 rps. Second, some schemes took longer to stabilize its consecutive allocation decisions after each spike. For example, PEAK-based exhibits the slowest transition from 26 to 17 containers on average, because it relies on the tail of workload distribution to make its decisions. CPU Threshold-based Predictive scheme experiences similar behavior. Third, all schemes (but 2SPRA) demonstrate over-provisioning problem even after stabilization as excess containers are allocated than actual demand. In contrast, not only 2SPRA steadily re-sizes the tier as the workload requires, but it minimizes over-provisioning while still maintaining mean response time below the SLO of 10ms (Table 1). Because it consistently allocates the optimal amount of resources actually needed by the workload. By tracking both request arrivals and processing rates of the tier in a history window of 60 sec, 2SPRA is aware of workload changes, therefore it is robust in its behavior.

Figure 9 shows complete distributions of amounts of resources allocated in the tier by each resource allocation scheme. Each bar is aggregated over the entire experiment runtime as this exposes tendency in each scheme's behavior. Since it is very conservative, 2SPRA outperforms others as it tends to allocate only 9 containers in over 57.9% of the time (i.e. out of total number of its allocation decisions). However, for only 14.4% of its decisions, it decides on 10 application containers to accommodate rises in HTTP requests that reach ∼1150 rps in some occasions, and to consistently maintain response time below the SLO limit. Provisioning the application tier with 8 containers is also made for about 11.18% of time. Recall that the tier was initially provisioned with only 2 PHP application containers at the beginning of each experiment, this is represented by the time fraction of 16.5% where workload was light with average of 200 rps. As indicated, in this period of time, 2 containers were sufficient and no scaling was performed by any scheme.

On the other hand, other schemes behave differently. Cost-Latency tradeoff scheme over-provisions 21 containers in the tier for 37% of the runtime. It tends to be too greedy

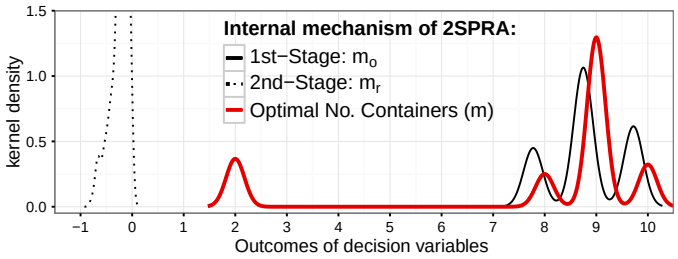

Fig. 10: Density of decision variables shows internal mechanism of 2SPRA.

in allocation whenever a rise in load is detected that may increase the latency. This non-resilient behavior is due to the bias (incurred by the importance ratio α's value; 0.8) that favors minimizing latency function $L(m)$ further by allocating more and more resources to reduce latency. Consequently, the cost function $C(m)$ that limits amounts of resources to allocate has less weight of $1 - \alpha$ (= 0.2) which explains the excessive resources allocated.

Since it relies on the tail of workload distribution to estimate peak workload, PEAK-based scheme experiences the widest range of allocations from 14 to 32 containers. For 30.2% of the time, it allocates 17 application containers in the tier. However, This scheme is highly vulnerable to any transient spikes in workload as they have immediate impact on the tail of workload distribution being profiled. Any spike in workload would enlarge its distribution's tail which would result in much larger amounts of resources to allocate than actually demanded. For example, the scheme decides to allocate 28, 30, and 32 containers in three different occasions where spikes have occurred. The scheme may suffer from high oscillation in its allocation decisions if the workload is highly variable. The CPU Threshold-based Predictive scheme performs slightly better. It allocates 13 and 14 containers for 21.8% and 42.8% of the time, respectively. However, Similar to PEAK-based, this scheme seems to over-provision resources (15-20 containers) when it encounters sudden rises in workload (see also Figure 8). Plausible reasons for this are: 1) it does not account for current throughput of the tier which makes it blind of processing rate at which requests are being served, it only uses arrival rates measured when making allocation decisions, and 2) it is a predictive approach, thus, it is inevitably prone to prediction errors when estimating maximum load.

To show the internal allocation mechanism of 2SPRA, Figure 10 presents the relationship of outcomes of the two stages' decision variables and their finalized values. Using smoothing kernel density of 512 points, an individual distribution for each variable is depicted, whose values are aggregated over entire runtime. First stage decision variable (m_o) lays in a range of positive values, whereas the recourse-decision variable (m_r) rectifies it, in the second stage, mainly from a range of negative values. The two variables are aggregated in red curve which represents the distribution of optimal final numbers (m) of application containers to allocate.

Resources Over-provisioning. In Table 2, we discuss over-provisioning problem incurred by each scheme throughout the entire execution of the application that demands number of containers (8.43) on average, and up to 10

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TPDS.2016.2639009, IEEE Transactions on Parallel and Distributed Systems

Allocation Scheme	Application Demand(D)		Resources Allocated by scheme		Over-provisioning (%) relative to Demand (D)	Total Allocation (%) relative to 2SPRA
	avg	max	avg	max		
2SPRA	8.43	10.00	9.04	10.00	7.75	n/a
CPUthres-based Predictive	8.43	10.00	14.34	20.00	68.33	56.22
PEAK-based	8.43	10.00	18.30	32.00	113.48	98.12
Cost-Latency tradeoff	8.43	10.00	18.99	21.00	121.45	105.51

TABLE 2: Performance comparison of schemes in over-provisioning incurred and total allocation achieved over entire runtime. 2SPRA provisions resources close to demand.

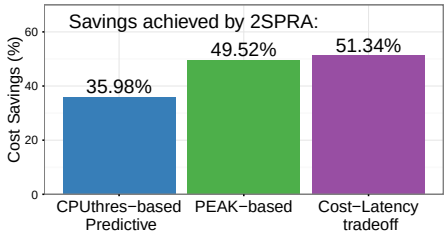

Fig. 11: Cost-savings 2SPRA achieves w.r.t others.

containers to serve workload of 1150 rps. 2SPRA achieves the best performance with only 7.75% over-provisioning of resources *relative to* the application's actual demand (i.e. 1.0775x the demand). While the CPU Threshold-based Predictive scheme allocates 68.33% more containers than application needs, PEAK-based and Cost-Latency schemes both incur severe over-provisioning of 113.48% and 121.45%, respectively; that is twice as much as the actual amount of resources demanded (∼2x demand). Hence, 2SPRA achieves the least allocations while still enabling the application to fulfill its performance SLO. Because it provisions resources close to the demand achieving the minimum over-provisioning and avoiding resource wastage. Table 2 also compares total allocations (in %) of different schemes *relative to* 2SPRA's. 2SPRA achieves 56.22-105.51% reduction in total resources provisioned while still maintaining the same QoS user experience. For example, 2SPRA allocates half what Cost-latency scheme has allocated (i.e. 105.51%).

Cost of Allocation. Figure 11 presents savings (in %) 2SPRA achieves in allocation costs aggregated over entire application's runtime. Since application containers are the resources being provisioned, we assume a single container represents a resource unit. Each container allocated to the application tier would incur the same cost per resource unit. That is because all containers are typically of identical capacity of processing requests, and all receive the same share of incoming requests distributed by the Round-Robin LB as stated earlier in Section 2.3. As such, cost savings (in %) are calculated based on weighted averages of different numbers of containers allocated in the tier over the entire runtime.

As can be seen, 2SPRA allocates amounts of resources whose costs are 35.98% cheaper than that of CPU Threshold-based Predictive scheme, and yet it achieves the same SLO-compliant response times (< 10ms) indicating the same web user experience is maintained intact. It also saves 49.52% and 51.34% of allocation costs incurred by PEAK-based and Cost-Latency tradeoff schemes, respectively. An apparent

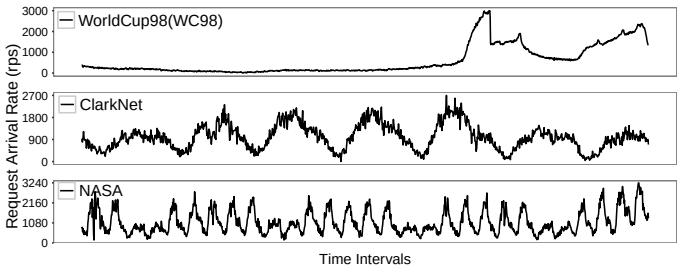

Fig. 12: Workloads generated based on 3 real web traces (WC98, ClarkNet, NASA).

Web Trace	Selected Trace Length	Load Intensity	Peak Load (rps)	Allocation Scheme	Application Demand(D) avg (max)	Resources Allocated by scheme avg (max)	(%) Total Allocation relative to 2SPRA
WC98	24 hrs	per-1min	2916	**2SPRA**	4.96 (24.92)	**5.23 (25.00)**	n/a
				CPUthres-based Predictive		7.39 (49.00)	41.19
				PEAK-based		11.59 (86.00)	121.52
				Cost-Latency tradeoff		15.77 (50.00)	201.47
ClarkNet	7 days	per-10min	2111	**2SPRA**	8.48 (18.00)	**9.03 (18.00)**	n/a
				CPUthres-based Predictive		18.57 (53.00)	105.68
				PEAK-based		22.74 (78.00)	151.83
				Cost-Latency tradeoff		23.34 (47.00)	158.46
NASA	31 days	per-30min	2976	**2SPRA**	9.55 (25.44)	**10.02 (26.00)**	n/a
				CPUthres-based Predictive		25.34 (88.00)	152.82
				PEAK-based		26.34 (93.00)	162.83
				Cost-Latency tradeoff		25.19 (50.00)	151.31

TABLE 3: Across entire execution, 2SPRA outperforms others in allocating amount of resources close to application's demands under workloads generated from 3 traces.

reason for such high performance, 2SPRA tends to allocate resources closely match what the application actually needs as the workload varies, which allows it to keep over-provisioning to a minimum. This enables 2SPRA to obtain huge savings in resources allocated on the long term, as savings would accumulate to significant figures especially for web services/applications that are typically designed with long-running nature.

Experiments with Other Real Web Traces. In addition to AOL traces used so far, we further evaluate 2SPRA performance with respect to other schemes' using workloads generated based on 3 additional real world web traces of different realistic traffic variations [46], as shown in Figure 12. We selected various lengths for the traces with different workload intensities to modulate the request rate of RUBiS benchmark generating load up to 3000 requests per second (rps), detailed in Table 3: (**1**) WorldCup98 (WC98) web server trace is from 1998-06-30 08:00:01 to 1998-07-01 08:00:00, (**2**) ClarkNet is from 1995-09-04 00:00:27 to 1995-09-10 23:59:04, and (**3**) NASA is from 1995-08-01 00:00:01 to 1995-08-31 23:59:53.

Table 3 presents the detailed results. Under WC98-based traffic variability, 2SPRA allocates 5.23 containers on average. It also provisions a maximum of 25 containers to handle peak load of ∼2900 rps in some occasions (Figure 13 shows internal decisions of the two stages). 2SPRA is

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TPDS.2016.2639009, IEEE Transactions on Parallel and Distributed Systems

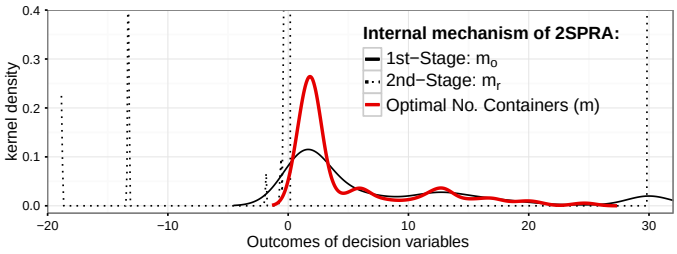

Fig. 13: Density of decision variables of the internal mechanism of 2SPRA resulted under WC98 workload. The range of values of the 2nd-stage variable is quite large i.e. (-18.84,30) with mean of 2.59.

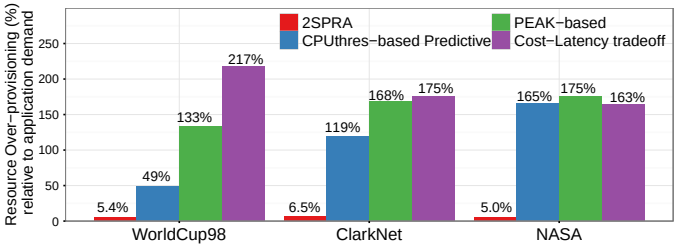

Fig. 14: Over-provisioning resources incurred by all allocation schemes relative to application demand in the entire runtime, under different trace-driven workloads.

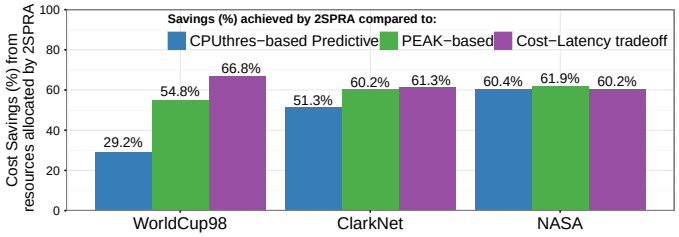

Fig. 15: Savings in allocation cost achieved by 2SPRA w.r.t. other schemes over entire runtime, under different trace-driven workloads.

able to estimate resource demands of the application as it takes into consideration both the current throughput of the application tier as well as request arrival rates. Therefore, it is capable to provision resources closely match with application demand estimated at 4.96 containers on average and up to 24.92 containers on maximum. As such, it minimizes resources over-provisioning down to 5.0-6.5% relative to application demand under different trace-driven loads, as Figure 14 shows. Other allocation schemes suffer from excessive resource wastage. For example, under WC98-driven load, Cost-Latency scheme allocates amounts of containers of 217% more than the demand, that is 3x what application needs. Similarly, CPU Threshold-based Predictive and PEAK-based schemes also excessively over-provision 49%(i.e. 1.5x demand) and 133%(i.e. >2x demand) more resources than application demands, respectively. Similar observations are made with ClarkNet- and NASA-driven experiments (Table 3 and Figure 14).

Figure 15 summarizes savings (in %) 2SPRA achieves in allocation costs throughout application's entire runtime under each trace-driven load. 2SPRA delivers substantial cost savings measured in the range 29-67% when allocating resources for all workloads considered. It is attributed to the consistent tendency 2SPRA has in reducing total resources allocated irrespective of workloads applied with different trace-based load variations. For example, 2SPRA achieves 41.19-201.47% total resource reduction compared to other schemes, under load driven by WC98 (see Table 3).

6 Related Work

Efficiency has long been a primary goal of resource management in computer systems irrespective of their sizes. Clouds are no exception. The large scale of clouds puts the significance of this goal at a new level. As a result, there have been many studies to deal with resource provisioning/scaling in clouds exploiting the elasticity and malleability of clouds (e.g., [7] [34] [47]). Typically, resource allocation mechanisms aim at provisioning resources for the sake to fulfill SLOs on application performance. Nevertheless, heterogeneity in resources and/or uncertainty in workloads pose difficulties to maintain SLOs. In our previous work [8], we address a resource allocation problem for data analytics applications for which computing clusters can be constructed from heterogeneous resources offered in clouds. The resource allocation mechanism proposed is a mixed-integer programming optimization model that optimizes the selection of VMs to construct clusters which can achieve performance targets, e.g. meeting application deadlines.

Generally speaking, related works on resource scaling can be classified in a number of categories. The time-series prediction category encloses those works we used in our evaluation ([14] [16]) except for the work [15] which may be categorized under queuing theory. Most other related works can fit in one or more of the following categories: threshold-based rules, queuing theory, and control theory [32]. Xiong et al. [20] propose a proportional-integral performance controller to control the mean round-trip time (RTT) for n-tier web applications. It uses CPU-resource partitioning scheme that employs M/G/1/PS queuing theory to model CPU resource in each tier with Poisson request arrivals. However, the controller neglects the effect of contention (i.e. queuing delay) for non-CPU resources (e.g. access to databases; DB) on mean RTT. Because it assumes adequate amounts of non-CPU resources are provisioned. As a result, optimal solutions obtained depend only on CPU consumption. In contrast, 2SPRA accounts for delays realistically posed during the entire processing lifecycle for a request to complete. It explicitly considers request service time incurred and its variability resulted from both resource contention and request type itself.

Rui et al. [44] present threshold-based lightweight scaling scheme. It sets a lower and upper bound to monitor end-to-end response times of multi-tier web applications to initiate resource- or VM-level scaling. However, this scheme has no way to speculate the incoming workload other than sensing its footprints on response time (application-level) and then via consumptions (resource-level). Because its design lacks explicit consideration for the variations in web traffic. As such, this scheme suffers from a prolonged scaling-optimization step. Because it has to repeatedly keep running until threshold violations are all cleared (which is subject to traffic variability). As a result, extra scaling-overhead and induced service disruption is highly likely to rise (similar to other schemes [14] [48]). In contrast, 2SPRA is inherently stochastic at its core model, which allows it to explicitly

account for any traffic variations. It builds up necessary input information with which it efficiently seeks optimal provisioning in an acceptable very short optimization step, i.e. eliminated overhead and reduced service disruption.

Li et al. [38] conduct extensive performance study analyzing the effects of queuing theory models on response tail latency. It examines the relationships that govern: server utilization, throughput, queuing service disciplines, requests arrival rates, and response time. It further highlights experimental results verified on different web-server architectures such as: a multi-threaded null RPC (remote procedure call) server (e.g. Apache), and event-driven architecture (e.g. Nginx servers). Since real queues tend to be non steady state and time-dependent, it is hard to solve real problems using classical queuing with 'exact' solutions. As such, we employ, in our work, an 'approximate' queuing analysis for multiple-servers tier which nicely suited servers containerized in our real web hosting platform setup.

7 Conclusion and Future Work

To maintain SLOs, over-provisioning resources has become a common practice to deal with uncertainty in workloads that imposes variations in resource demands. As a result, cloud-based datacenters are left with inefficient resource utilization for cloud providers, and cost-inefficient resource provisioning for application owners to deal with. In this paper, we propose, implement, and verify a novel resource allocator (2SPRA) that addresses uncertainty in workloads as a stochastic process to provision resources for containerized n-tier web applications in clouds. The resource allocation problem is modeled as a Two-stage stochastic programming optimization model. The 2SPRA optimizes resources provisioned in the application containers tier in accordance with the fluctuations of incoming workload to accommodate a predefined SLO latency. Compared to three existing allocation schemes, 2SPRA achieves the minimum over-provisioning as it allocates resources closely match application's actual demands, using 4 real-world web traces of different load variations.

There are possible avenues to extend these results for future work. As opposed to using G/G/m queuing, alternative models can be built to define the relationship between resources provisioned, workloads applied, and SLOs defined. This might require extensive performance study for web services. Two-stage stochastic programming modeling can then be used to drive searching for optimal solutions. Uncertainty is a key factor in such stochastic model. Another possible avenue to further optimize resources allocated and improve their utilization, is to adopt a workload-aware load dispatching policy to support consolidating incoming loads into application servers as per their utilizations. Introducing dynamic resource provisioning for database tier can be an another interesting extension to our work. The interaction between application and database tiers needs to be modeled to relate incoming workloads to the application performance observed.

Acknowledgments

We would like to thank David Cater-Cameron, IT Manager-ICT, and Greg Ryan for their assistance. We sincerely thank our colleagues for their constructive and thoughtful comments that helped improve this work.

References

[1] J. Leverich and C. Kozyrakis, "Reconciling high server utilization and sub-millisecond quality-of-service," in *Proc. of European Conf. on Computer Systems (EuroSys)*, 2014, p. 4.

[2] A. Gandhi, T. Zhu, M. Harchol-Balter, and M. A. Kozuch, "Softscale: stealing opportunistically for transient scaling," in *Proc. of ACM/IFIP/USENIX Middleware Conference*, 2012, pp. 142–163.

[3] "Apple services outage," Mar 11 2015. [Online]. Available: http://money.cnn.com/2015/03/11/technology/apple-app-store-down/.

[4] T. Flach, N. Dukkipati, A. Terzis, B. Raghavan, N. Cardwell, Y. Cheng, A. Jain, S. Hao, E. Katz-Bassett, and R. Govindan, "Reducing web latency: the virtue of gentle aggression," in *ACM SIGCOMM Computer Communication Review*, vol. 43, no. 4, 2013, pp. 159–170.

[5] H. Liu and B. He, "Reciprocal resource fairness: Towards cooperative multiple-resource fair sharing in iaas clouds," in *Proc. of Int'l Conf. for High Performance Computing, Networking, Storage and Analysis*, ser. SC '14, 2014, pp. 970–981.

[6] Benjamin Hindman, "Why the data center needs an operating system," 2016. [Online]. Available: https://www.oreilly.com/ideas/why-the-data-center-needs-an-operating-system.

[7] H. Nguyen, Z. Shen, X. Gu, S. Subbiah, and Wilkes, John, "Agile: Elastic distributed resource scaling for infrastructure-as-a-service," in *Proc. of USENIX Conference on Automated Computing (ICAC13)*, 2013.

[8] O. Adam, Y. Lee, and A. Zomaya, "Constructing performance-predictable clusters with performance-varying resources of clouds," *IEEE Transactions on Computers*, vol. PP, no. 99, pp. 1–1, 2015.

[9] W. Felter, A. Ferreira, R. Rajamony, and J. Rubio, "An updated performance comparison of virtual machines and linux containers," *IBM Research Division*, vol. 28, p. 32, 2014, Austin, TX, USA.

[10] R. Dua, A. Raja, and D. Kakadia, "Virtualization vs containerization to support paas," in *Proc. of IEEE Int'l Conference on Cloud Engineering (IC2E)*, March 2014, pp. 610–614.

[11] A. Verma, L. Pedrosa, M. Korupolu, D. Oppenheimer, E. Tune, and J. Wilkes, "Large-scale cluster management at google with borg," in *Proc. of European Conf. on Computer Systems (EuroSys)*, 2015, p. 18.

[12] L. K. Dmitrey, "OpenOpt Optimization Framework: Global Optimization Solvers (GLP)," *National Academy of Sciences of Ukraine*, 2015. [Online]. Available: http://openopt.org/StochasticProgramming.

[13] ——, "OpenOpt: A Python module for numerical optimization," *National Academy of Sciences of Ukraine*, 2016. [Online]. Available: https://pypi.python.org/pypi/openopt.

[14] Z. Abbasi, G. Varsamopoulos, and S. K. Gupta, "Thermal aware server provisioning and workload distribution for internet data centers," in *Proc. of ACM Int'l Symposium on High Performance Distributed Computing(HPDC)*, 2010, pp. 130–141.

[15] J. Jiang, J. Lu, G. Zhang, and G. Long, "Optimal cloud resource auto-scaling for web applications," in *Proc. of IEEE/ACM Int'l Symposium on Cluster, Cloud and Grid Computing (CCGrid)*, 2013, pp. 58–65.

[16] B. Urgaonkar, P. Shenoy, A. Chandra, P. Goyal, and T. Wood, "Agile dynamic provisioning of multi-tier internet applications," *ACM Transactions on Autonomous and Adaptive Systems (TAAS)*, vol. 3, p. 1, 2008.

[17] C. Reiss, A. Tumanov, G. R. Ganger, R. H. Katz, and M. A. Kozuch, "Heterogeneity and dynamicity of clouds at scale: Google trace analysis," in *Proc. of ACM Symposium on Cloud Computing (SoCC)*, 2012, p. 7.

[18] "Rice University Bidding System (RUBiS)," 2016. [Online]. Available: http://rubis.ow2.org/.

[19] A. Nigmatulin, "High-Performance PHP FastCGI Process Manager," 2016. [Online]. Available: http://php-fpm.org/about/.

[20] P. Xiong, Z. Wang, S. Malkowski, Q. Wang, D. Jayasinghe, and C. Pu, "Economical and robust provisioning of n-tier cloud workloads: A multi-level control approach," in *Proc. of Int'l Conference on Distributed Computing Systems (ICDCS)*, 2011, pp. 571–580.

[21] W. Randolph, *Queueing methods. For services and manufacturing*. Prentice Hall., 1991.

[22] W. Whitt, "Efficiency-driven heavy-traffic approximations for many-server queues with abandonments," *Management Science*, vol. 50, no. 10, pp. 1449–1461, 2004.
[23] ——, *Stochastic-process limits: an introduction to stochastic-process limits and their application to queues*. Columbia University: Springer Science & Business Media, 2002.
[24] CoreOS Inc., "CoreOS: A container-focused Operating System," 2016. [Online]. Available: https://coreos.com/using-coreos/.
[25] Docker Inc., "Docker platform: Docker Engine and Docker Containers," 2016. [Online]. Available: https://www.docker.com/docker-engine.
[26] "Containers Images on Docker Hub Registry," 2016. [Online]. Available: https://hub.docker.com/u/omera/.
[27] Google kubernetes, "kubernetes: Container Orchestration, deployment, scaling, and management," 2016. [Online]. Available: http://kubernetes.io/.
[28] Docker Inc., "Swarm: a Docker-native clustering system," 2016. [Online]. Available: https://github.com/docker/swarm/.
[29] Amazon Ltd., "Amazon Elastic Compute Cloud (EC2)," 2016. [Online]. Available: http://aws.amazon.com/ec2/.
[30] "AOL real Web Traces," 2016. [Online]. Available: http://www.cim.mcgill.ca/~dudek/206/Logs/AOL-user-ct-collection/.
[31] G. Pass, A. Chowdhury, and C. Torgeson, "A Picture of Search." in *InfoScale*, vol. 152, 2006, p. 1.
[32] P. Padala, K.-Y. Hou, K. G. Shin, X. Zhu, M. Uysal, Z. Wang, S. Singhal, and A. Merchant, "Automated control of multiple virtualized resources," in *Proc. of European Conf. on Computer systems (EuroSys)*, 2009, pp. 13–26.
[33] G. Jung, K. R. Joshi, M. A. Hiltunen, R. D. Schlichting, and C. Pu, "A cost-sensitive adaptation engine for server consolidation of multitier applications," in *Proc. of ACM/IFIP/USENIX Middleware Conference*, 2009, pp. 163–183.
[34] Z. Shen, S. Subbiah, X. Gu, and J. Wilkes, "Cloudscale: elastic resource scaling for multi-tenant cloud systems," in *Proc. of ACM Symposium on Cloud Computing (SoCC)*, 2011, p. 5.
[35] N. Kaviani, E. Wohlstadter, and R. Lea, "Cross-tier application and data partitioning of web applications for hybrid cloud deployment," in *Proc. of ACM/IFIP/USENIX Middleware Conference*, 2013, pp. 226–246.
[36] C. Wilson, H. Ballani, T. Karagiannis, and A. Rowtron, "Better never than late: Meeting deadlines in datacenter networks," in *ACM SIGCOMM Computer Communication Review*, vol. 41, 2011, pp. 50–61.
[37] B. Vamanan, J. Hasan, and T. Vijaykumar, "Deadline-aware datacenter tcp (d2tcp)," *ACM SIGCOMM Computer Communication Review*, vol. 42, no. 4, pp. 115–126, 2012.
[38] J. Li, N. K. Sharma, D. R. Ports, and S. D. Gribble, "Tales of the tail: Hardware, OS, and Application-level sources of Tail Latency," in *Proc. of ACM Symposium on Cloud Computing (SoCC)*, 2014, pp. 1–14.
[39] C. Curino, E. P. Jones, S. Madden, and H. Balakrishnan, "Workload-aware database monitoring and consolidation," in *Proc. of ACM SIGMOD International Conference on Management of data*, 2011, pp. 313–324.
[40] W. Zhao and H. Schulzrinne, "Predicting the upper bound of web traffic volume using a multiple time scale approach." in *Proc. of Int'l World Wide Web Conference (WWW)*, 2003.
[41] R. H. Shumway and D. S. Stoffer, *Time series analysis and its applications*. Springer Science & Business Media, 2013.
[42] W. Winston and J. Goldberg, *Operations Research: Applications and Algorithms*, 4th ed. Duxbury press Boston, 2004, ch. Queuing Theory.
[43] P. G. Harrison and N. M. Patel, *Performance Modelling of Communication Networks and Computer Architectures*. Addison-Wesley Longman Publishing Co., Inc., 1992.
[44] R. Han, L. Guo, M. Ghanem, and Y. Guo, "Lightweight resource scaling for cloud applications," in *Proc. of IEEE/ACM Int'l Symposium on Cluster, Cloud and Grid Computing (CCGrid)*, May 2012, pp. 644–651.
[45] "Docker runtime metrics," 2016. [Online]. Available: https://docs.docker.com/engine/admin/runmetrics/.
[46] "The Internet Traffic Archive," 2016. [Online]. Available: http://ita.ee.lbl.gov/html/traces.html.
[47] H. Han, Y. C. Lee, W. Shin, H. Jung, H. Y. Yeom, and A. Y. Zomaya, "Cashing in on the cache in the cloud," *IEEE Transactions on Parallel and Distributed Systems*, vol. 23, no. 8, pp. 1387–1399, Aug 2012.
[48] W. Iqbal, M. N. Dailey, and D. Carrera, "Sla-driven dynamic resource management for multi-tier web applications in a cloud," in *Proc. of IEEE/ACM Int'l Symposium on Cluster, Cloud and Grid Computing (CCGrid)*, 2010, pp. 832–837.

Omer Y. Adam is a PhD student in the School of Information Technologies at The University of Sydney, Australia. His research interests are in the area of distributed computing systems with particular interest in resource allocation and management in cloud computing.

Dr.Young Choon Lee is an Australian Research Council DECRA Researcher and a lecturer in the Department of Computing, Macquarie University, Sydney, Australia. His research interests include resource management and scheduling in distributed and high performance computing systems.

Prof.Albert Y. ZOMAYA is currently the Chair Professor of High Performance Computing & Networking in the School of Information Technologies, The University of Sydney. He is also the Director of the Centre for Distributed and High Performance Computing which was established in late 2009. Professor Zomaya published more than 500 scientific papers & articles and is author, co-author/ editor of more than 20 books.

He served as the Editor in Chief of the IEEE Transactions on Computers (2011-2014). Currently, Professor Zomaya serves as an associate editor for 22 leading journals, such as, the ACM Computing Surveys, IEEE Transactions on Computational Social Systems, IEEE Transactions on Cloud Computing, and Journal of Parallel and Distributed Computing. He delivered more than 150 keynote addresses, invited seminars, and media briefings and has been actively involved, in a variety of capacities, in the organization of more than 600 national and international conferences.

Professor Zomaya is the recipient of the IEEE Technical Committee on Parallel Processing Outstanding Service Award (2011), the IEEE Technical Committee on Scalable Computing Medal for Excellence in Scalable Computing (2011), and the IEEE Computer Society Technical Achievement Award (2014). He is a Chartered Engineer, a Fellow of AAAS, IEEE, and IET (UK). Professor Zomaya's research interests are in the areas of parallel and distributed computing and complex systems.