Softw Syst Model (2018) 17:509–526
https://doi.org/10.1007/s10270-017-0584-y

THEME SECTION PAPER

Model-driven optimal resource scaling in cloud

Anshul Gandhi[2] · **Parijat Dube**[1] · **Alexei Karve**[1] · **Andrzej Kochut**[1] · **Li Zhang**[1]

Received: 7 July 2015 / Revised: 3 August 2016 / Accepted: 19 January 2017 / Published online: 6 February 2017

Abstract Cloud computing offers the flexibility to dynamically size the infrastructure in response to changes in workload demand. While both horizontal scaling and vertical scaling of infrastructure are supported by major cloud providers, these scaling options differ significantly in terms of their cost, provisioning time, and their impact on workload performance. Importantly, the efficacy of horizontal and vertical scaling critically depends on the workload characteristics, such as the workload's parallelizability and its core scalability. In today's cloud systems, the scaling decision is left to the users, requiring them to fully understand the trade-offs associated with the different scaling options. In this paper, we present our solution for optimizing the resource scaling of cloud deployments via implementation in OpenStack. The key component of our solution is the modeling engine that characterizes the workload and then quantitatively evaluates different scaling options for that workload. Our modeling engine leverages Amdahl's Law to model service timescaling in scale-up environments and queueing-theoretic concepts to model performance scaling in scale-out environments. We further employ Kalman filtering to account for inaccuracies in the model-based methodology and to dynamically track changes in the workload and cloud environment.

Keywords Autoscaling · Modeling · Scale-up · Scale-out · Cost · Optimal · Experimentation · Implementation

1 Introduction

Cloud computing provides users the opportunity to quickly deploy their applications/workloads in an elastic setting using a "pay-as-you-go" pricing model. Cloud-deployed workloads can be easily scaled in response to changing workload demand by leveraging the cloud framework (see, for example [1,2]). Cloud workloads are scaled up when the workload demand increases to handle the additional traffic without violating performance Service Level Agreements (SLAs) and can be scaled down when the workload demand decreases to save on rental costs.

An application or workload can be horizontally scaled ("scale-out") by adding more virtual machines (VMs) to the existing deployment and can be vertically scaled ("scale-up") by adding more resources (such as vCPUs, memory, etc.) to the existing VMs. The choice of scaling can have a significant impact on the cost and performance of the workload. For example, we show in Sect. 5 that scale-up can be 3 times more cost-effective than scale-out when the workload demand is low, whereas scale-out can be 2 times more cost-effective when the workload demand is high. However, the results

Communicated by Dr. Kai Sachs and Catalina Llado.

✉ Parijat Dube
pdube@us.ibm.com

Anshul Gandhi
anshul@cs.stonybrook.edu

Alexei Karve
karve@us.ibm.com

Andrzej Kochut
akochut@us.ibm.com

Li Zhang
zhangli@us.ibm.com

[1] IBM T. J. Watson Research Center, Yorktown Heights, NY 10598, USA

[2] Stony Brook University, Stony Brook, NY 11790, USA

also depend on the performance SLA and the cost function.[1] Thus, it is not at all obvious as to how the workload should be scaled in response to varying traffic conditions.

Importantly, the choice of scaling also depends on the workload characteristics. Parallel workloads that can be parallelized on multiple cores to reduce their service time can significantly benefit from scale-up. Of course, the benefits of scale-up can start to diminish after a certain level of scaling due to parallelization and synchronization overhead that the workload and system incur (see Fig. 1b and Sect. 3). Sequential workloads, on the other hand, do not benefit from multiple cores in terms of service time. However, they can benefit from increased concurrency due to scale-up if multiple workload requests need to be serviced simultaneously. In terms of scale-out, however, both parallel and sequential workloads should be affected similarly—benefiting from the increased concurrency without any change in per-request service time (see Figs. 4a, 5a in Sect. 4.5).

Given these workload-dependent differences in scale-up and scale-out, we assert that the optimal scaling solution should account for the impact of workload characteristics on the resulting performance. One approach to scaling is to explore all possible scaling options and then pick the most cost-effective option that meets the performance SLA. This is clearly an inefficient approach, which is made even more complicated by the fact that a scaling decision can involve scale-out and scale-up simultaneously, thereby significantly increasing the space of possible scaling options.

There have been several empirical studies [3–5] aimed at understanding the scalability of workloads on multi-core, multi-thread systems in scale-up architectures. Most of these studies are targeted at characterizing workload performance in terms of system architecture (CPU cores, SMT level, cache size, memory size and type) and in identifying resource contentions in the software and/or hardware stack. While some models linking workload performance to specific architecture features have been proposed [6–8], to the best of our knowledge, no attempts have been made to leverage workload characterization for optimizing horizontal and/or vertical resource scaling in clouds.

In this paper, we propose a model-based approach to assess the impact of workload on cloud resource scaling. The key component of our solution is the modeling engine that characterizes the workload and its horizontal and vertical scalability. Our modeling engine leverages Amdahl's Law to model service timescaling in scale-up environments and queueing-theoretic concepts to model performance scaling in scale-out environments. Using our model, we dynamically determine the most cost-effective scaling option, including combinations of scale-out and scale-up, that will result in the required performance for any given workload.

The main contributions of this paper are:

- We experimentally investigate the impact of horizontal and vertical scaling on cost, performance, and provisioning times for different cloud-deployed applications (Sect. 3).
- We develop a performance model using Amdahl's Law and queueing-theoretic principles to predict the trade-off of various scaling options (Sect. 4). We also employ a Kalman filtering-based approach to dynamically infer the (possibly changing) system parameters required by our model with minimal benchmarking efforts (Sect. 4). The Kalman filter also provides robustness against inaccuracies that are inherent in model-driven approaches.
- We leverage our performance model to analyze the optimal scaling for various workloads on various architectures (Sect. 5).
- We implement and validate our scaling solution on OpenStack and demonstrate the benefits of our optimal scaling approach by comparing with other existing approaches (Sect. 6).

2 Experimental setup

We use OpenStack [9] as the underlying scalable cloud system. The VMs for the applications are created on an OpenStack-managed private cloud deployment on Softlayer [2] physical machines. We employ multiple hypervisors with 8 CPU cores and 8 GB of memory each. Scale-up and scale-out are executed via the OpenStack API. We use Chef [10] to automate the installation of software on VMs during boot. At a high level, applications are deployed as a collection of VMs in OpenStack. We experiment with two different application workloads: (i) RUBiS [11], which is a sequential workload, and (ii) LINPACK [12], which is a parallel workload. We discuss the specifics of these workloads and their deployment in Sect. 2.1. We monitor the resource consumption and application performance using our monitoring agent, described in Sect. 2.2. The monitored statistics are then forwarded to the modeling engine, described in Sect. 2.3, which performs the workload characterization and analyzes the different scaling options.

2.1 Workloads

2.1.1 RUBiS

We use the open-source multi-tier application, RUBiS [11], as a representative sequential workload for our experiments.

[1] We acknowledge that the per-VM prices of various cloud providers depend on external factors such as profit margins and market fluctuations. As such, we will often use the amount of resources employed by the workload or application as a proxy for the "cost" and only use cloud providers' listed prices for specific use cases.

RUBiS is an auction site prototype modeled after eBay.com supporting 26 different classes of web requests such as bid, browse, and buy. Our implementation of RUBiS employs Apache as the front-end web server, Tomcat as the Java servlets container, and MySQL as the back-end database. We employ RUBiS's benchmarking tool, which emulates user behavior by defining sessions consisting of a sequence of requests, to generate load for our experiments. The think time between requests is exponentially distributed with a mean of 1 s. We chose to keep mean think time to be 1 s as in commercial web benchmarks like TPC-W the think time is in the range of few seconds [13]. We fix the number of clients for each experiment and vary the load by dynamically changing the composition of the request classes mix. In our experimental investigation with RUBiS, we verified that the server utilization on Apache server and database is <30% and they were never the bottlenecks even at high loads. Thus we focus on scaling the Tomcat tier which quickly becomes bottleneck as load increases.

The Apache and MySQL tiers are each hosted on a 4-vCPU VM. The Tomcat application tier is hosted on multiple VMs. We use host aggregates and availability zones (which are essentially logical cloud partitions) offered by the OpenStack nova scheduler to place the Apache and MySQL VMs on one hypervisor and Tomcat VMs on a different set of hypervisors.

We set the SLA in terms of mean response time. We focus on the mean response time of browse requests since customers often base their web experience based on how long it takes to browse through online catalogs. We want the response time SLA for the browse requests to be <35 ms, on average. The secondary goal is to minimize the total cost of Tomcat VMs employed by the application. We use the total number of vCPUs employed by the application tier as a proxy for cost, unless stated otherwise. Thus, the overall objective is to minimize the resource cost while ensuring that the 35-ms SLA for the browse requests is not violated. In our experiments, we find that the application performance is most affected by the processing power (number of VMs and number of cores per VM) and is much less affected by the allocated memory and storage space. Similar observation about RUBiS performance was also made in [14]. We thus focus on scaling the processing power of the application tier.

2.1.2 LINPACK

We use Intel LINPACK [12] as a representative parallel workload for our experiments. LINPACK executes a collection of linear algebra operations on matrices of a given input size. The workload is inherently parallelizable, with a small sequential portion during the initialization phase. We run LINPACK in a transactional manner by using httperf [15] as the workload generator and Apache as the front-end load balancer that distributes requests to back-end application servers running LINPACK.

The httperf application and the Apache load balancer are each hosted on a 4-vCPU VM. The LINPACK application tier is hosted on multiple VMs. We set the mean response time SLA for LINPACK requests as 5 s. The overall objective is to minimize the resource cost of the LINPACK application tier while ensuring that the 5-s SLA is not violated. Since LINPACK is CPU-bound, we focus on scaling the processing power of the LINPACK VMs.

2.1.3 Other workloads

For modeling and analysis purposes, we also make use of SPECjbb2005 [16] and the MediaWiki [17] applications. These are both sequential workloads. SPECjbb2005 is a three-tier Java benchmark where each driver thread independently works on its own data. We focus on the scalability of the Java (business logic engine) middle tier. MediaWiki is a three-tier web server workload. We focus on the scalability of the PHP runtime instances. We make use of the scalability data published in [4] for our modeling and analysis of SPECjbb2005 and MediaWiki.

2.2 Monitoring agent

While most of the monitoring information required for our model-driven approach can be obtained directly from the physical infrastructure, some of the more critical information, such as per-tier service times, requires invasive benchmarking which is not possible for a cloud service provider. For estimating such unobservable parameters, we employ Kalman filtering (see Sect. 4).

We use virt-top (part of the libvirt [18] package) to collect VM CPU utilization statistics from each hypervisor periodically. For the application-level metrics, we periodically monitor the request URLs directed at the application deployment to compute the request rate and response time. Note that the user can choose to provide these metrics to us directly (for example, using a REST call). The monitoring interval is set to 10 s.

2.3 Modeling engine

The modeling engine is responsible for workload characterization and analysis of the various scaling options. The modeling engine is invoked whenever a new entry is recorded by the monitoring engine (once every 10 s). The modeling engine consumes this information and infers the important system parameters via the Kalman filtering approach (Sect. 4). The underlying queueing model, along with the inferred parameters, allows the modeling engine to determine the optimal scaling action, if any is needed. We experimen-

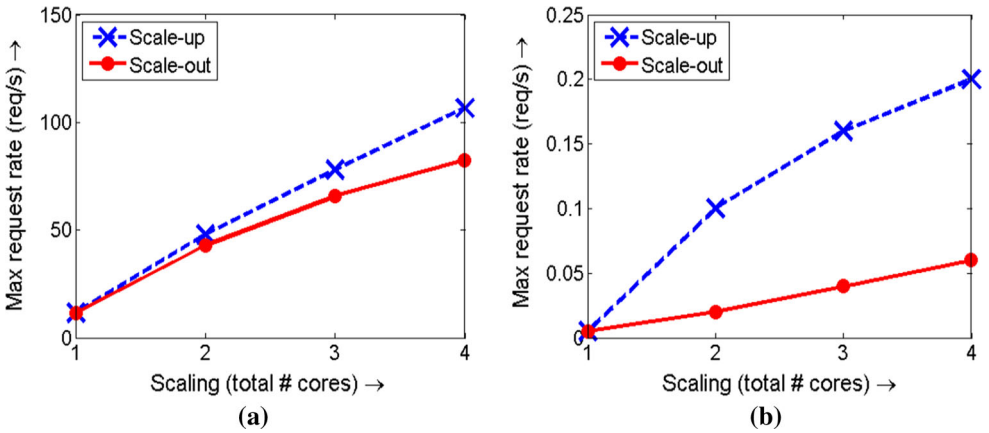

Fig. 1 Performance of scale-up and scale-out in terms of maximum allowable request rate for a fixed SLA. **a** Sequential workload RUBiS [11], **b** parallel workload LINPACK [12]

tally validate our modeling engine logic via implementation in OpenStack in Sect. 6.

3 Performance–cost trade-offs

In this section, we investigate the effects of scale-up and scale-out on performance and cost. For performance, we look at the maximum load, or request rate, that the system can handle without violating the response time SLA (35 ms for RUBiS and 5s for LINPACK). For cost, we look at hourly rental prices based on Amazon EC2's General Purpose On-Demand Instances [19], Rackspace's Performance 1 Flavor Managed Cloud Service [20], and Softlayer's CloudLayer Instances [21]. We also consider the provisioning time, which is the time taken to execute the scaling action.

3.1 Performance

Figure 1 shows our experimental results for the maximum request rate that can be handled by the application (without violating the SLA) as a function of the scaling. The scaling is in units of total number of cores employed by the application tier. In particular, scaling refers to the total number of vCPUs on the single VM in case of scale-up and the number of 1-vCPU VMs in case of scale-out.

In Fig. 1a, we see that performance scales almost linearly under RUBiS for both scale-out and scale-up. Since RUBiS is a sequential workload, adding more cores to the VM or adding more VMs has the same effect: increased computing power to handle greater concurrency. There are, of course, differences in the two different types of scaling. Scale-up introduces more contention for OS and locks, whereas scale-out requires more work at the load balancer.

In Fig. 1b for LINPACK, we see that performance increases significantly under scale-up initially and then starts to plateau. We claim that this concave downwards shape is because of diminishing gains obtained from scale-up. Under scale-out, performance scales linearly. However, for LINPACK, we see that scale-up results in much better performance than scale-out. This is because LINPACK is parallelizable across cores on a VM, and thus, scale-up decreases the service time of LINPACK. This allows for more aggressive sharing of resources at the VM. By contrast, under scale-out, the service time of a single LINPACK request does not change as threads of a single LINPACK request cannot be executed across different VMs (they can only be executed across cores of the same VM). Thus, to get good response times by directly decreasing the size (service time) of a LINPACK request, a VM should only host one request at any time.

Observe that the frequency of a single vCPU is not changed in our setting as we are not exploiting dynamic frequency scaling [22]. Dynamically changing frequency of vCPUs as an alternate to increasing number of vCPUs (with fixed, static frequency) has also been employed in existing solutions (see e.g., VMware vCenter [23]). This will add another dimension in the scaling decision problem and may result in different scaling behavior for sequential workloads like RUBiS. However, CPU frequency scaling is unappealing due to its limited range [24] and diminishing returns, since higher clock frequency causes more energy dissipation [22].

The observed differences in performance of scale-up and scale-out for the sequential and parallel workloads in Fig. 1 motivate us to develop a model to better understand the impact of workload on scaling. We explain our modeling and analysis in Sect. 4.

3.2 Cost

Figure 2 shows the per-instance hourly rental costs for different cloud providers. We obtain the cost of scale-out by

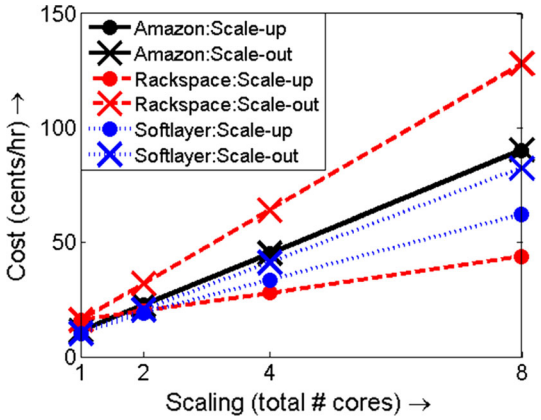

Fig. 2 Cost of scale-up and scale-out in terms of cents/h

linearly extrapolating the cost of a single VM. As such, we do not take into account any discounts that might be possible because of bulk orders. We see that the cost function is exactly the same (linear) for scale-up and scale-out under Amazon EC2's pricing model. We observed the same linear cost function under Microsoft Azure's pricing scheme as well [25] (not shown). In such cases, the decision between scale-up and scale-out depends on the performance impact. For Rackspace [20] and Softlayer [21], we find that the scale-up cost is lower than the scale-out cost for the same total number of cores purchased. Note that this analysis does not take other resources such as memory and storage into account. In choosing the above data, we tried to ensure that the memory and storage specifications were constant across different service providers for each CPU scaling size.

3.3 Provisioning time

In terms of the provisioning time, it takes about 30–40 s for scale-out in our OpenStack environment (via the `boot` command in OpenStack) with pre-configured application tier images. However, if vanilla images are used and application software is installed after boot-up, the provisioning time can easily extend to 4–10 min. For scale-up, the provisioning time is about 20 s. During scale-up (via the `resize` command in OpenStack), the instance is rebooted with the new resource configuration. While this results in application downtime during scale-up, we do not have to reinstall application software as for new scale-out instances.

4 Modeling

The modeling engine lies at the heart of our solution. Our modeling goal is to understand how scale-out and scale-up affect response time.

4.1 Modeling scale-out

We use a queueing-network model [26] to *approximate* our multi-tier cloud workloads. Consider the RUBiS workload described in Sect. 2.1.1. For RUBiS, we have a single front-end VM and a single database VM, but possibly many application tier VMs. We assume perfect load balancing among the homogeneous application tier VMs, thus allowing us to approximate the application tier using a single representative M/G/1/PS [26] queue. If the total workload request rate is λ, then the request rate at this single M/G/1/PS queue is $\frac{\lambda}{k}$, where k is the number of application tier VMs. Under this model, the mean response time, T, is [26]:

$$T = \left(\frac{1}{S_{fe}} - \lambda\right)^{-1} + \left(\frac{1}{S_{app}} - \frac{\lambda}{k}\right)^{-1} + \left(\frac{1}{S_{db}} - \lambda\right)^{-1} \quad (1)$$

where S_{fe}, S_{app}, and S_{db}, are the service times[2] at the front-end, application, and database tiers, respectively. Equation (1) tells us how scale-out affects response time.

4.2 Modeling scale-up

In order to understand the effect of scale-up on response time, we first have to analyze how service time at the application tier, S_{app}, is affected by scale-up. Amdahl's Law [6] suggests that scale-up should decrease a fraction, β, of S_{app} multiplicatively, assuming the workload is parallelizable, for example LINPACK (see Sect. 2.1.2). This β fraction refers to the portion of S_{app} that is parallelizable. In particular, if $S_{app}(x)$ denotes the application tier service time when the application tier VMs have x vCPUs each, then:

$$S_{app}(x) = S_{app}(1) \cdot \left(\frac{\beta}{x} + (1 - \beta)\right) \quad (2)$$

Interestingly, even for sequential workloads that cannot be parallelized, such as RUBiS, we can use a similar approach. Such workloads benefit from increased concurrency due to multiple cores. In this case, while the service time remains the same, the service rate (or peak throughput) of the VM increases. We expect the service rate to be proportional to the number of vCPUs, x. Since service time can be viewed as the inverse of service rate, we can use Eq. (2) to also model the increase in service rate for sequential workloads such as RUBiS. As such, β represents the scale-up efficiency of a workload. Our results in Sect. 4.5 validate our service time/rate modeling for sequential and parallel workloads.

[2] Service time for a tier is defined as the total time taken to serve the workload request at that tier, assuming no resource contention. In other words, it is the minimum execution time at a tier.

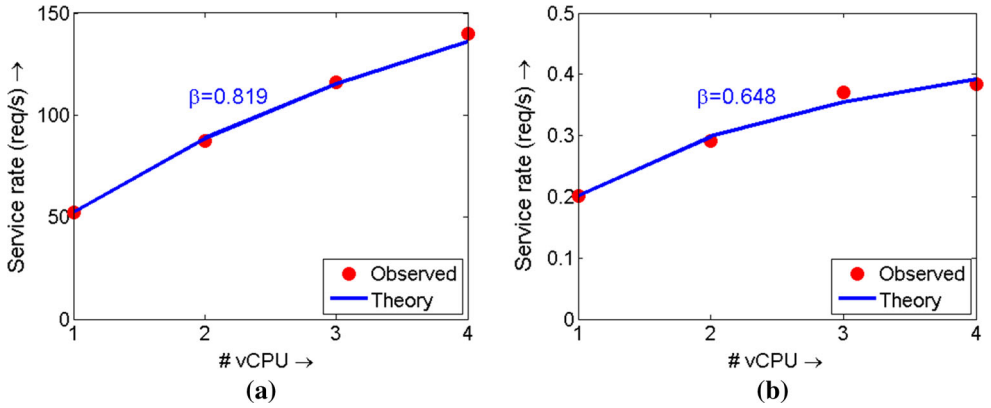

Fig. 3 Modeling service rate scale-up via Amdahl's Law. β represents the scale-up efficiency of the workload. Average modeling error is 1.3% in case of **a** the sequential workload RUBiS and 2.3% in case of **b** the parallel workload LINPACK

4.3 Kalman filtering and robustness

Cloud providers cannot access the user application directly to compute system parameters such as service times. We thus use a Kalman filtering technique to infer these unobservable parameters. Kalman filtering [27] works by leveraging observable monitoring information, such as end-to-end response time and VM utilization (see Sect. 2.2), and using model-based equations, such as Eq. (1), to estimate unobservable parameters. Importantly, by employing the Kalman filter to leverage the *actual* monitored values, we minimize our dependence on the approximate queueing model of the system. Another critical advantage of employing a Kalman filter in cloud performance modeling is the ability to quickly detect and react (by updating the system parameters) to changes in the workload, as evidenced by our previous work [28].

In "Appendix" we provide details on our Kalman filtering technique for parameter estimation of a generic three-tier system abstracted as a queueing-network model.

We employ Kalman filtering to estimate service times (and service rates). In particular, we run experiments with varying request rates for each scale-up setting (with 1 VM only) and process the collected monitored data (see Sect. 2.2) via Kalman filtering to estimate per-tier service times. This profiling step can be avoided if we have prior knowledge of β and $S_{app}(1)$. Figure 3 shows our estimated service times (crosses) based on observed experimental results with scale-up for (a) the sequential workload RUBiS and (b) the parallel workload LINPACK. We use our OpenStack implementation setup described in Sect. 2 for the experiments. We then use regression to fit (shown as solid line) the estimated service times to Eq. (2). This gives us $\beta = 0.819$ and $S_{app}(1) = 21$ ms for RUBiS with a modeling error of 1.3% (parallelism at the HTTP request level, allowing execution of multiple concurrent requests) and $\beta = 0.648$ and $S_{app}(1) = 4.96$ s for LINPACK with a modeling error of 2.3% (parallelism at the thread level, but with a small sequential initialization portion).

4.4 Combined model

Combining Eqs. (1) and (2), we get the final model for response time as a function of workload characteristics (β and $S_{app}(1)$) and scale-up (x) and scale-out (k). In particular, we replace S_{app} in Eq. (1) with $S_{app}(x)$ from Eq. (2) to derive T as a function of x (scale-up) and k (scale-out). We can use this $T(x, k)$ function to determine the maximum request rate that can be handled by a given scaling option without violating the SLA (see Sect. 4.5), which is the largest value of λ in Eq. (1) such that (s.t.) $T(x, k) < SLA$. For a given λ, we can obtain feasible values of x and k such that $T_\lambda(x, k) < SLA$, and further, combine this function with a cost function, such as those in Fig. 2, to determine the most cost-effective scaling option (see Sect. 5).

4.5 Model validation

Figures 4 and 5 show our theoretic and implementation results for (a) scale-out and (b) scale-up for RUBiS and LINPACK, respectively. We use our OpenStack implementation setup described in Sect. 2 for these experiments. We see that our model can predict performance (in terms of maximum load that can be sustained without violating SLA) fairly well for both scale-out and scale-up, including combinations of the two. The average prediction error is about 9% for RUBiS and 7% for LINPACK. In the next section, we employ our model to predict the optimal scaling under various scenarios. Then, in Sect. 6, we implement our modeling engine and experimentally validate our optimal scaling solution.

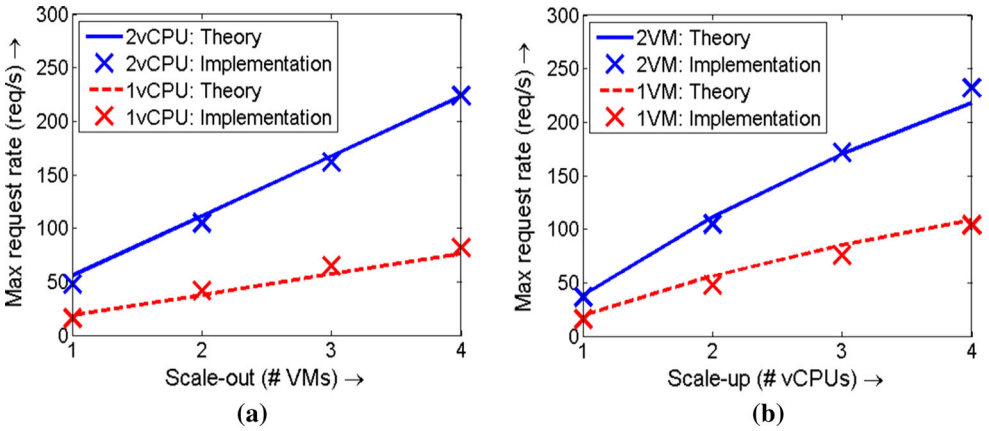

Fig. 4 Good agreement (9% error) between theoretic and implementation results for the sequential workload RUBiS. **a** Scale-out, **b** scale-up

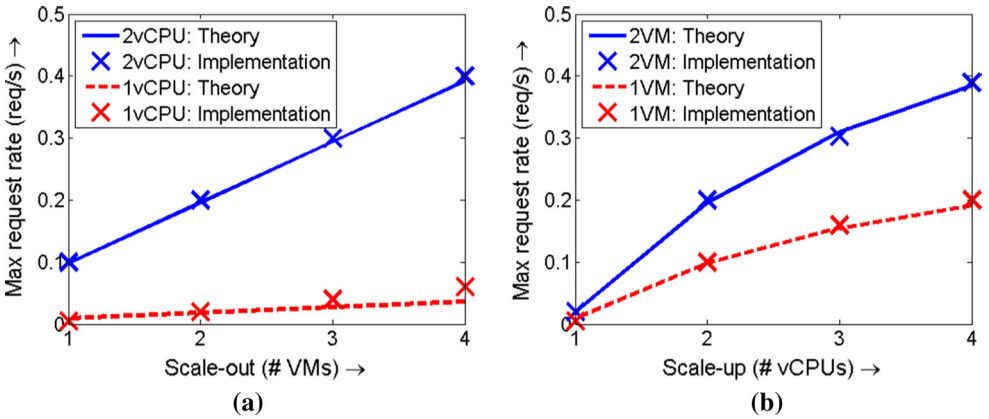

Fig. 5 Good agreement (7% error) between theoretic and implementation results for the parallel workload LINPACK. **a** Scale-out, **b** scale-up

5 Analysis

5.1 Comparison of different scaling options

We now employ our scaling model to examine the various scaling options for different workloads. Figure 6 shows our analytical results for RUBiS in terms of cost as a function of the load under scale-out, scale-up, and optimal, for different response time SLAs. A load of l is defined as the request rate (λ_l) that equals l times the peak throughput of a single 1-vCPU VM (λ_0). Here, optimal refers to the most cost-effective combination of scale-out and scale-up, as determined by our model, that satisfies the SLA. Mathematically, the optimal configuration for load l is a solution of the following optimization problem: $\min_{x,k} c(x,k)$, s.t. $T_{\lambda_l}(x,k) < SLA$, where $\lambda_l = l * \lambda_0$, and $c(x,k)$ is the cost of renting k VMs with x vCPUs each from the cloud provider. For cost, we use the total number of cores employed as opposed to the more volatile metric of cents/h.

We observe that the combination of scale-out and scale-up (optimal) can provide significant cost savings over the individual scaling options. The cost savings increase with load (or request rate) and are higher when the response time SLA is more strict. Further, scale-up is more cost-effective than scale-out when the SLA is more strict and when the load is low. However, scale-out is more cost-effective than scale-up when the load is high.

For the case of 25-ms SLA in Fig. 6a, scale-up is the optimal choice at low request rates and provides significant cost savings over scale-out (about 64% at a load of 2 for 1 5-vCPU VM vs 14 1-vCPU VMs). However, scale-up *cannot* achieve the required response time at a load of above 7 due to the nature of service timescaling explained in Sect. 4. For high loads, scale-out is superior to scale-up, but is still significantly more expensive than optimal (more than 3 times costlier than optimal at a load of 8 for 53 1-vCPU VMs vs 4 4-vCPU VMs). As the response time target becomes less strict in Fig. 6b, c, scale-out starts to approach optimal. The cost savings of optimal over scale-out for response time targets of 35 and 45 ms are about 29% (5 3-vCPU VMs vs 21 1-vCPU VMs) and 25% (6 2-vCPU VMs vs 16 1-vCPU VMs), respectively, at a load of 8.

Figure 7 shows our analytical results for LINPACK in terms of cost as a function of the load under scale-out, scale-

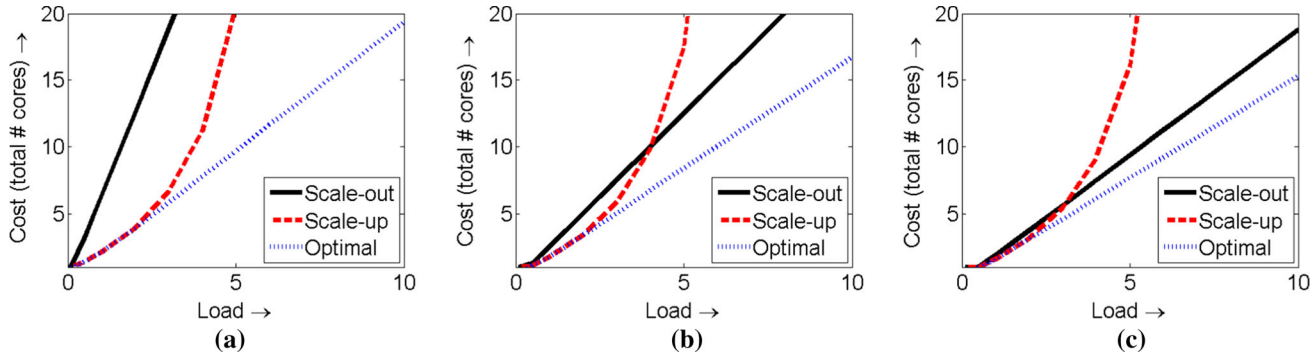

Fig. 6 Analysis of various scaling options for RUBiS as a function of load for various response time SLAs. **a** SLA = 25 ms, **b** SLA = 35 ms, **c** SLA = 45 ms

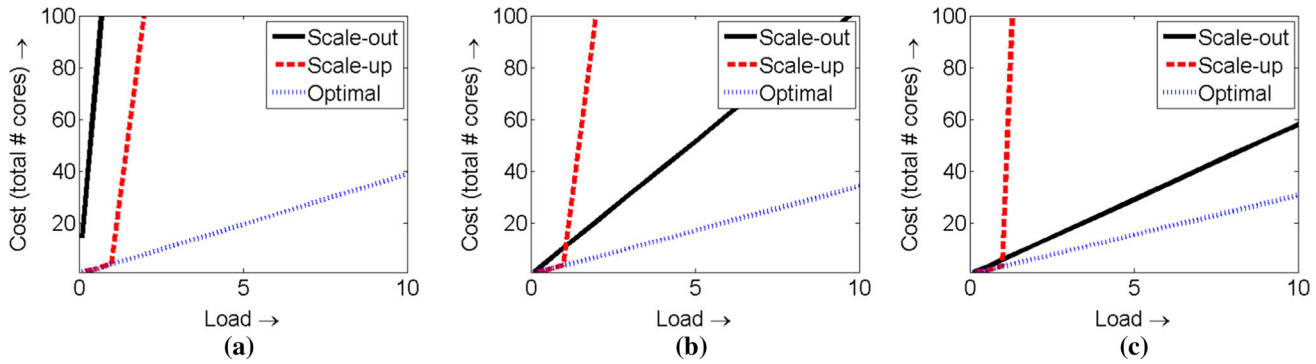

Fig. 7 Analysis of various scaling options for LINPACK as a function of load for various response time SLAs. **a** SLA = 5.0 s, **b** SLA = 5.5 s, **c** SLA = 6.0 s

up, and optimal, for different response time SLAs. We again see that optimal provides significant cost savings over individual scaling options, and the savings increase with load and are greater when the SLA is more strict. For the case of 5-s SLA in Fig. 7a, scale-up is more cost-effective than scale-out. This is to be expected based on our initial experimental results in Fig. 1b. However, for the case of 5.5-s SLA and 6-s SLA in Fig. 7b, c, respectively, we see that scale-out outperforms scale-up. This is because of the low β value (low scale-up efficiency) for LINPACK (see Fig. 3b), which makes scale-up less desirable at high scale. This is to be expected since a high scale-up value implies more OS and lock contention at the VM as opposed to the "shared nothing" scale-out option.

5.2 Analysis of the optimal scaling option

We now further examine the optimal scaling in order to better understand the optimal resource allocation. Figure 8 shows the resource allocation under the optimal scaling policy for different loads as a function of β. Here, we assume the parameters of the RUBiS workload setup except for β (that is, SLA and $S_{app}(1)$). The number of rectangles in each vertical bar represents the number of VMs in that allocation, and the height of each rectangle represents the number of vCPUs. We restrict our analysis to homogeneous VM configurations only.

We see that the optimal scaling prefers larger VMs (scale-up) as β increases. This is to be expected since β represents the scale-up efficiency of the workload. Interestingly, we see that the optimal VM size (number of vCPUs per VM) for a given β does not change with load, except for the case of $\beta = 1$. Thus, the optimal allocation can be viewed as scaling out the optimal-sized VM based on the load. Importantly, we find that the cost (total number of vCPUs) of optimal scaling decreases with β. Thus, it is less expensive to scale workloads that have high scale-up efficiency.

Figure 9 shows the resource allocation under the optimal scaling policy for different loads as a function of β. Here, we assume the parameters of the LINPACK workload setup except for β. We again see that the optimal scaling prefers larger VMs as β increases, and the optimal VM size for a given β appears to be insensitive to load. We also see that the cost of optimal scaling decreases significantly with β.

5.3 Other workloads

The modeling framework presented in Sect. 4 can also be used to model the scale-up of workloads on different architec-

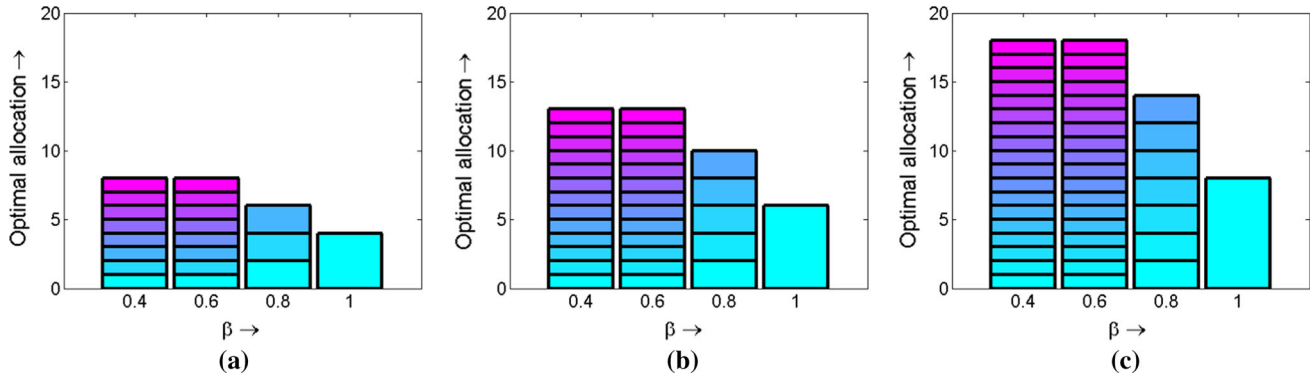

Fig. 8 Optimal allocation as a function of β for various loads assuming the RUBiS workload setup. The *number of rectangles in each vertical bar* represents the number of VMs, and the *height of each rectangle* represents the number of vCPUs. **a** Load = 3, **b** load = 5, **c** load = 7

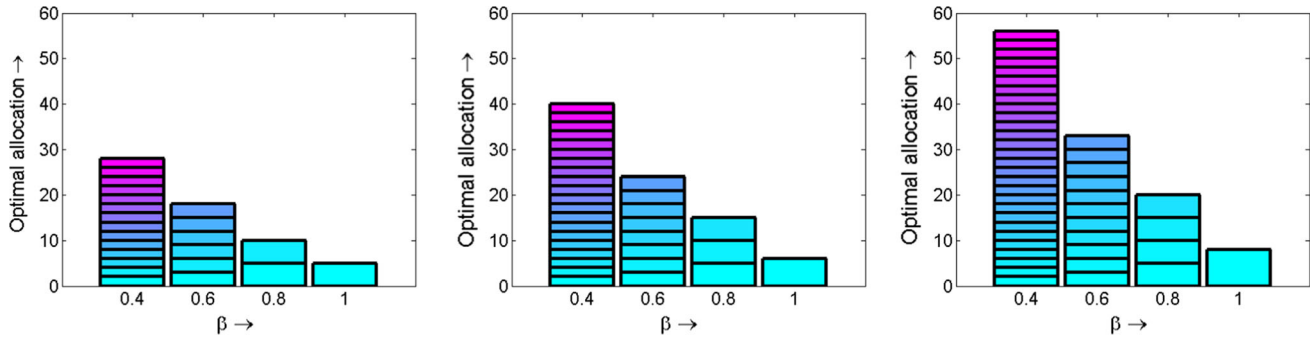

Fig. 9 Optimal allocation as a function of β for various loads assuming the LINPACK workload setup. The *number of rectangles in each vertical bar* represents the number of VMs, and the *height of each rectangle* represents the number of vCPUs. **a** Load = 3, **b** load = 5, **c** load = 7

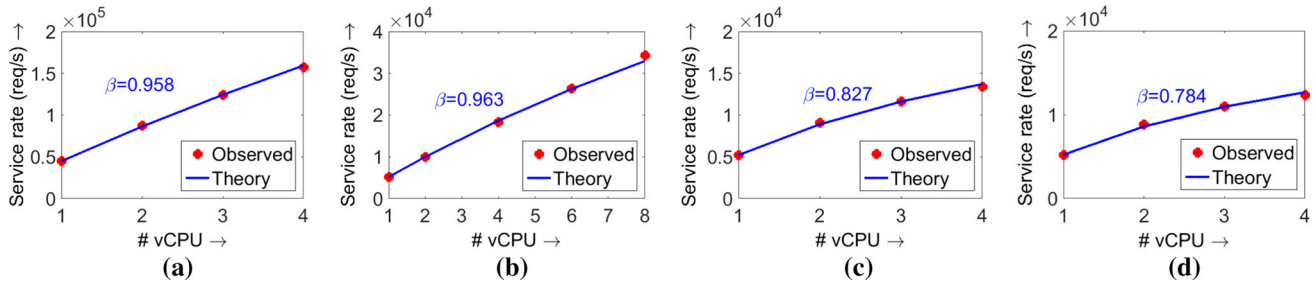

Fig. 10 Modeling service rate scale-up of SPECjbb2005 [16] via Amdahl's Law for the case of **a** multi-core scalability on Nehalem (average error of 0.8%), **b** multi-core scalability on Niagra (average error of 1.5%), **c** SMT multi-thread scalability on Niagra (average error of 1.2%), and **d** SMT multi-process scalability on Niagra (average error of 1.6%)

tures. Figure 10a, b shows our results for scale-up modeling of SPECjbb2005 [16] (see Sect. 2.1.3) on the Nehalem [29] and Niagra [30] architectures, respectively. We obtain the measured data points from [4]. Our results suggest that β does not change much between architectures, at least for this workload.

We can also employ our modeling framework to model the SMT (simultaneous multithreading) scalability of workloads on a single core. Figure 10c, d shows our results for SMT modeling of SPECjbb2005 on the Niagra architecture under the multi-thread and multi-process model, respectively. The low modeling error (1.2% and 1.6%, respectively, for multi-thread and multi-process SMT) highlights the accuracy of our approach for modeling SMT scalability. The SMT scalability (β) is less efficient than core scalability (β), as SMT only represents "virtual" processors. Note that we refer to both core scalability and SMT scalability as β. Further, we see that the multi-thread SMT model results in better scalability than the multi-process SMT model, thus confirming the findings in [4].

We also observed similar results for the MediaWiki [17] workload (see Sect. 2.1.3) in Fig. 11. Interestingly, Medi-

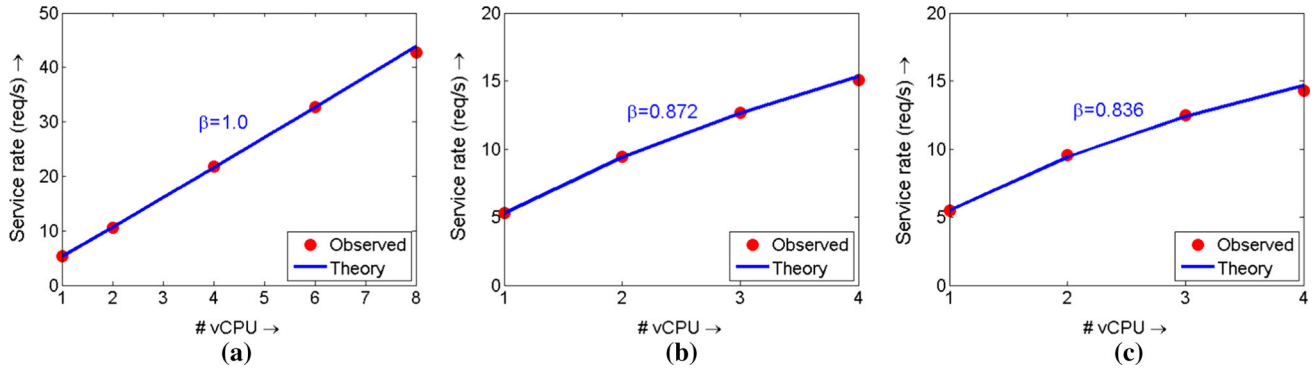

Fig. 11 Modeling service rate scale-up of MediaWiki [17] via Amdahl's Law for the case of **a** multi-core scalability (average error of 0.8%), **b** SMT multi-thread scalability (average error of 0.7%), and **c** SMT multi-process scalability (average error of 1.3%)

aWiki exhibits perfect core scalability ($\beta = 1$) on the Niagra architecture.

We now use our workload characterization of SPECjbb2005 and MediaWiki to analyze the various scaling options. Figures 12 and 13 show our analytical results for SPECjbb2005 and MediaWiki, respectively, on the Niagra multi-core architecture in terms of cost as a function of the load under various scaling options. Given the high scale-up efficiency (high β value) of these workloads on Niagra, we see that scale-up is near optimal for all SLAs and loads. Scale-out also approaches optimal when the SLA is not very strict, similar to our analysis of RUBiS and LINPACK in Figs. 6 and 7, respectively.

6 Experimental evaluation

We now present our implementation and experimental results on OpenStack that validate our performance modeling approach.

6.1 Setup

We employ our setup, described in Sect. 2, to host the RUBiS application, described in Sect. 2.1.1. We set the response time SLA target for the browse requests as 40 ms. We set the response time SLA for all other classes to be 100 ms. The secondary goal is to minimize the resource cost, which is the total number of cores (across all VMs) allocated to the Tomcat application tier. Thus, our objective is to dynamically determine and execute the optimal scaling action, in response to varying workload, to meet the SLA targets while minimizing the resource cost.

We use traces from the WITS traffic archive [31] as well as synthetic traces to drive our load generator. Typically RUBIS load models are static, whereas WITS traffic traces allow us to simulate time-varying workload. To show effectiveness of our dynamic scaling solution in scenarios with real-time workload changes we used WITS traffic traces and synthetic traffic traces. The WITS archive contains a large collection of recent internet traces from ISPs and University networks. We used the Waikato I–VIII trace data sets from within the WITS archive. We scaled the traces to fit our deployment. We determine the rate of issuing web requests for our workloads using arrival rate values from these traces.

Scale-out is executed via OpenStack nova commands (`nova boot`). The provisioning time for a new VM is about 30–40 s. For scale-up, we enable or disable VM cores via the `/sys/devices` interface. While we can resize VMs dynamically via OpenStack nova commands (`nova resize`), the execution delay in resizing the VM instance is significant (20–30 s), which adversely affects application performance and SLA compliance as the VM is unavailable during this time. We thus resort to directly enabling and disabling cores on VMs.

6.2 Scaling logic

We implement our scaling logic solution as a service on top of OpenStack. In order to determine the optimal scaling actions to meet the 40-ms response time target, we employ the combined scale-up and scale-out models (see Sect. 4.4). We obtain application-specific information, such as request rate (λ) and end-to-end response time (T) via the application logs collected at the load generator. These parameters are required for estimating the effect of scale-up and scale-out via Eqs. (1) and (2). Other required parameters, such as service time, are obtained via Kalman filtering, as explained in Sect. 4.3. The modeling engine then leverages this information to determine the optimal scaling action.

In our implementation, both monitoring agent and modeling agent are running on separate VMs. Our monitoring and modeling logic periodically makes scaling decision (once very 10 s) and communicates with OpenStack to execute these decisions (if needed). Since we only do the decisions once every 10 s the overhead is negligible.

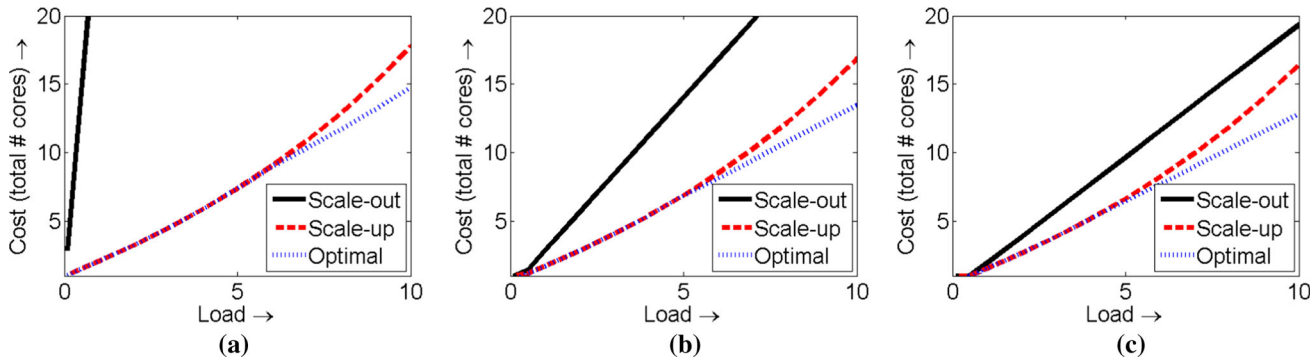

Fig. 12 Analysis of various scaling options for SPECjbb2005 as a function of load for various response time SLAs. **a** SLA = 0.2 ms, **b** SLA = 0.3 ms, **c** SLA = 0.4 ms

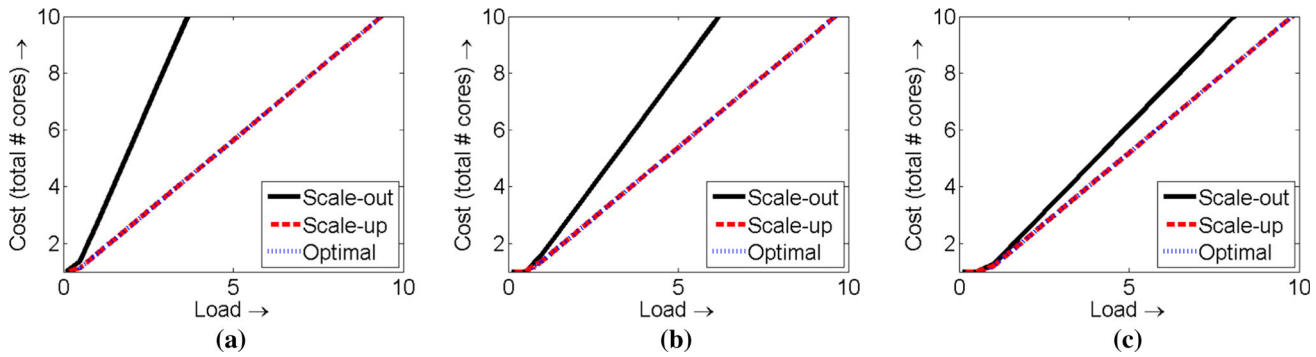

Fig. 13 Analysis of various scaling options for MediaWiki as a function of load for various response time SLAs. **a** SLA = 0.3 s, **b** SLA = 0.5 s, **c** SLA = 1.0 s

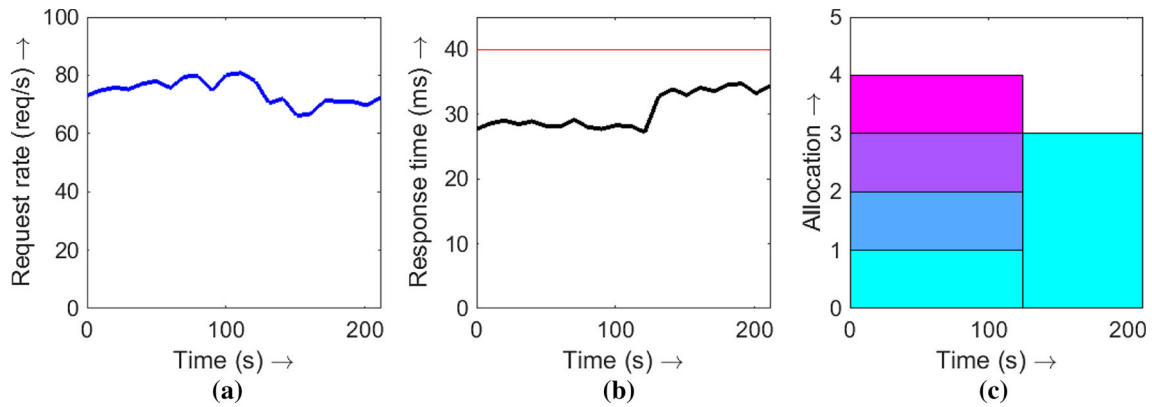

Fig. 14 A simple illustration of our resource scaling in action based on a WITS [31] trace

6.3 Experimental results

We start with a simple illustration of our combined scale-up and scale-out modeling solution. Figure 14 illustrates our scaling logic in action for a trace obtained from the WITS traffic archive. The figure contains three graphs for the observed request rate, the monitored response time (for browse requests), and the resource allocation. The red horizontal line in the response time graph represents our SLA target of 40 ms. For the resource allocation graph, the number of rectangles in each vertical bar represents the number of VMs in that allocation, and the height of each rectangle represents the number of vCPUs. In this experiment, we start with 4 1-vCPU VMs. At around the 120-s mark, the request rate drops sufficiently to invoke a scaling action. Our scaling logic determines the most cost-effective configuration to be 1 3-vCPU VM (as opposed to the 4 1-vCPU VMs) and executes the scaling action accordingly. As seen in the response time graph, we continue to be below the 40-ms target, thus

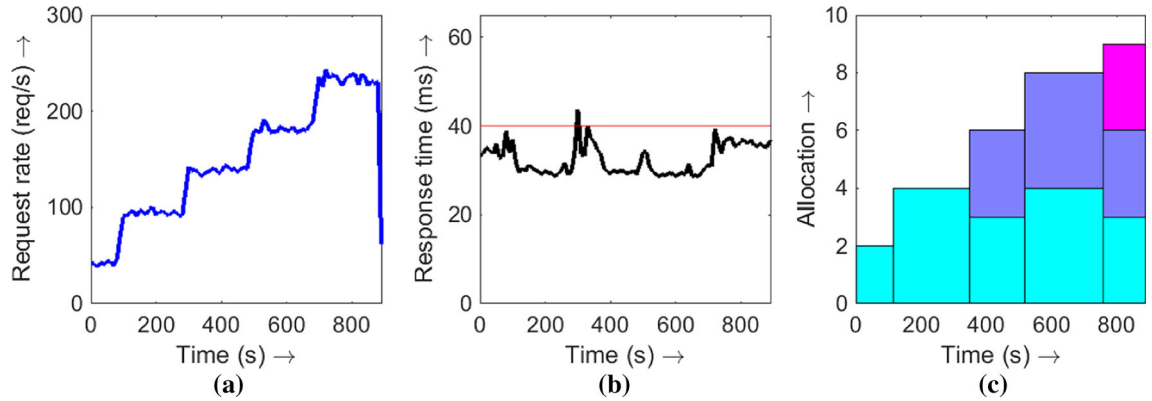

Fig. 15 A more complex scenario requiring multiple scaling actions for a synthetic load trace

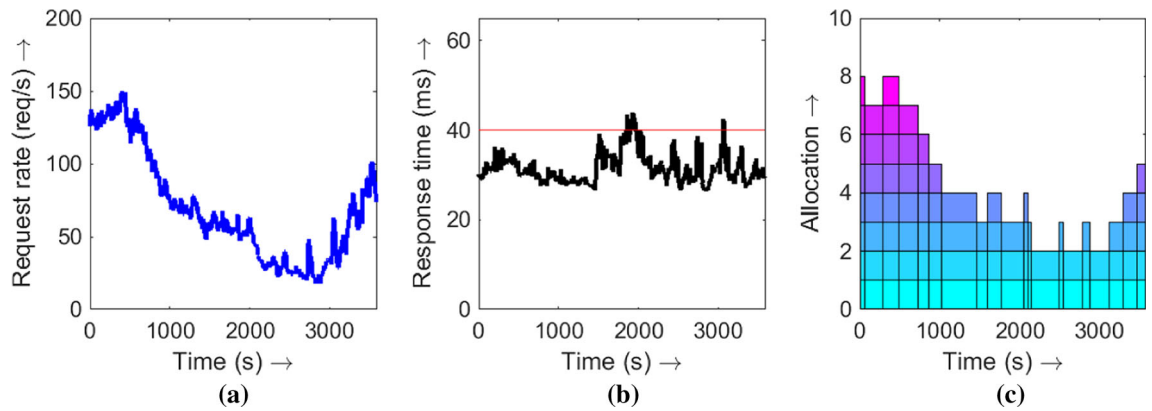

Fig. 16 Pure scale-out using 1-core VMs. Average resource cost is 4.2 cores

saving 25% of the resource cost (4 cores total → 3 cores total).

Figure 15 depicts a more complex scenario where the load increases abruptly every few hundred seconds. For this experiment, we use a synthetic trace. In response to the four abrupt increases, our solution logic takes the required scaling actions to ensure that the response time SLA is met for the increased load. Note that we sometimes momentarily violate the response time SLA; this is due to the bursty (and realistic) load generation tool. Interestingly, the optimal resource allocation involves both scale-up and scale-out. For instance, our first scaling action is a scale-up from 1 2-vCPU VM to 1 4-vCPU VM. Our next scaling action involves scale-down and scale-out. Specifically, we go from a 1 4-vCPU VM configuration to a 2 3-vCPU VMs configuration. This illustrates our ability to leverage both scale-up and scale-out simultaneously.

In order to illustrate the benefits of our combined model, we now compare against a pure scale-out solution logic. Figure 16 shows the optimal scale-out solution employing only 1-vCPU VMs for a 1-hour-long experiment using a WITS [31] trace. We see that scale-out is able to maintain the response time target. The average resource cost over the entire duration of the experiment is 4.2 cores. By contrast, as shown in Fig. 17, our optimal scaling logic achieves the response time compliance by only using 3.3 cores, a reduction of about 21% when compared to the 4.2 cores. We also compared our solution logic against pure scale-out using only 2-vCPU VMs and found that we are still able to reduce resource costs by about 15% (3.3 total cores versus 3.9 total cores). We are also superior compared to the pure scale-up solution that results in numerous SLA violations using 1 VM that can be scaled up to 6 vCPUs, and still consumes 11% more resources (3.3 total cores versus 3.7 total cores). Since cost is proportional to number of cores (refer to Sect. 3.2), core comparison directly translates to cost comparison. These results validate the benefits of the combined scale-up and scale-out approach and, importantly, the efficacy of our modeling engine that accurately predicts the optimal resource configuration.

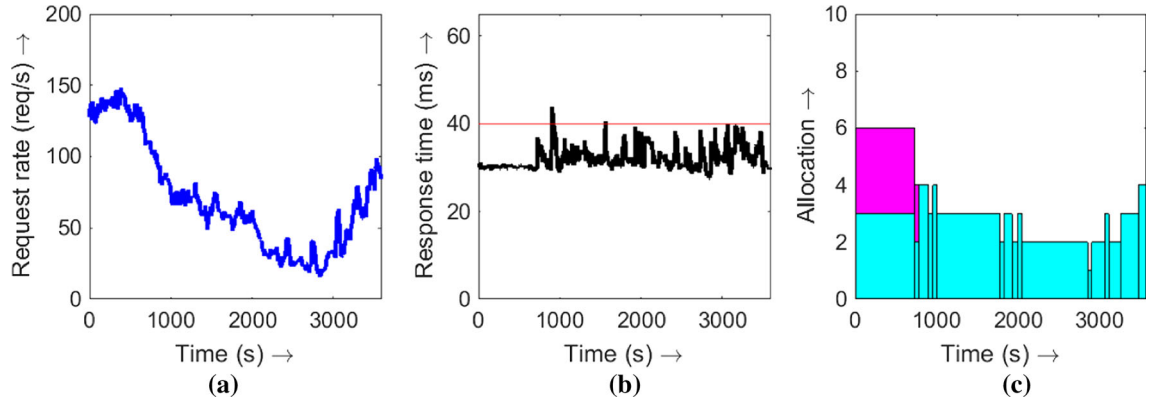

Fig. 17 Our optimal scale-up and scale-out solution. Average resource cost is 3.3 cores

7 Related work

7.1 Cloud autoscaling

Most CSPs offer simple rule-based autoscaling [1,32–34] which typically requires the user to specify the threshold values on the resource usage (e.g., CPU, memory, storage) for triggering scaling actions. Successful autoscaling with such rule-based approaches requires careful tuning of the scaling thresholds. Existing work on allocating resources for data processing jobs, such as ARIA [35], Elastisizer [36], and CRESP [37], focuses on optimal static allocation for capacity planning of future jobs that are not yet deployed. There are a few recent works that address dynamic resource allocation for data processing workloads. MR-Runner [38] enables the sharing of an elastic MapReduce cluster by redistributing resources based on user specifications. Fawkes [39] proposes a sharing mechanism for dynamically balancing the resource allocation across multiple MapReduce clusters. DynMR [40] makes better use of existing resources by interleaving tasks during idle periods. While the above approaches enable dynamic reconfiguration of resources, they do not support SLA-driven resource allocation, thus placing the burden of SLA compliance on the users. By contrast, we specifically focus on SLA-compliant resource autoscaling

7.2 Horizontal and vertical scaling

There has been a lot of work on pure scale-out, or horizontal scaling [14,41,42], and pure scale-up, or vertical scaling [43–45], of applications. There has been much less work on combined scale-out and scale-up. A comparison of scale-out and scale-up architectures is presented in [46,47], where the need for an automated approach for comparison among the two scaling options is advocated. The availability of different scaling options in the cloud makes the choice of scaling difficult, causing practitioners to often make incorrect choices [48]. The advantages and challenges of combined scale-out and scale-up were empirically analyzed in [49–52]. An experimental evaluation of scale-out versus scale-up architectures for commercial workloads was carried out in [53–55] where it was concluded that scale-out offers better cost-performance trade-off compared to scale-up. However, some recent studies [45,56] argue that scale-up can achieve comparable, and sometimes, even better performance than scale-out for certain workloads. All of the above-cited works focus solely on experimental research for evaluating the trade-offs between scale-out and scale-up. To the best of our knowledge, our work is the first to propose and validate an analytical model for evaluating different scaling options.

7.3 Workload scalability studies

Workload scalability is affected by different factors associated with the infrastructure configuration and the workload characteristics. Different workloads exhibit different levels and patterns of interaction with the infrastructure. In the past, there have been a lot of studies on the scalability of workloads on different platforms in a noncloud setting [3–5,57–59]. Majority of this work [3,4,57,60] has focused on identifying the effects of micro-architectural features on workload scalability such as the number of cores, sockets, threads per core, L1 and L2 cache sizes, and multi-processor architectures (SMP, CMP). The authors in [57] analyzed JVM scalability on symmetrical multiprocessing (SMP) systems with number of processors and application threads. The effects of memory system latency and resource contentions on JVM performance were studied. In [3], the scalability of a collection of commercial workloads, including SAP-SD, IBM Trade, Specjbb2005, SPEC SDET, DBench, and TBench, on a multi-core system, SUN Fire T2000, was reported. The applications mostly exhibit a close-to-linear scaling with number of cores and about 50% to 70% scaling efficiency with number of hardware threads per core. The interaction between parallelization models (multi-process and multi-

thread) of web servers and the hardware threads was studied empirically for a multi-core SMT processor (Sun Niagara) in [4] to characterize scalability of Java and PHP workloads. We use the results of this work as an example data point in our analysis.

The software stack (application/middleware/OS) is another dimension that affects workload scalability [5,58,59]. In [58], the authors studied scalability of three server-side Web applications on a chip multiprocessor by reducing lock contention. [59] studied emerging Web 2.0 workloads from applications like Mashups, Wikis, Blogs, and the specific scalability issues resulting from participatory nature of these workloads on multi-core architectures (IBM POWER6 and SUN Niagara 2). Scalability analysis of enterprise-grade Java workloads on multi-core system (IBM POWER7) was done in [5], where a top-down approach for identifying performance bottlenecks across different layers of the software stack was proposed.

While these studies were conducted on noncloud infrastructures, their findings are equally relevant when studying workload performance in the cloud and should be leveraged when developing optimal scaling solutions. Our work proposes a simple model for workload-specific performance scaling in the cloud which can be enriched by the findings of the above studies to develop a more comprehensive model incorporating the effects of other hardware and/or software features, apart from number of cores, when studying resource scaling in the cloud.

8 Conclusion and future work

In this paper, we present an analytical model to understand the impact of workload characteristics on the efficacy of scale-out and scale-up in the cloud. We examine the core scalability of sequential and parallel workloads, and model its effects on performance scaling. Our model provides great accuracy when compared with implementation results and is very useful in determining the optimal scaling for any given workload as a function of workload demand and required SLA. We demonstrate the efficacy of our model via implementation on OpenStack.

Our results indicate that combining scale-up and scale-out can provide significant cost savings over pure scale-up or pure scale-out. Further, we find that scale-up is typically superior to scale-out when the SLA is strict, whereas scale-out is typically superior to scale-up when the load is high. Importantly, we find that the relative ordering of the different scaling options depends critically on the workload characteristics. For workloads that have high scale-up efficiency, vertical scaling is near optimal for all SLAs and loads. Further, the total amount of resources required by a workload for meeting its SLA decreases with an increase in the scale-up efficiency.

In our experiments with RUBiS, there are multiple request types concurrently running on the system. For simplicity, we only focus on the performance of a single request type (browse) and restrict our application performance model in Sect. 4 to a single request-type system. However, it is possible to extend our performance modeling to workloads with multiple request types, as in our previous work [28]. In our analysis, we did not take into account cost savings due to bulk orders. This will result in a nonlinear cost function which will change the results of Fig. 2. However, our performance model is independent of the specific form of cost function. Since we do not have access to bulk discount pricing from providers, we could not undertake that study.

Our performance metric in this paper is mean response time. We can extend our approach to capture other metrics as well, such as higher percentiles of response time, or combinations of metrics such as response time and latency. However, we do require the performance metric(s) to be expressed in terms of system variables, as in Eq. (1). An alternative approach would be to model the system as a black box and infer the relationship between performance and observable system parameters. Such a black box modeling approach, however, does have its limitations. For example, a 100% CPU usage is not surprising for a batch workload, such as Hadoop, but is alarming, and indicates the need for scaling in the case of transactional workloads, such as RUBiS.

For the workloads in this paper, we only focused on scaling compute resources. In the future, we will consider workloads, such as Hadoop, that require the scaling of multiple resources including compute, I/O, and network.

Our modeling efforts represent the first step toward the development of an automated optimal scaling solution for cloud applications. While in this paper we are focussed on IaaS, our approach can easily be extended to SaaS and PaaS environments. In such environments the application topology is easily available which makes its easy to employ our Kalman filtering-based autoscaling solution. As part of future work, we will leverage our model to build an online scaling service for cloud applications that will execute the most cost-effective scale-out and scale-up actions in response to varying workload demand while taking the workload characteristics into account.

Appendix

Figure 18 shows a queueing-network model of a general three-tier system with each tier representing a collection of homogeneous servers. We assume that the load at each tier is distributed uniformly across all the servers in that tier. The system is driven by a workload consisting of i distinct request classes, each class being characterized by its arrival rate, λ_i, and end-to-end response time, R_i. Let n_j be the number of

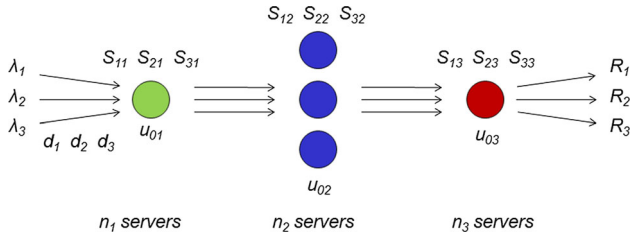

Fig. 18 Queueing model for our system. The system parameters are: λ_i, arrival rate of class i; R_i, response time for class i; d_i, network latency for class i; u_{0j}, background utilization for tier j; S_{ij}: service time of class i at tier j

servers at tier j. With homogeneous servers and perfect load balancing, the arrival rate of requests at any server in tier j is $\lambda_{ij} := \lambda_i / n_j$. Since servers at a tier are identical, for ease of analysis, we model each tier as a single representative server. With some abuse of terminology, we refer to the representative server at tier j as tier j. Let $u_j \in [0, 1)$ be the utilization of tier j. The background utilization of tier j is denoted by u_{0j} and models the resource utilization due to other jobs (not related to our workload) running on that tier. The end-to-end network latency for a class i request is denoted by d_i. Let $S_{ij}(\geq 0)$ denote the average service time of a class i request at tier j. Assuming we have Poisson's arrivals and a processor-sharing policy at each server, the stationary distribution of the queueing network is known to have a product form [26], for any general distribution of service time at servers. Under the product-form assumption, we have following analytical results from queueing theory:

$$u_j = u_{0j} + \sum_i \lambda_{ij} S_{ij}, \quad \forall j \tag{3}$$

$$R_i = d_i + \sum_j \frac{S_{ij}}{1 - u_j}, \quad \forall i \tag{4}$$

While u_j, R_i, and λ_i $\forall i, j$ can be monitored easily and are thus observable, the parameters S_{ij}, u_{0j}, and d_i are nontrivial to measure and are thus unobservable.

We employ a parameter estimation technique, Kalman filtering, to derive estimates for the unobservable parameters. Further, since the system parameters can dynamically change during runtime, we employ the Kalman filter as an online parameter estimator to continually adapt our parameter estimates. It is important to note that while the product form is shown to be a reasonable assumption for tiered web services [61], we only use it as an approximation for our complex system. By employing the Kalman filter to leverage the actual monitored values, we minimize our dependence on the approximation.

For a three-class, three-tier system (i.e., $i = j = 3$), let $\mathbf{z} := (u_1, u_2, u_3, R_1, R_2, R_3)^T = \mathbf{h}(\mathbf{x})$ and $\mathbf{x} = (u_{01}, u_{02}, u_{03}, d_1, d_2, d_3, S_{11}, S_{21}, S_{31}, S_{12}, S_{22}, S_{32}, S_{13}, S_{23}, S_{33})^T$.

Note that $\mathbf{z}$ is a 6-dimensional vector, whereas $\mathbf{x}$ is a 15-dimensional vector. The problem is to determine the unobservable parameters $\mathbf{x}$ from measured values of $\mathbf{z}$ and $\lambda = (\lambda_1, \lambda_2, \lambda_3)$.

The dynamic evolution of system parameters can be described through the following Kalman filtering equations [27]:

$$\begin{aligned} \text{System State } \mathbf{x}(t) &= \mathbf{F}(t)\mathbf{x}(t-1) + \mathbf{w}(t), \\ \text{Measurement Model } \mathbf{z}(t) &= \mathbf{H}(t)\mathbf{x}(t) + \mathbf{v}(t), \end{aligned}$$

where $\mathbf{F}(t)$ is the state transition model and $\mathbf{H}(t)$ is the observation model mapping the true state space into observed state space. In our case, $\mathbf{F}(t), \forall t$ is the identity matrix. The variables $\mathbf{w}(t) \sim \mathcal{N}(0, \mathcal{Q}(t))$ and $\mathbf{v}(t) \sim \mathcal{N}(0, \mathcal{R}(t))$ are process noise and measurement noise which are assumed to be zero-mean, multi-variate normal distributions with covariance matrices $\mathcal{Q}(t)$ and $\mathcal{R}(t)$, respectively. The matrices $\mathcal{Q}(t)$ and $\mathcal{R}(t)$ are not directly measurable but can be tuned via best practices [62].

Since the measurement model $\mathbf{z}$ is a nonlinear function of the system state $\mathbf{x}$ [see Eqs. (3) and (4)], we use the extended Kalman filer [27] with $\mathbf{H}(t) = \left[\frac{\partial \mathbf{h}}{\partial \mathbf{x}}\right]_{\mathbf{x}(t)}$ which for our model is a 6×15 matrix with $\mathbf{H}(t)_{ij} = \left[\frac{\partial \mathbf{h_i}}{\partial \mathbf{x_j}}\right]_{\mathbf{x}(t)}$. Since $\mathbf{x}(t)$ is not known at time t, we estimate it by $\hat{\mathbf{x}}(t|t-1)$, which is the *a priori* estimate of $x(t)$ given all the history up to time $t-1$. The state of the filter is described by two variables $\hat{\mathbf{x}}(t|t)$ and $\mathbf{P}(t|t)$, where $\hat{\mathbf{x}}(t|t)$ is the *a posteriori* estimate of state at time t and $\mathbf{P}(t|t)$ is the *a posteriori* error covariance matrix which is a measure of estimated accuracy of the state estimate.

The Kalman filter has two phases: predict and update. In the predict phase *a priori* estimates of state and error matrix are calculated, and in the update phase, these estimates are refined using the current observation to get *a posteriori* estimates of state and error matrix. The filter model for the predict and update phase for our 3-class, 3-tier model is given by:

Predict:

$$\begin{aligned} \hat{\mathbf{x}}(t|t-1) &= \mathbf{F}(t)\hat{\mathbf{x}}(t-1|t-1) \\ \mathbf{P}(t|t-1) &= \mathbf{F}(t)\mathbf{P}(t-1|t-1)\mathbf{F}^T(t) + \mathcal{Q}(t) \end{aligned}$$

Update:

$$\begin{aligned} \mathbf{y}(t) &= \mathbf{z}(t) - \mathbf{h}(\hat{\mathbf{x}}(t|t-1)) \\ \mathbf{H}(t) &= \left[\frac{\partial \mathbf{h}}{\partial \mathbf{x}}\right]_{\hat{\mathbf{x}}(t|t-1)} \\ \mathbf{S}(t) &= \mathbf{H}(t)\mathbf{P}(t|t-1)\mathbf{H}^T(t) + \mathcal{R}(t) \\ \mathbf{K}(t) &= \mathbf{P}(t|t-1)\mathbf{H}^T(t)\mathbf{S}^{-1}(t) \end{aligned}$$

$$\hat{\mathbf{x}}(t|t) = \hat{\mathbf{x}}(t|t-1) + \mathbf{K}(t)\mathbf{y}(t)$$
$$\mathbf{P}(t|t) = (\mathbf{I} - \mathbf{K}(t)\mathbf{H}(t))\mathbf{P}(t|t-1)$$

We employ the above filter model by seeding our initial estimate of $\hat{\mathbf{x}}(t|t-1)$ and $\mathbf{P}(t|t-1)$ with random values, then applying the update equations by monitoring $\mathbf{z}(t)$ to get $\hat{\mathbf{x}}(t|t)$ and $\mathbf{P}(t|t)$, and finally using the predict values to arrive at the estimated $\hat{\mathbf{x}}(t|t-1)$ and $\mathbf{P}(t|t-1)$. We continue this process iteratively at each 10-s monitoring interval to derive new estimates of the system state.

References

1. Amazon Inc.: Amazon Auto Scaling. http://aws.amazon.com/autoscaling
2. SoftLayer Technologies, Inc., http://www.softlayer.com
3. Tseng, J.H., Yu, H., Nagar, S., Dubey, N., Franke, H., Pattnaik, P., Inoue, H., Nakatani, T.: Performance studies of commercial workloads on a multi-core system. In: Proceedings of the 2007 IEEE International Symposium on Workload Characterization, Boston, MA, USA, pp. 57–65 (2007)
4. Inoue, H., Nakatani, T.: Performance of multi-process and multi-thread processing on multi-core SMT processors. In: Proceedings of the 2010 IEEE International Symposium on Workload Characterization, Atlanta, GA, USA, pp. 209–218 (2010)
5. Guerin, X., Tan, W., Liu, Y., Seelam, S., Dube, P.: Evaluation of multi-core scalability bottlenecks in enterprise Java workloads. In: Proceedings of the 20th International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems, Arlington, VA, USA, pp. 308–317 (2012)
6. Hill, M.D., Marty, M.R.: Amdahl's law in the multicore era. Computer **41**(7), 33–38 (2008)
7. Moreira, J.E., Michael, M.M., Da Silva, D., Shiloach, D., Dube, P., Zhang, L.: Scalability of the nutch search engine. In: Proceedings of the 21st Annual International Conference on Supercomputing, Seattle, WA, USA, pp. 3–12 (2007)
8. Williams, S., Waterman, A., Patterson, D.: Roofline: an insightful visual performance model for multicore architectures. Commun. ACM **52**(4), 65–76 (2009)
9. Openstack.org: OpenStack Open Source Cloud Computing Software. http://www.openstack.org
10. Opscode Inc.: Chef. http://www.opscode.com/chef
11. RUBiS: Rice University Bidding System. http://rubis.ow2.org
12. Intel Corp.: Intel Math Kernel Library - LINPACK 11.1 Update 2. https://software.intel.com/en-us/articles/intel-math-kernel-library-linpack-download
13. Bouchenak, S., Cox, A., Dropsho, S., Mittal, S., Zwaenepoel, W.: Caching dynamic web content: designing and analysing an aspect-oriented solution. In: Middleware 2006, (2006)
14. Urgaonkar, B., Pacifici, G., Shenoy, P., Spreitzer, M., Tantawi, A.: An analytical model for multi-tier internet services and its applications. In: Proceedings of the 2005 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems, Banff, Alberta, Canada, pp. 291–302 (2005)
15. Mosberger, D., Jin, T.: httperf—a tool for measuring web server performance. SIGMETRICS Perform. Eval. Rev. **26**(3), 31–37 (1998)
16. Standard Performance Evaluation Corporation: SPECjbb2005. http://www.spec.org/jbb2005
17. Wikimedia Foundation: MediaWiki. http://www.mediawiki.org
18. libvirt virtualization API. http://libvirt.org
19. Amazon Web Services, Inc.: Amazon EC2 Pricing. http://aws.amazon.com/ec2/pricing
20. Rackspace, US Inc.: Cloud Servers Pricing—Rackspace Hosting. http://www.rackspace.com/cloud/servers/pricing
21. SoftLayer Technologies, Inc.: Build Your Own Cloud Server. http://www.softlayer.com/cloudlayer/build-your-own-cloud
22. Le Sueur, E., Heiser, G.: Dynamic voltage and frequency scaling: The laws of diminishing returns. In: Proceedings of the 2010 International Conference on Power Aware Computing and Systems, ser. HotPower'10, pp. 1–8 (2010)
23. VMware: VMware vCenter Server. http://www.vmware.com/products/vcenter-server
24. Why has CPU frequency ceased to grow? . https://software.intel.com/en-us/blogs/2014/02/19/why-has-cpu-frequency-ceased-to-grow (2014)
25. Microsoft, Inc.: Pricing Calculator | Windows Azure. http://www.windowsazure.com/en-us/pricing/calculator/?scenario=virtual-machines
26. Walrand, J.: An Introduction to Queueing Networks. Prentice Hall, Upper Saddle River (1988)
27. Simon, D.: Optimal State Estimation: Kalman, H Infinity, and Nonlinear Approaches. Wiley, New York (2006)
28. Gandhi, A., Dube, P., Karve, A., Kochut, A., Zhang, L.: Adaptive, model-driven autoscaling for cloud applications. In: Proceedings of the 11th International Conference on Autonomic Computing, Philadelphia, PA, USA (2014)
29. Singhal, R.: Inside Intel Next Generation Nehalem Microarchitecture. Intel Developer Forum, San Francisco (2008)
30. Kongetira, P., Aingaran, K., Olukotun, K.: Niagara: a 32-way multithreaded Sparc processor. IEEE Micro **25**(2), 21–29 (2005)
31. WAND Network Research Group: WITS: Waikato Internet Traffic Storage. http://www.wand.net.nz/wits/index.php
32. Google Cloud Platform: Auto Scaling on the Google Cloud Platform. http://cloud.google.com/resources/articles/auto-scaling-on-the-google-cloud-platform
33. WindowsAzure: How to Scale an Application. http://www.windowsazure.com/en-us/manage/services/cloud-services/how-to-scale-a-cloud-service
34. VMware, Inc.: VMware vFabric AppInsight. http://pubs.vmware.com/appinsight-5/index.jsp
35. Verma, A., Cherkasova, L., Campbell, R.H.: ARIA: automatic resource inference and allocation for mapreduce environments. In: Proceedings of the 8th ACM International Conference on Autonomic Computing, ser. ICAC '11, Karlsruhe, Germany, pp. 235–244 (2011)
36. Herodotou, H., Lim, H., Luo, G., Borisov, N., Dong, L., Cetin, F.B., Babu, S.: Starfish: a self-tuning system for big data analytics. In: Proceedings of the 5th Biennial Conference on Innovative Data Systems Research, ser. CIDR '11, Asilomar, CA, USA, pp. 261–272 (2011)
37. Chen, K., Powers, J., Guo, S., Tian, F.: Cresp: towards optimal resource provisioning for mapreduce computing in public clouds. Parallel Distrib. Syst. IEEE Trans. **25**(6), 1403–1412 (2014)
38. Ghit, B., Yigitbasi, N., Epema, D.: Resource management for dynamic mapreduce clusters in multicluster systems. In: Proceedings of the 2012 SC Companion: High Performance Computing, Networking Storage and Analysis, ser. SCC '12. Washington, DC, USA. IEEE Computer Society, pp. 1252–1259 (2012)
39. Ghit, B., Yigitbasi, N., Iosup, A., Epema, D.: Balanced resource allocations across multiple dynamic mapreduce clusters. SIGMETRICS Perform. Eval. Rev. **42**(1), 329–341 (2014)
40. Tan, J., Chin, A., Hu, Z.Z., Hu, Y., Meng, S., Meng, X., Zhang, L.: Dynmr: dynamic mapreduce with reducetask interleaving and maptask backfilling. In: Proceedings of the Ninth European Conference on Computer Systems, ser. EuroSys '14. New York, NY, USA: ACM, pp. 2:1–2:14 (2014)

41. Gandhi, A., Harchol-Balter, M., Raghunathan, R., Kozuch, M.: AutoScale: dynamic, robust capacity management for multi-tier data centers. Trans. Comput. Syst. **30**, 14 (2012)
42. Krioukov, A., Mohan, P., Alspaugh, S., Keys, L., Culler, D., Katz, R.: NapSAC: design and implementation of a power-proportional web cluster. In: Proceedings of the 1st ACM SIGCOMM Workshop on Green Networking, New Delhi, India, pp. 15–22 (2010)
43. GmbH, ProfitBricks: Live Vertical Scaling. Technical Report, PROFITBRICKS IAAS (2012)
44. Kalyvianaki, E., Charalambous, T., Hand, S.: Self-adaptive and self-configured CPU resource provisioning for virtualized servers using Kalman filters. In: Proceedings of the 6th International Conference on Autonomic Computing, Barcelona, Spain, pp. 117–126 (2009)
45. Rowstron, A., Narayanan, D., Donnelly, A., O'Shea, G., Douglas, A.: Nobody ever got fired for using Hadoop on a cluster. In: Proceedings of the 1st International Workshop on Hot Topics in Cloud Data Processing, Bern, Switzerland, pp. 2:1–2:5 (2012)
46. Gigaspaces Resource Center: Scale Up vs. Scale Out. http://www.gigaspaces.com/WhitePapers (2011)
47. Sevilla, M., Nassi, I., Ioannidou, K., Brandt, S., Maltzahn, C.: A framework for an in-depth comparison of scale-up and scale-out. In: Proceedings of the 2013 International Workshop on Data-Intensive Scalable Computing Systems, Denver, CO, USA, pp. 13–18 (2013)
48. Schwarzkopf, M., Murray, D.G., Hand, S.: The seven deadly sins of cloud computing research. In: Proceedings of the 4th USENIX Conference on Hot Topics in Cloud Computing, Boston, MA, USA (2012)
49. Iqbal, W., Dailey, M.N., Carrera, D.: SLA-driven dynamic resource management for multi-tier web applications in a cloud. In: Proceedings of the 10th International Symposium on Cluster, Cloud and Grid Computing, Melbourne, Victoria, Australia, pp. 832–837 (2010)
50. Sedaghat, M., Hernandez-Rodriguez, F., Elmroth, E.: A virtual machine re-packing approach to the horizontal vs. vertical elasticity trade-off for cloud autoscaling. In: Proceedings of the 2013 ACM Cloud and Autonomic Computing Conference, Miami, FL, USA, pp. 6:1–6:10 (2013)
51. Bonvin, N., Papaioannou, T., Aberer, K.: Autonomic SLA-driven provisioning for cloud applications. In: Proceedings of the 11th International Symposium on Cluster, Cloud and Grid Computing, Newport Beach, CA, USA, pp. 434–443 (2011)
52. Vaquero, L.M., Rodero-Merino, L., Buyya, R.: Dynamically scaling applications in the cloud. SIGCOMM Comput. Commun. Rev. **41**(1), 45–52 (2011)
53. Michael, M., Moreira, J., Shiloach, D., Wisniewski, R.: Scale-up x scale-out: a case study using Nutch/Lucene. In: Proceedings of the 2007 International Parallel and Distributed Processing Symposium, Long Beach, CA, USA, pp. 1–8 (2007)
54. Yu, H., Moreira, J., Dube, P., Chung, I.-H., Zhang, L.: Performance studies of a WebSphere application, trade, in scale-out and scale-up environments. In: Proceedings of the 2007 International Parallel and Distributed Processing Symposium, Long Beach, CA, USA, pp. 1–8 (2007)
55. Brebner, P., Gosper, J.: How scalable is J2EE technology? SIGSOFT Softw. Eng. Notes **28**(3), 4–4 (2003)
56. Appuswamy, R., Gkantsidis, C., Narayanan, D., Hodson, O., Rowstron, A.: Scale-up vs scale-out for Hadoop: time to rethink? In: Proceedings of the 4th annual symposium on cloud computing, Santa Clara, CA, USA, pp. 20:1–20:13 (2013)
57. Cao, Z., Huang, W., Chang, J.M.: A study of Java virtual machine scalability issues on SMP systems. In: Proceedings of the 2005 IEEE International Symposium on Workload Characterization, Austin, TX, USA, pp. 119–128 (2005)
58. Ishizaki, K., Nakatani, T., Daijavad, S.: Analyzing and improving performance scalability of commercial server workloads on a chip multiprocessor. In: Proceedings of the 2009 IEEE International Symposium on Workload Characterization, Austin, TX, USA, pp. 217–226 (2009)
59. Ohara, M., Nagpurkar, P., Ueda, Y., Ishizaki, K.: The data-centricity of Web 2.0 workloads and its impact on server performance. In: Proceedings of the 2009 International Symposium on Performance Analysis of Systems and Software, Boston, MA, USA, pp. 133–142 (2009)
60. Iyer, R., Bhat, M., Zhao, L., Illikkal, R., Makineni, S., Jones, M., Shiv, K., Newell, D.: Exploring small-scale and large-scale CMP architectures for commercial Java servers. In: Proceedings of the 2006 IEEE International Symposium on Workload Characterization, San Jose, CA, USA, pp. 191–200 (2006)
61. Dube, P., Yu, H., Zhang, L., Moreira, J.: Performance evaluation of a commercial application, trade, in scale-out environments. In: Proceedings of the 15th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems, Istanbul, Turkey, pp. 252–259 (2007)
62. Kumar, D., Tantawi, A., Zhang, L.: Estimating model parameters of adaptive software systems in real-time. In: Ardagna, D., Zhang, L. (eds.) Run-Time Models for Self-managing Systems and Applications, ser. Autonomic Systems. Springer Basel, pp. 45–71 (2010). doi:10.1007/978-3-0346-0433-8_3

Anshul Gandhi is an Assistant Professor in the Department of Computer Science at Stony Brook University. He obtained his Ph.D. (2013) from Carnegie Mellon University and spent a year as a postdoc at the IBM T.J. Watson Research Center. His current research interests are in Performance Modeling and Cloud Computing.

Parijat Dube received his Ph.D. (2002) in Computer Science from INRIA, France. Dr. Dube is a Research Staff Member at the IBM T.J. Watson Research Center. His research interests are in performance modeling, analysis, and optimization of systems.

Alexei Karve is a Senior Software Engineer at the IBM T.J. Watson Research Center currently focussing on extensions to OpenStack Compute and Glance and deployment on SoftLayer. He is an IBM Redbooks thought leader for cloud. He holds several patents and has over 22 years of varied experience with application technology solutions, services, systems software, and firmware.

Andrzej Kochut received his Ph.D. (2005) in Computer Science from the University of Maryland, College Park. Dr. Kochut is a Research Staff Member and Manager at the IBM T.J. Watson Research Center where his research interests are in computer systems, networking, stochastic modeling, and performance evaluation. He has published extensively in top conferences, and his work has been recognized wit two Best Paper Awards.

Li Zhang received his Ph.D. degree in Operations Research from Columbia University. He is the manager of the System Analysis and Optimization group at the IBM T.J. Watson Research Center. He has coauthored more than 100 technical articles and more than 30 patents.

Terms and Conditions

Springer Nature journal content, brought to you courtesy of Springer Nature Customer Service Center GmbH (“Springer Nature”).
Springer Nature supports a reasonable amount of sharing of research papers by authors, subscribers and authorised users (“Users”), for small-scale personal, non-commercial use provided that all copyright, trade and service marks and other proprietary notices are maintained. By accessing, sharing, receiving or otherwise using the Springer Nature journal content you agree to these terms of use (“Terms”). For these purposes, Springer Nature considers academic use (by researchers and students) to be non-commercial.
These Terms are supplementary and will apply in addition to any applicable website terms and conditions, a relevant site licence or a personal subscription. These Terms will prevail over any conflict or ambiguity with regards to the relevant terms, a site licence or a personal subscription (to the extent of the conflict or ambiguity only). For Creative Commons-licensed articles, the terms of the Creative Commons license used will apply.
We collect and use personal data to provide access to the Springer Nature journal content. We may also use these personal data internally within ResearchGate and Springer Nature and as agreed share it, in an anonymised way, for purposes of tracking, analysis and reporting. We will not otherwise disclose your personal data outside the ResearchGate or the Springer Nature group of companies unless we have your permission as detailed in the Privacy Policy.
While Users may use the Springer Nature journal content for small scale, personal non-commercial use, it is important to note that Users may not:

1. use such content for the purpose of providing other users with access on a regular or large scale basis or as a means to circumvent access control;
2. use such content where to do so would be considered a criminal or statutory offence in any jurisdiction, or gives rise to civil liability, or is otherwise unlawful;
3. falsely or misleadingly imply or suggest endorsement, approval , sponsorship, or association unless explicitly agreed to by Springer Nature in writing;
4. use bots or other automated methods to access the content or redirect messages
5. override any security feature or exclusionary protocol; or
6. share the content in order to create substitute for Springer Nature products or services or a systematic database of Springer Nature journal content.

In line with the restriction against commercial use, Springer Nature does not permit the creation of a product or service that creates revenue, royalties, rent or income from our content or its inclusion as part of a paid for service or for other commercial gain. Springer Nature journal content cannot be used for inter-library loans and librarians may not upload Springer Nature journal content on a large scale into their, or any other, institutional repository.
These terms of use are reviewed regularly and may be amended at any time. Springer Nature is not obligated to publish any information or content on this website and may remove it or features or functionality at our sole discretion, at any time with or without notice. Springer Nature may revoke this licence to you at any time and remove access to any copies of the Springer Nature journal content which have been saved.
To the fullest extent permitted by law, Springer Nature makes no warranties, representations or guarantees to Users, either express or implied with respect to the Springer nature journal content and all parties disclaim and waive any implied warranties or warranties imposed by law, including merchantability or fitness for any particular purpose.
Please note that these rights do not automatically extend to content, data or other material published by Springer Nature that may be licensed from third parties.
If you would like to use or distribute our Springer Nature journal content to a wider audience or on a regular basis or in any other manner not expressly permitted by these Terms, please contact Springer Nature at

onlineservice@springernature.com