Stochastic Resource Allocation

Liran Funaro
Technion—Israel Institute of Technology
Haifa, Israel
funaro@cs.technion.ac.il

Orna Agmon Ben-Yehuda
Technion—Israel Institute of Technology
Haifa, Israel
ladypine@cs.technion.ac.il

Assaf Schuster
Technion—Israel Institute of Technology
Haifa, Israel
assaf@cs.technion.ac.il

Abstract

Suboptimal resource utilization among public and private cloud providers prevents them from maximizing their economic potential. Long-term allocated resources are often idle when they might have been subleased for a short period. Alternatively, arbitrary resource overcommitment may lead to unpredictable client performance.

We propose a mechanism for fixed availability (traditional) resource allocation alongside stochastic resource allocation in the form of shares. We show its benefit for private and public cloud providers and for a wide range of clients. Our simulations show that our mechanism can increase server consolidation by 5.6 times on average compared with selling only fixed performance resources, and by 1.7 times compared with burstable instances, which is the most prevalent flexible allocation method. Our mechanism also yields better performance (i.e., higher revenues) or a lower cost than burstable instances for a wide range of clients, making it more profitable for them.

CCS Concepts • **Social and professional topics → Pricing and resource allocation**; • **Computing methodologies → Simulation evaluation**; • **Computer systems organization → Cloud computing**; • **Software and its engineering →** *Scheduling*;

Keywords Cloud, Resource-Allocation, Pricing, Scheduler

ACM Reference Format:
Liran Funaro, Orna Agmon Ben-Yehuda, and Assaf Schuster. 2019. Stochastic Resource Allocation. In *Proceedings of the 15th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE '19), April 14, 2019, Providence, RI, USA.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3313808.3313815

1 Introduction

Most cloud provider costs result from purchased servers, power requirements, and infrastructure (power and cooling systems) [30, 54, 79]. Hence, most of these costs are proportional to the number of servers. Although the CPUs on active servers are underutilized [13, 42], these servers still draw most of the power they would draw if their CPUs were fully utilized [61]. Further client consolidation would increase the revenues per server without increasing the costs [14].

CPU underutilization originates from the provider's obligation to provide its clients with their contracted quality of service (QoS) according to their service-level agreement (SLA) [1]. Infrastructure as a Service (IaaS) and Container as a Service (CaaS) clients rent a bundle of resources in the form of a virtual machine (VM) or an OS container. Even though clients may choose a fixed instance contract (e.g., a fixed performance instance on Amazon EC2), with a bundle that meets their load needs, they will not use their resources all the time. Moreover, because the proportion of resources in the bundle, e.g. the ratio of RAM to CPUs, is determined by the provider, it is not always optimal for the client. Therefore, most resources will generally have unused margins [56]. Suboptimal utilization might still be a problem even with less rigid services such as Application as a Service (AaaS) and serverless computing, where the client does not necessarily rent a bundle of resources but rather a black-box execution environment. Under such models, providers still set resources aside to handle unexpected loads [21] or cater to preferred clients requiring resources on short notice [39]. Providers lose money on these contingency plans because they do not maximize resource utilization. Maximizing resource utilization will enable providers to consolidate more clients on each machine, increase the income per machine, and reduce the pressure to expand their infrastructure.

How should providers allocate unutilized resources residing in a single physical machine among their clients in a manner that will increase the revenues of the former and incentivize the latter to agree to this allocation scheme?

A simple method for utilizing momentarily available resources in a single physical machine is to divide them among the clients/services that reside on that server: either evenly or proportionally to the amount of fixed resources they rented. Without an additional billing mechanism, the provider has no direct benefit from this approach. Moreover, if this is an ongoing state of affairs, the clients might take the higher

QoS for granted, and be disappointed when it decreases to the fixed (paid for) level.

The most popular approach for utilizing momentarily available resources is *burstable performance* [8, 19, 27, 49, 55]. On Amazon [8] and Azure [49], for example, a client gains credits periodically, at an even rate. The client either consumes credits by using the resource or hoards the credits and "bursts" later, using more resources than its periodic credits allow. If the client runs out of credits, it must wait for the next period to use the resource again. The credit mechanism limits the client to a certain average resource consumption according to its credit allocation rate.

We assume a client that consumes resources at an even rate, in line with its credits, is guaranteed never to be starved. This assumption requires the provider to prioritize clients that have not yet consumed the credits of the current period over clients that are currently "bursting". To never starve a non-bursting client, the provider must reserve for each such client a resource amount that equals the client's credit rate.

Both these quantities, the average resource consumption and the reserved resource quantity, are defined by the same number—the credit rate. This coupling is the main drawback of burstable instances. It induces two limitations: First, it forces a client that can function well without reserved resources to rent a bundle that offers them, just to get an average consumption rate. This requires that the provider reserve these resources, which in turn limits the number of clients per server. Second, this coupling limits each client's average utilization (over a period) to its reserved allocation. The sum of reserved resources in a server thereby serves as an upper limit on the server's total resource utilization. Resources not reserved for clients (e.g., the provider's reserves) cannot contribute to the total average utilization. Clients that did not reach their average utilization limit further reduce the total average utilization.

In *unlimited burstable instances*, the provider allows the client to exceed its average consumption rate, and charges a fixed (higher) price for the surplus average consumption during a billing period[1]. This overcomes the utilization limit but does not increase the number of clients per server.

In this paper, we show that decoupling the reserved resource quantity from the average consumption rate allows clients to explicitly reserve only the resources they truly require. In turn, this allows the provider to increase the number of clients per server.

Our contribution is twofold. First, we introduce a *Stochastic Allocation* (SA) mechanism that allows the provider to sell reserved resources alongside an additional stochastic allocation. We compare this mechanism to other mechanisms using simulations (Section 5) that cover a wide range of clients (Section 4). For over 56% of these clients, our mechanism is more profitable than the burstable performance mechanism. We show a 1.7 times increase in the number of clients per server compared with burstable instances (Section 7). We further show that such a mechanism can increase the provider's overall profits by 28%–44%, depending on our assumptions about the provider's profit margins. Moreover, we show that a private cloud provider can utilize this mechanism to increase its clients' aggregated economic benefit, while reducing the provider's costs.

Second, we present the Stochastic Allocation Simulator (SAS), a validated infrastructure for cloud simulations. This infrastructure can generate a large dataset of realistic clients with different behaviors and simulate their rational bundle selection given a known distribution of available resources. It then simulates the load on a server using these clients (using a completely fair scheduler (CFS) [52]) and yields highly detailed statistical information. SAS is published as an open-source project along with the code to replicate our simulations and their data[2].

2 Allocation Mechanisms and Incentives

A resource allocation mechanism is useful only if it is incentive-compatible for clients and providers. In other words, they must all gain from participating. In this section we classify clients by their requirements, discuss providers' goals, and then review a number of allocation mechanisms in view of the parties' incentives.

2.1 Client Requirements

Most client requirements range between *long-term requirements* and *immediate requirements*. A *long-term requirements* client may have non-interactive workloads. It might value finishing the workload by or before a deadline [20], but it might not value getting partial results ahead of time. It only cares about long-term promises that guarantee meeting deadlines with high probability.

At the other end of the spectrum is the *immediate requirements* client. It runs brief independent workloads or an interactive workload, and sleeps the rest of the time. The failure or fulfillment of one workload does not affect the client's future requirements. It only cares for instant gratification. Such a client may not wish to rent a full (usually underutilized) machine, which might guarantee each workload is finished on time. Rather, it may prefer to yield its resources when they are idle and be compensated accordingly.

The clients in between these two extremes have a combination of long-term and immediate requirements. They need a guarantee that their long-term requirements will be met, but might demand additional ad hoc resources to support an immediate load surge. They are *mixed requirements* clients.

Websites, for example, might partition their budget proportionately to the gain from satisfying these dual requirements. They would not like to miss an opportunity to show

[1]A day in Amazon [8], 5 minutes in CloudSigma [19].

[2]Available from: https://bitbucket.org/funaro/stochastic-allocation.

an advertisement to their visitors. Hence, the budget for their immediate requirements might be proportionate to the income from an ad. In addition, they would like to preserve their customers' visit rate. Users are unlikely to abandon them because of a momentary slowdown, but regular low responsiveness might reduce user visits. Thus, their budget for long-term requirements might be proportionate to the estimated loss of revenues due to an expected abandon rate.

The Azure Public Dataset [20] offers insight into how real cloud users are distributed by category/type. Most clients (60%) that ran over three days in the dataset were classified as delay insensitive (i.e., long-term requirements clients), and 33% were classified as interactive (i.e., short-term requirements clients). The other 7% could not be classified. Cortez et al. [20], who classified the clients, suggested that clients with short workloads, each with a deadline, might be classified as either interactive or unknown.

2.2 Provider Goals

Public cloud providers that rent computing resources to paying clients would like to maximize their profit from renting their machines. However, prices are limited due to price wars among providers [3]. Consequently, to increase their profits, public cloud providers resort to higher consolidation and overcommitment [26, 69]: they sell the same resources to more clients, risking an SLA violation and having to pay client compensation [68].

Private cloud providers would like to maximize the aggregated value all their clients draw from the cloud: the game-theoretic concept of *social welfare*. Accordingly, they would like to prioritize the most financially valuable clients, because their workloads carry the maximal benefit to the organization. Additionally, they wish to maximize client consolidation in their existing infrastructure, similar to public cloud providers.

2.3 Allocation Mechanisms

In this section we survey allocation mechanisms and pricing schemes that increase the server consolidation by incentivizing clients to reduce their reserved requirements. Providers often offer their clients one or more of these mechanisms simultaneously.

Fixed performance instances consist of a bundle of reserved resources. They are guaranteed to be constantly available to the client throughout the rental period. A long-term requirements client, however, usually only fully utilizes one resource in the bundle—which is its bottleneck. If the bundle's size and shape are determined by the provider, a long-term requirements client is likely to have non-required, unutilized resource margins. Clients that also have immediate requirements need to compromise: over-provision according to the maximal load at high costs—or under-provision and save money, at the risk of being short on resources. Thus, most clients pay for resources they do not utilize, and which the provider cannot resell.

Burstable performance instances offer a baseline resource guarantee, which may be exceeded when necessary. These instances are suitable for clients with immediate requirements, which are mostly inactive until driven by an event. Long-term requirements clients might not require bursting, as they typically use the resources at an even, maximal rate.

We compared pricing of burstable and fixed performance instances using identical CPU models and optimization types. According to the regression analysis, using Amazon's and Azure's publicly available pricing data [7, 8, 49], a burstable instance with similar price and characteristics to a fixed instance is limited to an average performance of 10%–30% of the fixed instance maximal performance, depending on the instance type.

Therefore, clients utilizing, on average, less than 10%–30% of their maximal resources will save money by renting burstable instances instead of fixed ones. Accordingly, pure long-term requirements clients will pay for burstable instances 3.3–10 times more than their fixed instance bill, to get the same performance.

Preemptible instances, deployed by many providers [6, 9, 28, 48, 53], offer a low-cost VM whose availability depends on the available resources in the cloud. The provider can shut down the instance at any time to reclaim the resources. An immediate requirements client can scale horizontally, i.e., expand the number of active VMs with an increasing load, at a low cost. Nevertheless, horizontal expansion incurs an overhead, for the provider and clients, when booting a machine and gracefully shutting it down. A long-term requirements client might use these instances whenever available or fall back to higher cost, nonpreemptible, instances [45].

This mechanism allows the provider to rent unallocated resources while waiting for higher paying clients to rent them. Unused reserved resources of other clients, however, cannot be used to create a new preemptible instance. Reserved resources must be supplied on demand, which is not possible due to the long notification [3] required before shutting down the preemptible instance.

Posted prices, formerly deployed by CloudSigma [37], are a mechanism for resource pressure management. In this mechanism, the provider periodically changes the resource unit-prices, which it posts publicly via an online API. Clients with immediate requirements can use the resource when the prices are low, while using the baseline for their long-term requirements.

If clients do not cap their resource utilization in response to price changes, posted prices might be ineffective in increasing client density. Clients might not reduce their consumption in response to price surges, as they might value steady

[3] 30 seconds in Google Cloud [28], two minutes for Amazon [9].

performance as long as the average cost remains within their budget [40, 77]. Moreover, clients will agree to participate only if price surges are limited by the cost of a horizontal expansion—the clients' alternative.

Immediate resource auctions allow clients to rent a baseline performance, and bid—every few seconds—for an immediate, temporary, resource allocation. Such a mechanism was implemented in Ginseng for RAM [4] and last level cache (LLC) [25]. It is suitable for clients with immediate requirements that need not plan ahead. Such clients can bid according to their momentary expected valuation of the resource. Clients with long-term requirements can also benefit from the mechanism by getting cheap resources when these are abundant. Nevertheless, it is hard for such clients to assign a momentary value to a resource with unknown future availability.

Similarly to posted prices, a horizontal expansion might be more profitable for some clients than costly, temporary but immediate, allocation.

3 Stochastic Allocation

Client performance can be quantified in terms of stochastic properties such as mean, standard deviation, minimum or maximum. The client can infer these on the basis of its experience [71]. Different clients might assign different monetary values to these properties [17, 24, 32, 47, 51, 62, 73], as described in Section 2.1. When offered a choice of bundles, each differently priced and stochastically characterized, the client can estimate its valuation for the bundle and its expected profit if selected [18].

Accordingly, to effectively utilize the resources, we propose the *Stochastic Allocation* (SA) mechanism. Under the SA mechanism, the provider offers clients a combination: a choice of a stochastic allocation class and an amount of reserved resources. The provider posts fixed unit-prices for both goods. Each client may choose to rent reserved and/or stochastic resources—the latter enable consumption of resources that are unused by other clients. The provider prioritizes clients when they consume their reserved resources. To allow clients to make an educated decision when renting such a stochastic bundle, the provider publishes statistics on resource availability, for each SA class.

SA supports an asymmetrical bundle of reserved resources and SA classes. It allows clients to reduce their reserved resource requirements, and thus increases the number of clients per server. The SA mechanism bridges the idle resource gap between the provider's obligation to safeguard clients' reserved resources and their dynamic demands.

3.1 Implementing Stochastic Allocation via Shares

To evaluate the SA mechanism for CPU using shares, we simulated it on the basis of the completely fair scheduler (CFS) [52] algorithm, as was implemented in the Linux kernel. CFS combines a share-based resource allocation system with a hard rate limit. Each task is assigned a number of shares, which entitle it to a portion of the resources proportional to the number of allocated shares. A task accumulates virtual runtime according to its actual runtime divided by its number of shares. In each period, the task with the least accumulated virtual runtime is run. Thus, at any given time, the virtual accumulated time of all active clients is nearly identical

CFS can easily implement a credit system, because having a portion of the shares is effectively the same as reserving the same portion of the resources. For example, in a machine with 64 CPUs, a process allocated 1 share out of a total of 64 is guaranteed at least 1 CPU. Nevertheless, CFS does not support a key feature of SA: defining a different consumption share for the leftover CPUs. Rather, the consumption rate is constrained to be identical to the reserved portion of the shares.

We added a degree of freedom to CFS, adapting it to support asymmetric reserved resources and share allocations. We duplicated the CFS logic, to have a second, alternative, virtual runtime clock and a second priority queue that is sorted according to the alternative clock. The existing knobs (share and limit) are associated with the original, main CFS, which is used as a reserve mechanism. The alternative CFS takes the newly introduced alt-share and alt-limit knobs. Once the limit rate of the main CFS is reached, the task is moved to the alternative one. The scheduler only pulls tasks from the alt-queue if the original queue is empty. For example, to reserve 1 CPU out of 64, allocate 10 shares (e.g., of 100) and set a total limit of 2 CPUs, the administrator would allocate a process with

- 1 main share such that $\frac{\text{client main share}}{\text{total main shares}} = \frac{\text{reserved CPU}}{\text{total CPUs}}$,
- a main limit of 1 CPU to mark the point where excess resources start being consumed,
- 10 alt-shares (out of 100) for the stochastic part, and
- an alt-limit of 2 CPUs, for the actual capping.

Our adapted CFS allows the provider to implement our SA mechanism. Once the client chooses its bundle of reserved resources and number of shares, the provider assigns the client to the appropriate server while trying to maintain an even distribution of total shares across the servers.

4 Realistic Workload Modeling

How effective would the SA mechanism be on a cloud? How does it compare with other mechanisms such as fixed performance and burstable instances? How would it affect client density? How would it affect the provider's profits?

Experimentation at this scale requires a full commercial cloud and thousands of real clients. To answer these questions, we resorted to simulations, showing the method's potential on equal grounds with the simulated burstable-performance. We modeled client workload and then simulated a cloud with various mechanisms (Section 5).

Our realistic modeling is based on real data from the Azure Public Dataset [20], which was used to deduce real consumption and market demand. We generated 12 client-datasets, which we ran in parallel on our 12-core machine. Each dataset contained 1024 clients.

To create a single dataset, we sampled a random group of clients from Azure's dataset. To model each client, we generated three functions that were consistent with its given statistics: performance, load, and valuation.

4.1 Performance

The performance function (resource → perf) indicates the maximal performance (in the range [0, 1]) that a resource allocation can yield, assuming the workload can utilize the resource. We generated a random monotonically rising function for each client. These functions are not necessarily always concave; they can have inflection points, as a real application utility function might have [70, 83]. Examples of generated performance functions are shown in Fig. 1.

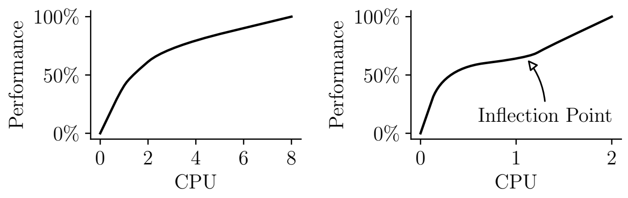


Figure 1. Generated performance function examples.

4.2 Load

The load function (time → perf) indicates the client's required performance (in the range [0, 1]) at a given time. For each sampled client, we generated a realistic load function for a single day, in 12-second intervals (i.e., 7200 samples for each client for each day). For IaaS and CaaS clients, this means that their VM/container was active for at least a day. For AaaS and serverless clients, this means that they had many small tasks which may span across a whole day and are considered as their day's load.

To do this, we used the client's sampled load from Azure's data. Azure's data contain statistics in 5-minute intervals per client, for up to 30 days. Each sample contains minimum, maximum and average CPU usage. To get enough statistical information, we chose only clients with at least a day's worth of data (288 samples).

The simplest way to interpret the data would be to maintain the average CPU usage constant over each 5-minute period, but then the extremum values would not be reached. To remedy that, the usage must reach other values, in particular the minimum and maximum values, and yet maintain the average usage. To take all the values in the sample into account, we divided each sample time into multiple samples that adhere to the given minimum, maximum and average CPU usage: min and max are visited, usage values are only between these values, and the average value is according to the measured data. This mandatory enhancement of the data introduces several degrees of freedom: which values to visit and when.

A simple solution is to visit the minimum and maximum once, and then fill the rest of the time with a value that will correct the average. This solution is arbitrary: the minimum and maximum values can be visited more. Also, it is natural for more values in the min-max range to be visited as well. This simple but reality-consistent solution can be smoothly extended using a beta distribution, which can be defined by its average, bounds, and density—i.e., having more samples near the mean value or near the bounds. The density represents a degree of freedom in the function choice.

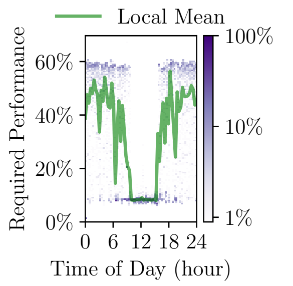
For each 5-minute sample, we generated a beta distribution with the sample's characteristics (i.e., minimum, maximum and average) and a density of one, which yields a uniform distribution when the average is exactly in the middle of the bounds (see Algorithm 1). We then drew samples from this beta distribution to fill the 5-minute interval with 25 samples (a sample every 12 seconds during the five minutes).

Algorithm 1: Generating a random sample.

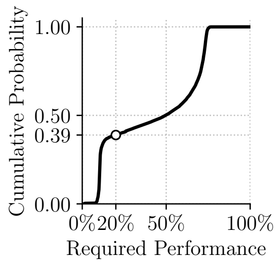
```
Data: $b$: minimum, $t$: maximum, $m$: mean, $d$: density
1 $m_s \longleftarrow \frac{m-b}{t-b}$ ; // scale to beta's domain: [0, 1]
2 if $m_s < 0.5$ then
3     $\alpha \longleftarrow \frac{d \cdot m_s}{1-m_s}, \beta \longleftarrow d$;
4 else
5     $\alpha \longleftarrow d, \beta \longleftarrow \frac{d \cdot (1-m_s)}{m_s}$;
6 end
7 return $beta(\alpha, \beta) \cdot (t-b) + b$ ; // draw and scale
```

We assumed that some clients might choose a smaller bundle than their maximal potential consumption. Hence, we extrapolated the client's consumption to what it would have been had it not been limited by its rented cores. To this end, if a sample's maximum was near the client's limit (i.e., within 90% of its number of virtual cores), we matched the sample with a beta function with a higher maximal value (64 cores), while maintaining the same average and minimum. That is, we created a modified beta function that allows for some over-the-top samples. Fig. 2a shows an example of a generated load from a real client.

Each client gets its realistic load samples and treats them as a load history. It models this history using a cumulative distribution function (CDF) (Fig. 2b), and uses it to statistically predict its load for the upcoming day, assuming that "That which hath been is that which shall be". The client later uses its statistical load prediction to predict its expected revenue and profit from the various bundles.



(a) The load over a single day. The local mean shows the average load over a local window, and the heat map shows the density of required performance over that window.



(b) The cumulative distribution of the load over that day. For example, this client's required performance will be less than 20% for 39% of the time.

Figure 2. A client's generated load. This client's required performance was 42% on average.

4.3 Valuation

Real (human) clients may choose an offering in any way they like. They may choose randomly, take some time to make a decision, or go through a long iterative process of selection and improvement. In the simulation we needed to create realistic artificial intelligence agents which mimic the behavior of real clients. We did this using the valuation function tool. A valuation function (perf ⟶ $) indicates the monetary value that a client attributes to the stochastic properties of the performance. It is based on business logic, such as the expected revenue from this performance.

The valuation of a client is the sum of two sub-functions; each takes different properties of the performance into account. The immediate valuation function, V_{imm}, represents the expected income from the immediate performance, defined in this work as the average performance over a short period of 12 seconds. The long-term valuation function, V_{lt}, represents the benefit from the long-term performance, i.e., the average performance over the entire day.

Using these two functions, a client can estimate the expected value of its revenue from a combined bundle of reserved resources and shares. Let us define two random variables: X_{load} denotes the client's load and X_s denotes the resource availability given a share allocation (s). Let r denote the client's reserved resources. The actual performance, $P_{r,s}$, is the minimum value of the load and the performance that the resources allow:

$$P_{r,s} = \min\{X_{load}, perf(r + X_s)\}. \quad (1)$$

The client calculates its expected revenue by adding two valuation functions: $E\left(V_{imm}(P_{r,s})\right)$, its expected revenue from immediate performance, and $V_{lt}\left(E(P_{r,s})\right)$, its expected revenue from long-term performance. Accordingly, the client estimates these random variables on the basis of its self-created load CDF and the CDF representing the potential use of a resource attributed to a share, supplied by the provider.

Some researchers modeled valuations using analytic functions (e.g., power law [59]), which are easy to symbolically analyze. But real user valuations are sophisticated [40]. We tailored to each client an individual, piece-wise linear function, to model actual consumption. Such non-symbolic functions may be harder to manipulate; however, they cover a wider range of functions.

To produce these valuation functions, we first characterized each client using three values: rented number of cores (i.e., its bundle), the expected revenue and the portion of the revenue affiliated with each requirement.

The number of rented cores was obtained from Azure's data. We used cores as the basic currency for the simulation: one core costs $1 a day. Assuming the client is rational, the cost of the cores the client rents is a lower bound on its valuation of these cores. We modeled the clients' expected revenue using a Pareto distribution (standard in economics) with an index of 1.1. A Pareto distribution with this parameter translates to the 80-20 rule: 20% of the population has 80% of the valuation, which is reasonable for income distributions [65].

For each client, we drew a value from this Pareto distribution, with the condition that the value is higher than the client's number of cores (i.e., a conditional probability distribution).

We used Azure's strict client classification (interactive, delay insensitive and unknown) to infer for each client the portions of the revenue affiliated with each requirement. To this end, we assigned to each class a different truncated normal distribution function (in the range [0, 1]), which describes the division between the requirements, as depicted in Fig. 3. For each client, we chose a distribution function according to its class, and drew a sample from it to get the client's budget portions.

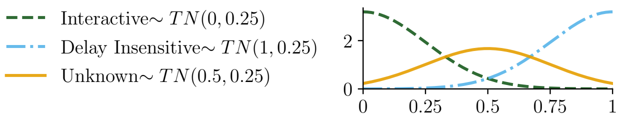
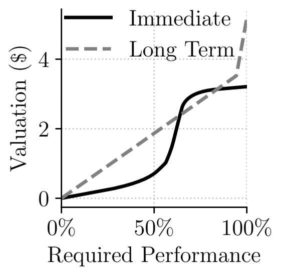


Figure 3. A probability density function (PDF) of the portion between the two valuation types. The horizontal axis describes the long-term requirements portion, and the immediate requirements portion completes it to 1.

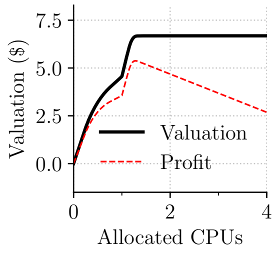
Using these three values (bundle, revenue and portion), we produced two monotonically rising functions, one for each valuation type. We engineered these functions such that when used to produce the valuation of each bundle, the client's bundle yields the maximal profit ($value - cost$), and its expected revenue for this bundle will be the revenue we draw for this client. To this end, we used an iterative process: we assumed that the revenue from zero performance is always zero and thus started with the identity functions $V_{imm}(x) = x$ and $V_{lt}(x) = x$. Then, in each iteration, we

1. estimated the client's expected value for different bundles using these functions;
2. adjusted each function, such that the value attributed to the client's selected bundle matched its portion of the revenue; and
3. adjusted the values that the functions attributed to other bundles, such that they were less profitable for the client.

Fig. 4 depicts an example of a generated valuation, and a simple example of a choice of fixed resources, i.e., $X_s \equiv 0$. The choice of a combination of reserved resources and shares is of a higher dimension, as the valuation function is $(shares \times reserved) :\longrightarrow money$.



(a) A client's immediate and long-term valuations.



(b) A client's valuation and expected profit for a fixed CPU allocation.

Figure 4. Creating the valuation function for various maximal available CPU usage values. This client's immediate requirements constitute 77% of its budget.

5 Evaluation Methodology

We simulated a cloud with various mechanisms: fixed performance, stochastic allocation, and the most prevalent "burst" mechanism. We compared the provider's revenue, resource utilization, and client density.

Our evaluation method was an iterative process. A single iteration simulated a day and took 2-3 real minutes to run. It is described in Fig. 5. An example of the progress of the full process is given in Fig. 6. The initial iteration simulated a cloud that offered only fixed performance (reserved) resources. The subsequent iterations simulated the introduction of another mechanism (e.g., stochastic allocation) alongside the reserved resources. Some clients were free to change their bundle choice at each step. We continued the process until the measurements were steady for at least 128 iterations (Fig. 6), and considered only the results of the last 60 iterations. Here we describe each step of our simulations, as depicted in Fig. 5.

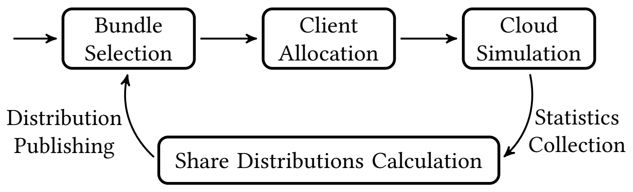


Figure 5. Iterative states in our evaluation methodology.

Selecting Fixed Performance and/or Share Allocation. Each client computed, for each possible bundle, the valuation it will draw from it. It used its own load statistics and the provider's statistical description of the resources that every share amount yields. Because the client could not foresee its exact load for the upcoming day, it used the load statistics gathered over the entire recorded period. The client selected the most profitable bundle of fixed performance and/or share allocation for its load and resource requirement distribution. Formally, the client's decision can be described as:

$$\underset{r,s}{\text{argmax}} \; \{E(V_{imm}(P_{r,s})) + V_{lt}(E(P_{r,s})) - \text{Cost}_{r,s}\}, \quad (2)$$

where r is the number of reserved cores, s is the number of shares, and $P_{r,s}$ is defined as in Section 4.3.

Changing Choices. Initially, each client chose a number of reserved cores. In each subsequent iteration, 128 out of the 1024 clients in each dataset (12.5%) were allowed to switch their bundle to any offer available in that simulation. This is consistent with the behavior of a real market, in which clients are unlikely to update their bundles all at once. Numerically, the limitation on the number of clients changing bundles simultaneously makes the solution method more stable, reducing oscillation over iterations and enabling the solution to converge.

At this stage, the provider's revenue was calculated by summing the prices of the clients' bundles.

Allocating Clients to Machines. To allocate clients to 64-core servers, we randomly shuffled them. Then, one at a time, each client was assigned to the first server that could accommodate the reserved component of its bundle. To ensure an even distribution of shares among the servers, a client that rented only shares was assigned to the server with the least sum of shares at that point.

We then calculated the average number of clients per server by repeating the allocation process and taking the average over the active servers. For a single resource, the assignment algorithm achieves near-optimal allocation, except for the last active server, which may be partially full. Accordingly, to measure the average number of clients per server, we disregarded the last active server. For a large cloud with more machines and clients, a single last server is negligible. Each dataset contains 1024 clients, so there are always enough full, representative active servers in the simulation.

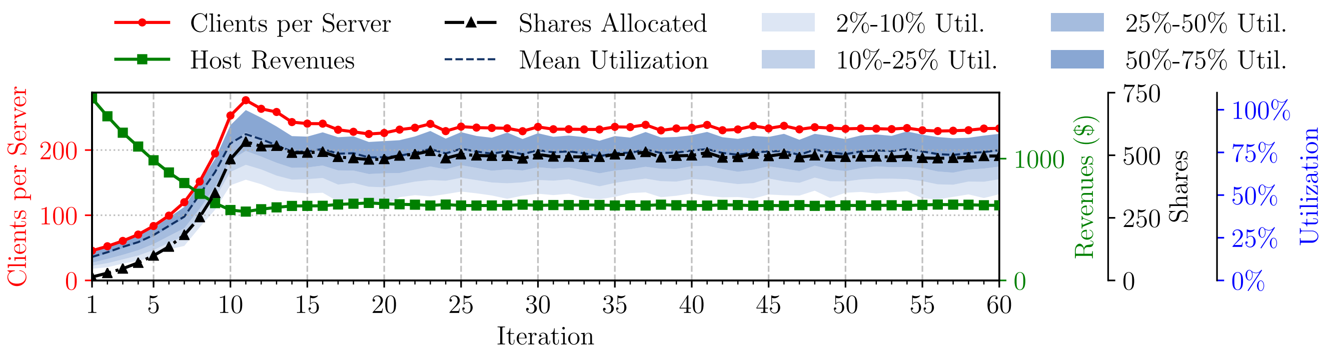


Figure 6. A typical iterative process and its convergence. The number of clients per server increases over time until it peaks and starts to drop due to the high utilization and the increasing number of shares per server, which reduce the value of a share. The provider's revenue decreases over time as more clients switch to cheaper bundles (with shares).

Simulating a Cloud. Each client load for the current day (iteration) was selected cyclically from its data over multiple days. We applied the server allocation algorithm 16 times for each of the 12 datasets and simulated the actual resource allocation of the first server of each dataset each time. Due to our assignment algorithm, the first server is the busiest. This serves as a worst-case analysis as these clients experience the most resource stress, and would thus be more reluctant to reduce their reserve requirements. The server's resource allocation was simulated using our modified CFS (Section 3.1).

At this stage, we collected client and server statistics: clients' expected revenue (i.e., their valuation), clients' effective revenue from their effective performance, and the server's resource utilization distribution.

Calculating the Statistical Potential of a Share. To allow the clients to rationally select a bundle of reserved resources and shares, our simulated provider supplies statistical information regarding the shares, which represents their potential: the distribution of the maximal resource amount that a client might attain over a short period—12 seconds in our case—with the commensurate number of shares. To this end, we collected the utilization statistics of the machines and computed their distributions. This produced an effective two-dimensional probability density function ($PDF_{cpu}(t, r)$) for the total unconsumed CPU (t), and the CPU that was not exploited by clients that reserved it (r). A client can utilize an unused reserved CPU that adheres to its proportional share, or utilize the entire total unconsumed CPU. Thus, the CDF of the probability for a client with portion p of shares to get x of the resource is given by

$$CDF_p(x) = \int_{t=0}^{x} \int_{r=0}^{\frac{x}{p}} PDF_{cpu}(t, r) \cdot dt \cdot dr. \quad (3)$$

To deduce the portion (p) of each offered number of shares, the provider uses the average total share allocation per server. It publishes, accordingly, the corresponding distributions for each offered number of shares, in the form of a CDF (CDF_p, according to Eq. 3). Identical distributions were published to all of the clients, regardless of their actual server allocation. The provider's utilization and average share allocation statistics remain concealed. Each client, then, uses the CDF of shares in the next bundle choice stage (Fig. 7).

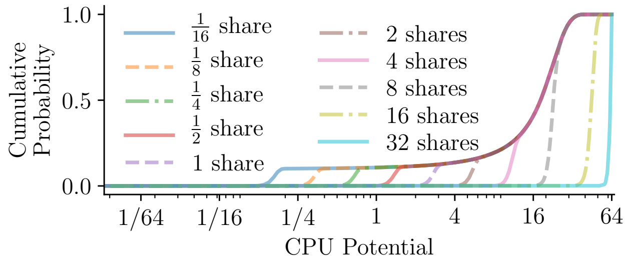


Figure 7. CDF of the potential resource use for a number of shares.

6 Compared Mechanisms

Similarly to public cloud providers, we offer clients a choice of CPU performance units. In our simulations, these units are core portions, i.e., $\frac{1}{16}$, $\frac{1}{8}$, $\frac{1}{4}$, $\frac{1}{2}$, 1, 2, 4, 8, 16, 32 core(s). In addition, the client can rent shares in the amount of $\frac{1}{16}$, $\frac{1}{8}$, $\frac{1}{4}$, $\frac{1}{2}$, 1, 2, 4, 8, 16, 32 shares. A client that rents shares is not obligated to rent reserved resources. We evaluated each of the following mechanisms separately:

Fixed Performance (FP). Each initial iteration is a choice among fixed performance offerings. This is our baseline. When evaluating the rest of the mechanisms, FP was always offered to the clients as an alternative.

Limited Stochastic Allocation (LSA). In our mechanism, the client can rent shares alongside reserved resources, and utilize them only up to their absolute value. E.g., a client that rented $\frac{1}{8}$ of a share can utilize up to $\frac{1}{8}$ of a core in addition to its reserved allocation, even if the machine is underutilized.

Unlimited Stochastic Allocation (USA). For completeness, we also tested our SA mechanism in a scenario where the client can rent shares and use them without any capping. Its utilization is limited only by the server's available resources, and is in proportion to the total number of actively used shares on the machine.

Burstable Performance (BP). We compared our mechanism to burstable performance, where instances are modeled assuming the allocated credit rate represents reserved resources. Moreover, the credit system limits the client to a certain average utilization per day; the provider will impose a fine for overutilization. Hence, we assumed rational clients will try to avoid exceeding the bundle's average allocation. To adhere to the strict coupling of the burstable instance offerings, we only let clients rent bundles in which the number of reserved resources equaled the number of shares.

To select the most profitable bundle, a client using BP has to predict, for each bundle, the limit that will prevent it from exceeding the bundle's average (i.e., its reserved allocation). To do this, the client takes into account its potential load and the statistical potential attributed to a share [76]. Once a bundle is selected, the client will not exceed its predicted limit so as to not incur penalties. However, overutilization was not fined by the provider.

According to our regression analysis (Section 2.3), a burstable instance is limited to an average performance of 10%–30% of the fixed instance maximal performance. Consequently, the cost of a bundle of matching reserved resources and shares should be 3.3–10 times the cost of renting only reserved resources. Given the fact that reserved resources cost \$1 per core unit, the corresponding share should bear the rest of the cost—that is, a share cost of \$2.3–\$9 per share unit.

We tested BP using share unit prices of \$2, \$3 and \$4. For LSA, we used share prices of \$0.15, \$0.5, \$0.6, \$0.7 and \$0.9 per share unit. For USA, we used higher share prices of \$3, \$5 and \$8, as they allow clients to use more of the resource and thus are more valuable to them.

LSA or BP (LSA/BP). We also tested the case where clients had a choice between these two mechanisms: LSA with a unit price of \$0.5 or USA with a unit price of \$3.

7 Results

First, we review our raw results by comparing the increase of clients per server while maintaining revenue. Then, using our simulation results, we make some assumptions to infer server costs, and use them to compare the provider's profit from the various mechanisms.

As Fig. 8 shows, LSA .5 (Limited SA with a unit price of \$0.5) can pack 1.7 times more clients into each server than BP 3 with the same revenue, and 5.7 times more than FP. USA has a similar number of clients per server (CPS) as LSA—albeit with significantly lower revenues.

LSA .15 allowed 233 CPS—the most among our tested cases, but reduced the provider's revenue significantly (35%) compared with BP 3. LSA .5 matched the provider's revenue when using BP 3, and allowed nearly as many CPS as LSA .15. LSA/BP allowed nearly as much CPS as LSA .5, however with lower revenues.

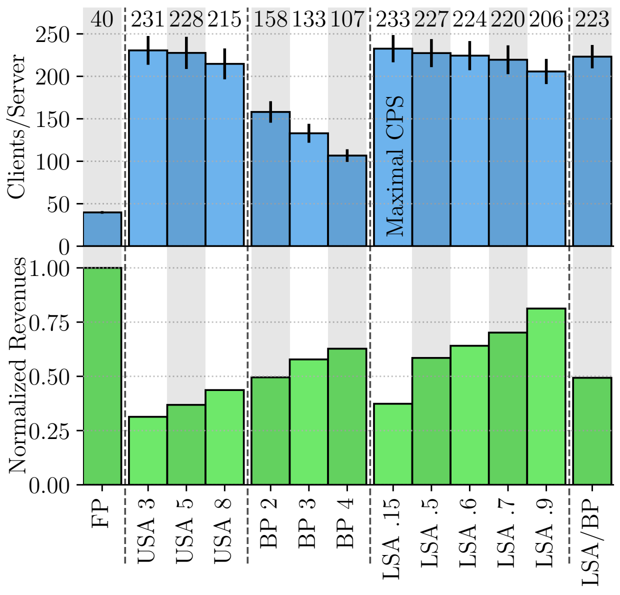


Figure 8. Comparison of allocated clients per server and provider's revenue. The number following the mechanism's name is the share price of each tested case. The revenues are normalized by the FP revenue. The error bars indicate the variance in CPS between the servers.

7.1 Provider's Economic Benefit

The public provider's profit depends on its expenses on hardware, energy and infrastructure purchasing: data we do not have. This data can be used to estimate the daily server costs and derive the profit from the revenue. Although revenues for non-FP schemes are reduced, the providers' profit may still grow due to lower daily server costs, which increase the profit margins—the profit divided by the revenue.

We estimated the server cost on the basis of our simulation results, and the assumption that the first provider to offer BP (Amazon) chose to offer this scheme to increase its profits. Amazon's BP pricing matches BP 3. Hence, we assume that the FP profits are lower than those of BP 3. The profits are equal when the profit margin of FP is 38%, which implies a daily cost of \$39 [4] (in reserved core price units), assuming no new clients joined due to the new attractive track. A lower profit margin of 25% implies a server cost of \$47, and a higher profit margin of 50% implies a server cost of \$31.

[4]The daily server cost is calculated as follows: $\frac{((1-\text{profit margin})\cdot\text{profit})}{\text{number of servers}}$.

In Fig. 9, we demonstrate the provider's profits in the different tracks, using these three hypothesized values for server costs. If a server costs \$39—the break-even case for FP and BP—the provider can increase its profit by over 35% and its profit margins by at least 22% by offering LSA .6 instances instead of BP 3. The higher server cost (\$47) leads to a 44% increase in profit. Moreover, in all of these cases, the profit margins grow when any stochastic allocation instances (excluding USA 3) are offered.

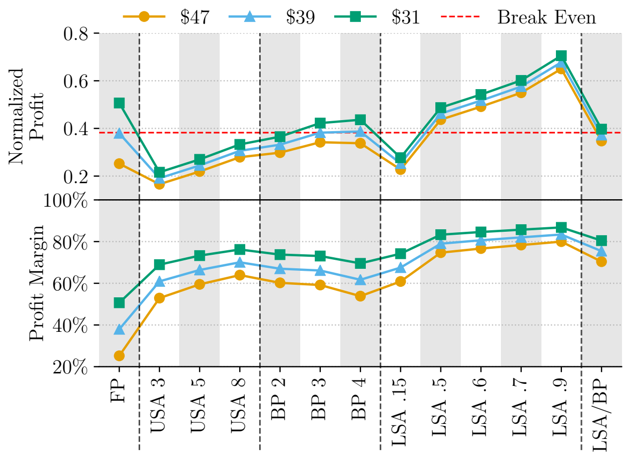


Figure 9. The provider's profit and profit margin. Given a server cost, the plot shows the expected profit and profit margin of the provider if it were to offer each tested mechanism. The profits are normalized by the FP revenues.

A private cloud provider is interested in maximizing the aggregated value of all its clients (social welfare). Fig. 10 shows that the BP track nearly maximized the social welfare (over 99% of the maximal social welfare, achieved when resources are abundant). LSA achieved over 97% of the maximal social welfare. It did so with fewer machines, meaning it produced 55% more value per machine than did BP. USA achieved a higher social welfare than LSA because it achieved higher resource utilization and thus created more value.

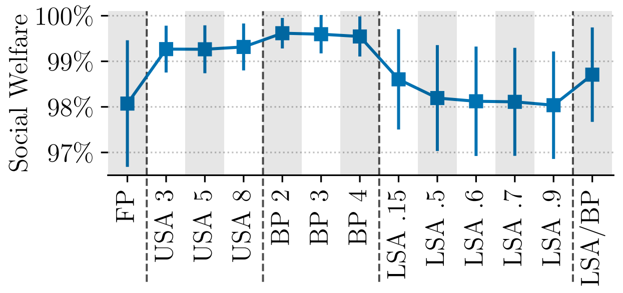


Figure 10. Comparison of the social welfare.

7.2 Server Utilization

The more expensive the stochastic allocation is, the lower the CPS will be (Fig. 8) and the lower the average utilization will be, as seen in Fig. 11. This is because fewer clients prefer it over a reserved allocation. The reserved allocations, which grow larger, are generally less utilized, as also seen in Fig. 11.

For BP, the mean and median reserved utilization were higher than for LSA and USA, but the average total utilization was lower. This is because BP forces the clients to rent a reserved resource in order to rent a proportionate share, although they might want a smaller amount of reserved resources. This proves our claim: decoupling the reserved allocation from the average allocation will be preferred by clients, and will also yield higher server consolidation.

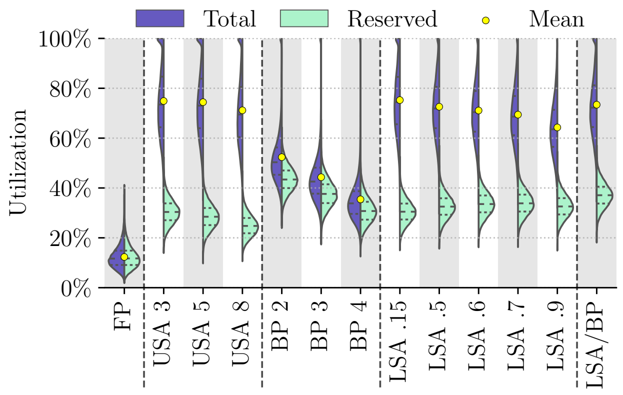


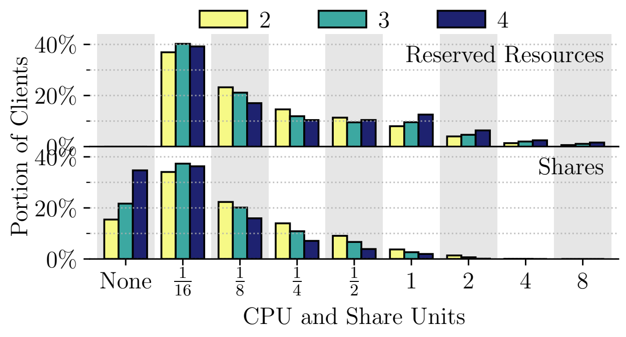
Figure 11. CPU utilization distribution for each tested case, differentiated by the total utilized CPU (left) and the portion of the CPU consumed only by clients that reserved it (right).

7.3 Clients' Preferences

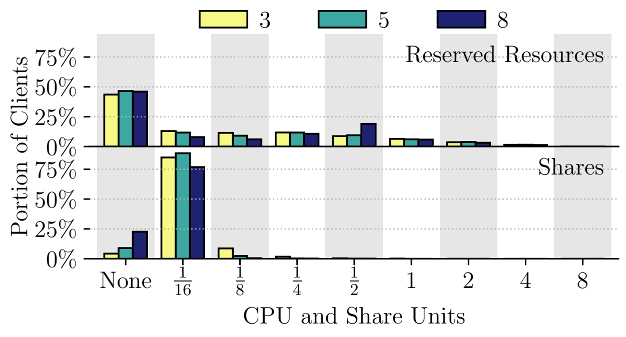
When offered a choice of FP and a flexible mechanism, how many would go for the flexible one? 92%–99% of the clients preferred LSA to FP (Fig. 12c), and 77%–96% preferred USA to FP (Fig. 12b). Only 65%–84% chose BP over FP (Fig. 12a. When offered BP 3 or LSA .5, 56% preferred LSA and 42% preferred BP. This indicates SA is more attractive to most clients than BP. Its flexibility enables it to cater to a wider set of client needs. Moreover, when offered any kind of stochastic allocation (either USA or LSA), 34%-46% of the clients avoided reserving resources at all.

7.4 Clients' Attainment Ratio

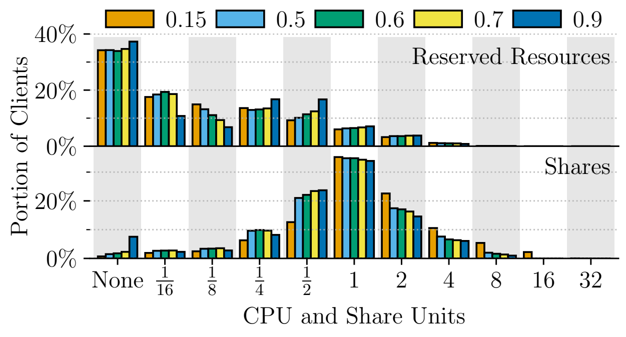
Every 12 seconds, we calculated each client's attained CPU utilization divided by its required CPU utilization—i.e., the client's attainment ratio. In all the tested cases (FP, BP, SA and USA), the average attainment ratio for all the clients was over 99.8%. This indicates that the clients were able to satisfy most of their load requirements.



(a) Distribution of burstable performance bundle.



(b) Distribution of unlimited shares bundle.



(c) Distribution of limited shares bundle.

Figure 12. The distribution of clients' selected bundles. Each color represents a different tested unit price for the shares.

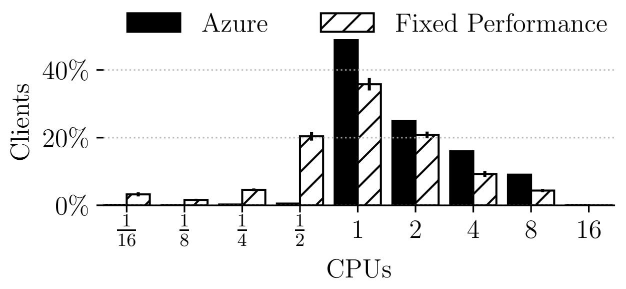


Figure 13. The selected number of virtual cores in our fixed-performance experiment and in the Azure dataset.

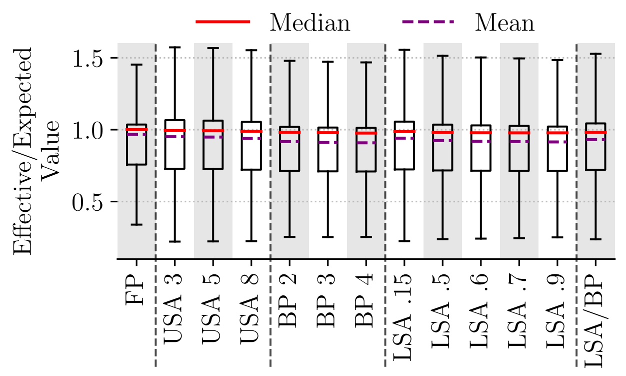


Figure 14. The distribution of clients' revenue normalized by their expected value in each tested case.

7.5 Validation

To validate our simulation, we compared the distribution of the selected bundles in our FP simulation to the distribution in the entire Azure dataset (2,013,767 clients). Our FP response profile distribution matches Azure's, with 10% less overprovisioning (Fig. 13). This is consistent with real clients being more risk averse than rational, simulated clients. This indicates that most of our simulated realistic clients have the same utility function distribution as real clients.

For further validation, we compared the utilization distribution in our FP simulation to real cloud data. Measurements taken before the introduction of burstable instances indicated average CPU utilization of 15%–20% [13, 42], which is consistent with the FP's mean utilization in our simulation results, shown in Fig. 11.

We also confirmed that the clients in our simulations act rationally, that their load expectations are realistic, and that the published shares' potential is accurate. To do this, we compared the clients' expected value (before the server simulation) to their effective value (their actual revenues from the simulated performance). The average effective value tended to be slightly lower than the expected value, as seen in Fig. 14. The high variance is expected as the clients use statistics from up to a month to predict the load of a single day.

We validated that the iterations we chose—the last 60—indeed converged. Over the course of the last 60 iterations, up to 12% of the clients changed their selected bundle from the first iteration to the last, in all the tested cases. Moreover, the standard deviation of the selected bundles' distribution over these iterations was under 0.6% and the standard deviation of the shares CDF was under 0.01%, in all the tested cases.

Finally, we analyzed the effect the different assumptions would have on the results. When we modified the **beta density** to be 0.5, 10 or 50, CPS was increased by up to 6% for

SA, on the one hand, and reduced by up to 5% for BP, on the other, compared with the main value (1). When we avoided the **over-the-top** extrapolation of the generated load values, CPS was reduced by up to 7%. When we modified the **performance functions** so they were linear and concave, CPS was reduced by up to 3% compared with monotonically increasing ones. When we modified the **Pareto index**, CPS was reduced by up to 6% for a Pareto index of 0.8 and increased by up to 1% for an index of 1.3, compared with the main index (1.1). We also modified the number of clients that can **change their bundle**. The average CPS was not affected when 384 (or less) clients changed their bundle at once. When more than 256 clients changed their bundle, however, the results fluctuated. When more than 384 clients changed their bundle, the results failed to converge.

In all of the simulations, we compared the CPS ratio of LSA over BP. Throughout the above-mentioned modifications, this ratio turned out higher than in the main results presented earlier (Fig. 8). This indicates that the main results are numerically sound.

8 Related Work

Agmon Ben-Yehuda et al. [1, 3] predicted that cloud providers will reduce the resource allocation intervals, as they currently do. They also predicted the need for sophisticated economic mechanisms to efficiently allocate resources in the cloud. This work and the other mechanisms mentioned here follow this principle.

Many researchers have suggested ways to improve server consolidation and social welfare other than those used in the industry. Dynamic pricing schemes have been proposed to regulate demand [82] or reduce interference [35]. Shahrad et al. [59] also suggested incentivizing clients to limit their burstiness via an incentive compatible pricing scheme, in which clients profit from limiting themselves.

Other researchers suggested allowing clients to communicate information to the provider (the desired availability [60, 63], long-term (months) required service level objectives [17] or short term requirements [31]). This approach places the burden of placement and scheduling on the provider's allocator, which must ensure that the client's requirements are met with high probability. That is, it must solve an optimization problem. Our solution is simpler for the provider, who only needs to publish its statistics, and leaves the burden of making an informed choice to clients. Other researchers collected statistics about clients to improve utilization [16, 46, 74], efficiency [66, 67], consolidation [78] or energy consumption [43] via placement algorithms or resources reallocation.

Many solutions have been proposed for improving the utilization of dedicated large-scale clusters given a job scheduler [5, 10–12, 15, 22, 23, 29, 33, 34, 36, 38, 41, 44, 50, 57, 58, 64, 72, 74, 80, 81]. Such schedulers are more flexible, and thus reach higher utilization than public clouds. They can do so because the data center provider is often the one that deploys this system and can control the fine-tuning of each task. Real clients' utility functions were characterized on such systems [40, 77], but the characterization is inapplicable to public clouds or private clouds without a centralized scheduler.

9 Conclusions and Future Work

Stochastic CPU allocation via shares allows clients to reduce their reserved resource requirements. This allows the provider to increase the number of clients per server by more than 70% compared with burstable performance. As such, our method can increase the profits of the public cloud provider by over 28% compared with burstable instances. Furthermore, our method also benefits private cloud providers as it increases the client social welfare per server. It increases the value each server generates for the corporation by over 55%.

The private cloud's social welfare can be improved further by allowing clients to bid for shares, just as clients bid for spot instances in horizontal scaling [2, 9, 28, 48]. Formulating a bidding and valuation language for stochastic allocation remains as future work. The provider might also charge the clients an additional, fixed, price-per-use to discourage resource waste. Analyzing this is left for future work as well.

Our simulations show that almost all clients will prefer using our mechanisms versus the fixed performance track, and over 56% will prefer it over burstable performance. Our SA mechanisms offer a cheaper track than either fixed performance or burstable, and do not mandate reserving any resources. Hence, new clients, who were unable to afford other cloud services, might now join the cloud and further increase the provider's revenues, without cannibalizing the market share of the existing offerings.

Our simulation infrastructure was validated using real data from a real cloud. Our methodology and the clients' rationale were validated by the accuracy of the clients' expectations, despite the reduced and compact data at their disposal. We hope that our infrastructure, published as an open source project, will allow more research on the applicability of novel allocation methods.

To implement our stochastic allocation method in a real cloud, our modified CFS should be implemented in the Linux kernel. It can also be implemented on other resources that can be allocated via a proportional share mechanism [75]. Investigating the coexistence of this mechanism over multiple resources remains as future work as well.

Acknowledgments

We thank Sharon Kessler, Deborah Miller, Danielle Movsowitz, Tamar Camus, Shunit Agmon and Muli Ben-Yehuda for fruitful discussions. This work was partially funded by the Hasso Platner Institute, and by the Pazy Joint Research Foundation.

References

[1] Orna Agmon Ben-Yehuda, Muli Ben-Yehuda, Assaf Schuster, and Dan Tsafrir. 2012. The Resource-as-a-Service (RaaS) Cloud. In *Proceedings of the 4th USENIX Conference on Hot Topics in Cloud Computing (HotCloud)*. USENIX Association. http://portal.acm.org/citation.cfm?id=2342775

[2] Orna Agmon Ben-Yehuda, Muli Ben Yehuda, Assaf Schuster, and Dan Tsafrir. 2013. Deconstructing Amazon EC2 Spot Instance Pricing. *ACM Transactions on Economics and Computation (TEAC)* 1, 3 (2013). https://doi.org/10.1145/2509413.2509416

[3] Orna Agmon Ben-Yehuda, Muli Ben-Yehuda, Assaf Schuster, and Dan Tsafrir. 2014. The Rise of RaaS: The Resource-as-a-Service Cloud. *Commun. ACM* 57, 7 (2014), 76–84. https://doi.org/10.1145/2627422

[4] Orna Agmon Ben-Yehuda, Eyal Posener, Muli Ben-Yehuda, Assaf Schuster, and Ahuva Mu'alem. 2014. Ginseng: Market-driven Memory Allocation. In *Proceedings of the 10th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE)*, Vol. 49. ACM, ACM, 41–52. https://doi.org/10.1145/2576195.2576197

[5] Orna Agmon Ben-Yehuda, Assaf Schuster, Artyom Sharov, Mark Silberstein, and Alexandru Iosup. 2012. ExPERT: Pareto-Efficient Task Replication on Grids and a Cloud. In *IEEE 26th International Parallel & Distributed Processing Symposium (IPDPS)*. IEEE, 167–178. https://doi.org/10.1109/ipdps.2012.25

[6] Alibaba. 2018. Alibaba Cloud Spot Instances. https://www.alibabacloud.com/help/doc-detail/52088.htm. Accessed: 2018-05-03.

[7] Amazon. 2017. Amazon EC2 On-Demand Pricing. https://aws.amazon.com/ec2/pricing/on-demand/. Accessed: 2017-07-24.

[8] Amazon. 2018. Amazon EC2 Burstable Performance Instances. https://aws.amazon.com/ec2/instance-types/#burst. Accessed: 2018-07-25.

[9] Amazon. 2018. Amazon EC2 Spot Instances. https://aws.amazon.com/ec2/spot/details/. Accessed: 2018-07-25.

[10] Apache. 2018. Apache Aurora Project. http://aurora.incubator.apache.org/. Accessed: 2018-07-25.

[11] Apache. 2018. Apache Hadoop Project. http://hadoop.apache.org/. Accessed: 2018-07-25.

[12] Apache. 2018. Apache Spark Project. http://spark.apache.org/. Accessed: 2018-07-25.

[13] Michael Armbrust, Armando Fox, Rean Griffith, Anthony D. Joseph, Randy Katz, Andy Konwinski, Gunho Lee, David Patterson, Ariel Rabkin, Ion Stoica, and Matei Zaharia. 2010. A View of Cloud Computing. *Commun. ACM* 53, 4 (2010), 50–58. https://doi.org/10.1145/1721654.1721672

[14] Luiz André Barroso, Jimmy Clidaras, and Urs Hölzle. 2013. The datacenter as a computer: An introduction to the design of warehouse-scale machines. *Synthesis Lectures on Computer Architecture* 8, 3 (2013), 1–154.

[15] Eric Boutin, Jaliya Ekanayake, Wei Lin, Bing Shi, Jingren Zhou, Zhengping Qian, Ming Wu, and Lidong Zhou. 2014. Apollo: Scalable and Coordinated Scheduling for Cloud-Scale Computing. In *OSDI*, Vol. 14. 285–300.

[16] Brendan Burns, Brian Grant, David Oppenheimer, Eric Brewer, and John Wilkes. 2016. Borg, Omega, and Kubernetes. *Queue* 14, 1 (2016), 10.

[17] Marcus Carvalho, Walfredo Cirne, Francisco Brasileiro, and John Wilkes. 2014. Long-term SLOS for reclaimed cloud computing resources. In *Proceedings of the ACM Symposium on Cloud Computing*. ACM, 1–13.

[18] Sivadon Chaisiri, Bu-Sung Lee, and Dusit Niyato. 2012. Optimization of resource provisioning cost in cloud computing. *IEEE Transactions on Services Computing* 5, 2 (2012), 164–177.

[19] CloudSigma. 2018. CloudSigma Cloud Pricing. https://www.cloudsigma.com/pricing/. Accessed: 2018-07-25.

[20] Eli Cortez, Anand Bonde, Alexandre Muzio, Mark Russinovich, Marcus Fontoura, and Ricardo Bianchini. 2017. Resource Central: Understanding and Predicting Workloads for Improved Resource Management in Large Cloud Platforms. In *Proceedings of the 26th Symposium on Operating Systems Principles*. ACM, 153–167.

[21] Rostand Costa, Francisco Brasileiro, Guido Lemos, and Dênio Sousa. 2013. Analyzing the impact of elasticity on the profit of cloud computing providers. *Future Generation Computer Systems* 29, 7 (2013), 1777–1785.

[22] Mehiar Dabbagh, Bechir Hamdaoui, Mohsen Guizani, and Ammar Rayes. 2015. Energy-Efficient Resource Allocation and Provisioning Framework for Cloud Data Centers. *IEEE Trans. Network and Service Management* 12, 3 (2015), 377–391.

[23] Christina Delimitrou and Christos Kozyrakis. 2014. Quasar: resource-efficient and QoS-aware cluster management. *ACM SIGPLAN Notices* 49, 4 (2014), 127–144.

[24] Benjamin Farley, Ari Juels, Venkatanathan Varadarajan, Thomas Ristenpart, Kevin D Bowers, and Michael M Swift. 2012. More for your money: exploiting performance heterogeneity in public clouds. In *Proceedings of the Third ACM Symposium on Cloud Computing*. ACM, 20.

[25] Liran Funaro, Orna Agmon Ben-Yehuda, and Assaf Schuster. 2016. Ginseng: market-driven LLC allocation. In *Proceedings of the 2016 USENIX Conference on Usenix Annual Technical Conference*. USENIX Association, 295–308.

[26] Rahul Ghosh and Vijay K Naik. 2012. Biting off safely more than you can chew: Predictive analytics for resource over-commit in iaas cloud. In *2012 IEEE 5th International Conference on Cloud Computing (CLOUD)*. IEEE, 25–32.

[27] Google. 2018. Google Cloud Compute Engine Pricing. https://cloud.google.com/compute/pricing. Accessed: 2018-07-25.

[28] Google. 2018. Google Cloud Preemptible VM Instances. https://cloud.google.com/compute/docs/instances/preemptible. Accessed: 2018-07-25.

[29] Robert Grandl, Ganesh Ananthanarayanan, Srikanth Kandula, Sriram Rao, and Aditya Akella. 2015. Multi-resource packing for cluster schedulers. *ACM SIGCOMM Computer Communication Review* 44, 4 (2015), 455–466.

[30] James Hamilton. 2018. Cost of Power in Large-Scale Data Centers. https://perspectives.mvdirona.com/2008/11/cost-of-power-in-large-scale-data-centers/. Accessed: 2018-07-25.

[31] Michael R. Hines, Abel Gordon, Marcio Silva, Dilma Da Silva, Kyung Ryu, and Muli Ben-Yehuda. 2011. Applications Know Best: Performance-Driven Memory Overcommit with Ginkgo. In *2011 IEEE 3rd International Conference on Cloud Computing Technology and Science (CloudCom)*. IEEE, 130–137. https://doi.org/10.1109/cloudcom.2011.27

[32] Alexandru Iosup, Simon Ostermann, M Nezih Yigitbasi, Radu Prodan, Thomas Fahringer, and Dick Epema. 2011. Performance analysis of cloud computing services for many-tasks scientific computing. *IEEE Transactions on Parallel and Distributed systems* 22, 6 (2011), 931–945.

[33] Michael Isard. 2007. Autopilot: automatic data center management. *ACM SIGOPS Operating Systems Review* 41, 2 (2007), 60–67.

[34] Michael Isard, Vijayan Prabhakaran, Jon Currey, Udi Wieder, Kunal Talwar, and Andrew Goldberg. 2009. Quincy: fair scheduling for distributed computing clusters. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*. ACM, 261–276.

[35] Vatche Ishakian, Raymond Sweha, Azer Bestavros, and Jonathan Appavoo. 2012. Cloudpack* exploiting workload flexibility through rational pricing. In *Proceedings of the 13th International Middleware Conference*. Springer-Verlag New York, Inc., 374–393.

[36] Navendu Jain, Ishai Menache, Joseph Seffi Naor, and Jonathan Yaniv. 2015. Near-optimal scheduling mechanisms for deadline-sensitive jobs in large computing clusters. *ACM Transactions on Parallel Computing* 2, 1 (2015), 3.

[37] Kristof Kovacs. 2018. Charting CloudSigma burst prices. https://kkovacs.eu/cloudsigma-burst-price-chart. Accessed: 2018-07-25.

[38] Kubernetes 2018. Kubernetes. http://kubernetes.io. Accessed: 2018-07-25.

[39] Ewnetu Bayuh Lakew, Cristian Klein, Francisco Hernandez-Rodriguez, and Erik Elmroth. 2015. Performance-based service differentiation in clouds. In *IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid), 2015 15th*. IEEE, 505–514.

[40] Cynthia B Lee and Allan E Snavely. 2007. Precise and realistic utility functions for user-centric performance analysis of schedulers. In *Proceedings of the 16th International Symposium on High Performance Distributed Computing*. ACM, 107–116.

[41] Jacob Leverich and Christos Kozyrakis. 2014. Reconciling high server utilization and sub-millisecond quality-of-service. In *Proceedings of the Ninth European Conference on Computer Systems*. ACM, 4.

[42] Huan Liu. 2012. Host Server CPU Utilization in Amazon EC2 Cloud. http://huanliu.wordpress.com/2012/02/17/host-server-cpu-utilization-in-amazon-ec2-cloud/. Accessed: 2018-07-25.

[43] Xiao-Fang Liu, Zhi-Hui Zhan, Ke-Jing Du, and Wei-Neng Chen. 2014. Energy aware virtual machine placement scheduling in cloud computing based on ant colony optimization approach. In *Proceedings of the 2014 Annual Conference on Genetic and Evolutionary Computation*. ACM, 41–48.

[44] David Lo, Liqun Cheng, Rama Govindaraju, Parthasarathy Ranganathan, and Christos Kozyrakis. 2015. Heracles: Improving resource efficiency at scale. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture*. ACM, 450–462.

[45] Tania Lorido-Botrán, José Miguel-Alonso, and Jose Antonio Lozano. 2012. Auto-scaling techniques for elastic applications in cloud environments. *Department of Computer Architecture and Technology, University of Basque Country, Tech. Rep. EHU-KAT-IK-09* 12 (2012).

[46] Barnaby Malet and Peter Pietzuch. 2010. Resource allocation across multiple cloud data centres. In *Proceedings of the 8th International Workshop on Middleware for Grids, Clouds and e-Science*. ACM, 5.

[47] Ming Mao and Marty Humphrey. 2012. A performance study on the VM startup time in the cloud. In *2012 IEEE International Conference on Cloud Computing*. IEEE, 423–430.

[48] Microsoft. 2018. Azure batch low-priority VMs. https://azure.microsoft.com/en-gb/blog/announcing-public-preview-of-azure-batch-low-priority-vms/. Accessed: 2018-07-25.

[49] Microsoft. 2018. Microsoft Azure AKS B-series Burstable VM. https://azure.microsoft.com/en-us/blog/introducing-b-series-our-new-burstable-vm-size/. Accessed: 2018-07-25.

[50] Hiep Nguyen, Zhiming Shen, Xiaohui Gu, Sethuraman Subbiah, and John Wilkes. 2013. AGILE: Elastic Distributed Resource Scaling for Infrastructure-as-a-Service.. In *ICAC*, Vol. 13. 69–82.

[51] Zhonghong Ou, Hao Zhuang, Jukka K. Nurminen, Antti Ylä-Jääski, and Pan Hui. 2012. Exploiting hardware heterogeneity within the same instance type of Amazon EC2. In *4th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud)*.

[52] Chandandeep S. Pabla. 2009. Completely Fair Scheduler. *Linux Journal* 2009, 184 (2009), 4.

[53] Packet. 2018. Packet Cloud Spot Instances. https://www.packet.net/bare-metal/deploy/spot/. Accessed: 2018-06-02.

[54] Steven Pelley, David Meisner, Pooya Zandevakili, Thomas F Wenisch, and Jack Underwood. 2010. Power routing: dynamic power provisioning in the data center. In *ACM Sigplan Notices*, Vol. 45. ACM, 231–242.

[55] Rackspace. 2018. Rackspace Cloud Flavors. https://developer.rackspace.com/docs/cloud-servers/v2/general-api-info/flavors/. Accessed: 2018-09-27.

[56] Charles Reiss, Alexey Tumanov, Gregory R Ganger, Randy H Katz, and Michael A Kozuch. 2012. Heterogeneity and dynamicity of clouds at scale: Google trace analysis. In *Proceedings of the Third ACM Symposium on Cloud Computing*. ACM, 7.

[57] Thomas Sandholm and Kevin Lai. 2010. Dynamic proportional share scheduling in hadoop. In *Workshop on Job Scheduling Strategies for Parallel Processing*. Springer, 110–131.

[58] Malte Schwarzkopf, Andy Konwinski, Michael Abd-El-Malek, and John Wilkes. 2013. Omega: flexible, scalable schedulers for large compute clusters. In *Proceedings of the 8th ACM European Conference on Computer Systems*. ACM, 351–364.

[59] Mohammad Shahrad, Cristian Klein, Liang Zheng, Mung Chiang, Erik Elmroth, and David Wentzlaff. 2017. Incentivizing self-capping to increase cloud utilization. In *Proceedings of the 2017 Symposium on Cloud Computing*. ACM, 52–65.

[60] Mohammad Shahrad and David Wentzlaff. 2016. Availability knob: Flexible user-defined availability in the cloud. In *Proceedings of the Seventh ACM Symposium on Cloud Computing*. ACM, 42–56.

[61] Yogesh Sharma, Bahman Javadi, Weisheng Si, and Daniel Sun. 2016. Reliability and energy efficiency in cloud computing systems: Survey and taxonomy. *Journal of Network and Computer Applications* 74 (2016), 66–85.

[62] Supreeth Shastri. 2018. *System Support for Managing Risk in Cloud Computing Platforms*. Ph.D. Dissertation. University of Massachusetts Amherst.

[63] Supreeth Shastri, Amr Rizk, and David Irwin. 2016. Transient guarantees: maximizing the value of idle cloud capacity. In *Networking, Storage and Analysis, SC16: International Conference for High Performance Computing*. IEEE, 992–1002.

[64] Mark Silberstein, Artyom Sharov, Dan Geiger, and Assaf Schuster. 2009. GridBot: execution of bags of tasks in multiple grids. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*. ACM, 11.

[65] Wataru Souma. 2001. Universal structure of the personal income distribution. *Fractals* 9, 04 (2001), 463–470.

[66] Byung Chul Tak, Youngjin Kwon, and Bhuvan Urgaonkar. 2013. Towards An Effective and General Resource Accounting and Control Framework in Consolidated IT Platforms. In *Proceedings of the Seventh Workshop on Large-Scale Distributed Systems and Middleware (LADIS '13)*.

[67] Priyanka Tembey, Ada Gavrilovska, and Karsten Schwan. 2014. Merlin: Application-and platform-aware resource allocation in consolidated server systems. In *Proceedings of the ACM Symposium on Cloud Computing*. ACM, 1–14.

[68] Luis Tomás, Cristian Klein, Johan Tordsson, and Francisco Hernández-Rodríguez. 2014. The straw that broke the camel's back: safe cloud overbooking with application brownout. In *2014 International Conference on Cloud and Autonomic Computing (ICCAC)*. IEEE, 151–160.

[69] Luis Tomás and Johan Tordsson. 2013. Improving cloud infrastructure utilization through overbooking. In *Proceedings of the 2013 ACM Cloud and Autonomic Computing Conference*. ACM, 5.

[70] Alexey Tumanov, James Cipar, Gregory R Ganger, and Michael A Kozuch. 2012. alsched: Algebraic scheduling of mixed workloads in heterogeneous clouds. In *Proceedings of the Third ACM Symposium on Cloud Computing*. ACM, 25.

[71] Ruben Van den Bossche, Kurt Vanmechelen, and Jan Broeckhove. 2015. IaaS reserved contract procurement optimisation with load prediction. *Future Generation Computer Systems* 53 (2015), 13–24.

[72] Vinod Kumar Vavilapalli, Arun C Murthy, Chris Douglas, Sharad Agarwal, Mahadev Konar, Robert Evans, Thomas Graves, Jason Lowe, Hitesh Shah, Siddharth Seth, et al. 2013. Apache hadoop yarn: Yet another resource negotiator. In *Proceedings of the 4th Annual Symposium on Cloud Computing*. ACM, 5.

[73] Christian Vecchiola, Suraj Pandey, and Rajkumar Buyya. 2009. High-performance cloud computing: A view of scientific applications. In *2009 10th International Symposium on Pervasive Systems, Algorithms,*

and Networks. IEEE, 4–16.

[74] Abhishek Verma, Luis Pedrosa, Madhukar Korupolu, David Oppenheimer, Eric Tune, and John Wilkes. 2015. Large-scale cluster management at Google with Borg. In *Proceedings of the Tenth European Conference on Computer Systems*. ACM, 18.

[75] Carl A Waldspurger. 1995. *Lottery and stride scheduling: Flexible proportional-share resource management.* Technical Report. Cambridge, MA, USA.

[76] Cheng Wang, Bhuvan Urgaonkar, Neda Nasiriani, and George Kesidis. 2017. Using Burstable Instances in the Public Cloud: Why, When and How? *Proceedings of the ACM on Measurement and Analysis of Computing Systems* 1, 1 (2017), 11.

[77] John Wilkes. 2009. *Utility Functions, Prices, and Negotiation.* New York: Wiley.

[78] Dan Williams, Hani Jamjoom, Yew-Huey Liu, and Hakim Weatherspoon. 2011. Overdriver: Handling memory overload in an oversubscribed cloud. In *ACM SIGPLAN Notices*, Vol. 46. ACM, 205–216.

[79] Qiang Wu, Qingyuan Deng, Lakshmi Ganesh, Chang-Hong Hsu, Yun Jin, Sanjeev Kumar, Bin Li, Justin Meza, and Yee Jiun Song. 2016. Dynamo: Facebook’s data center-wide power management system. In *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 469–480.

[80] Hailong Yang, Alex Breslow, Jason Mars, and Lingjia Tang. 2013. Bubble-flux: Precise online qos management for increased utilization in warehouse scale computers. In *ACM SIGARCH Computer Architecture News*, Vol. 41. ACM, 607–618.

[81] Zhuo Zhang, Chao Li, Yangyu Tao, Renyu Yang, Hong Tang, and Jie Xu. 2014. Fuxi: a fault-tolerant resource management and job scheduling system at internet scale. *Proceedings of the VLDB Endowment* 7, 13 (2014), 1393–1404.

[82] Liang Zheng, Carlee Joe-Wong, Chee Wei Tan, Mung Chiang, and Xinyu Wang. 2015. How to bid the cloud. In *ACM SIGCOMM Computer Communication Review*, Vol. 45. ACM, 71–84.

[83] Xiaoyun Zhu, Zhikui Wang, and Sharad Singhal. 2006. Utility-driven workload management using nested control design. In *American Control Conference, 2006.* IEEE.