BurScale: Using Burstable Instances for Cost-Effective Autoscaling in the Public Cloud

Ataollah Fatahi Baarzi
The Pennsylvania State University

Timothy Zhu
The Pennsylvania State University

Bhuvan Urgaonkar
The Pennsylvania State University

ABSTRACT

Cloud providers have recently introduced burstable instances - virtual machines whose CPU capacity is rate limited by token-bucket mechanisms. A user of a burstable instance is able to burst to a much higher resource capacity ("peak rate") than the instance's long-term average capacity ("sustained rate"), provided the bursts are short and infrequent. A burstable instance tends to be much cheaper than a conventional instance that is always provisioned for the peak rate. Consequently, cloud providers advertise burstable instances as cost-effective options for customers with intermittent needs and small (e.g., single VM) clusters. By contrast, this paper presents two novel usage scenarios for burstable instances in larger clusters with sustained usage. We demonstrate (i) how burstable instances can be utilized alongside conventional instances to handle the transient queueing arising from variability in traffic, and (ii) how burstable instances can mask the VM startup/warmup time when autoscaling to handle flash crowds. We implement our ideas in a system called BurScale and use it to demonstrate cost-effective autoscaling for two important workloads: (i) a stateless web server cluster, and (ii) a stateful Memcached caching cluster. Results from our prototype system show that via its careful combination of burstable and regular instances, BurScale can ensure similar application performance as traditional autoscaling systems that use all regular instances while reducing cost by up to 50%.

CCS CONCEPTS

• **Computer systems organization → Cloud computing**.

KEYWORDS

autoscaling, resource provisioning, burstable instances, cloud computing

ACM Reference Format:
Ataollah Fatahi Baarzi, Timothy Zhu, and Bhuvan Urgaonkar. 2019. BurScale: Using Burstable Instances for Cost-Effective Autoscaling in the Public Cloud. In *ACM Symposium on Cloud Computing (SoCC '19), November 20–23, 2019, Santa Cruz, CA, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3357223.3362706

1 INTRODUCTION

One of the key appeals of operating in the public cloud is autoscaling, the ability to elastically scale cluster capacity to match the workload's dynamically evolving resource needs. Autoscaling can help a cloud customer ("tenant") lower its costs compared to provisioning resources based on the peak needs at all times, which would be the case if the tenant were to use privately owned IT infrastructure. Many workloads exhibit predictable workload variations, e.g., time-of-day effects or seasonal patterns. Such workloads stand to benefit from an autoscaling facility. Autoscaling can also help a tenant respond effectively to less predictable phenomena, e.g., "flash crowds", particularly if it has high agility in the sense of allowing addition of new capacity (or removal of existing capacity) quickly.

Existing autoscaling solutions tend to employ regular instances whose resource capacities remain *fixed* over time, e.g., on-demand or reserved instances in Amazon Web Services (AWS). However, for interactive workloads, these fixed capacities are not utilized entirely. Fundamentally, provisioned capacity must be higher than used capacity to have enough spare capacity for handling transient periods of queueing and workload spikes. As these effects occur at fine time scales, there is not enough time for autoscaling, so the resources must be instantly available. We find that the recently introduced *burstable* instances offer a cheaper alternative to regular instances for procuring such spare capacity.

Burstable instances are virtual machines whose CPU (and sometimes network bandwidth) capacity is a mere fraction of that of a regular instance on average. However, they have the ability to occasionally burst up to a peak rate comparable to that of the regular instance in a manner regulated by a token-bucket mechanism. That is, burstable instances accrue CPU credits during periods of CPU utilization lower than their long-term average ("base") allocated capacity. These credits are in turn spent during periods when CPU utilization exceeds the base capacity. Burstable instances are a lot cheaper than *comparable* on-demand (regular) instances, where by "comparable" we mean regular instances whose CPU rate equals the peak rate available to the burstable instance. For example, as of May 2019, an Amazon EC2 $m5.large$ instance (with 2 vCPUs and 8 GB of memory) was almost 10 times more expensive than a $t3.micro$ instance (with 2 vCPUs each at 10% sustained rate and the same peak rate and 1 GB of memory). Note that these savings come with the caveat of low memory usage. Despite this, we show how our techniques can apply both to a stateless web application and a stateful caching layer.

Our interest lies in devising an enhancement for autoscaling that combines burstable instances with regular instances for improving costs over state-of-the-art autoscaling. We use the term autoscaling to mean elastically provisioning enough resources to meet a workload's performance needs as demand changes over

SoCC '19, November 20–23, 2019, Santa Cruz, CA, USA
© 2019 Association for Computing Machinery.
ACM ISBN 978-1-4503-6973-2/19/11…$15.00
https://doi.org/10.1145/3357223.3362706

time. Our solution, BurScale, uses existing mechanisms to add and remove resources from the system as demand changes over time. While we do not innovate in all aspects of autoscaling, we focus on the core aspects of autoscaling pertaining to cost efficiency, which primarily involves resource provisioning. Our work considers two key research questions: (i) how many of the instances should be burstable, and (ii) how should requests be load balanced or unbalanced between regular and burstable instances? We address these questions in the context of a stateless web application, a stateful cache, and flash crowd scenarios. The key idea is exchanging a fraction of the regular instances for burstable instances and modifying the load balancer to only utilize burstable instances when there is substantial queueing. Thus, the burstable instances do not exhaust their CPU credits while still being able to mask the effects of transient queueing and the startup delay when autoscaling.

We make three primary research contributions:

- We introduce a new technique for reducing the cost of autoscaling systems by provisioning using a combination of burstable and regular instances. Our technique involves two main parts: (i) determining the number of regular instances that should be replaced with burstable instances[1], and (ii) dynamically adjusting the load balancer to avoid exhausting the burstable instances' CPU credits. We implement and evaluate our technique in our prototype BurScale system, which is open-sourced at https://github.com/PSU-Cloud/BurScale.
- We show how our technique applies even to stateful caches, where memory is important for performance. As the cost benefit of using burstable instances depends on low memory usage, it is insightful to see how certain caching scenarios can take advantage of burstable instances.
- We adapt our technique for flash crowds and demonstrate how it masks the startup delay when acquiring new resources in response to a flash crowd (i.e., an unexpected traffic surge). By using burstable instances to supply the necessary overprovisioning, we see savings up to 50% compared to only using regular instances.

2 BACKGROUND

In this section, we discuss the features of burstable instances and the context under which we utilize them.

2.1 Burstable Instances

In order to monetize their idle resources, public cloud providers like *Google Cloud Engine* [15], *Amazon Web Services* [3, 4], and *Microsoft Azure* [26] offer a variety of instance types that are cheaper on average than the more well-established on-demand or "regular" instances. These include reserved instances, preemptible (e.g., spot) instances, and **burstable instances**. Our interest in this work lies in burstable instances, which are cheap rate-limited instances that can sporadically burst to peak CPU rate for short durations. Specifically, we focus on burstable instances offered by AWS, the cloud provider with the oldest and most diverse burstable offerings. However, our ideas readily apply to offerings from other providers as well.

Currently, Amazon offers burstable instances in the *t2* and *t3* categories, each with 7 different options for resources ranging from 1-8 virtual CPUs (vCPUs) and 0.5-32 GiB RAM. The appeal of burstable instances is in their significantly reduced price. For example, a t3.nano burstable instance has 2 vCPUs and is only 5.4% the cost of a m5.large regular instance with 2 vCPUs. With such a low price, burstable instances also come with their drawbacks. For example, the t3.nano instance is rate-limited to an average of 5% of the CPU capacity and has 6.25% of the memory capacity as compared to the m5.large.

In a sense, a burstable instance represents a fractional CPU that is able to burst to 100% capacity for limited periods of time. Amazon controls this through a token-bucket rate limiter mechanism where a token is called a CPU credit. One CPU credit means the instance can use 100% of one vCPU for 1 minute. Alternatively, an instance can use one CPU credit for running a vCPU at 50% for two minutes or 25% for 4 minutes. Credit usage and CPU utilization are tracked at a millisecond granularity. As long as an instance has CPU credits it can use its whole vCPU capacity, otherwise its usage is limited to the baseline CPU rate for that instance type. Different burstable instance types are configured to earn different rates of CPU credits with higher rates being priced higher. Credits that are not used will accumulate up to 24×*Hourly Credit Earning Rate* credits. Amazon has also provided a mechanism to not run out of CPU credits by purchasing them when the token bucket is empty. However, the cost of CPU credits makes burstable instances more expensive than regular instances if one wanted to use them all the time.

Thus, burstable instances are not a good choice for batch workloads that need maximal utilization all the time. Rather, this dynamism in CPU performance and having a lower price compared to regular instances make burstable instances a good candidate for workloads that use low-moderate levels of CPU most of the time, but sometimes need higher levels of CPU. Thus, applications like microservices, low-latency interactive applications, small databases, virtual desktops, development, build and stage environments, code repositories, and product prototypes can benefit from burstable instances.

2.2 Applications

In this paper, we present novel use cases for using burstable instances to reduce cost while maintaining performance goals known as service-level objectives (SLOs). We focus on interactive workloads wherein a server application responds to requests from latency sensitive clients. In section 3, we demonstrate how our ideas generalize to two canonical examples of such an application: (i) a replicated web server, and (ii) an in-memory key-value cache.

Web Application. The first type of application we consider is a PHP web application serving a combination of static pages and dynamically-generated content. In our experiments, we use the Wikimedia application, which is the application used for operating the Wikipedia website. We find this application is both CPU and network intensive. So to support a high rate of requests, an application will often be replicated across many independent web servers with a central load balancer that directs each request to one

[1] For simplicity, we assume the peak performance of burstable instances matches that of regular instances so that a burstable instance can replace a regular instance as long as its CPU credits are not exhausted.

of the servers. Traditionally, all of these servers would be regular instances, and our work will show that a subset of these can be converted to burstable instances without significant loss of performance once appropriate changes are made to the load balancer.

In-Memory Cache. The second type of application we consider is an in-memory key-value cache, such as Memcached [25]. The key distinction with this type of application is its statefulness, which requires memory. Whereas CPU and network bandwidth can easily be shared between instances across time, memory capacity is not easily shared across time. Since a burstable instance has memory dedicated to it, this limits the number of other instances that a cloud provider can co-locate on the same physical machine. Consequently, a burstable instance with the same allocated memory capacity as a regular instance has nearly the same (only slightly lower) price. Thus, burstable instances prove to be cost-effective replacements of regular instances only in cases where, in addition to CPU/network needs being intermittent, the memory needs are also low.

One might imagine that the cost of memory would preclude the use of burstable instances with memory caches, and this is true in some scenarios. If more instances are needed in a system to increase the total memory capacity of the cache, then burstable instances have very limited benefits in these cases. However, there are cases such as in Microsoft's web cache [9] where a cache is replicated for a higher aggregate CPU or network bandwidth rather than memory capacity. In such scenarios, burstable instances are a perfect fit since they can still provide benefit when only replicating the most popular ("hot") key-value pairs. As items within caches often exhibit skewness in their popularity that can be captured by a heavy-tailed distribution (e.g., Zipfian [34]), the hot keys constitute a relatively small portion of the whole working set, but are used to serve a disproportionately large fraction of the incoming traffic. Our work investigates the issues arising from such a use case – namely the fact that burstable instances are unable to serve all of the requests, and adding a new cache server to the system will incur a warm up time to populate the cache.

3 BURSCALE DESIGN

In this section, we show how BurScale uses burstable instances in combination with regular instances to improve the cost-efficiency of autoscaling in two applications: a web application and an in-memory key-value cache.

3.1 Our Autoscaling Approach

A typical autoscaling system monitors and records workload behavior over time and periodically adjusts the number of servers in the cluster to minimize costs while meeting performance goals. It is often composed of two components: (i) a component that estimates future workload properties based on current and previous behavior, and (ii) an application performance model that determines the number of servers, k, needed to satisfy the performance goal based on the estimated workload properties. As the desired number of servers changes over time, the autoscaling system adds or removes servers from the system and updates the cluster configuration accordingly.

Future Workload Predictor. Autoscaling systems are often categorized into predictive [14, 20], reactive [10, 12], and hybrid [30, 35] approaches. Predictive autoscaling systems utilize complex future workload predictors and sometimes need to build in mechanisms for protecting against mispredictions. On the other hand, reactive autoscaling systems are often much simpler and assume the future behavior will closely resemble the current behavior. *The techniques in BurScale can directly apply to any autoscaling system, but for the sake of simplicity in understanding experimental results, we employ a reactive approach where the future behavior is assumed to be the same as the current behavior.* In subsection 3.4, we explore how to handle flash crowd where the future behavior is drastically different from the current behavior.

Application Performance Model. The application performance model is used to determine the number of servers, k, needed to maintain low latency for a given request arrival rate, λ, and server service rate, μ. In this work, we consider systems composed of k servers receiving work from a central *Join the Shortest Queue* (*JSQ*) load balancer. Under the JSQ load balancing policy, prior work [23, 27] has shown that the system can be approximated by a $M/M/k$ queueing system, which is defined as a system with k servers processing requests from a central queue where requests arrive according to a Poisson process and request sizes follow an exponential distribution. Under the $M/M/k$ queueing model, we can apply the queueing theory result known as the Square-Root Staffing rule (SR rule) [17], which approximates the minimum number of servers needed to keep the probability of queueing, P_Q, below a user-defined threshold (SLO), α:

Theorem 1. ***Square-Root Staffing Rule (SR Rule)*** *[17] Given an $M/M/k$ queueing system with arrival rate λ and service rate μ, let $R = \frac{\lambda}{\mu}$, and let ${k_\alpha}^*$ denote the least number of servers needed to ensure that the probability of queueing ${P_Q}^{M/M/k} < \alpha$. Then ${k_\alpha}^* \approx R + c\sqrt{R}$ where c is the solution for the equation $\frac{c\Phi(c)}{\phi(c)} = \frac{1-\alpha}{\alpha}$ where $\Phi(.)$ denotes the c.d.f. of the standard Normal distribution and $\phi(.)$ denotes its p.d.f.*

Here, the c parameter is directly related to the upper bound on the probability of queueing, α, which is in turn related to latency, our target metric. That is, higher values of c result in more instances and consequently reduced queueing and latency. To calculate c, one would perform a binary search on c until the equation in Theorem 1 is approximately[2] satisfied. To approximate a latency SLO via a probability of queueing (P_Q) SLO, one would use the following equation [17]:

$$E[T] = \frac{1}{\lambda}.P_Q.\frac{\rho}{1-\rho} + \frac{1}{\mu} \tag{1}$$

where $\rho = \frac{\lambda}{k\mu}$ is the load of the system.

We find that this $M/M/k$ model is reasonable for our workloads, but *we do not claim that it is the best performance model for all scenarios. Rather, it is a reasonable model that we use in BurScale to demonstrate our ideas on using burstable instances, which we believe can extend to other autoscaling models that calculate k differently.*

Preliminary Validation. To confirm that this model applies to our system, we experimentally determine the minimum number of servers needed to meet a latency SLO for different arrival rates (Figure 1). We compare this with two theoretical formulas: one using the SR rule with a fixed probability of queueing $P_Q < 10\%$

[2] Theorem 1 is an approximation, so the calculation of c need not be exact.

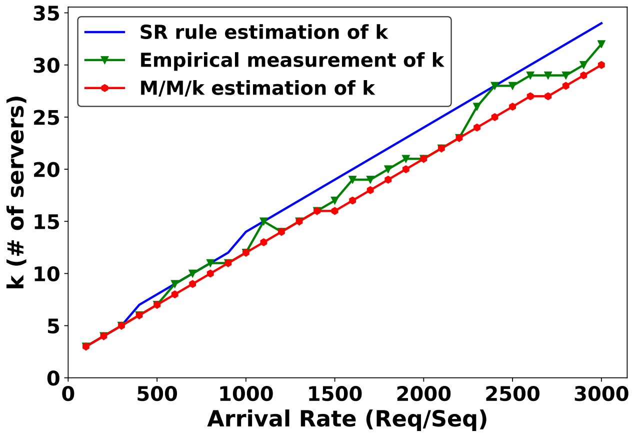

Figure 1: Comparing the Square-Root Staffing rule (SR rule) and M/M/k model with empirically determining the number of servers, k, needed to satisfy a latency SLO. Both theoretical models are reasonable approximations for capacity provisioning.

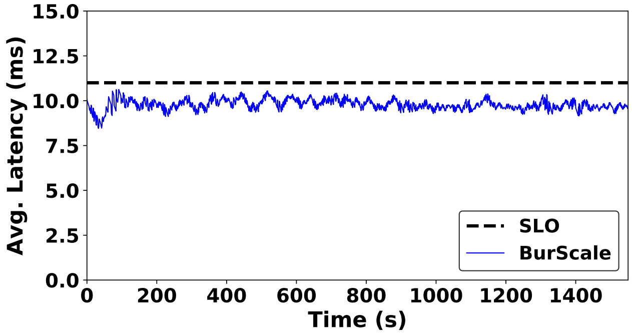

Figure 2: Measured latency in our microbenchmark web application stays under the desired SLO when BurScale uses R regular instances and $(k - R)$ burstable instances.

and the other based on constraining Equation 1 to be less than the SLO. As seen in Figure 1, the SR rule and Equation 1 are both reasonable heuristics for approximating the number of servers needed for maintaining a latency SLO.

For the purposes of this validation experiment, we use a simple CPU intensive PHP script for our application, and we generate requests with exponentially distributed request sizes and Poisson process arrival times. Our later experiments in section 5 show that the SR rule is a good approximation for real-world workloads where the arrival process is not a Poisson process and request sizes do not follow an exponential distribution.

3.2 Cost-effective Autoscaling

In this section, we develop a new technique for enhancing autoscaling systems by replacing some regular instances with burstable instances. We address two key research questions: (i) how many burstable instances should be used, and (ii) what dispatching policy should be used?

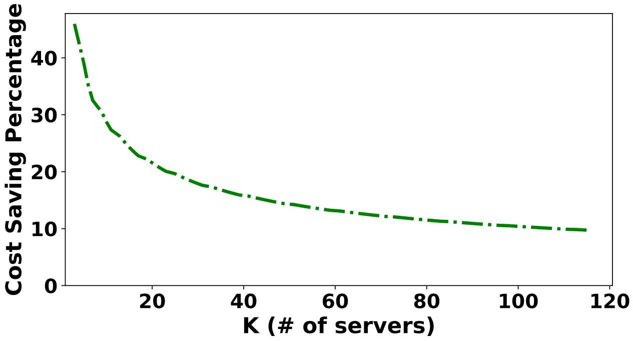

Figure 3: Percentage cost savings from using $(k - R) = c\sqrt{R}$ burstable instances vs. using all regular instances for different cluster sizes.

How many burstable instances? As described in subsection 3.1, we assume the autoscaling system provides information about the number of instances, k, needed to satisfy the SLO. For our experiments, we use a reactive policy based on the mean latency in a $M/M/k$ model, but our idea extends to other autoscaling policies that are predictive or based on tail latency or any other metric of interest. Thus, our contribution is in determining how many of these k instances should be of the burstable kind. If we use burstable instances too much, then they may run out of CPU credits and be unable to operate at peak performance when needed. For simplicity, we select burstable instances with the same number of vCPUs as our regular instances so that the peak performance of the burstable instances roughly matches that of our regular instances.

The key intuition behind our approach stems from Theorem 1, which suggests $k \approx R + c\sqrt{R}$. Here, the R term indicates the minimum number of instances needed to keep the system stable (i.e., not overloaded with more work than it can handle). With R instances, a system could theoretically handle all the incoming traffic without dropping requests, but queues would grow undesirably long. The extra $c\sqrt{R}$ instances are then used to handle the transient queueing that results from variability of traffic (i.e., both arrival time variability and request size variability). Thus, we propose to use R regular instances and $(k - R)$ burstable instances. The R regular instances would be used to handle the bulk of the traffic, whereas the $(k - R)$ burstable instances would only be used during short transient periods with higher amounts of queueing. We will often refer to these two parts of the expression for the number of instances as the "R part" and the "$(k - R)$ part," respectively. Even if we use an oracular scaling policy that selects the optimal number of k instances to use at each moment in time, the key idea behind BurScale of splitting k into R regular and $(k - R)$ burstable instances is beneficial in reducing cost.

Request Dispatching Policy. To make the above simple idea workable, we need to be careful about how we distribute requests among the regular and burstable instances. On the one hand, we would like to avoid situations wherein the burstable instances are used too frequently and run out of credits; on the other hand, we

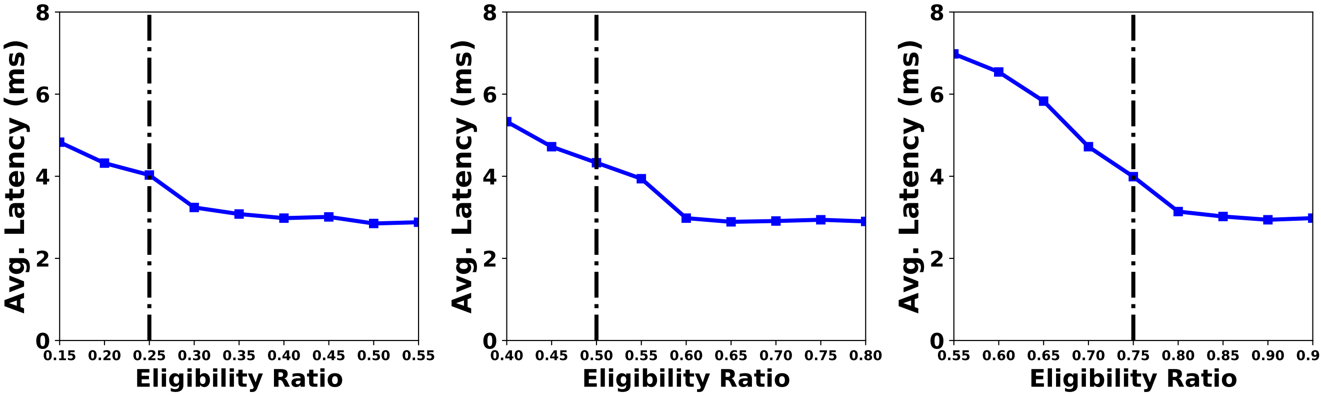

Figure 4: The effect of the eligibility ratio (i.e., fraction of traffic that can be handled by burstable instances) on latency in an in-memory key-value cache. Left: (3 regular, 1 burstable). Middle: (2 regular, 2 burstable). Right: (1 regular, 3 burstable). The latency stabilizes when the eligibility ratio exceeds $\frac{\# burstable}{total\ instances}$ (i.e., $\frac{k-R}{k}$), which indicates that the burstable instances have sufficient items cached to serve enough traffic for maintaining low latencies.

do not want to underutilize them thereby causing performance to degrade during transient surges in the workload. To address this problem, we explore the use of a weighted version of the join the shortest queue (*JSQ*) dispatching policy. The key challenge here is to choose the relative weight for burstable instances such that they accumulate credits during periods of low workload intensity and use credits during periods of flash crowds and high workload intensity. Towards this, BurScale periodically monitors the CPU usage of burstable instances and adjusts their weights until the CPU usage of the burstable instances is equal or less than the baseline credit earning rate allocated to them. This is because burstable instances have a net gain in CPU credits when their CPU usage is below the baseline rate.

Preliminary Validation. To validate our decomposition of k instances into R regular and $(k - R)$ burstable instances, we run a preliminary microbenchmark with weights chosen for *JSQ* as described above. We run a workload with an average 800 reqs/sec arrival rate against our test web application (subsection 3.1) with a service rate of 113 reqs/sec. BurScale's autoscaling policy sets k to be 11, of which 8 are regular instances (R part) and 3 are burstable instances ($(k - R)$ part). BurScale dynamically updates the weights of instances, which prevents the burstable instances from losing credits in the long-term average while also meeting the SLO, as seen in Figure 2.

Cost Savings. From a cost saving point of view, we compare BurScale's approach to the case where all k of the instances are regular instances (m5.large). By converting some of the instances to burstable instances (t3.small), we are able to lower costs while still maintaining the desired SLO. Figure 3 shows the cost savings as a function of k for $P_Q = 0.1$ with current AWS pricing. As seen, small clusters may see up to 46% cost savings while larger clusters may see up to 10% cost savings.

3.3 Enhancements for Stateful Applications

We next enhance our technique for stateful applications such as an in-memory key-value cache (e.g., Memcached). As described in subsection 2.2, burstable instances are expensive when configured with a large memory capacity. Thus, applications that require a large amount of memory are unsuitable for burstable instances. One might think that this would preclude an in-memory cache, but surprisingly, there are scenarios when it is beneficial to use burstable instances for these types of applications. Specifically, when a large cluster of caching servers are needed to serve a high arrival rate of requests (e.g., as in [9]) due to network or CPU bottlenecks, then burstable instances can be used in the same way as in subsection 3.2. So in our experiments, we replicate items across caching servers and focus on the scenario where we use many caching servers to handle high request rates rather than increase hit ratios or the total cache size.

Existing works such as ElMem [16] and CacheScale [43] have demonstrated techniques for how to autoscale in-memory caches, so the focus of our work is on determining how to select the right type of burstable instance. The challenge here is that since the burstable instances will not contain as many items as regular instances, they will be incapable of serving some of the items. So in addition to choosing the number of burstable instances, BurScale also needs to select the amount of memory allocated to the instances. The way BurScale handles this is by first selecting the number of burstable instances as $(k - R)$ as before. Ideally, during periods where there is substantial queueing, one would expect them to handle a $\frac{k-R}{k}$ fraction of the traffic. Thus, we need the burstable instances to at least be eligible to serve $\frac{k-R}{k}$ of the requests as compared to a regular instance. Practically, this needs to be slightly higher than this ratio since the requests arrive randomly, so we use $\frac{k-R}{k} + 0.1$ (determined experimentally below). We define the fraction of traffic that can be served by the burstable instances as the *eligibility ratio*, which is the ratio between the burstable and regular instances' hit ratios. Now once we have a desired eligibility ratio, we can then profile the workload offline to determine its popularity distribution. We then select the memory capacity of the burstable instances such that they achieve the desired eligibility ratio when replicating the most popular "hot" items. The hot items are determined through profiling, and we leverage mcrouterâĂŹs *PrefixRouting* feature (âĂIJhâĂİ for hot keys and âĂIJcâĂİ for cold keys in our case) [24] to differentiate between the hot and cold keys so that the load balancer appropriately direct the traffic to the appropriate cache based on a weighted JSQ policy similar to our approach in subsection 3.2. Given that most popularity distributions exhibit skewness that can be captured by a heavy-tailed Zipfian distribution [34], only a small memory capacity is needed.

Preliminary Validation. To validate our selection of memory capacities, we run a microbenchmark where we vary the eligibility ratio and measure the resulting latency in BurScale. In this microbenchmark, network bandwidth is the constraining resource, and we use a fixed arrival rate of 15,000 requests per second, which results in $k = 4$. For the purposes of this microbenchmark, we experiment with all three combinations of regular and burstable instances: (3 regular, 1 burstable), (2 regular, 2 burstable), and (1 regular, 3 burstable). We adjust the cache in the burstable instances to only cache a hot subset of the items so as to achieve a given eligibility ratio. Figure 4 depicts the resulting average latency as a function of the eligibility ratio. With lower eligibility ratios, the burstable instances are unable to serve the colder items, and thus the regular instances experience greater queueing and latency as a result. The vertical dashed line in each graph indicates the $\frac{k-R}{k}$ ratio. As can be seen in these graphs, when the eligibility ratio is greater than $\frac{k-R}{k} + 0.1$, the burstable instances have enough items cached to serve a sufficient fraction of the traffic and maintain low latencies.

3.4 Cost-Effective Flash Crowd Handling

In this section, we show how BurScale yields even greater cost savings when a user is provisioning for a flash crowd, an unpredictable sudden increase in load. Flash crowds present a significant challenge for autoscaling systems as there is a delay in acquiring new resources [11, 13] during which performance suffers. Fundamentally, there needs to be hot spare capacity that is available for immediate use when a flash crowd occurs. *Our key insight is to use burstable instances for this hot spare capacity to mask the delay of starting and adding new regular instances.*

A common approach to handling flash crowds is to overprovision by a factor m to mask the VM startup delay. For example, Netflix in the Nimble project [28] uses spare capacity (so called shadow resources) to handle the flash crowds that are caused by datacenter failures. That is, when a datacenter fails for whatever reason, they re-route the traffic to another datacenter, which is a type of flash crowd for the target datacenter. So the shadow resources act as an overprovisioning mechanism for this type of flash crowd. Like other autoscaling solutions [28], BurScale overprovisions resources to mask the VM startup delay. However, it does so more cost-effectively by using burstable instances for the overprovisioning.

As long as load does not increase by more than a factor m within the time period during which the system detects and incorporates new resources in response to a flash crowd, there is sufficient capacity to meet the SLO. Figure 5 illustrates this point when the load changes by a factor of $m = 1.75$ in a short time period. Provisioning for an arrival rate of λ in the top figure leads to periods of underprovisioning since the autoscaling cannot react fast enough to the increase in load. Provisioning for an arrival rate of $m\lambda$, on the other hand, can properly handle the flash crowd. BurScale uses the latter approach with an appropriately chosen m for handling flash crowds and hence does not suffer from underprovisioning in Figure 5.

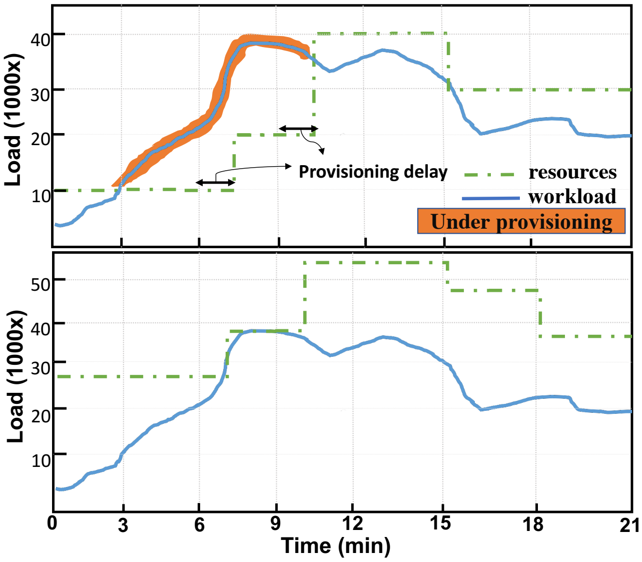

Figure 5: The effect of underprovisioning for flash crowds (top) and overprovisioning to handle flash crowds (bottom). BurScale overprovisions resources to mask the provisioning delay, but does so using cost-effective burstable instances.

Our work takes m as an input parameter and investigates *how to overprovision by a factor m in a cheaper fashion.* Note that selecting m is fundamentally a business decision based on the magnitude of a flash crowd that one is willing to spend money to overprovision for. Consequently, our results in section 5 demonstrate that BurScale works in various scenarios with different values of m.

Mathematically, to provision for an arrival rate of λ using the SR rule, we would use $k_\lambda = R + c\sqrt{R}$ instances with R regular instances and $c\sqrt{R}$ burstable instances. Provisioning for $m\lambda$ would consequently use $k_{m\lambda} = mR + c\sqrt{mR}$ instances with mR regular instances and $c\sqrt{mR}$ burstable instances.

We propose to utilize R rather than mR regular instances and $(k_{m\lambda} - R)$ burstable instances. In making this change, burstable instances are now responsible for handling the initial onset of a flash crowd. This introduces a new problem where the *JSQ* dispatching policy needs to be weighted such that the burstable CPU credits are not exhausted during normal operation, but are weighted during flash crowds to take advantage of the burst capability of burstable instances. We address this problem by designing BurScale to detect flash crowds and dynamically change the weights of burstable instances during a flash crowd (details in section 4). That is, in a flash crowd, all the regular and burstable instances will be set to the same weight whereas during normal operation, burstable instances will be weighted to use them sparingly. Thus, burstable instances will collect CPU credits during non-flash crowd times so that they

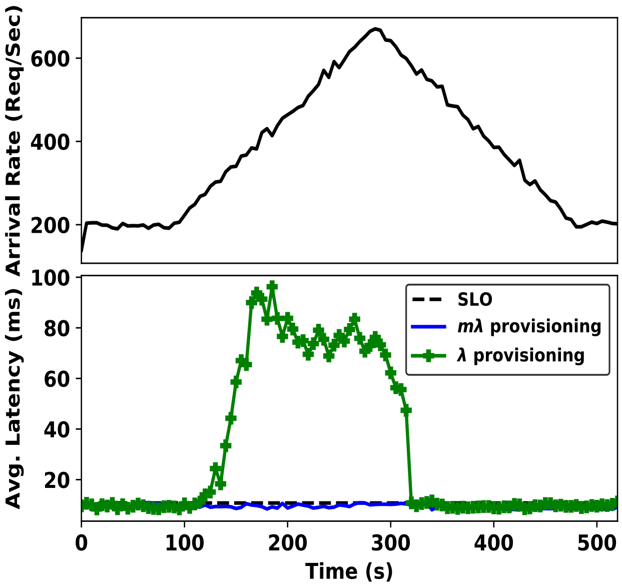

Figure 6: Overprovisioning is necessary to maintain low latency (bottom) during flash crowds (top). Even when overprovisioning with burstable instances, the average latency is below the desired dotted SLO line.

will have enough credits to use to cover the autoscaling delay when new resources are acquired for flash crowds.

Preliminary Validation. Figure 6 shows the effect of a flash crowd on a web application's performance (bottom graph) using a synthetic trace of a flash crowd (top graph). With normal provisioning for $\lambda = 200$ reqs/sec, there is substantial performance degradation during the flash crowd. By contrast, provisioning for $m\lambda$ ($m = 2$) with BurScale yields latency below the dotted SLO line.

Cost Savings. From a cost savings point of view, we compare our approach to the case where all $k_{m\lambda}$ of the instances are regular instances. Using burstable instances (t3.small) as hot spare capacity yields substantial cost savings in comparison to overprovisioning with regular instances (m5.large). Figure 7 shows the amount of savings as a function of $k_{m\lambda}$ for $m = 1.25$ and $P_Q = 0.1$ under the current Amazon AWS prices. As we can see, BurScale saves up to 50% in costs for small cluster sizes and over 20% for larger cluster sizes. We also compare this to the case where BurScale does not have the flash crowd enhancement (i.e., uses mR regular instances and $(k_{m\lambda} - mR)$ burstable instances). As expected, cost savings are not nearly as high, pointing to the importance of using burstable instances for flash crowds[3].

4 BURSCALE IMPLEMENTATION

We have implemented BurScale [4] on Amazon AWS EC2. Figure 8 shows BurScale's high-level architecture.

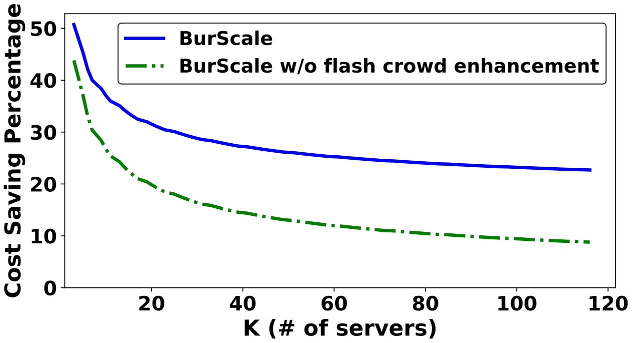

Figure 7: Percentage of cost saving using BurScale in flash crowd scenarios under an overprovisioning factor of $m = 1.25$.

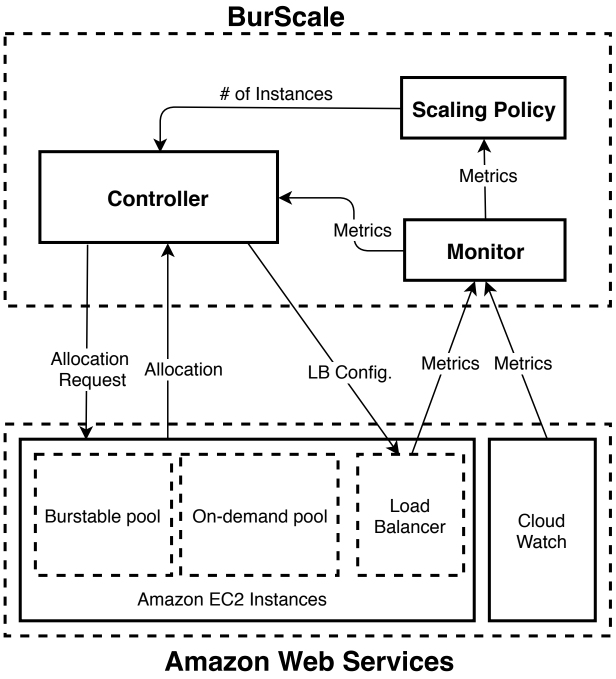

Figure 8: BurScale Architecture.

Scaling Policy. The *Scaling Policy* module answers the question of *how many resources does our application need?* To make a decision and find an answer to this question, it needs a model along with some inputs. As described in Section 3.1, BurScale uses the *Square Root Staffing* Rule model, which only needs the current request arrival rate to the system and the profiled service rate of the application as inputs. The *Monitor* module sends the current request arrival rate to the *Scaling Policy* module. The *Scaling Policy* module then determines the number of instances, k, that the application needs. *BurScale can adapt any other scaling policy including those based on* predictive models, learning-based models, etc. One can easily implement a scaling policy of their own choosing - all the

[3] While the percentage cost saving for BurScale without the flash crowd enhancement will approach 0 in the limit, BurScale will continue to provide savings even for very large clusters.

[4] The source code is available at: https://github.com/PSU-Cloud/BurScale

metrics that such a model may need as inputs are available from the *Monitor* module.

Monitor. The *Monitor* module periodically queries *AWS CloudWatch* [8] and the *Load Balancer.* It collects various metrics from the *CloudWatch* (e.g., CPU usage and credits) and from the load balancer (e.g., the current request arrival rate). It then provides these metrics to the *Scaling Policy* and the *Controller* modules.

Controller. The *Controller* module, which is the main component of BurScale, handles two primary tasks: (*i*) determining the number of regular and burstable instances based on the scaling policy, and (*ii*) adjusting the load balancer weights for the instances (LB Config).

The *Controller* module keeps track of the state of current resources that are used. The state is maintained by having two pools of resources: an regular pool and a burstable pool along with their properties, such as their weights for the request dispatching policy and their state (e.g., available, unavailable, etc). This module maintains a total of k instances, where k provided by the *Scaling Policy* module. Based on the current arrival rate, the controller calculates R, the minimum number of instances to keep the system stable. It then will procure R many regular instances and $(k - R)$ burstable instances. To maintain these numbers of instances, the controller will allocate and deallocate instances from the cloud provider and update the load balancer appropriately.

Flash Crowd Detection. The controller also detects flash crowds by maintaining a history of changes in the number of resources. If the changes in the number of servers is increasing in 3 consequent time windows (30 seconds each), the controller identifies it as a potential flash crowd. As the penalty for accidentally detecting a flash crowd is negligible (only a few extra burtable CPU credits would be spent), we opt for a simple mechanism that is not meant to be perfect. During a flash crowd, the controller will update the burstable weights in the load balancer dispatching policy so that the burstable instances will be utilized as much as the on-demand instances. During a non-flash crowd scenario, the controller will set the burstable instances weights so that the load balancer will only utilize them with sufficient queueing. This prevents burstable instances from running out of credits. The controller periodically gets the CPU usage of burstable instances from the *Monitor* module and decreases their weights until the CPU usage of the burstable instances is equal or less than the baseline rate allocated to them (40% for $t3.small$ instances in our experiments). This is because burstable instances start accumulating their CPU credits only when their CPU usage does not exceed the baseline rate.

5 EVALUATION

We evaluate BurScale's performance using real-world applications and traces. We consider two case studies. First, we use the Wikimedia application, which is used for the Wikipedia website, along with a recent dump of the Wikipedia database of English articles. Second, we extend our ideas to the distributed key-value caching application Memcached. Our results show that BurScale performs well for these real-world applications - it is able to procure resources from EC2 cost-effectively and manage them such that the desired SLOs (in terms of mean latency) are met. We note that BurScale does not provide any guarantee on tail latency. Provisioning for tail latency may require selecting a different value for k. However, BurScale is primarily addressing replacing some of the regular instances from the k total instances with burstable instances, and we experimentally show that BurScaleâĂŹs tail latency is comparable to the case where we use only regular instances.

5.1 Web Application Case Study

Experimental Setup. We use Wikimedia $v1.31.0$ [42] as our web application. Wikimedia is a PHP-based application that is used by the Wikimedia foundation for operating the Wikipedia website. We use a dump of Wikipedia English articles from March 2018 [41] for populating the database tier of this application. To run a workload against the application, we use wikibench [36], a widely used benchamrking tool for the Wikimedia application. We use $m5.large$ instances with 2 vCPUs and 8 GB of memory each as our on-demand/regular instances and $t3.small$ instances with 2 vCPUs (with peak and sustainable rates of 100% and 20% of each vCPU respectively) and 2 GB of memory each as our burstable instances. As of May 2019, the unit price for $m5.large$ and $t3.small$ instances is $0.096 and $0.0208 per hour respectively. Our experiments use 20-70 instances, and are performed in the `us-east-2` AWS region.

Workload. For our workload, we use the most recent publicly available Wikipedia access traces from September 2007 [34]. We are unaware of other more recent public traces that provide fine-grain timing information (millisecond) along with request accesses (i.e., urls) and the application (i.e., Wikimedia) to execute the requests. The traces that we use are from September 19, 2007 in the access traces. Finally, the SLO target is set to 200ms.

5.1.1 *Handling Transient Queueing.*

To evaluate BurScale in typical operating conditions, we replay 2.5 hours of the trace where the arrival rate increases by 50% during the trace. Figure 9(a) depicts the request arrival rate, which is scaled to fit our AWS cluster.

Meeting the SLO. We compare the performance of BurScale with *Reg-Only*, a baseline that only uses regular instances. As seen in Figure 9(b) BurScale is able to meet the SLO with performance comparable to *Reg-Only*, even though the burstable instances are lightly utilized for the transient queueing that develops during normal operating conditions.

Effect on Tail Latency. Figure 9(c) depicts the *95th* percentile of observed latency for BurScale and the *Reg-Only* baseline. BurScale has a comparable tail latency to *Reg-Only* while using cheaper burstable instances.

Cost Savings. Figure 9(d) depicts the total number of instances used throughout the experiment along with a break down of instances (i.e., regular and burstable) that are used. Since burstable instances are cheaper than regular instances, BurScale is able to save 16.8% in costs from this Figure 9 experiment.

How Burstable Instances are Used. Figure 9(e) depicts the CPU utilization of regular and burstable instances during the experiment. Since BurScale is operating under normal conditions without flash crowds, it sets the weights of burstable instances such that they accumulate credits. Hence, their CPU utilization must be less than their credit accumulation rate of 40%. Initially, their weights are equal to the regular instances, and BurScale dynamically updates the weights so that the CPU utilization is slightly under 40%. During the time that the weights are stabilizing, the CPU utilization is high, and the burstable instances consume some of their credits.

As mentioned, one of the challenges of using burstable instances is that they can run out of credits and hence become less effective in serving the traffic. Therefore, our solution is to use them in a way where they slowly accumulate credits in normal operation so that they can be used in flash crowd situations. Figure 9(f) shows the credit state of burstable instances where *t3.small* instances start with 60 initial credits. As it can be seen, BurScale is able to set the weights so that the burstable instances start accumulating credits. The initial decrease in credits is because initially instances start with equal weights and BurScale updates the burstable weights so that their CPU utilization is low enough to accumulate credits.

5.1.2 Handling Flash Crowds. Next, we evaluate BurScale's handling of a flash crowd by artificially inducing a sudden surge in traffic at $t = 30$ minutes. As seen in Figure 10(a), the arrival rate suddenly increases by about a factor of 3, corresponding to a flash crowd with $m = 3$.

Meeting the SLO. Since resources are overprovisioned, BurScale is able to utilize the burstable instances at peak utilization during the flash crowd, thereby meeting the SLO as seen in Figure 10(b). As the *Reg-Only* baseline performs the same actions except with regular instances, it is also able to meet the SLO, but at a higher cost.

Effect on Tail Latency. Figure 10(c) depicts the *95th* percentile of observed latency for BurScale and the *Reg-Only* baseline. BurScale has a comparable tail latency to *Reg-Only*, which indicates that swapping some regular instances with lightly utilized burstable instances is acceptable for performance.

Cost Savings. Figure 10(d) depicts the total number of instances used throughout the experiment along with a break down of instances (i.e., regular and burstable) that are used. We calculate the total cost savings from Figure 10 to be 46.3%.

How Burstable Instances are Used. As seen in Figure 10(d), BurScale's controller identifies the increased traffic at $t = 30$ minutes as a flash crowd and initiates the procurement of additional instances. During this time, the load balancer weights for burstable instances are set to the same as regular instances so that the burstable instances can effectively assist in the initial flash crowd rush. As can be seen in Figure 10(e), the CPU utilization of burstable instances increases during the initial flash crowd onset, during which new instances are being initialized and warmed up. Figure 10(f) depicts the credit state of the burstable instances. Initially the weights are equal and hence we see a decrease in the number of credits. BurScale periodically updates the weights so that burstable instances start slowly accumulating credits, and hence we see the increase in the credits in this graph. Once the flash crowd is detected at $t = 30$ minutes, BurScale sets the burstable and regular instance weights to be equal so that the burstable instances can be fully utilized to serve the traffic, and hence we see a significant decrease in the number of credits. When the new regular instances are added and warmed up in the autoscaling process at $t = 40$ minutes, BurScale updates the weights so that burstable instances start slowly accumulating credits again.

5.1.3 Sensitivity: Number of Regular vs. Burstable Instances. Given k, the total number of desired instances, BurScale assigns R regular instances and $k - R$ burstable instances, where $R = \lambda/\mu$, λ is the measured arrival rate, and μ is the measured service rate.

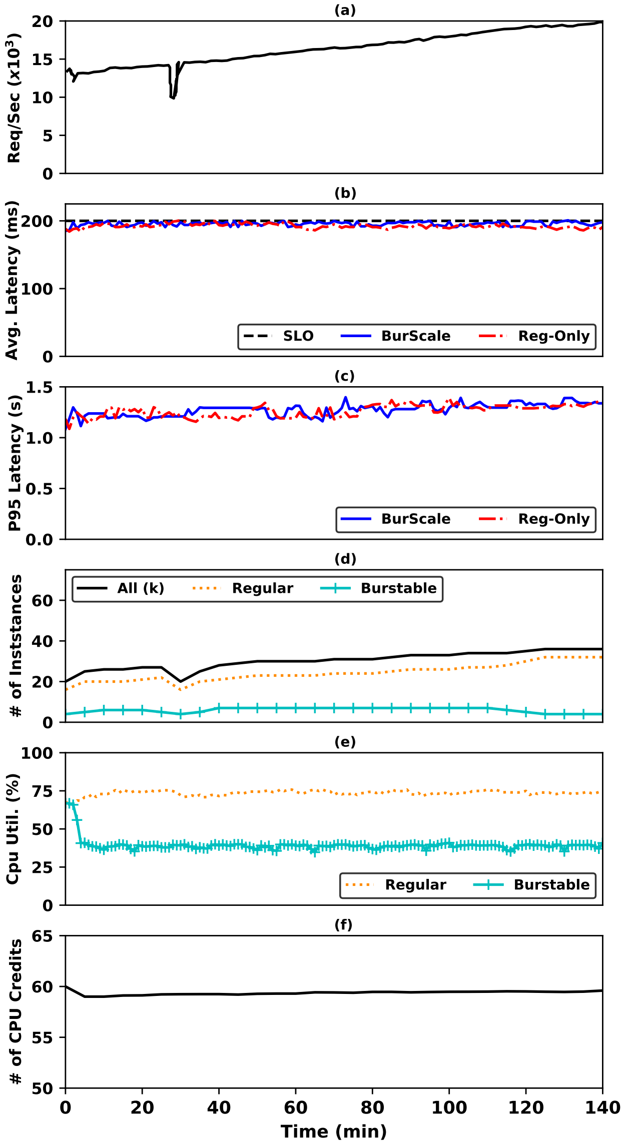

Figure 9: Results for the transient queueing experiment (details in section 5.1.1) using 2.5 hours of the Wikipedia trace (a). Switching some of the instances to burstable instances (d) results in 16.8% cost savings while matching the performance of using only regular instances (b & c). The burstable instances' CPU credits (f) are not exhausted as the load balancer avoids extensively utilizing the burstable instances (e).

We choose R based on intuition from queueing theory, but in this section, we evaluate the sensitivity of the split between regular and burstable instances. We run an experiment with the same workload as in Section 5.1.1 except that we vary the number of regular and burstable instances while keeping the total number k the same (e.g., R - 1 regular and k - R + 1 burstable instances). Figure 11 depicts the number of SLO violations for different choices for the number

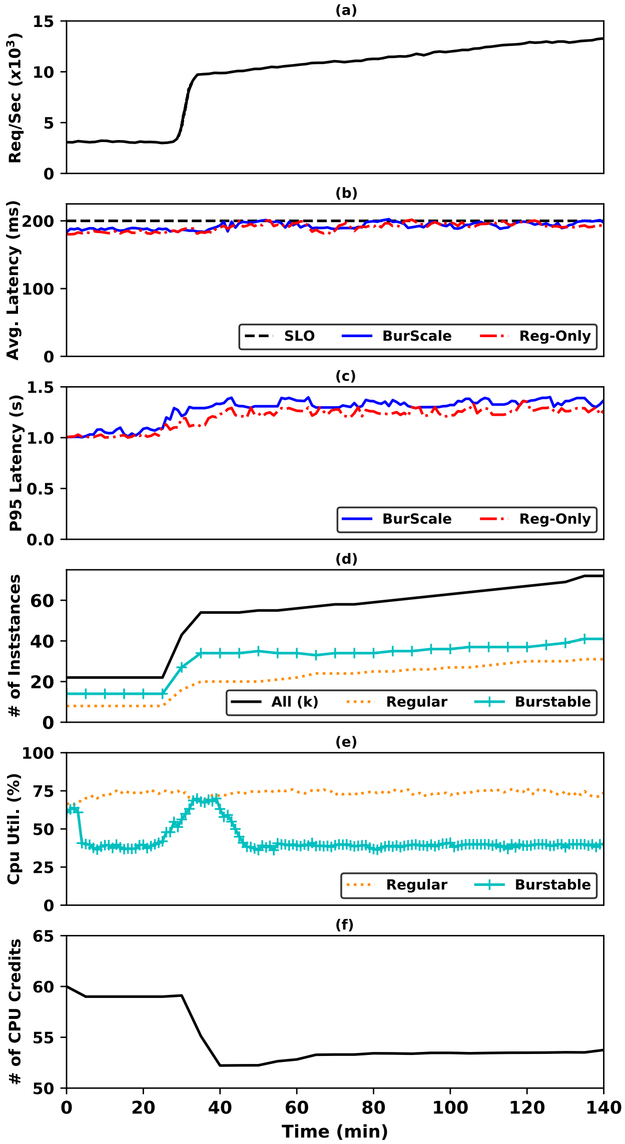

Figure 10: Results for the flash crowd experiment (details in section 5.1.2) with a flash crowd induced at $t = 30$ minutes (a). BurScale handles the flash crowd by overprovisioning using burstable instances (d), resulting in a 46.3% cost savings over using only regular instances while matching performance (b & c). The burstable instances' CPU credits (f) are only significantly utilized during the onset of the flash crowd where BurScale configures the load balancer to fully utilize the burstable instances (e).

of regular instances. We count SLO violations by identifying how many 1-minute long intervals have an average latency exceeding the SLO out of a total of 143 intervals. As expected, provisioning more than R regular instances will work, but will be more expensive. More interestingly, we find that although provisioning slightly

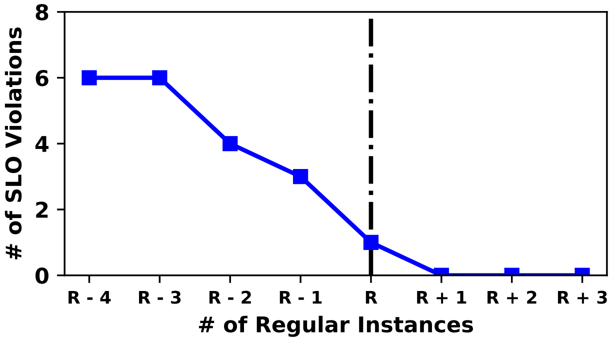

Figure 11: Sensitivity to the number of regular and burstable instances. We vary the number of regular (x-axis) and burstable instances while maintaining a fixed total of k instances. We measure the number of 1-minute intervals where the latency exceeds the target SLO (y-axis). This experimentally demonstrates that R, as defined in Theorem 1, is a good choice for the number of regular instances.

fewer than R regular instances results in some periods of SLO violation, the number of violations is low (below 5%). Thus, one could provision fewer regular instances if cost is more important than SLO violations. The extreme case of using only burstable instances would result in running out of CPU credits and substantially more SLO violations. Furthermore, if a burstable only cluster were provisioned to not run out of credits, then the cost would be higher than using regular instances as described in subsection 2.1.

5.2 Distributed Key-Value Cache Case Study

Experimental Setup. We deploy a web application that spans eight instances and uses a Memcached based cluster for its caching tier. We use the same $m5.large$ and $t3.small$ instances as before.
Workload. We use ETC, a read-dominant trace from Atikoglu et al. [5], for generating our workload. Figure 12(a) depicts its request arrival rate that we use in our evaluation. We set the working set size to be 3 GB. As the available memory capacity within our regular instances is more than the working set size, the hit rate is always 100% (ignoring capacity misses). Thus, the purpose of replicating instances in this experiment is for the ability to handle a high rate of cache traffic. We use a Zipf distribution for key popularity wherein 20% of the whole key set (i.e., 600 MB) are "hot" and account for 90% of the overall requests. We choose an average latency SLO of 5 ms.
Baselines. We compare BurScale's performance against two baselines. First, *RegRep* reactively procures additional capacity, but uses only regular instances and does not overprovision. Second, *RegOver* behaves similarly to *RegRep* except that it overprovisions resources. BurScale overprovisions just like *RegOver*, except it does so with burstable instances.

As the regular instances have enough memory to hold the working set, all regular instances replicate the entire working set. Thus, additional instances are not added for increasing the hit ratio, but rather the instances are used for serving the high rate of traffic.

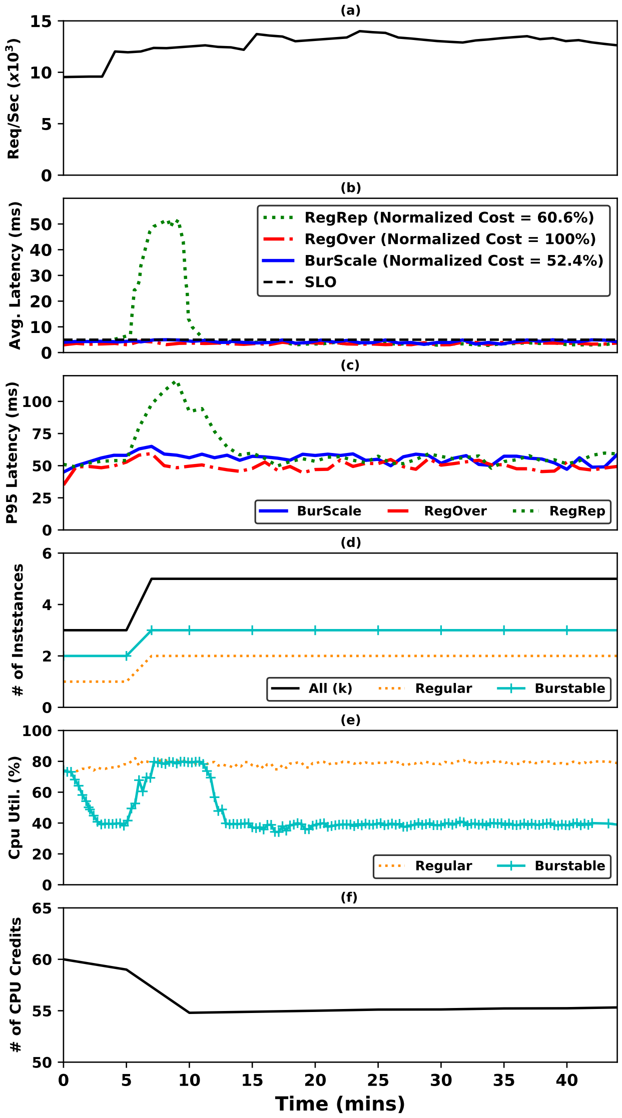

Figure 12: Results for the Memcached experiment (details in section 5.2) using the ETC trace (a) from [5]. Using burstable instances (d) results in a 47.6% cost savings compared to overprovisioning using only regular instances (RegOver). Even when the burstable instances cannot cache all the items, BurScale is able to match the performance of RegOver (b & c). Without overprovisioning (RegRep), latency can significantly increase (b & c) during periods of autoscaling (t = 5 minutes in d). During these periods, the burstable instances' CPU credits are significantly utilized (e & f).

Burstable instances, on the other hand, do not have enough memory for the entire dataset, so they are only configured to contain a hot subset of the data.

Meeting the SLO. Figure 12(b) depicts the mean latency of BurScale and the other two baselines. After a fast increase in the arrival rate, the instances in *RegRep* get overloaded and result in high latency during the time when new resources are added and warmed up. By contrast, both *RegOver* and BurScale overprovision resources and are able to handle the fast increase. It is important to note that BurScale overprovisions using burstable instances, which only have enough memory to cache the hot content. Thus, these results show that even if a hot subset of data can be cached, the burstable instances can handle enough of the traffic to be useful.

Effect on Tail Latency. Figure 12(c) depicts the 95th percentile of latency for BurScale and the other two baselines. It can be seen that BurScale has a comparable tail latency with the *RegOver* baseline.

Cost Savings. Figure 12(d) shows the number and type of instances used by BurScale. As BurScale uses predominantly burstable instances in this experiment, it only spent 52.4% the cost of *RegOver*, which equates to a 47.6% savings. By contrast, *RegRep* spent 60.6% the cost of *RegOver* while still suffering from SLO violations. The reason for this is that BurScale achieves cost savings by using burstable instances for both (i) overprovisioning and (ii) transient queueing. By contrast, *RegRep* only saves costs by eliminating the overprovisioning.

How Burstables are Used. Figure 12(e) and Figure 12(f) show the CPU utilization and CPU credits of the burstable instances respectively. During the increase in traffic at around t = 5 minutes, the load balancer weights are adjusted to utilize the burstable instances more frequently while new resources are added to the cluster. This corresponds to the increase in burstable CPU utilization and decrease in CPU credits. Once the new resources have been added to the cluster, the burstable instances revert to just helping with the transient queueing that normally occurs, and hence they start slowly accumulating credits as it can be seen in Figure 12(f).

6 RELATED WORK

We classify the related work along the following themes: (i) cost-effective resource scaling, (ii) exploiting burstable instances, and (iii) exploiting different cloud products/services in general.

Cost-effective Resource Scaling. Autoscaling has been extensively studied in the past decade with much research on reactive [10, 12], predictive [13, 14, 20, 31, 35], and hybrid [30, 35] approaches. Major cloud providers such as AWS, Google Cloud Engine, and Mircosoft Azure offer rule-based autoscaling systems [2, 6, 7]. For example, the AWS AutoScale service [2] autoscales resources based on tenant-defined rules. An example of a commercial entity configuring such a public cloud autoscaler is Spotify's BigTable Autoscaler [33]. This autoscaler keeps the cluster size within a preselected range while keeping the average CPU utilization at 70%. BurScale can be made to work with these autoscaling approaches by swapping a subset of the instances with burstable instances. Given the number of total instances, k, from the autoscaling policy, BurScale measures the load and system performance to determine the number of burstable instances to use. It then adjusts the load balancer to avoid overutilizing the burstable instances.

While the body of the related work focuses on determining the amount of resources (e.g., k in our case), the key novelty of BurScale is introducing a new technique for incorporating burstable instances into autoscaling systems while considering of cost-efficiency and performance.

Characterizing and Exploiting Burstable Instances. Wang, C. et al. [38] present a study of the token-bucket mechanisms that governs the dynamism in CPU capacity and network bandwidth of AWS burstable instances. Jiang et al. [21] present a unified analytical model for evaluating burstable services. Such a model enables tenants to optimize their usage and also allows cloud providers to maximize their revenue from burstable offerings. BurScale goes beyond theoretical analysis and proposes a practical approach for utilizing burstable instances in multiple use cases including flash crowd provisioning. Wang, C. et al. [37] explore two cases where using burstable instances leads to cost savings. First, it uses burstable instances for backup of popular Memcached content stored on cheap but revocation-prone spot instances. This helps mitigate performance degradation during the period following a spot revocation while its replacement is being procured. Second, it shows how multiplexing multiple burstable instances over time can sometimes offer CPU capacity and network bandwidth equivalent to a regular instance at a lower price. BurScale builds upon this work and demonstrates how burstable instances can benefit more general use cases such as web applications and distributed key-value stores.

Our work describes completely novel use cases for burstable instances that can bring new cost saving opportunities that are complementary to some of these earlier works.

Combining Different Cloud Products and Services. Several works have exploited the diversity of instance types to achieve cost-effective resource provisioning, and we discuss a subset here. In particular, researchers have successfully exploited the low prices of revocable spot instances [1, 18, 37] and reserved instances [40]. BurScale is complementary to these approaches since there are burstable instance types within the spot and reserved instance price models (i.e., it is possible to purchase a spot/reserved burstable instance). One would first use BurScale to partition the total number of instances between burstable and regular instance types. Then, one would apply these related works twice, once for provisioning the set of burstable instances and once for the regular instances, to determine the number of reserved/spot burstable instances and reserved/spot regular instances.

There are also systems that use complex optimizations to provision resources. For example, Kingfisher [32] is a cost-aware provisioning system that uses a combination of different instance types (although not burstable instances) using an integer linear programming model. Once a system such as Kingfisher decides the number of instances to provision of a given type, BurScale can then exchange some of them for corresponding burstable instances to reduce costs.

Comparison with Lambdas. Recent work has considered exploiting the agility of serverless offerings such as AWS lambdas for helping mask the latency of launching new VMs [29]. Burstable instances as used in BurScale are available nearly instantly whereas even warm-start lambdas take about 100 ms; cold-start lambdas may take a few minutes just like VMs [39]. BurScale requires minimal to no changes to the application (recall our changes were limited to only the load balancer) whereas using lambdas requires more extensive changes. In fact, some applications may not be portable with lambdas in their current restrictive form [19, 22]. Among the most prominent restrictions are their limited lifetime of a few minutes and their inability to engage in point-to-point communication (for state transfer) with other lambdas or from other VMs, which in turns necessitates the use of a slow external storage medium such as AWS S3.

7 CONCLUSION

The recently introduced burstable instances in the public cloud are advertised as being suitable for tenants with low intensity and intermittent workloads. By contrast, this paper presented two novel usage scenarios for burstable instances in larger clusters with sustained usage. We demonstrate (i) how burstable instances can be utilized alongside conventional instances to handle the transient queueing arising from variability in traffic, and (ii) how burstable instances can mask the VM startup/warmup time when autoscaling to handle flash crowds. We implement our ideas in a system called BurScale and use it to demonstrate cost-effective autoscaling for two important workloads: (i) a stateless web server cluster, and (ii) a stateful Memcached in-memory caching cluster. Results from our prototype system show that via its careful combination of burstable and regular instances along with a dynamically adapting weighted *JSQ* request dispatching policy, BurScale is able to meet latency SLOs while reducing costs by up to 50% compared to traditional autoscaling systems that only use regular instances.

ACKNOWLEDGEMENT

This research is supported by the National Science Foundation under Grant No. CNS-1717571 and an AWS research credit award. We also thank Sudipto Das for helping shepherd our paper.

REFERENCES

[1] A. Ali-Eldin, J. Westin, B. Wang, P. Sharma, and P. Shenoy. Spotweb: Running latency-sensitive distributed web services on transient cloud servers. In *Proceedings of the 28th International Symposium on High-Performance Parallel and Distributed Computing (HPDC)*. ACM, 2019.

[2] Amazon aws autoscaling service. https://aws.amazon.com/autoscaling/. Accessed: 2017-10-02.

[3] Amazon aws t2 series. https://aws.amazon.com/ec2/instance-types/t2/. Accessed: 2017-12-30.

[4] Amazon aws t3 series. https://aws.amazon.com/ec2/instance-types/t3/. Accessed: 2018-08-24.

[5] B. Atikoglu, Y. Xu, E. Frachtenberg, S. Jiang, and M. Paleczny. Workload analysis of a large-scale key-value store. In *ACM SIGMETRICS Performance Evaluation Review*, volume 40, pages 53–64. ACM, 2012.

[6] Auto scaling on the google cloud platform. https://cloud.google.com/compute/docs/autoscaler/. Accessed: 2018-01-16.

[7] Auto scaling on the microsoft azure. https://docs.microsoft.com/en-us/azure/cloud-services/cloud-services-how-to-scale-portal. Accessed: 2018-01-22.

[8] Aws cloudwatch. https://aws.amazon.com/cloudwatch/. Accessed: 2019-01-02.

[9] D. S. Berger, B. Berg, T. Zhu, S. Sen, and M. Harchol-Balter. Robinhood: Tail latency aware caching – dynamic reallocation from cache-rich to cache-poor. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 195–212, Carlsbad, CA, 2018. USENIX Association.

[10] F. Cruz, F. Maia, M. Matos, R. Oliveira, J. Paulo, J. Pereira, and R. Vilaça. Met: workload aware elasticity for nosql. In *Proceedings of the 8th ACM European Conference on Computer Systems*, pages 183–196. ACM, 2013.

[11] A. Gandhi, P. Dube, A. Karve, A. Kochut, and L. Zhang. Model-driven optimal resource scaling in cloud. *Software & Systems Modeling*, 17(2):509–526, 2018.

[12] A. Gandhi, M. Harchol-Balter, R. Raghunathan, and M. A. Kozuch. Autoscale: Dynamic, robust capacity management for multi-tier data centers. *ACM Transactions on Computer Systems (TOCS)*, 30(4):14, 2012.

[13] A. Gandhi, T. Zhu, M. Harchol-Balter, and M. A. Kozuch. Softscale: stealing opportunistically for transient scaling. In *ACM/IFIP/USENIX International Conference on Distributed Systems Platforms and Open Distributed Processing*, pages 142–163. Springer, 2012.

[14] Z. Gong, X. Gu, and J. Wilkes. Press: Predictive elastic resource scaling for cloud systems. *CNSM*, 10:9–16, 2010.

[15] Google cloud engine micro instances. https://cloud.google.com/compute/docs/machine-types#sharedcore. Accessed: 2018-01-13.

[16] U. U. Hafeez, M. Wajahat, and A. Gandhi. Elmem: Towards an elastic memcached system. In *2018 IEEE 38th International Conference on Distributed Computing Systems (ICDCS)*, pages 278–289. IEEE, 2018.

[17] M. Harchol-Balter. *Performance modeling and design of computer systems: queueing theory in action.* Cambridge University Press, 2013.

[18] A. Harlap, A. Chung, A. Tumanov, G. R. Ganger, and P. B. Gibbons. Tributary: spot-dancing for elastic services with latency slos. In *2018 USENIX Annual Technical Conference (USENIX ATC 18).* USENIX Association, 2018.

[19] J. M. Hellerstein, J. Faleiro, J. E. Gonzalez, J. Schleier-Smith, V. Sreekanti, A. Tumanov, and C. Wu. Serverless computing: One step forward, two steps back. *arXiv preprint arXiv:1812.03651*, 2018.

[20] J. Jiang, J. Lu, G. Zhang, and G. Long. Optimal cloud resource auto-scaling for web applications. In *Cluster, Cloud and Grid Computing (CCGrid), 2013 13th IEEE/ACM International Symposium on*, pages 58–65. IEEE, 2013.

[21] Y. Jiang, M. Shahrad, D. Wentzlaff, D. H. Tsang, and C. Joe-Wong. Burstable instances for clouds: Performance modeling, equilibrium analysis, and revenue maximization. In *Proc. IEEE INFOCOM*, 2019.

[22] E. Jonas, J. Schleier-Smith, V. Sreekanti, C.-C. Tsai, A. Khandelwal, Q. Pu, V. Shankar, J. Carreira, K. Krauth, N. Yadwadkar, et al. Cloud programming simplified: A berkeley view on serverless computing. *arXiv preprint arXiv:1902.03383*, 2019.

[23] H.-C. Lin and C. S. Raghavendra. An analysis of the join the shortest queue (jsq) policy. In *[1992] Proceedings of the 12th International Conference on Distributed Computing Systems*, pages 362–366. IEEE, 1992.

[24] mcrouter prefix routing. https://github.com/facebook/mcrouter/wiki/Routing-Prefix. Accessed: 2019-01-03.

[25] Memcached. https://memcached.org/. Accessed: 2018-06-03.

[26] Microsoft azure b series. https://docs.microsoft.com/en-us/azure/virtual-machines/windows/b-series-burstable. Accessed: 2018-01-13.

[27] R. D. Nelson and T. K. Philips. *An approximation to the response time for shortest queue routing*, volume 17. ACM, 1989.

[28] Netflix project nimble. https://bit.ly/31l4KoF. Accessed: 2019-04-02.

[29] J. H. Novak, S. K. Kasera, and R. Stutsman. Cloud functions for fast and robust resource auto-scaling. In *2019 11th International Conference on Communication Systems & Networks (COMSNETS)*, pages 133–140. IEEE, 2019.

[30] P. Padala, K.-Y. Hou, K. G. Shin, X. Zhu, M. Uysal, Z. Wang, S. Singhal, and A. Merchant. Automated control of multiple virtualized resources. In *Proceedings of the 4th ACM European conference on Computer systems*, pages 13–26. ACM, 2009.

[31] N. Roy, A. Dubey, and A. Gokhale. Efficient autoscaling in the cloud using predictive models for workload forecasting. In *Cloud Computing (CLOUD), 2011 IEEE International Conference on*, pages 500–507. IEEE, 2011.

[32] U. Sharma, P. Shenoy, S. Sahu, and A. Shaikh. A cost-aware elasticity provisioning system for the cloud. In *2011 31st International Conference on Distributed Computing Systems*, pages 559–570. IEEE, 2011.

[33] Spotify big table autoscaler. https://github.com/spotify/bigtable-autoscaler. Accessed: 2019-01-02.

[34] G. Urdaneta, G. Pierre, and M. van Steen. Wikipedia workload analysis for decentralized hosting. *Elsevier Computer Networks*, 53(11):1830–1845, July 2009. http://www.globule.org/publi/WWADH_comnet2009.html.

[35] B. Urgaonkar, G. Pacifici, P. Shenoy, M. Spreitzer, and A. Tantawi. An analytical model for multi-tier internet services and its applications. In *ACM SIGMETRICS Performance Evaluation Review*, volume 33, pages 291–302. ACM, 2005.

[36] E.-J. van Baaren. Wikibench: A distributed, wikipedia based web application benchmark. *Master's thesis, VU University Amsterdam*, 2009.

[37] C. Wang, B. Urgaonkar, A. Gupta, G. Kesidis, and Q. Liang. Exploiting spot and burstable instances for improving the cost-efficacy of in-memory caches on the public cloud. In *Proceedings of the Twelfth European Conference on Computer Systems*, pages 620–634. ACM, 2017.

[38] C. Wang, B. Urgaonkar, N. Nasiriani, and G. Kesidis. Using burstable instances in the public cloud: Why, when and how? *Proc. ACM Meas. Anal. Comput. Syst.*, 1(1):11:1–11:28, June 2017.

[39] L. Wang, M. Li, Y. Zhang, T. Ristenpart, and M. Swift. Peeking behind the curtains of serverless platforms. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 133–146, Boston, MA, 2018. USENIX Association.

[40] W. Wang, B. Li, and B. Liang. To reserve or not to reserve: Optimal online multi-instance acquisition in iaas clouds. In *ICAC*, pages 13–22, 2013.

[41] Wikimedia database dump. https://archive.org/details/enwiki-20180320. Accessed: 2018-05-17.

[42] Wikimedia software. https://www.mediawiki.org/wiki/MediaWiki. Accessed: 2018-05-17.

[43] T. Zhu, A. Gandhi, M. Harchol-Balter, and M. A. Kozuch. Saving cash by using less cache. In *Proceedings of the 4th USENIX Conference on Hot Topics in Cloud Ccomputing*, HotCloud'12, pages 3–3, Berkeley, CA, USA, 2012. USENIX Association.