AutoBurst: Autoscaling Burstable Instances for Cost-effective Latency SLOs

Rubaba Hasan
The Pennsylvania State University
rxh655@psu.edu

Timothy Zhu
The Pennsylvania State University
tuz68@psu.edu

Bhuvan Urgaonkar
The Pennsylvania State University
bhuvan@cse.psu.edu

ABSTRACT

Burstable instances provide a low-cost option for consumers using the public cloud, but they come with significant resource limitations. They can be viewed as "fractional instances" where one receives a fraction of the compute and memory capacity at a fraction of the cost of regular instances. The fractional compute is achieved via rate limiting, where a unique characteristic of the rate limiting is that it allows for the CPU to burst to 100% utilization for limited periods of time. Prior research has shown how this ability to burst can be used to serve specific roles such as a cache backup and handling flash crowds. Our work provides a general-purpose approach to meeting latency SLOs via this burst capability while optimizing for cost. AutoBurst is able to achieve this by controlling both the number of burstable and regular instances along with how/when they are used. Evaluations show that our system is able to reduce cost by up to 25% over the state-of-the-art while maintaining latency SLOs.

CCS CONCEPTS

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

KEYWORDS

Autoscaling, Burstable Instances, Resource Provisioning, Cloud Computing

ACM Reference Format:
Rubaba Hasan, Timothy Zhu, and Bhuvan Urgaonkar. 2024. AutoBurst: Autoscaling Burstable Instances for Cost-effective Latency SLOs. In *ACM Symposium on Cloud Computing (SoCC '24), November 20–22, 2024, Redmond, WA, USA.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3698038.3698530

1 INTRODUCTION

Burstable instances are a low-cost tier of resources in the public cloud where compute is significantly rate limited. These instances use a token-bucket mechanism to accumulate credits, and the credits can be utilized to run the instances at full capacity (i.e., burst to full capacity) for limited periods of time. With cost savings sometimes exceeding 70%, multiple papers have studied and modeled their characteristics [19–21, 35] while other papers have designed techniques to take advantage of burstable instances for specific use cases such as spot instance revocation, transient load variation, flash crowds, etc. [1, 2, 4, 11, 29, 33, 34]. These works demonstrate the effectiveness of burstable instances in specific use cases, but in our work, we address how to make burstable instances more generally applicable such as for meeting latency Service Level Objective (SLO) goals. Trying to generally use burstable instances as a regular instance is challenging since they're rate limited. Using them all the time or for heavy compute workloads will quickly lead to hitting the rate limits, but when paired with regular instances, they can be an effective tool for reducing cost.

Our goal is to design an autoscaling system that balances both burstable and regular instance usage to minimize cost while meeting latency SLOs. The target of our work is an autoscaling system where a load balancer divides the incoming traffic among the instances. For example, a web application is a common workload where web requests are balanced across an autoscaled set of resources. We focus on user-facing workloads where there is a rate of incoming traffic that varies over time (i.e., open queueing workload) and latency is at the scale of hundreds of milliseconds. These workloads are challenging to manage because there is often significant short term variability that causes queueing and high latency. Fortunately, one of the advantages of burstable instances is the ability to burst their CPU usage for short time periods, giving them the flexibility to handle load spikes and autoscaling mispredictions. We demonstrate how to take advantage of this burst capability to build a robust and efficient autoscaling system.

Our work addresses two essential questions. First, we need to figure out how to balance the workload between burstable and regular instances. This is challenging because we need to balance saving burstable credits for the future while ensuring good performance by meeting user-specified latency SLOs. Second, we need to determine the number of burstable and regular instances to provision to ensure there are enough resources to handle the workload. This involves simultaneously autoscaling both types of resources while also designing the autoscaler to be compatible with the load balancing.

Most of the prior works on using burstable instances do not address these questions since they primarily focus on specific use cases and techniques where burstable instances are advantageous, such as dealing with spot revocation [33, 34]. Furthermore, we are unaware of prior work that meets user-configurable latency SLOs using the cost-effectiveness of burstable instances. BurScale [4] is the closest work where it balances the workload such that burstable instances are used for transient load spikes and flash crowds. While it can maintain low latency, it does not directly support a user-configurable latency SLO. So for workloads with relaxed SLOs, it suboptimally balances the workload between burstable and regular instances, leaving much room for cost optimization. Our research shows that by tuning the workload balance between burstable and regular instances, one can significantly reduce cost even over state-of-the-art solutions. Our key insight is that by tuning the workload balance at fine timescales, we can use the burst ability of burstable instances to adapt to sudden workload changes and maintain latency SLOs. This in turn makes the autoscaling problem easier since we do not need to worry as much about mispredictions and VM startup delays. Even with significant mispredictions in the number of instances, we can use the burst ability of burstable instances to make up the difference. This allows us to optimize the number of burstable and regular instances to minimize cost.

AutoBurst is our new tool that adopts this novel design for load balancing and autoscaling. Traditionally, autoscaling is the primary mechanism for controlling latency and meeting latency SLOs, but its adaptivity is low since it takes time (e.g., ½ minute to minutes) to add/remove resources from a system. In AutoBurst, we instead utilize the load balancing and the ability of burstable instances to burst CPU usage to quickly adapt to maintain latency SLOs. These changes can happen within seconds and quickly shift work to burstable instances when latency is high. Or when latency is low, it can divert work from burstable instances to conserve credits. Thus, we use a feedback control loop to adjust the load balancing to maintain the desired latency. Since latency is maintained by the load balancer, the autoscaler is then free to operate at longer timescales where it can ensure there are sufficient resources in the long term. Specifically, we use a feedback control loop to adjust the ratio of burstable and regular instances such that we maintain a total burstable credit level that is user configurable. Users could set the desired credit level based on how large/long of a burst they want to handle. This gives them the flexibility to save up credits or utilize them depending on their needs since the desired credit level can be dynamically configured over time.

Thus, our main novelty is separating the latency control and credit balance control so that latency can be adapted quickly via load balancing between burstable/regular instances, and the credit balance is controlled over longer time periods by adapting the number of instances. Through our design, AutoBurst provides flexibility in using burstable and regular instances, which results in significant cost savings while meeting user-configurable latency SLOs. AutoBurst has been deployed and evaluated in a real-world cloud environment (AWS). Our evaluation shows AutoBurst is able to achieve 12-25% cost savings even over BurScale [4], the state-of-the-art system for using burstable instances in autoscaling.

Contributions:

- We design a new system, AutoBurst, that jointly solves the load balancing and autoscaling problem using burstable instances and regular instances resulting in 12-25% cost savings over the state-of-the-art solution. The code is open-sourced at:
  https://github.com/rubabahasan/AutoBurst
- We demonstrate how the burst ability of burstable instances can be used to meet user-configurable latency SLOs, which is not supported in prior designs using burstable instances.
- We demonstrate how our solution takes advantage of burstable instances to increase the robustness to mispredictions in autoscaling. Even with 20-40% fewer resources than expected, AutoBurst is able to utilize the burstable credits to keep latency close to the SLO.

2 BACKGROUND AND RELATED WORKS

Many public cloud providers offer burstable instances, a type of low-cost instance designed for customers that utilize minimal CPU performance for the majority of the time with the capability to quickly ramp up to full capacity during brief bursts. This behavior is typically controlled by a token-bucket mechanism via credits that allow the instances to burst in CPU utilization. This model has been adopted by major cloud providers such as AWS and Azure (e.g., AWS's $t3$ instances and Azure's B-series).

The primary advantage of burstable instances is it can have much lower cost than regular on-demand instances. For example, at AWS, a $t3$ instance with 2 vCPUs has an hourly cost of \$0.0208 while an $m5$ instance with 2 vCPUs costs \$0.096, which translates to a 78.33% lower cost.

The primary downside is burstable instances cannot always get 100% CPU utilization since it is limited by a token-bucket rate limiter. Credits are accrued over time based on the specifications of the particular burstable instance type. Each type has a baseline utilization percentage, and if the CPU utilization is higher than the baseline, then the instance loses credits. If the CPU utilization is lower than the baseline, then it gains credit [6]. For AWS, the credit accrued in t minutes can be calculated using the following equations:

$$CreditEarned = BaselineUtilization \times vCPUcount \times t \quad (1)$$

$$CreditSpent = CPUutilization \times vCPUcount \times t \quad (2)$$

$$CreditAccrued = CreditEarned - CreditSpent \quad (3)$$

In other words, 1 credit is equivalent to 1 vCPU worth of work for 1 minute. Credits can be accumulated up to a limit, which is generally equal to the maximum amount of credits earned in 24 hours. Some older versions of burstable instances (e.g., $t2$ instances) included some initial startup credits when launching new instances, but newer generations (e.g., $t3$ instances) don't include such credits.

When an instance runs out of credits, then the cloud provider typically rate limits the CPU time so that the instance doesn't over-utilize the compute. Clouds can also offer an "unlimited" mode where instances can have a negative credit balance that would incur extra cost.

While the primary target of burstable instances is small customers with low CPU utilization, researchers have studied how burstable instances can be used for various other purposes. Several initial studies [21, 35] developed performance models for burstable instances. Later studies [19, 20] have proposed a unified framework to model the resource provisioning of burstable instances in a multi-user environment and offer insights for both cloud providers and tenants to optimize their expenses.

In addition to studying the performance characteristics of burstable instances, there have also been works that propose ways to take advantage of their low cost. BurScale [4] proposes the idea of using burstable instances for overprovisioning in a cost effective way, particularly for handling flash crowds. However, BurScale does not try to meet latency SLOs and the cost savings only come from switching the overprovisioned regular instances to burstable instances. This results in higher cost for BurScale while our solution adapts to the user requirements while optimizing cost. The BIAS Autoscaler [11] also implements a very similar approach to BurScale in Google Cloud, which does not use a token bucket based rate limiter in burstable instances.

Several works use the idea of throttling resources to extend the credit depletion period of burstable instances. CEDULE+ [29] aims to do resource management in the cloud using burstable instances by maximizing the credit depletion period of the startup credit of the older generation AWS $t2$ instances by throttling resources. It focuses on selecting which specific burstable instance to use for a certain application. When the credit is depleted, they migrate to another instance with startup credits. Additionally, CEDULE+ only does vertical scaling, moving to a larger or smaller instances, but does not have any provision for autoscaling. Other works [1, 2] also use similar throttling ideas as CEDULE+ to extend credit depletion periods. However, since the newer generation of burstable instances do not have startup credits, the concept of credit depletion is not directly applicable anymore. [27] uses a similar idea of exploiting two characteristics of burstable storage - startup burst credit and the fact that burst throughput limit is per volume and not by capacity. These artifacts are specific to burstable storage and don't apply to the compute context that we are exploring.

Burstable instances have been used along with other non-burstable instances for optimizing cost and performance for various use cases. Wang et. al. [34] make use of spot instances to lower the cost of operating in-memory storage, and they use burstable instances as a backup for spot revocation to help reduce performance degradation during spot failures. BURST-HADS [33] is another scheduling framework that uses spot and burstable instances for scheduling Bag-of-Tasks applications with deadline constraints. It uses a fixed percentage of burstable instances compared to the spot instances and make use of them for spot revocation. However, our system does not use burstable instances as a backup for migrating tasks and makes use of them like a regular instance. Another use case for burstable instances is for distributed data processing. CASH [31] is a burst credit aware scheduler that keeps track of burst credits associated with individual hardware resources in public cloud clusters for distributed data processing frameworks. However, CASH uses a cluster with only burstable instances, and it focuses on a different problem of task placement across burstable instances, while our work focuses on both autoscaling and load balancing among regular and burstable instances. Burstable instances have also been used in [12, 24] for computational sprinting, which is the idea of intentionally running workloads quickly in a burst to get to an idle low power/low thermal state. This is a different context from our work on autoscaling and load balancing. In [24], burstable instances are considered from the provider's perspective for co-locating instances whereas our work focuses on the user's perspective. Additionally, these works control sprinting on a per-query basis since the queries/jobs take longer to run (e.g., seconds to minutes) than our requests that operate at hundreds of milliseconds.

There has also been a lot of general autoscaling research for optimizing either cost or performance or both. Prior works such as [13] proposed an engine to model the workload and evaluate scaling policies. There are autoscaling policies for mutitier datacenters [14] and Hadoop clusters [15], online autoscaling algorithms where requests can be rejected with a penalty [32], and autoscaling techniques to reserve servers in

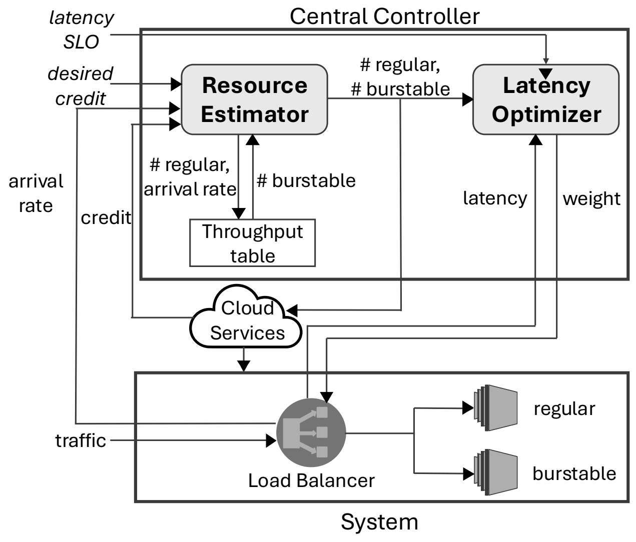


Figure 1: AutoBurst Architecture.

Regular \ Burstable	5	7	9
5	314	**372**	419
7	381	432	508
9	444	509	569

Table 1: Throughput table. If the overprovisioned arrival rate is 360 and the number of regular instances is 5, then we would select 7 burstable instances based on this throughput table.

the cloud while considering the variability of workloads [18]. SIA [9] is an autoscaling solution for ML workloads where it introduces an anticipatory scheduler that is robust to prediction errors. There are also Reinforcement Learning based autoscaling policies (e.g., [5, 17]) that focus on cloud workloads. Prior works have also considered heterogeneity aware scaling and load balancing [7, 8, 16]. However, none of these autoscaling solutions take into account the utilization of burstable instances and the associated token-bucket mechanism.

3 ARCHITECTURE AND DESIGN

Our goal is to design a system that can keep latency near a desired Service Level Objective (SLO) while minimizing cloud costs. Our solution, AutoBurst, achieves this using a combination of cheap burstable instances and expensive regular instances.

Figure 1 shows the architecture of AutoBurst. It consists of two primary components that operate at different timescales – the Latency Optimizer and the Resource Estimator. The Latency Optimizer controls the load balancer, so it can operate quickly every few seconds, ensuring a fast response for adapting to load changes and maintaining the desired latency. This enables the Resource Estimator to operate at longer timescales (e.g., every few minutes) to adjust the number of instances to ensure we have enough credits in the long term. Through this division of responsibilities, the Latency Optimizer is responsible for quickly adapting to workload changes and meeting latency SLOs via load balancing whereas the Resource Estimator is responsible for autoscaling the number of resources to optimize cost and maintain long term stability.

By contrast, traditional autoscalers only control the number of instances, which needs to address one of the key autoscaling challenges: startup delay. Since the delay to startup new VMs or containers could take 30 seconds to a few minutes, prior autoscaling research has explored multiple methods for dealing with the delay including overprovisioning extra resources, developing sophisticated future load prediction techniques, designing smarter policies for adding/removing resources, etc. [14]. AutoBurst adopts a novel design that avoids this problem entirely by using burstable instances and load balancing to meet latency SLOs in the short term, which allows us sufficient time to adjust the number of instances over a longer time period to maintain desired burstable credit levels.

As shown in Figure 1, we divide the decision-making between the two components as follows. The Latency Optimizer assumes a fixed number of regular and burstable instances as input, which is provided by the Resource Estimator. The Latency Optimizer's sole responsibility is meeting the user configured latency SLO by monitoring the latency and adjusting the weights at the load balancer. We use a weighted shortest queue first load balancer where the regular instances use a fixed weight and we dynamically vary the weights of the burstable instances. Lowering the weight will reduce the traffic to the burstable instances, which saves credits (and indirectly reduces cost) but increases queueing and latency at the regular instances. This works in conjunction with the Resource Estimator since having more credits will cause the Resource Estimator to adjust the number of regular and burstable instances to reduce cost. Increasing the weight will send more traffic to the burstable instances, which improves the latency since the burstable instances help reduce the burden on the regular instances. At the highest weight (i.e., equal to regular instance weight), the burstable instances are treated as equivalents of regular instances, and at the lowest weight, the burstable instances are hardly used at all. Details about the Latency Optimizer are explain in Section 4.1.

The Resource Estimator is responsible for autoscaling the number of regular and burstable instances based on the arrival rate of traffic and the total burstable credit level. To optimize cost, we adjust the number of regular instances based on the current credit levels and a user specified desired credit level, which represents the size of a burst that the user wants to tolerate. Using fewer expensive regular instances will reduce cost, but result in consuming more burstable credits, whereas using more regular instances will increase cost, but build up burstable credits. We then use a throughput table (Table 1) to determine the number of burstable instances based on the selected number of regular instances and the arrival rate. The throughput table is generated as a simple offline profiling step where we measure the throughput supported by various numbers of regular and burstable instances. Details about the Resource Estimator are explained in Section 4.2.

Configuring AutoBurst. The two user specified parameters for AutoBurst are the latency SLO and the desired burstable credit level. The latency SLO indicates the desired latency value and would be configured based on performance requirements for interactivity/responsiveness. The other parameter is the desired total burstable credit level. It should be configured to be high enough such that the Latency Optimizer won't run out of credits when trying to meet the latency SLO. That is, the parameter should be configured to sustain the workload during periods of overload or sudden fluctuations, particularly in scenarios with high workload variability. Hence, for workloads with high variability, the desired credit level should be set higher to accommodate such potential surges. Conversely, for low variability workloads, a lower desired credit level may suffice. We can think of desired credit level in terms of how long the burstable instances could operate solely on credits. It can be calculated as the product of the desired duration of the system to run solely on credits and the number of burstable instances currently active, as these instances store the credits. Future research could also investigate how to automatically set this parameter based on other user goals/metrics and how to dynamically vary the configuration over time.

4 IMPLEMENTATION DETAILS

4.1 Latency Optimizer

The Latency Optimizer adjusts the weights of a weighted shortest queue first load balancer to adapt latency to a user-provided SLO. The weights of the load balancer control the traffic that will be sent to each instance by sending new requests to the shortest queue length divided by the weight. That is, higher weights result in more traffic being sent to the instance. We dynamically control the weights of burstable instances to vary between a minimum (100) and maximum (1000) and fix the weights of regular instances at the maximum (1000)[1]. At most, burstable instances will be used equivalently to regular instances, and at minimum, burstable instances will be used when the queueing at regular instances is 10× higher.

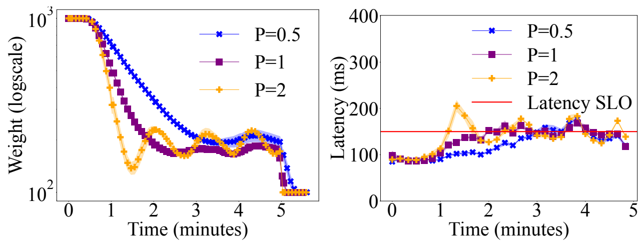
4.1.1 Weight Adjustment Using PD controller. The Latency Optimizer adjusts the weights of the burstable instances to meet the desired latency SLO. If the current measured latency is above the target SLO, burstable instances are utilized more to help reduce queueing, which is achieved by increasing the weights of the burstable instances. If the latency is under the target SLO, burstable instances are utilized less to accumulate burstable credits, which is achieved by reducing the weights of the burstable instances.

The weight adjustment is based on the well-established PD feedback controller from control theory [28]. Other controllers may work as well, but we use a PD controller since it is simple and works for our purposes. A PD controller uses an error term, which is defined as the difference between a desired setpoint (e.g., latency SLO) and a measured process variable (e.g., current latency), and it minimizes the error over time using two terms – a proportional term and a derivative term. The proportional term is proportional to the current value of the error and the derivative term uses the rate of change of the error to provide a damping effect. Intuitively, the proportional term changes the control knob (e.g., weights) more when there is a larger error (e.g., difference between latency and SLO), and the derivative term reduces the overshooting by damping the effect when the error is rapidly changing.

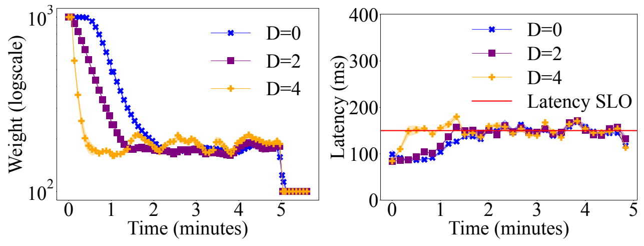
Control Knob: Log of weight. In our load balancer, Nginx, the queue length is divided by the weight where higher weights result in more requests being sent to the instance. But since the weight is a divisor, it does not exhibit a linear relationship, so it is not suitable for a PD controller, which is a linear controller. We address this issue by instead controlling the log of the weight. That is, our PD controller adjusts a log_weight value, and then we compute the actual weight via 10^{log_weight}.

Goal: Latency SLO. The goal of the Latency Optimizer is to meet a desired latency SLO set by the user. We measure the average latency every control loop iteration (5 seconds) and compute the PD controller error term as the difference between the measured latency and the SLO. As a result, there will be times where the latency is higher than the SLO and other times when the latency is lower than the SLO. Rather than treating the SLO as an upper limit, our goal is to instead

[1]Our load balancer, Nginx, only allows the configuration parameter to be input as an integer, so we use 1000 and 100 instead of 10 and 1, respectively.



(a) Weights (Tuning P, D=0). **(b) Latency (Tuning P, D=0).**



(c) Weights (Tuning D, P=1). **(d) Latency (Tuning D, P=1).**

Figure 2: PD tuning for Latency Optimizer. Error bands are included for standard error over 3 runs. We pick P=1 and D=2 for quick convergence without significant oscillations.

keep the latency near the SLO when possible. We believe our approach should generalize to other latency metrics as well, but we leave the evaluation of other metrics to future work. The main change to support higher tail percentiles would be adjusting the measurement window to accurately measure latency.

PD Parameter Tuning. There are two parameters that need to be tuned in a PD controller. The P parameter is multiplied by the proportional term and the D parameter is multiplied by the derivative term to adjust the magnitude of each term's impact on changing the weights. A common practice for tuning PD controllers is to test various parameters and analyze the resulting system behavior. Figure 2 shows an example of how we do this where we run the same input traffic trace with different values of P and D and analyze the weights and latency to see if it is converging to the SLO. In the first two subfigures, we start by tuning the P parameter where high values (P=2) lead to oscillations and low values (P=0.5) take longer to converge. We select P=1 and then tune the D parameter in the third and fourth subfigures. We select D=2 as it converges quickly without significant oscillatory behavior. Evaluations for different workload patterns in Section 6.1, Section 6.3, and Section 6.5 show that this tuning generalizes to other traces.

4.1.2 *Load balancing policy modification.*

We use Nginx as our load balancer with the weighted least connections load balancing policy. Unfortunately, this policy does not allow us to avoid sending requests to burstable instances even if their weight is as low as possible. This is because the load balancing policy is implemented to select the instance i using the following equation: $\min_i \frac{\#\ active\ connections_i}{weight_i}$. Under this equation, instances would never have more than 1 request until all instances have at least 1 active request. This makes sense under typical load balancing use cases, but in our case, it prevents us from keeping burstable instances idle so that burstable credits can accumulate.

To address this issue, we modified the policy to select the instance i based on this equation: $\min_i \frac{\#\ active\ connections_i + 1}{weight_i}$. This can be interpreted as representing the count including the new request. This simple modification ensures that requests are only sent to the burstable instances once the regular instances have built up a long enough queue (based on the weight ratio between regular and burstable instances). Thus, burstable instances are still utilized for short term bursts when there are long queues, but they can remain idle and accumulate credits during non bursty times when the weights are low.

4.2 Resource Estimator

The Resource Estimator autoscales the number of regular and burstable instances to handle the current arrival rate and ensures that we don't run out of burstable credits by adapting the ratio of burstable and regular instances. To avoid running out of burstable credits, we control the ratio of regular and burstable instances by using a PD controller to set the number of regular instances. Having more regular instances will cause the Latency Optimizer to not use burstable instances as much, allowing them to accumulate credits. Having fewer regular instances will cause the Latency Optimizer to utilize burstable instances more to maintain latency SLOs, resulting in a decrease in credit level.

After determining the number of regular instances, we use a profiled throughput table to look up the number of burstable instances needed to handle the incoming traffic load. Table-based methods are one of the simplest autoscaling solutions, and we find that even this simple approach works well since the Latency Optimizer handles meeting the latency SLO using the flexibility of burstable instances.

4.2.1 *Controlling Number of Instances using PD controller.*

Since the Latency Optimizer already adapts to minimize credit usage while meeting latency SLOs, the Resource Estimator just needs to adjust the number of instances to control credit levels, which is also adapted via a PD controller.

Control Knob: Number of Regular Instances. We use the number of regular instances as the control knob for the controller, which is counter-intuitive since our goal is to control the burstable credit level. To increase the credit level, we increase the number of regular instances, which may actually reduce the number of burstable instances. The reason this works in the long term is there's a higher ratio of regular to burstable instances. This in turn causes our Latency Optimizer to load balance more traffic to the regular instances, thereby reducing burstable utilization and increasing the burstable credit level. Similarly, we decrease the credit level by decreasing the number of regular instances. This increases the load on the burstable instances, causing the credit level to deplete.

When decreasing the number of burstable instances, we don't actually immediately remove them if they have remaining credits so that the credits are not wasted. Instead, they are placed on a pending deletion list where we bias them to be utilized more. Only after the instance has utilized all of the instance's credits do we remove the instance. Since we maintain a pending deletion list, we also first pick from these resources if we're increasing the number of burstable instances. We pick the instance in the pending deletion list with the highest number of credits when increasing burstable instances and we move the instance with the lowest credits to the pending deletion list when decreasing burstable instances. This approach naturally phases out burstable instances without losing any existing credits.

Goal: Desired Burstable Credit Level. The goal of the Resource Estimator is to achieve a desired total burstable credit level. This level is user configurable and can be set to different values for different use cases. For example, if the user expects a highly variable workload, they might want to maintain a high credit level to avoid running out of credits. The desired level can also be dynamically changed to handle unexpected load changes.

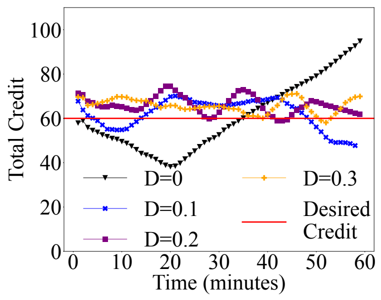
PD Controller Frequency. To avoid changing the number of instances too quickly, we run the control loop every minute and adjust the number of regular instances in fractional units. Thus, it may take several iterations for the number to increase or decrease. The frequency can be tuned, though we find our settings allow for reasonably quick adaptation without significantly over-reacting.

PD Parameter Tuning. We tune the PD parameters in the same way as the Latency Optimizer where we test various parameters and analyze the resulting system behavior. Figure 3 shows an example of how we tune the system by testing various D and P parameters. We find that P=0.01 and D=0.2 provides the best behavior in terms of convergence and avoiding oscillations. Evaluations for different workload patterns in Section 6.1, Section 6.3, and Section 6.5 demonstrate that this tuning generalizes to other traces.

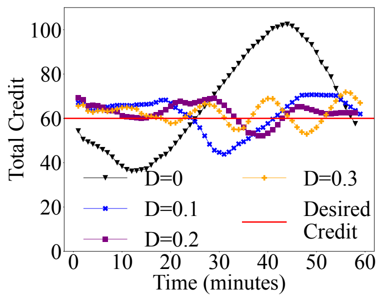
4.2.2 Number of Burstable Instances using Throughput Table Lookup.

Once the Resource Estimator determines the number of regular instances, it chooses the number of burstable instances via a table lookup. This table contains the system capacity (i.e., max throughput of the system) for different numbers of regular and burstable instances (Table 1). The table is two-dimensional, corresponding to one regular and one burstable instance type for whatever size fits the application (e.g., memory constraints). The system capacity is measured in an offline profiling step via closed queueing experiments where we send many requests to the system over a few minutes to measure the maximum achievable throughput. This only needs to be performed once unless the system performance changes significantly. To accelerate the profiling, we measure a subset of the throughputs and linearly interpolate the rest. Thus, we do not anticipate the throughput table to be expensive or time consuming to generate. We find that the interpolation works reasonably well, and any mispredictions can be handled by the Latency Optimizer since burstable instances can burst to address mispredictions in the short term. Our Resource Estimator then adapts the number of burstable and regular instances to deal with mispredictions in the long term. Since AutoBurst is designed to control latency via load balancing, we don't rely on the fidelity of the throughput table in meeting latency SLOs, and a good enough approximation is sufficient.

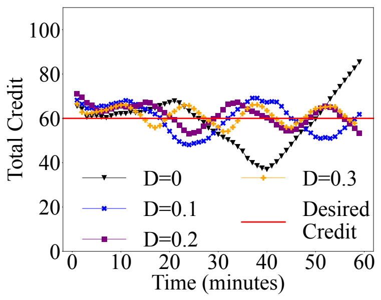
Overprovisioning arrival rate. The throughput table contains the system capacities, which are a different quantity than the arrival rate. In particular, we always want the arrival rate to be less than the system capacity to avoid overloading the system. Based on fundamental queueing theory principles, latency will grow to infinity as the arrival rate approaches the system capacity, and the system will be unable to keep up with the arrival rate if it exceeds the system capacity. Furthermore, since the load (i.e., arrival rate) changes over time, we need to account for increases to the current arrival rate. Thus, we scale the measured arrival rate by an overprovisioning factor to account for both of these concerns. In our experiments, we use a factor of 1.375, which allows us to operate at a reasonable utilization while handling the load variations in our experimental workloads. Users can adjust the overprovisioning factor if needed based on their workload needs. We multiply the current arrival rate by this factor and select the minimum number of burstable instances where the throughput is higher than this value. Note that we only consider one row in the throughput table since the number of regular instances is already chosen by the PD controller.



(a) Tuning D, P=0.005.

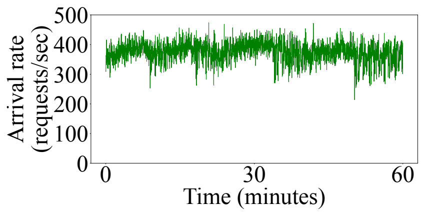


(b) Tuning D, P=0.01.

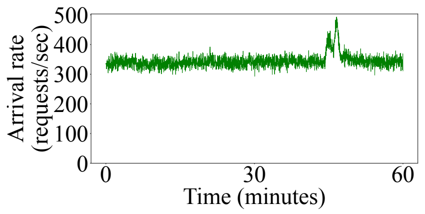


(c) Tuning D, P=0.02.

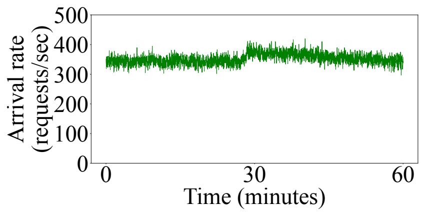
Figure 3: PD tuning for Resource Estimator. P=0.005 gives total credit values higher than desired credit or total credit value diverges from desired credit. P=0.02 has high oscillations. So, we pick P=0.01. Within P=0.01, D=0.2 converges more closely than others. So we pick D=0.2.



(a) Real Trace 1 with wavy pattern.



(b) Real Trace 2 with a load spike.



(c) Real Trace 3 with a load increase.

Figure 4: Arrival rates of downscaled Wikipedia access traces used in Section 6.1 and Section 6.2.

5 EVALUATION METHODOLOGY

5.1 System description

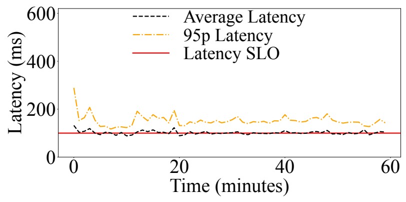
For the evaluation, we deploy a 3-tier web application system. We use a PHP-based web application called MediaWiki [22] for the application layer. In our setup, each instance of MediaWiki is deployed with Memcached [23] as the caching layer. An Nginx [26] load balancer is used to balance the load among the MediaWiki nodes. The load balancing policy in the Nginx load balancer is modified for AutoBurst as discussed in Section 4.1.2. We also deploy a MySQL [25] database instance as the database layer of the setup. A trace replayer is used as the client application which generates requests based on the trace supplied to it. It is a multi-threaded application, where each thread works as a different client, replicating the real-world traffic consisting of multiple clients. We also have our Central Controller in a separate node.

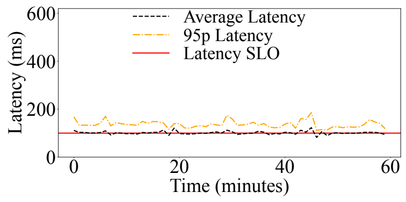
We deploy our setup on Amazon EC2 instances. Each of the MediaWiki servers with Memcached, the load balancer, the database and the client application are in separate VMs in separate instances. We use *m*5.*4xlarge* instances (16 vCPU, 64 GiB Memory) for the databases, *m*5.*xlarge* instance (4 vCPU, 16 GiB Memory) for the load balancer, *m*5.*large* instance (2 vCPU, 8 GiB Memory) for the client and the controller, and a combination of *m*5.*large* and *t*3.*small* instances (2 vCPU, 2 GiB Memory) for the regular and burstable server instances respectively [3].

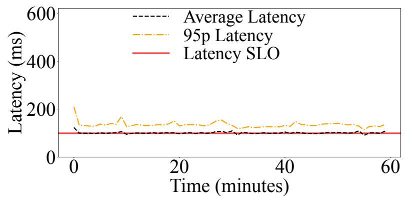
5.2 Metrics

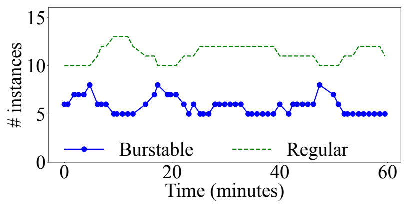
While comparing AutoBurst with the baseline, our primary focus is on cost and latency as the key metrics for evaluation. To avoid either the baseline or AutoBurst from experiencing poor latency due to running out of burstable credits, we run the burstable instances in unlimited mode where we get charged for surplus credits when they run out of burstable credits. For calculating cost, we take into account both the expenses associated with running the instances and the expenditure on credits utilized or accrued during the experiment's duration. We also incorporate the cost of surplus credits in the cost calculation.

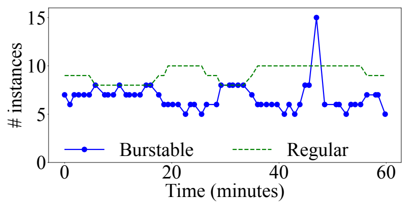
We make use of AWS cloudwatch to collect the credit values. AWS cloudwatch [10] offers CPU utilization values every minute and credit level values every five minutes. We use the per minute CPU utilization values to estimate per minute credit level values and synchronize it with the credit level values collected every five minutes similar to the ideas in [31].


(a) AutoBurst latency for Real Trace 1 with SLO=100ms.


(b) AutoBurst latency for Real Trace 2 with SLO=100ms.


(c) AutoBurst latency for Real Trace 3 with SLO=100ms.


(d) AutoBurst instance count for Real Trace 1 with SLO=100ms, which gives a cost saving of 11.85%.


(e) AutoBurst instance count for Real Trace 2 with SLO=100ms, which gives a cost saving of 20.41%.

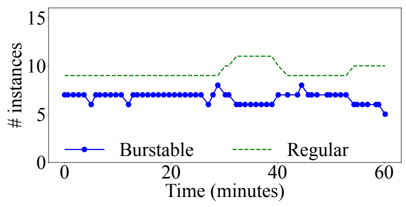
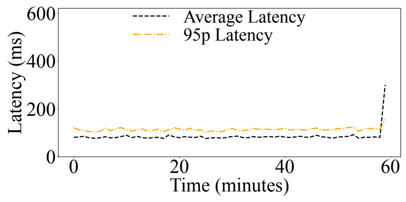
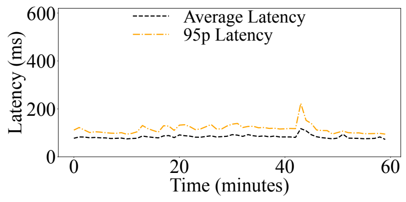
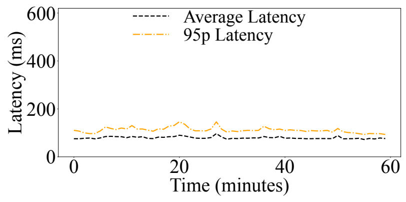
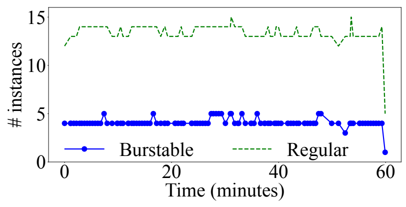

(f) AutoBurst instance count for Real Trace 3 with SLO=100ms, which gives a cost saving of 17.61%.

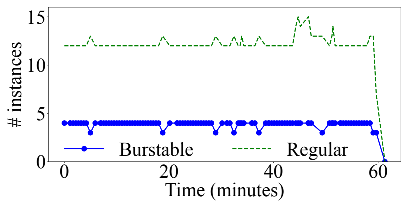
Figure 5: Performance of AutoBurst for SLO=100ms.


(a) BurScale latency for Real Trace 1.


(b) BurScale latency for Real Trace 2.


(c) BurScale latency for Real Trace 3.


(d) BurScale instance count for Real Trace 1.


(e) BurScale instance count for Real Trace 2.

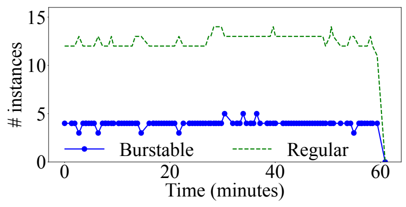

(f) BurScale instance count for Real Trace 3.

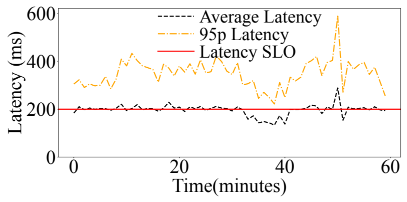
Figure 6: Performance of BurScale.

5.3 Baseline

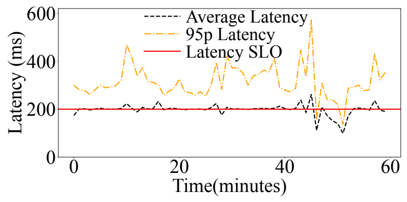
BurScale [4] is the state-of-the-art system for using burstable instances to reduce cost. Their results show how BurScale can save up to 50% in costs over traditional autoscaling systems with only regular instances. Our evaluation compares against BurScale and shows how AutoBurst can improve an additional 12-25% in cost savings.

5.4 Traces

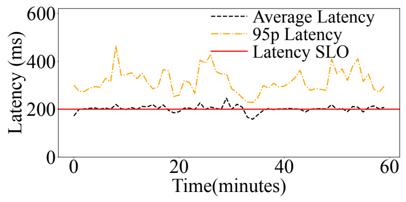
We evaluate our system with both real-world Wikipedia access traces [36], with actual timestamps exhibiting time-varying load and non-Poisson arrival processes, and synthetic traces. Since the arrival rates of these traces are higher than what our system can handle, we downscale the traces using a trace downscaling tool [30] that preserves these arrival timing characteristics. Figure 4 shows the arrival rates for the three one hour-long traces that we used for experimentation. The first trace exhibits a wavy pattern with variability



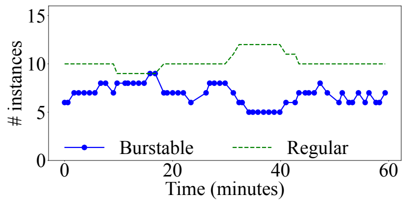
(a) AutoBurst latency for Real Trace 1 with SLO=200ms.



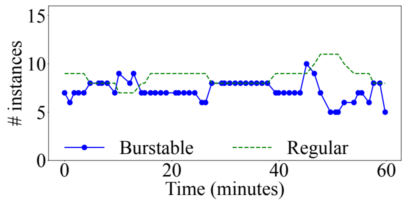
(b) AutoBurst latency for Real Trace 2 with SLO=200ms.



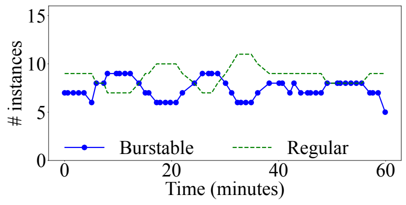
(c) AutoBurst latency for Real Trace 3 with SLO=200ms.



(d) AutoBurst instance count for Real Trace 1 with SLO=200ms, which gives a cost saving of 20.12%.

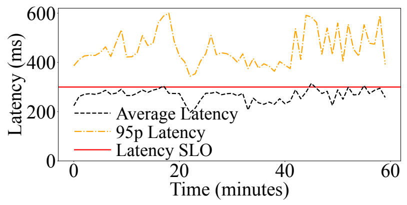


(e) AutoBurst instance count for Real Trace 2 with SLO=200ms, which gives a cost saving of 24.52%.

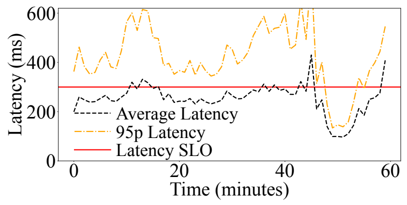


(f) AutoBurst instance count for Real Trace 3 with SLO=200ms, which gives a cost saving of 23.51%.

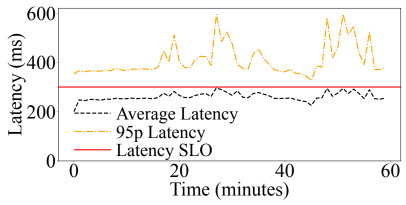
Figure 7: Performance of AutoBurst for SLO=200ms.



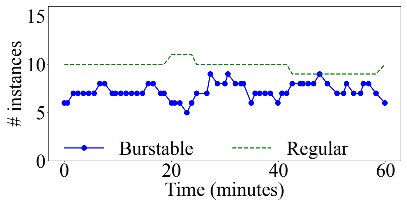
(a) AutoBurst latency for Real Trace 1 with SLO=300ms.



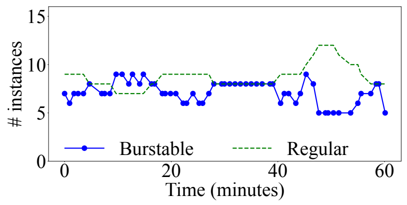
(b) AutoBurst latency for Real Trace 2 with SLO=300ms.



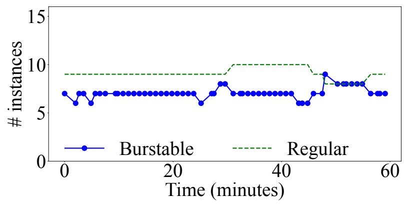
(c) AutoBurst latency for Real Trace 3 with SLO=300ms.



(d) AutoBurst instance count for Real Trace 1 with SLO=300ms, which gives a cost saving of 21.93%.



(e) AutoBurst instance count for Real Trace 2 with SLO=300ms, which gives a cost saving of 22.8%.



(f) AutoBurst instance count for Real Trace 3 with SLO=300ms, which gives a cost saving of 23.23%.

Figure 8: Performance of AutoBurst for SLO=300ms.

in load over time. The second trace exhibits a load spike that lasts a few minutes. The third trace exhibits a load increase pattern where the load increases halfway through the trace. We selected these traces with different arrival rate patterns to evaluate a range of scenarios that autoscaling systems may face in practice.

We also use synthetic traces to evaluate how well Auto-Burst can handle load changes (e.g., gradual and instantaneous) in a more controlled setting. We generate these as a Poisson process with time-varying load. We use simple fixed-load Poisson processes as well in cases where we want to illustrate other effects without the impact of load changes.

6 RESULTS

We evaluate AutoBurst for cost effectiveness and study its two components: the Latency Optimizer and the Resource Estimator. For the Latency Optimizer, we evaluate its robustness in meeting latency SLOs. For the Resource Estimator,

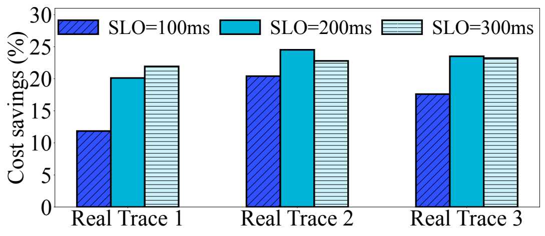


Figure 9: AutoBurst's cost savings over BurScale.

we evaluate its ability to converge to desired burstable credit levels.

The first set of experiments demonstrate how AutoBurst is able to provide cost savings over BurScale, the state-of-the-art system using burstable instances for cost savings. We compare AutoBurst and BurScale across 3 different real traces (Section 6.1) and find that AutoBurst provides 12-20% cost savings over BurScale with similar latency. We then evaluate the cost effectiveness of AutoBurst across different latency SLO values (Section 6.2) and find that it is able to achieve even greater cost savings with more relaxed latency SLOs, whereas BurScale does not support any notion of a latency SLO.

The second set of experiments test the robustness of AutoBurst's Latency Optimizer component in meeting latency SLOs. We use synthetic traces to demonstrate how AutoBurst is robust to sudden and gradual load changes (Section 6.3). We also disable the Resource Estimator and use a mispredicted number of regular and burstable instances to demonstrate how the Latency Optimizer can adapt to meet latency SLOs despite having too many or too few instances (Section 6.4). We find that even if the Resource Estimator mispredicts the number of instances, the Latency Optimizer can utilize burstable instances to cope with short term mispredictions, which can be rectified in future iterations of the Resource Estimator. This also shows how both the components working together play a crucial role to ensure a smooth operation.

The third set of experiments focus on the convergence of burstable credit levels for the Resource Estimator (Section 6.5). Whether the load is changing or the user is dynamically adjusting the desired credit level, the Resource Estimator is able to autoscale the number and ratio of regular and burstable instances to converge to the desired credit level reasonably quickly.

6.1 Cost Savings of AutoBurst over BurScale

We run three one hour-long traces (Figure 4) to demonstrate the cost benefits of using AutoBurst over BurScale. The traces come from Wikipedia [36] and show that AutoBurst is able to adjust to different types of real-world traffic patterns.

Figure 5 shows AutoBurst's latency, number of regular/burstable instances, and cost savings over BurScale. We note that AutoBurst is able to save cost over BurScale for every trace with cost savings ranging from 11.85% up to 20.41% depending on the trace.

From the latency graphs, we see that AutoBurst's Latency Optimizer properly controls the average latency to be near the latency SLO. We see that the 95th percentile latency exhibits a similar behavior, albeit at a higher latency, which is expected. The instance count graphs demonstrate how AutoBurst autoscales resources. While the total number of instances changes a little based on the trace, mostly we see AutoBurst's Resource Estimator adapting the ratio of regular and burstable instances to control the total burstable credit levels, which allows for less time when expensive regular instances are used.

Figure 6 shows the corresponding latency and instance count graphs for BurScale. BurScale achieves a similar latency, but is much more expensive since it uses many more expensive regular instances than burstable instances. Thus, AutoBurst uses burstable instances more efficiently than BurScale, leading to greater cost savings at similar latency levels.

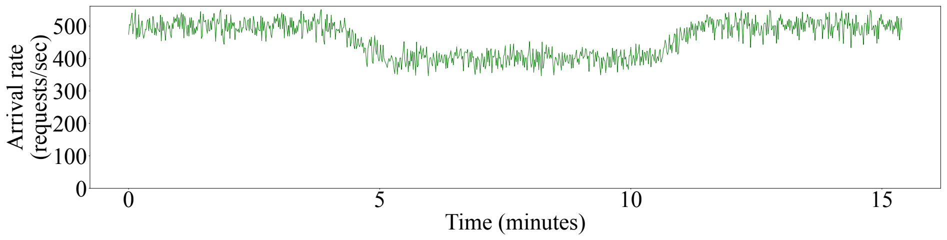
Summary: AutoBurst is able to provide significant cost savings over BurScale across multiple real-world traffic patterns.

6.2 Effect of SLO on Cost Savings

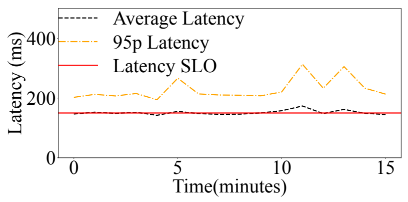
In this section, we set the latency SLO to different values to demonstrate the impact on cost savings in AutoBurst. A higher latency SLO gives AutoBurst more flexibility to save cost by load balancing regular instances to have longer queues, thereby reducing work and credit usage at burstable instances. BurScale does not support latency SLOs so it is unable to save any cost from users with more relaxed latency requirements.

Figure 5, Figure 7, and Figure 8 show the results from AutoBurst with a latency SLO of 100ms, 200ms, and 300ms, respectively. AutoBurst meets the latency SLO at 100ms and 200ms and is below the latency SLO at 300ms since the regular instances are able to handle the workload without significant help from burstable instances. Since the Latency Optimizer is able to conserve more burstable credits, the Resource Estimator is able to use fewer expensive regular instances while maintaining the desired credit level, thereby reducing cost. Figure 9 summarizes the cost savings with AutoBurst achieving up to about 25% savings over BurScale.

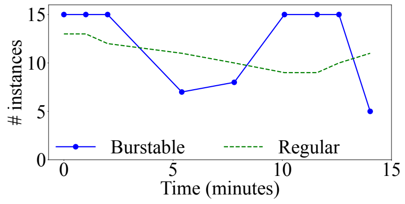
Summary: Higher values of latency SLO gives AutoBurst more flexibility in optimizing burstable credit usage and cost, which generally increases cost savings.



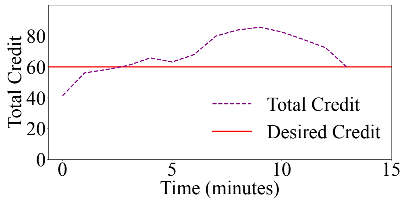
(a) Synthetic trace with gradual change in load.



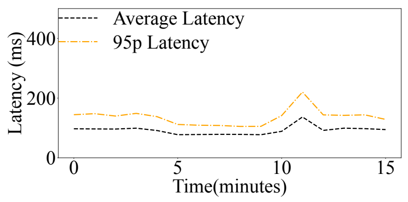
(b) AutoBurst latency.



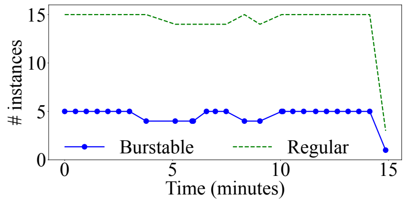
(c) AutoBurst instance count.



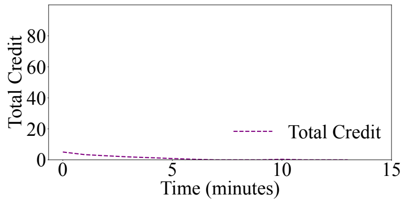
(d) AutoBurst total burstable credit.



(e) BurScale latency.



(f) BurScale instance count.



(g) BurScale total burstable credit.

Figure 10: Performance of AutoBurst vs. BurScale with a gradual load change synthetic trace (Figure 10a).

6.3 Robustness to Load Changes

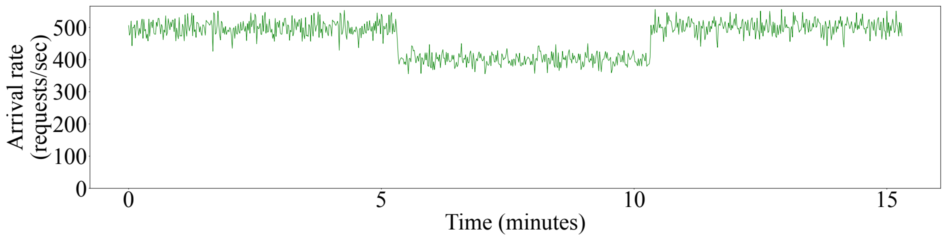
We use two different synthetic traces where the load changes with time to demonstrate that AutoBurst is robust to the changes while saving cost over BurScale. The difference between these two traces is that in the first trace (Figure 10a), the load changes happen gradually over a minute while in the second trace (Figure 11a), the load changes happen instantaneously. In Figure 10, we observe how AutoBurst and BurScale perform with respect to the trace shown in Figure 10a. We set a latency SLO of 150 ms, which both the systems are able to meet. However, BurScale uses up all of its credit immediately as shown in Figure 10g while AutoBurst is able to accumulate credits (Figure 10d) while using fewer number of regular instances (Figure 10c) than BurScale (Figure 10f). This results in a cost saving of 13.85% for AutoBurst over BurScale. We observe a similar effect with the trace in Figure 11, resulting in a 13.75% cost savings over BurScale. Note that, for both of these experiments, all of the instances were able to accumulate credit for the same duration, but since BurScale decides to use fewer burstable instances than AutoBurst, it uses up the credits and ends up with a lower total credit than AutoBurst.

Summary: AutoBurst is robust to both gradual and sudden load changes and is able to meet latency SLOs while sustaining cost savings.

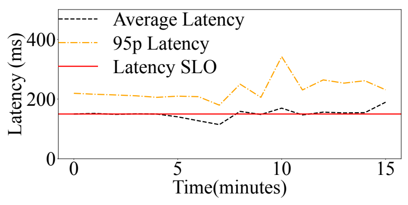
6.4 Robustness to Mispredictions

In this section, we show that the Latency Optimizer is robust to mispredictions of the Resource Estimator. If the Resource Estimator is unable to correctly predict the required number of instances (e.g., the arrival rate has changed right after the decision was made), we run the risk of being unable to meet the latency SLOs with an incorrect number of instances. However, in AutoBurst, the Latency Optimizer is able to adjust the weights and make use of the accumulated credits to meet the latency SLO.

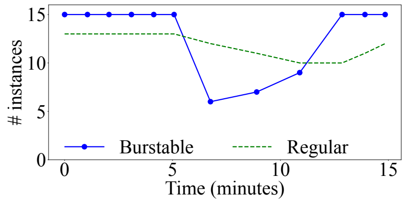
To show this effect, we disable the Resource Estimator for the experiments in this section and use a fixed number of instances. Ideally, we would provision 10 regular and 10 burstable instances to meet a 200ms latency SLO with the trace in Figure 12a, which is another trace from the Wikipedia dataset with no significant load change. Figure 12b shows how the latency and credit level are impacted by different types of mispredictions. The number of burstable instances is kept fixed at 10 instances, but the number of regular instances is set to different values corresponding to different



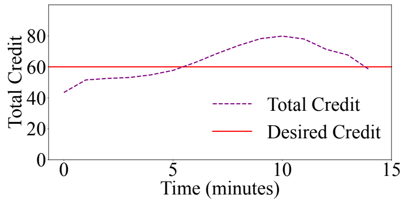
(a) Synthetic trace with instantaneous change in load.



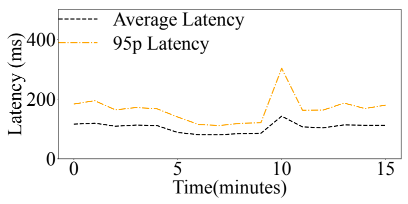
(b) AutoBurst latency.



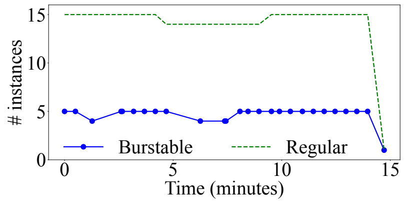
(c) AutoBurst instance count.



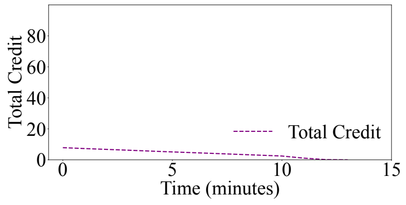
(d) AutoBurst total burstable credit.



(e) BurScale latency.



(f) BurScale instance count.



(g) BurScale total burstable credit.

Figure 11: Performance of AutoBurst vs. BurScale with an instantaneous load change synthetic trace (Figure 11a).

degrees of misprediction in the number of regular instances. That is, the x-axis shows the misprediction in the number of regular instances, and the two y-axes show the average latency and average credit level. We change the number of regular instances by up to 4 regular instances, corresponding to a 40% misprediction.

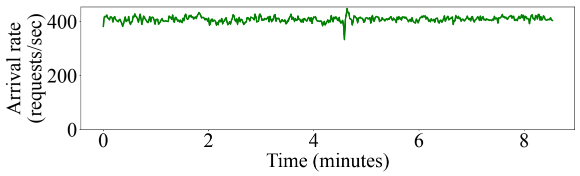
When underestimating the number of instances by 20%, AutoBurst is able to meet the latency SLO by adjusting the weights of the burstable instances to utilize more credits, which can be seen by the lower credit values. When overestimating the number of instances by 20%, AutoBurst is able to save credits while meeting the latency SLO, resulting in higher credit values. In extreme cases of underestimation, AutoBurst will maximize the burstable instance usage to be equivalent to regular instance usage, but there may not be enough total instances, resulting in elevated latency. In extreme cases of overestimation, AutoBurst will avoid using burstable instances, but there may be so many regular instances that latency is lower than the SLO. So with modest amounts of misprediction, AutoBurst's Latency Optimizer is able to adapt the load balancing to maintain the latency SLO while saving as many credits as possible. Thus, this robustness is able to cover the time it takes to start/stop new instances or handle sudden traffic changes that aren't accounted for by the Resource Estimator.

Summary: Mispredictions in the Resource Estimator are gracefully handled by the Latency Optimizer in AutoBurst.

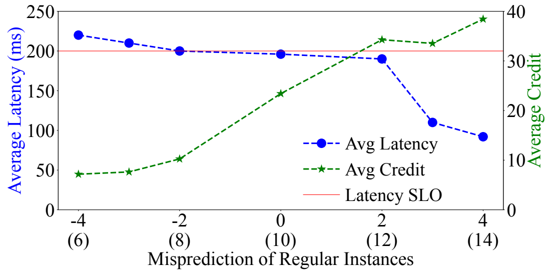
6.5 Convergence of Credit Level

In this section, we show how AutoBurst is able to converge to the desired credit level when facing changes in the (i) desired credit level or (ii) traffic load.

6.5.1 Change in desired credit level. AutoBurst adjusts the number of instances to bring the burstable credit level to be close to the desired credit level, which can be dynamically changed. Figure 13 shows how the system adjusts itself when the desired credit level changes during the experiment. In this experiment, we change the value of the desired credit level after 75 minutes to a higher value and then again change it after another 90 minutes to the initial lower value (60 and 40, respectively, for Figure 13b, and 80 and 40, respectively, for Figure 13c). The input traffic (Figure 13a) is a synthetic trace generated from a fixed load Poisson process. We choose such a fixed load trace so that we can demonstrate only the effect of changing the desired credit level.

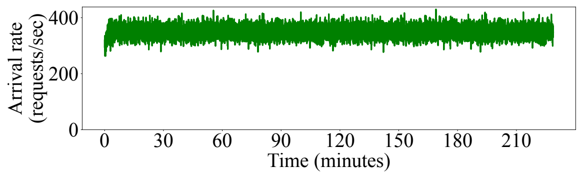


(a) Real Trace 4 without significant load changes, used in Section 6.4.

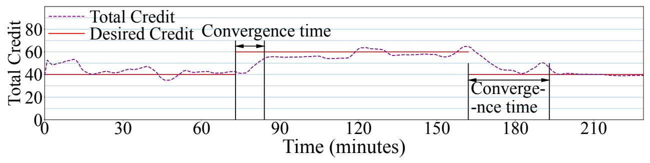


(b) Latency and credit level for different degrees of misprediction.

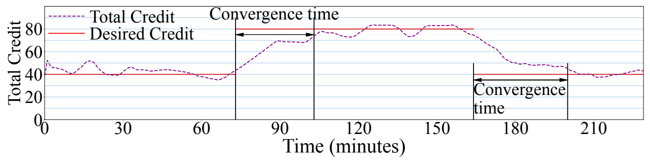
Figure 12: Effect of mispredicting the number of instances. We use a trace (Figure 12a) without significant load changes where there are ideally 10 regular and 10 burstable instances. Figure 12b shows the effect of mispredicting the number of regular instances by up to 40% (4 instances). The x-axis represents the misprediction on the number of regular instances, where the number in the parentheses represents the number of regular instances used. The number of burstable instances is always fixed at 10. There are two y-axes corresponding to the latency and average of the total credit level. With modest degrees of misprediction, AutoBurst is able to meet the latency SLO of 200ms by adapting how many credits are used/saved.



(a) Synthetic trace with a fixed load, used in Section 6.5.1.



(b) Total burstable credit over time when the desired credit level increases from 40 to 60 and then decreases back to 40. It took 11 minutes to converge when increasing the desired credit level by 20, and it took 31 minutes to converge when decreasing the desired credit level by 20.



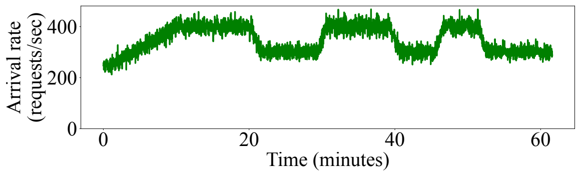
(c) Total burstable credit over time when the desired credit level increases from 40 to 80 and then decreases back to 40. It took 30 minutes to converge when increasing the desired credit level by 40, and it took 35 minutes to converge when decreasing the desired credit level by 40.

Figure 13: Effect of change in desired credit level on convergence. We use a fixed load trace (Figure 13a) and run two experiments (Figure 13b and Figure 13c) where we change the desired credit level to 60 and 80, respectively, from a desired credit level of 40.

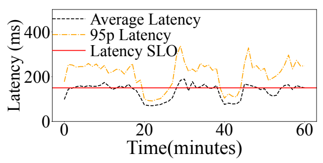
When the desired credit level increases by 20, we are able to converge within 11 minutes and when it increases by 40, we are able to converge within 30 minutes. When the desired credit level decreases by 20, we are able to converge within 31 minutes and when it decreases by 40, we are able to converge within 35 minutes. The Resource Estimator intentionally operates at a low frequency and doesn't change the number of instances every cycle, so it can take some time to converge. However, maintaining a desired credit level is not as critical as meeting latency SLOs, so we are not very concerned about optimizing credit convergence behaviors.

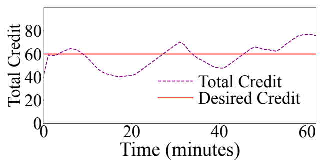
6.5.2 Change in traffic load. Real systems do not always have a fixed load, so we next evaluate a synthetic trace (Figure 14a) with significant load changes to see how it affects the burstable credit level convergence. Figure 14 shows that AutoBurst is able to adjust the instance count and weight of the instances to maintain the desired credit level and meet latency SLOs even with frequent load changes. As the load changes, the Latency Optimizer adapts the weight (Figure 14d) to load balance more or less traffic to the burstable instances, resulting in increases and decreases to the credit level (Figure 14c). The Resource Estimator then adapts the number of instances (Figure 14e) to the load as well as the credit level. The total number of instances changes roughly according to the load changes, whereas the ratio between the regular and burstable instances changes based on how close it is to the desired credit level.

Summary: AutoBurst is able to converge to the desired credit level by reacting to the changes in the desired credit level and traffic load.

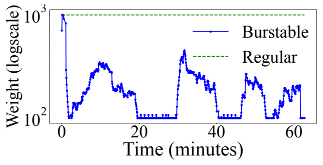


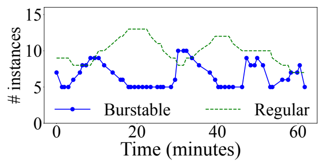
(a) Synthetic trace with multiple load changes, used in Section 6.5.2.





(b) Latency. **(c) Total burstable credit.**





(d) Load balancing weights. **(e) Instance Count.**

Figure 14: Effect of change in load on convergence. Figure 14b shows the latency for running a trace with multiple load changes (Figure 14a). Figure 14c shows the total credit of the burstable instances with a desired credit level of 60. Since the Latency Optimizer changes the load balancing weights (Figure 14d) in response to load changes to meet the latency SLO, the total credit level fluctuates. The Resource Estimator is able to bring the total credit level up to the desired credit level by adjusting the number of burstable and regular instances (Figure 14e).

7 CONCLUSION

Low cost burstable instances are offered by cloud providers as an alternative to regular instances. Burstable instances are rate limited, but have the capacity to "burst" to a high CPU utilization for limited periods of time. These instances can be used along with regular instances to optimize cost while maintaining latency SLOs. We propose AutoBurst as a general-purpose approach to leveraging burstable instances in the cloud. AutoBurst has an autoscaling component that determines the number of regular and burstable instances needed and a load balancing component that divides the traffic between the regular and burstable instances. By designing the load balancing component to operate at short timescales (seconds), AutoBurst is robust to mispredictions by utilizing the burst capability of burstable instances to meet user-configurable latency SLOs. The autoscaling component can then operate at longer timescales (minutes) to ensure there are sufficient burstable credits. Our evaluation demonstrates AutoBurst offers significant cost savings of 12-25% over the state-of-the-art by managing the allocation and utilization of burstable and regular instances.

ACKNOWLEDGMENTS

We thank our shepherd Alain Tchana and the anonymous reviewers who provided constructive and helpful feedback. This research was supported in part by National Science Foundation (NSF) grants 2122155 and 1909004.

REFERENCES

[1] Ahsan Ali, Riccardo Pinciroli, Feng Yan, and Evgenia Smirni. 2018. CEDULE: A Scheduling Framework for Burstable Performance in Cloud Computing. In *2018 IEEE International Conference on Autonomic Computing (ICAC)*. 141–150. https://doi.org/10.1109/ICAC.2018.00024

[2] Ahsan Ali, Riccardo Pinciroli, Feng Yan, and Evgenia Smirni. 2019. It's Not a Sprint, It's a Marathon: Stretching Multi-Resource Burstable Performance in Public Clouds (Industry Track). In *Proceedings of the 20th International Middleware Conference Industrial Track* (Davis, CA, USA) *(Middleware '19)*. Association for Computing Machinery, New York, NY, USA, 36–42. https://doi.org/10.1145/3366626.3368130

[3] aws 2023. Amazon EC2 Instances. https://aws.amazon.com/ec2/instance-types/. Accessed: 2023-11-15.

[4] Ataollah Fatahi Baarzi, Timothy Zhu, and Bhuvan Urgaonkar. 2019. BurScale: Using Burstable Instances for Cost-Effective Autoscaling in the Public Cloud. In *Proceedings of the ACM Symposium on Cloud Computing* (Santa Cruz, CA, USA) *(SoCC '19)*. Association for Computing Machinery, New York, NY, USA, 126–138. https://doi.org/10.1145/3357223.3362706

[5] J. V. Bibal Benifa and D. Dejey. 2019. RLPAS: Reinforcement Learning-Based Proactive Auto-Scaler for Resource Provisioning in Cloud Environment. *Mob. Netw. Appl.* 24, 4 (aug 2019), 1348–1363. https://doi.org/10.1007/s11036-018-0996-0

[6] burstableAWSformula 2023. Burstable formula. https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/burstable-credits-baseline-concepts.html. Accessed: 2023-12-26.

[7] Mohan Baruwal Chhetri, Abdur Rahim Mohammad Forkan, Quoc Bao Vo, Surya Nepal, and Ryszard Kowalczyk. 2021. Exploiting Heterogeneity for Opportunistic Resource Scaling in Cloud-Hosted Applications. *IEEE Transactions on Services Computing* 14, 6 (2021), 1739–1750. https://doi.org/10.1109/TSC.2019.2908647

[8] Mohan Baruwal Chhetri, Quoc Bao Vo, Ryszard Kowalczyk, and Surya Nepal. 2018. Towards Resource and Contract Heterogeneity Aware Rescaling for Cloud-Hosted Applications. In *2018 18th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*. 153–162. https://doi.org/10.1109/CCGRID.2018.00030

[9] Tapan Chugh, Srikanth Kandula, Arvind Krishnamurthy, Ratul Mahajan, and Ishai Menache. 2023. Anticipatory Resource Allocation for ML Training. In *Proceedings of the 2023 ACM Symposium on Cloud Computing* (Santa Cruz, CA, USA) *(SoCC '23)*. Association for Computing Machinery, New York, NY, USA, 410–426. https://doi.org/10.1145/3620678.3624669

[10] cloudwatch 2024. Burstable formula. https://aws.amazon.com/cloudwatch/. Accessed: 2024-01-31.

[11] Jaime Dantas, Hamzeh Khazaei, and Marin Litoiu. 2021. BIAS Autoscaler: Leveraging Burstable Instances for Cost-Effective Autoscaling on Cloud Systems. In *Proceedings of the Seventh International Workshop on Serverless Computing (WoSC7) 2021* (Virtual Event, Canada) *(WoSC*

'21). Association for Computing Machinery, New York, NY, USA, 9–16. https://doi.org/10.1145/3493651.3493667

[12] Songchun Fan, Seyed Majid Zahedi, and Benjamin C. Lee. 2016. The Computational Sprinting Game. *SIGARCH Comput. Archit. News* 44, 2 (March 2016), 561–575. https://doi.org/10.1145/2980024.2872383

[13] Anshul Gandhi, Parijat Dube, Alexei Karve, Andrzej Kochut, and Li Zhang. 2014. Modeling the Impact of Workload on Cloud Resource Scaling. In *2014 IEEE 26th International Symposium on Computer Architecture and High Performance Computing*. 310–317. https://doi.org/10.1109/SBAC-PAD.2014.16

[14] Anshul Gandhi, Mor Harchol-Balter, Ram Raghunathan, and Michael A. Kozuch. 2012. AutoScale: Dynamic, Robust Capacity Management for Multi-Tier Data Centers. *ACM Trans. Comput. Syst.* 30, 4, Article 14 (nov 2012), 26 pages. https://doi.org/10.1145/2382553.2382556

[15] A. Gandhi, S. Thota, P. Dube, A. Kochut, and L. Zhang. 2016. Autoscaling for Hadoop Clusters. In *2016 IEEE International Conference on Cloud Engineering (IC2E)*. IEEE Computer Society, Los Alamitos, CA, USA, 109–118. https://doi.org/10.1109/IC2E.2016.11

[16] Anshul Gandhi, Xi Zhang, and Naman Mittal. 2015. HALO: Heterogeneity-Aware Load Balancing. In *2015 IEEE 23rd International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems*. 242–251. https://doi.org/10.1109/MASCOTS.2015.14

[17] Yisel Garí, David A. Monge, Cristian Mateos, and Carlos García Garino. 2020. Learning Budget Assignment Policies for Autoscaling Scientific Workflows in the Cloud. *Cluster Computing* 23, 1 (mar 2020), 87–105. https://doi.org/10.1007/s10586-018-02902-0

[18] Cheng Hu, Yingya Guo, Yuhui Deng, and Longya Lang. 2022. Improve the Energy Efficiency of Datacenters With the Awareness of Workload Variability. *IEEE Transactions on Network and Service Management* 19, 2 (2022), 1260–1273. https://doi.org/10.1109/TNSM.2022.3144508

[19] Yuxuan Jiang, Mohammad Shahrad, David Wentzlaff, Danny H.K. Tsang, and Carlee Joe-Wong. 2019. Burstable Instances for Clouds: Performance Modeling, Equilibrium Analysis, and Revenue Maximization. In *IEEE INFOCOM 2019 - IEEE Conference on Computer Communications*. 1576–1584. https://doi.org/10.1109/INFOCOM.2019.8737634

[20] Yuxuan Jiang, Mohammad Shahrad, David Wentzlaff, Danny H. K. Tsang, and Carlee Joe-Wong. 2020. Burstable Instances for Clouds: Performance Modeling, Equilibrium Analysis, and Revenue Maximization. *IEEE/ACM Transactions on Networking* 28, 6 (2020), 2489–2502. https://doi.org/10.1109/TNET.2020.3015523

[21] Philipp Leitner and Joel Scheuner. 2015. Bursting with Possibilities–An Empirical Study of Credit-Based Bursting Cloud Instance Types. In *2015 IEEE/ACM 8th International Conference on Utility and Cloud Computing (UCC)*. IEEE, 227–236.

[22] MediaWiki 2023. MediaWiki. https://www.mediawiki.org/wiki/MediaWiki. Accessed: 2023-11-15.

[23] memcached 2023. Memcached. https://memcached.org/. Accessed: 2023-11-15.

[24] Nathaniel Morris, Christopher Stewart, Lydia Chen, Robert Birke, and Jaimie Kelley. 2018. Model-Driven Computational Sprinting. In *Proceedings of the Thirteenth EuroSys Conference* (Porto, Portugal) *(EuroSys '18)*. Association for Computing Machinery, New York, NY, USA, Article 38, 13 pages. https://doi.org/10.1145/3190508.3190543

[25] mysql 2023. MySQL. https://www.mysql.com/. Accessed: 2023-11-15.

[26] nginx 2023. Nginx. https://www.nginx.com/. Accessed: 2023-11-15.

[27] Hojin Park, Gregory R. Ganger, and George Amvrosiadis. 2020. More IOPS for Less: Exploiting Burstable Storage in Public Clouds. In *12th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 20)*. USENIX Association. https://www.usenix.org/conference/hotcloud20/presentation/park

[28] PIDcontroller 2023. PID controller. https://en.wikipedia.org/wiki/Proportional%E2%80%93integral%E2%80%93derivative_controller. Accessed: 2023-12-22.

[29] Riccardo Pinciroli, Ahsan Ali, Feng Yan, and Evgenia Smirni. 2021. CEDULE+: Resource Management for Burstable Cloud Instances Using Predictive Analytics. *IEEE Transactions on Network and Service Management* 18, 1 (2021), 945–957. https://doi.org/10.1109/TNSM.2020.3039942

[30] Sultan Mahmud Sajal, Rubaba Hasan, Timothy Zhu, Bhuvan Urgaonkar, and Siddhartha Sen. 2021. TraceSplitter: A New Paradigm for Downscaling Traces. In *Proceedings of the Sixteenth European Conference on Computer Systems* (Online Event, United Kingdom) *(EuroSys '21)*. Association for Computing Machinery, New York, NY, USA, 606–619. https://doi.org/10.1145/3447786.3456262

[31] Aakash Sharma, Saravanan Dhakshinamurthy, George Kesidis, and Chita R. Das. 2021. CASH: A Credit Aware Scheduling for Public Cloud Platforms. In *2021 IEEE/ACM 21st International Symposium on Cluster, Cloud and Internet Computing (CCGrid)*. 227–236. https://doi.org/10.1109/CCGrid51090.2021.00032

[32] Wen Si, Li Pan, and Shijun Liu. 2022. A cost-driven online auto-scaling algorithm for web applications in cloud environments. *Knowledge-Based Systems* 244 (2022), 108523. https://doi.org/10.1016/j.knosys.2022.108523

[33] Luan Teylo, Luciana Arantes, Pierre Sens, and Lúcia Maria de A. Drummond. 2023. Scheduling Bag-of-Tasks in Clouds Using Spot and Burstable Virtual Machines. *IEEE Transactions on Cloud Computing* 11, 1 (2023), 984–996. https://doi.org/10.1109/TCC.2021.3125426

[34] Cheng Wang, Bhuvan Urgaonkar, Aayush Gupta, George Kesidis, and Qianlin Liang. 2017. 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* (Belgrade, Serbia) *(EuroSys '17)*. Association for Computing Machinery, New York, NY, USA, 620–634. https://doi.org/10.1145/3064176.3064220

[35] Jiawei Wen, Lei Lu, Giuliano Casale, and Evgenia Smirni. 2015. Less Can Be More: Micro-managing VMs in Amazon EC2. In *2015 IEEE 8th International Conference on Cloud Computing*. 317–324. https://doi.org/10.1109/CLOUD.2015.50

[36] wikipediaTrace 2023. Wikipedia Access Trace. http://www.wikibench.eu/?page_id=60. Accessed: 2023-11-15.