CATCLOUD: CLOSING SEMANTIC GAPS OF CPU INTERFACES FOR PRECISE AUTOSCALING IN THE CLOUD

BY

PRATIK RAJESH SAMPAT

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Computer Science
in the Graduate College of the
University of Illinois Urbana-Champaign, 2024

Urbana, Illinois

Advisers:

Assistant Professor Saugata Ghose
Assistant Professor Tianyin Xu

ABSTRACT

Precise CPU allocation for a multi-programmed computer is crucial to application performance and resource efficiency, but is notoriously difficult under dynamic cloud workloads, where multiple users executing diverse applications often share the CPUs. We argue that the fundamental problem is rooted in the mismatch of the existing CPU allocation interface between the cloud and the OS—while the cloud represents CPU resources as a percentage quota of the host CPU (i.e., millicpu), the OS interprets CPU resources as time-shared quota slices allowed to run within a defined period. The cloud interface's disregard for periodicity stems from the fundamental difficulty of capturing fine-grained application runtime behavior in userspace. Consequently, existing solutions rely on coarse-grained, surrogate metrics such as CPU utilization, throttle, and queue lengths, leading to slow and imprecise allocation.

We present CATCloud, an OS extension that closes the semantic gap of cloud CPU allocation. CATCloud views CPU resources as a shared bandwidth interface and implements a millisecond-scale CPU bandwidth autotuner for quota and periodicity. Implemented in the OS scheduler, CATCloud realizes observability of fine-grained run time and yield time behavior of target applications; which was previously opaque to the userspace autoscalers. By continuously capturing historical data, it accurately estimates the short-term CPU period and quota requirements. With an execution latency of only a few milliseconds, CATCloud can quickly and effectively react to bursty, dynamic workloads with simple statistical algorithms. We show that CATCloud significantly outperforms state-of-the-art techniques in terms of responsiveness, precision, and efficiency. Our evaluation on various cloud workloads shows that CATCloud can improve CPU efficiency by on an average of 27.8%, up to 81.3% and performance improvements on average of 27.91%, up to 152.5% with negligible memory and compute overheads, over existing autoscaling solutions

ACKNOWLEDGMENTS

This work would not have been possible without a special few people supporting me, including my advisors Professors Saugata Ghose and Tianyin Xu, friends, and family.

I owe my sincere gratitude to Professor Saugata Ghose. Saugata introduced me to operating systems research and gave a naive undergraduate student a chance to fly over seven seas to work with him back in 2018 at Carnegie Mellon University, which changed the trajectory of my life. He has taught me everything I know about performing research and persevering through challenges. Without his unwavering support and belief in me, I would not have pursued an MS or a career in systems. I will always be grateful for his patience and compassion toward me. I am also incredibly lucky to have Professor Tianyin Xu as my co-advisor. Tianyin's deep knowledge of how real-world systems work and his ability to identify key insights through a sea of complicated and cluttered variables has never ceased to amaze me. I will always cherish our conversations around building better real-world systems in academia or otherwise, and imbibe these learnings within me.

I would also like to thank Dr. Ranjal Gautham Shenoy, who was my mentor during my tenure at IBM. Gautham's knowledge of scheduling and synchronization is unparalleled. I truly admire his ability to explain complex concepts of real-world operating systems and architecture to a recent college graduate like myself with ease. Thank you for championing my eccentric ideas and working with me to realize them.

On a personal note, words would not suffice to express how lucky I am to have my mother, Jyoti Rajesh Sampat, my father, Rajesh Gokaldas Sampat, and my sister, Khushbu Kajaria, who instilled values of hard work and constantly supported me, giving me the confidence to pursue my dreams. Thank you, my incredible partner, Nikita Jaiswal, for your companionship and inspiring me to do better each day. Lastly, my experience at Illinois would have, of course, not been the same without my friends Shaurya Gomber, Nirav Diwan, and Mukul Kaushik, for all the fun and meals that we have shared.

TABLE OF CONTENTS

CHAPTER 1: INTRODUCTION

Precise allocation of CPUs is crucial in the cloud setting. Cloud providers can utilize this information to efficiently provision these resources and schedule applications accordingly. As a result, cloud users experience predictable performance and costs while running their workloads. Unfortunately, a poor estimation of CPU limits can lead to either a loss in performance due to *throttling* or *CPU slack*, where allocated but unused CPUs collectively lower the efficiency of the data center. Estimating CPU limits upfront, however, is non-trivial, especially in dynamic behavior settings where load spikes may occur in an instant. Traditionally, the user mitigates this by over-allocating resources to prevent performance penalties. However, this leads to significantly higher costs and poor utilization of the data an analysis from Google's clusters [1] observing an average CPU utilization of 60% and Alibaba [2] observing the average CPU utilization to not exceed over 40%.

In today's cloud landscape, users widely adopt *autoscaling* techniques to optimize resource utilization and cost efficiency. Autoscaling is the process to automatically requesting for resources when the applications require it, so that applications do not experience a slowdown. Major cloud providers like Google [3], Amazon [4], Microsoft [5], and IBM [6] offer autoscaling capabilities for resource management that observe application behavior metrics in userspace to suggest resource recommendations. The Kubernetes Vertical Pod Autoscaler [7] (K8s VPA) has emerged as a popular open-source solution in industry. K8s VPA continuously monitors resource utilization within defined windows, providing recommendations based on statistical analyses such as the 50th and 95th percentiles of past CPU usage over a window. Cloud providers may employ various techniques, including rules-based tuning and machine learning algorithms, to accurately estimate and suggest CPU allocations for optimal performance and cost savings.

We observe, that there exists a fundamental discrepancy between cloud interfaces and the OS cgroup interface. The Cgroup file system is an interface exposed by the OS to regulate the utilization of resources by tasks groups bound by it. The Linux cgroup bandwidth controller necessitates CPU limits to be specified in terms of quota and period bandwidth. However, cloud users typically request CPU limits using a *millicpu* (i.e., a *vCPU*), which represents a percentage ratio of quota to period. Presently, leading autoscaling solutions focus solely on adjusting the quota based on observed metrics, while the period often remains constant, typically defaulting to 100 ms. As we discover in this work, this approach leads to inaccurate resource estimations, resulting in suboptimal performance with high levels of throttling, and potentially leading to recommendations of unnecessarily high amount of vCPUs (i.e over-

entitlement/allocation) in the following periods.

New autoscaling techniques have emerged [3, 4, 5, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18], aiming to enhance traditional threshold- and utilization-based methods. These approaches leverage queuing theory, reinforcement learning, and introduce novel metrics such as performance awareness. Despite these advancements, these techniques (1) do not account for tuning both the quota *and* the period; and (2) encounter issues related to reactivity and precision when making recommendations, further lowering their efficacy. Due to these inefficiencies, some cloud developers have requested interfaces [19] from orchestrators such as Kubernetes to manually tune both the quota and period, while other users have called for not using the quota–period mechanism altogether [20, 21, 22] for when dealing with dynamic workload when their periodicity is not close to default.

The core problem is that userspace autoscalers rely on surrogate metrics as a proxy to model the application behavior. The application runtime behavior, while opaque to the userspace, is fairly transparent within the OS kernel. A task's behavior in terms of admittance, runtime, and yield can be traced within the scheduler, which can be further be used to accurately model the current application behavior. We propose *CATCloud*, an in-kernel CPU auto tuning solution that profiles the OS scheduler to extract the application behavior and recommend quota and period based on past behavior. CATCloud presents a lightweight approach to tracing and recommendation by minimally instrumenting the Linux bandwidth scheduler to track runqueue statistics of admittance, runtime, yield and collates them at the cgroup level. CATCloud can operate in the granularity of milliseconds and employs simple statistical techniques to recommend future CPU limits.

Our proposed solution addresses two primary challenges. First, accurately modeling runtime behavior proves to be a complex task. Instead of relying on surrogate metrics such as CPU utilization or throttle count, CATCloud extracts application runtime behavior directly from the OS scheduler. While the scheduler manages operations such as running and yielding, extracting these metrics and modeling application behavior presents difficulties. Tracing bandwidth runtime poses challenges due to the scheduler's nature of task-switching based on metrics such as context switches, IO wait times, and vruntime slice expiration. These fragmented runtime and yield slices fail to accurately represent actual application runtime behavior. Initially, we outline a straw man approach involving *period-bound tracing* to gather this information, and highlighting its limitations using micro-benchmarks. Subsequently, we introduce a novel technique called *period-agnostic tracing* to track fragmented runtime and yield data. This data is then aggregated across multiple cores where the application concurrently runs.

Second, contemporary CPU autoscaling solutions often rely on heavyweight algorithms

such as machine learning and reinforcement learning. These algorithms demand extensive telemetry data and time to build models, resulting in slow and sluggish reactivity (on the order of several minutes) when suggesting recommendations. This delay becomes particularly undesirable when dealing with workloads exhibiting high levels of dynamic utilization. CATCloud offers a contrasting approach with its lightweight tracing infrastructure, enabling millisecond-scale tracing. This results in significantly improved reactivity when observing spikes in application load. Moreover, CATCloud empowers users to fine-tune the level of reactivity based on their domain-specific knowledge of application behavior.

We implement CATCloud as an extension to the Linux bandwidth scheduler on a real system, providing CATCloud interfaces through the Linux CPU cgroup. These interfaces can be leveraged by container orchestrators to enable monitoring and recommendations. To assess CATCloud's effectiveness, we deploy a custom Linux kernel on QEMU x86 Kernel Virtual Machine (KVM) [23] and utilize Kubernetes as the cloud orchestrator. Our experiment encompasses a diverse range of cloud applications [24, 25, 26], including microservices, streaming, and serverless architectures, each exhibiting varying degrees of dynamic behavior.

In our evaluation, CATCloud surpasses state-of-the-art autoscaling techniques such as Holt–Winters exponential smoothing (HW) and Long Short-Term Memory (LSTM) [13], and Kubernetes Vertical Pod Autoscaler [7]. CATCloud demonstrates notable improvements in CPU efficiency, achieving on an average of 27.8%, up to 81.3%, and delivering performance gains on average of 27.91%, up to 152.5% compared to the evaluated baselines.

CHAPTER 2: BACKGROUND AND MOTIVATION

In this chapter, we outline the interfaces responsible for allocating CPU resources for cloud applications. First, we delve into the abstractions provided by the operating system (OS) and describe the disparity in the way CPU resources are allocated by container orchestrators. Next, we introduce Vertical CPU autoscaling, one of the most prevalent methods for CPU resource allocation, and examine the current state of autoscalers. Lastly, we illustrate the ramifications of this mismatch through a simple benchmark to motivate our solution.

2.1 OS ABSTRACTION: CPU BANDWIDTH

The Linux operating system implements functionality to limit CPU utilization of certain task groups in terms of bandwidth time. This mechanism allows for better performance isolation and resource accounting between tasks groups, which are crucial pillars of the pay-per-use cloud model.

The CPU bandwidth [27] is a function of quota and period that can be tuned by userspace applications via the control group (cgroup) [28] interface.

- *Quota:* maximum amount of CPU time that a group of processes can consume.
- *Period:* duration for which the quota is enforced.

At a high level, by utilizing both quota and period in tandem, the maximum amount of time an application can run is determined. If the application exhausts its runtime quota, the scheduler throttles the application until the next period begins. Once the new period starts, the quota is reset, and the application can once again accumulate runtime.

The simplest implementation of this mechanism is to maintain a global quota for a task group in which threads use hard synchronization mechanisms to coordinate with global to ascertain their utilization and limits. The global store however can face severe contention as a task group and its threads scale. Another approach is to maintain a local quota limit at a per-CPU level to regulate runtime. If an application spawns threads spanning multiple CPU runqueues, this mechanism also coordinates with other local quotas. The advantage of this approach is that it can account for its own utilization locklessly. In practice, this approach of having quotas locally for each task group does not scale well due to the computational overhead of the many to many relationships with other groups to compute remaining quota. The Linux kernel scheduler features several optimizations in its design to primarily cater to

these scalability challenges by using a hybrid global-local pool mechanism [29]. The following new parameters are implemented internally within the scheduler:

- *Global quota:* quota consumed tracked at the task group level
- *Local quota:* local pool of quota for each run queue.
- *Bandwidth slice:* a user configurable batch size of runtime. The local quota borrows runtime from the global in these fixed batch sizes.

Summarizing the working of the modern Linux bandwidth scheduler, when a task group is restricted using the quota and period, the cgroup quota is assigned as the global quota of the group. Each runqueue spawned possess a local quota. The local quota borrows runtime, in the form of small batched bandwidth slices, from the global pool to enable running of the tasks. When the local pool is exhausted, more runtime is requested from the global pool. If the global pool runs dry, then the runqueue of the local quota is throttled for the duration until the next period. This hybrid approach of global-local accounting is akin to real-time scheduling, but with the key difference of avoiding many-to-many CPU interactions on refresh and expiration therefore providing better scalability. In subsequent sections, we describe the implications of this design on accurately estimating application runtime behavior.

In Figure 2.1, we show the controller in action for a single period in the Completely Fair Scheduler (CFS). According to the example, the user spawns a task group and assigns a quota of 15 ms, a period of 100 ms, and a bandwidth slice of 5 ms. Within the scheduler, the global quota of the cfs_bandwidth structure is set to 15 ms as well. The cfs_rq run queue structure houses the local quota. Initially for the runqueue to be first scheduled on the system, the local quota borrows 5 ms slice from the global quota, reducing the global quota to 10 ms. The task is then run (highlighted in green in the figure) for the bandwidth slice amount of time, and yields (highlighted in blue) when the local quota is exhausted. If the run queue requires more quota, it is again requested from the global quota, further reducing the global pool to 5 ms. Once the global pool is exhausted and the task still requires more CPU time, the bandwidth controller throttles (highlighted in red) the application for the duration of the period. In this case, our example granted a total of 15 ms of runtime in slices of 5 ms each, and now that the quota is exhausted, the application is throttled for 85 ms (i.e., the remainder of the 100 ms period).

By default in the scheduler, the bandwidth slice is set to 5 ms; the period is set to 100 ms; and the quota is set to max, which lets regular applications run unrestricted. In addition to CPU limit, the cgroup also provides statistics regarding usage, pressure, throttle, etc.

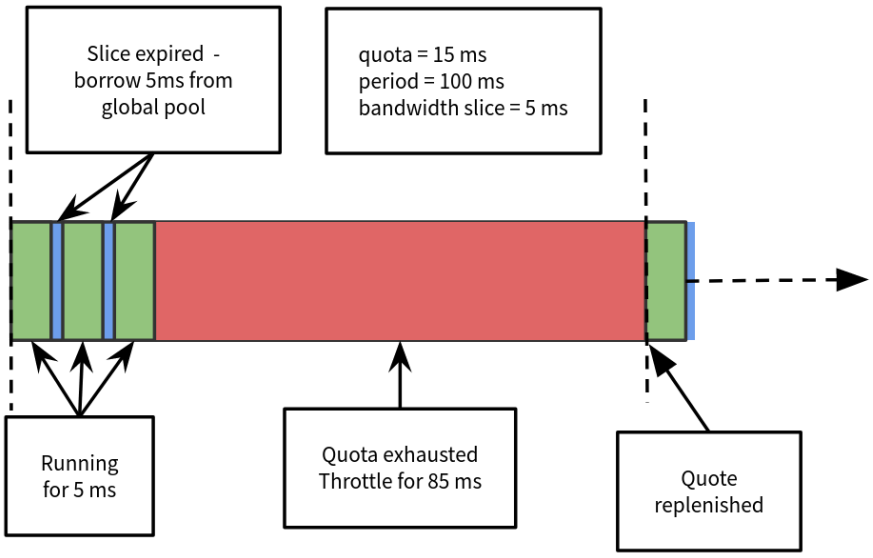

Figure 2.1: Linux bandwidth controller

2.2 CLOUD ABSTRACTION: MILLICORE

User applications typically request CPU resources in terms of cores or logical CPUs. However, in a multi-tenant cloud environment, applications must interact with the CPU bandwidth interface to manage their CPU resource requirements. This presents an interface mismatch, as applications request limits in logical CPUs, while the operating system expects limits to be provided as a quota–period pair.

To address this interface discrepancy, cloud orchestrators and vendors offer abstractions to translate between logical CPUs and CPU bandwidth. Various vendors employ different terms (e.g., millicpu, millicore, vCPU) to fulfill the same functional purpose. We focus on Kubernetes [30] as our orchestrator of choice, which uses the CPU limit interface to assign millicores, with autoscaling as the preferred choice of managing the CPU resource.

Kubernetes Container Orchestration. In production cloud environments, containers are a prevalent deployment method. Kubernetes (K8s) serves as an orchestration platform designed for automating scaling, deployment, and management of containerized applications. The fundamental unit in Kubernetes is the pod, which encompasses the containers of an application with defined resource limits. CPU resource limits can be specified in Kubernetes either as cpu-set CPUs [28], where pods are pinned to a specific set of CPUs for improved performance isolation, or more commonly in multi-tenant environments, as millicores [31].

Millicpu or Millicore. The limits to the CPU resource in a cloud setting is defined as a unit of millicpu or millicore. A millicpu is defined as a thousandth part of a physical CPU worth of runtime. Within the OS, millicpu translates to the ratio of quota to the period. For example, 500 millicpu = 0.5 vCPU = 50 ms of quota, for a 100 ms of period. When a user adjusts millicpu limits, the period always remains constant, typically set to the default 100 ms, while the quota is adjusted proportionally to maintain the specified ratio provided by the user.

Vertical Pod Autoscaler. In the cloud model, users typically pay for the resources they request. Therefore, it is crucial that the requested resources meet the application's performance requirements at the lowest possible cost. Additionally, applications may exhibit dynamic CPU utilization patterns due to varying loads. This necessitates periodic adjustment of CPU requests to optimize performance per dollar spent. To automate this process and avoid manual estimation of resource limits, Kubernetes provides functionality to *autoscale* (i.e., automatically adjust) resource limits over time based on application behavior. Specifically, for CPU resources, the Kubernetes Vertical Pod Autoscaler (VPA) [7] continuously monitors resource utilization within defined time windows and recommends millicpu adjustments based on statistical analyses, such as the 50th and 95th percentiles of past CPU usage.

Kubernetes also allows for custom autoscalers that can recommend and tune for CPU limits based on the application behavior. This include using various techniques [8, 9] to dynamically tune CPU limits. These approaches include rule- and threshold-based [4, 5], statistical models [7, 10, 11], time series / machine learning / reinforcement learning techniques [3, 12, 13, 14], queuing-theory-based [15, 16, 17], and performance-aware models [18]. Each approach presents unique advantages and challenges (Chapter 5) that cloud vendors account for while choosing their environment's autoscaler.

2.3 IMPLICATIONS OF INTERFACE MISMATCH

As discussed in the previous section, various techniques exist for tuning CPU requirements using millicpu units. Tuning millicpu involves modifying only the quota variable, while the period remains constant, typically set to the default 100 ms in the Linux kernel. However, applications have started encountering issues due to the oversight of their periodicity when tuning millicores. An example of this was uncovered using production traces of the Golang garbage collector running in a CPU bandwidth-shared environment [32]. High levels of throttling were attributed to the use of the default period, as the workload's periodic nature

resulted in short bursts of runtime. Similar behavior was reported by other users [20], prompting the Kubernetes orchestration framework to implement the capability to tune both the quota and the period [19].

Although Kubernetes provides the interface to adjust both the quota and period, most users and autoscalers do not utilize this feature. The consequences of setting incorrect bandwidth limits are often manifested in performance-related metrics (e.g., service level objectives, or SLOs) and the frequency of throttling. In such cases, autoscalers often compensate by requesting more millicpus, resulting in degraded performance and/or inefficient CPU resource utilization.

2.3.1 Limitations of Quota-Only Models

To illustrate the consequences of adjusting only the quota, we conduct two experiments using the Ebizzy microbenchmark [33]. Ebizzy emulates a typical web-search application workload and features a variant [24] capable of simulating a sleep-wakeup pattern among threads, introducing burstiness. This micro-benchmark enables us to observe predictable and consistent CPU utilization patterns. Leveraging the Kubernetes Vertical Pod Autoscaler (VPA), we analyze application behavior and propose millicpu allocations. Furthermore, we present scenarios of under-provisioning and ideal bandwidth limits. The following experiments represent a diverse set of applications that may demonstrate varied periodicity as seen often in real-world [34]. The aim of these experiments is to highlight the modern autoscaling's inability to account for an application's periodicity and the resulting implications, even when the workload demonstrate consistent utilization.

Experiment #1; Periodicity Greater Than Default

Consider a scenario where the Ebizzy micro-benchmark is configured to exhibit a periodicity greater than the default 100 ms. In this experiment, we define the runtime of an Ebizzy thread to be 20 ms, followed by a yield for 150 ms before servicing another set of requests for 20 ms.

When using the default periodicity of 100 ms, the workload illustrated using Figure 2.2 behaves as follows:

- **1st period:** 20 ms of runtime and 80 ms of idle time observed.
- **2nd period:** The remaining 70 ms of idle time from the previous period carries over, and then the application runs for 20 ms, with the remaining 10 ms occupied by idle

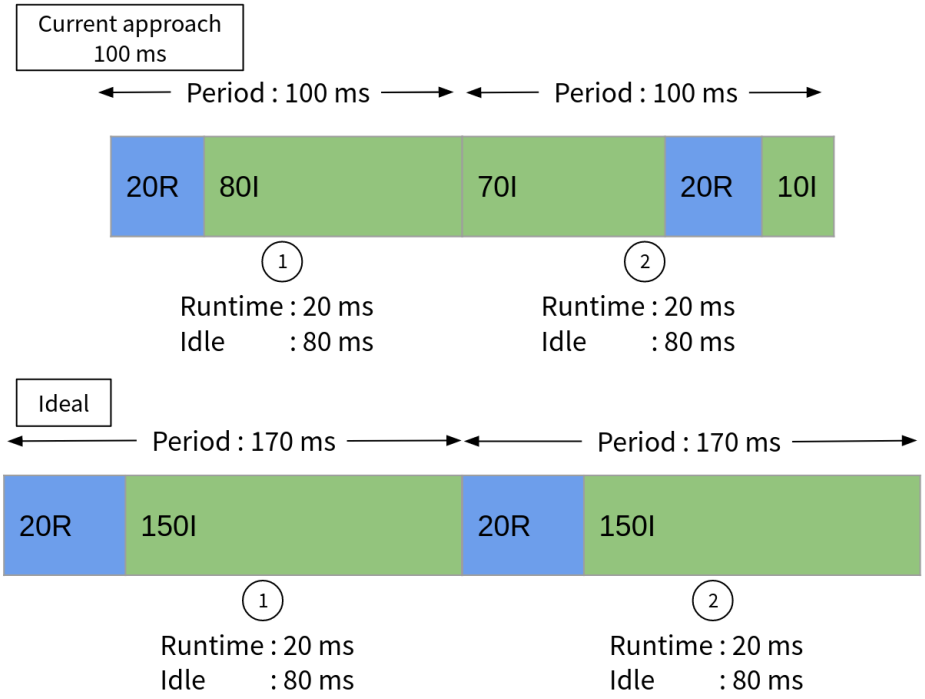

Figure 2.2: Exp #1 Illustrative example - workload runtime 20 ms, idle time 150 ms

time.

Based on this observed behavior, manually estimating CPU bandwidth limits would result in a quota of 20 ms and a period of 100 ms, equating to 200 millicores.

However, knowing that this instance of Ebizzy has a yield duration of 150 ms, setting the periodicity to the sum of the runtime and yield (170 ms in this case) leads to a different workload behavior. In this scenario, each period of 170 ms accumulates 20 ms of runtime and experiences 150 ms of idle time. This equates to a requirement of 117 millicores.

From Figure 2.3, we first observe that in an under-provisioned scenario when an application is limited to 100 millicores, the response time latency is large due to the throttling it experiences. The second case wherein the Kubernetes Vertical Pod Autoscaler is employed, we are able to achieve ideal latency characteristics, albeit with double the number of millicores than were allocated in the under-provisioned case (200 millicores). Even though optimal latency is achieved, since the workload behavior is known, further optimizing for period can yield better CPU allocation. Therefore in the third case, setting the quota to 20 ms and period to 170 ms (which allocates 117 millicores) also achieves ideal latency but at a significantly lower lower CPU cost (41.5%) than the recommendations of K8s VPA.

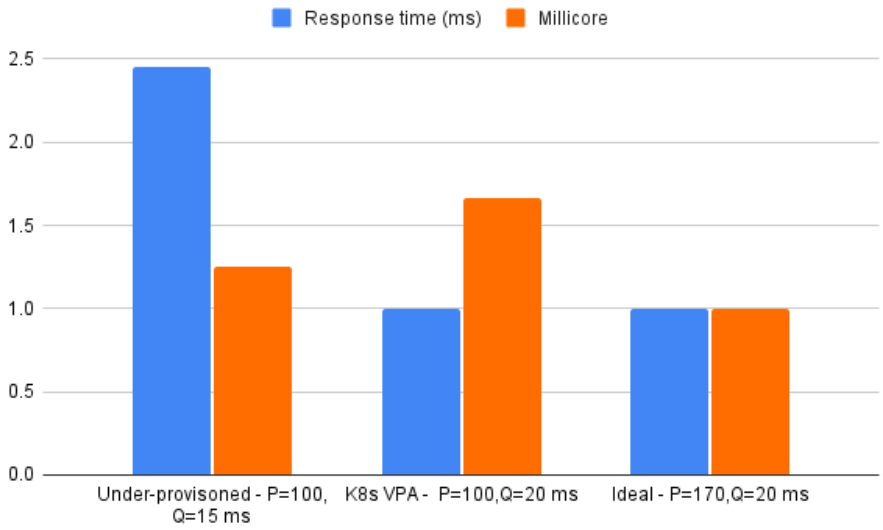

Figure 2.3: Ebizzy micro-benchmark - 20 ms work, 150 ms idle - normalized. Tuning for a higher periodicity aids in lowering CPU requirement without the compromise of ideal performance

Experiment #2: Periodicity Lower Than Default

To illustrate the implications of a workload with periodicity lower than the default 100 ms, we define the runtime of an Ebizzy thread to be 40 ms, followed by a yield for 30 ms.

When using the default periodicity of 100 ms, the workload (Figure 2.4) behaves as follows:

- **1st period**: 40 ms runtime, 30 ms yield time, followed by 30 ms runtime. Cumulative runtime = 70 ms
- **2nd period**: residual 10 ms runtime, 30 ms yield time, 40 ms runtime, and 20 ms. Cumulative runtime = 50 ms
- **3rd period**: residual 10 ms yield time, 40 ms runtime, 30 ms yield time, and 20 ms runtime. Cumulative runtime = 60 ms

Based on this observed behavior, manually estimating CPU bandwidth limits proves to be challenging. A user may choose the lowest limits of quota = 50 ms (500 millicores), the average at quota = 55 ms (550 millicores), or the 99th percentile at quota = 70 ms (700 millicores), with the period remaining constant at 100 ms. The 99th percentile is also the recommendation observed from Kubernetes VPA. However, tuning the CPU limits to match the ideal periodicity of the application leads to a quota of 40 ms and a period of runtime + yield = 70 ms (571 millicores) for all accumulated runtime within each period.

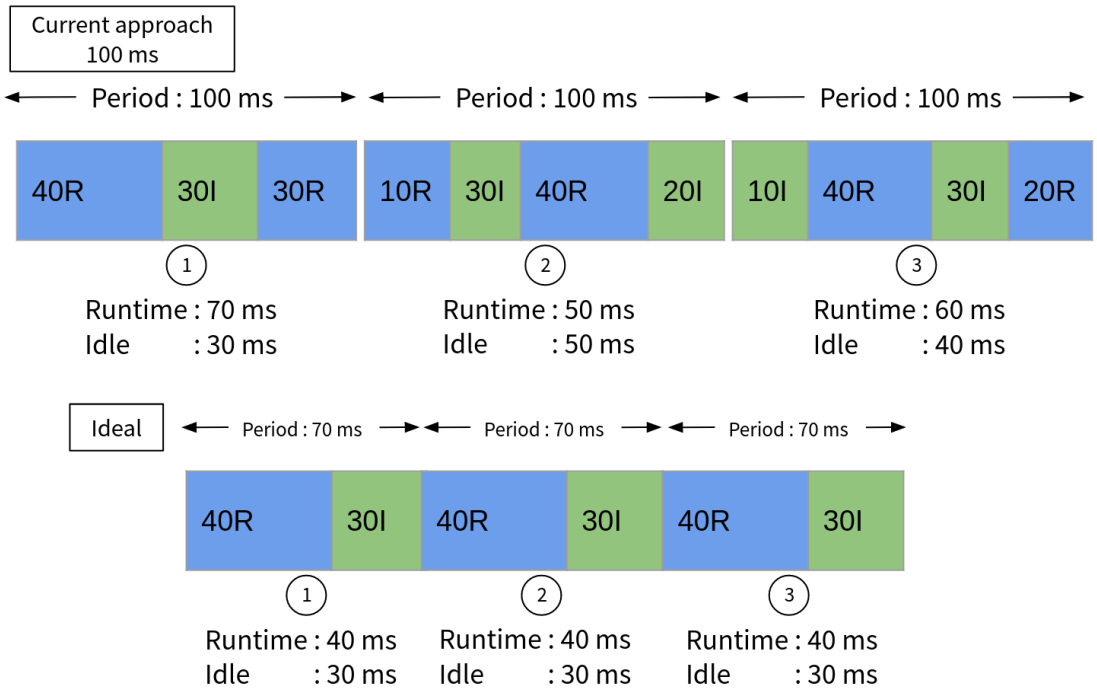

Figure 2.4: Exp #2 Illustrative example - workload runtime 40 ms, idle time 30 ms

Figure 2.5 presents the latency characteristics for the various CPU limits discussed above. Initially, in an under-provisioned case (400 millicores), poor response time latency is observed. Increasing the quota in subsequent runs leads to improved latency. The lowest achievable latency under a fixed period of 100 ms is attained when the quota is set to 70 ms (700 millicores). Finally, when tuning the quota to 40 ms and the period to 70 ms according to the application behavior, ideal latency is observed at a lower cost (570 millicores) than the 99th percentile case.

All of the performance and efficiency issues observed from both trace-based studies and micro-benchmarks stem from the fundamental discrepancy of not tuning the quota–period pair in tandem. This discrepancy arises because current autoscaling solutions cannot observe the entirety of application behavior in the userspace, and only see single-period snapshots. As a result, they rely on surrogate metrics to estimate this behavior. However, this inability to directly observe, coupled with the interface mismatch between the bandwidth controller and the cloud, leads to recommendations that may appear correct under the context of the default period but are often not the most efficient allocation. This highlights the need to design a solution at the layer where collecting this information is fairly transparent: the OS scheduler. The OS scheduler processes data regarding an application thread's runtime, yield, and throttle behavior. Any solution operating within the kernel's scheduler must not impose significant computational overhead and should be reactive to millisecond load

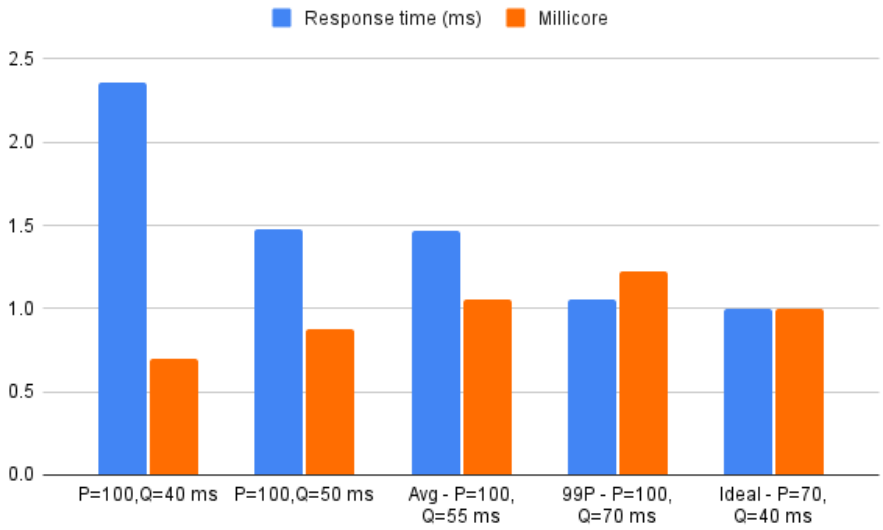

Figure 2.5: Ebizzy micro-benchmark - 40 ms work, 30 ms idle - normalized. Tuning in accordance to lower periodicity affects both, quota-period combination and aids in lowering CPU requirement without the compromise of ideal performance

changes. Additionally, it should possess the capability to bridge the interface gap between cloud orchestrators and the bandwidth controller.

CHAPTER 3: CATCLOUD DESIGN

CATCloud is a vertical CPU autoscaling solution designed for applications that rely on the Linux cgroup CPU bandwidth controller to regulate utilization (e.g., containerized applications on multi-tenant cloud). Drawing from our previous discussion, where we highlighted the limitations of monitoring application behavior solely in userspace, we now bypass the necessity for userspace surrogate metrics. Instead, we seamlessly integrate directly into the kernel scheduler to extract the operating system's view of the application, which is opaque to other autoscaling solutions. Residing in kernel space, CATCloud minimally instruments the Linux scheduler to trace and analyze application runtime behavior. It also extends the CPU cgroup by exposing a userspace interface to finely tune the granularity of profiling by determining the length of observed history. In addition to profiling, CATCloud also exposes CPU limit recommendations and adjustments based on observed history and gives an option to automatically tune the CPU bandwidth based on these recommendations.

Design Goals : CATCloud has three main design goals:

- *Correctness in capturing application behavior.* CATCloud must be able to accurately capture application runtime and idle-time behavior within the scheduler and must account for events such as throttle.

- *Lightweight profiling and recommendations.* Since CATCloud is designed as an extension to the OS scheduler, it must be computationally inexpensive in order to not slow down the decisions made by task scheduler. Therefore CATCloud's profiling must be simple and minimal.

- *Usability.* Users must be able to provide hints to tune the reactivity of tracing and recommendations, along with the ability to extract and apply recommendations on the fly. These userspace hints can help reduce the overheads of tracing as well as improve the quality of the recommendations. CATCloud must implement a userspace interface that is in tandem with the already established CPU cgroup interface

3.1 OVERVIEW

CATCloud's workflow comprises three essential components: profiling application behavior, analyzing collected runtime data, and deploying CPU limit recommendations. As discussed in Chapter 2.1, within a bandwidth-controlled environment, the combination of quota

period and its ratio determines the CPU allocation for applications. In contrast to other autoscaling solutions, CATCloud fine-tunes both the quota and period parameters to achieve precise and efficient entitlement for the tasks within that cgroup.

Most VPAs rely on proxy metrics, often leading to ineffective profiling of application runtime and periodicity. This results in incorrect and inefficient CPU entitlement. CATCloud addresses this challenge by leveraging the Linux scheduler to profile each runqueue scheduled by the application's cgroup. Each runqueue records the duration it runs before yielding, storing both runtime and yield time in a history buffer. Simultaneously, a global view of the cgroup runtime within a period is gathered. Once the history buffer is filled with data from both per-runqueue and global cgroup views, CATCloud computes the worst-case run/yield times and recommends a quota–period pair based on that. These recommendations are then exposed to the userspace via the CPU cgroup. CATCloud's lightweight implementation in the OS enables millisecond-scale accounting, ensuring high reactivity when sudden load spikes and drops are observed.

3.2 PROFILING

To accurately model the current application behavior, we need techniques to observe its runtime and yield interactions within the scheduler. As described from Chapter 2.1, with the introduction of hybrid global–local quotas and a system of borrowing runtime as a unit of scheduler slices, the behavior of the request queue (RQ) during execution is fragmented (Figure 3.1). This fragmentation complicates the distinction between actual runtime and involuntary yields. Consequently, we propose two strategies. First, a straw man approach involves tracing runtime within a defined period. We discuss its similarities with existing tracing methodologies, its limitations, and its applicability in specific scenarios. Second, we introduce a novel tracing methodology capable of capturing application behavior agnostic to the period in which the runtime resides.

3.2.1 Period-Bound Tracing: Straw Man Solution

Our approach for period-bound tracing is simple: we account for the cumulative runtime within a given period currently set by the cgroup controller. This approach is akin to the CPU utilization reported by container telemetry tools, with the key difference that the data is collected in-kernel and accounted for at every single period.

Described in Algorithm 3.1, only the runtime duration is accounted for, while the yield (idle) duration is implicit at the period boundary. For example, in a 100 ms period, if the

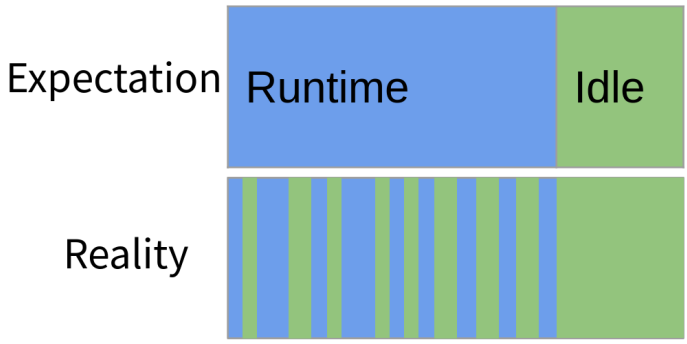

Figure 3.1: Challenges with application behaviour tracing

Algorithm 3.1 Period-Bound Tracing

Require: Task bound by the CFS Bandwidth controller
Require: $quota! = max$
Require: $recommend.status == true$
  $tot\_runtime \leftarrow 0$
  $yeildtime \leftarrow 0$
  while $curr\_period \leq default\_period$ **do**
    if $target\_runtime$ is assigned **then**
      $tot\_runtime \leftarrow tot\_runtime + runtime\_slice$
    end if
  end while
  $yeildtime \leftarrow period - tot\_runtime$

cumulative runtime slices accounted for is 30 ms, then the yield duration is 100 ms - 30 ms = 70 ms. The implications of this inability to consider the period lead to incorrect allocation as demonstrated in Section 2.3.1. Despite these drawbacks, the approach does present utility in cases when applications experience moderate amounts of throttle, which is described in Section 3.2.3

3.2.2 Period-Agnostic Tracing

While our implementation of period-bound tracing in the last section demonstrates a lightweight and simple approach, it presents significant shortcomings when the periodicity of the applications is not in line with the default period. Therefore, in this section, we design a period-agnostic tracing model to profile application runtime and yield-time behavior whose periodicity can either be lower or greater than the currently-defined period boundaries.

The core idea described in Algorithm 3.2 revolves around accounting for runtime per runqueue until the task self-yields. However, runqueues can yield for various reasons, with

Algorithm 3.2 Period-Agnostic Tracing

Require: Task bound by the CFS Bandwidth controller
Require: $quota \neq max$
Require: $recommend.status == true$
  while $\_\_assign\_cfs\_rq\_runtime()$ **do**
    if $yieldtime \neq 0$ **then**
      $curr\_yieldtime \leftarrow rq\_clock() - yieldtime$
    end if
    $corr\_yieldtime = curr\_yieldtime - prev\_vruntime$
    if $accumilated\_runtime \neq 0$ and $corr\_yieldtime \geq cfs\_bandwidth\_slice$ **then**
      $yeildtime\_hist[idx] \leftarrow corr\_yieldtime$
      $runtime\_hist[idx] \leftarrow rq\_clock() - accumilated\_runtime - corr\_yieldtime$
      $idx \leftarrow idx + 1$
      $accumilated\_runtime \leftarrow 0$
    end if
    $curr\_yieldtime \leftarrow rq\_clock()$
    $prev\_vruntime \leftarrow vruntime$
    if $accumilated\_runtime == 0$ **then**
      $accumilated\_runtime \leftarrow rq\_clock()$
    end if
  end while

the most common being the expiry of the CFS bandwidth slice. Therefore, our primary approach is to account for runtime when the yield duration exceeds the scheduler bandwidth slice. To implement this approach, we maintain two runqueue clocks: one for runtime and another for yield time. The yield-time clock is initialized each time a runqueue is assigned a vruntime. The runtime clock is initialized either if it was not previously initialized, or when a legitimate self-yield is detected. In a low-contention scenario, when a runqueue has exhausted its CFS bandwidth slice but still needs to run and has remaining quota, it will be assigned runtime to be scheduled in the next bandwidth slice. Hence, to identify a self-yield, we observe when the runqueue is scheduled and check if the yield-time clock exceeds the scheduler bandwidth slice. This, coupled with the absence of throttling in previous periods and the deduction of the current yield duration, signifies the current periodic runtime cycle.

3.2.3 Tracing During Throttle

When a task requires more runtime than the user-set quota within a period, it will be involuntarily yielded due to throttle. This means that the task is evicted from the queue for the rest of the period and is re-queued only in the next period when the quota replen-

ishes. This can significantly change the run–yield behavior and must be accounted for when estimating the CPU bandwidth requirements.

We characterize throttle behavior into three main categories (light, medium, and high degrees of throttle) and describe strategies for tracing during each of these conditions. During light throttle, the period-agnostic algorithm will experience higher runtime and yield-time durations, potentially leading to incorrect recommendations. To accurately model the runtime behavior, we must take into account the amount of time the task was throttled and make corrections based on that. As described in Algorithm 3.3, we maintain a throttle clock to record the amount of time a task stayed throttled. When the task is reset in the queue to be unthrottled, we adjust the yield-time and runtime clocks forward by the duration the runqueue was throttled for.

Algorithm 3.3 Throttle Correction

Require: $quota \neq max$
Require: $recommend.status == true$
  procedure THROTTLE_CFS_RQ(cfs_rq)
    $cfs\_rq.throttled\_clock \leftarrow rq\_clock(rq)$
  end procedure
  procedure UNTHROTTLE_CFS_RQ(cfs_rq)
    $curr\_throttle\_time \leftarrow rq\_clock(rq) - cfs\_rq.throttled\_clock$
    if $yieldtime \neq 0$ and $accumulated\_runtime \neq 0$ **then**
      $yieldtime \leftarrow curr\_throttle\_time + yieldtime$
      $accumulated\_runtime \leftarrow curr\_throttle\_time + accumulated\_runtime$
    end if
  end procedure

Experiencing some degree of throttle is unavoidable in the life cycle of an auto-tuned application, and Algorithm 3.3 effectively mitigates the effects of capturing the runtime duration. However, in scenarios where an application experiences moderate levels of throttle, temporarily altering its behavior, period-bound tracing can provide a useful recommendation baseline. As discussed earlier, period-bound tracing may sometimes lead to higher, incorrect entitlements, especially when behavior is altered. Nevertheless, it can significantly help in promptly reducing throttle and stabilizing the workload for more accurate runtime profiling.

Lastly, if an application experiences high degrees of throttle, then the application behavior itself can be severely affected, and correction factors may prove to be inadequate. Therefore, as described in Algorithm 3.4, when throttle constitutes the majority ($\geq 50\%$) of the period-bound history size, then for a short duration of time (five bandwidth periods), the quota is scaled to over-provision the application's millicore allocation to stabilize the tracing. The

amount of over-provisioning is determined by an exponential scaling factor of CPUs. Once a valid runtime is discovered, the over-entitlement-based tracing is disabled.

Algorithm 3.4 Quota Scaling

Require: $quota \neq max$
Require: $recommend.status == true$
  $ulim\_interval \leftarrow 5$
  $curr\_throttle \leftarrow 0$
  $cpu\_scale \leftarrow 1$
  procedure DO_SCHED_CFS_PERIOD_TIMER
    $throttled = !list\_empty(throttled\_cfs\_rq)$
    if $curr\_throttle \geq period\_bound\_history$ **then**
      if $ulim\_interval \geq 0$ **then**
        $quota \leftarrow quota + cpu\_scale$
        $ulim\_interval \leftarrow ulim\_interval - 1$
        $cpu\_scale \leftarrow cpu\_scale * 2$
      else
        $ulim\_interval \leftarrow 5$
        $cpu\_scale \leftarrow 1$
      end if
    end if
  end procedure

3.3 CPU PERIOD AND QUOTA RECOMMENDATION

Runtime information for both period-based and period-agnostic tracing is stored in history buffers. When the user-tunable history buffers are full, the decision-making algorithm considers this past behavior to determine the best period and quota limits for the future.

To determine the quota–period combination, the 99th percentile of runtime and yield time duration are calculated from both period-based and agnostic history buffers. Quota is attributed to the runtime, while the period is determined by the summation of runtime and yield time. The ratio of quota to period is compared for both period-bound and period-agnostic techniques, and the lower ratio, indicating lower CPU limits, is then recommended to be applied.

If the **recommended.status** CPU cgroup interface (described in Section 3.4) is set to *recommend-only* mode, only the **recommend.max** is updated for the user to manually make a decision based on these insights. If the status is set to *auto mode*, the period and quota of the task are updated automatically as well, and tracing for the following periods is based on this new quota and period.

3.4 MULTICORE SUPPORT

Tasks can often spawn multiple threads simultaneously on multiple cores. Therefore, for period-agnostic tracing, CATCloud profiles all the runqueues (RQs) separately and models the quota and period for each. During each assign_cfs_rq, it goes through the entire list of active RQs and computes cumulative runtime and period, which forms the period-agnostic recommendation. However, as described in the last section, RQs can yield briefly for several reasons from scheduler ticks to throttle. This can lead to some genuine RQs that are active not being present in the list during the collation of all the RQs, resulting in incorrect CPU limit recommendations. To mitigate this behavior, RQs are added to the active list immediately when they appear and are persisted for the entirety of a single period, after which the contents of the active list are purged. This ensures that all active RQ behaviors are considered when a decision is made. One downside of this approach is that RQs that have been dequeued and will not enqueue anytime soon can influence the recommendation and cause over-allocation. However, this case is less frequent, and even in the cases of over-allocation, the effect will be brief due to the dequeued RQ being removed from consideration within the span of a single bandwidth period.

3.5 IMPLEMENTATION

We extend the Linux version 6.3 scheduler by introducing per-bandwidth-controller, per-RQ tracking of runtime and yield. Within the `struct cfs_rq`, we implement heuristics to monitor behavior independent of time periods. Each RQ includes raw current clock values, historical data, and the 99th percentile values of both runtime and yield. Additionally, we extend the `struct cfs_bandwidth` to incorporate similar statistics for period-bound tracing mechanisms. Furthermore, we integrate variables for externally controlled interfaces of status, history, and max values within the same structure.

3.5.1 Interface

We augment the Linux CPU cgroup interface to add userspace knobs to control CATCloud.

```
/sys/fs/cgroup/cpu
├── cpu.recommend.status
├── cpu.recommend.history
└── cpu.recommend.max
```

The extended controls and their definitions are as follows:

- **cpu.recommend.status**: Tribool value [0/1/2]

  - $0 \rightarrow off$: Disable CATCloud
  - $1 \rightarrow recommend\_only\_mode$: Enable tracing, export resource recommendations to cpu.recommend.max
  - $2 \rightarrow auto\_mode$: Enable tracing, and automatically apply period and quota recommendations to cpu.max

- **cpu.recommend.history**:
  - *period-bound history size, period-agnostic history size*

  Size of the history buffer of runtimes for both period-bound and period-agnostic tracing.

  The core idea behind the history sizes is to control the aggressiveness of suggestions for different kind of applications. If the history is set too small, then accurate runtime behavior may not be captured, while if it is set too large, then stale past information may taint the buffer.

- **cpu.recommend.max**:
  - *recommended quota, recommended period*

  Read-only file that mimics the cpu.max file format, and presents the current quota and period recommended by CATCloud.

  The cpu.recommend.max file is updated both in recommend_only mode and auto mode for the interface cpu.recommend.status: the recommendations are only suggested for the former, while they are automatically applied to the bandwidth controller in the latter. In the latter case, cpu.max and cpu.recommend.max will display the same entitlement.

CHAPTER 4: EVALUATION

4.1 METHODOLOGY

Benchmark Applications. We deploy three cloud applications: *(1)* Sleeping Ebizzy microbenchmark [24] to simulate webpage traffic. *(2)* EPFL CloudSuite [25] - web search and media streaming benchmarks, and Hotel-Reservation from DeathStarBench [26]. These applications are representative of real-world web-based cloud applications. These display varied application behaviors from streaming to microservices that demonstrate stateless and data services.

Comparison. We compare CATCloud to *(1)* Kubernetes CPU Vertical Pod Autoscaler (K8s VPA), *(2)* Holt–Winters exponential smoothing (HW), and *(3)* Long Short-Term Memory (LSTM) autoscaler.

K8s VPA [7] periodically (every 15 seconds) monitors the CPU utilization and recommends scale-up or scale-down millicore limits for the pod that it is attached to (every 300 seconds). The autoscaler also maintains a history of past runs and recommends initial limits based on its past run behavior.

We implement the the HW [35] and LSTM [36] strategies presented in [13] and integrate both into a userspace autoscaler for real-time prediction. The implementation uses the weights supplied as-is and makes a recommendation approximately every 3 minutes.

Experimental Setup. We deploy CATCloud on a patched Linux 6.3, on a testbed of x86 KVM QEMU, with 32 cores, and 32 GB of memory. The benchmarks are containerized, and are unrestricted in all resources except CPU allotment (which is autoscaled). Deployments are performed on the original container images and are managed by Docker and Kubernetes. K8s VPA is set up as an add-on to the pod deployment. HW, and LSTM are tested using a custom userspace autoscaler that resides in the host, monitors the telemetry, and applies the recommendation directly on the cgroup interface. In the case of CATCloud, we identify the container's cgroup to be auto-tuned, and activate tracing and recommendation using the cpu.recommend.status knob in the cgroup file system.

4.2 CLOUD BENCHMARKS

4.2.1 Sleeping Ebizzy Microbenchmark

Ebizzy [33] is a web traffic simulation benchmark. We used a customized variant [24] that introduces burstiness to better model real-world behavior. Starting with the experiment described in Section 2.3.1, we evaluate the two baselines against CATCloud (Figure 4.1 for latency, Figure 4.2 CPU limits) to ascertain if our approach is able to attain the hypothetical CPU limits minima while sustaining maximum performance (lowest latency). K8s VPA recommends and auto-tunes the median CPU limits to 200 millicores, while also achieving close to the ideal latency. HW and LSTM perform poorly and require 164 and 175 millicores respectively. Lastly CATCloud, achieves ideal latency, while the median CPU limits recommended are quota = 22 ms, and period = 171 ms (128 millicores). CATCloud offers the same performance characteristics of k8s while reducing the CPU limits by 56.25%. Compared to HW and LSTM, CATCloud performs 150% better in terms of latency, with 22% and 26.8% CPU allocation improvements, respectively.

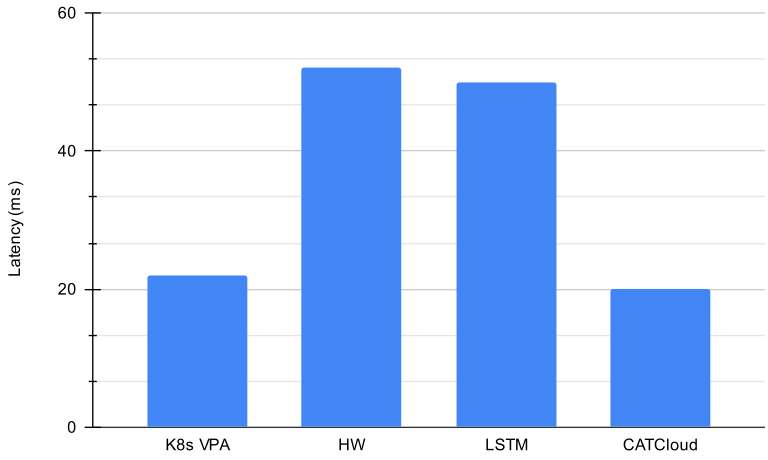

Figure 4.1: Sleeping Ebizzy - latency (lower is better)

4.2.2 EPFL CloudSuite

CloudSuite [25] is modern benchmarking suite that evaluates popular cloud services such as data serving, web search, and media streaming. Within CloudSuite, we evaluate two compute-heavy benchmarks, web search and media streaming, for varying degrees of resource utilization scale. The results are normalized to peak performance for ease of analysis, and

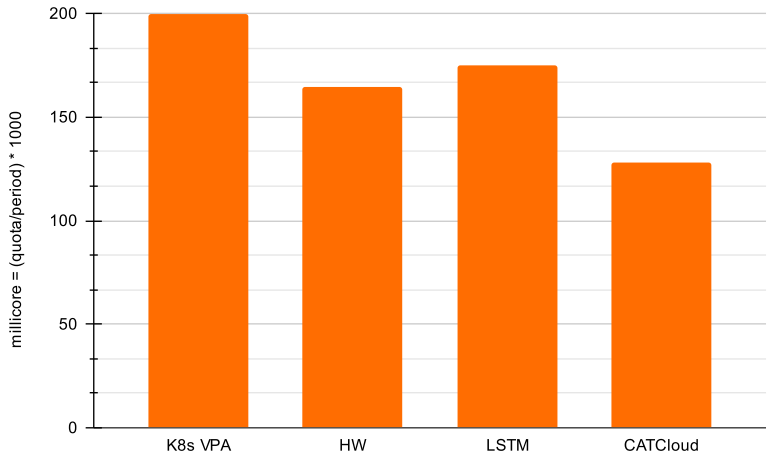

Figure 4.2: Sleeping Ebizzy - CPU entitlement (lower is better)

we evaluate throughput and CPU entitlement.

For a stable running web search benchmark (Figure 4.3), K8s VPA, HW, LSTM, and CATCloud all perform close to peak performance. However, K8s VPA, HW, and LSTM require 20%, 8.6%, and 2.6% more CPUs, respectively, compared to CATCloud. For the media streaming benchmark (Figure 4.4) - K8s VPA performs 152.5% worse in terms of performance, with HW and LSTM performing at par to peak. In terms of efficiency, K8s, HW, and LSTM require 25.3%, 62.1%, and 81.3% higher CPU entitlement, respectively, compared to CATCloud.

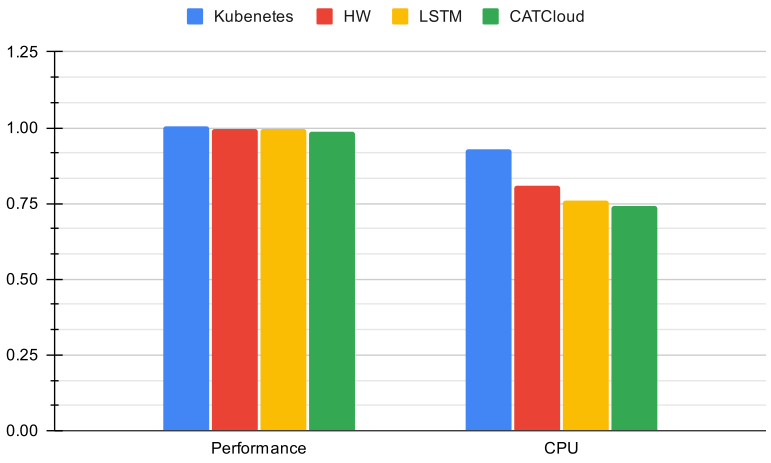

Figure 4.3: EPFL CloudSuite - Web search

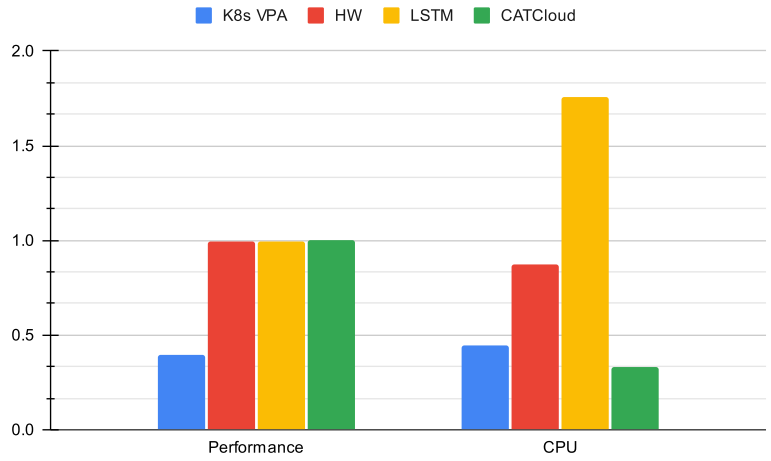

Figure 4.4: EPFL CloudSuite - Media Streaming

4.2.3 Hotel Reservation Microservice-Based Application

The HotelReservation benchmark from the DeathStarBench [26] suite simulates a microservice-based application. The workload contains many microservices that can be either independently or collectively controlled. We choose to control the CPU entitlement recommendation individually based on each microservice (recommend, search, reserve, etc). This allows for a finer granularity in monitoring (for all the baselines), and avoids the tuning and tainting of microservices that display either low or high levels of utilization. The results are normalized for ease of analysis. The metrics of measurement here are P99 (i.e., 99th percentile) latency and CPU entitlement. Note that results displayed are after warmup for all autoscalers. This warmup takes 20 minutes for K8s VPA, 30 minutes for HW and LSTM. CATCloud does not need any warmup or learning time, as it has the ability to trace information at the millisecond scale.

If the results are viewed from the best-case recommendations after warmup from all the autoscalers (Figure 4.5) on a stable load, CATCloud performs better than K8s (10%), HW (11.95%) and LSTM (10.5%) in terms of latency. In terms of efficiency, K8s, HW, and LSTM require 8%, 12.4%, 8.9% higher CPU entitlement, respectively, compared to CATCloud. Note that CATCloud performs the same regardless of the duration it is run for and starts making accurate decisions right from the first few milliseconds.

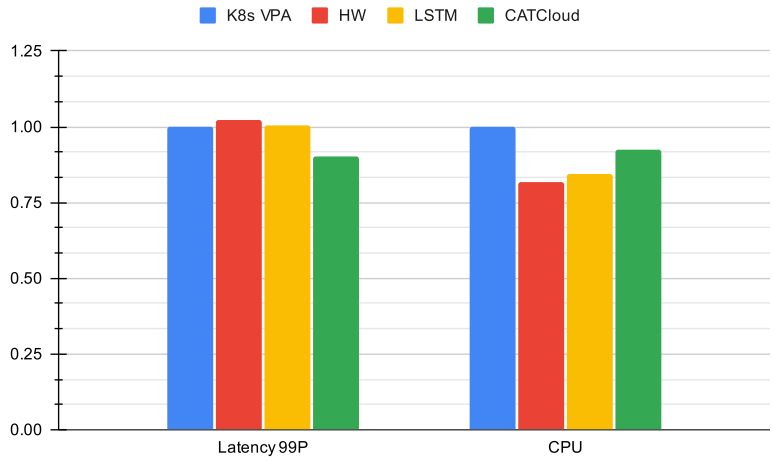

Figure 4.5: DeathStarBench - Hotel Reservation

4.3 OVERHEADS

Autoscalers are widely deployed across multi-tenant clouds and data center applications. Often, they reside on the same machine as the applications they monitor to observe behavior and provide resource allocation recommendations. Quantifying the compute and memory overheads of these approaches is crucial because resource-intensive autoscalers collectively reduce the resources available in data centers for scaling and running other workloads.

To assess the overhead of userspace autoscalers, we capture the peak CPU usage and memory usage (in kilobytes) for their process IDs. Since CATCloud is deeply integrated with the Linux Scheduler, estimating its resource requirements can be challenging. Therefore, we estimate the runtime utilized by CATCloud over periods to estimate CPU utilization and estimate memory usage using static analysis of the allocated structures.

The overheads of userspace autoscalers depend on their internal algorithms rather than the applications they monitor. However, for CATCloud, CPU and memory overheads can vary depending on the number of runqueues spawned. We estimate the overhead of the four autoscalers during the previously evaluated DeathStarBench workload.

As summarized in Table 4.1, the Kubernetes VPA's simple algorithm results in low CPU utilization of 1%. However, Kubernetes (K8s) incurs a higher memory overhead of 18 MB. This overhead can be attributed partially to the container orchestration required by VPA and the maintenance of a large history buffer. On the other hand, the HW and LSTM approaches run as separate userspace processes, leading to low memory overheads of 0.482 KB and 0.502 KB, respectively. However, these approaches require substantially higher CPU resources during peak utilization, with HW and LSTM utilizing 11% and 10.5% of CPU, respectively.

Lastly, CATCloud exhibits a CPU overhead estimated at around 50 microseconds within a period. In terms of memory overhead, the additional structure variables, including the history window (default 5 entries), result in an overhead of 0.328 KB, which is comparable to the HW and LSTM approaches

Overheads	CPU Utilization	Memory (KB)
K8s VPA	1.02%	18432
HW	11%	0.482
LSTM	10.5%	0.502
CATCloud	$\approx 0.05\%$	≈ 0.328

Table 4.1: Vertical CPU autoscaling overheads

In summary, we assess CATCloud's capacity to achieve peak performance with reduced resource utilization compared to state-of-the-art heuristic and ML-based autoscalers. Across various applications, CATCloud consistently maintains the lowest latency while utilizing up to 56.25% fewer CPU resources for Sleeping-ebizzy, at minimum 2.6%, maximum of 20% less CPU entitlement for web search, and 25.3% to 81.3% lower CPU usage for media streaming in CloudSuite. In the case of the HotelReservation benchmark under simple load conditions, CATCloud outperforms existing solutions by up to 10% in terms of performance and requires up to 8% fewer CPU resources.

CHAPTER 5: RELATED WORK

We now review related work on resource scaling for workloads in a cloud setting, with a specific focus on CPU allocation management, as well as the techniques employed to model application behavior to predict future application resource requirements.

Scaling of cloud applications can either be horizontal or vertical. Horizontal scaling allocates additional systems, and is employed when an application's resource requirements exceed what the current system(s) can provide. Vertical scaling allocates for resources within the same system(s). Unlike horizontal CPU scaling, vertical CPU autoscalers adjust for resource limits at a finer granularity, e.g., millicores or fractional vCPUs.

As briefly mentioned in in Section 2.2, vertical autoscaling techniques are broadly categorized as [8] (a) rule- and threshold-based, (b) statistical models, (c) time series / machine learning / reinforcement learning techniques, (d) queuing-theory-based, and (e) performance aware. We describe prior works that utilize one or more of these techniques to recommended CPU limits to applications. The comparison of these approaches with CATCloud is summarized in Table 5.1

Categories	**Rules**	**Statistics**	**ML/RL**	**Queuing**	**Performance**	**CATCloud**
Primary Metric	CPU Util, Throttle	CPU Util, Throttle	CPU Util, Throttle	Queue Congestion	SLOs	Runtime In Scheduler
Reactivity	✓	✓	✗	✓	✗	✓
Ability to Partially tune CPUs w/o stepping	✓	✓	✓	✗	✓	✓
Accuracy in quota tuning	✗	✗	✓	✗	✓	✓
quota-period combo	✗	✗	✗	✗	✗	✓

Table 5.1: Vertical CPU autoscaling techniques comparison

5.1 RULE- AND THRESHOLD-BASED APPROACHES

Rule- and threshold-based techniques represent one of the simplest ways to decide CPU scaling. Autoscalers may look at various surrogate metrics such as CPU utilization, throttle count, or average response time and if the observed metric is higher or lower than a set threshold, the CPU limit for an application is scaled accordingly. This approach is simple to implement and displays high degrees of reactivity. The key disadvantage of using threshold-based approaches is to identify the right threshold to set for applications. In addition, these autoscalers need an set an inertia, cooldown, or calm period, a time during which no scaling decisions can be made so that the workload can be stabilized [37]. While the approach of

rules and threshold present merits of simplicity, it is often paired with other techniques to attempt to overcome its shortcomings.

5.2 STATISTICS-BASED APPROACHES

This approach often applies statistics on the various heuristics described in Rules and Threshold based approach. The key advantage of this approach is that aims to create a more representative model of the application behavior by charting a distribution. The Kubernetes Vertical Pod Autoscaler (VPA) [7] collates CPU utilization over a period of time and uses a combination of thresholds and 90th/95th percentile to recommend millicores. Similarly, RUBAS [11] uses the the sum of the median and the deviation of CPU utilization observations to recommend limits. Statistical approaches while improve accuracy, also largely experience similar pitfalls as to rules and threshold based approaches. Distinct statistical approaches may be suited better for different workload trends and classifying workload trends can prove challenging for simple statistical workloads.

5.3 MACHINE LEARNING APPROACHES

In an attempt to accurately model application runtime behavior for future recommendations, autoscalers use neural network based models to train on the same surrogate metrics to derive a pattern from them. Autopilot [3] employs moving window predictors and recommends based on machine learning model. CPU usage prediction techniques have also been designed [13] using Holt–Winters exponential smoothing (HW) and Long Short-Term Memory (LSTM) methods. Sinan [38] uses ML to model for SLO violations, while FIRM [12] uses reinforcement learning to not only react to SLO violations but also identify key microservices. Autothrottle [14] uses the throttle heuristics to scale for CPUs, and uses reinforcement learning to ascertain targets based on SLOs. Most ML- and RL-based solutions boast improvements in accuracy of tuning for the CPU bandwidth quota. These approaches, however, suffer large overheads in terms of training and inferences, which leads to lower reactivity. Scalers such as Autothrottle require 12 hours of warm-up period to train its model, during which SLO violations are permitted.

5.4 QUEUING THEORY

The use of queuing theory is an approach that avoids the use of commonly used surrogate metrics such as utilization and throttle. The core idea of this approach is to identify congestion in a queue as and when requests are scheduled on to it. A queue is said to be congested if the number outgoing response cannot be serviced faster than the incoming requests. The advantage of this approach is the simplicity of technique in identifying bottlenecks which leads to high degrees of reactivity in dynamic workloads. Shenango [17] and Caladan [15] make use of this technique to build a CPU scheduler that achieves better performance at the microsecond scale. Solutions like these however suffer from two major downsides. First, they require a redesign of how schedulers observe and monitor tasks which can often be impractical to integrate in existing systems. Second, these solutions lack precision in terms of recommendations of CPUs as they are only able to identify congestion but not ascertain the precise ratio of CPU bandwidth needed to alleviate the congestion. These solutions therefore do not recommend CPUs in terms of partial cores which is a key requirement for the multi-tenant cloud.

5.5 PERFORMANCE-AWARE MODELS

Performance aware models focus on the performance metrics of latency, throughput and other SLO requirements of the workload as the basis of their surrogate metrics. Performance aware models may also often overlap with other established techniques such as FIRM [12], Sinan [38], and Autothrottle [14], which view SLO violations along with other metrics to learn the behavior of the application. Cilantro [18] framework is designed to minimize SLO violations by reallocating a fixed set of resources. Cilantro however, is not directly comparable as it is built to optimize utilization for a fixed cluster and not the elastic cloud.

Aside from all the merits and downsides discussed for related landscape of autoscaling research, all the approaches do not acknowledge tuning for the quota-period duo in tandem and rather either recommend entire CPUs, or millicores which only tune for quota. Autoscalers cannot recommend accurate CPU bandwidth required due to their inability to observe in the userspace. This has led to incorrect limits set for CPUs, which translates to impacts on performance, efficiency and higher costs for the cloud application. CATCloud, does not rely on these surrogates and rather implements light weight extensions to the OS to extract runtime information from the scheduler. This approach significantly reduces the need for models required to analyze and predict trends making CATCloud lightweight and highly reactive to dynamic load changes.

CHAPTER 6: CONCLUSION AND FUTURE WORK

6.1 CONCLUSION

In this thesis, we studied the various factors plaguing vertical CPU autoscaling in the cloud application landscape, including the core problem of the semantic gap between the OS and cloud interfaces used for limiting CPU resources. It was asserted that modern CPU autoscalers tuned for CPU resources using surrogate metrics of CPU utilization, queue lengths, throttle and, SLOs incorrectly, as they operated in userspace and could not ascertain true application runtime behavior. Lastly we show that recommendations from the state of the art can lead to incorrect CPU limit recommendations along with performance regressions.

We introduce CATCloud, a state-of-the-art vertical CPU autoscaler that operates directly within the OS. It minimally instruments the scheduler to extract run and yield characteristics to model the application behavior. Operating in millisecond timescales, it uses simple techniques to recommend CPU limits and offers the ability to tune both quota and period of the CPU bandwidth controller. This unique approach enables CATCloud to optimize performance by achieving high degrees of performance for the lower CPU limits, thereby reducing operating costs for cloud-based applications.

6.2 FUTURE WORK

6.2.1 Evaluation of Emerging Applications

With the provision of an in-kernel mechanism to trace application behavior, new applications as well as workloads beyond the cloud can attempt to utilize and scale efficiently. One such potential application is serverless computing. Serverless workloads typically exhibit highly bursty behavior with shorter time-spans, which most modern autoscaling solutions do not cater to. CATCloud with its light-weight tracing and high reactivity properties may cater well for workloads like these. Future work can aim to evaluate and optimize CATCloud for these applications.

6.2.2 Evaluation of Emerging Autoscaling Techniques

Modern autoscalers are beginning to recognize the drawbacks of their computation overheads, leading to low reactivity and poorer performance. To address this, solutions such as

using lightweight reinforcement learning (RL) techniques, as described in Autothrottle [14], pair threshold-based techniques with RL to achieve higher reactivity. In the future, we aim to compare CATCloud to these solutions to assess its efficacy.

6.2.3 Beyond Autotuning Recommendations

CATCloud, at its core, is a mechanism that accurately traces application runtime behavior in the scheduler. However, this knowledge can potentially be utilized to make decisions for task scheduling as well. Currently, the scheduler employs a suite of simple techniques to determine which core and when a task should be scheduled. With the analysis of a task's admittance, runtime, and yield statistics, we can attempt to make simple predictions about the task and schedule tasks to potentially achieve better performance and efficiency.

REFERENCES

[1] C. Reiss and A. Tumanov, "Heterogeneity and Dynamicity of Clouds at Scale: Google Trace Analysis."

[2] C. Lu, K. Ye, G. Xu, C.-Z. Xu, and T. Bai, "Imbalance in the cloud: An analysis on Alibaba cluster trace," in *2017 IEEE International Conference on Big Data (Big Data)*, Dec. 2017. [Online]. Available: https://ieeexplore.ieee.org/document/8258257 pp. 2884–2892.

[3] K. Rzadca, P. Findeisen, J. Swiderski, P. Zych, P. Broniek, J. Kusmierek, P. Nowak, B. Strack, P. Witusowski, S. Hand, and J. Wilkes, "Autopilot: workload autoscaling at Google," in *Proceedings of the Fifteenth European Conference on Computer Systems*, ser. EuroSys '20. New York, NY, USA: Association for Computing Machinery, Apr. 2020. [Online]. Available: https://dl.acm.org/doi/10.1145/3342195.3387524 pp. 1–16.

[4] "AWS Auto Scaling," https://docs.aws.amazon.com/autoscaling/plans/userguide/what-is-a-scaling-plan.html. [Online]. Available: https://docs.aws.amazon.com/autoscaling/plans/userguide/what-is-a-scaling-plan.html

[5] EdB-MSFT, "Autoscale in Azure Monitor - Azure Monitor," Mar. 2023, https://learn.microsoft.com/en-us/azure/azure-monitor/autoscale/autoscale-overview. [Online]. Available: https://learn.microsoft.com/en-us/azure/azure-monitor/autoscale/autoscale-overview

[6] "IBM Turbonomic." [Online]. Available: https://www.ibm.com/products/turbonomic

[7] "kubernetes vertical-pod-autoscaler," https://github.com/kubernetes/autoscaler. [Online]. Available: https://github.com/kubernetes/autoscaler

[8] T. Lorido-Botran, J. Miguel-Alonso, and J. A. Lozano, "A Review of Auto-scaling Techniques for Elastic Applications in Cloud Environments," *Journal of Grid Computing*, vol. 12, no. 4, pp. 559–592, Dec. 2014. [Online]. Available: http://link.springer.com/10.1007/s10723-014-9314-7

[9] C. Qu, R. N. Calheiros, and R. Buyya, "Auto-Scaling Web Applications in Clouds: A Taxonomy and Survey," *ACM Computing Surveys*, vol. 51, no. 4, pp. 73:1–73:33, July 2018. [Online]. Available: https://doi.org/10.1145/3148149

[10] V. Sachidananda and A. Sivaraman, "Collective Autoscaling for Cloud Microservices," Aug. 2022, arXiv:2112.14845 [cs, eess]. [Online]. Available: http://arxiv.org/abs/2112.14845

[11] G. Rattihalli, M. Govindaraju, H. Lu, and D. Tiwari, "Exploring Potential for Non-Disruptive Vertical Auto Scaling and Resource Estimation in Kubernetes," in *2019 IEEE 12th International Conference on Cloud Computing (CLOUD)*, July 2019, iSSN: 2159-6190. pp. 33–40.

[12] H. Qiu, S. S. Banerjee, S. Jha, Z. T. Kalbarczyk, and R. K. Iyer, "FIRM: An Intelligent Fine-Grained Resource Management Framework for SLO-Oriented Microservices."

[13] T. Wang, S. Ferlin, and M. Chiesa, "Predicting CPU usage for proactive autoscaling," in *Proceedings of the 1st Workshop on Machine Learning and Systems*, ser. EuroMLSys '21. New York, NY, USA: Association for Computing Machinery, Apr. 2021. [Online]. Available: https://doi.org/10.1145/3437984.3458831 pp. 31–38.

[14] Z. Wang, P. Li, C.-J. M. Liang, F. Wu, and F. Y. Yan, "Autothrottle: A Practical Bi-Level Approach to Resource Management for SLO-Targeted Microservices," Oct. 2023, arXiv:2212.12180 [cs]. [Online]. Available: http://arxiv.org/abs/2212.12180

[15] J. Fried, Z. Ruan, A. Ousterhout, and A. Belay, "Caladan: Mitigating Interference at Microsecond Timescales," 2020. [Online]. Available: https://www.usenix.org/conference/osdi20/presentation/fried pp. 281–297.

[16] A. Ali-Eldin, J. Tordsson, and E. Elmroth, "An adaptive hybrid elasticity controller for cloud infrastructures," in *2012 IEEE Network Operations and Management Symposium*, Apr. 2012, iSSN: 2374-9709. [Online]. Available: https://ieeexplore.ieee.org/document/6211900 pp. 204–212.

[17] A. Ousterhout, J. Fried, J. Behrens, A. Belay, and H. Balakrishnan, "Shenango: Achieving High {CPU} Efficiency for Latency-sensitive Datacenter Workloads," 2019. [Online]. Available: https://www.usenix.org/conference/nsdi19/presentation/ousterhout pp. 361–378.

[18] R. Bhardwaj, K. Kandasamy, A. Biswal, W. Guo, B. Hindman, J. Gonzalez, M. Jordan, and I. Stoica, "Cilantro: {Performance-Aware} Resource Allocation for General Objectives via Online Feedback," 2023. [Online]. Available: https://www.usenix.org/conference/osdi23/presentation/bhardwaj pp. 623–643.

[19] "fix #51135 make CFS quota period configurable by szuecs · Pull Request #63437 · kubernetes/kubernetes." [Online]. Available: https://github.com/kubernetes/kubernetes/pull/63437

[20] "Avoid setting CPU limits for Guaranteed pods · Issue #51135 · kubernetes/kubernetes." [Online]. Available: https://github.com/kubernetes/kubernetes/issues/51135

[21] "Requests are all you need - CPU Limits and Throttling in Kubernetes," July 2023. [Online]. Available: https://www.numeratorengineering.com/requests-are-all-you-need-cpu-limits-and-throttling-in-kubernetes/

[22] "For the Love of God, Stop Using CPU Limits on Kubernetes (Updated) | Robusta." [Online]. Available: https://home.robusta.dev/blog/stop-using-cpu-limits

[23] "QEMU." [Online]. Available: https://www.qemu.org/

[24] S. Bhat, "pratiksampat/sleeping-ebizzy," June 2014. [Online]. Available: https://github.com/pratiksampat/sleeping-ebizzy

[25] M. Ferdman, A. Adileh, O. Kocberber, S. Volos, M. Alisafaee, D. Jevdjic, C. Kaynak, A. D. Popescu, A. Ailamaki, and B. Falsafi, "Clearing the clouds: a study of emerging scale-out workloads on modern hardware," in *Proceedings of the seventeenth international conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS XVII. New York, NY, USA: Association for Computing Machinery, Mar. 2012. [Online]. Available: https://doi.org/10.1145/2150976.2150982 pp. 37–48.

[26] Y. Gan, Y. Zhang, D. Cheng, A. Shetty, P. Rathi, N. Katarki, A. Bruno, J. Hu, B. Ritchken, B. Jackson, K. Hu, M. Pancholi, Y. He, B. Clancy, C. Colen, F. Wen, C. Leung, S. Wang, L. Zaruvinsky, M. Espinosa, R. Lin, Z. Liu, J. Padilla, and C. Delimitrou, "An Open-Source Benchmark Suite for Microservices and Their Hardware-Software Implications for Cloud & Edge Systems," in *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '19. New York, NY, USA: Association for Computing Machinery, Apr. 2019. [Online]. Available: https://dl.acm.org/doi/10.1145/3297858.3304013 pp. 3–18.

[27] "CFS Bandwidth Control — The Linux Kernel documentation." [Online]. Available: https://docs.kernel.org/scheduler/sched-bwc.html

[28] "Control CPU Management Policies on the Node," section: docs. [Online]. Available: https://kubernetes.io/docs/tasks/administer-cluster/cpu-management-policies/

[29] P. Turner, B. B. Rao, and N. Rao, "CPU bandwidth control for CFS," 2010. [Online]. Available: https://www.semanticscholar.org/paper/CPU-bandwidth-control-for-CFS-Turner-Rao/c33fca3c4e4f541a5066cc4b9415a75511c05b00

[30] "Production-Grade Container Orchestration." [Online]. Available: https://kubernetes.io/

[31] "Resource Management for Pods and Containers," section: docs. [Online]. Available: https://kubernetes.io/docs/concepts/configuration/manage-resources-containers/

[32] "runtime: long GC STW pauses (80ms) · Issue #19378 · golang/go." [Online]. Available: https://github.com/golang/go/issues/19378

[33] V. Henson, "ebizzy Benchmark," Jan. 2008, https://openbenchmarking.org/test/pts/ebizzy. [Online]. Available: https://openbenchmarking.org/test/pts/ebizzy

[34] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder, "Automatically characterizing large scale program behavior," in *Proceedings of the 10th international conference on Architectural support for programming languages and operating systems*, ser. ASPLOS X. New York, NY, USA: Association for Computing Machinery, Oct. 2002. [Online]. Available: https://doi.org/10.1145/605397.605403 pp. 45–57.

[35] C. Chatfield, "The Holt-Winters Forecasting Procedure," *Journal of the Royal Statistical Society. Series C (Applied Statistics)*, vol. 27, no. 3, pp. 264–279, 1978, publisher: [Wiley, Royal Statistical Society]. [Online]. Available: https://www.jstor.org/stable/2347162

[36] S. Hochreiter and J. Schmidhuber, "Long Short-Term Memory," *Neural Computation*, vol. 9, no. 8, pp. 1735–1780, Nov. 1997, conference Name: Neural Computation. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/6795963

[37] X. Dutreilh, N. Rivierre, A. Moreau, J. Malenfant, and I. Truck, "From Data Center Resource Allocation to Control Theory and Back," July 2010, pp. 410–417.

[38] Y. Zhang, W. Hua, Z. Zhou, G. E. Suh, and C. Delimitrou, "Sinan: ML-based and QoS-aware resource management for cloud microservices," in *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. Virtual USA: ACM, Apr. 2021. [Online]. Available: https://dl.acm.org/doi/10.1145/3445814.3446693 pp. 167–181.