CPU Autoscaling With a Kernel of Truth

Pratik Rajesh Sampat
Univ. of Illinois Urbana-Champaign
Urbana, IL, USA
psampat2@illinois.edu

Tianyin Xu
Univ. of Illinois Urbana-Champaign
Urbana, IL, USA
tyxu@illinois.edu

Saugata Ghose
Univ. of Illinois Urbana-Champaign
Urbana, IL, USA
ghose@illinois.edu

Abstract

Cloud computing paradigms such as microservices and functions-as-a-service have made autoscaling an essential component of cloud application management. However, existing autoscalers struggle at capturing application dynamics, and have difficulties with precisely allocating quotas of shared system resources to the applications. We argue that one fundamental issue is the gap between native OS resource interfaces and surrogate user metrics that existing autoscalers use. In this paper, we take CPU autoscaling as an example: the cloud interface treats CPU resources as a percentage of the host CPU (e.g., millicore), while the OS kernel interprets CPU resources as time-shared quota slices allowed to run within a set period. We advocate for OS kernel support for CPU autoscaling to close the semantic gap, as it allows the autoscaler to perform precise, highly responsive resource allocation. We demonstrate the idea by developing Kscaler, a millisecond-scale CPU autoscaler for Linux. With kernel-level observability of fine-grained scheduler behavior, Kscaler outperforms state-of-the-art CPU autoscalers in responsiveness, precision, and efficiency while employing simple statistical methods.

CCS Concepts: • **Software and its engineering → Scheduling**; **Cloud computing**.

Keywords: CPU autoscaling, cloud computing, scheduling, operating systems

ACM Reference Format:
Pratik Rajesh Sampat, Tianyin Xu, and Saugata Ghose. 2025. CPU Autoscaling With a Kernel of Truth. In *16th ACM SIGOPS Asia-Pacific Workshop on Systems (APSys '25), October 12–13, 2025, Seoul, Republic of Korea.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3725783.3764407

1 Introduction

Cloud computing has experienced a constant evolution since its introduction. Today, cloud providers are practicing new forms of programming paradigms (e.g., microservices [13, 14]) and service models (e.g., serverless computing [20]). These practices require the cloud to be equipped with sophisticated *autoscalers*, which automate the dynamic scaling of cloud system resources by predicting the needs of each application and guiding application-to-resource assignment.

Autoscalers have direct impacts on the performance, efficiency, and profitability of the cloud. They are responsible for allocating enough resources to each application, based on the application's resource requests and budget, while sharing physical hardware across concurrent applications. If an autoscaler's allocation is too aggressive (overprovisioning resources), hardware resources may remain *stranded*,[1] resulting in low utilization. This, in turn, decreases the number of services assigned per cloud node and drives up operating costs. Conversely, if the allocation is too conservative, it may cause services to breach service-level agreements (SLAs).

Despite significant efforts on improving autoscalers, we find that state-of-the-art autoscalers continue to struggle at achieving the correct resource balance in the presence of diverse, dynamic cloud applications. In practice, hardware resources are significantly overprovisioned as a trade-off for stable performance [11, 15, 19, 24, 25, 28, 34].

We illustrate this struggle using *vertical CPU autoscaling*, which scales the fraction of CPU time assigned to each application instance on a cloud node. Figure 1 shows the effectiveness of state-of-the-art autoscalers upon a load spike, running the Hotel Reservation application from DeathStarBench [13] (see §3.4 for experimental setup). Figure 1 shows that the widely used Kubernetes Vertical Pod Autoscaler (K8s VPA), and recently proposed autoscalars using advanced statistical algorithms (HWES and LSTM [32]) and reinforcement learning (Autothrottle [33]), have a hard time accommodating the load spike. All four reduce application performance *and* overprovision resources during the dynamics.

While this makes for great opportunities to keep advancing autoscaling algorithms, **we argue that one fundamental problem lies not with how the autoscalers choose resource allocation, but with *what interfaces* the autoscalers make use of**. Revisiting CPU scaling, state-of-the-art vertical CPU autoscalers all operate in the user space, where they allocate a fraction of the host CPU (e.g., millicores [8], vCPUs [1]) to each application. However, the OS kernel does not have the ability to assign millicores. On Linux platforms, CPU autoscalers make use of cgroups [16],

[1] Even if a tenant uses fewer resources than it was allocated, the unused resources are typically not reallocated to other tenants, in order to maintain sufficient headroom in case the tenant uses more resources in the future [6].

APSys '25, Seoul, Republic of Korea

ACM ISBN 979-8-4007-1572-3/25/10
https://doi.org/10.1145/3725783.3764407

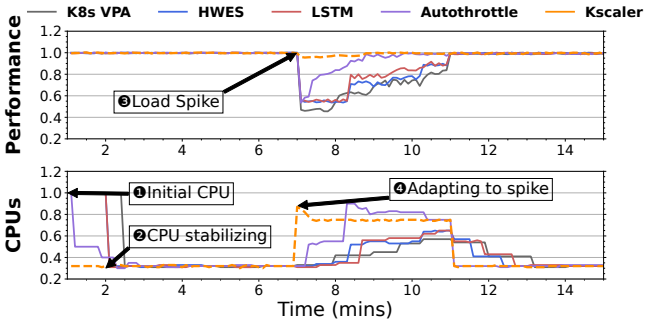


Figure 1. Tenant performance (vs. ideal) and CPU share by userspace autoscalers (K8s VPA, HWES, LSTM, Autothrottle) and Kscaler, our in-kernel autoscaler.

which manage the CPU using a *bandwidth* interface that splits up a given time *period* into time-shared *quota* slices. A CPU autoscaler sets a fixed period, and then translates millicore assignments into per-application quotas.

The inherent issue is that applications exhibit various ranges of periodic behavior, yet current CPU autoscalers lack visibility of it from user space. Moreover, an application's periodicity is often dynamic, influenced by ever-changing workload patterns. If an autoscaler could capture the periodicity effectively (e.g., identifying a service's request duration and frequency), it could assign a precise amount of millicores for each of the application's periods. Instead, existing CPU autoscalers use the same fixed time-sharing period across all applications. This period typically does not align with the application's periodicity, causing the application's demands to erroneously *appear* as if they are fluctuating (Figure 3).

We assert that autoscaling in the user space is prone to inherent semantic mismatches. Hence, userspace autoscalers rely on surrogate metrics and complex modeling techniques, which have slow reactivity and inaccurate predictions. We advocate that **autoscaling should be done in the OS kernel**. In fact, modern OS kernels already track fine-grained information on applications that are currently executing. We show that this information can enable us to implement efficient autoscalers that eliminate semantic mismatches.

We develop Kscaler, a millisecond-scale vertical CPU autoscaler that resides in the Linux kernel. By making use of fine-grained information maintained by Linux's CPU scheduler, Kscaler rapidly identifies the periodicity *and* the optimal CPU quota of each application. Kscaler's algorithm correctly provisions millisecond-scale CPU allocations that are much more responsive to application dynamics than those of userspace CPU autoscalers (Figure 1), despite being a significantly simpler algorithm. As a result, Kscaler outperforms state-of-the-art autoscalars substantially. We believe this shows new opportunities of in-kernel autoscaling and the potential for simpler yet smarter resource allocation.

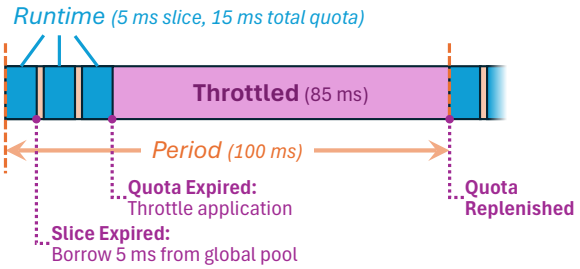


Figure 2. Task group CPU behavior with a period of 100 ms, a quota of 15 ms, and a runtime slice of 5 ms.

2 CPU Autoscaling

Vertical CPU autoscaling dynamically allocates a machine's CPU resources to cloud applications based on observed metrics (e.g., CPU utilization). Today, cloud providers expose CPU resources using logical units such as *millicores* [8], i.e., $\frac{1}{1000}$ of a physical CPU's worth of runtime. However, the millicore is not an OS primitive, so autoscalers must translate the millicore value to the CPU abstraction employed by OS schedulers.

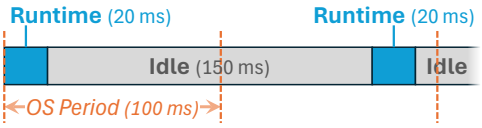
Modern OSes such as Linux manage the CPU resources of a task group using *CPU bandwidth*. CPU bandwidth [21] is a function of *quota* and *period*: the quota limits the maximum amount of CPU time a task group can run, and the period specifies the duration of quota enforcement. The CPU time that an application can run for is determined by setting both the quota and period. If the application exhausts its quota, the OS scheduler throttles it until the start of the next period (which resets the quota).

Building on the time-sharing scheduling foundations introduced by Multics [10], Linux schedulers (e.g., CFS [22], EEVDF [30]) use a data structure called the *run queue* to manage processes that are ready to execute on the CPU. Each CPU core in a multicore system has its own run queue. The following parameters [31] are employed to enforce the multicore CPU bandwidth limit:

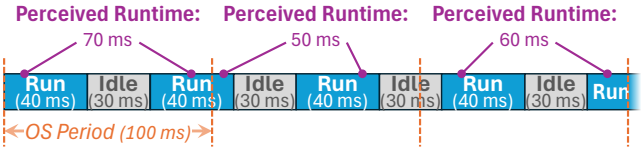
- *global quota:* quota that is tracked per task group;
- *local quota:* local pool of quota for each run queue;
- *runtime slice:* batch size of runtime, i.e., the size at which the local quota borrows runtime from the global quota.

Figure 2 depicts task group scheduling. When a task group has a quota–period configuration, the quota is assigned as a global quota. Each run queue possesses a local quota. The local quota borrows from the global quota one runtime slice at a time to run processes. When the local quota is exhausted, more runtime is requested from the global quota. If the global quota is depleted, the run queue is throttled for this period.

CPU autoscalers translate millicores to a quota–period configuration. Existing autoscalers [4, 7, 27, 29, 32, 33] all use a static period (typically the default 100 ms) and convert



(a) Period being too short (a quota of 20 ms in a period of 100 ms is translated into 200 millicores); the application's periodicity is 170 ms, i.e., only 117 millicores are needed.



(b) Period being too long (the application's periodicity is 70 ms), forcing autoscalers to observe erratic runtime and thus alternate between overprovisioning and underprovisioning.

Figure 3. Imprecise CPU autoscaling due to not understanding an application's periodicity.

millicores into a ratio of quota divided by period (e.g., a 500-millicore allocation translates to 50 ms of quota for a 100 ms period). However, this leads to imprecise quota assignments from period to period, *because real-world applications exhibit inherent periodicity that rarely matches the fixed period.*

A cloud application's periodicity is not only *diverse* (depending on the application's characteristics) but also *dynamic* (depending on application usage), making it difficult to tune manually. Figure 3 shows two examples of periodicity mismatches. When a fixed period does not include all of an application period's idle time, the autoscaler overprovisions millicores (e.g., Figure 3a). When a fixed period does not include all of an application period's runtime, the autoscaler underprovisions millicores and unnecessarily throttles performance. If back-to-back periods alternate between these two cases (e.g., Figure 3b), the autoscaler incorrectly presumes that a stable application is fluctuating erratically.

Essentially, the lack of observability into an application's periodicity, and the resulting use of a fixed, static period, is due to inherent limitations of existing autoscalers that rely on userspace surrogate metrics. We posit that no advanced algorithms or learning techniques can account for this gap. However, OS kernels already possess rich information on application behavior. For example, the task scheduler knows an application's split between runtime and idle time at a scheduler slice granularity (as an idle process is not on the active queue, and hence is not a candidate for scheduling at that time). By positioning the autoscaler in the kernel, it can use such information to dynamically identify each application's periodicity.

3 The Kscaler Approach

To demonstrate the power of autoscaling in the kernel, we prototype an in-kernel CPU autoscaler named Kscaler and show its ability to (1) avoid semantic mismatches between the cloud interface and kernel abstraction by directly accessing the OS resource allocation interface, (2) precisely model application behavior such as periodicity by observing fine-grained scheduler behavior, and (3) react rapidly to workload changes.

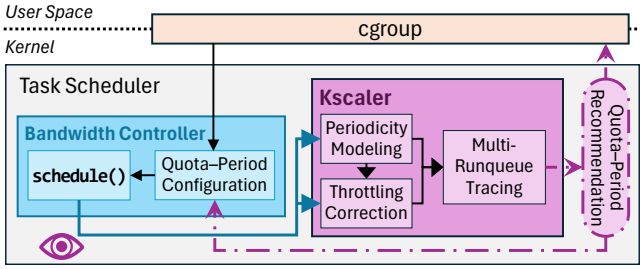


Figure 4. Overview of Kscaler and its interactions with existing Linux components (task scheduler and cgroup).

3.1 Kscaler Overview

We implement Kscaler in the task scheduler of the Linux kernel, which provides millisecond-scale observability of fine-grained application behavior (e.g., runtime and idle time). Kscaler interfaces with the Linux CPU cgroup controller and continuously adjusts both the period and the CPU quota for that period for a cgroup based on the observed application behavior. Kscaler does not rely on any userspace surrogate metrics and eliminates their associated drawbacks.

Figure 4 gives an overview of Kscaler and its interactions with existing Linux components. Kscaler models an application's behavior by observing its interactions within the task scheduler. The core is a lightweight technique that captures application run–yield patterns, which can be collated to model an application's periodicity and the quota within estimated periods. Kscaler makes quota–period recommendations using a simple statistical method, which is applied to the cgroup interface, if operating in the autoscaling mode.

3.2 Observability

Kscaler observes an application's behavior from the bandwidth scheduler loop. While Kscaler can observe fine-grained scheduler behavior such as every voluntary and involuntary yield, recording and maintaining this information for each context switch can be expensive, especially with fragmented time slices. Kscaler consists of three main components: (1) capturing application runtime and idle time across periods, without knowing periodicity a priori; (2) identifying throttling effects and applying corrections to observed behavior; and (3) collating observations in a multicore environment.

3.2.1 Modeling Periodicity. For a fixed period, it is possible to observe and record the cumulative runtime of an

application. For each period P, the total runtime R is the sum of individual slices s_i for a specific group of run queues within a period, expressed as $R = \sum s_i$. The idle time is then determined by subtracting R from P. The per-period runtime will be recorded in a history buffer. Compared to userspace metrics (e.g., CPU utilization), the kernel approach is more efficient and eliminates polling.

However, as discussed in §2, periodicity can be dynamic and hard to know a priori. We develop an algorithm to capture the dynamically changing periodicity of a task. The core idea is to account for the runtime per run queue until the task *voluntarily* yields. We introduce two high-resolution clocks in Kscaler: C_r for tracking runtime and C_l for tracking idle time. Both clocks are initialized to 0 at the start of the first runtime slice for the task. Kscaler also tracks the last time the task was scheduled, denoted as T_s', which is also initialized to 0.

Whenever Kscaler() observes that the task has been selected for scheduling, it performs two steps. First, Kscaler calculates the time elapsed since the last time the task was scheduled (i.e., $C_r - T_s'$). If the elapsed time is approximately equal to the local CPU bandwidth slice r, it indicates that the yield is involuntarily due to the expiration of the scheduling slice. Likewise, if the elapsed time since the last yield is less than r, this indicates task preemption, which is also an involuntary yield. In both of these cases, Kscaler resets C_l to 0, indicating that the task has not yet voluntarily yielded (and, thus, the task did not previously stop due to its own periodicity). Otherwise, if $C_r - T_s'$ exceeds the local CPU bandwidth slice r, it indicates that the task had previously voluntarily yielded, and has thus entered the idle time portion of its period. In this case, the task was scheduled because it is starting a new period, and Kscaler can calculate the last period's idle time T_l as $C_l - r$, and the last period's cumulative runtime T_r as $C_r - T_l$. Second, if the task has not voluntarily yielded, T_s' is set to the current time for use during the next time the task is scheduled.

Figure 5 illustrates this algorithm using an example. We present a CPU bandwidth cycle with a quota of 25 ms, a period of 40 ms, a local bandwidth slice r of 5 ms, and a measured yield overhead y of 0.1 ms. At the start (i.e., when the first local runtime slice is scheduled), C_r, C_l, and T_s' are all initialized to 0. When the second runtime slice is scheduled, C_r is $0 + r + y = 5.1$ ms; C_l is reset because $C_r - T_s' \approx r$. This cycle continues until $C_r - T_s' = 20$, which is significantly greater than $r = 5$. The idle time T_l is calculated as $T_l = C_l - r_{i-1} = 20 - 5 = 15$ ms. The cumulative runtime T_r is calculated as $C_r - \sum y - T_l = 40.4 - 0.4 - 15 = 25$ ms.

3.2.2 Accounting for Throttling. When a task requires more runtime than the defined quota within a period, it will be involuntarily yielded due to throttling. The task is evicted from the run queue for the remainder of the period and is re-queued only in the next period when the quota replenishes. Such throttling behavior is not considered in the above application behavior modeling (e.g., Figure 5).

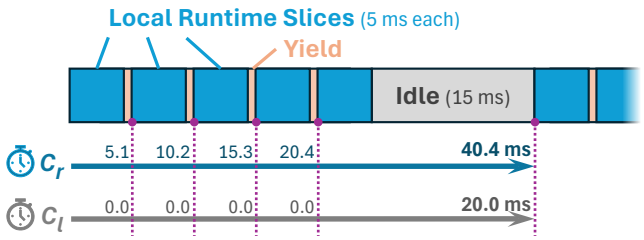


Figure 5. Illustration of algorithm to infer the period and quota of a running application.

The Linux scheduler maintains throttle information on a per-cgroup basis. We extend this throttle-accounting mechanism to track the throttle duration for each run queue within that cgroup. During throttling, Kscaler initializes a throttle clock and measures the time until the task is reassigned to run. Since the runtime and idle time clocks continue ticking during this period, we use the throttle time as a correction to compensate for this additional time gained.

In case of overloads, high degrees of throttling can severely affect application behavior, with few runtime slices available. Kscaler defines high degrees of throttling to be when the majority of observed periods ($\geq$ 50%) in the history were throttled. In such cases, the quota is scaled to overprovision CPU resources, allowing the application to stabilize with an observable run–yield duration. After stabilization, the application can then scale down to its required limits.

3.2.3 Multicore Support. Tasks can potentially spawn multiple threads simultaneously on multiple cores. To model multicore applications, we must collate data collected from all active run queues. The challenge is that multiple run queues of an application can be in an inactive state only to be brought back to running (e.g., upon interrupts). Such behavior leads to inaccurate accounting and misrepresents the actual application behavior, resulting in under-recommending CPU resources. However, synchronizing information across run queues at each slice is not only prohibitively expensive but also futile.

To address the problem, run queues added to the active list are treated as immutable for the duration of the current period. Kscaler queries the active list of run queues to extract runtime information only (1) when a new run queue is added or (2) at the end of a period. The runtime durations are summed, and the P99 idle time is recorded to form the cumulative observation as a single entry in the history. While this may result in brief periods of overallocation when run queues that do not get scheduled back have their utilization recorded, the overallocation is minimal and lasts only for a single period, after which the lists are refreshed.

Table 1. Autoscalers used in preliminary evaluation.

Autoscaler	Policy	Key Metric	Warm-Up	Decision
K8s VPA [7, 9]	Histogram	CPU util.	20 min	60 sec
HMES [32]	Time series	CPU util.	3 min	1 sec
LSTM [32]	RNN	CPU util.	3 min	1 sec
Autothrottle [33]	RL	Throttle	6 hour	60 sec
Kscaler *(our work)*	Percentile	Run queues	0	10 ms

3.3 Autoscaling Policy

For in-kernel tuning, once the history buffer is full, Kscaler determines the worst-case CPU resource utilization. Kscaler uses a simple statistical method to tune quota and period values based on observability data—the P99 runtime is recommended as the quota, and the sum of the P99 runtime and idle time constitutes the recommended period for next CPU limit. This approach is simple, fast, and lightweight, and thus can reside in the core scheduler.

Kscaler also exposes observability data to the user space via eBPF and the debugfs trace buffer. This can allow a userspace version of Kscaler to employ more sophisticated algorithms (e.g., hierarchical policies), but with a significantly reduced reactivity.

3.4 Preliminary Evaluation

Reference Autoscalers. We compare Kscaler with four reference autoscalers (Table 1). The Kubernetes CPU Vertical Pod Autoscaler (K8s VPA) represents the state of the practice. We also implement two advanced autoscalers [32], which use Holt–Winters exponential smoothing (HWES) [5], and long short-term memories (LSTM) [17] for prediction on top of K8s VPA. Autothrottle [33] is a state-of-the-art autoscaler that uses throttle count (the number of times a service exhausts its CPU quota in a period) as the surrogate metric, and employs reinforcement learning for prediction.

Experimental Setup. We use four performance benchmarks: Sleeping Ebizzy [2], which simulates web page traffic; the Web Search and Media Streaming applications from CloudSuite [12]; and the Hotel Reservation application from DeathStarBench [13]. All benchmarks are deployed in containers on Kubernetes, with unlimited memory and CPU autoscaling. K8s VPA, HWES, LSTM, and Autothrottle are all deployed as userspace autoscalers, while Kscaler is deployed in the host Linux kernel (ver. 6.3.0) and is activated for the cgroups of the target containers.

All autoscalers are given sufficient time for learning (model training) for each benchmark, which is not counted in the results. We find that K8s VPA requires 20 minutes, HWES and LSTM require approximately 10 minutes, and Autothrottle requires 6 hours for model training. *In contrast, Kscaler requires no training or warm-up time.*

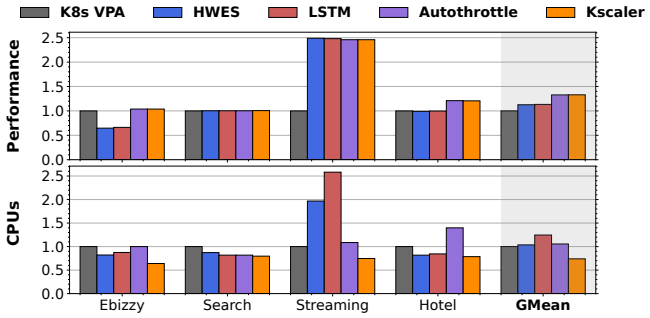


Figure 6. Application performance and CPU cost.

Initial Results. Figure 6 shows the normalized application performance and normalized multicore CPU resource allocation (cost) of evaluated autoscalers. Kscaler outperforms all the references in all the benchmarks. On average, Kscaler saves 29.94% of the CPU resources while preserving the same performance of the best-performing reference autoscaler.

Deeper analysis shows that Sleeping Ebizzy and Media Streaming exhibit highly dynamic periodicity, which does not conform to the 100 ms default period value. When surrogate metrics observe saturation, the reference autoscalers request more CPUs to maintain performance. Kscaler accurately identifies and adapts to this periodicity change without increasing the CPU allocation. Web Search does not present significant workload variations; hence, all autoscalers perform similarly. Lastly, Hotel Reservation exhibits both dynamic changes in periodicity (albeit lower than Media Streaming) and load spikes that require CPU resources to be scaled up. HWES and LSTM were unable to adapt to the load change quickly—while they request marginally more CPU resources than Kscaler, the application performance degrades. Autothrottle is able to adapt to load changes due to its reliance on throttle counting. However, it lacks precision and often overallocates CPU resources (by as much as 44% compared to Kscaler). Figure 1 shows this pattern at one of the load spike points of the Hotel Reservation benchmark.

4 Discussion and Open Problems

Kscaler shows promising initial results for kernel-based autoscaling approaches. This warrants continued development of Kscaler, and opens up several research questions.

Passing Application SLOs Into the Kernel. Kscaler currently aims to minimize throttling and maximize application performance. However, cloud applications have varying service-level objectives (SLOs). Prior work [4, 27, 33] has demonstrated the benefits of incorporating SLO information in resource management to ensure that allocated resources meet application needs. As part of future work, we plan to enhance Kscaler by using SLO information to enable objective-specific CPU resource scaling.

Challenges of High Reactivity for Cloud Providers. Cloud users are charged based on CPU resources they request throughout the runtime duration; when autoscaling modifies these requests, charges are adjusted accordingly. Due to Kscaler's highly reactive nature to load spikes and application jitters, CPU usage fluctuates constantly. This may result in high telemetry costs for cloud providers, as they must continuously gather and process CPU usage data. Moreover, orchestrators will need faster, more intelligent provisioning and load balancing that cater to this new model.

Challenges With Multiple Multicore Tenants. Kscaler estimates resource demands of multicore workloads by maintaining an active task list within each scheduling period. While localized run queue designs offer reduced overhead, they can encounter several challenges under high contention such as accurate CPU accounting due to frequent task migrations. Some of these issues can be mitigated at the kernel level, e.g., by migrating scheduling statistics alongside tasks as they move between run queues. A thorough contention analysis is essential to evaluate the interactions between multiple co-located workloads and the broader cloud stack.

Implications on Scheduling and Load Balancing. OS schedulers consider various metrics, such as runtime, load, priority, cache locality, NUMA affinity, and CPU idleness. However, during task placement and load balancing, they currently overlook periodicity as a metric for cgroups governed by quota and period constraints. With the emergence of custom scheduling frameworks such as sched_ext [23] and ghOSt [18], periodicity information has the potential to open up a new class of specialized OS schedulers optimized for bandwidth-controlled applications.

Multi-Resource Autoscaling. Kscaler shows that kernel observability is instrumental for developing insights of application behavior and resource requirements. We believe that extending kernel observability can unlock new opportunities for building more effective multi-resource autoscalers. For example, the memory cgroup subsystem already exposes useful information (e.g., pressure metrics and reclaim policies) that span diverse types of memory [16]. Autoscaling can benefit from kernel interfaces [3, 26] that help to characterize fine-grained memory access patterns. These insights can help optimize allocation and eviction to make efficient use of memory. A similar opportunity exists for I/O resources [16], where existing kernel mechanisms expose interfaces for utilization, latency, and prioritization, which can serve as the basis for developing cost models for each application.

Acknowledgments

We thank Gautham R. Shenoy, Haoran Qiu, and Francis Y. Yan for their feedback and insightful discussions. Thanks to Alan Andrade and Linkai Song for their help with experiments. The work is supported in part by NSF CNS-2145295, CNS-2130560, and CNS-1956007.

References

[1] Amazon Web Services, Inc. 2025. Auto Scaling Documentation. https://docs.aws.amazon.com/autoscaling/.
[2] Valerie Aurora and Srivatsa Bhat. 2020. Sleeping-Ebizzy Benchmark — GitHub Repository. https://github.com/pratiksampat/sleeping-ebizzy.
[3] Bharata B Rao. 2025. Kernel Daemon for Detecting and Promoting Hot Pages. https://lore.kernel.org/lkml/20250306054532.221138-1-bharata@amd.com/.
[4] Romil Bhardwaj, Kirthevasan Kandasamy, Asim Biswal, Wenshuo Guo, Benjamin Hindman, Joseph Gonzalez, Michael Jordan, and Ion Stoica. 2023. Cilantro: Performance-Aware Resource Allocation for General Objectives via Online Feedback. In *OSDI*.
[5] Christopher Chatfield. 1978. The Holt–Winters Forecasting Procedure. *J. R. Stat. Soc. Ser. C (Appl. Stat.)* (1978).
[6] Cloud Native Computing Foundation. 2024. "Kubernetes Scheduler". https://kubernetes.io/docs/concepts/scheduling-eviction/kube-scheduler/
[7] Cloud Native Computing Foundation. 2024. Kubernetes Vertical Pod Autoscaling. https://kubernetes.io/docs/concepts/workloads/autoscaling/.
[8] Cloud Native Computing Foundation. 2024. Resource Management for Pods and Containers. https://kubernetes.io/docs/concepts/configuration/manage-resources-containers/.
[9] Cloud Native Computing Foundation. 2025. Kubernetes Autoscaler — GitHub Repository. https://github.com/kubernetes/autoscaler.
[10] Fernando J. Corbató and Victor A. Vyssotsky. 1965. Introduction and Overview of the Multics System. In *FJCC*.
[11] Eli Cortez, Anand Bonde, Alexandre Muzio, Mark Russinovich, Marcus Fontoura, and Ricardo Bianchini. 2017. Resource Central: Understanding and Predicting Workloads for Improved Resource Management in Large Cloud Platforms. In *SOSP*.
[12] Michael Ferdman, Almutaz Adileh, Onur Kocberber, Stavros Volos, Mohammad Alisafaee, Djordje Jevdjic, Cansu Kaynak, Adrian Daniel Popescu, Anastasia Ailamaki, and Babak Falsafi. 2012. Clearing the Clouds: A Study of Emerging Scale-Out Workloads on Modern Hardware. In *ASPLOS*.
[13] Yu Gan, Yanqi Zhang, Dailun Cheng, Ankitha Shetty, Priyal Rathi, Nayan Katarki, Ariana Bruno, Justin Hu, Brian Ritchken, Brendon Jackson, Kelvin Hu, Meghna Pancholi, Yuan He, Brett Clancy, Chris Colen, Fukang Wen, Catherine Leung, Siyuan Wang, Leon Zaruvinsky, Mateo Espinosa, Rick Lin, Zhongling Liu, Jake Padilla, and Christina Delimitrou. 2019. An Open-Source Benchmark Suite for Microservices and Their Hardware–Software Implications for Cloud & Edge Systems. In *ASPLOS*.
[14] Sanjay Ghemawat, Robert Grandl, Srdjan Petrovic, Michael Whittaker, Parveen Patel, Ivan Posva, and Amin Vahdat. 2023. Towards Modern Development of Cloud Applications. In *HotOS*.
[15] Jing Guo, Zihao Chang, Sa Wang, Haiyang Ding, Yihui Feng, Liang Mao, and Yungang Bao. 2019. Who Limits the Resource Efficiency of My Datacenter: An Analysis of Alibaba Datacenter Traces. In *IWQoS*.
[16] Tejun Heo. 2015. "Control Group v2". https://docs.kernel.org/admin-guide/cgroup-v2.html#cpu
[17] Sepp Hochreiter and Jürgen Schmidhuber. 1997. Long Short-Term Memory. *Neural Comput.* (1997).
[18] Jack Tigar Humphries, Neel Natu, Ashwin Chaugule, Ofir Weisse, Barret Rhoden, Josh Don, Luigi Rizzo, Oleg Rombakh, Paul Turner, and Christos Kozyrakis. 2021. ghOSt: Fast & Flexible User-Space Delegation of Linux Scheduling. In *SOSP*.
[19] Seyyed Ahmad Javadi, Amoghavarsha Suresh, Muhammad Wajahat, and Anshul Gandhi. 2019. Scavenger: A Black-Box Batch Workload Resource Manager for Improving Utilization in Cloud Environments. In *SoCC*.

[20] Eric Jonas, Johann Schleier-Smith, Vikram Sreekanti, Chia-Che Tsai, Anurag Khandelwal, Qifan Pu, Vaishaal Shankar, Joao Menezes Carreira, Karl Krauth, Neeraja Yadwadkar, Joseph Gonzalez, Raluca Ada Popa, Ion Stoica, and David A. Patterson. 2019. *Cloud Programming Simplified: A Berkeley View on Serverless Computing.* Technical Report UCB/EECS-2019-3. Univ. of California at Berkeley Dept. of Electrical Engineering and Computer Sciences.

[21] Linux Kernel Organization, Inc. 2024. CFS Bandwidth Control. https://docs.kernel.org/scheduler/sched-bwc.html

[22] Linux Kernel Organization, Inc. 2025. CFS Scheduler Design. https://docs.kernel.org/scheduler/sched-design-CFS.html.

[23] Linux Kernel Organization, Inc. 2025. Extensible Scheduler Class. https://www.kernel.org/doc/html/next/scheduler/sched-ext.html.

[24] Qixiao Liu and Zhibin Yu. 2020. The Elasticity and Plasticity in Semi-Containerized Co-Locating Cloud Workload: A View From Alibaba Trace. In *SoCC*.

[25] Chengzhi Lu, Kejiang Ye, Guoyao Xu, Cheng-Zhong Xu, and Tongxin Bai. 2017. Imbalance in the Cloud: An Analysis on Alibaba Cluster Trace. In *BigData*.

[26] SeongJae Park, Madhuparna Bhowmik, and Alexandru Uta. 2022. DAOS: Data Access-Aware Operating System. In *HPDC*.

[27] Haoran Qiu, Subho S. Banerjee, Saurabh Jha, Zbigniew T. Kalbarczyk, and Ravishankar K. Iyer. 2020. FIRM: An Intelligent Fine-Grained Resource Management Framework for SLO-Oriented Microservices. In *OSDI*.

[28] Charles Reiss, Alexey Tumanov, Gregory R. Ganger, Randy H. Katz, and Michael A. Kozuch. 2012. Heterogeneity and Dynamicity of Clouds at Scale: Google Trace Analysis. In *SoCC*.

[29] Krzysztof Rzadca, Pawel Findeisen, Jacek Swiderski, Przemyslaw Zych, Przemyslaw Broniek, Jarek Kusmierek, Pawel Nowak, Beata Strack, Piotr Witusowski, Steven Hand, and John Wilkes. 2020. Autopilot: Workload Autoscaling at Google. In *EuroSys*.

[30] Ion Stoica and Hussein Abdel-Wahab. 1996. *Earliest Eligible Virtual Deadline First: A Flexible and Accurate Mechanism for Proportional Share Resource Allocation.* Technical Report TR-95-22. Old Dominion Univ. Dept. of Computer Science.

[31] Paul Turner, Bharata B. Rao, and Nikhil Rao. 2010. CPU Bandwidth Control for CFS. In *Linux Symposium*.

[32] Thomas Wang, Simone Ferlin, and Marco Chiesa. 2021. Predicting CPU Usage for Proactive Autoscaling. In *EuroMLSys*.

[33] Zibo Wang, Pinghe Li, Chieh-Jan Mike Liang, Feng Wu, and Francis Y. Yan. 2025. Autothrottle: A Practical Bi-Level Approach to Resource Management for SLO-Targeted Microservices. In *NSDI*.

[34] Tianyi Yu, Qingyuan Liu, Dong Du, Yubin Xia, Binyu Zang, Ziqian Lu, Pingchao Yang, Chenggang Qin, and Haibo Chen. 2020. Characterizing Serverless Platforms With ServerlessBench. In *SoCC*.