μslice: Toward Accurate Sub-Core Allocation with eBPF

Changyuan Lin
University of British Columbia

Mohammad Shahrad
University of British Columbia

Abstract

The rise of cloud service models such as serverless computing and microservices has driven the adoption of fine-grained resource provisioning and billing, allowing users to request fractional CPU allocations at sub-core granularity. However, existing operating system mechanisms for CPU bandwidth control were designed for much coarser resource allocations. As a result, they can introduce substantial resource overallocation and performance variability when enforcing the small CPU shares required by modern cloud workloads.

In this paper, we investigate whether eBPF can bridge this gap by enabling more precise and responsive CPU bandwidth enforcement. We design and implement an eBPF-based scheduling framework, μslice, that complements existing kernel mechanisms to improve the accuracy of fine-grained CPU allocation while maintaining low overhead. Our evaluation demonstrates that the proposed approach significantly reduces resource overallocation and performance variability compared to conventional OS resource control mechanisms.

CCS Concepts

• **Software and its engineering → Scheduling**; • **Computer systems organization → Cloud computing**.

Keywords

OS Scheduling, eBPF, CPU Bandwidth Control, Cloud Computing, Serverless Computing

1 Introduction

Modern cloud platforms offer diverse deployment models, flexible control knobs for resource allocation, and fine-grained billing. As a result, running short workloads with low resource consumption on multi-tenant infrastructure has become increasingly popular. Many production workloads, such as serverless functions and microservice stages, are often configured with small fractional vCPU allocations and consist of short CPU bursts that run for only a few milliseconds per invocation or request. This trend results in a higher degree of co-tenancy on servers compared to traditional virtual machine (VM) hosting environments [9, 48].

In such environments, a primary role of the OS kernel is to ensure resource isolation and fair allocation across co-tenant workloads. Commodity OS schedulers (e.g., CFS and EEVDF), together with CPU bandwidth control mechanisms (e.g., Cgroup), can provide effective long-term fairness in resource sharing. However, short workloads expose a fundamental mismatch between scheduling timescales and fair resource allocation. These requests may require only a few milliseconds of CPU time, which is on the same order of magnitude as, or even smaller than, scheduler tick intervals, bandwidth control periods, and runtime quotas.

At this timescale, OS scheduling and enforcement granularity become visible at the request level, which may lead to overrun and overallocation [37]. For example, when a request's CPU demand is smaller than the runtime quota available in a cgroup period, the request can consume all the CPU time it needs and complete before bandwidth enforcement can pace its execution. While a workload may remain compliant with its period-level cgroup quota, it can still receive CPU service faster than the rate implied by its configured fraction, leading to resource overallocation. Such overallocation can weaken the resource assumptions used by scheduling [18, 53], admission control [3, 33], autoscaling [3, 32], isolation [36, 38, 39], and billing [4, 37, 47] in multi-tenant systems, where many of these mechanisms use the configured allocation as a proxy for the CPU service a tenant can receive and the load it can impose on shared resources.

Prior work has characterized serverless CPU overallocation and has shown that short functions on public cloud platforms can complete faster than the duration implied by their configured CPU [37]. Most of the existing scheduling frameworks for short workloads focus on latency reduction or work conservation [13, 14, 27, 41–43]. None of them targets enforcing accurate sub-core CPU allocation and avoiding overallocation for millisecond-scale workloads with independently configured sub-core allocations in multi-tenant environments.

eBPF '26, Prague, Czech Republic

ACM ISBN 979-8-4007-2911-9/26/09
https://doi.org/10.1145/3837779.3838168

To fill this gap, we propose μslice[1], an eBPF-based scheduling framework that complements existing kernel mechanisms to improve the accuracy of sub-core CPU allocation for millisecond-scale workloads. μslice runs as a sched_ext scheduler and applies CPU pacing to managed cgroups. It tracks delivered CPU and releases a runnable task only when the next CPU grant keeps the cgroup within the service allowance implied by its configured sub-core allocation. We evaluate μslice on isolated and heterogeneous workload sets and compare it against commodity Linux CPU control baselines. Our results show that μslice keeps short CPU-bound requests close to their configured allocation, mitigates CPU overallocation, narrows request-level service variation across tenants, and adds small measured overhead.

2 Background and Motivation

Cloud workloads increasingly exhibit millisecond-scale CPU bursts under fractional allocations. For such workloads, period-based CPU bandwidth control can overallocate CPU when a burst fits within the quota. We describe these characteristics (§2.1 and §2.2), explain the overallocation effect (§2.3), and motivate accurate sub-core allocation enforcement (§2.4).

2.1 Millisecond-Scale Cloud Workloads

Millisecond-scale workloads running under fractional vCPU allocations in multi-tenant settings have become common in the cloud. This pattern appears across diverse cloud deployment models and applications, such as serverless functions, request handlers, microservice stages, and edge computing workloads, where execution is increasingly organized as short, request-scoped units of work rather than long-running instances provisioned with dedicated cores.

While these workloads are not uniformly small, a large and growing fraction consists of millisecond-scale CPU bursts. Recent serverless traces from Azure [56], Huawei [25, 26], and IBM [45] show that 40–70% of requests have durations below 10 ms. Microservices can be short as well; about 37% of remote procedure calls to microservices in the first-day slice of Alibaba traces [40] are below 10 ms. Edge workloads can be even shorter: Cloudflare Workers consume only 2.2 ms of CPU time per request on average, while heavier tasks typically use 10–20 ms [8]. The trend is likely to persist, driven by improved workflow orchestration, continued adoption of event-driven architectures, and increasingly lightweight isolation mechanisms (e.g., WebAssembly) [3, 49, 55].

2.2 Fractional vCPUs in the Cloud

Small fractional vCPU allocations have also become common in production. Public cloud platforms increasingly expose CPU capacity at sub-core granularity, enabling fractional vCPU shares as a fine-grained, user-visible resource control knob. For example, Google Cloud Run supports allocations as low as 0.08 vCPUs in increments of 0.001 vCPUs [17], while the default AWS Lambda configuration is 0.072 vCPUs (with the default memory setting of 128 MB) [1]. In the IBM Code Engine traces, 44.8% of applications are configured with less than one vCPU, and 34.2% are configured with 0.25 vCPUs or below [45]. Huawei serverless traces show a similar pattern: over 40% of pods are configured with only 0.3 vCPUs [26].

2.3 CPU Bandwidth Control and Resource Overallocation

Commodity Linux kernels leverage the Completely Fair Scheduler (CFS) or, since kernel 6.6, the Earliest Eligible Virtual Deadline First (EEVDF) scheduler to select a runnable task, while cgroups provide CPU bandwidth control [10, 29, 52]. The kernel maintains a data structure (cfs_bandwidth) for bandwidth control through cpu.max, which specifies an enforcement period and a runtime quota. While CFS and EEVDF use different fair-scheduling policies, the bandwidth control mechanism and interface remain the same. We use P and Q to denote the bandwidth control period and quota, respectively, for the remainder of the paper. During each period, runnable tasks in a cgroup may collectively consume at most Q units of CPU time. Once the quota is exhausted, tasks are throttled until the budget is replenished in the next period. A fractional allocation $f \in (0, 1)$ is represented as $Q = fP$.

Consider a single-threaded, CPU-bound workload requiring CPU time C. Ideally, the workload should receive CPU service at rate f and complete in approximately $d^* = C/f$ wall-clock time. In other words, the resource contract naturally implied by the fractional CPU allocation should follow reciprocal scaling: half the core allocation, double the execution time. However, cgroup enforces the service rate contract only at the period level, not at the timescale of short CPU bursts. This makes resource allocation quantized. Each cgroup starts with a full quota available. We can calculate the actual wall-clock execution duration as:

$$d = \begin{cases} \lfloor C/Q \rfloor P + C \bmod Q, & \text{if } C \bmod Q \neq 0, \\ (\lfloor C/Q \rfloor - 1) P + Q, & \text{otherwise.} \end{cases} \quad (1)$$

For long-running CPU-bound workloads, execution alternates between running and throttling, and the average CPU service rate approaches $Q/P = f$. Millisecond-scale workloads with fractional vCPUs interact differently. If a workload requires CPU time $C \leq Q$ and sufficient quota is available, it can consume its entire demand at full-core speed within one period and complete in roughly C time, even though the resource contract is $d^* = C/f$. While the workload remains

[1]The name μslice refers to scheduling at fine granularity, rather than microsecond-scale scheduling.

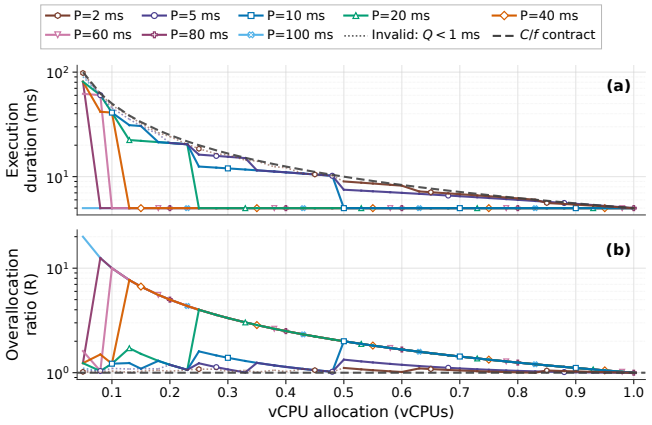

Figure 1: Period-level bandwidth control leads to resource overallocation for short CPU bursts. Longer periods let workloads complete much faster than the C/f resource contract, especially for small fractional vCPUs.

compliant with the period-level limit, it receives CPU service faster than its configured fractional rate. We refer to the speedup factor of $R = d^*/d$ as the overallocation ratio.

Resource overallocation is not a corner case, but a consequence of period-level bandwidth control for short CPU bursts. Consider a single-threaded, CPU-bound workload with $C = 5$ ms. Figure 1(a) compares its completion time under different parameters with C/f, while Figure 1(b) shows the corresponding overallocation ratio R. With the default $P = 100$ ms period, extremely small fractional allocations can still expose enough upfront quota for the workload to run at full-core speed. For example, at $f = 0.05$, the quota is $Q = 5$ ms, allowing the workload to finish in roughly 5 ms instead of the contract-compliant 100 ms, yielding a 20× overallocation. Also, there can be severe performance jitter near quantization boundaries (e.g., $f = 0.25$ and $P = 20$ ms). Reducing the period mitigates but does not eliminate overallocation. Overallocation can still reach 2×–20× across common period settings (e.g., 10–100 ms [37]), and even fine-grained 2 ms and 5 ms periods still permit up to 11-33% overallocation. Moreover, small periods narrow the usable fractional vCPU space due to Linux's 1 ms minimum quota: allocations with $fP < 1$ ms are invalid (dotted segments in the figure).

Note that, under period-level bandwidth control, overallocation can arise whenever a workload's instantaneous runnable parallelism exceeds its configured CPU allocation, including allocations of one core or more. This paper focuses on sub-core allocations and uses *fractional allocation* to denote $f \in (0, 1)$, a common resource setting (§2.2) where overallocation appears even for a single-threaded workload.

2.4 Motivation

The mismatch between period-level CPU bandwidth control and millisecond-scale workloads is not hypothetical. Recent work showed that CPU overallocation and quantization are widespread on major public serverless platforms, where short functions can complete substantially faster than the duration implied by their configured CPU allocation and experience performance jitter near quantization boundaries [37]. In particular, the following issues emerge as a result of inaccurate enforcement of the resource quota for tasks:

(1) Overallocation is inconsistent, as we show later in §4, where tasks of different durations or arrival times receive unequal excess resources, even with the same allocation configuration. The configured allocation therefore does not provide consistent CPU service across requests or tenants. This leads to **fairness issues** in multi-tenant systems.
(2) Inconsistent overallocation also distorts **billing and capacity planning**. Public serverless platforms usually bill configured allocation multiplied by wall-clock duration. Unequal effective rates price the same CPU service differently across tenants and may require corrective mechanisms (e.g., fixed per-request charges). Similarly, admission control and placement rely on configured CPU resources to account for capacity [3, 20, 50, 51, 53], while period-level enforcement can allow short bursts with fractional allocations to execute at full-core speed.
(3) Depending on C, a task may incur performance jitter of up to $P - Q$ due to deferral to the next period. Under overload, this bound increases with queue contention. When C is comparable to $P - Q$, which is the case for our target workloads, this performance variation becomes a dominant source of **execution time uncertainty** (§4).

These motivate a low-overhead, request-scale pacing mechanism that mitigates overallocation and keeps the execution duration of millisecond-scale workloads close to the C/f resource contract.

3 System Design

We present `μslice`, an eBPF-based mechanism for enforcing accurate sub-core allocation for millisecond-scale workloads. `μslice` enforces the C/f resource contract by releasing CPU grants only when the delivered CPU service remains within the configured fractional share. Figure 2 shows the system design of `μslice`, which is composed of a userspace controller and an in-kernel CPU pacer.

3.1 Userspace Controller

A privileged controller runs in userspace and serves as the control path for the CPU pacer (§3.2). It loads the CPU pacer as a `sched_ext` scheduler, configures it through BPF maps, exposes interfaces for configuration and pacing bypass, performs feature gating for individual cgroups, and exports runtime metrics. When policies or runtime configuration

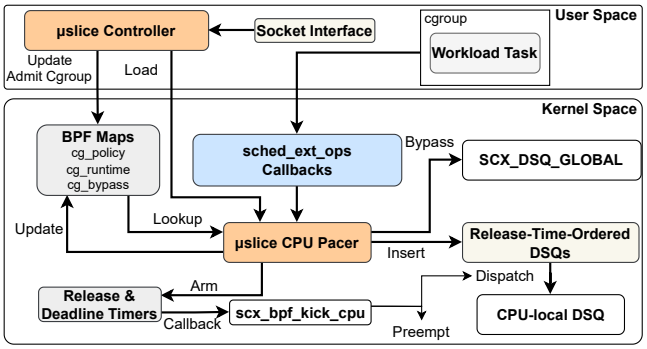

Figure 2: μslice System Architecture.

change, the controller updates the corresponding BPF maps. The controller attaches the `sched_ext struct_ops` map and keeps the returned BPF link open until the CPU pacer is detached. A feature-gate configuration determines which cgroups belong to the μslice managed set. The controller interacts with the CPU pacer mainly through three BPF maps (`BPF_MAP_TYPE_HASH`) with per-cgroup data: `cg_policy` stores the fractional CPU allocation policy for μslice-managed cgroups, `cg_runtime` stores the corresponding scheduler state (e.g., task progress and current state), and `cg_bypass` records whether temporary bypass is active. For each target cgroup, the controller reads the configured CPU share $f \in (0, 1)$, resolves its cgroupfs path to a kernel cgroup ID, derives bandwidth parameters with $Q = fP$, and stores f in `cg_policy` with the cgroup ID as the key. Cgroups without the enabled feature gate are absent from these BPF maps, and tasks within them will bypass the pacing path (§3.2).

Not every request in an admitted cgroup has to be paced (e.g., depending on service tiers and performance requirements). The controller therefore exposes a Unix `SEQPACKET` socket to trusted local data-path components (e.g., provider-managed gateway and queue proxy), which can issue IPC commands to efficiently update `cg_bypass` and enable/disable pacing bypass for a target cgroup on a per-request level. Additionally, the controller exposes configuration interfaces for system-wide pacing bypass (e.g., disabling pacing under high server load to reduce queuing delay) and CPU placement to restrict μslice-managed workloads to a selected CPU set.

3.2 CPU Pacer

μslice is non-work-conserving by design under normal operation. Runnable tasks in managed cgroups may be deferred until their next grant release time that complies with the resource contract. The CPU pacer implements this logic as a `sched_ext` BPF scheduler. Its pacing path starts from the `sched_ext_ops.enqueue` callback and enforces paced grants through the `dispatch`, `running`, `stopping`, `quiescent`, and BPF timer callbacks. For each task, the pacer derives the cgroup ID and looks up `cg_policy`, `cg_runtime`, and `cg_bypass`. Cgroups that are absent from these maps or marked as bypassed are inserted into an unpaced FIFO dispatch queue (e.g., `SCX_DSQ_GLOBAL`), where runnable tasks are dispatched whenever an eligible CPU is available. All other tasks are paced with the per-cgroup state recorded in `cg_runtime`, such as the anchor time, accumulated CPU time, next grant release time, timer state, and the state needed to determine whether the cgroup is waiting for a grant, eligible to run, or quiescent.

For a managed cgroup, the enqueue callback starts a new pacing epoch when the cgroup has no paced work that is running, queued, or waiting for release (i.e., quiescent state). The pacer records the current time as the anchor time a and sets the delivered service S to zero. The C/f resource contract requires that, at wall-clock time t, the configured share f allows at most $f(t-a)$ CPU service, and the remaining service allowance is $f(t-a)-S$. A grant of size g is compliant only if the cgroup remains within its share after the grant completes, namely $S + g \le f(t + g - a)$. Therefore, for sub-core allocation $f \in (0, 1)$, the largest grant that remains compliant with the resource contract is

$$g_c(t) = \max\left(0, \left\lfloor \frac{f(t-a) - S}{1-f} \right\rfloor\right) \tag{2}$$

The pacer should never release a grant g that would make $g > g_c(t)$. We use adaptive eligibility quantum g_e to determine when a task becomes eligible for grant release and a maximum quantum g_r to cap the slice. Therefore, for a target grant quantum $g = g_e$, the earliest release time is $r = a + \left\lceil \frac{S+g_e(1-f)}{f} \right\rceil$. In our setting, the base values of g_e and g_r are 250 μs and 8 ms. Under contention, low-share tenants ($f \le 0.3$) use $g_e = g_r = 1$ ms when eligible tasks from at least two cgroups are waiting to run, or no idle CPU is available. If r is in the future, the pacer inserts the task into a paced dispatch queue (DSQ), chosen by two-choice load balancing, with virtual time r and arms a `bpf_timer`. If r has passed, the task is eligible for dispatch immediately. When the timer fires, it marks the paced queue as having eligible work and kicks a CPU into the `dispatch` path (`scx_bpf_kick_cpu`). The `dispatch` callback moves the task to the corresponding CPU-local DSQ only after its r has passed.

When a task in a managed cgroup with bypass disabled starts running (`running` callback), the pacer computes the slice $g = min(g_c(t), g_r)$ using the service rule defined in Eq. 2. If dispatch is delayed by τ beyond the eligibility time r, the task can still catch up by receiving a potentially larger grant $g_c(r+\tau) = \left\lfloor \frac{f(r+\tau-a)-S}{1-f} \right\rfloor \approx g_e + \frac{f}{1-f}\tau$. If $g_c(t) < g_e$, the task is reparked instead of being overallocated. Otherwise, the pacer arms a deadline timer for the slice end. On `stopping`, the pacer computes the elapsed CPU time from `se.sum_exec_runtime` and updates the runtime state (e.g.,

S) in `cg_runtime`. The pacer invalidates pending timers and clears transient runtime state on the `quiescent` callback.

High-density deployments can enter a region where pacing adds queuing delay: many cgroups are parked waiting for future release times while new work continues to arrive. Therefore, the pacer maintains an event-driven count of active managed workloads, updated on scheduler-observed transitions to and from quiescence. An overload bypass mechanism is implemented. When the count exceeds a bypass activation threshold, the controller places admitted cgroups on the work-conserving path. It restores pacing only after the count remains below a recovery threshold for a hold-down window. We use thresholds of 500 and 400 active workloads and a 30 ms window in our setting. This count is not intended to model total server load. It is a local pressure signal that helps identify when paced work is accumulating, and the userspace controller can combine it with external metrics to enable system-wide pacing bypass under heavy load.

4 Evaluation

We compare `μslice` with baselines that vary the cgroup enforcement period, push the quota to the finest granularity, and apply proportional weight, in terms of overallocation, fairness across tenants, and performance jitter.

Experimental Setup. We run experiments on a KVM-based VM with 16 dedicated cores (Intel Xeon CPU E5-2673 v4) and Linux kernel 6.17 with EEVDF and cgroup v2. Each tenant is represented by one cgroup and one worker that runs one workload at a time, matching the single-concurrency execution model (i.e., no intra-runtime concurrency) used by AWS Lambda [2], Google Cloud Run functions (1st gen) [16], Huawei FunctionGraph (default setting) [7], and edge computing [12]. To perform controlled experiments, avoid request-level variation in CPU demand, and isolate the effect of CPU pacing, each request consumes a predefined, precise amount of CPU time (C ms) by spinning in calibrated compute chunks until `CLOCK_THREAD_CPUTIME_ID` reaches C.

Baselines. We use five baselines with different CPU bandwidth control configurations/policies from the commodity Linux kernel: `P=100 ms` and `P=20 ms` fix the period P at 100 ms (the Linux default, used by Google Cloud) and 20 ms (used by AWS Lambda), with quota $Q = fP$. `Q=1 ms` and `Q=2 ms` instead fix the quota Q at 1 (minimum quota supported by Linux) and 2 ms and set the period to $P = Q/f$, pushing period control to the finest granularity. `CPU Weight` is a work-conserving baseline, which sets `cpu.weight` proportional to f and does not enforce a hard CPU bandwidth cap.

We additionally use `MicroQuanta` [42, 54], a microsecond-scale scheduling class for efficient CPU sharing. We set P to 100 μs and $Q = fP$ and pin a filler task. `MicroQuanta` cannot enforce distinct fractional allocations across co-located tenants on a CPU and lacks cgroup support. Therefore, we include it only in the overallocation mitigation experiment.

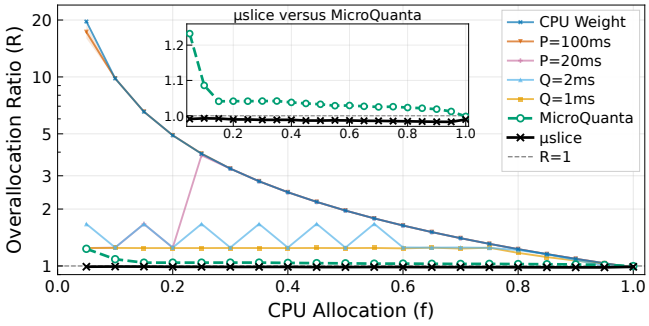

Figure 3: Overallocation Ratio versus CPU allocation.

Overallocation Mitigation. To quantify how effectively `μslice` mitigates CPU overallocation, we run a single workload with $C = 5$ ms and sweep the fractional allocation f from 0.05 to 1.0. As shown in Figure 3, across all f, `μslice` keeps the overallocation ratio close to one, between 0.983 and 0.993, a significant mitigation. In contrast, the strongest baseline from the commodity kernel, `Q=1 ms`, remains near 1.25 for $f \leq 0.75$, meaning that the workload receives 25% more CPU service than its configured allocation implies. `P=100 ms` shows similar results to the work-conserving baseline `CPU weight` without a hard CPU cap, because such a long period allows a 5 ms request to finish before quota enforcement takes effect. Finer-grained baselines (`P=20 ms` and `Q=2 ms`) show step-like patterns (e.g., $f = 0.25$ and $P = 20$ ms), suggesting performance jitter near quantization boundaries, while `μslice` eliminates such jitter. The results for `P=20 ms` and `P=100 ms` also conform to the theoretical analysis presented in Figure 1. `MicroQuanta` reduces overallocation with microsecond-scale runtime and period enforcement, achieving $R = 1.012$–1.233 for $f < 1$, where the remaining overallocation is mainly due to timer latency. While `MicroQuanta` outperforms commodity baselines, it cannot enforce independent sub-core allocations across tenants (i.e., co-located tenants with different f), requires a new scheduling class, and lacks cgroup support.

Evaluation with Heterogeneous Workloads. We then evaluate `μslice`'s effectiveness in mitigating overallocation across heterogeneous workload sets under substantial server load. We define a tenant-class grid $\mathcal{G} = \mathcal{C} \times \mathcal{F}$, where $\mathcal{C} = \{1, 2, \ldots, 10\}$ ms is the set of workload CPU demands and $\mathcal{F} = \{0.05, 0.10, \ldots, 1.00\}$ is the set of fractional CPU allocations. Each set contains one tenant from every one of the 200 classes $(C, f) \in \mathcal{G}$ and adds extra tenants according to a set-specific weighting over $\mathcal{G}$. Specifically, `Balanced` adds the remaining tenants by uniformly sampling the grid. `Short` samples $f \in \mathcal{F}$ uniformly and samples $C \in \mathcal{C}$ with weights $(10, 9, 8, 7, 6, 4, 3, 2, 1, 1)$ for $C = 1, \ldots, 10$, favoring short workloads. `Low Share` uniformly samples C and favors

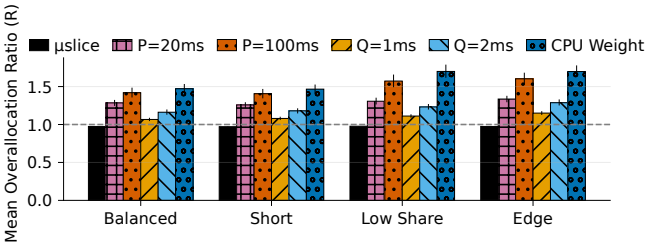

Figure 4: Average Overallocation Ratio across Heterogeneous Workload Sets.

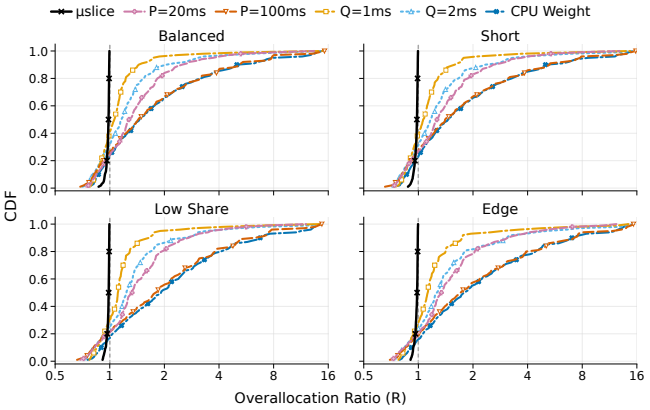

Figure 5: CDF of Overallocation Ratios.

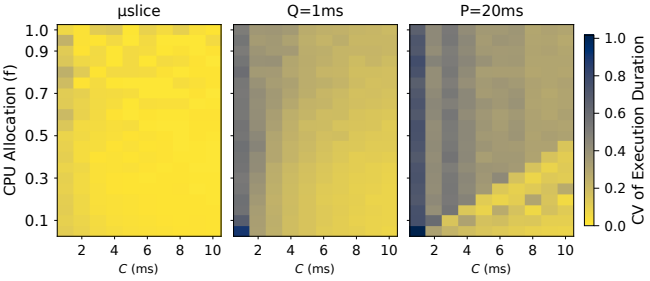

Figure 6: Coefficient of Variation of Execution Durations of the Edge Workload Set.

smaller fractional allocations. Edge favors both smaller C and f. Each tenant runs one workload at a time. After a workload completes, the worker idles for 100 ms before running the next request. We choose the number of tenants in each workload set so that, under this fixed release rule, all four sets impose a comparable aggregate CPU load of about 12 cores, or 75% of the experiment VM capacity. The workload sets contain 276, 304, 275, and 306 tenants, respectively. Each run uses a 5 s warmup period, a 30 s measurement window, and a 5 s cooldown period, and we repeat runs 10 times and report results only from the measurement window.

Figure 4 reports the mean overallocation ratio R for the four heterogeneous workload sets. Across all sets, μslice keeps mean R close to one (geometric mean 0.974), while the strongest baseline Q=1 ms achieves mean R between 1.067 and 1.15. The period-based baselines are further from the target: P=20 ms and P=100 ms have mean R between 1.258 and 1.602.

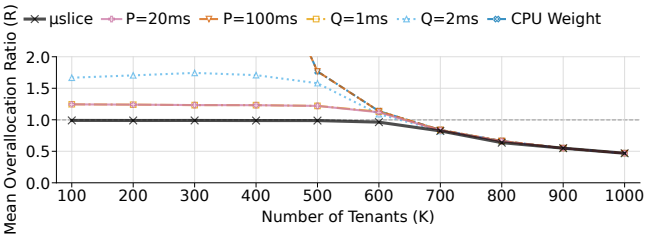

Figure 7: High-Density Behavior of μslice.

The gap is largest on the Edge workload set, which contains more tenants with short C and low f, leading to more overallocation under the period-level bandwidth control mechanism. The request-level overallocation ratio distribution (Figure 5) shows that this effect is not uniform across tenants. Across the four workload sets, μslice produces a narrow distribution close to $R = 1$ ($p5 = 0.924$ and $p95 = 0.994$), meaning that most tenants receive CPU service close to their configured fractional allocation. In contrast, the baselines have longer tails on both sides. Even the strongest baseline Q=1ms has a wide 5–95th percentile range of 0.803–2.070. Thus, baselines overallocate on average and give different tenants substantially different effective CPU rates.

Figure 6 further shows how this imbalance appears as performance jitter, which reports the coefficient of variation (CV) of request execution duration for each (C, f) class in the Edge workload set. The baselines show high CVs for short C and low f. Since all requests in a class execute the same measured CPU demand, this suggests such variation comes from the CPU-control mechanism rather than application-level variation. μslice removes most of this jitter by accurately pacing CPU service: its geometric mean CV is 0.023, compared with 0.210 for Q=1 ms and 0.315 for P=20 ms.

These results expose a practical fairness problem for commodity baselines. A tenant with $R > 1$ receives more CPU service per unit of wall-clock time than its configured fractional allocation implies, while a tenant with $R < 1$ receives less. Under allocation- and wall-clock-duration-based billing models that are widely used by cloud providers [6, 47], this directly shifts cost and performance across tenants: some tenants obtain faster completion for the same configured allocation, while others pay longer wall-clock time for the same CPU demand and may experience performance jitter. μslice effectively reduces performance variations and makes effective CPU service consistent across tenants.

High-Density Behavior and Overheads. We perform stress tests for μslice by varying the number of homogeneous tenants (K) with $C = 5$ ms and $f = 0.05$. As shown in Figure 7, for $K \leq 500$, μslice keeps mean R tightly bounded between 0.987 and 0.990. At $K = 600$, mean R decreases slightly to 0.962, while the overload bypass remains inactive. Beyond

this point, the density approaches the practical dispatch capacity of the server. At $K \geq 700$, μslice converges towards the commodity scheduler. At $K = 800$, overload bypass is used for 48% of requests and 99% at $K \geq 900$. This suggests that overload does not create an unbounded pacing backlog for μslice. When local pressure exceeds overload thresholds (§3.2), scheduling falls back to the work-conserving path.

We measure the overhead of μslice at $K = 500$. The timer, dispatch, and handoff paths add 50.3 μs per request on average ($p99 = 78.6\,\mu$s). Relative to the Q=1 ms EEVDF baseline, μslice increases the workload-level and system-wide context-switch rates by 6.77% and 20.53%, respectively, and incurs 207.33 μs of additional kernel time per request on average. The path-level measurement captures wall-clock latency within μslice, whereas the kernel time increase also includes scheduling and context switch overhead incurred by pacing. Most of the remaining gap from the ideal completion time stems from platform wakeup and scheduling latency due to contention under high load (e.g., an average dispatch queue delay of 0.90 ms at $K = 500$). In addition, the BPF maps require around 0.43 MB for 1000 managed workloads.

5 Discussion

μslice is designed to enforce accurate sub-core allocation for millisecond-scale workloads, rather than to minimize workload latency by exploiting excess CPU resources outside the resource contract. Period-level bandwidth control allows a short request to run at full-core speed until its quota is consumed, so the configured share no longer bounds per-request service. μslice fixes this mismatch without changing the cgroup interface: it preserves the share but paces workloads so that transient quota availability does not implicitly create a full-core service class. Removing this excess service can increase latency relative to an overallocating baseline, but this increase is the expected consequence of complying with the fractional allocation. In return, μslice makes delivered CPU service more predictable, narrows variability, and improves inter-tenant fairness.

The degree of this excess service depends on the workload and sets the boundary of where μslice helps. μslice targets short, CPU-bound requests under sub-core allocations, where period-level quota availability creates the largest gap. It does not target long jobs or I/O-dominant requests, and the current evaluation does not cover requests with CPU bursts interleaved with substantial I/O. It assumes single-concurrency execution, where an instance serves one request at a time, as in major serverless platforms such as AWS Lambda [2] and Google Cloud Run [16]. Pacing concurrent requests and parallel threads within a single cgroup would require attributing CPU time to individual requests and coordinating across threads and CPUs, which the current design does not support. We leave this extension to future work.

6 Related Work

Fine-grained CPU Scheduling. A wide range of work makes CPU scheduling flexible rather than fixed in the kernel or designs schedulers for short, latency-sensitive work. ghOSt delegates scheduling decisions to userspace agents [22], sched_ext enables verified policies as eBPF programs without a kernel fork [30], and many systems extend scheduler frameworks that separate policy from mechanism [28, 34, 44, 57]. Prior systems have also explored fine-grained schedulers for microsecond-scale tasks with core reallocation, request preemption, core reservation, queuing, and scheduling policies [11, 14, 23, 27, 41, 43, 46]. Junction uses a dedicated polling core to dynamically allocate cores among userspace instances for low-latency execution, rather than enforce fractional CPU allocations [13]. Snap proposes MicroQuanta for microsecond-scale CPU sharing through runtime/period control, but does not independently enforce fractional allocations among tasks sharing a CPU [42, 54]. Also, these systems mainly optimize latency or work conservation. μslice instead targets resource contract compliance for millisecond-scale workloads and paces each admitted cgroup according to its own configured sub-core allocation, while preserving the existing cgroup CPU allocation interface.

eBPF for Kernel Resource Management. eBPF has grown from observability, packet processing, and security [19, 21, 24] into a widely used in-kernel extension mechanism for various optimizations, including storage [35, 58, 59], caching [15], memory prefetching [5], application acceleration [31, 60], and scheduling [30]. Inspired by these works, we leverage eBPF to implement a pacing mechanism in the scheduler fast path for resource contract compliance.

7 Conclusion

This work presents μslice, an eBPF-based scheduling framework for enforcing accurate sub-core allocation for millisecond-scale workloads in multi-tenant environments. Our results show μslice substantially mitigates CPU overallocation, reduces performance variations, and improves fairness across tenants over commodity baselines.

Acknowledgments

We thank the anonymous reviewers for their valuable feedback, and Marios Kogias for encouraging us to keep pursuing an eBPF-based approach to this problem. This work was supported by the Natural Sciences and Engineering Research Council of Canada (NSERC), through research grants RGPIN-2021-03714 and DGECR-2021-00462, and the Canada Graduate Scholarship (CGS D) program.

References

[1] AWS. 2026. Configure Lambda function memory - AWS Lambda. https://docs.aws.amazon.com/lambda/latest/dg/configuration-memory.html [Online; accessed June-17-2026].

[2] AWS. 2026. Understanding Lambda function scaling. https://docs.aws.amazon.com/lambda/latest/dg/lambda-concurrency.html [Online; accessed June-20-2026].

[3] Mohammadamin Baqershahi, Changyuan Lin, Visal Saosuo, Paul Chen, and Mohammad Shahrad. 2026. Hierarchical Integration of WebAssembly in Serverless for Efficiency and Interoperability. In *23rd USENIX Symposium on Networked Systems Design and Implementation (NSDI 26)*. 2481–2498.

[4] Tingjia Cao, Andrea C Arpaci-Dusseau, Remzi H Arpaci-Dusseau, and Tyler Caraza-Harter. 2025. Making Serverless Pay-For-Use a Reality with Leopard. In *22nd USENIX Symposium on Networked Systems Design and Implementation (NSDI 25)*. 189–204.

[5] Xuechun Cao, Shaurya Patel, Soo Yee Lim, Xueyuan Han, and Thomas Pasquier. 2024. FetchBPF: Customizable prefetching policies in linux with eBPF. In *2024 USENIX Annual Technical Conference (USENIX ATC 24)*. 369–378.

[6] Google Cloud. 2026. Cloud Run pricing. https://cloud.google.com/run/pricing [Online; accessed June-24-2026].

[7] Huawei Cloud. 2025. Concurrency Managing Functions User Guide FunctionGraph. https://support.huaweicloud.com/intl/en-us/usermanual-functiongraph/functiongraph_01_0303.html [Online; accessed August-11-2026].

[8] Cloudflare. 2026. Limits · Cloudflare Workers docs. https://developers.cloudflare.com/workers/platform/limits/ [Online; accessed June-4-2026].

[9] Marcin Copik, Grzegorz Kwasniewski, Maciej Besta, Michal Podstawski, and Torsten Hoefler. 2021. SeBS: A serverless benchmark suite for function-as-a-service computing. In *Proceedings of the 22nd International Middleware Conference*. 64–78.

[10] Jonathan Corbet. 2023. An EEVDF CPU scheduler for Linux. https://lwn.net/Articles/925371/ [Online; accessed June-12-2026].

[11] Kumar Kartikeya Dwivedi, Rishabh Iyer, and Sanidhya Kashyap. 2024. Fast, flexible, and practical kernel extensions. In *Proceedings of the ACM SIGOPS 30th Symposium on Operating Systems Principles*. 249–264.

[12] Fastly. 2026. Getting started with Compute | Fastly Documentation. https://www.fastly.com/documentation/guides/compute/getting-started-with-compute/ [Online; accessed August-11-2026].

[13] Joshua Fried, Gohar Irfan Chaudhry, Enrique Saurez, Esha Choukse, Inigo Goiri, Sameh Elnikety, Rodrigo Fonseca, and Adam Belay. 2024. Making Kernel Bypass Practical for the Cloud with Junction. In *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*. 55–73.

[14] Joshua Fried, Zhenyuan Ruan, Amy Ousterhout, and Adam Belay. 2020. Caladan: Mitigating interference at microsecond timescales. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. 281–297.

[15] Yoann Ghigoff, Julien Sopena, Kahina Lazri, Antoine Blin, and Gilles Muller. 2021. BMC: Accelerating memcached using safe in-kernel caching and pre-stack processing. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*. 487–501.

[16] Google. 2026. Compare Cloud Run functions | Google Cloud Documentation. https://docs.cloud.google.com/run/docs/functions/comparison [Online; accessed August-4-2026].

[17] Google. 2026. Configure CPU limits for services | Cloud Run | Google Cloud Documentation. https://docs.cloud.google.com/run/docs/configuring/services/cpu [Online; accessed June-17-2026].

[18] Robert Grandl, Ganesh Ananthanarayanan, Srikanth Kandula, Sriram Rao, and Aditya Akella. 2014. Multi-resource packing for cluster schedulers. *ACM SIGCOMM Computer Communication Review* 44, 4 (2014), 455–466.

[19] Brendan Gregg. 2019. *BPF performance tools*. Addison-Wesley Professional.

[20] Adam Hall, Anirudh Sarma, Esha Choukse, Umakishore Ramachandran, and Sameh Elnikety. 2026. Harvesting Spare CPU Resources in Container Systems. In *23rd USENIX Symposium on Networked Systems Design and Implementation (NSDI 26)*. 2323–2337.

[21] Toke Høiland-Jørgensen, Jesper Dangaard Brouer, Daniel Borkmann, John Fastabend, Tom Herbert, David Ahern, and David Miller. 2018. The express data path: Fast programmable packet processing in the operating system kernel. In *Proceedings of the 14th international conference on emerging networking experiments and technologies*. 54–66.

[22] 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 *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*. 588–604.

[23] Rishabh Iyer, Musa Unal, Marios Kogias, and George Candea. 2023. Achieving microsecond-scale tail latency efficiently with approximate optimal scheduling. In *Proceedings of the 29th Symposium on Operating Systems Principles*. 466–481.

[24] Jinghao Jia, YiFei Zhu, Dan Williams, Andrea Arcangeli, Claudio Canella, Hubertus Franke, Tobin Feldman-Fitzthum, Dimitrios Skarlatos, Daniel Gruss, and Tianyin Xu. 2023. Programmable system call security with eBPF. *arXiv preprint arXiv:2302.10366* (2023).

[25] Artjom Joosen. 2025. Huawei Cloud Production FaaS Trace Data Releases. https://github.com/sir-lab/data-release [Online; accessed June-4-2026].

[26] Artjom Joosen, Ahmed Hassan, Martin Asenov, Rajkarn Singh, Luke Darlow, Jianfeng Wang, Qiwen Deng, and Adam Barker. 2025. Serverless cold starts and where to find them. In *Proceedings of the Twentieth European Conference on Computer Systems*. 938–953.

[27] Kostis Kaffes, Timothy Chong, Jack Tigar Humphries, Adam Belay, David Mazières, and Christos Kozyrakis. 2019. Shinjuku: Preemptive Scheduling for μsecond-scale Tail Latency. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*. 345–360.

[28] Kostis Kaffes, Jack Tigar Humphries, David Mazières, and Christos Kozyrakis. 2021. Syrup: User-defined scheduling across the stack. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*. 605–620.

[29] Linux Kernel. 2015. Control Group v2. https://www.kernel.org/doc/Documentation/cgroup-v2.txt [Online; accessed June-12-2026].

[30] Linux Kernel. 2026. Extensible Scheduler Class. https://docs.kernel.org/scheduler/sched-ext.html [Online; accessed June-19-2026].

[31] Kornilios Kourtis, Animesh Trivedi, and Nikolas Ioannou. 2020. Safe and efficient remote application code execution on disaggregated NVM storage with eBPF. *arXiv preprint arXiv:2002.11528* (2020).

[32] Kubernetes. 2026. Node Autoscaling. https://kubernetes.io/docs/concepts/cluster-administration/node-autoscaling/ [Online; accessed August-8-2026].

[33] Kubernetes. 2026. Resource Management for Pods and Containers. https://kubernetes.io/docs/concepts/configuration/manage-resources-containers/ [Online; accessed June-24-2026].

[34] Baptiste Lepers, Redha Gouicem, Damien Carver, Jean-Pierre Lozi, Nicolas Palix, Maria-Virginia Aponte, Willy Zwaenepoel, Julien Sopena, Julia Lawall, and Gilles Muller. 2020. Provable multicore schedulers with Ipanema: application to work conservation. In *Proceedings of the Fifteenth European Conference on Computer Systems*. 1–16.

[35] Chuandong Li, Ran Yi, Zonghao Zhang, Jing Liu, Changwoo Min, Jie Zhang, Yingwei Luo, Xiaolin Wang, Zhenlin Wang, and Diyu Zhou. 2025. Aeolia: A Fast and Secure Userspace Interrupt-Based Storage Stack. In *Proceedings of the ACM SIGOPS 31st Symposium on Operating Systems Principles*. 479–495.

[36] Suyi Li, Wei Wang, Jun Yang, Guangzhen Chen, and Daohe Lu. 2023. Golgi: Performance-aware, resource-efficient function scheduling for serverless computing. In *Proceedings of the 2023 ACM Symposium on Cloud Computing*. 32–47.

[37] Changyuan Lin, Yuanzhi Ma, and Mohammad Shahrad. 2026. Demystifying serverless costs on public platforms: Bridging billing, architecture, and os scheduling. In *Proceedings of the 21st European Conference on Computer Systems*. 1964–1981.

[38] Li Liu, Haoliang Wang, An Wang, Mengbai Xiao, Yue Cheng, and Songqing Chen. 2021. Mind the gap: Broken promises of CPU reservations in containerized multi-tenant clouds. In *Proceedings of the ACM Symposium on Cloud Computing*. 243–257.

[39] Qingyuan Liu, Yanning Yang, Dong Du, Yubin Xia, Ping Zhang, Jia Feng, James R Larus, and Haibo Chen. 2024. Harmonizing Efficiency and Practicability: Optimizing Resource Utilization in Serverless Computing with Jiagu. In *2024 USENIX Annual Technical Conference (USENIX ATC 24)*. 1–17.

[40] Shutian Luo, Huanle Xu, Kejiang Ye, Guoyao Xu, Liping Zhang, Guodong Yang, and Chengzhong Xu. 2022. The power of prediction: microservice auto scaling via workload learning. In *Proceedings of the 13th symposium on cloud computing*. 355–369.

[41] Zhihong Luo, Sam Son, Dev Bali, Emmanuel Amaro, Amy Ousterhout, Sylvia Ratnasamy, and Scott Shenker. 2024. Efficient microsecond-scale blind scheduling with Tiny Quanta. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*. 305–319.

[42] Michael Marty, Marc de Kruijf, Jacob Adriaens, Christopher Alfeld, Sean Bauer, Carlo Contavalli, Michael Dalton, Nandita Dukkipati, William C. Evans, Steve Gribble, Nicholas Kidd, Roman Kononov, Gautam Kumar, Carl Mauer, Emily Musick, Lena Olson, Erik Rubow, Michael Ryan, Kevin Springborn, Paul Turner, Valas Valancius, Xi Wang, and Amin Vahdat. 2019. Snap: A microkernel approach to host networking. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*. 399–413.

[43] Sarah McClure, Amy Ousterhout, Scott Shenker, and Sylvia Ratnasamy. 2022. Efficient Scheduling Policies for Microsecond-Scale Tasks. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*. 1–18.

[44] Samantha Miller, Anirudh Kumar, Tanay Vakharia, Ang Chen, Danyang Zhuo, and Thomas Anderson. 2024. Enoki: High velocity linux kernel scheduler development. In *Proceedings of the Nineteenth European Conference on Computer Systems*. 962–980.

[45] Nima Nasiri, Nalin Munshi, Simon Daniel Moser, Marius Pirvu, Vijay Sundaresan, Daryl Maier, Thatta Premnath, Norman Böwing, Sathish Gopalakrishnan, and Mohammad Shahrad. 2026. In-production characterization of an open source serverless platform and new scaling strategies. In *Proceedings of the 21st European Conference on Computer Systems*. 2054–2072.

[46] Amy Ousterhout, Joshua Fried, Jonathan Behrens, Adam Belay, and Hari Balakrishnan. 2019. Shenango: Achieving high CPU efficiency for latency-sensitive datacenter workloads. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*. 361–378.

[47] Amazon Web Services. 2026. Serverless Computing – AWS Lambda Pricing – Amazon Web Services. https://aws.amazon.com/lambda/pricing/ [Online; accessed June-24-2026].

[48] Mohammad Shahrad, Jonathan Balkind, and David Wentzlaff. 2019. Architectural implications of function-as-a-service computing. In *Proceedings of the 52nd annual IEEE/ACM international symposium on microarchitecture*. 1063–1075.

[49] Simon Shillaker and Peter Pietzuch. 2020. Faasm: Lightweight isolation for efficient stateful serverless computing. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*. 419–433.

[50] Lalith Suresh, João Loff, Faria Kalim, Sangeetha Abdu Jyothi, Nina Narodytska, Leonid Ryzhyk, Sahan Gamage, Brian Oki, Pranshu Jain, and Michael Gasch. 2020. Building scalable and flexible cluster managers using declarative programming. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. 827–844.

[51] Chunqiang Tang, Kenny Yu, Kaushik Veeraraghavan, Jonathan Kaldor, Scott Michelson, Thawan Kooburat, Aravind Anbudurai, Matthew Clark, Kabir Gogia, Long Cheng, Ben Christensen, Alex Gartrell, Maxim Khutornenko, Sachin Kulkarni, Marcin Pawlowski, Tuomas Pelkonen, Andre Rodrigues, Rounak Tibrewal, Vaishnavi Venkatesan, and Peter Zhang. 2020. Twine: A unified cluster management system for shared infrastructure. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. 787–803.

[52] Paul Turner, Bharata B Rao, and Nikhil Rao. 2010. CPU bandwidth control for CFS. In *Linux Symposium*, Vol. 10. Citeseer, 245–254.

[53] Abhishek Verma, Luis Pedrosa, Madhukar Korupolu, David Oppenheimer, Eric Tune, and John Wilkes. 2015. Large-scale cluster management at Google with Borg. In *Proceedings of the tenth european conference on computer systems*. 1–17.

[54] Xi Wang. 2019. [PATCH 0/1] Add micro quanta scheduling class. https://lore.kernel.org/all/20190906093412.233463-1-xii@google.com/ [Online; accessed July-31-2026].

[55] Minchen Yu, Tingjia Cao, Wei Wang, and Ruichuan Chen. 2023. Following the data, not the function: Rethinking function orchestration in serverless computing. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*. 1489–1504.

[56] Yanqi Zhang, Íñigo Goiri, Gohar Irfan Chaudhry, Rodrigo Fonseca, Sameh Elnikety, Christina Delimitrou, and Ricardo Bianchini. 2021. Faster and cheaper serverless computing on harvested resources. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*. 724–739.

[57] Yusheng Zheng, Tong Yu, Yiwei Yang, Minghui Jiang, Xiangyu Gao, Jianchang Su, Yanpeng Hu, Wenan Mao, Wei Zhang, Dan Williams, and Andi Quinn. 2025. gpu_ext: Extensible OS Policies for GPUs via eBPF. *arXiv preprint arXiv:2512.12615* (2025).

[58] Yuhong Zhong, Haoyu Li, Yu Jian Wu, Ioannis Zarkadas, Jeffrey Tao, Evan Mesterhazy, Michael Makris, Junfeng Yang, Amy Tai, Ryan Stutsman, and Asaf Cidon. 2022. XRP: In-Kernel Storage Functions with eBPF. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*. 375–393.

[59] Yuhong Zhong, Hongyi Wang, Yu Jian Wu, Asaf Cidon, Ryan Stutsman, Amy Tai, and Junfeng Yang. 2021. BPF for storage: an exokernel-inspired approach. In *Proceedings of the Workshop on Hot Topics in Operating Systems*. 128–135.

[60] Beihao Zhou, Samer Al-Kiswany, and Mina Tahmasbi Arashloo. 2025. Toward eBPF-Accelerated Pub-Sub Systems. In *Proceedings of the 3rd Workshop on eBPF and Kernel Extensions*. 38–44.