An Exploratory Study of ALPS Scheduler Performance for I/O and CPU-Bound Serverless Workloads

Amogh Dixit
asdixit3@wisc.edu

Dhruv Nikhilbhai Kachhadia
kachhadia@wisc.edu

Maithil Rupesh Mehta
mmehta28@wisc.edu

Abstract

Serverless computing, or Function-as-a-Service (FaaS), has radically simplified cloud application development by abstracting infrastructure management. However, the unique characteristics of serverless workloads such as short lifespans, high concurrency and extreme burstiness expose significant limitations in traditional operating system schedulers like the Linux Completely Fair Scheduler (CFS), which prioritizes fairness over latency. Recent research introduced ALPS (Adaptive Learning, Priority Scheduler)[1], a state-of-the-art solution that uses historical CPU usage data to implement a Shortest Remaining Processing Time (SRPT) policy, demonstrating significant reductions in function turnaround time. In this paper, we conduct an exploratory study of ALPS, reproducing its core performance benefits for CPU-bound workloads while critically examining its behavior under I/O-intensive scenarios. We hypothesize that ALPS's reliance on CPU execution time as a primary learning signal creates a "blind spot" for I/O-bound functions, causing them to be misclassified as high-priority short jobs. Using a custom implementation based on the ghOSt kernel framework, we demonstrate that this misclassification leads to aggressive prioritization of I/O tasks at the expense of compute-heavy functions in mixed workloads. Our evaluation reveals that while ALPS reduces the 99th percentile latency of I/O-bound tasks by up to 30% compared to CFS, this advantage imposes a penalty on concurrent CPU-bound tasks, which suffer a 13–15% increase in latency. These findings confirm the efficacy of SRPT for short jobs but empirically motivate the need for I/O-aware scheduling mechanisms to prevent starvation in mixed serverless environments.

1 Introduction

Serverless computing (FaaS) has emerged as a dominant paradigm in modern cloud architecture, allowing developers to deploy code without managing the underlying servers. Unlike traditional long-running applications, FaaS workloads are characterized by ephemeral, stateless functions that scale to zero and exhibit highly unpredictable, bursty invocation patterns. While this model offers economic and operational benefits to users, it places immense pressure on the underlying Operating System (OS) scheduler.

The default Linux scheduler, the Completely Fair Scheduler (CFS), is designed to ensure proportional CPU sharing among processes over long intervals. While effective for traditional workloads, this "fairness-first" approach is often detrimental to serverless functions.

To address these inefficiencies, recent work proposed ALPS (Adaptive Learning, Priority Scheduler). ALPS is an application-aware scheduler that leverages the observation that function execution times are often predictable based on historical data. By learning the expected CPU duration of functions, ALPS approximates a Shortest Remaining Processing Time (SRPT) policy, prioritizing shorter jobs to minimize average turnaround time.

However, existing evaluations of ALPS focus predominantly on CPU usage as the sole metric for scheduling priority. This creates a potential gap in handling real-world serverless workloads, which often spend a significant portion of their lifecycle waiting on external I/O, such as database queries or HTTP requests. A large-scale characterization of Azure Functions revealed that a substantial percentage of serverless workloads are I/O-bound. We hypothesize that ALPS's design, which prioritizes tasks with low CPU time, may inadvertently mis-

classify long-running I/O-bound functions as "short" jobs. This could lead to a scenario where I/O-heavy tasks are aggressively scheduled, only to immediately block again, causing frequent, wasteful context switches that degrade system throughput.

In this paper, we bridge this gap by conducting an independent evaluation of ALPS using the Google ghOSt (userspace delegation) framework.

We make the following contributions:

1. **Reproduction of Baseline Results:** We reproduce the key findings of the ALPS paper (specifically the baseline comparison in Figure 2), confirming its superiority over CFS for CPU-bound microbenchmarks
2. **Evaluation of I/O-Bound Workloads**: We extend the evaluation to include synthetic I/O-bound and mixed workloads, an area not deeply explored in the original study.
3. **Empirical Motivation for I/O-Awareness:** We provide experimental evidence suggesting that while ALPS is highly effective for compute tasks, future serverless schedulers must incorporate I/O wait times into their priority calculations to handle mixed workloads efficiently.

2 Background

This section details the workload characteristics of serverless functions, the limitations of current Linux scheduling for these workloads, and the design principles of the ALPS scheduler.

2.1 Serverless Workloads

Serverless computing, or Function-as-a-Service (FaaS), represents a shift in cloud application deployment where developers focus solely on code logic while the platform manages infrastructure scaling. These workloads exhibit unique characteristics distinct from traditional cloud applications:

1. **Ephemeral and Bursty:** Serverless functions are highly ephemeral, with execution durations ranging from a few milliseconds to a few minutes. They are also characterized by extreme burstiness and high concurrency, where application demand can spike unpredictably.
2. **Variable Resource Requirements:** While often lightweight, function invocations can be CPU-intensive or I/O-bound. Real-world characterization studies, such as Serverless in the Wild by Shahrad et al.[2], indicate that a significant portion of production workloads (e.g., Azure Functions) have extremely short execution times, with 37.3% lasting less than 300 ms.
3. **I/O Prevalence:** Many serverless use cases are event-driven, involving heavy interaction with external services like databases or object storage. These I/O-bound functions often spend the majority of their lifecycle in a blocked state waiting for responses rather than consuming CPU cycles.

2.2 Linux CFS Limitations

The default process scheduler in Linux, the Completely Fair Scheduler (CFS), was designed to handle general-purpose workloads by ensuring proportional CPU sharing among competing tasks. While effective for long-running processes, this design philosophy creates significant inefficiencies for serverless workloads: from traditional cloud applications:

1. **Fairness vs. Latency:** CFS allocates time slices to ensure fairness, which can be detrimental to short-lived jobs. Short functions may be preempted before they can complete, leading to prolonged "scheduling cycles" where tasks wait in the run queue for their next turn.
2. **Context Switching Overhead:** The frequent preemption required to maintain fairness results in high context switching overhead. For functions with execution times in the millisecond range, the time spent context switching and waiting in the run queue can exceed the actual execution time of the function itself.
3. **Tail Latency Degradation:** By treating all tasks equally regardless of their remaining work, CFS fails to prioritize short jobs that could finish quickly. This leads to poor tail latency for short, bursty serverless functions, as they are forced to wait behind longer-running tasks.

2.3 ALPS Overview

To address these limitations, recent research introduced ALPS (Adaptive Learning, Priority Scheduler), a specialized kernel scheduler designed explicitly for serverless environments.

1. **Design Goal:** ALPS aims to minimize function turnaround time by approximating the Short-

est Remaining Processing Time (SRPT) policy, which is theoretically optimal for minimizing average response time.

2. **Adaptive Learning:** Unlike static priority schemes, ALPS is application-aware. It employs a user-space frontend that analyzes historical function traces to "learn" the expected CPU duration of incoming function invocations. This allows it to predict which functions are likely to be short and prioritize them accordingly.
3. **Performance Advantage:** By dynamically assigning higher priorities to predicted short jobs, ALPS drastically reduces the waiting time for short functions. The authors reported a 57.2% reduction in average function execution duration compared to CFS in CPU-bound scenarios.

3 ALPS Architecture

This section describes the design and implementation of ALPS (Adaptive Learning, Priority Scheduler). As illustrated in the Figure 1, ALPS employs a novel split-level architecture that decouples policy learning (frontend) from policy enforcement (backend), ensuring both flexibility and kernel-level efficiency.

3.1 Design Philosophy

The architecture of ALPS is driven by the need to reconcile the conflict between complex, application-aware policy decisions and the low-latency requirements of kernel scheduling.

1. **Decoupled Architecture:** ALPS separates the "intelligence" (frontend) from the "mechanism" (backend). The frontend runs in user space, where it has the luxury of performing computationally intensive simulations without stalling the kernel.
2. **Kernel Transparency:** By injecting policies via eBPF rather than modifying the core kernel source code, ALPS maintains compatibility with the existing Linux CFS infrastructure. This ensures that the scheduler benefits from the stability and work-conserving properties of CFS while overlaying a smarter priority mechanism.
3. **Learning-Driven Priorities:** Instead of relying on static priorities, ALPS continuously learns from recent execution history to predict which functions are "short" (high priority) and which are "long" (low priority), approximating an optimal SRPT schedule.

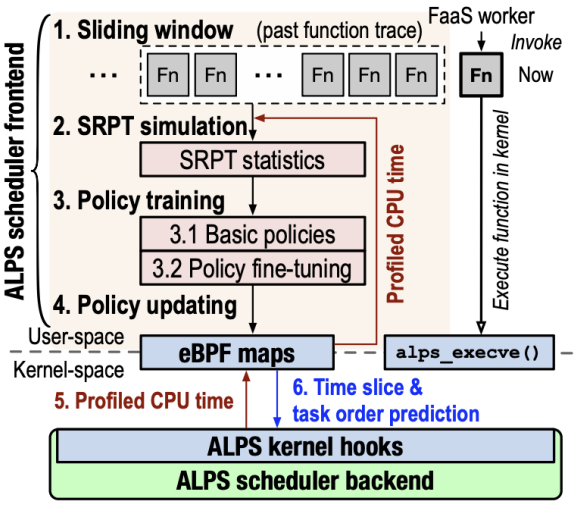

Figure 1: Overview of ALPS architecture

3.2 Frontend (User-Space Learning)

The frontend acts as the brain of the scheduler, responsible for analyzing past behavior to guide future decisions. It operates in a continuous loop:

1. **Trace Collection:** It maintains a sliding window of recent function execution traces, capturing the duration and arrival patterns of recent invocations.
2. **SRPT Simulation:** The frontend runs an offline simulation of the Shortest Remaining Processing Time (SRPT) policy on this historical data. This simulation determines how an ideal scheduler would have handled these tasks to minimize turnaround time.
3. **Policy Training:** Based on the simulation results, the frontend trains two key policies: a Task Ordering Policy (to classify functions into priority groups) and a Time Slice Policy (to assign dynamic time quanta). These policies are designed to prioritize functions predicted to be short.
4. **Policy Update:** Once trained, the updated policy parameters are pushed to the kernel via eBPF maps, making them immediately available to the backend.

3.3 Backend (Kernel-Space Enforcement)

The backend serves as the muscle, executing the frontend's decisions with minimal overhead deep within the kernel.

1. **eBPF Hooks:** ALPS attaches eBPF programs to specific hooks in the Linux CFS scheduler (e.g., during context switches or execve calls). These hooks allow ALPS to intercept scheduling events and inject custom logic.
2. **alps_execve()**: When a function starts, a custom system call alps_execve() tags the task with a unique function ID (func_id), allowing the kernel to associate the process with its learned policy profile.
3. **Priority Enforcement:** During scheduling decisions, the backend retrieves the task's assigned rank and time slice from the shared eBPF maps. It then manipulates the task's vruntime (virtual runtime) in CFS to ensure that high-priority (short) jobs are picked first from the Red-Black tree.
4. **Efficiency:** The backend is highly optimized; evaluating the task ordering policy takes only 66 nanoseconds per call, adding negligible overhead to the context switch path.

4 Related Work

The evolution of serverless scheduling can be traced from classical operating system theory to modern, application-aware mechanisms designed specifically for the cloud.

Modern schedulers are built on foundational ideas from traditional operating system research. Waldspurger and Weihl's seminal work [3] on Lottery Scheduling introduced randomized mechanisms for proportional-share resource management. While not directly applied in FaaS, its core principle of fair sharing profoundly influenced the design of the Linux Completely Fair Scheduler (CFS), the default scheduler in most cloud environments. However, recent studies have highlighted the limitations of these general-purpose schedulers in serverless contexts. Isstaif and Mortier[4] analyzed CFS and demonstrated that its focus on long-term fairness is fundamentally mismatched with ephemeral functions. For serverless workloads, CFS's fairness mechanisms often lead to frequent, high-overhead context switches that can consume more time than the function's actual execution. They proposed CFS-LLF (Least Loaded First) to adjust dynamic priorities, favoring the long tail of short, idle functions. A critical turning point in understanding these workloads was the 2020 study Serverless in the Wild by Shahrad et al.,[2] which provided the first comprehensive characterization of production FaaS workloads from Azure. Their findings showed that 50% of functions execute in less than one second, and 81% of applications are invoked at most once per minute. This established the central trade-off of serverless scheduling: balancing the cost of idle resources against the latency of cold starts, motivating the need for predictive scheduling policies.

Given the difficulty of modifying the kernel in production environments, researchers have explored solutions at the platform level. Kaffes et al.[5] took a "first principles" approach with Hermod, an application-level scheduler that challenges the common practice of late binding. Hermod combines early binding with worker-level processor sharing to avoid head-of-line blocking, achieving up to 85% lower function slowdown. As a direct predecessor to ALPS, Fu et al.[6] developed SFS (Smart OS Scheduling for Serverless Functions), which operates entirely in user space. SFS employs a filtering policy to approximate SRPT behavior, directing predicted short functions to a low-latency FIFO queue. However, its user-space nature suffers from high polling overhead and a lack of transparency into kernel-level events.

Recent approaches have incorporated machine learning to handle the high variability of FaaS workloads. FaaSRank (Yu et al.)[7] uses deep reinforcement learning (RL) to learn placement strategies directly from system observations, rather than relying on static heuristics. This approach reduced average function completion time by 23%, demonstrating the efficacy of learning-based schedulers. Complementary work has focused on the control plane. Fuerst et al.[8] introduced Ilúvatar, identifying that scheduling algorithms are irrelevant if the control plane itself introduces significant latency. By using a worker-centric architecture, Ilúvatar reduces overhead to less than 3 milliseconds, providing a high-performance substrate for advanced schedulers.

Gap in Prior Work: While ALPS builds upon

these innovations to bring SRPT scheduling into the kernel, existing evaluations have focused largely on CPU-bound microbenchmarks. Unlike prior work, our study focuses on evaluating how a CPU-centric scheduler behaves under realistic I/O-heavy workloads, addressing a critical blind spot in current design assumptions.

5 Problem Statement & Hypothesis

This section outlines the core intellectual contribution of our study: identifying the dissonance between the design assumptions of ALPS and the reality of I/O-heavy serverless workloads.

5.1 Observation

The primary innovation of ALPS is its ability to learn function behaviors to approximate SRPT scheduling. To achieve this, it relies on a specific signal: historical CPU execution duration. As detailed in the ALPS design, the scheduler predicts the remaining processing time of a task based on how much CPU time it has consumed in previous invocations.

While effective for CPU-bound functions, this metric creates a critical blind spot for I/O-bound workloads. As noted in our proposal, serverless architectures are frequently used for event-driven tasks (e.g., image processing pipelines, database triggers) that spend a significant portion of their lifecycle in a blocked state, waiting for external resources.

5.2 Hypothesis

We hypothesize that ALPS's CPU-centric learning model fundamentally misclassifies I/O-bound functions, leading to pathological scheduling behavior in mixed workloads. Specifically, we posit the following failure mode:

1. **Misclassification:** An I/O-bound function typically executes a short burst of CPU instructions (e.g., formulating a database query) before blocking. ALPS observes this short CPU burst and incorrectly classifies the function as a "short job" (High Priority).
2. **Priority Inversion:** When this function wakes up from I/O, ALPS treats it as a high-priority task because its accrued CPU time is low. It preempts currently running tasks to schedule the I/O-bound function immediately.
3. **Thrashing:** The I/O-bound function runs for another brief moment and blocks again. This cycle repeats, causing frequent, unnecessary context switches (preemptions) that disrupt the execution of actual CPU-bound work without allowing the I/O-bound task to make significant progress.

5.3 Expected Impact

We anticipate that this scheduling misalignment will manifest in three key areas during our evaluation:

1. **Increased Preemptions:** We expect a spike in context switch rates for mixed workloads under ALPS compared to CFS, as the scheduler aggressively chases "short" I/O bursts.
2. **Throughput Degradation:** The overhead of these context switches will likely reduce the overall aggregate throughput of the system.
3. **Diminished Advantage:** While ALPS may still outperform CFS for pure CPU tasks, we predict its advantage will significantly diminish or vanish when I/O-bound functions constitute a major fraction of the workload, potentially performing worse than CFS in highly mixed scenarios.

6 Experimental Methodology

To rigorously evaluate our hypothesis, we designed a set of experiments comparing the performance of ALPS against the default Linux CFS scheduler. Our methodology focuses on reproducing baseline results for CPU-bound tasks and extending the evaluation to I/O-intensive and mixed workloads.

6.1 Testbed & Environment

We conducted all experiments on the CloudLab testbed, ensuring a controlled, reproducible environment free from the noise of multi-tenant public clouds.

- **Hardware:** We utilized the c6420 node type at the Clemson site. The node is equipped with x86_64 Intel Xeon Gold 6142 processors (16 physical cores) and 384 GB of RAM. This high-core-count environment is critical for simulating high-concurrency serverless scenarios.
- **Operating System & Kernel**:
  - **Baseline:** Standard Ubuntu 22.04 LTS running the default Linux kernel (CFS).

– **ALPS Implementation:** Due to deprecation issues with the original ALPS artifact (which required Docker v17.06), we utilized a port of ALPS built on the Google ghOSt (Generic Hosted Operating System Scheduler) framework. ghOSt allows for safe, userspace delegation of scheduling policies, enabling us to implement the ALPS logic without custom kernel patches. We booted these nodes into a custom ghOSt-enabled kernel provided by the ghOSt project.

6.2 Schedulers Compared

We compare two distinct scheduling disciplines:

1. **Linux CFS (Baseline):** The Completely Fair Scheduler, configured with default parameters. This serves as the standard against which improvements and regressions are measured.
2. **ALPS (ghOSt-Backend):** Our reproduction of the ALPS scheduler. It employs the logic described in Section 3: tracking function execution history and utilizing a userspace agent to update eBPF maps that enforce task priorities in the ghost-kernel.

6.3 Workloads

We constructed three distinct workload categories to stress the schedulers under different resource constraints using the Azure function trace that was used in the ALPS [1] paper:

1. **CPU-Bound (Baseline Reproduction):** To validate our ALPS implementation, we used the standard fibonacci function commonly used in serverless benchmarks. This function is computationally intensive and performs no I/O, representing the ”best-case” scenario for ALPS.
2. **I/O-Bound (The Blind Spot):** We created synthetic functions designed to mimic common serverless tasks like database queries or external API calls. These functions execute a short CPU burst (simulating request parsing), sleep for a configurable duration (simulating I/O wait), and then execute another short CPU burst (processing the response).
3. **Mixed Workloads:** To simulate a realistic production environment, we generated workloads containing a mix of CPU-bound and I/O-bound functions arriving concurrently. This scenario tests the scheduler's ability to distinguish between ”true” short jobs and long-running but frequent-blocking I/O jobs.

6.4 Metrics

We quantify performance using the following metrics:

1. **Function Completion Time (FCT):** The wall-clock time from function submission to completion. This is the primary metric for user-perceived latency.
2. **Tail Latency:** We analyze the 99th percentile latencies using Cumulative Distribution Function (CDF) plots. Tail latency is critical in serverless computing, where a single slow function can bottleneck an entire microservice chain.
3. **Throughput:** Measured in requests per second (RPS), this indicates the system's aggregate capacity to handle load under the respective scheduling policies.

7 Implementation Challenges

Reproducing systems research is often as challenging as the initial development, particularly when the work relies on complex interactions between the kernel, container runtimes, and orchestration frameworks. This section details the significant engineering hurdles we encountered while attempting to reproduce the ALPS system and the architectural pivots required to overcome them.

7.1 Reproducing ALPS Artifact Challenges

Our initial approach focused on deploying the original ALPS artifact provided by the authors. However, we immediately encountered a critical version dependency conflict that rendered the artifact unusable on modern testbeds.

- **Docker Version Mismatch:** The ALPS paper specifies the use of Docker v20.10.25. However, the provided GitHub repository and installation scripts are hardcoded for Docker v17.06, a version that is deprecated, no longer supported, and incompatible with current Ubuntu distributions.
- **Broken Patch Chain:** The original ALPS implementation relies on a fragile chain of patches

to propagate metadata (e.g., function IDs) from the OpenLambda worker down to the kernel scheduler. This involved patching OpenLambda to pass data to Docker, which then passed it to containerd, which finally wrote to custom cgroup files. Because Docker's architecture has changed significantly since v17.06, specifically regarding containerd integration and the adoption of cgroups v2, this patching mechanism is fundamentally broken on modern systems.

7.2 Attempted Modernization and Engineering Constraints

Post the above challenges, we attempted to modernize the original Docker-based approach to better understand the system's metadata flow.

- **OCI Hooks vs. Source Patching:** To avoid the brittleness of patching Docker source code, we attempted to implement metadata passing using OCI Runtime Hooks. Our goal was to inject task priority information into the container runtime lifecycle cleanly. However, we discovered that OCI hooks behave inconsistently on Ubuntu 24.04 due to newer systemd and cgroup configurations.
- **OS Downgrade:** To stabilize the environment, we standardized our testbed on Ubuntu 22.04 LTS, which provides a compatible toolchain for experimental kernels while avoiding bleeding-edge incompatibilities. Despite these efforts, the Docker-centric design remained brittle, motivating a more fundamental architectural shift.

7.3 ghOSt-Based ALPS Implementation

Facing these blockers, we contacted the authors of ALPS paper for guidance. They acknowledged the deprecation issues and suggested an alternative path: implementing the ALPS logic using Google ghOSt (Generic Hosted Operating System Scheduler), a framework that delegates scheduling decisions to userspace.

- **Why ghOSt?** ghOSt decouples the scheduling policy from the kernel release cycle. By running the scheduling logic in a userspace agent, we can implement the ALPS "learning" and "SRPT" policies without maintaining a custom fork of the Linux kernel source, drastically simplifying debugging and iteration.
- **Backend Success:** We successfully deployed the ghOSt kernel on CloudLab's c6420 node. After resolving initial boot configuration issues with the CloudLab support team, we confirmed that the kernel is stable and ready to accept userspace policy agents.
- **Frontend Adaptation:** We ported the sliding-window learning mechanism to run within the ghost-userspace agent, allowing it to communicate task priorities directly to the kernel via shared memory segments rather than cgroups.

7.4 Limitations

The shift to ghOSt, while necessary, introduces a slight deviation from the original paper's "native" kernel implementation. While ghOSt is highly efficient, the context switch overhead theoretically includes a userspace round-trip (or shared memory poll), which is distinct from the original in-kernel execution.

8 Evaluation

8.1 Baseline Reproduction (CPU-Only)

As established in our baseline reproduction in Figure 2, ALPS successfully minimizes the average function completion time (FCT) for pure CPU-bound workloads. By correctly identifying shorter tasks based on historical execution profiles, it approximates Shortest Remaining Processing Time (SRPT), yielding a distribution heavily skewed toward lower latencies. This is in-line with the ALPS paper evaluation.

8.2 Mixed Workload Analysis

The CDF comparison (Figure 3) strongly reinforces our hypothesis by visualizing a clear performance "crossover" between task types. The steeper initial rise of the ALPS curve confirms that the scheduler successfully identifies and accelerates short, I/O-bound tasks (< 2s) compared to CFS. However, for long-running tasks (> 100s), the ALPS curve shifts noticeably to the right of CFS, demonstrating that this prioritization strategy directly increases tail latency for the CPU-bound workload. We evaluated a mixed workload consisting of four distinct classes of functions arriving concurrently:

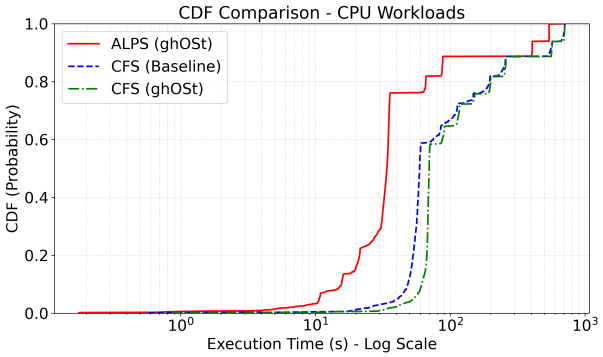

Figure 2: Execution time CDF for ALPS vs CFS vs ghOSt CFS for CPU bound workloads

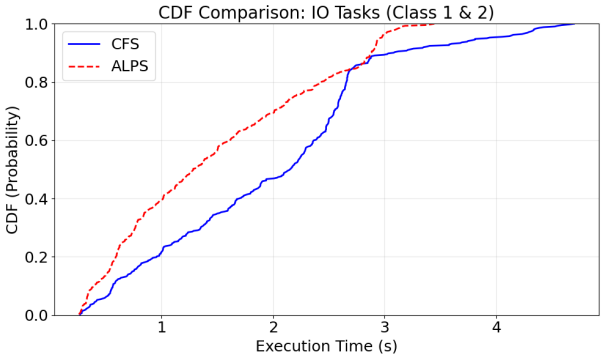

Figure 4: Execution time CDF for IO workloads(class 1 and 2)

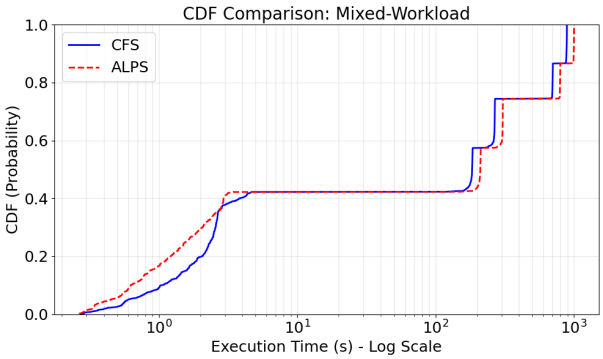

Figure 3: Execution time CDF of mixed workload for all classes

- **Classes 1–2 (I/O-Bound):** increasing in ”size” (work duration), but dominated by I/O waits (sleep calls).
- **Classes 3–4 (CPU-Bound):** increasing in computational complexity (fibonacci with larger arguments), representing ”heavy” processing tasks.

8.2.1 The ”False Short Job” Phenomenon (Classes 1 and 2)

Our data indicates that ALPS prioritizes Classes 1 and 2 aggressively as shown in Figure 4, often processing them faster than the CPU-bound classes.

- **Mechanism:** ALPS tracks CPU usage to determine priority. Since these functions spend the majority of their lifecycle in a sleep or wait state, their accumulated CPU runtime remains negligibly small.
- **Result:** ALPS incorrectly classifies these as ”Short Jobs” (High Priority). Whenever an I/O-bound function wakes up (e.g., I/O completion), ALPS immediately preempts currently running tasks to schedule it.
- **Implication:** While this results in low latency for I/O tasks, it confirms our hypothesis that ALPS cannot distinguish between a truly short job and a long job that barely uses the CPU.

8.2.2 Starvation of Long Jobs (Classes 3 & 4)

The most significant performance degradation is observed in Classes 3 and 4 (the CPU-bound heavy lifters).

- **Priority Inversion:** Because Classes 1 and 2 are treated as high-priority ”short” jobs, they frequently interrupt the execution of Classes 3 and 4.
- **Context Switch Storms:** Unlike a true short job that finishes and leaves the system, Class 1 and 2 functions run for a brief burst, sleep, and wake up again. This creates a cycle of constant preemption.
- **Tail Latency Blowout:** The CPU-bound functions (Classes 3 & 4) are forced to wait not just for other CPU jobs, but for the frequent, high-priority interruptions of the I/O tasks. This manifests as a ”fat tail” in the mixed workload CDF, where heavy jobs take significantly longer to complete than they would in a pure CPU environment (Figure 5).

Class	Type	CFS Avg (ms)	ALPS Avg (ms)	Delta (Avg)	CFS P99 (ms)	ALPS P99 (ms)	Delta (Tail)
1	IO	1,952	1,466	-25%	4,395	3,117	-29%
2	IO	2,104	1,503	-28%	4,560	3,161	-30%
3	CPU	224,770	257,902	+15%	268,485	306,919	+14%
4	CPU	796,348	900,212	+13%	888,502	1,001,569	+13%

Table 1: Scheduler Performance Summary: Average and Tail (P99) Latency by Task Class.

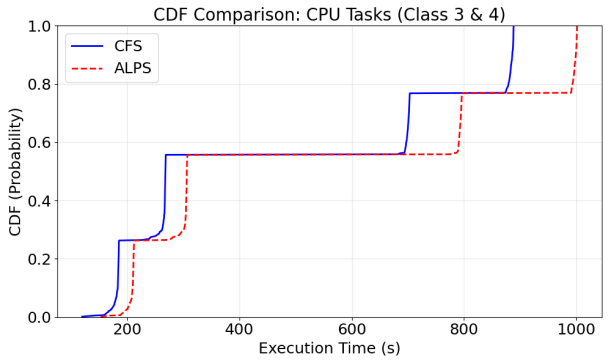

Figure 5: Execution time CDF for CPU workloads (class 3 and 4)

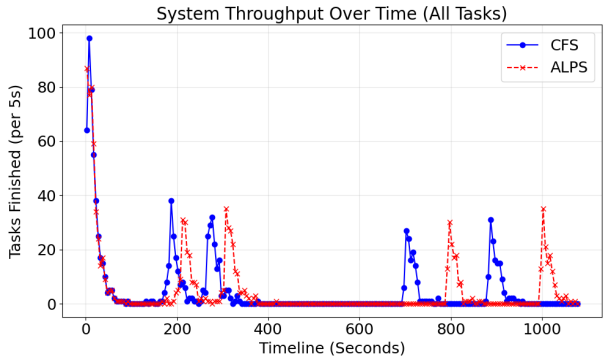

Figure 6: Throughput comparison between CFS and ALPS under mixed workloads

8.2.3 Throughput Observations and Analysis

ALPS exhibits severe starvation for CPU-bound tasks in mixed workloads.

As illustrated in Figure 6, the default Linux CFS (blue line) maintains a consistent and steady rate of task completion throughout the duration of the experiment. This indicates that CFS effectively interleaves the execution of I/O-bound and CPU-bound tasks, ensuring long-term fairness.

In sharp contrast, the ALPS scheduler (red line) demonstrates a distinct "stop-and-go" behavior. We observe prolonged periods of near-zero throughput (specifically between 700s–800s and 890s–1000s) where effectively no tasks complete. These starvation periods are immediately followed by massive spikes in task completion, where the scheduler flushes the backlog of waiting tasks.

This behavioral pattern provides strong empirical confirmation of our core hypothesis. The SRPT-approximation policy employed by ALPS appears to misclassify I/O-bound tasks as "short" jobs because they frequently yield the CPU to wait for I/O.

Consequently, ALPS assigns these tasks the highest priority, aggressively preempting the longer-running CPU-bound tasks. The CPU-bound tasks are effectively starved until the burst of I/O-bound tasks clears, causing the distinct lag in throughput observed in the red line of Figure 6. This starvation introduces significant tail latency for CPU-bound functions, negating the performance benefits ALPS achieves in isolation.

8.2.4 Latency Observation and Analysis

Table 1 quantifies the distinct performance trade-off observed in our mixed-workload experiment. I/O-bound tasks (Classes 1 and 2) benefited significantly from the ALPS policy, seeing a 25–28% reduction in average latency and a 30% improvement in tail (P99) latency compared to CFS. However, this gain came at the direct expense of CPU-bound tasks (Classes 3 and 4), which suffered a performance degradation of approximately 13–15% across both average and tail metrics. These results confirm that while ALPS effectively prioritizes tasks it perceives as "short" (I/O), it does so by starving longer-running CPU computations.

9 Discussion & Future Work

Our findings suggest that the fundamental flaw in ALPS (and similar SRPT-based schedulers) is the definition of "work" as purely CPU cycles con-

sumed. In a serverless environment dominated by microservices and API composition, "work" often involves waiting. A scheduler that ignores wait time is destined to misprioritize I/O-bound chains. The potential solutions are:

- **I/O-Aware Learning:** Future iterations of ALPS could incorporate "Wait Time" into the cost function. If a function historically sleeps for 90% of its lifespan, it should perhaps be deprioritized or scheduled on a separate "I/O Lane" to prevent it from thrashing CPU-bound tasks.
- **Preemption Throttling:** Implementing a minimum time slice (quantum) even for high-priority tasks could prevent the "wake-up storm" caused by I/O-bound functions, reducing context switch overhead.

10 Conclusion

This paper presented an exploratory study of ALPS (Adaptive Learning, Priority Scheduler), a state-of-the-art kernel scheduler designed to optimize serverless function execution. By reproducing its core mechanisms using the Google ghOSt framework, we confirmed that ALPS significantly outperforms the default Linux CFS for CPU-bound microbenchmarks by approximating Shortest Remaining Process Time (SRPT).

However, our extensive evaluation of I/O-bound and mixed workloads exposed a critical limitation in the scheduler's design. We found that ALPS's reliance on historical CPU usage as the primary learning signal creates a "blind spot" for modern, event-driven serverless applications. Our experiments demonstrated that ALPS systematically misclassifies I/O-bound functions as high-priority short jobs, leading to priority inversion and severe starvation of concurrent compute-heavy tasks. This results in a "false efficiency" where I/O tasks are serviced quickly at the expense of overall system throughput and stability.

To address these shortcomings, future research should explore I/O-aware learning signals, incorporating metrics such as "voluntary context switch count" and "average blocking duration" into the priority cost function. Additionally, investigating hybrid scheduling policies that isolate I/O-bound and CPU-bound functions into separate scheduling classes could prevent the cross-talk and thrashing observed in our study. Ultimately, as serverless workloads become increasingly data-centric, the operating system must evolve to view I/O not as a lack of work, but as a different kind of demand.

References

[1] Y. Fu, R. Shi, H. Wang, S. Chen, and Y. Cheng. ALPS: An Adaptive Learning, Priority OS Scheduler for Serverless Function. In *Proceedings of the 2024 USENIX Annual Technical Conference (USENIX ATC 24)*, 2024.

[2] M. Shahrad, R. Fonseca, I. Goiri, et al. Serverless in the Wild: Characterizing and Optimizing the Serverless Workload at a Large Cloud Provider. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, 2020.

[3] Carl A. Waldspurger and William E. Weihl, "Lottery Scheduling: Flexible Proportional-Share Resource Management In *Proceedings of the First USENIX Symposium on OperatingSystems Design and Implementation (OSDI), Monterey, CA, USA, November 1994*. USENIX Association.

[4] A. A. T. Isstaif and R. Mortier, "Towards Latency-Aware Linux Scheduling for Serverless Workloads" In *Proceedings of the 1st Workshop on Serverless Systems, Applications and Methodologies (WoSC5)*, May 2023.

[5] Kostis Kaffes, Neeraja J. Yadwadkar, and Christos Kozyrakis. Hermod: Principled and Practical Scheduling for Serverless Functions. In *Proceedings of the 2022 ACM Symposium on Cloud Computing (SoCC '22), San Francisco, CA, USA, November 7–11, 2022*. ACM.

[6] Yuqi Fu, Li Liu, Haoliang Wang, Yue Cheng, and Songqing Chen. SFS: Smart OS Scheduling for Serverless Functions. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC22), Dallas, Texas, USA, November 13–18, 2022*. IEEE.

[7] H. Yu, A. A. Irissappane, H. Wang and W. J. Lloyd. FaaSRank: Learning to Schedule Functions in Serverless Platforms. In *2021 IEEE International Conference on Autonomic Computing and Self-Organizing Systems (ACSOS), Washington, DC, USA, 2021*, pp. 31-40, doi: 10.1109/ACSOS52086.2021.00023.

[8] Andrea Fuerst, Prateek Sharma, Vaibhav Gogte, and Huaicheng Li. Ilúvatar: A Fast Control Plane for Serverless Computing. In *Proceedings of the 32nd IEEE International Symposium on High-Performance Parallel and Distributed Computing (HPDC 2023), Minneapolis, MN, USA, June 2023*.

[9] Mohsen Ghorbian, Mostafa Ghobaei-Arani, and Leila Esmaeili. A survey on the scheduling mechanisms in serverless computing: A taxonomy, challenges, and trends. In *Cluster Computing*, vol. 27, no. 5, pp. 5571–5610, 2024. Springer.