ALPS: An Adaptive Learning, Priority OS Scheduler for Serverless Function

Yuqi Fu, *University of Virginia;* Ruizhe Shi, *George Mason University;*
Haoliang Wang, *Adobe Research;* Songqing Chen, *George Mason University;*
Yue Cheng, *University of Virginia*

https://www.usenix.org/conference/atc24/presentation/fu

This paper is included in the Proceedings of the
2024 USENIX Annual Technical Conference.

July 10–12, 2024 • Santa Clara, CA, USA

978-1-939133-41-0

Open access to the Proceedings of the
2024 USENIX Annual Technical Conference
is sponsored by

جامعة الملك عبدالله
للعلوم والتقنية
King Abdullah University of
Science and Technology

ALPS: An Adaptive Learning, Priority OS Scheduler for Serverless Functions

Yuqi Fu
University of Virginia

Ruizhe Shi
George Mason University

Haoliang Wang
Adobe Research

Songqing Chen
George Mason University

Yue Cheng
University of Virginia

Abstract

FaaS (Function-as-a-Service) workloads feature unique patterns. Serverless functions are ephemeral, highly concurrent, and bursty, with an execution duration ranging from a few milliseconds to a few seconds. The workload behaviors pose new challenges to kernel scheduling. Linux CFS (Completely Fair Scheduler) is workload-oblivious and optimizes long-term fairness via proportional sharing. CFS neglects the short-term demands of CPU time from short-lived serverless functions, severely impacting the performance of short functions. Preemptive shortest job first—shortest remaining process time (SRPT)—prioritizes shorter functions in order to satisfy their short-term demands of CPU time, and therefore, serves as a best-case baseline for optimizing the turnaround time of short functions. A significant downside of approximating SRPT, however, is that longer functions might be starved.

In this paper, we propose a novel application-aware kernel scheduler, ALPS (Adaptive Learning, Priority Scheduler), based on two key insights. First, approximating SRPT can largely benefit short functions but may inevitably penalize long functions. Second, CFS provides necessary infrastructure support to implement user-defined priority scheduling. To this end, we design ALPS to have a novel, decoupled scheduler frontend and backend architecture, which unifies approximate SRPT and proportional-share scheduling. ALPS' frontend sits in the user space and approximates SRPT-inspired priority scheduling by adaptively learning from an SRPT simulation on recent past workload. ALPS' backend uses eBPF functions hooked to CFS to carry out the continuously learned policies sent from the frontend to inform scheduling decisions in the kernel. This design adds workload intelligence to workload-oblivious OS scheduling while retaining desirable properties of OS schedulers. We evaluate ALPS extensively using two production FaaS workloads (Huawei and Azure) and results show that ALPS achieves a reduction of 57.2% in average function execution duration, compared to CFS.

1 Introduction

Serverless computing, also known as Function-as-a-Service (FaaS), has revolutionized the development and scaling of applications and services. FaaS abstracts away the underlying infrastructure from developers, relieving them of the notoriously tedious tasks of server provisioning and management, and enabling them to focus predominantly on code and application logic. FaaS solutions are becoming increasingly popular and are commonly found in both commercial clouds (e.g., AWS Lambda [3], Azure Functions [4], Google Cloud Functions [13]) and open source projects (e.g., OpenWhisk [25], OpenFaaS [23], etc.).

The majority of FaaS use cases demonstrate highly concurrent and bursty workloads, typically invoked through lightweight HTTP requests [1,51]. In this context, FaaS applications can generate a large volume of requests within a short time period [40,45–47,83]. Moreover, the execution time of a serverless function is typically short and highly variable—ranging from a few milliseconds to a few minutes [61,77]. An analysis of the Azure Functions trace datasets shows that about 37.3%, 57.2%, and 99.9% of the function requests have an average execution duration[1] shorter than 300 ms, 1 second, and 224 seconds, respectively [77].

Serverless functions are ultimately scheduled and executed by a host OS. Serverless functions typically have a small CPU-memory footprint [61,77], making FaaS workloads increasingly consolidated. It is not uncommon to pack tens of thousands of, if not more, function instances onto a single host [39,44]. While statistical multiplexing [53,72,75] makes it possible for a FaaS provider to achieve high workload throughput, the short-lived nature of serverless functions makes them extremely sensitive to resource contentions caused by FaaS clouds' deep consolidation.

The heterogeneity of FaaS workloads poses new challenges to existing kernel scheduling solutions. Linux's default CPU scheduler, the Completely Fair Scheduler [6] commonly used by most cloud providers including AWS Lambda [3, 39], is a general-purpose and proportional-share kernel scheduler, which performs poorly for short-task-dominant workloads [54]. Specifically, to achieve fairness (so-called "proportional share" or "fair share"), CFS squeezes the time slice for each competing function and proportionally shares the physical CPU time among them. All functions, no matter short or long, experience the same expected slowdown. This achieves fairness [42], but at the same time, also leads to frequent context switches for short functions, causing significantly prolonged "scheduling cycles": a function that has used up its time slice is descheduled and must wait for a long time before it gets rescheduled. For short functions,

[1] A function's *execution duration* measures the time when a function starts execution till the time when the function finishes execution and returns, the same definition as the *turnaround time* metric. We use both terms interchangeably in the paper.

this prolonged waiting time hurts their turnaround time: they could have completed much earlier without being preempted if given a slightly longer time slice instead. Ultimately, kernel schedulers' workload obliviousness severely impacts FaaS providers's quality-of-service (QoS) and FaaS applications' performance and cost.

Designing and implementing a new, application-aware kernel scheduler is notoriously hard [58,67,70]. On the one hand, OS facilities are implemented using low-level programming languages (typically C and assembly) and developers cannot use advanced libraries and debugging tools for development and testing. On the other hand, it is extremely hard to cover all possible workload cases during development/testing phase.

A representative user-defined kernel scheduler for serverless functions is SFS [54]. SFS dynamically steers existing scheduling policies in Linux (FIFO and CFS) from the user space based on workload history to enable more effective function scheduling. SFS prioritizes short functions that can finish within a configurable time slice. However, since SFS works entirely in the user space, it lacks transparency, flexibility, and efficiency compared to a kernel scheduler solution. (1) SFS requires non-trivial modifications of the applications (i.e., the FaaS platforms) in order to use its user-space scheduling interface. (2) SFS has to rely on user-space tools [28, 29, 36] and/or "hacky" methods to control kernel activities like context switch and task priority. This approach is indirect and often coarse-grained. (3) SFS incurs non-trivial user-space scheduling overhead; for example, SFS needs to constantly poll the kernel task status in order to make appropriate decisions from the user space.

In order to overcome the aforementioned limitations, a new, application-aware kernel scheduler is required. First, a new scheduler should not enforce FaaS workloads to use statically baked policy that is workload-agnostic. Rather, it should be application-aware and can dynamically adapt its policy based on workload dynamics. Second, this new kernel scheduler should be transparent to the user-space applications. That is, it should require no or minimum modifications of existing FaaS platforms and serverless functions. Third, this new scheduler should retain desirable OS scheduler properties such as high efficiency, low overhead, and flexible preemption.

To address all these requirements, we propose ALPS (Adaptive Learning, Priority Scheduler), a new application-aware kernel scheduler for FaaS workloads. ALPS takes a fundamentally different approach for user-defined kernel scheduling. At its core, ALPS adaptively learns the statistical clairvoyance of functions' time slice distribution and priorities by applying SRPT (Shortest Remaining Processing Time) to past function traces and uses the continuously learned policies to inform future scheduling decisions in the kernel.

We design ALPS to have a novel scheduler architecture that decouples scheduler frontend and backend. ALPS' frontend sits in the user space and approximates SRPT-inspired priority scheduling by adaptively learning two separate yet correlated policies (a time slice policy and a task ordering policy) for each individual function from an SRPT simulation running on recent past workload. ALPS' backend uses eBPF functions [10] hooked to CFS to carry out learned policies sent from the frontend to inform scheduling decisions within the kernel space. ALPS adds workload intelligence to workload-oblivious OS scheduling: for short functions, ALPS learns to extend the time slice so that short functions run to completion with minimal context switches; for long functions, ALPS reuses CFS to avoid starvation; moreover, ALPS retains desirable properties of kernel schedulers (work conserving, low overhead) by carrying out decision making in the kernel.

In summary, this paper makes the following contributions:

- We explore the design space of approximate SRPT in kernel scheduling and identify a practical and effective method to abstract approximate SRPT scheduling into two easily solvable, lightweight sub-policies.
- We design ALPS, an intelligent, application-aware kernel scheduler that learns from SRPT behaviors on past workload from the user space to inform function scheduling decisions in the kernel.
- We evaluate ALPS extensively using production FaaS workload traces. Results show that ALPS outperforms Linux CFS and a state-of-the-art serverless OS scheduler SFS by 57.2% and 20.6% in terms of average function execution duration while mitigating the long tail latency issue suffered by the offline SRPT.

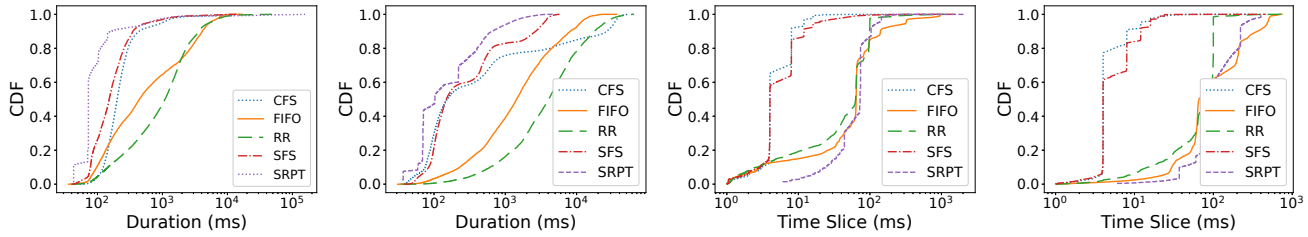
2 Background

2.1 Overview of FaaS

A FaaS platform allows users to build and deploy function applications as `.zip` archives [8] or custom container images [20]. To deploy or update a function, a user pushes the function's container image to a centralized container registry. A deployed function can be invoked via an invocation request, e.g., through an HTTP URL or from an event source such as a cloud object store. Each invoked function runs in a sandbox environment (e.g., a container [9, 14] or a VM [39]). Without loss of generality, we assume containers as the underlying sandbox technique for function execution and isolation. Upon an invocation request, a serverless function is ultimately scheduled and executed by a function worker in the underlying host OS. Kernel scheduling therefore determines the "last-mile" efficiency of function invocation requests.

2.2 Completely Fair Scheduler

Completely Fair Scheduler, or CFS, is Linux's default kernel scheduler. CFS is a general-purpose, proportional-share scheduler with the goal of fairly balancing the CPU resource usage among all CPU tasks. CFS divides the physical time into fine-grained time slices among all CPU tasks in proportion to their weights (i.e., priorities). To achieve fairness among all tasks, CFS uses the virtual runtime scheme: CFS keeps track of a task's CPU time—the time that this task has



(a) Duration CDF (Huawei). **(b)** Duration CDF (Azure). **(c)** Time slice CDF (Huawei). **(d)** Time slice CDF (Azure).

Figure 1: Performance CDF of the Huawei (a) and Azure (b) traces and corresponding time slice distribution, Huawei (c) and Azure (d).

be running on a CPU—weighted by its priority, as `vruntime`, and orders tasks by `vruntime` in a per-CPU-based, red-black (RB) tree priority queue structure called `runqueue`. At each scheduling tick, CFS picks the next runnable task that has the smallest `vruntime` from the `runqueue`. When a new task is first admitted to the system, it will be initialized with the smallest `vruntime` value, and therefore, it will be placed to the left-most of the RB tree so that it can be immediately scheduled to run to avoid starvation for new tasks. The length of the time slice is determined by the number of CPU tasks in the `runqueue`. As the FaaS workload is increasingly consolidated, it is common to have thousands of concurrently running functions that multiplex the limited CPU resources on the host machine. Thus, under CPU contention, all concurrent function requests colocated on the same host experience the same expected slowdown, hence CFS achieves fairness.

2.3 eBPF Primer

eBPF (extended Berkeley Package Filter) is a new kernel technique that makes it easy to extend the capabilities of the Linux kernel [10]. eBPF enables developers to dynamically load user-defined functions into the kernel without requiring to change kernel source code or load kernel modules. User-defined functions can be added to various kernel hook points (e.g., the network interface). A user-defined function is executed when the attached hook point is triggered. eBPF enables user and kernel communication via eBPF maps, which provide a generic in-memory storage of different types of sharing data between the user space and kernel space. Linux eBPF is used for networking [15, 16, 26, 69], load balancing [7, 19, 37], platform observability [21, 34, 38], resource monitoring and profiling [2, 5, 30], and network security [18, 31, 32]. eBPF serves as an effective tool for bringing FaaS workload knowledge to kernel scheduling. eBPF functions can be placed at various places within CFS to manipulate the scheduling decision making of CFS. eBPF maps offer an efficient communication channel to share the application-level knowledge with the kernel scheduler.

3 Motivation

We measured the performance of two FaaS workloads sampled from Huawei Cloud Functions [61] and Azure Functions [77] on an open-source FaaS platform OpenLambda [57]. OpenLambda was configured to run on 24 CPU cores in a Ubuntu 22.04 OS. We tested three Linux kernel schedulers: `SCHED_FIFO` (first in, first out), `SCHED_RR` (round-robin), and `SCHED_NORMAL` (Linux CFS), as well as a state-of-the-art serverless function scheduler SFS [54] implemented using our eBPF framework (§7). We also simulated an offline oracle SRPT for comparison. We tuned the inter-arrival times (IATs) of the functions so that the average CPU utilization of both the Huawei and Azure workloads was around 90%. See details about the workload generation and function grouping in §7.1.

3.1 Scheduler Performance

Figure 1a and 1b show the duration time distribution of the Huawei and Azure workload traces, respectively. We make several observations from these figures.

First, Linux's two existing schedulers, namely, FIFO and RR, suffer a significant performance gap compared to the offline SRPT for most of the function requests in both the Huawei and Azure workloads. CFS shortened the gap, but still saw a big performance difference from SRPT. For example, 89.2% of function requests ran slower under CFS than SRPT, due to lack of workload awareness.

Second, SFS sits in the middle between SRPT and CFS in terms of function execution duration due to its workload awareness. SFS adopts a two-level scheduler design with a level-one FIFO queue and a level-two CFS queue. At the FIFO queue level, SFS schedule functions in the order they are admitted to the system and preempts and demotes them to CFS if they do not finish in a configurable time slice. This FIFO time slice is dynamically tuned by SFS based on historical workload information and is applied to all functions in the current time window. However, SFS' coarse-grained time slice design is sub-optimal (compared to SRPT, which has the knowledge of the future), and therefore, caused significant delays for most of the function requests.

Third, although SRPT shows a clear advantage over other online schedulers for relatively short functions, SRPT suffered a significant long-tail delay at the 98.9^{th} percentile and above for the Huawei workload. This is because the Huawei workload is heavily dominant by short functions (see §7), with 69.7% and 1% of function requests falling into Group 1 (shortest functions) and Group 4 (longest functions), respec-

tively. In contrast, the Azure workload is more evenly spread across all four function groups, as shown in Table 1.

Observation

- *Linux's existing schedulers achieve considerably poorer overall performance compared to the offline SRPT due to workload obliviousness.*
- *SFS takes the FaaS workload pattern into account but its coarse-grained decision making leads to sub-optimal performance.*
- *While SRPT is provably optimal for average performance, its strategy, which prioritizes shorter function jobs, penalizes the longer function jobs. This is particularly evident in real-world FaaS workloads that are heavily skewed towards short functions.*

3.2 Analysis

To better understand the performance gaps between online schedulers and offline SRPT, we profiled the time slice information and the number of context switches of all the baselines that we tested, as shown in Figure 1c and 1d. Interestingly, despite the huge performance differences between FIFO/RR and SRPT, these three policies exhibit similar trends in time slice distribution. The reason for SRPT's performance superiority is apparent: SRPT makes perfect prediction about the remaining time of a function job and intelligently prioritizes jobs with the shortest remaining time. Though FIFO and RR may allow a short function to run to completion without a context switch (see Figure 7), they enforce functions to execute by the order they enter the system, thereby increasing the likelihood of head-of-line blocking.

On the other hand, CFS and SFS show dramatically different time slice patterns compared to the other set of schedulers. Under CFS, a CPU task's priority and its allotted time slice are determined by different factors. CFS controls tasks' priorities using a configuration parameter called `weight`, which can be set by a *nice* parameter passed from the user space [22]. In the kernel, a task's `weight` determines the growth rate of its `vruntime`. That is, a task with a higher weight will have a `vruntime` with a slower increasing rate. Recall that CFS ranks all CPU tasks within an RB-tree-based `runqueue` by their `vruntime` (§2.2) and the rank of a task on the `runqueue` roughly determines when it will be rescheduled to execute in the near future. The higher the weight of a task, the sooner it will get rescheduled, and the longer CPU time it will receive in the long run. The time slice that each task receives is roughly the same and is determined by the total number of tasks in the system.

Instead of explicitly determining when (priority) and how long (time slice) a task should run during each scheduling cycle, CFS implicitly controls, via task weight, the proportion of CPU time that each task is expected to receive in a given scheduling cycle. In the context of highly consolidated FaaS workloads, the function execution order becomes virtually

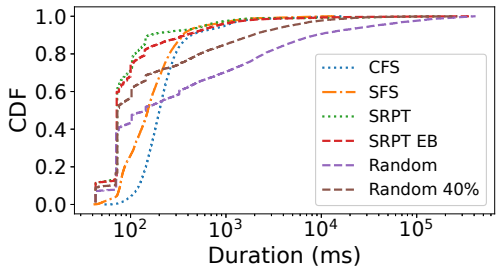


Figure 2: Execution duration CDF of the Huawei workload.

random and all running functions, whether short or long, get equally penalized with the same expected slowdown ratio, provided that all functions are assigned the same weight.

Observation and Implication

- *CFS lacks the tool to explicitly control task order and time slices for individual tasks, simply because CFS is not designed for optimizing the turnaround time but rather for achieving so-called proportional-share fairness.*
- *However, the proportional-share nature of CFS does not negate the feasibility of turning CFS into a priority scheduler that approximates the behavior of SRPT. In fact, CFS offers the ideal facility to support priority scheduling: the task execution order is determined by a parameter (`vruntime`), which can be manipulated to prioritize shorter functions.*
- *CFS' time slice mechanism can be dynamically adjusted based on function behavior so as to minimize the context switches for shorter functions.*

3.3 Approximating SRPT

One approach to approximating SRPT is to use heuristics. For example, approximate and deployable SRPT (ADS) uses multiple FIFO queues and dispatches requests to one of the FIFO queues based on the (hinted) request size (e.g., message size) [84]. While the message size can serve as a hint accurate enough to predict the task processing time in the context of network scheduling, it is challenging to predict the execution duration of a serverless function. We tested several "approximate" variants of SRPT that make prediction mistakes about the execution duration of a function: (1) SRPT EB: this SRPT variant makes error-bounded (EB) mistakes, where the estimated execution duration of a function is the actual execution duration (obtained via profiling under perfectly isolated environment) plus an error bounded $\pm$50% of its actual execution duration. (2) SRPT RANDOM 40%: this SRPT variant makes random mistakes for 40% of the function requests, whose predicted execution duration was generated randomly. (3) RANDOM: an online scheduler that predicts function execution time using a random number generator.

We used the same Huawei trace to drive the tests and the result is shown in Figure 2. Surprisingly, both SRPT EB and SRPT RANDOM 40% achieved reasonably good perform-

nce: they both managed to improve short functions' medium turnaround time by 51.2% compared to SFS, though both of them mispredicted to some extent.

Observation and Implication

This result suggests that a predictive or learned priority scheduler can achieve near-optimal performance for most functions if one of the following relaxed conditions are satisfied:

- *All predictions are inaccurate but are error-bounded, and they collectively and approximately follow the statistical behavior of SRPT like SRPT EB did.*
- *A portion of predictions are accurate while the others are completely random guesses like SRPT RANDOM 40% did.*

4 Design Goals

§3 suggests that approximating SRPT may improve average turnaround time for FaaS workloads, especially for short functions, which constitute a large portion of real-world workloads. Yet, SPRT causes significant long-tail delay for long functions, especially for the Huawei workload that is highly skewed. Therefore, our goal is to design an application-aware kernel scheduler that optimizes the average turnaround time by approximating SRPT while striking a balance to offer a worst-case guarantee for long functions. To achieve this goal, we ask two questions: (1) What is the mechanism of the new scheduler, i.e., how the scheduling machinery works? (2) What is the scheduling policy, i.e., when a function should execute and how long it should run in each scheduling cycle?

Designing a new kernel scheduler is never easy, considering the tools a developer can use for development and debugging a kernel scheduler and the too many corner cases that the new scheduler needs to cover. CFS, as a battle-tested kernel scheduling solution, is well-proven for balancing the CPU resource usage and avoiding starvation for long tasks. More importantly, CFS as a facility can be augmented in order to support user-defined priority scheduling policies.

4.1 Scheduler Mechanism

The key insight of this paper, as revealed from §3, is that Linux CFS can be adapted to find a middle ground that combines the best of both priority scheduling for short functions and proportional-share scheduling for long functions. To this end, this paper takes a different route and directly builds the application-aware scheduling policy atop the CFS infrastructure. Recall in §3.2 our key implication is that CFS' task ordering and time slice policy can be manipulated to realize new priority scheduling. This offers a tool to design new scheduler abstraction that abstracts and modularizes the new priority scheduling policy into two small sub-problems, each of which can be separately solved:

- **Task ordering policy** determines how functions should be ordered, and for a function, how it can be prioritized based on its (predicted) future need: in our case, the likelihood of how soon or how far in the future it would finish. This policy directly affects the time a function is waiting in the `runqueue`, and consequently, impacts the turnaround time.
- **Time slice policy** determines how long a function is allowed to execute in a scheduling cycle. This policy also has a direct impact on the turnaround time of a function.

The key enabler of the new scheduler mechanism is eBPF. The modularized sub-policies can then be hooked into CFS as eBPF functions. We describe the detailed design in §5.3.

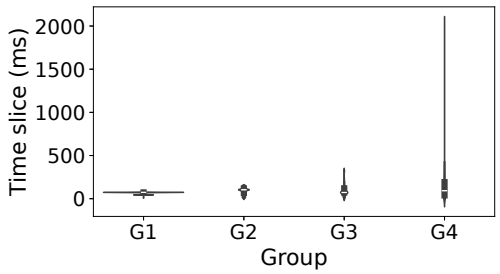


Figure 3: SRPT's time slice distribution of different function groups.

4.2 Scheduling Policy: Learning from SRPT

Instead of directly learning a model to predict function execution duration, we use a different approach to approximate SRPT. We break the approximate SRPT policy into two sub-problems (§5.2.1) and learn simple policies from the SRPT behavior on historical workload to solve the sub-problems. We next discuss the design choices we make.

- **Learning a task ordering policy:** We observe that production FaaS workloads have a mix of short and long functions. As such, functions can be categorized into different groups based on their execution duration. This design choice significantly simplifies the design of the learning rule as it is much easier for a model to accurately predict the function groups rather than how long a function will run or how much remaining time a function is left.
- **Learning a time slice policy:** Under SRPT, a short function is likely to have the highest absolute priority among all actively running functions at the time when it arrives, and therefore, it is highly likely to run to completion without being preempted (unless a shorter function arrives right after that particular function starts execution). On the other hand, a function with relatively long execution duration is likely to be frequently preempted during its whole lifespan, and the preemption frequency is determined by the number of shorter functions that arrive during the execution of that long function. This can be observed in Figure 3 where longer functions (Group 3 and 4) show much more variable time slice distribution than that of shorter functions (Group 1 and 2). An interesting SRPT behavior–the time slice pattern–can be learned or approximated, because, statistically, the collective time slice pattern of short functions is highly predictable (Figure 3). Given this observation and hypothesis, to learn from the past behavior of SRPT, if with the SRPT policy a function gets assigned a particular time

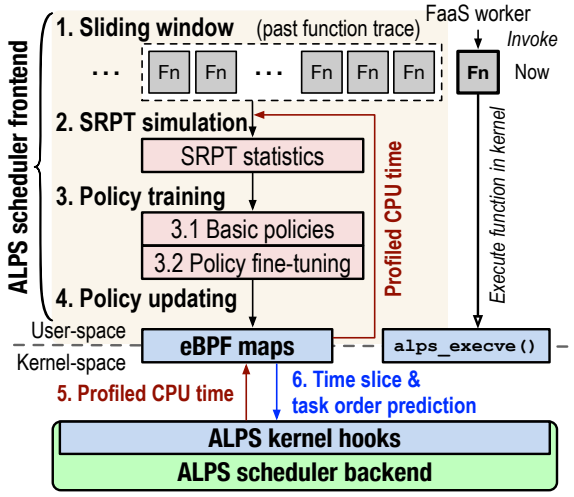


Figure 4: Overview of the ALPS scheduler architecture.

slice, then in the near future, the same function should be assigned with the same or similar time slice.

Our new scheduler thus consists of two critical components. The first, serving as a user-space frontend, reconstructs and learns SRPT's behavior from the past FaaS workloads. The second, serving as a kernel-space backend, is a predictor that steers CFS to inform scheduling decisions for future workloads by the same functions.

5 ALPS Design

5.1 Design Overview

This section provides a high-level overview of the ALPS scheduler design. Figure 4 illustrates the overall ALPS architecture. ALPS consists of a scheduler frontend and a scheduler backend. ALPS' frontend sits in the user space, collects past function traces, performs SRPT simulations on the collected past workload, adaptively trains the scheduling policies using a sliding window, periodically updates the policies deployed in the kernel, and finally, obtains kernel statistics using eBPF maps and feed them to the frontend for next-round simulation. ALPS' backend sits in the kernel space and performs scheduling decision making based on the policies trained from the user-space frontend. ALPS introduces a new Linux system call `alps_execve()` to differentiate serverless function tasks against all other tasks running in the same host. `alps_execve()` is the only system call for a FaaS platform to use ALPS. We discuss how to port a FaaS platform (in our case, OpenLambda [35, 57]) to use ALPS in §6.

Scheduling Workflow. Next, we describe the scheduling workflow of ALPS as illustrated in Figure 4.

1. The frontend uses a sliding window to collect past function traces. The frontend collects detailed function request information including the following: *<function_UID, arrival_time, termination_time, CPU_time>*. The length of the sliding window is dynamically determined by a configurable time period parameter W. ALPS depletes the sliding window and starts refilling every W seconds. The function trace dataset collected during the past W seconds is then used for SRPT training and simulation.
2. The frontend performs SRPT simulation using the function trace collected from the sliding window in the last W seconds. Application-experienced function execution duration (i.e., end-to-end turnaround time) fails to measure the true CPU resource consumption in an ideal environment without CPU contention. Therefore, ALPS uses eBPF to profile the ideal CPU time for each function UID and feeds the profiled statistics to the frontend for SRPT simulation (see Step 5). The simulation generates SRPT behavior statistics including the segmented waiting times (a function task might be preempted) and the time slice of each function request.

3.1 ALPS trains basic policies in the frontend using the statistics data generated by the SRPT simulation. This step is described in detail in §5.2.1.

3.2 Real-world FaaS workloads have uncertainties including mispredictions (variable function execution duration) and workload shifts (dynamically changing load). To tackle these issues, ALPS uses several strategies to fine-tune the basic policies trained from the previous step We discuss detailed fine-tuning strategies in §5.2.2.

4. ALPS deploys the learned policies in the kernel-space backend via the eBPF maps.
5. ALPS uses eBPF to profile function executions and store the function's ideal CPU time[2] into the eBPF maps. The frontend periodically fetches functions' ideal CPU time for next-round SRPT simulation (Step 2).
6. In the kernel space, the ALPS backend queries the eBPF maps to inform scheduling decisions from two CFS hook points defined in two CFS functions: `entity_before()` and `__schedule()`. We describe ALPS' eBPF hooks in §5.3.

ALPS's decoupled frontend/backend design provides an important benefit. The frontend can directly utilize the observability and/or monitoring service of a FaaS platform [17, 24, 33, 57] to fetch fine-grained historical FaaS workload information. Thus, *the frontend is essentially a look-aside OS utility, which is a part of the "blackbox" kernel scheduler but functions in the user space.* Unlike other user-defined kernel schedulers such as ghOSt [58] and Syrup [63], ALPS' frontend does not need to be embedded into the application, which requires application modification to explicitly intercept requests. This design makes ALPS a transparent scheduler, which can be flexibly applied to other FaaS platforms.

5.2 ALPS Frontend

5.2.1 Basic Learning Policies

This subsection presents how ALPS learns basic policies based on the output of SRPT simulation.

[2]The ideal CPU time is defined as the function turnaround time minus the waiting time, excluding the initial queuing delay and fragmented waiting times caused by context switches.

Learning a Task Ordering Policy. SRPT prioritizes tasks with shorter remaining time. While it is seemingly straightforward to use predicted (remaining) execution time as the task ordering priority, the predicted execution time is not able to statistically capture the relative ranks of functions in the entire workload if all function tasks, including both active and inactive, are accounted. We observe that, under SRPT, those tasks that experience shorter waiting time are the ones with shorter execution time. Those tasks are typically positioned more near the front rather than the end of the task queue. Those tasks with shorter waiting time, if submitted at a different time, are still highly likely to be ranked high among all active tasks that arrive roughly at the same time. Therefore, the metric of average waiting time can be used as an effective approximation of a function's relative rank with respect to all other functions in the workload.

The training step is straightforward. ALPS calculates the average waiting time $\bar{wt}_i$ for each function i from the simulation results. ALPS then sorts all the functions that were executed in the previous sliding window by $\overline{wt_i}$ and assigns the rank values from a predefined rank range. For instance, a range of $(1\ldots100)$ means that the host is currently serving function requests from a total of 100 unique function deployments, where each function deployment has a UID, regardless of how many function requests have been received in the previous sliding window. The lower the rank value, the higher priority it has. Since calculated function ranks are relative values drawn from the same rank range, different functions appeared at different sliding windows might be assigned to same rank value. In this case, ALPS' kernel backend breaks the tie using the `vruntime` field of these two function tasks (§5.3).

Learning a Time Slice Policy. After completing the SRPT simulation, ALPS collects the time slice statistics and groups them by function UIDs to produce a time slice vector $\vec{ts}_i$ for each function i. ALPS then computes an approximate, basic time slice value for each function i:

$$ts_i = train\_ts(\vec{ts}_i) \tag{1}$$

where $train\_ts()$ represents one of the four lightweight predictive models: *average*, *linear regression*, *random forest*, or *exponentially weighted moving average (EWMA)*.

The phenomenon that SRPT tends to penalize long tasks is also evident in the time slice distribution generated by SRPT simulations. By analyzing the SRPT behavior, we make the following observations. First, short functions typically exhibit a more predictable and stable time slice distribution than long functions, because short functions have drastically less number of context switches than long functions under SRPT. Second, long functions have a much wider range in their time slice distribution compared to short ones. On one side, long functions are more frequently preempted by shorter functions, leading to many small, fragmented time slices. Conversely, there are cases where some long functions may experience extremely long time slices. This could happen in a situation where several long functions with similar execution duration are running concurrently and SRPT chooses to execute the relatively shorter functions first, causing head-of-line blocking for the other long functions in the queue. We next discuss how ALPS addresses this unpredictability issue using a strategy that we call fine-tuning.

5.2.2 Time Slice Fine-tuning

ALPS fine-tunes the trained, basic time slice values using heuristics in two sub-steps. The first sub-step adjusts the basic time slices to mitigate the impact of misprediction. The second sub-step considers the load of the system and applies a penalty factor to all actively running functions when the system is overloaded.

Tackling Unpredictability. As mentioned, trained time slices may mispredict. Mispredictions fall into two categories: underestimation and overestimation, both may cause performance degradation. If, for example, the predicted time slice is just a few ms shorter than the actual remaining execution time, i.e., the function is not able to complete before the trained time slice elapses, then this function would be preempted and needs to wait for a new scheduling cycle before it gets rescheduled to complete the rest few ms of execution time. Otherwise, an unnecessarily long predicted time slice will cause extended queuing delay for all other functions that are queued in the same `runqueue`. The mitigation heuristic is defined as:

$$ts_i^u = \max(\alpha \times ts_i - \beta \times stdev(\overline{ts_i}), 0) \tag{2}$$

where ts_i is the trained time slice of function i (Equation (1)) and $\vec{ts}_i$ is the vector that contains all fragmented time slice values of function i collected from the previous simulation. The first term of the left argument of *max* applies a coefficient α as a reward factor to extend the basic time slice in order to alleviate the negative effect of underestimation. The second term of the left argument of *max* applies a coefficient β as a penalty factor to penalize functions with a high variation of historical time slices. This penalty term also helps mitigate execution time variance caused by the function input size (we will discuss this more in §9). We apply *max* to safely set the time slice value ≥ 0 (a value of 0 means the function will follow CFS' default time slice policy).

Tackling System Overload. SRPT simulation makes simplified assumptions and therefore has limitations. For example, the sliding-window-based SRPT simulation is not able to well capture the dynamic workload shifts, e.g., a sudden load increase caused by a burst of concurrent function requests [51]. The SRPT simulation does not assume the cost of a context switch in Linux, therefore neglecting the impact of such cost under overloaded circumstances. To tackle this issue, ALPS introduces a global penalty factor p that is stacked on the unpredictability fine-tuning adjustment (Equation (2)) by $ts_i^s = \min(p \times ts_i^u, ts_i^u)$. The function *min* is to make sure that the fine-tuned policy is at least less than or equal to the basic

```
1  /* Definition of the time slice eBPF function */
2  struct time_slice_map;
3  SEC("sched/bpf_time_slice")
4  int BPF_PROG(__schedule, struct task_struct *prev,
5          sched_entity *se_prev){
6          s64 upper_bound = bpf_map_lookup_elem(
7                      &time_slice_map, prev->func_id);
8          s64 delta = s->sum_exec_runtime
9                  - s->prev_sum_exec_runtime;
10         if (upper_bound && delta < upper_bound){
11              /* needs to extend current task's execution */
12             return 1;
13         }
14         return 0
15 }
```

Listing 1: Definition of the time slice eBPF function. Note that further detail of our implementation (declaration, configuration, etc.) is omitted for readability.

```
6265 /* Linux kernel v5.18-rc5 {kernel/sched/core.c} */
6266 void __sched notrace __schedule (unsigned int sched_mode){
6267    /* prev: the current task */
6268    struct task_struct *prev, *next;
6269    struct sched_entity *se_prev = &prev->se;
6270    ret = bpf_time_slice(prev, se_prev); /* Calling eBPF
         func */
6271    if (se_prev->on_rq && ret){
6272         next = prev; /* ALPS branch */
6273    } else {  /* CFS branch */
6274         next = pick_next_task(rq, prev, &rf);
6275    }
6276    clear_tsk_need_resched(prev);
6277    clear_prempt_need_resched();
6278 }
```

Listing 2: Hook point of the time slice eBPF function in Linux.

policy value. The penalty factor p is defined as:

$$p = \begin{cases} 1 & \text{if } l < \theta \\ \frac{100+\theta-l}{100\times\gamma} & \text{if } l \geq \theta \end{cases}, \tag{3}$$

where l denotes the real-time CPU utilization, θ denotes the CPU utilization threshold to toggle penalty, and γ denotes the "reaction" to the high load. A lower γ results in a higher penalty factor p. The higher the value of p, the more closely the behavior of ALPS resembles that of CFS. We evaluate the effectiveness of the fine-tuning strategies in §7.4 and the sensitivity of the parameters in §7.5.

5.3 ALPS Backend

eBPF Maps and System Call. ALPS' frontend learns two policy values for each function UID: an `int`-typed rank value that represents the relative order of a particular function (§5.2.1), and an `s64`-typed time slice value that estimates a "soft" upper bound of the average time slice that this particular function should run during each scheduling cycle (§5.2.1 and §5.2.2). ALPS uses two eBPF maps, namely `task_order_map` and `time_slice_map`, to store and share the learned policy values that are indexed by function UIDs between the frontend and the backend. ALPS introduces a new Linux system call API `alps_execve()`, which is modified based on Linux's

```
1  /* Definition of the task ordering eBPF function */
2  SEC("sched/bpf_task_ordering")
3  int BPF_PROG(struct sched_entity *a, struct sched_entity
       *b){
4      int rank_a = bpf_map_lookup_element(&task_order_map,
5      container_of(a, struct task_struct, se)->func_id);
6      int rank_b = bpf_map_lookup_element(&task_order_map,
7      container_of(b, struct task_struct, se)->func_id);
8      if (rank_a == rank_b){ /* CFS branch */
9          return (s64)(a->vruntime - b->vruntime) < 0;
10     } else { /* ALPS branch */
11         return (s64)(pri_a - pri_b) < 0;
12     }
13 }
```

Listing 3: Definition of the task ordering eBPF function.

```
568 /* Linux kernel v5.18-rc5 {kernel/sched/fair.c} */
569 bool entity_before(struct sched_entity *a, struct
        sched_entity *b){
570     return bpf_task_ordering(a, b); /* Calling eBPF func */
571 }
```

Listing 4: Hook point of the task ordering eBPF function in Linux.

original `execve` [11] system call. `alps_execve()` passes a new argument `func_id` and stores it in a new field `func_id` within the kernel data structure `task_struct`. This enables ALPS' backend eBPF function to identify learned function policies by the function UID from within the kernel. ALPS' frontend updates the eBPF maps through an eBPF helper function `bpf_map_update_elem` once every training loop (from Step 1 to Step 4 in Figure 4). ALPS' backend queries the learned policies stored in the eBPF maps for function scheduling in the kernel space. This subsection describes ALPS' backend eBPF functions that implement the time slice policy and task ordering policy.

Time Slice eBPF Function. The `__schedule()` function is the core logic that implements CFS task scheduling and context switch. Upon an interrupt (e.g., a timer interrupt) or an explicit blocking event (e.g., mutex or semaphore), `__schedule()` is called to pick the next task to run. ALPS hooks an eBPF function called `bpf_time_slice()` (Listing 1 and 2) in line 6270 within `__schedule()` to determine whether to extend the time slice of the current task `prev`. Specifically, `bpf_time_slice()` reads the time slice value into a variable called `upper_bound` associated with the function UID (line 6 and line 7). Then it checks if the CPU time that this function has consumed within the current scheduling tick (`delta`) is still shorter than the learned time slice (line 10). If so, `bpf_time_slice()` returns 1 and extends the current time slice (line 6,272). Otherwise, if the learned time slice already expires, or if the eBPF map does not find a populated entry corresponding to this particular function UID (in which case `bpf_map_lookup_elem()` return 0, and this function has not been trained by the frontend), `bpf_time_slice()` returns 0. If this is the case, ALPS falls back to the CFS branch, as shown in line 6,274.

Task Ordering eBPF Function. The kernel function `entity_before` determines the rank of the CPU tasks in the `runqueue` by `vruntime` (§2.2). ALPS overrides this logic by us-

ing an eBPF function `bpf_task_ordering()` (Listing 3 and 4) `bpf_task_ordering()` fetches the learned task ranks from the mapping data structure `task_order_map` then uses the ranks to determine the function task order in the `runqueue` (line 11). Note that if the rank information is missing from the eBPF map (in which case `bpf_map_lookup_elem()` return 0, the highest rank in ALPS's ranking system), or if both functions have the exact same rank value, ALPS falls to the CFS branch (line 9).

6 Implementation

We have implemented ALPS on Linux (`v5.18-rc5`) and ported ALPS to OpenLambda [35,57], an open-source FaaS platform written in Go. OpenLambda supports two types of sandboxes, the SOCK container [71] and the Docker container [9]. Our implementation of ALPS supports Docker-based serverless functions. We modified Docker (`v20.10.25`) to execute functions under ALPS, enabling seamless integration with other FaaS platforms that use Docker containers. Specifically, we made 135 lines of code (LoC) modifications in OpenLambda and 223 LoC changes in Docker. ALPS can be easily ported to other FaaS platforms with minimal engineering effort.

We made two modifications to OpenLambda in order for it to use ALPS for function scheduling. (1) We modified OpenLambda's JSON configuration schema by adding a new field for function UID so that the OpenLambda worker can pass the function UID to the underlying function sandbox layer (the Docker containers). (2) We modified Docker (spanning Docker client, containerd, and runc) to use ALPS' new system call `alps_execve()` (§5.3) to execute and schedule serverless functions using ALPS in the kernel. The ALPS frontend was implemented using Python and is a generic module that can be used as a drop-in module in order to support any open-source FaaS platforms.

7 Evaluation

7.1 Experimental Methodology

Experimental Setup. We deploy ALPS and OpenLambda on a bare-mental machine with 56 CPUs and 256 GB memory running Linux Ubuntu 22.04.1 LTS. The bare-metal machine provides us exclusive access to the hardware to avoid the impact of random multi-tenant interference.

Function Applications. We implement a serverless function benchmark suite that includes the following six different serverless function applications based on the serverless benchmark [48,66,88]: *Fibonacci Sequence (fib)*, *Breath First Search (bfs)*, *Minimum Spanning Tree (mst)*, *Page Rank (pr)*, *Feature Extraction (fe)*, and *Float Operation (fo)*. We use different parameters to control the execution duration of each function application.

Workload Generation. We generate FaaS workloads from publicly available FaaS workload traces collected from Huawei Cloud Functions [61] and Azure Functions [77]. The Huawei trace datasets contain invocation requests spanning 200 unique function applications collected in a period of 141 days, while the Azure trace datasets feature invocation requests from 82,375 unique function applications spanning a period of 14 days. To downscale, we sampled a total of 15,000 invocation requests from Day 1's workloads in both datasets and Table 1 shows the execution duration distribution across four ranges (in ms): (0,50], (50,200], (200,400] and [400,∞) as four function groups. These two production workloads show distinct behaviors. The Huawei workload exhibits a highly skewed distribution towards short functions, with over 93% of sampled function requests running 200 ms or less. In contrast, the sampled Azure workload has a more balanced distribution compared to the Huawei one. The distribution skew results in a higher workload throughput in terms of requests per second (RPS) when compared to the Azure one: the average RPS of our sample workload is 190 and 133 for Huawei and Azure, respectively.

Table 1: Probability distribution of function execution duration ranges among the four function groups for the two traces.

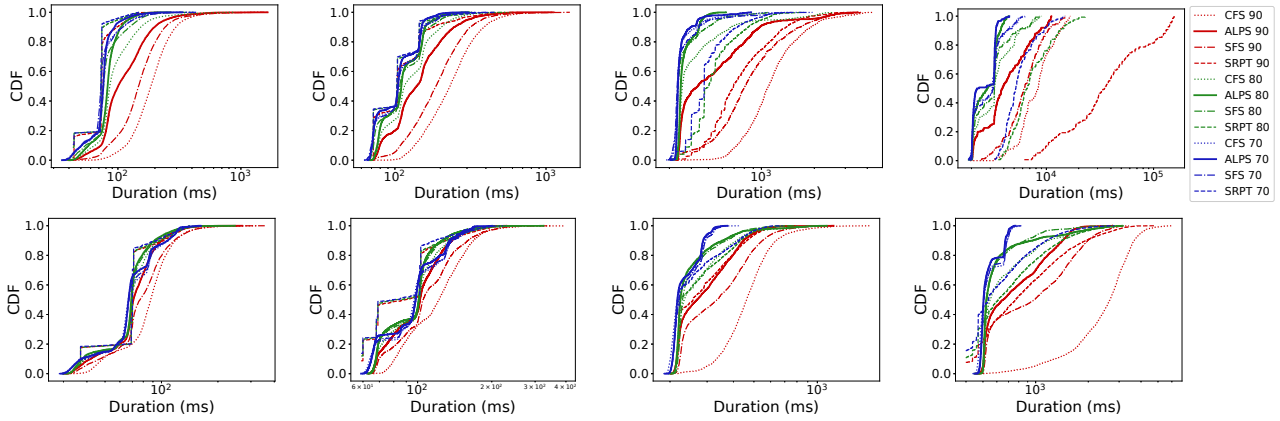
Group	Duration Range	Huawei	Azure
1	0-50 ms	69.8%	42.9%
2	50-200 ms	23.2%	17.2%
3	200-400 ms	6%	16.7%
4	$\geq$ 400 ms	1%	23.2%

We created 50 function deployments based on the six function applications from our benchmark suite by varying the function parameters. We then mapped the sampled invocation requests to these function deployments. The execution duration of a serverless function may have variance due to different resource configurations [50] and/or variable input sizes [86]. Our analysis of the Huawei and Azure traces confirmed this: the majority of functions in both workloads have a variance of up to 25% of the average execution duration. To add variance to the function execution duration for a particular function deployment, we randomly pick a parameter from its configured parameter range. For example, the execution duration of a function deployment named *mst1* on a fully connected graph with 5-10 vertices (the number of vertices is the parameter for mst) varies from 20-40 ms, therefore we map <*mst1*, $[5,10]$> to Group 1 (Table 1) and randomly select a parameter from the $[5,10]$ parameter range when the workload generator issues an invocation request targeting *mst1*.

We configure the inter-arrival times (IATs) of requests to follow the Poisson distribution. We then scaled up or down the IAT configuration to vary the load level of the generated workloads from 70% to 90% of average CPU utilization. In our experiments, by default, $\alpha = 2$, $\beta = 1$, $\gamma = 1$, $\theta = 50$ and sliding window is set to 5s.

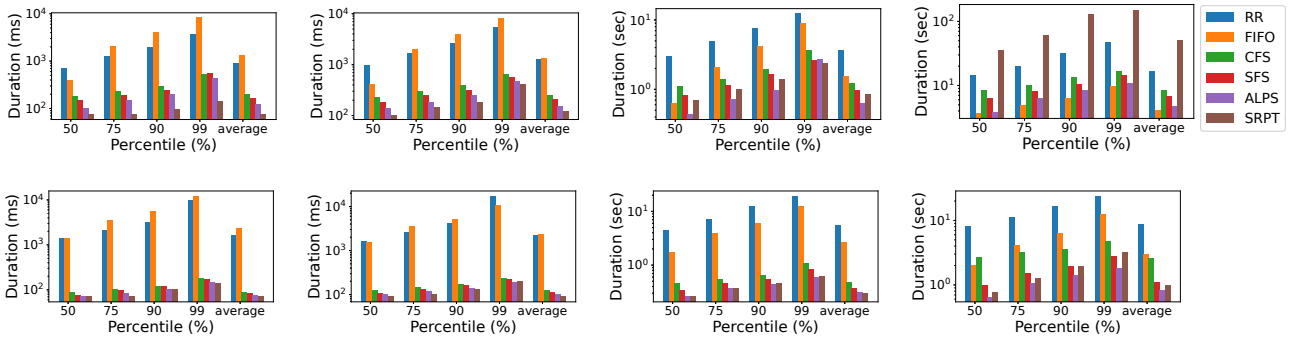
Goals. Our evaluation aims to answer the following questions:

- How does ALPS perform compared with different schedulers under various load levels (§7.2 and §7.6)?
- How does platform-level overhead affect the performance improvement contributed by ALPS (§7.3)?



(a) Group 1 **(b)** Group 2 **(c)** Group 3 **(d)** Group 4

Figure 5: Performance CDF of the Huawei (top) and Azure (bottom) sampled workloads under different load levels.



(a) Group 1 **(b)** Group 2 **(c)** Group 3 **(d)** Group 4

Figure 6: Percentile breakdowns of function execution duration of the Huawei (top) and Azure (bottom) sampled workloads at 90% average CPU utilization.

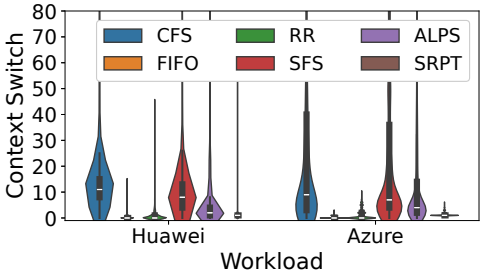


Figure 7: Comparison of number of context switches.

- Is ALPS' time slice fine-tuning strategy effective (§7.4)?
- How do ALPS' different parameter configurations affect its performance (§7.5)?
- What is ALPS' frontend and backend cost (§7.7)?

7.2 End-to-End Performance

We first compare the end-to-end workload performance of ALPS against baseline schedulers under different load levels. Figure 5 reports the CDFs of the function execution duration across four function groups. *CFS 90* represents the system using CFS as the scheduling policy under 90% CPU utilization. We can see that ALPS outperforms both CFS and SFS for all four groups under all three load levels. The performance gap between ALPS and SFS/CFS becomes wider from Group 1 to Group 4 as well as from low load to high load. For example, ALPS achieves a 99^{th} percentile tail execution duration of 198 ms under the 70% CPU utilization for Group 1 given the Huawei workload, which is 10.8% and 15.1% shorter than that of SFS and CFS, respectively. SFS' medium execution duration, under the 90% CPU utilization is 1.1× and 1.5× higher than that of ALPS under Group 1 and Group 4, respectively, for the Azure workload (Figures 5 and 6). Again, for the Azure workload, 70.1% of Group 4's function requests finish within 1 second, while only 50.9% and 5% of requests finish within 1 second under SFS and CFS, respectively.

Overall, ALPS achieves an improvement of average execution duration that is 20.6% to 27.3% lower than SFS and 34.1% to 57.2% lower than CFS, respectively (Figure 6). This is also evident from ALPS' reduction on the number of context switches when compared to SFS and CFS, as shown in Figure 7. ALPS has significantly better performance than CFS as ALPS uses workload-aware learning to inform scheduling decisions. ALPS outperforms SFS because, rather than pre-

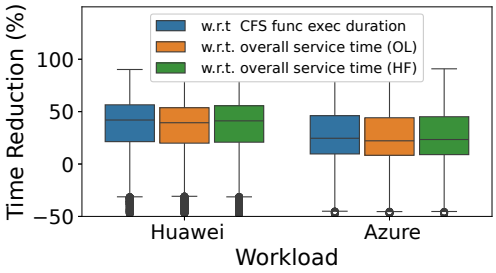


Figure 8: Time reduction contributed by ALPS w.r.t. the function execution duration achieved by CFS, Openlambda's overall service time (OL), and Huawei Functions' overall service time (HF).

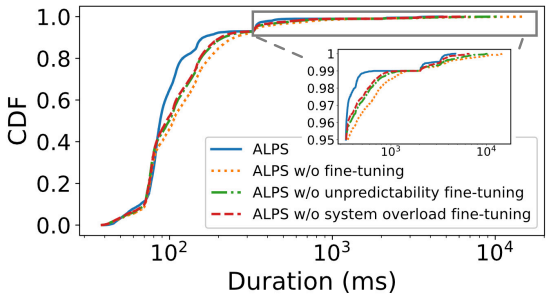


Figure 9: Ablation test of unpredictability and system overload adjustment.

dicting a coarse-grained time slice and applies it blindly to all functions within the current time window, ALPS learns the behaviors of individual functions and makes fine-grained scheduling decisions at per-function level.

The performance superiority of short functions under SRPT does not come for free: there is a tradeoff in balancing the scheduler efficiency for both short and long jobs [85]. SRPT outperforms ALPS only for Group 1 and 2 for the shortest function requests. This trend reverses in favor of ALPS for the other two groups under the Huawei workload. For Group 3, with functions completing in the range of 200-400 ms, 73.3% of the ALPS requests had a shorter execution duration compared to SPRT. SRPT saw a 1.5× higher 99^{th} percentile tail execution duration than that of ALPS (Figure 6d top). The performance benefits of ALPS come from the design that unifies SRPT learning and proportional sharing.

7.3 Time Reduction w.r.t. Overall Service Time

FaaS platforms incur platform-level overhead, including platform-level scheduling and network delay [61]. The overall function service time includes both the platform-level overhead and the function execution duration. The Huawei Functions workload study reveals that platform-level cost accounts for only 1-10% of the function execution duration, with a medium platform-level overhead of 1 ms and more than 95% of all requests having a platform delay less than 10 ms [61]. Our measurement of OpenLambda indicates that OpenLambda has an average platform-level overhead of 11 ms, which is less efficient than Huawei. We used 11 ms as the platform delay for OpenLambda. To simulate Huawei Functions' platform delay, we randomly drew a value from the 1–10 ms range for each function request. We observe from Figure 8 that ALPS' contribution to time reduction holds even when the platform-level cost is considered. Specifically, for the Huawei workload, the medium time reduction decreases slightly from 41.9% to 39.4% and 41.2% for the OpenLambda and Huawei Functions platform, respectively. Consequently, reducing the last-mile function execution duration can greatly reduce the end-to-end service time.

7.4 Ablation Study

In this set of experiments, we perform an ablation study to assess the contribution of the individual time slice fine-tuning heuristics (§5.2.2). We tested the following configurations: *Config (1)*: ALPS with both fine-tuning heuristics enabled, *Config (2)*: ALPS without fine-tuning, *Config (3)*: ALPS with unpredictability fine-tuning enabled, and *Config (4)*: ALPS with system overload fine-tuning enabled. Figure 9 plots the results. We find that, overall, ALPS with both heuristics enabled achieves the best performance, with an average execution duration reduction of 28.9% compared to *Config (2)*, demonstrating the effectiveness of our fine-tuning heuristics. The improvement is significant, especially for the tail latency: with both heuristics enabled, ALPS reduces the 99^{th} percentile latency by 51.6% and 43.3% compared to ALPS with *Config (3)* and *Config (4)*, respectively. This is because our fine-tuning strategies made more conservative time slice allocation decisions in the face of unpredictable worst-case scenarios such as long functions' volatile time slice distribution and sudden request bursts. The conservative scheduling decisions effectively mitigated long function starvation in a way similar to what a proportional-share scheduler would do.

7.5 Sensitivity Analysis

Next, we conduct a sensitivity analysis along the following dimensions.

Impact of Learning Methods. We first test the impact of different learning methods, average (or Avg, ALPS' default method), linear regression (LR), random forest (LF), and EWMA, on overall function performance. As shown in Figure 10, varying the learning methods does not have a huge impact on ALPS' overall performance, as long as the learning method can capture the magnitude of a function's average time slice under SRPT. Among the four methods, EWMA observed a higher tail latency than the other three. This is because EWMA is biased towards more recent workload history, thus causing high variance in time slice prediction, especially under non-stationary workload behavior (e.g., IAT shifts).

Impact of Sliding Window Size. We then test the impact of the sliding window size on ALPS' performance. We vary the sliding window size from 5 to 30 seconds and results in Figure 11 indicate that a smaller sliding window might yield better scheduling decisions. For example, a sliding window of 30 seconds resulted in an average execution duration of 148 ms, which is 12.7% higher than a sliding window of 5

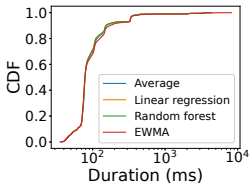


Figure 10: Impact of learning methods.

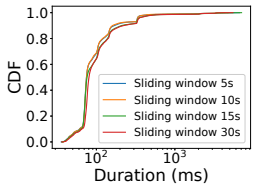


Figure 11: Impact of sliding window size.

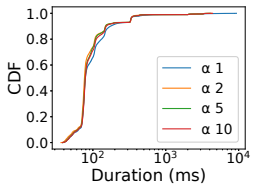


Figure 12: Impact of α.

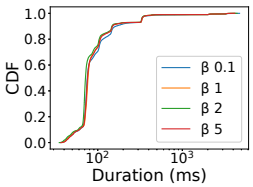


Figure 13: Impact of β.

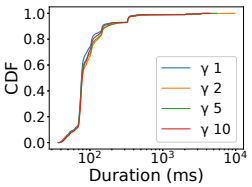


Figure 14: Impact of γ.

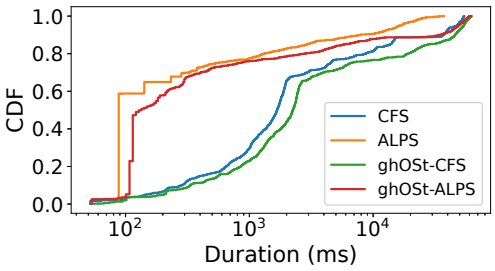


Figure 15: Comparing with ghOSt-ALPS.

seconds. This suggests that basing scheduling decisions on most recent temporal behavior, as opposed to learning from a longer workload history, helps improve the performance of FaaS workloads.

Impact of Fine-Tuning Parameters. We analyze the impact of fine-tuning parameters on function performance. We use a fixed θ as 50 in Equation (3) because we observed that all schedulers perform similarly under a load lower than an average CPU utilization of 50%. We therefore only tested the sensitivity of the workload performance on different values of α, β for Equation (2) and γ for Equation (3). Figure 12 suggests that none of the four α configurations exhibits a clear advantage over other values. Among these α values, $\alpha = 1$ shows a slightly worse performance because an underestimation of time slice by a small α may introduce more (unnecessary) context switches for short functions. Similarly, varying the configuration of β and γ does not significantly affect the overall performance. With regard to β performance shown in Figure 13, overall, a larger β tends to achieve a slightly better performance, as a large value of β poses a greater penalty that can better mitigate the head-of-line blocking caused by unpredictable time slices of long functions. For *gamma*, as shown in Figure 14, among all four configurations, setting γ as 1 yields the best performance.

7.6 Comparing against ghOSt

ghOSt [58] is a Linux scheduling framework that delegates kernel scheduling decisions to user-space applications. ghOSt allows user-space application to make scheduling decisions as transactions through a user-space agent and send them to the kernel scheduler via message queues. To demonstrate the effectiveness of the ALPS scheduling algorithm, we implement ALPS in C++ on ghOSt, since ghOSt only supports C++ applications. We also implement a Fibonacci Sequence function application and used the sampled Azure workload trace to drive the experiment. We measure the workload performance of vanilla ALPS against a ghOSt-based ALPS. For performance comparison, we also test the same workload using vanilla CFS and a (simplified) ghOSt-based CFS [12]. We tested the schedulers under a high load with an average CPU utilization of 90%.

Table 2: ALPS' frontend cost of SRPT simulation and policy learning (in second).

Methods	100	200	500	1,000	2,000	5,000
Avg	0.005	0.006	0.007	0.008	0.01	0.013
LR	0.009	0.009	0.01	0.011	0.012	0.016
RF	0.079	0.08	0.083	0.087	0.094	0.127
EWMV	0.006	0.006	0.007	0.007	0.009	0.014
Simulation	0.007	0.015	0.039	0.079	0.161	0.406

We make several observations from the results shown in Figure 15. (1) ALPS ported to ghOST is effective in improving the function performance compared to both versions of CFS. (2) The ghOSt framework itself introduces non-trivial runtime overhead for serverless function scheduling. For the top 50% shortest function requests, ghOSt-ALPS saw an average execution duration of 111.4 ms, which is 25.3% higher than that of vanilla ALPS. The reason behind this is that ghOSt's user-space scheduling agent needs to constantly communicate with the kernel scheduler via transaction messages. Since serverless functions are ephemeral and ALPS needs to frequently adjust the function time slices for many concurrent short-lived functions at the same time, this generates a large number of scheduling transactions that need to be completed within a short amount of time, a non-trivial overhead of more than 20% for short functions. ALPS, on the other hand, makes scheduling decisions directly in the kernel by querying policies learned from the user space, therefore avoiding the message communication and transaction cost.

7.7 ALPS Cost

Table 2 reports ALPS' frontend cost. With an input size of 100 function request records (ALPS' default configuration), calculating the average time slice and SPRT simulation takes an average of 5 ms and 7 ms, respectively. Even when scaling the input size to 5,000, the simulation takes less than a second to finish. The frontend costs are negligible with respect to the default sliding window size of 5 seconds.

ALPS' backend cost mainly comes from two eBPF hook points: `bpf_time_slice()` and `bpf_task_ordering()`. Table 3 summarizes the cost associated with these eBPF hook points and other operations. The average cost of `bpf_time_slice()`

Table 3: ALPS' backend cost. Item 5 and 6 are for reference.

1. ghOSt local scheduling (1 per sched)	1,613 ns
ALPS backend	
2. `bpf_task_ordering()` (0.46 per sched)	66 ns
3. `bpf_time_slice()` (1 per sched)	125 ns
4. ALPS overall backend cost (per sched)	155 ns
5. Syscall cost	459 ns
6. CFS process context switch cost	3,512 ns

and `bpf_task_ordering()` is 66 ns and 125 ns, respectively. The `bpf_time_slice()` hook is invoked once with each scheduling decision made by the kernel's `__schedule` function, whereas `bpf_task_ordering()` is sporadically called during a context switch. Profiling reveals that a single context switch triggers an average of 8.01 scheduling decisions, with `bpf_task_ordering()` being invoked 3.69 times per context switch. Therefore, the average invocation frequency of `bpf_task_ordering()` is 0.46 times per scheduling decision. This combined overhead of 155.36 ns is significantly lower than ghOSt's local scheduling overhead (1,613 ns per scheduling decision: the message delivery to local agent cost of 725 ns plus the local scheduling transaction cost of 888 ns [58]).

8 Related Work

Serverless Function Scheduling. CFS-LLF [59] modifies CFS to grant higher priority to long-tail, least loaded functions. CFS-LLF assumes function sandbox processes (Kubernetes pods) are long-running, therefore can exploit PELT [27] to track function process load. ALPS assumes each function creates an ephemeral process, thus demanding a completely different solution. A body of research is focused on distributed platform-level function scheduling [64, 65, 79–81, 87]. Hermod [65] proposes execution-time-agnostic early-binding and processor-sharing load balancing strategies for platform function scheduling. These solutions would benefit from effective OS scheduling by ALPS.

Approximating SRPT. Improving turnaround time by approximating SRPT is a known approach that has been investigated in many domains [52, 55, 56, 76, 78]. A series of systems use request sizes as the hint to approximate SRPT. Size-based scheduling gives preference to requests for small files targeting web servers serving static HTTP requests [56]. Similarly, Harchol-Balter et al. applied SRPT to webserver request scheduling based on sizes of Linux kernel socket buffers [55]. Inspired by these works, ALPS presents a practical kernel scheduler that unifies approximate SRPT scheduling and proportional-share scheduling to address new challenges in emerging, real-world FaaS workloads.

eBPF Augmentation. Plugsched [67] proposes an efficient live update mechanism for Linux schedulers in datacenter workloads. Plugsched targets a different dimension of scheduler adaptability (less-frequent, coarse-grained, datacenter-wide code update), while ALPS focuses on much finer-grained job-level scheduling policy adaptability for FaaS. SPRIGHT [74] uses eBPF to enable efficient shared-memory processing for serverless functions.

9 Discussion

Fairness. Proportional-share CPU schedulers like Linux's CFS and Lottery Scheduling [82] were designed to optimize long-term fairness by evenly allocating CPU time across all (presumably long-running) jobs. These schedulers roughly follow Jain's fairness index [60], which measures throughput variability across users. However, they may not provide the desired fairness to short-lived jobs, as transient jobs add randomness to scheduling [41]. A more effective fairness metric for short-job-dominant workloads with heterogeneous CPU demand remains an open research problem. CFS tends to penalize all jobs equally, leading to disproportionately long waiting times for short functions due to its focus on long-running jobs. To address this, ALPS dynamically adjusts the waiting time based on the execution duration. Additionally, ALPS maintains fairness for long-running functions by reusing CFS, addressing SRPT's starvation issue (§5). Our profiling shows that ALPS minimizes the average waiting-time to CPU-time ratio for short functions, with a ratio of 0.96% and 12.94% for functions in Group 1 and Group 3, respectively. In contrast, CFS' ratio is significantly higher at 4.04% and 63.13%.

Impact of Function Input Size. Serverless function execution durations are variable and are proportional to variable input sizes [43, 68, 77]. Researchers propose highly efficient mechanisms for preemption, context switch, and core scheduling policies to optimize microsecond-level datacenter network requests with variable service time [49, 62, 73]. These works may not be directly applicable to FaaS as FaaS workloads exhibit much higher execution time variance than microsecond-level datacenter requests. ALPS takes into account the impact of input size variability in two ways. (1) ALPS considers the standard deviation of the historical execution duration and applies a penalty factor to functions with high execution duration variance (§5.2.2). (2) The experimental workload generator introduces variance into the function execution duration to reflect real-world workload behaviors (Table 1).

10 Conclusion

ALPS is a new kernel scheduler design that is radically different from existing kernel scheduling solutions. ALPS continuously learns FaaS workload intelligence from an offline oracle scheduler SRPT in the user space and deploys the learned policies into the kernel using eBPF to inform function scheduling decisions. We have built a prototype of ALPS, comprising a user-space frontend that learns from SRPT and a kernel-space backend that hooks eBPF functions to Linux CFS. Extensive evaluation shows that ALPS improves the performance for both short functions and long functions compared to CFS and state-of-the-art application-aware schedulers (SFS and ghOSt). ALPS is open-sourced and is available at:

https://github.com/ds2-lab/ALPS.

Acknowledgments

We are grateful to our shepherd, Ashraf Mahgoub, and the anonymous reviewers for their valuable feedback and suggestions. This work was sponsored in part by NSF grants: CNS-2322860 (an NSF CAREER Award and its CloudBank AWS credit), CCF-2318628, CNS-2007153, CCF-1919113, and was generously supported by an Adobe Research gift.

References

[1] 2018 Serverless Community Survey: huge growth in serverless usage. https://serverless.com/blog/2018-serverless-community-survey-huge-growth-usage/.

[2] Android uses eBPF to monitor network usage, power, and memory profiling. https://source.android.com/docs/core/architecture/kernel/bpf.

[3] AWS Lambda. https://aws.amazon.com/lambda/.

[4] Azure Functions. https://azure.microsoft.com/en-us/services/functions/#overview.

[5] BPFAgent: eBPF for Monitoring at DoorDash. https://doordash.engineering/2023/08/15/bpfagent-ebpf-for-monitoring-at-doordash/.

[6] CFS scheduler. https://docs.kernel.org/scheduler/sched-design-CFS.html.

[7] Cilium Standalone Layer 4 Load Balancer XDP. https://cilium.io/blog/2022/04/12/cilium-standalone-L4LB-XDP/.

[8] Deploy Lambda functions with .zip file archives. https://docs.aws.amazon.com/lambda/latest/dg/python-package.html.

[9] Docker: Accelerated Container Application Development. https://www.docker.com/.

[10] eBPF. https://ebpf.io/.

[11] execve(2) — Linux manual page. https://man7.org/linux/man-pages/man2/execve.2.html.

[12] ghOSt CFS Agent. https://github.com/google/ghost-userspace/tree/main/schedulers/cfs.

[13] Google Cloud Functions. https://cloud.google.com/functions.

[14] gVisor: The Container Security Platform. https://gvisor.dev/.

[15] How Does Alibaba Cloud Build High-Performance Cloud-Native Pod Networks in Production Environments? https://www.alibabacloud.com/blog/how-does-alibaba-cloud-build-high-performance-cloud-native-pod-networks-in-production-environments_596590.

[16] How Wildlife Studios built a Global Multi Cluster Gaming Infrastructure with Cilium. https://cilium.io/blog/2020/09/03/wildlife-studios-multi-cluster-gaming-platform/.

[17] IBM Cloud OpenWhisk: Viewing Logs. https://github.com/ibm-cloud-docs/openwhisk/blob/master/logs.md.

[18] Introducing Datadog Network Performance Monitoring. https://www.datadoghq.com/blog/network-performance-monitoring/.

[19] Introducing Walmart's L3AF Project: How do we use eBPF to provide network visibility in a multi-cloud environment. https://medium.com/walmartglobaltech/introducing-walmarts-l3af-project-how-do-we-use-ebpf-to-provide-network-visibility-in-a-8b9ae4d26200.

[20] New for AWS Lambda – Container Image Support. https://aws.amazon.com/blogs/aws/new-for-aws-lambda-container-image-support/.

[21] Next-Generation Observability with eBPF. https://isovalent.com/blog/post/next-generation-observability-with-ebpf/.

[22] nice(2) — Linux manual page. https://man7.org/linux/man-pages/man2/nice.2.html.

[23] OpenFaaS. https://github.com/openfaas/faas.

[24] OpenLambda Stats. https://github.com/open-lambda/open-lambda/blob/main/src/common/stats.go.

[25] OpenWhisk. https://openwhisk.apache.org/.

[26] Panel Disucssion: Is There Actually a Byte Behind All the Buzz? eBPF in Production! https://www.youtube.com/watch?v=qmrHONqsV2M.

[27] Per-entity load tracking. https://lwn.net/Articles/531853/.

[28] perf-sched. https://man7.org/linux/man-pages/man1/perf-sched.1.html.

[29] psutil (process and system utilities) documentation. https://psutil.readthedocs.io/en/latest/.

[30] Rate limiting access to internal services in a virtual network – Nick Bouliane, DigitalOcean. https://www.youtube.com/watch?v=gcHxfhDT-I4.

[31] Replacing HTB with EDT and BPF. https://netdevconf.info//0x14/session.html?talk-replacing-HTB-with-EDT-and-BPF.

[32] Securing the IoT with eBPF and Rust - Giovanni Alberto Falcione. https://www.youtube.com/watch?v=vmRQXRit-sY&list=PLDg_GiBbAx-mAwr1P5H4xFEn7X-qWZ2V_&index=17.

[33] Serverless Framework Logs. https://www.serverless.com/framework/docs/providers/aws/cli-reference/logs.

[34] Skyfall: eBPF agent for infrastructure observability. https://engineering.linkedin.com/blog/2022/skyfall--ebpf-agent-for-infrastructure-observability.

[35] The OpemLambda GitHub repository. https://github.com/open-lambda.

[36] Ubuntu Mannuals: schedtool. https://wiki.xenproject.org/wiki/Credit_Scheduler.

[37] User Story - How Trip.com uses Cilium. https://cilium.io/blog/2020/02/05/how-trip-com-uses-cilium/.

[38] Using BPF Iterators to Gain Insight into Kubernetes - Alban Crequy, Microsoft. https://www.youtube.com/watch?v=ilcYXPDSgu8&list=PLj6h78yzYM2Pm5nF_GmNQHMyt9CUZr2uQ&index=6.

[39] Alexandru Agache, Marc Brooker, Alexandra Iordache, Anthony Liguori, Rolf Neugebauer, Phil Piwonka, and Diana-Maria Popa. Firecracker: Lightweight virtualization for serverless applications. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, pages 419–434, Santa Clara, CA, February 2020. USENIX Association.

[40] Lixiang Ao, Liz Izhikevich, Geoffrey M. Voelker, and George Porter. Sprocket: A serverless video processing framework. In *Proceedings of the ACM Symposium on Cloud Computing*, SoCC '18, page 263–274, New York, NY, USA, 2018. Association for Computing Machinery.

[41] Remzi H Arpaci-Dusseau and Andrea C Arpaci-Dusseau. *Operating systems: Three easy pieces (Chapter 9, Figure 9.2)*. Arpaci-Dusseau Books, LLC, 2018.

[42] Nikhil Bansal and Mor Harchol-Balter. Analysis of srpt scheduling: Investigating unfairness. In *Proceedings of the 2001 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '01, page 279–290, New York, NY, USA, 2001. Association for Computing Machinery.

[43] Vivek M Bhasi, Jashwant Raj Gunasekaran, Aakash Sharma, Mahmut Taylan Kandemir, and Chita Das. Cypress: Input size-sensitive container provisioning and request scheduling for serverless platforms. In *Proceedings of the 13th Symposium on Cloud Computing*, pages 257–272, 2022.

[44] James Cadden, Thomas Unger, Yara Awad, Han Dong, Orran Krieger, and Jonathan Appavoo. Seuss: Skip redundant paths to make serverless fast. In *Proceedings of the Fifteenth European Conference on Computer Systems*, EuroSys '20, New York, NY, USA, 2020. Association for Computing Machinery.

[45] Benjamin Carver, Runzhou Han, Jingyuan Zhang, Mai Zheng, and Yue Cheng. λFS: A Scalable and Elastic Distributed File System Metadata Service using Serverless Functions. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 4*, ASPLOS '23, page 394–411, New York, NY, USA, 2024. Association for Computing Machinery.

[46] Benjamin Carver, Jingyuan Zhang, Ao Wang, Ali Anwar, Panruo Wu, and Yue Cheng. Wukong: a scalable and locality-enhanced framework for serverless parallel computing. In *Proceedings of the 11th ACM Symposium on Cloud Computing*, SoCC '20, page 1–15, New York, NY, USA, 2020. Association for Computing Machinery.

[47] Benjamin Carver, Jingyuan Zhang, Ao Wang, and Yue Cheng. In search of a fast and efficient serverless dag engine. In *2019 IEEE/ACM Fourth International Parallel Data Systems Workshop (PDSW)*, pages 1–10, Los Alamitos, CA, USA, nov 2019. IEEE Computer Society.

[48] Marcin Copik, Grzegorz Kwasniewski, Maciej Besta, Michal Podstawski, and Torsten Hoefler. Sebs: A serverless benchmark suite for function-as-a-service computing. In *Proceedings of the 22nd International Middleware Conference*, Middleware '21, page 64–78, New York, NY, USA, 2021. Association for Computing Machinery.

[49] Henri Maxime Demoulin, Joshua Fried, Isaac Pedisich, Marios Kogias, Boon Thau Loo, Linh Thi Xuan Phan, and Irene Zhang. When idling is ideal: Optimizing tail-latency for heavy-tailed datacenter workloads with perséphone. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, SOSP '21, page 621–637, New York, NY, USA, 2021. Association for Computing Machinery.

[50] Simon Eismann, Long Bui, Johannes Grohmann, Cristina Abad, Nikolas Herbst, and Samuel Kounev.

Sizeless: Predicting the optimal size of serverless functions. In *Proceedings of the 22nd International Middleware Conference*, pages 248–259, 2021.

[51] Simon Eismann, Joel Scheuner, Erwin Van Eyk, Maximilian Schwinger, Johannes Grohmann, Nikolas Herbst, Cristina L Abad, and Alexandru Iosup. A review of serverless use cases and their characteristics. *arXiv preprint arXiv:2008.11110*, 2020.

[52] D. H. J. Epema. An analysis of decay-usage scheduling in multiprocessors. SIGMETRICS '95/PERFORMANCE '95, page 74–85, New York, NY, USA, 1995. Association for Computing Machinery.

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

[54] Yuqi Fu, Li Liu, Haoliang Wang, Yue Cheng, and Songqing Chen. SFS: Smart OS Scheduling for Serverless Functions. In *SC22: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–16. IEEE, 2022.

[55] Mor Harchol-Balter, Nikhil Bansal, Bianca Schroeder, and Mukesh Agrawal. Implementation of srpt scheduling in web servers. 04 2001.

[56] Mor Harchol-Balter, Bianca Schroeder, Nikhil Bansal, and Mukesh Agrawal. Size-based scheduling to improve web performance. *ACM Trans. Comput. Syst.*, 21(2):207–233, May 2003.

[57] Scott Hendrickson, Stephen Sturdevant, Tyler Harter, Venkateshwaran Venkataramani, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Serverless computation with openlambda. In *8th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 16)*, Denver, CO, June 2016. USENIX Association.

[58] Jack Tigar Humphries, Neel Natu, Ashwin Chaugule, Ofir Weisse, Barret Rhoden, Josh Don, Luigi Rizzo, Oleg Rombakh, Paul Turner, and Christos Kozyrakis. ghost: Fast & flexible user-space delegation of linux scheduling. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, pages 588–604, 2021.

[59] Al Amjad Tawfiq Isstaif and Richard Mortier. Towards latency-aware linux scheduling for serverless workloads. In *Proceedings of the 1st Workshop on SErverless Systems, Applications and MEthodologies*, pages 19–26, 2023.

[60] Rajendra K Jain, Dah-Ming W Chiu, William R Hawe, et al. A quantitative measure of fairness and discrimination. *Eastern Research Laboratory, Digital Equipment Corporation, Hudson, MA*, 21:1, 1984.

[61] Artjom Joosen, Ahmed Hassan, Martin Asenov, Rajkarn Singh, Luke Darlow, Jianfeng Wang, and Adam Barker. How does it function? characterizing long-term trends in production serverless workloads. In *Proceedings of the 2023 ACM Symposium on Cloud Computing*, SoCC '23, page 443–458, New York, NY, USA, 2023. Association for Computing Machinery.

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

[63] Kostis Kaffes, Jack Tigar Humphries, David Mazières, and Christos Kozyrakis. Syrup: User-defined scheduling across the stack. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, SOSP '21, page 605–620, New York, NY, USA, 2021. Association for Computing Machinery.

[64] Kostis Kaffes, Neeraja J Yadwadkar, and Christos Kozyrakis. Centralized core-granular scheduling for serverless functions. In *Proceedings of the ACM symposium on cloud computing*, pages 158–164, 2019.

[65] Kostis Kaffes, Neeraja J Yadwadkar, and Christos Kozyrakis. Hermod: principled and practical scheduling for serverless functions. In *Proceedings of the 13th Symposium on Cloud Computing*, pages 289–305, 2022.

[66] Jeongchul Kim and Kyungyong Lee. Functionbench: A suite of workloads for serverless cloud function service. In *2019 IEEE 12th International Conference on Cloud Computing (CLOUD)*, pages 502–504. IEEE, 2019.

[67] Teng Ma, Shanpei Chen, Yihao Wu, Erwei Deng, Zhuo Song, Quan Chen, and Minyi Guo. Efficient scheduler live update for linux kernel with modularization. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, pages 194–207, 2023.

[68] Ashraf Mahgoub, Edgardo Barsallo Yi, Karthick Shankar, Eshaan Minocha, Sameh Elnikety, Saurabh Bagchi, and Somali Chaterji. Wisefuse: Workload characterization and dag transformation for serverless workflows. *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, 6(2):1–28, 2022.

[69] Sebastiano Miano, Xiaoqi Chen, Ran Ben Basat, and Gianni Antichi. Fast in-kernel traffic sketching in ebpf. *ACM SIGCOMM Computer Communication Review*, 53(1):3–13, 2023.

[70] Djob Mvondo, Antonio Barbalace, Jean-Pierre Lozi, and Gilles Muller. Towards user-programmable schedulers in the operating system kernel. In *SPMA 22-11th workshop on Systems for Post-Moore Architectures*, pages 1–4, 2022.

[71] Edward Oakes, Leon Yang, Dennis Zhou, Kevin Houck, Tyler Harter, Andrea Arpaci-Dusseau, and Remzi Arpaci-Dusseau. {SOCK}: Rapid task provisioning with {Serverless-Optimized} containers. In *2018 USENIX annual technical conference (USENIX ATC 18)*, pages 57–70, 2018.

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

[73] George Prekas, Marios Kogias, and Edouard Bugnion. Zygos: Achieving low tail latency for microsecond-scale networked tasks. In *Proceedings of the 26th Symposium on Operating Systems Principles*, pages 325–341, 2017.

[74] Shixiong Qi, Leslie Monis, Ziteng Zeng, Ian-chin Wang, and K. K. Ramakrishnan. Spright: extracting the server from serverless computing! high-performance ebpf-based event-driven, shared-memory processing. In *Proceedings of the ACM SIGCOMM 2022 Conference*, SIGCOMM '22, page 780–794, New York, NY, USA, 2022. Association for Computing Machinery.

[75] Henry Qin, Qian Li, Jacqueline Speiser, Peter Kraft, and John Ousterhout. Arachne:{Core-Aware} thread management. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 145–160, 2018.

[76] Bianca Schroeder and Mor Harchol-Balter. Web servers under overload: How scheduling can help. *ACM Trans. Internet Technol.*, 6(1):20–52, February 2006.

[77] Mohammad Shahrad, Rodrigo Fonseca, Inigo Goiri, Gohar Chaudhry, Paul Batum, Jason Cooke, Eduardo Laureano, Colby Tresness, Mark Russinovich, and Ricardo Bianchini. Serverless in the wild: Characterizing and optimizing the serverless workload at a large cloud provider. In *2020 USENIX annual technical conference (USENIX ATC 20)*, pages 205–218, 2020.

[78] Armando P. Stettner. *The Design and Implementation of the 4.3BSD UNIX Operating System*. Addison-Wesley Longman Publishing Co., Inc., USA, 1988.

[79] Amoghavarsha Suresh, Gagan Somashekar, Anandh Varadarajan, Veerendra Ramesh Kakarla, Hima Upadhyay, and Anshul Gandhi. Ensure: Efficient scheduling and autonomous resource management in serverless environments. In *2020 IEEE International Conference on Autonomic Computing and Self-Organizing Systems (ACSOS)*, pages 1–10, 2020.

[80] Amoghvarsha Suresh and Anshul Gandhi. Fnsched: An efficient scheduler for serverless functions. In *Proceedings of the 5th International Workshop on Serverless Computing*, WOSC '19, page 19–24, New York, NY, USA, 2019. Association for Computing Machinery.

[81] Gustavo Totoy, Edwin F Boza, and Cristina L Abad. An extensible scheduler for the openlambda faas platform. *Min-Move'18*, 2018.

[82] Carl A. Waldspurger and William E. Weihl. Lottery scheduling: Flexible Proportional-Share resource management. In *First Symposium on Operating Systems Design and Implementation (OSDI 94)*, Monterey, CA, November 1994. USENIX Association.

[83] Ao Wang, Shuai Chang, Huangshi Tian, Hongqi Wang, Haoran Yang, Huiba Li, Rui Du, and Yue Cheng. {FaaSNet}: Scalable and fast provisioning of custom serverless container runtimes at alibaba cloud function compute. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*, pages 443–457, 2021.

[84] Zhiyuan Wang, Jiancheng Ye, Dong Lin, Yipei Chen, and John CS Lui. Designing approximate and deployable srpt scheduler: A unified framework. In *2021 IEEE/ACM 29th International Symposium on Quality of Service (IWQOS)*, pages 1–6. IEEE, 2021.

[85] Adam Wierman and Bert Zwart. Is tail-optimal scheduling possible? *Oper. Res.*, 60(5):1249–1257, sep 2012.

[86] Hanfei Yu, Christian Fontenot, Hao Wang, Jian Li, Xu Yuan, and Seung-Jong Park. Libra: Harvesting idle resources safely and timely in serverless clusters. In *Proceedings of the 32nd International Symposium on High-Performance Parallel and Distributed Computing*, HPDC '23, page 181–194, New York, NY, USA, 2023. Association for Computing Machinery.

[87] Hanfei Yu, Athirai A. Irissappane, Hao Wang, and Wes J. Lloyd. Faasrank: Learning to schedule functions in serverless platforms. In *2021 IEEE International Conference on Autonomic Computing and Self-Organizing Systems (ACSOS)*, pages 31–40, 2021.

[88] Tianyi Yu, Qingyuan Liu, Dong Du, Yubin Xia, Binyu Zang, Ziqian Lu, Pingchao Yang, Chenggang Qin, and Haibo Chen. Characterizing serverless platforms with serverlessbench. In *Proceedings of the ACM Symposium on Cloud Computing*, SoCC '20. Association for Computing Machinery, 2020.