SFS: Smart OS Scheduling for Serverless Functions

Yuqi Fu
University of Virginia
Charlottesville, VA, USA
jwx3px@virginia.edu

Li Liu
George Mason University
Fairfax, VA, USA
lliu8@gmu.edu

Haoliang Wang
Adobe Research
San Jose, CA, USA
hawang@adobe.com

Yue Cheng
University of Virginia
Charlottesville, VA, USA
mrz7dp@virginia.edu

Songqing Chen
George Mason University
Fairfax, VA, USA
sqchen@gmu.edu

Abstract—Serverless computing enables a new way of building and scaling cloud applications by allowing developers to write fine-grained serverless or cloud functions. The execution duration of a cloud function is typically short—ranging from a few milliseconds to hundreds of seconds. However, due to resource contentions caused by public clouds' deep consolidation, the function execution duration may get significantly prolonged and fail to accurately account for the function's true resource usage. We observe that the function duration can be highly unpredictable with huge amplification of more than $50\times$ for an open-source FaaS platform (OpenLambda). Our experiments show that the OS scheduling policy of cloud functions' host server can have a crucial impact on performance. The default Linux scheduler, CFS (Completely Fair Scheduler), being oblivious to workloads, frequently context-switches short functions, causing a turnaround time that is much longer than their service time.

We propose SFS (Smart Function Scheduler), which works entirely in the user space and carefully orchestrates existing Linux FIFO and CFS schedulers to approximate Shortest Remaining Time First (SRTF). SFS uses two-level scheduling that seamlessly combines a new FILTER policy with Linux CFS, to trade off increased duration of long functions for significant performance improvement for short functions. We implement SFS in the Linux user space and port it to OpenLambda. Evaluation results show that SFS significantly improves short functions' duration with a small impact on relatively longer functions, compared to CFS.

Index Terms—Cloud computing, Operating systems, Performance evaluation

I. INTRODUCTION

Serverless computing, or Function-as-a-Service (FaaS), enables a new way of building and scaling applications and services by allowing developers to break traditionally monolithic server-based applications into finer-grained cloud functions. Developers write function logic while the service provider performs the notoriously tedious tasks of provisioning, scaling, and managing the backend servers [28] that the functions run on. Serverless computing solutions are growing in popularity and finding their way into both commercial clouds (e.g., AWS Lambda [3], Azure Functions [4], and Google Cloud Functions [7], etc.) and open-source projects (e.g., Open-Lambda [31, 13], OpenWhisk [55]). Popular uses of serverless computing today are event-driven and stateless applications such as web/API serving, image processing, and batch ETL (extract, transform, load) [1].

The execution duration of a cloud function is typically short–ranging from a few milliseconds (ms) to a few seconds [48]. Therefore, FaaS providers charge users at a fine granularity. For example, AWS Lambda bills on a per invocation basis ($0.02 per 1 million invocations) and charges the usage of bundled CPU-memory resources by rounding up the function's execution duration to the nearest 1 ms with a rate of $0.0000166667 per second for each GB of memory.

This fine-grained pricing model would be advantageous and fair to FaaS users if the execution duration of a function does not vary much (ideally one would expect that to be equivalent to the turnaround time as if the function was executed on a dedicated machine). This is particularly important considering the short-lived and highly heterogeneous nature of cloud functions: a majority of cloud functions have short execution duration while the execution times of all functions span seven orders of magnitude (from ms to hundreds of seconds). However, due to resource contentions caused by public clouds' deep consolidation, function execution duration—the turnaround time that measures the time when a function starts execution till the time when the function finishes execution and returns—gets prolonged and fails to accurately account for the actual resource usage of a successfully finished function. This covertly leads to overcharges to the users and potentially making them game the system in the long run [27].

Admittedly, function execution duration amplification may be caused by contentions from various levels of resources including CPU cache, CPU, memory, and network. However, our study shows that the CPU scheduling policy of host machines, e.g., the widely-used Linux CPU scheduling policy, Completely Fair Scheduler or CFS, can have a crucial impact on the execution duration of cloud functions hosted therein, therefore, a function scheduler must incorporate the unique FaaS workload patterns.

How to mitigate this amplification for short-job-dominant FaaS workloads is an open challenge, which, to the best of our knowledge, has not been well investigated. On a similar note, there are no well-defined performance SLOs (service level objectives) for short-job-dominant FaaS applications; one potential example SLO can be: "*X% of function invocations*

must be finished within a soft/hard-bounded ratio with respect to the duration that this function would observe if running in an ideally isolated environment".

Our key observation in this paper is that a majority of cloud functions in production FaaS workloads are short-lived with a wide spectrum of execution duration; the default Linux scheduler, CFS, frequently context-switches short functions, causing unfairly long waiting time, and therefore, longer turnaround time than they should have.

CFS is general-purpose and workload-oblivious, attempting to achieve CPU-task-level fairness: CPU tasks, no matter long-running or short-lived, get proportional share of the CPU resource under fine-grained time slices. This causes all CPU tasks with same priority to spend "fair" amounts of time waiting to be rescheduled. This inevitably leads to severely imbalanced function *run-time effectiveness (RTE)*, a new efficiency metric that we define to capture the ratio of function service time (aggregate CPU time) to end-to-end turnaround time (sum of the aggregate CPU time and the waiting time).

This motivates us to adopt Shortest Remaining Time First (SRTF)—a preemptive version of shortest job first (SJF)—which always schedules jobs that will complete the quickest. However, it is impossible to directly apply SRTF as it is an offline algorithm. To this end, we present SFS, a user-space function scheduler that minimizes turnaround time for short-function-intensive FaaS workloads. SFS works entirely in the user space, leveraging existing kernel scheduling policies (FIFO and CFS) to approximate SRTF. For this purpose, SFS adopts two-level scheduling: at the top level, SFS uses a new FILTER (FIFO-like) algorithm that schedules functions in the order they are enqueued and preempts them if they do not finish in a dynamically changing time slice; at the bottom level, those filtered functions from the top level continue in Linux CFS. This way, short functions can execute in their entirety without any context switch, or with minimum context switches if needed, in order to finish faster. The objective is to minimize the function execution duration and maximize the RTE metric such that the "pay-per-use" promise is delivered and unfair overcharges are reduced.

SFS presents a novel and practical user-space scheduling solution that bridges the divide between custom, user-space scheduling and kernel scheduling: existing OS scheduling is FaaS-workload-oblivious and thus affects function performance; SFS utilizes historical workload statistics obtained in the user space to make informed scheduling decisions by automatically steering underlying OS scheduling policies. SFS strikes a balance between waiting time and request service time. SFS is transparent to existing FaaS platforms and requires minimum modifications for them to use SFS. SFS is also OS-scheduler-agnostic and does not require kernel modifications.

In summary, this paper makes the following contributions:

- Through a performance characterization study on an open-source FaaS platform (OpenLambda), we identify efficiency problems of existing Linux schedulers (CFS, FIFO, and RR) on serverless function scheduling.

- We design a new scheduler, SFS, which approximates SRTF. SFS features a novel FILTER algorithm in the user space that dynamically steers existing OS schedulers based on workload patterns to enable more efficient scheduling for FaaS workloads.
- We implement SFS as a standalone, user-space scheduler that can be easily ported with existing FaaS platforms.
- We perform extensive evaluation on standalone SFS and an SFS-ported OpenLambda. Results show that SFS significantly outperforms CFS: SFS improves turnaround time of short functions by two orders of magnitude against CFS with very small user-space overhead.

SFS targets the overall performance of a majority of functions that are short-lived. Experimental results show that SFS improves the execution duration of 83% of the functions by $49.6\times$ on average compared to CFS; for the remaining 17% of the functions that are relatively longer, they run $1.29\times$ longer on average under SFS than CFS. SFS is open sourced and publicly available at https://github.com/ds2-lab/SFS.

II. BACKGROUND

A. FaaS Overview

Serverless computing handles virtually all system administration tasks, making it easier for users to deploy and scale their cloud applications and services [35]. FaaS providers offer a flexible interface for defining cloud functions, which allows developers to focus on core application logic using languages such as Python, JavaScript, Java, Go, and others. FaaS providers in turn auto-scale function executions in a demand-driven manner, hiding tedious server configuration and management tasks from the users.

Cloud functions are deployed and executed in virtualized environments such as containers or virtual machines (VMs) for isolation and safety. A typical workflow of function deployment and execution works as follows. Step 1: A user submits the function code (via either a web interface or packaged .zip/container image files) to the FaaS platform for function creation. Step 2: The user executes the created function by sending an HTTP invocation request to a FaaS scheduler. Step 3: the FaaS scheduler forwards the invocation request to a FaaS worker that is running on a resource-rich host machine. Step 4: The FaaS worker creates a virtualized environment and installs the necessary dependencies for the virtualized environment before the function can be started. Step 5: After all previous steps are successfully completed, the FaaS worker sends the function request to the host OS, which in turn starts the function execution as an OS process.

While there are already extensive studies focusing on reducing functions' cold startup penalty (Step 2-4 in last paragraph) in FaaS [48, 17, 42, 41, 22, 25], in this paper, we aim to fill the missing gap by focusing on the "*last mile*" efficiency of function execution, i.e., *OS scheduling*, in Step 5.

B. OS Task Scheduling

Cloud functions are eventually scheduled and executed by a host OS. Functions typically have a short execution duration and small CPU-memory footprint, making FaaS workloads

increasingly consolidated. For example, a large bare-metal machine with 96 CPU cores, 384 GB of memory, and multi-TBs of NVMe SSDs can easily host tens of thousands of, if not more, function instances [16]. This statistical multiplexing makes it feasible for a FaaS provider to execute thousands of function processes concurrently on a single host.

Basic Scheduling Policies. First in, First out (FIFO) and Round-Robin (RR) are among the most basic scheduling policies. They have different tradeoffs. When a FIFO task starts running, it runs to completion[1]. Similar to FIFO, core-granular scheduling [38] designates a single core to a function and allows it to run to completion. However, also like FIFO, core-granular scheduling may hurt response time when the system is highly consolidated and under high utilization with a line of queued tasks. RR can be used to optimize responsiveness. RR runs a CPU task for a time slice and then switches to the next queued task. However, since the execution of a CPU task is divided into multiple slices, RR sacrifices turnaround time.

Proportional-Share Scheduling. Proportional-share scheduling is a type of CPU scheduling algorithms commonly used by today's OSes including VM scheduling and Linux scheduling. Proportional-share scheduling focuses on fairness and attempts to guarantee that each CPU task obtains a certain percentage of CPU time based on the task's priority. Well-known examples of proportional-share scheduling include lottery scheduling [60], Xen's credit scheduler [18, 15], and Linux's default scheduler CFS. CFS is the *de facto*, and the most commonly-used open-source OS task scheduler in productive environments including public clouds [2, 8, 10] and companies [32, 58].

Given its popularity and prevalence, we choose Linux's general-purpose CFS scheduler as a baseline and briefly describe how it works. In fact, the two virtualization techniques commonly used by today's FaaS platforms, containers and KVM-based VMs, both rely on CFS for OS task scheduling. For example, Docker containers [5] are used by open source FaaS platforms such as OpenLambda [31], OpenWhisk [55], and OpenFaaS [11], while AWS Lambda's Firecracker microVM [16] uses KVM for managing Lambda functions.

Linux CFS. CFS proportionally divides the physical time into fine-grained time slices among all CPU tasks based on their weights (priorities). CFS tracks the CPU time usage of each task using a virtual runtime (`vruntime`) scheme. `vruntime` records the CPU time that a CPU task has used weighted by its priority. In a multi-core system, each physical CPU core has its own `runqueue`, which is a red-black (RB) tree ordered by `vruntime`. A task will first be assigned by CFS to a `runqueue`; the task's location in the `runqueue` RB tree determines roughly when in the future it can execute; at each scheduling tick (i.e., the end of a time slice), CFS picks the next task that has the smallest `vruntime` from the RB tree. As the FaaS workload is increasingly consolidated, it is common to have thousands of concurrently running function processes

---

[1]Modern OSes must handle sophisticated situations such as I/O and priority. A FIFO task, once started, continues to run until it voluntarily yields control over CPU, blocks, or is preempted by a higher priority CPU task.

that multiplex the limited amount of CPU cores on the host machine. Therefore, the weight of a task simply indicates a *relative* CPU share, but not an absolute CPU share that a user would expect the function to get based on the function's resource configuration. Once preempted, the task needs to *wait* in the `runqueue` for its next turn to run. While waiting, the task' `vruntime` does not tick.

III. WHY IS CFS A POOR MATCH?

Run-time Effectiveness (RTE). CFS is a poor match for the emerging, short-function-intensive FaaS workloads, which values turnaround time. The fundamental mismatch comes from CFS' lack of workload awareness: CFS ensures a fair proportion of CPU time to all the CPU tasks but does not distinguish if a CPU task is long-running or short-lived. However, this application-level knowledge is critical to application performance, especially if applications mostly consist of short jobs, e.g., a FaaS workload [48]. Under a proportional-share scheduler such as CFS, the *fairness* is defined as follows: within a given time interval, all CPU tasks, if with the same priority, are assigned the same amount of CPU time to execute. We argue in this work, while it is "fair" to all CPU tasks from the low-level OS perspective, such "fairness" may inevitably create *unfairness* to the user-level applications – in our case these contained in the FaaS workload, since the waiting time may be disproportional to the execution time, considering the execution time diversity of FaaS workloads. In fact, FaaS workloads have unique characteristics that make existing Linux's "fair" scheduler actually *unfair*. Cloud functions feature a long spectrum of execution duration (§IV-A). Short functions with an execution duration of several ms to tens of ms are more sensitive to waiting time than longer functions that execute for, say tens of seconds. A mixture of such short and long functions co-located in the same server could spend roughly equal amounts of time waiting in `runqueues` before those short functions finish, resulting in disproportionally long waiting time. To quantify this affect, we define a new efficiency metric in this paper, function Run-Time Effectiveness (RTE) as follows:

$$RTE = \frac{\sum CPU^i}{turnaround time} \qquad (1)$$

where CPU^i means the CPU time allocated to this function in the ith round before the function returns. Thus, effectively, RTE reflects the ratio of the service time to the turnaround time. An RTE of 1 is the theoretically highest that a function could achieve, meaning that the function runs to completion without being preempted, so higher scores are better. Furthermore, RTE also indicates if a FaaS user has been overcharged. The closer an RTE to 1, the less overcharges that a user had to pay. For ideally CPU-intensive functions, an RTE of 1 indicates that the user is not overcharged at all. However, one should note that, in practice, it is not common to have functions with pure CPU bursts; therefore, an RTE score achieved under zero interference, though smaller than 1, would still represent a best-case baseline for comparison purposes.
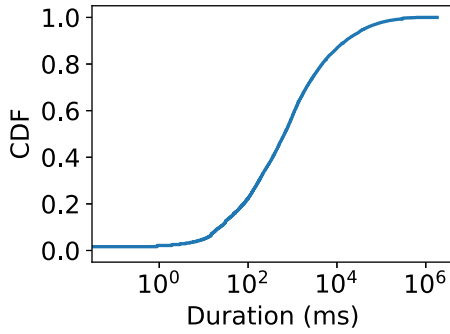
Fig. 1: CDF of the average function execution duration of Azure Functions traces.

Tradeoffs. Efficiently scheduling short and long jobs is a decades-old problem [64, 47, 24, 51, 30, 29, 36, 43]. Long jobs' performance will get affected under priority scheduling that approximates SRTF. The challenge is how to balance the tradeoff between the performance improvement for short jobs and performance loss for long jobs. We revisit this problem from a new angle—minimizing severely *disproportional waiting time* for short functions by trading off *disproportionally-increased turnaround time* for long functions—in the context of serverless function scheduling. That is, SFS aims to trade a smaller impact on long functions for significant performance improvement for short functions, a huge win for the short ones and a much smaller (relative) penalty for the longer ones.

IV. Motivation

A. Azure Functions Workloads

The Azure Functions workload datasets [48] are by far the only publicly available FaaS workload traces that we have access to. The traces were collected during a two-week period, containing the average, minimum, maximum execution duration breakdown per function and invocation counts per function sampled at each one-minute interval. We analyzed the distribution of the average execution duration of all function invocations in the two-week period (Figure 1). We observe that the function execution duration spans a total of seven orders of magnitude; specifically, about 37.2%, 57.2%, and 99.9% of the functions have an average execution duration shorter than 300 ms, 1 second, and 224 seconds, respectively.

Observation 1: *While real-world FaaS workloads have a mixture of short and long functions, a majority of them are extremely short-lived and latency-sensitive. Optimizing the execution duration of these short functions will provide a huge benefit for the overall performance of FaaS platforms.*

B. OpenLambda Measurement

We next measured the performance of an Azure-sampled FaaS workload on OpenLambda. We generated the workload based on the Azure Functions workload datasets [48]. Since our focus is on single-server scheduling, we downscaled the original trace by sampling the execution duration and request inter-arrival times of $49,712$ function requests from Day 1. More details about workload generation are described in §VII.

We configured OpenLambda to use 12 CPU cores and Linux's real-time (RT) schedulers, `SCHED_FIFO` (FIFO) and `SCHED_RR` (RR), as well as Linux's default proportional-share scheduler, `SCHED_NORMAL` (CFS). We tested the workload with two load levels, an average load of 80% over all 12 cores, and an average of 100% load, and compared OpenLambda's performance against an offline oracle scheduler SRTF. SRTF always selects the job with the smallest remaining time to execute. SRTF is optimal as it assumes *a priori* knowledge of function duration. IDEAL scheduling represents the ideal scenario where there are infinite resources with zero contention.

Figure 2 shows the performance and RTE results. In calculating the RTE, the aggregate CPU time of a function is measured under the IDEAL scenario while the turnaround time of the function is measured under the workload. From this figure, we have the following observations. (1) SRTF, as an offline scheduling policy in favor of short jobs, is provably optimal for turnaround time [21]; SRTF approached the IDEAL performance, which was achieved with infinite resources. (2) None of Linux's three CPU schedulers was able to offer good performance under both the 80% and 100% load under practical FaaS workloads (Figure 2(a)); CFS performed the best among all Linux scheduling policies, but still, there were about 11.4% and 89.9% of the function requests that achieved an RTE score < 0.2 (Figure 2(b)). This explains why SRTF outperformed CFS: with the same service time, functions were preempted more under CFS, causing longer waiting times. (3) Under the 100% load, functions executed more than one order of magnitude slower under CFS than SRTF, with a 40th and 70th percentile slowdown of $16\times$ and $24\times$, respectively, again, because of the dominant waiting time. (4) RT schedulers offered the worst performance: FIFO performed the worst due to the "convoy effect", where short functions were blocked behind long functions.

Observation 2: *Approximating the offline oracle SRTF by improving the run-time effectiveness will promise a significant performance boost for short serverless functions.*

V. SFS Design

Our study in §IV shows that cloud functions often suffer high execution duration amplifications. Among all scheduling strategies, SRTF is promising (than CFS). This motivates the design of a new scheduler SFS to prioritize short functions. In this section, we present the design principle and challenges of SFS, followed by the design details.

A. Design Goals and Challenges

To prioritize short functions, a priority-based scheduler is needed. However, as described earlier in §II-B, proportional-share schedulers such as CFS are designed for optimizing fairness for long-running jobs and avoiding starvation. To achieve such goals when multiple concurrently running jobs are consolidated on a single server, CFS squeezes the time slice for each competing job and proportionally shares the physical CPU time among them. This leads to a significantly prolonged "scheduling cycle": a job that has used up its time slice is descheduled and must wait for a long time before it gets rescheduled. For short jobs, this prolonged waiting time
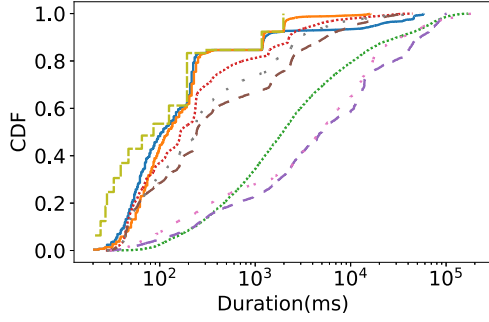
(a) Execution duration distribution.
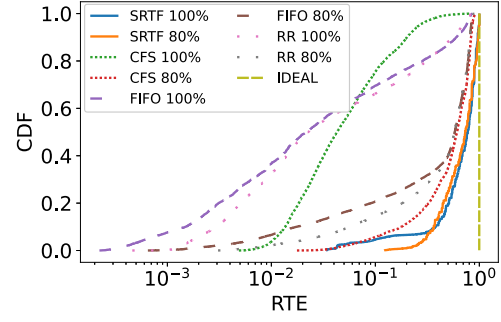
(b) Run-time effectiveness (RTE) distribution.

Fig. 2: Performance and RTE of an Azure-sampled workload on OpenLambda with different scheduling policies and different loads.

hurts turnaround time: they could have finished much earlier without preemption if given a *long-enough* time slice.

Our motivational study from Figure 2 shows that SRTF can achieve much better performance than CFS. SRTF provides a theoretical lower bound in terms of turnaround time for short-function-dominant FaaS workloads because SRTF allows a short-enough function to be scheduled instantly and run to completion without preemption. However, SRTF is not practical as it assumes *a priori* knowledge about job duration.

Our goal is to design an online scheduler that approximates SRTF. We achieve this goal by addressing the following challenges. First, cloud functions are much shorter with a typical duration ranging from tens of ms to several seconds, and FaaS workloads exhibit transient overload. Such workload characteristics pose a challenge in designing effective prioritization strategies, which should prioritize short functions in a timely manner with minimal impact on longer functions.

Second, existing FaaS platforms use a client-server-based microservice architecture, where clients issue HTTP invocation requests to execute cloud function instances hosted by backend FaaS servers. Our design must provide a transparent and portable function scheduler that requires no or minimum modification of existing FaaS platforms while being OS-scheduler-agnostic. That is, even if the FaaS platform uses CFS or other OS-level proportional scheduling schemes, short functions should gain higher priority than longer functions. Thus, a second challenge is how to design an efficient and practical function scheduler, which (1) works entirely in the user space and does not require kernel modifications, (2) works alongside (rather than replacing) OS schedulers and exploits (whenever needed) OS scheduling properties such as work conservation to provide better support for FaaS workloads, (3) while being transparent to FaaS servers.

B. Design Overview

Typically, a FaaS platform uses a client-server architecture as depicted in Figure 3: a directly-user-facing gateway forwards HTTP invocation requests from users to a backend FaaS server that hosts requested function instances. To be transparent and portable to existing FaaS platforms, we design SFS by following a black-box approach. A backend FaaS server dispatches function invocations to the underlying OS.
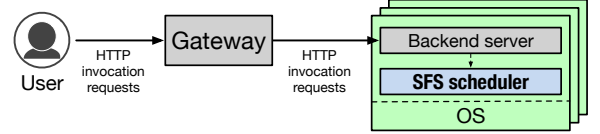
Fig. 3: Overview of a typical FaaS platform deployment.

SFS assumes the existence of such a backend FaaS server. SFS serves as a user-space middle layer between a FaaS server and the OS (Figure 3), intercepting function requests and performing function scheduling for the FaaS server.

Figure 4 illustrates the overall function scheduling flow. SFS orchestrates Linux's existing schedulers (FIFO and CFS) in the user space. SFS adopts two-level scheduling that seamlessly combines a FIFO-like scheduling policy based on Linux FIFO at top level and a kernel-space scheduling policy offloaded to Linux CFS at bottom level. The top-level scheduler schedules function requests by the order in which they are enqueued in the global queue and filters out those longer functions that are not finished in a (dynamically configurable) time slice. This way, the top-level scheduler effectively serves as a **FILTER**[2]. Under SFS, a function's lifespan may experience one or two phases: a *FILTER phase* and/or a *CFS phase*. A function by default starts in FILTER mode. SFS dynamically adapts a *time slice* parameter S (discussed later) using a sliding window approach and uses S to bound a function's execution in FILTER mode. This way, SFS approximates SRTF. SFS is inherently work-conserving following a single queue model: SFS workers fetch requests whenever they are idle. To minimize context switches, SFS guarantees that a function that is executing in FILTER mode would be preempted only if it has used up the time slice or it is waiting for an event (e.g., I/O).

Scheduling Flow. Next, we describe the main components and the scheduling flow of SFS as illustrated in Figure 4:

1. A backend FaaS server, serving as a client, dispatches function invocation requests, launches requested functions in a virtualization environment in OS, and sends the information of the dispatched function requests (tuples of

---

[2]FILTER: **F**irst **I**n but **L**onger jobs **T**o **E**xtra **R**unqueue.
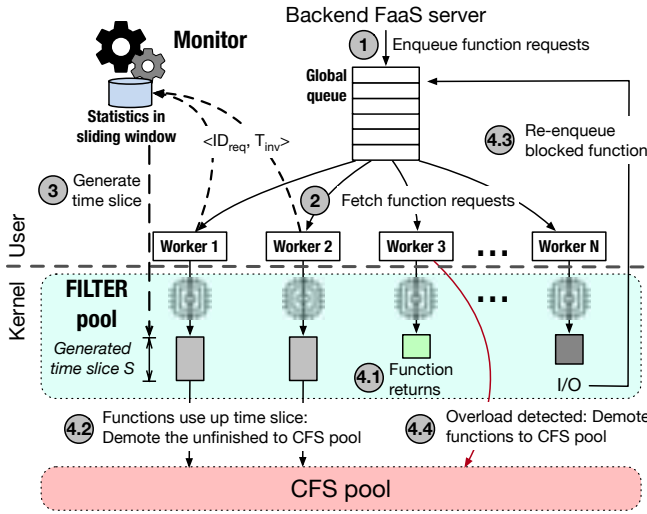
Fig. 4: SFS architecture. SFS' components are highlighted in condensed bold font.

unique function request ID and the invocation timestamp) to SFS' **global queue**.

2. Multiple SFS **workers**, each responsible for scheduling function requests on a separate CPU core, concurrently fetch function requests from the global queue whenever workers are free. Note a non-empty global queue indicates that all cores are busy serving requests. Each SFS worker is responsible for intercepting the dispatched function process by using the tuple information fetched from the queue and scheduling the function using FILTER policy. This way, each function by default starts execution in FILTER mode, unless otherwise specified (§V-E). This effectively forms a **FILTER pool** of multiple SFS workers. Functions that are executing under FILTER mode naturally gain higher priority than those under CFS mode. To realize this, SFS changes a running function process' OS scheduling policy to FIFO (SCHED_FIFO), which has higher static priorities than CFS (SCHED_NORMAL) processes [12].

3. Each SFS worker stores the following statistics information in memory: (1) function request ID and its invocation timestamp, which is initially recorded in the global queue when the function request was submitted, and (2) function execution timestamp, which is the time when the function starts execution. An SFS **monitor** periodically re-calculates a global *time slice* parameter S based on the collected statistics (§V-C). Next, we describe several cases of the function execution.

4.1. The ideal case is that a short function finishes execution and returns before using up S. This way, the SFS worker marks its completion, removes the corresponding entry from the global queue, fetches the next function request, and restarts the time slice timer.

4.2. The SFS worker keeps track of the runtime of the FILTER function, forcibly preempts it if its time slice expires, and demotes it to CFS.

4.3. If a function is blocked by some event, e.g., an I/O event, the worker will instantly preempt its execution and add it

back to the global queue (§V-D).

4.4. If an SFS worker detects a transient overload by observing increasing queuing delay (above a certain threshold), it temporarily disables FILTER and directly schedules next requests using CFS (§V-E).

C. Dynamically Adapting Time Slices

While it may be impossible to design a perfect scheduler that assigns a precise time slice that perfectly matches the remaining execution duration of any job, SFS uses a simple yet effective heuristic approach based on queuing theory to estimate and dynamically adapt the time slice parameter S.

The time slice parameter presents an interesting tradeoff between queuing delays and the turnaround time of the workload. On the one hand, an (improperly) short time slice value would reduce global queuing delays of outstanding function requests but unnecessarily cause a longer turnaround time due to increased context switches. CFS falls to this end of the spectrum. On the other hand, an (unnecessarily) long time slice value may reduce number of context switches but increase global queuing delays, and as a result, hurt turnaround time.

Therefore, we use queuing theory to set the time slice. For this purpose, we can model a multi-core scheduler as a multi-server queuing system—an $M/G/c$ model according to Kendall's notation—using the following equation:

$$\rho = \frac{\lambda}{c\mu} \qquad (2)$$

where λ is the arrival rate of the requests, μ is the service rate of a single core, c is the number of cores used, and ρ is the traffic intensity per core (i.e., per-core utilization). We can thus use the utilization metric ρ as a measure of queuing delay: if ρ is greater than one, meaning the arrival rate λ is larger than the aggregate service rate $c\mu$, the length of SFS' global queue will grow without bound. Intuitively, adapting the service rate μ based on the changing λ can bound ρ, thus the overall queuing delay. However, in practice, μ is solely determined by the workload and the capacity of the underlying hardware. Therefore, SFS enforces a global time slice to cap the duration for how long any function may run in FILTER mode. SFS dynamically changes the time slice in response to the variable request arrival rate, which is estimated using historical IATs.

Following Equation 2, SFS keeps track of a small sliding window of last N requests' inter-arrival times (IATs) to determine the time slice parameter S. For a single-core system, SFS calculates the average IAT of last N functions, $\overline{IAT}$, and uses it as the feedback to dynamically tune S. With c cores, $S = \overline{IAT} * c$. Intuitively, S is used to bound the service rate μ of Equation 2, which in turn affects the traffic intensity ρ; ρ further affects the queuing delay of function requests that are executing under FILTER mode, and therefore, SFS uses the historical IAT information to strike a balance between queuing delay and execution time. When a global S is selected, SFS guarantees that all functions whose execution duration is shorter than S run to completion without being preempted. SFS re-calculates a new S for every N function requests that

has been enqueued in order to provide dynamic adaptation to workloads. N is configurable and we choose 100 as N in our evaluation.

There may always be functions that are not able to finish before the time slice elapses. To solve this issue, SFS uses a single-level FILTER pool concatenated with CFS to approximate SRTF. SFS steers Linux FIFO directly from the user space and builds the FILTER policy as a high-priority queue for short functions. SFS transparently leverages CFS as a black-box, lower-priority queue for longer functions. Note that functions running in CFS share the same set of cores as those running in FILTER mode. Starvation is mitigated since CFS is work-conserving and can immediately schedule any demoted functions on any available CPU core.

D. Handling I/Os

Since SFS is a user-space scheduler, it cannot transparently handle kernel-level tasks such as context switches, interrupts, and preemptions, etc. That is, an SFS worker could be waiting for a blocked function that has already been preempted due to an I/O event. This leads to sub-optimal decision-making with regard to function timekeeping and time slice estimation. To solve this issue, SFS workers track the kernel-level process status of the function by periodically issuing a polling request to the OS. When a function is in its CPU burst, its kernel-level status is in `running` mode. Whenever a function changes its kernel-level status from `running` to `sleep`, the SFS worker detects this transition, stops its timekeeping and records the unused time slice, reduces its priority, and schedules the next available function from the global queue. Note that, when a high-priority function blocks by I/O, CFS automatically sneaks in and executes other functions that have been filtered by SFS. This guarantees seamless work conservation. When the status of a waiting function changes to `runnable`, SFS adds it back to global queue. When this function gets rescheduled in the FILTER pool, it will execute until it completes or it uses up the rest of the time slice. We use 4 ms as the polling interval. We evaluate this scheme in §VIII-B and its overhead in §IX-B.

E. Handling Overload

Real-world FaaS workloads exhibit highly bursty and unpredictable load patterns [48, 61]. Alibaba Function Compute workload analysis reports transient spikes of concurrent invocations to the same function [61]. When an increasing number of short functions get piled up at global queue in a very short time (increasing arrival rate λ in Equation 2), the service rate of SFS' FILTER pool, $c\mu$, cannot catch up with the workload spike. This transient (temporary) overload leads to increased traffic intensity, ρ, therefore, increased queuing delay and even function request drop. Reducing the time slice of the FILTER pool helps little in this scenario. This is because a FILTER time slice shorter than that of CFS would cause more context switches than CFS; as a result, the piled-up FIFO function requests from the transient overload create backlog that cannot be quickly consumed by FILTER workers (see Figure 12(a) as an example). To solve this issue, SFS temporarily switches to CFS when any SFS worker detects increasing queuing delay of

the function request that it is about to schedule using FILTER. As long as the queuing delay lowers back to normal, SFS workers roll back to the normal scheduling flow.

This strategy, though simple, is in fact very effective because of the following reasons. Offloading accumulated functions to CFS alleviates high queuing delays in the FILTER pool by draining the backlog more quickly. Since overload is transient, regular load coming after that can then be serviced by SFS' default, time-slice-based FILTER pool first, thus, short functions experience no further queuing delays and can finish in one round before the time slice expires. Those function requests that are offloaded during the overload to CFS are eventually complete thanks to CFS' work conservation. An SFS worker detects overload if the queuing delay is at least $O \times S$. We set O as 3 empirically. We evaluate the efficacy of this strategy in §VIII-B.

VI. SFS IMPLEMENTATION

We have implemented SFS as a standalone, user-space function scheduler in Go. We have also ported SFS to an open-source FaaS platform OpenLambda [31]. Porting SFS to OpenLambda required a very small engineering effort: we modified/added 29 lines of Go/Python code in OpenLambda worker and sandbox server to interface with SFS.

Standalone SFS. We implemented the SFS global queue structure using Go's built-in, thread-safe channel. SFS workers are goroutines (a lightweight user-level thread of execution managed by the Go runtime), which are responsible for dispatching and scheduling function processes in the FILTER pool. Function invocation requests are pushed into the global queue channel by an external, backend FaaS server (Figure 3-4). SFS implements the switching from FILTER pool to CFS pool by using Linux `schedtool` [14]. Function invocation requests are executed in sandboxed processes scheduled by either SFS workers or the CFS scheduler.

We chose to implement a global queue instead of using a per-core-queue (i.e., multi-queue) design because a single global queue guarantees natural work conservation with good load balancing across all CPU cores. It is demonstrated that a per-core-queue design has multiple downsides, e.g., severe load imbalance, core under-utilization, and degraded performance [44]. In our current implementation, the global queue is implemented using a Go channel (with nanosecond enqueue/dequeue latency under multi-threaded environments), which is capable of handling up to 100 CPU cores each running tasks with a duration from ms to seconds. A single global queue, however, might become a bottleneck if assuming hundreds of CPU cores and microsecond-level function execution duration. Since the server hosts deployed by a typical FaaS provider may have up to 100 vCPUs [16], our global queue design is an appropriate solution.

SFS workers are work-conserving: each worker is blocked on global queue and fetches a function request whenever the queue has entries to consume. To implement FILTER policy, SFS workers use `schedtool` to change a running function process' OS scheduling policy from CFS (`SCHED_NORMAL`) to
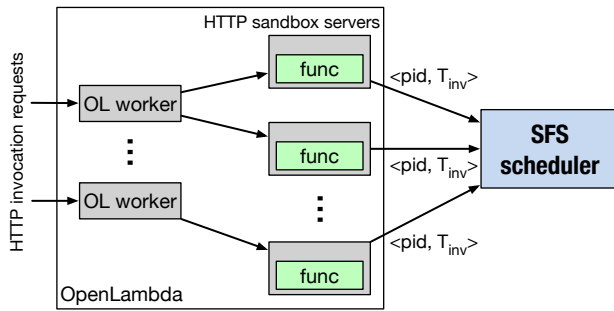
Fig. 5: Porting SFS to OpenLambda. OL: OpenLambda.

FIFO (`SCHED_FIFO`). When the server is under low utilization, a function that is dispatched by the FaaS server may execute in CFS for a very short period of time (hundreds of microseconds, depending on the communication overhead between the FaaS server and SFS). Under this situation, CFS performs the same as SFS due to zero contention. A non-empty global queue indicates that all CPU cores are busy serving function requests; if so, newly dispatched functions are internally queued at OS-level run queues (§II-B) as CFS jobs have inherently lower priority than actively running FIFO jobs. SFS workers preempt a FIFO function's execution by using `schedtool` to assign it a lower priority. SFS workers detect blocking events by periodically (4 ms) polling the function process' OS status using a go process utilities library `gopsutil` [9].

SFS is designed to be, in principle, portable to any open source FaaS platforms such as OpenLambda [31], Open-Whisk [55], and OpenFaaS [11], as they all share the same client-server architecture. To demonstrate the portability, we have ported SFS to OpenLambda. As shown in Figure 5, a backend OpenLambda deployment consists of two components: (1) a cluster of OpenLambda workers that are responsible for receiving function invocation requests, performing sandbox auto-scaling, and tracking statistics, and (2) a cluster of HTTP servers that manage function sandboxes. In our implementation, we chose Docker containers as the function sandbox. On-demand container provisioning (i.e., cold start) incurs high overhead. Therefore, we disabled auto-scaling and pre-warmed enough function containers to simulate a stable-phase FaaS backend so as to accurately quantify the performance of schedulers.

Porting SFS to OpenLambda. We modified OpenLambda's HTTP sandbox server to communicate with SFS scheduler using UDP: whenever a sandbox server dispatches a function request to OS, it sends to SFS a UDP message containing function process PID and invocation timestamp.

VII. EXPERIMENTAL METHODOLOGY

Setup. We developed and tested SFS on CloudLab [23]. We evaluated SFS and SFS-ported OpenLambda on two AWS EC2 VMs: a small, `c5a.4xlarge` VM with 16 vCPUs and 64 GB memory (standalone SFS), and a large, bare-metal, `m5.metal` EC2 instance with 96 vCPUs and 384 GB memory (OpenLambda). The goal of using a large bare-metal machine is to simulate a similar FaaS deployment environment used by major FaaS providers such as AWS Lambda [16].

FaaSBench. We have built a FaaS workload generator called FaaSBench, which creates FaaS workloads modeled after the Azure Functions workload [48]. FaaSBench is highly configurable along the following dimensions: (1) FaaSBench configures per-function behaviors by using a Fibonacci (`fib`) function with two knobs: an integer knob N for controlling the compute time, and a boolean knob IO that toggles the I/O operation (if set true) to simulate I/O-intensive functions. The distributions of (2) function duration and (3) requests' inter-arrival times (IATs) are also configurable.

Generating FaaS Workloads. We based the Azure Functions traces to generate testing workloads. The original Azure traces contain the execution duration, memory sizes, and invocation timestamps of $82,375$ unique function applications spanning a period of 14 days. To downscale, we generated the distribution of function execution duration based on Day 1's invocation statistics. We found that the duration roughly follows a multimodal distribution, where about 40.6%, 22.7%, and 15.7% of invocations fall in a duration (unit of ms) range of $(0, 50]$, $[200, 400)$, and $[1550, \infty)$, respectively. We built a table that maps the range of execution duration recorded in Day 1 to `fib`'s Ns (Table I), and then used Azure function duration distribution to generate Ns using FaaSBench. For example, `fib` with an N between 20-26 finishes execution in less than 45 ms, therefore, we programmed FaaSBench to generate `fib` functions with an N between 20-26 with a probability of 40.6%. FaaSBench can also generate different duration distributions (results omitted due to page limit).

TABLE I: Probability distribution of function duration ranges and the corresponding `fib` Ns. Note ranges are non-contiguous and each missing range has less than 1% probability in Azure traces (Day 1).

Probability	Duration	N
40.6%	0-50 ms	20-26
9.8%	50-100 ms	27-28
6.8%	100-200 ms	29
22.7%	200-400 ms	30-31
15.7%	≥ 1550 ms	34-35

The released Azure trace datasets only contain the high-level statistical breakdown information that describes the function execution behaviors. To make sure that our study captures the original Azure Functions workload as accurately as possible, we took the 50th percentile execution duration as the expected execution time for a function. This way, our benchmark rules out outliers that do not represent the typical behaviors of the original workload.

To configure IATs of the generated workload, we randomly sampled 100 unique function applications, each with a total invocation count greater than 200 on Day 1, and extracted the IAT statistics. We then replayed the first $10,000$ invocation requests by strictly following the extracted IAT patterns. This is to guarantee that our generated workload preserves similar load patterns as real-world, production workloads. In addition to modeling existing trace's IATs, FaaSBench can also generate Poisson and uniform IATs. We ran each test multiple times and results had negligible variation across runs.

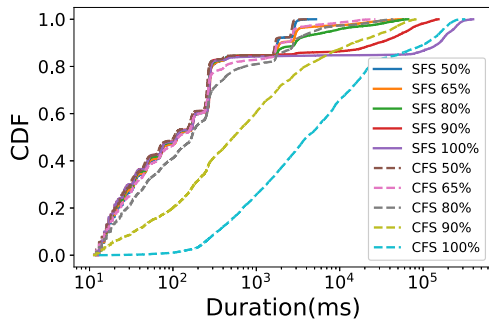**Goals.** Our evaluation aims to answer the following questions:
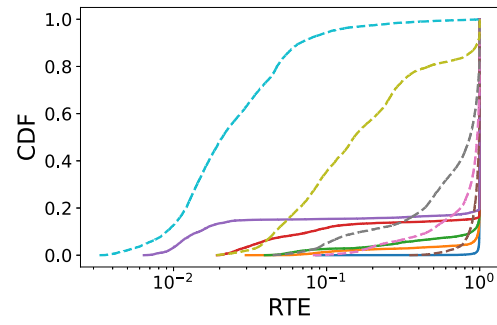
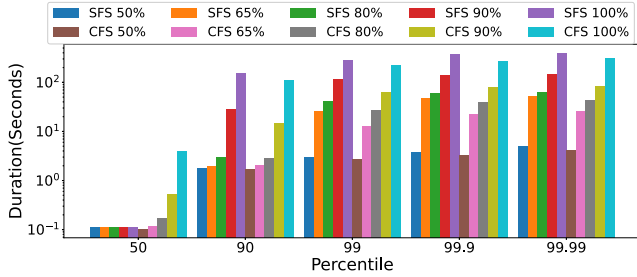**Fig. 6:** Performance CDF.

Fig. 7: RTE CDF.

Fig. 8: Percentile breakdowns of function execution duration.

- How does SFS perform under various loads (§VIII-A)?
- How do different SFS configurations affect its performance (§VIII-B)?
- How does an SFS-ported FaaS platform (OpenLambda) perform (§IX)?

VIII. EVALUATING STANDALONE SFS

In this section, we evaluate SFS as a standalone function scheduler using FaaSBench. The goal of evaluating standalone SFS is to better understand the true performance characteristics of task scheduling without the extra overhead introduced by a FaaS platform.

A. SFS Efficiency under Various Loads

We first test SFS under different loads using the Azure-sampled workload generated by FaaSBench, which follows the duration distribution of Azure traces (Table I) with a Poisson IAT distribution. We adjusted the IAT of the generated workload proportionally to simulate different loads ranging from 50% to 100% of overall CPU utilization across all CPU cores. Figure 6 reports the CDF of the execution duration. SFS performed almost the same as CFS under the lowest 50% load and slightly outperformed CFS under medium loads when the load increased from 65% to 80%. SFS' marginal improvement was obtained because of a higher RTE. As shown in Figure 7, about 93% and 88% of all function requests receive an RTE ≥ 0.95 under a load of 65% and 80%, respectively, indicating that these functions run to completion without any context switch under SFS (with a very short queuing delay when the request was initially submitted). CFS is workload-oblivious, which introduces more context switches; under a load of 65% and 80% with CFS, only 55% and 35% of all function requests

receive an RTE score ≥ 0.95, where a lower RTE translates to prolonged waiting time.

An interesting observation is that SFS maintains almost identical performance for 83% of the function requests across all load levels. In other words, at least 83% of the function requests can achieve optimal execution duration and an RTE score of almost 1 even under a high load where all CPUs are 100% utilized.

Under CFS, the same set of functions, on the other hand, saw dramatically increased execution duration because of prolonged waiting time (Figure 7). This result demonstrates SFS' efficacy in sustaining dynamic FaaS workloads.

The performance gain of shorter functions under SFS does not come for free: there is always a tradeoff in balancing the scheduler efficiency for short and long jobs [64]. SFS observed slightly higher tail latency. For the 17% relatively long functions, SFS observed an average increase of 1.29× in execution duration compared to CFS under the 100% load. The 99.9^{th} percentile latency of SFS under 80% load is only 47.1% higher than that of CFS (Figure 8). CFS, while being a proportional-share scheduler, does suffer long tail latency even under relatively lower load; this can be seen from the fact that the 99.9^{th} percentile latency of CFS increases from 3.3 seconds under the 50% load to 22.1 seconds under the 65% load; though the increase of the 99.9^{th} percentile tail latency under SFS is slightly higher than that under CFS.

Interestingly, SFS achieved a consistent medium (50th percentile) latency of 0.1 second across all load levels, while CFS' medium latency increases as the load increases. Longer functions could be potentially offloaded to relatively lighter-loaded FaaS servers by the global FaaS scheduler to mitigate the performance impact, which we plan to investigate as part of our future work.

More importantly, even under a 100% load, SFS offers short functions consistently competitive performance comparable to a performance level that would have been achieved under less stringent load situations, say 65%–90% load. This would bring desired benefits for both parties, including mitigated overcharges for FaaS users and higher resource utilization (thus reduced deployment costs) for FaaS providers.
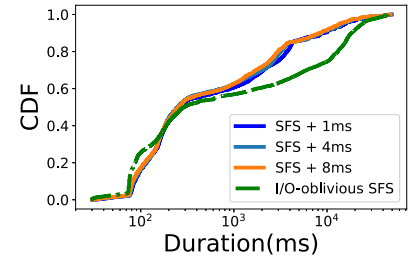
B. Sensitivity Analysis

Impact of Time Slice Configurations. Next, we conduct a sensitivity test by varying the time slice parameters S. We
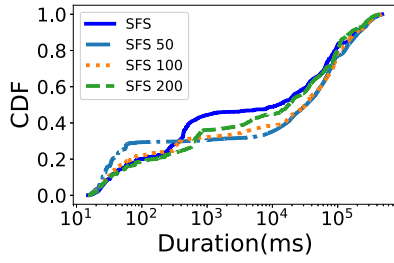
Fig. 9: Adaptive time slice tuning vs. statically fixed time slices.
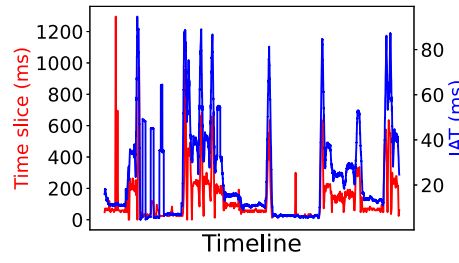
Fig. 10: Timeline of time slice changes vs. IATs during the whole workload.

Fig. 11: Handling I/O. SFS was configured to use different polling intervals.

fixed S to 50, 100, 200 and compared against SFS's dynamic adaptation heuristic. Figure 9 shows that none of the three statically configured S led to optimal performance. SFS's adaptive strategy yields better performance than a static S of 100 ms and 200 ms by adapting S based on the last 100 observed IAT samples. This is because a long, fixed time slice inevitably increased queuing delay of waiting functions.

Figure 10 depicts the timeline of the adaptation. Having a smaller, fixed S as short as 50 ms resulted in better performance for around 30% short function requests compared to SFS, but at the same time, it suffered from significantly prolonged duration for the rest of 70% requests. SFS struck a good balance of queuing delays and service time, leading to better overall performance.

Handling I/O. To evaluate how SFS handles I/O events in functions, we toggled the I/O knob for 75% of the function requests, for which we added a single I/O operation at the beginning of the function execution; the added I/O operation took X ms, where X was randomized drawn from a range between 10 to 100 ms. As shown in Figure 11, I/O-oblivious SFS had worse performance, because FILTER pool wasted time slice credit waiting for the I/O to be served, causing them to be filtered out to CFS. In contrast, SFS was able to detect I/O-caused waiting by using periodic status polling. We varied the polling interval from 1 ms to 8 ms and found that the performance was not sensitive to the polling frequency.

Handling Overload. We finally test the effectiveness of SFS's hybrid strategy to handle the transient overload. The Azure-sampled workload exhibits transient overloads, which can be spotted from the five queuing delay spikes shown in Figure 12(a). Note we only measured the time a function spent waiting in SFS' global queue. With overload detection disabled, SFS suffered significantly long queuing delays. Spiked queuing delays took long to diminish because the normal workload coming after the temporary load spikes caused a longer backlog of requests. By detecting the sudden increase of queuing delay, SFS temporarily switched to CFS. This helped quickly consume the backlog from the global queue so that normal load coming after the spikes can be served by using SFS's default FILTER pool. A direct effect is a smooth queuing delay curve (Figure 12(a)) and considerable reduction of turnaround times for about 50% of function requests (Figure 12(b)). More importantly, Figure 12(b) demonstrates that

CFS or FILTER policy alone (SFS with overload detection disabled) is not sufficient to handle transient overload; SFS's hybrid strategy effectively combines the best of both policies to achieve minimum turnaround time.

IX. OPENLAMBDA EVALUATION

A. End-to-End Efficiency

We used FaaSBench to generate a more comprehensive FaaS workload, which includes three function applications: **Fibonacci sequence** (`fib`), **markdown generation** (`md`), and **sentiment analysis** (`sa`). As mentioned, `fib` calculates a sequence of N Fibonacci numbers and is CPU-heavy. . `md` reads a JSON file from the function's local storage and transfers it to the markdown format; `md`'s execution is I/O-intensive. `sa` reads a file that contains a sentiment vocabulary dictionary and then predicts the sentimentality given a target sentence; `sa` is both CPU-intensive and I/O-intensive. This workload reused the same function duration distribution and IAT distribution of the Azure-sampled workload. OpenLambda was deployed to use 72 cores of the EC2 bare-metal instance, following an architecture depicted in Figure 5. We varied the IAT to generate three load levels: 80%, 90%, and 100%. The OpenLambda deployment introduced extra overhead at various levels, including the OpenLambda worker servers and the HTTP sandbox servers. These overheads diminished the performance benefits of SFS to some extent; however, as we will show, SFS can still provide huge performance improvement for the majority of functions that are short.

Figure 13 and 14 report the distributions of function execution duration and RTE. The functions ran on average 14.1% longer with OpenLambda+CFS under 80% load than OpenLambda+SFS under the same load. When the load increased, OpenLambda+CFS started to see performance degradation, while OpenLambda+SFS achieved almost identical performance under all the three loads. As shown in Figure 15, OpenLambda+SFS observed a 99th percentile duration of 4.75 seconds, a $1.65\times$, $4.04\times$, and $7.93\times$ speedup compared to OpenLambda+CFS under the load of 80%, 90%, and 100%, respectively. We also measured the number of context switches occurred under the three loads. Figure 16 shows the normalized context switches for each function request. Under the 80% and 100% load, more than 99% of function requests scheduled by CFS had more context switches than
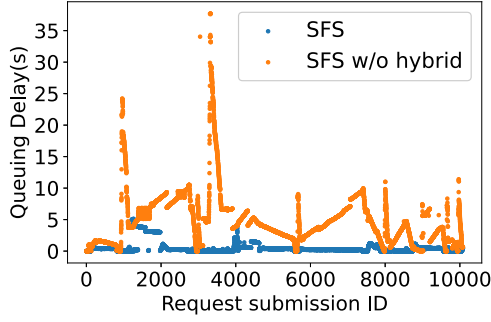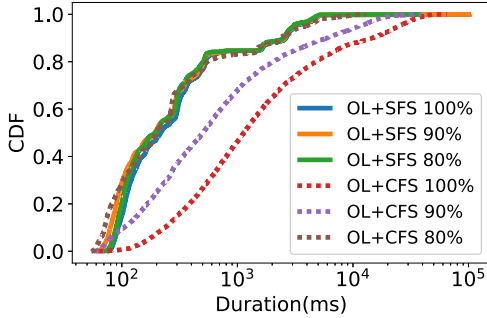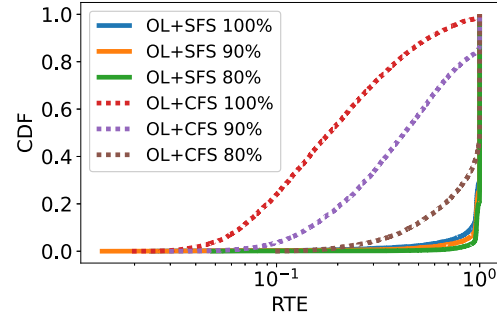
(a) Timeline of queuing delays.
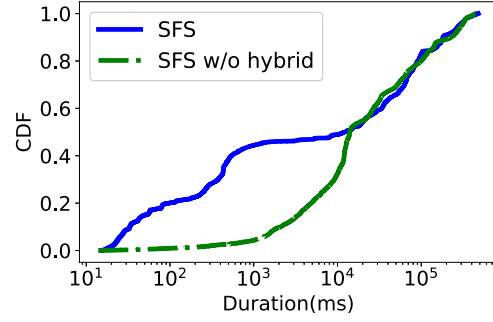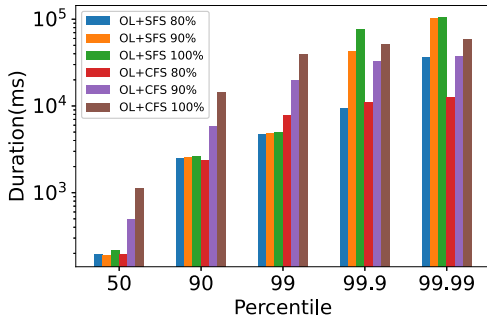

(b) CDF of function duration.

Fig. 12: Effect of SFS's overload handling mechanism. `SFS w/o hybrid` refers to SFS's baseline implementation with the hybrid FILTER+CFS mode disabled (see §V-E).
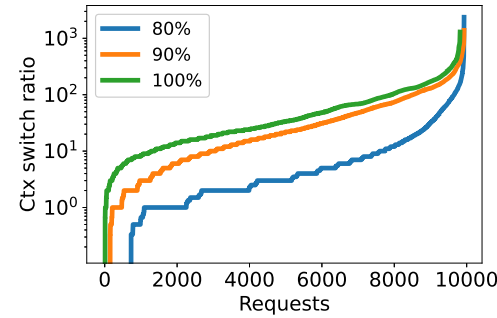

Fig. 13: Performance CDF.

Fig. 14: RTE CDF.

Fig. 15: Percentile breakdowns of function execution duration.

Fig. 16: The ratio of CFS context switches to that of SFS.

SFS. For about 85% of requests, CFS suffered $10\times$ more context switches than SFS.

B. SFS Overhead

SFS incurs a small runtime overhead. There are two sources of overhead: (1) SFS uses goroutines as scheduling workers: function scheduling incurs some overhead; (2) SFS workers perform periodic polling to check the kernel status of the function process. The polling overhead is the dominant overhead. Table II shows SFS' CPU usage in the 72-core OpenLambda tests. With a

TABLE II: SFS' (relative) CPU overhead in support of a 72-core OpenLambda deployment.

Interval	min	average	medium	max
1 ms	1.6%	3.8%	3.8%	6.2%
4 ms	1.3%	3.6%	4.0%	6.2%
8 ms	1.4%	3.4%	3.9%	6.6%

polling interval of 4 ms, SFS' average CPU usage was 259.8% for the Azure-sampled workload, meaning that an extra of 2.6 cores were needed in order to boost a 72-core OpenLambda deployment, a relative overhead of only $2.6/72 = 3.6\%$. About 74.4% of the total overhead was for periodic status polling, while the rest of 25.6% was for scheduling activities.

X. DISCUSSION

In this section, we discuss the limitations and possible future directions of SFS.

Impact of Function Cold Start. Significant function cold start costs may offset the benefit of SFS, especially for short functions. Optimizing the cold start cost of serverless functions is an important and challenging problem that has drawn great

attention from the community. Commercial FaaS platforms use sandbox and runtime caching extensively to mitigate the impact of cold start on function performance [62, 63]. The Azure Functions workload analysis [48] reports that even a naive keep-alive function warmup policy can guarantee zero cold start for around 50% of the function applications; with even smarter policies [48, 25, 46], the cold start rate could be further reduced to less than 10% for all the function requests served on a single function host server. We foresee that most if not all the function requests would be executed without a cold start penalty with the recent advancement in cold start optimization [22, 39, 50, 41, 42, 61]; this makes the OS-level function scheduling—the "last mile" of a function request—a practical and urgent research problem that demands effective solutions like SFS.

Why User-Space? SFS is designed to be a standalone, user-space function scheduler, which can be transparently plugged into existing FaaS platforms. While a kernel implementation of SFS would certainly work, with possibly less runtime overhead but much higher engineering efforts, a user-space implementation offers future-proof flexibility by retaining all the desirable properties of existing Linux scheduling facilities. With decades of research in datacenter workload co-location [59, 56, 40, 45, 26], soon we will see co-location of production FaaS workloads with other cloud computing workloads. CFS, as the battle-tested, general-purpose scheduling solution for a wide range of workloads, would still play a key role in balancing the CPU resource usage. SFS is designed to co-exist with and complement an existing OS scheduler in these scenarios. Moreover, co-location of highly diverse workloads is likely to cause more intense CPU contention, thus demanding future research.

XI. Related Work

Scheduling Short and Long Jobs. Improving turnaround time by approximating SRTF is a well-known approach that has been investigated in many domains [47, 24, 51, 30, 29]. A series of systems use request sizes as the hint to approximate SRTF. Size-based scheduling gives preference to requests for small files targeting web servers serving static HTTP requests [30]. Similarly, Harchol-Balter et al. applied SRTF to webserver request scheduling based on sizes of Linux kernel socket buffers [29]. Inspired by these works, SFS presents a practical priority scheduler that addresses many of the challenges in emerging, real-world FaaS workloads.

Scheduling for Fine-grained I/O Workloads. Shenango [43], Shinjuku [36], and ZygOS [44] use scheduling techniques such as core re-allocation, preemption, and work-stealing. These techniques optimize tail latency of small key-value requests whose service time is highly predictable; Shenango and Shinjuku assume long jobs co-located with small key-value request serving jobs—batch applications or range queries—whose application type is either known ahead or can be obtained from packet inspection. In contrast, SFS does not assume *a priori* knowledge about function types or execution time but instead requires a very small amount of historical statistics for online time slice adjustment.

Serverless Function Scheduling. Centralized, core-granular scheduling [38] uses two-level scheduling: it uses centralized scheduling to eliminate queue imbalance and core granular scheduling to reduce the interference caused by proportional-share. Core-granular scheduling assumes: (1) non-preemption, meaning a function, once scheduled to a worker core, runs to completion (i.e., running in FIFO), and (2) massive distributed resources, meaning the scheduler can always find available cores to schedule a function request. SFS shares similar goals but targets a local server scheduling environment, where the OS scheduling plays a critical role. Another line of work is focused on distributed or FaaS platform-level function scheduling [52, 53, 33, 57, 65] by using function placement optimization, low-latency I/Os, data locality, and reinforcement learning. Serverless dataflow frameworks use variants of cluster scheduling techniques [20, 19, 34, 49, 54] for serverless workflow applications. These works use the Linux scheduler for "last mile", OS-level task scheduling and would benefit from SFS.

User-defined Scheduling. Syrup [37] and ghOSt [32] allow developers to implement application-specific scheduling policies directly in the user space. Syrup uses the eBPF [6] maps data structure to support user-kernel communication, while ghOST uses message queues and transactions for user-kernel communication. A user-defined policy may, however, observe significant user-kernel communication cost if the application needs frequent user-level scheduling adjustment; this is the case for serverless scheduling, where the time slice needs to be frequently and dynamically tuned by the scheduler.

XII. Conclusion

Serverless computing promises fine-grained resource management, accounting, and billing at the milliseconds level. However, in practice, FaaS workloads are highly heterogeneous and latency-sensitive, and have shown great volatility in execution durations. In this work, we have shown, via the design and implementation of a user-space scheduler SFS and empirical evaluation, that SFS, by approximating SRTF scheduling, can significantly reduce the execution duration of short functions. SFS approximates SRTF with a dynamic and adaptive time slice in a first-level, global queue to combine the best worlds of FIFO and RR, while defaulting to the underlying OS-level scheduler in the second-level queue. SFS is transparent and can be easily ported to existing FaaS platforms. as we have demonstrated through an open-source FaaS platform OpenLambda. We hope that SFS will inspire new OS-level scheduling policies attuned to FaaS applications and open doors to new, FaaS-oriented SLO rules. SFS is available at: **https://github.com/ds2-lab/SFS**.

REFERENCES

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

[2] Amazon Web Services. https://aws.amazon.com/.

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

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

[5] Docker. https://www.docker.com/.

[6] eBPF Project. https://ebpf.io/.

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

[8] Google Cloud Platform. https://cloud.google.com/.

[9] gopsutil. https://github.com/shirou/gopsutil.

[10] Microsoft Azure Cloud. https://azure.microsoft.com/en-us/.

[11] OpenFaaS. https://www.openfaas.com/.

[12] sched(7) — Linux manual page. https://man7.org/linux/man-pages/man7/sched.7.html.

[13] The OpemLambda Project. https://open-lambda.org/.

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

[15] Xen Credit Scheduler. http://manpages.ubuntu.com/manpages/trusty/man8/schedtool.8.html.

[16] 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.

[17] Istemi Ekin Akkus, Ruichuan Chen, Ivica Rimac, Manuel Stein, Klaus Satzke, Andre Beck, Paarijaat Aditya, and Volker Hilt. SAND: Towards high-performance serverless computing. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 923–935, Boston, MA, 2018. USENIX Association.

[18] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. *SIGOPS Oper. Syst. Rev.*, 37(5):164–177, October 2003.

[19] 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.

[20] Benjamin Carver, Jingyuan Zhang, Ao Wang, and Yue Cheng. In search of a fast and efficient serverless dag engine. In *4th International Parallel Data Systems Workshop (PDSW 2019)*, 2019.

[21] Edward G Coffman Jr and Leonard Kleinrock. Computer scheduling methods and their countermeasures. In *Proceedings of the April 30–May 2, 1968, spring joint computer conference*, pages 11–21, 1968.

[22] Dong Du, Tianyi Yu, Yubin Xia, Binyu Zang, Guanglu Yan, Chenggang Qin, Qixuan Wu, and Haibo Chen. Catalyzer: Sub-millisecond startup for serverless computing with initialization-less booting. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '20, page 467–481, New York, NY, USA, 2020. Association for Computing Machinery.

[23] Dmitry Duplyakin, Robert Ricci, Aleksander Maricq, Gary Wong, Jonathon Duerig, Eric Eide, Leigh Stoller, Mike Hibler, David Johnson, Kirk Webb, Aditya Akella, Kuangching Wang, Glenn Ricart, Larry Landweber, Chip Elliott, Michael Zink, Emmanuel Cecchet, Snigdhaswin Kar, and Prabodh Mishra. The design and operation of CloudLab. In *Proceedings of the USENIX Annual Technical Conference (ATC)*, pages 1–14, July 2019.

[24] D. H. J. Epema. An analysis of decay-usage scheduling in multiprocessors. In *Proceedings of the 1995 ACM SIGMETRICS Joint International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '95/PERFORMANCE '95, page 74–85, New York, NY, USA, 1995. Association for Computing Machinery.

[25] Alexander Fuerst and Prateek Sharma. Faascache: Keeping serverless computing alive with greedy-dual caching. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS 2021, page 386–400, New York, NY, USA, 2021. Association for Computing Machinery.

[26] Panagiotis Garefalakis, Konstantinos Karanasos, Peter Pietzuch, Arun Suresh, and Sriram Rao. Medea: Scheduling of long running applications in shared production clusters. In *Proceedings of the Thirteenth EuroSys Conference*, EuroSys '18, pages 4:1–4:13, New York, NY, USA, 2018. ACM.

[27] Samuel Ginzburg and Michael J Freedman. Serverless isn't server-less: Measuring and exploiting resource variability on cloud faas platforms. In *Proceedings of the 2020 Sixth International Workshop on Serverless Computing*, pages 43–48, 2020.

[28] Jim Gray. Why do computers stop and what can be done about it?, 1985.

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

[30] 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.

[31] 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.

[32] 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*, SOSP '21, page 588–604, New York, NY, USA, 2021. Association for Computing Machinery.

[33] Zhipeng Jia and Emmett Witchel. Nightcore: Efficient and scalable serverless computing for latency-sensitive, interactive microservices. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS 2021, page 152–166, New York, NY, USA, 2021. Association for Computing Machinery.

[34] Eric Jonas, Qifan Pu, Shivaram Venkataraman, Ion Stoica, and Benjamin Recht. Occupy the cloud: Distributed computing for the 99In *Proceedings of the 2017 Symposium on Cloud Computing*, SoCC '17, pages 445–451, New York, NY, USA, 2017. ACM.

[35] Eric Jonas, Johann Schleier-Smith, Vikram Sreekanti, Chia-Che Tsai, Anurag Khandelwal, Qifan Pu, Vaishaal Shankar, Joao Menezes Carreira, Karl Krauth, Neeraja Yadwadkar, Joseph Gonzalez, Raluca Ada Popa, Ion Stoica, and David A. Patterson. Cloud programming simplified: A berkeley view on serverless computing. Technical Report UCB/EECS-2019-3, EECS Department, University of California, Berkeley, Feb 2019.

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

[37] 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.

[38] Kostis Kaffes, Neeraja J. Yadwadkar, and Christos Kozyrakis. Centralized core-granular scheduling for serverless functions. In *Proceedings of the ACM Symposium on Cloud Computing*, SoCC '19, page 158–164, New York, NY, USA, 2019. Association for Computing Machinery.

[39] Zijun Li, Linsong Guo, Quan Chen, Jiagan Cheng, Chuhao Xu, Deze Zeng, Zhuo Song, Tao Ma, Yong Yang, Chao Li, and Minyi Guo. Help rather than recycle: Alleviating cold startup in serverless computing through Inter-Function container sharing. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*, pages 69–84, Carlsbad, CA, July 2022. USENIX Association.

[40] Qixiao Liu and Zhibin Yu. The elasticity and plasticity in semi-containerized co-locating cloud workload: A view from alibaba trace. In *Proceedings of the ACM Symposium on Cloud Computing*, SoCC '18, page 347–360, New York, NY, USA, 2018. Association for Computing Machinery.

[41] Anup Mohan, Harshad Sane, Kshitij Doshi, Saikrishna Edupuganti, Naren Nayak, and Vadim Sukhomlinov. Agile cold starts for scalable serverless. In *11th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 19)*, Renton, WA, July 2019. USENIX Association.

[42] 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, Boston, MA, 2018. USENIX Association.

[43] 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, Boston, MA, February 2019. USENIX Association.

[44] 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.

[45] Charles Reiss, Alexey Tumanov, Gregory R. Ganger, Randy H. Katz, and Michael A. Kozuch. Heterogeneity and dynamicity of clouds at scale: Google trace analysis. In *Proceedings of the Third ACM Symposium on Cloud Computing*, SoCC '12, New York, NY, USA, 2012. Association for Computing Machinery.

[46] Rohan Basu Roy, Tirthak Patel, and Devesh Tiwari. Icebreaker: Warming serverless functions better with heterogeneity. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '22, page 753–767, New York, NY, USA, 2022. Association for Computing Machinery.

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

[48] 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. USENIX Association, July 2020.

[49] Vaishaal Shankar, Karl Krauth, Kailas Vodrahalli, Qifan Pu, Benjamin Recht, Ion Stoica, Jonathan Ragan-Kelley, Eric Jonas, and Shivaram Venkataraman. Serverless linear algebra. In *Proceedings of the 11th ACM Symposium on Cloud Computing*, SoCC '20, page 281–295, New York, NY, USA, 2020. Association for Computing Machinery.

[50] Wonseok Shin, Wook-Hee Kim, and Changwoo Min.

Fireworks: A fast, efficient, and safe serverless framework using vm-level post-jit snapshot. In *Proceedings of the Seventeenth European Conference on Computer Systems*, EuroSys '22, page 663–677, New York, NY, USA, 2022. Association for Computing Machinery.

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

[52] 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.

[53] 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.

[54] Ali Tariq, Austin Pahl, Sharat Nimmagadda, Eric Rozner, and Siddharth Lanka. Sequoia: Enabling quality-of-service in serverless computing. In *Proceedings of the 11th ACM Symposium on Cloud Computing*, SoCC '20, page 311–327, New York, NY, USA, 2020. Association for Computing Machinery.

[55] Markus Thömmes. Squeezing the milliseconds: How to make serverless platforms blazing fast! https://goo.gl/zvqtBP.

[56] Muhammad Tirmazi, Adam Barker, Nan Deng, Md E. Haque, Zhijing Gene Qin, Steven Hand, Mor Harchol-Balter, and John Wilkes. Borg: The next generation. In *Proceedings of the Fifteenth European Conference on Computer Systems*, EuroSys '20, New York, NY, USA, 2020. Association for Computing Machinery.

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

[58] Paul Turner, Bharata B Rao, and Nikhil Rao. Cpu bandwidth control for cfs. In *Proceedings of the Linux Symposium*, pages 245–254, 2010.

[59] Abhishek Verma, Luis Pedrosa, Madhukar Korupolu, David Oppenheimer, Eric Tune, and John Wilkes. Large-scale cluster management at google with borg. In *Proceedings of the Tenth European Conference on Computer Systems*, EuroSys '15, pages 18:1–18:17, New York, NY, USA, 2015. ACM.

[60] Carl A. Waldspurger and William E. Weihl. Lottery scheduling: Flexible proportional-share resource management. In *Proceedings of the 1st USENIX Conference on Operating Systems Design and Implementation*, OSDI '94, page 1–es, USA, 1994. USENIX Association.

[61] 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. USENIX Association, July 2021.

[62] Ao Wang, Jingyuan Zhang, Xiaolong Ma, Ali Anwar, Lukas Rupprecht, Dimitrios Skourtis, Vasily Tarasov, Feng Yan, and Yue Cheng. InfiniCache: Exploiting ephemeral serverless functions to build a Cost-Effective memory cache. In *18th USENIX Conference on File and Storage Technologies (FAST 20)*, pages 267–281, Santa Clara, CA, February 2020. USENIX Association.

[63] Liang Wang, Mengyuan Li, Yinqian Zhang, Thomas Ristenpart, and Michael Swift. Peeking behind the curtains of serverless platforms. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 133–146, Boston, MA, 2018. USENIX Association.

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

[65] 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.

Appendix: Artifact Description/Artifact Evaluation

SUMMARY OF THE EXPERIMENTS REPORTED

We evaluate SFS (standalone) and an SFS-port Openlambda as described in Section 8 and 9.

We set up two AWS EC2 VMs: a small, c5a.4xlarge VM with 16 vCPUs and 64 GB memory for standalone SFS, and a large, bare-metal, m5.metal EC2 instance with 96 vCPUs and 384 GB memory for SFS-ported OpenLambda. The goal of using a large bare-metal machine is to simulate a similar FaaS deployment environment used by major FaaS providers such as AWS. Given that there are 82,375 unique functions in Azure trace, we generate downsized FaaS workloads from Day 1, and we build a simple FaaSBench to generate about 10,000 Fibonacci (fib) function tasks. FaasBench is configurable. One can configure the behaviors of generated function workload using the following two configuration parameters: an integer parameter N for controlling the compute time of each fib function, and a boolean parameter IO that toggles the I/O operation (if set true) to simulate I/O activities.

We evaluate SFS standalone using various load levels. We change the average IAT ratio defined in the workload reading module to control overall CPU utilization. On the small VM (c5a.4xlarge) with 16 vCPU, we configure SFS standalone to use 12 vCPU. The workload configuration file specifies the function name, program, parameters, submission time, and invocation id for each function. For example, a workload record can be "fib1 fib.py 27 5 1". This simple configuration record encodes a Fibonnaci function job that executes fib.py with a parameter of 27 (i.e., calculate the 27th Fibonnaci sequence number) at time 1 (unit of millisecond). To enable variable load, we allow users to configure the load level by changing the IAT ratio in the readTrace module. We evaluate the performance under an IAT ratio from 7 to 13, which leads to average CPU utilization from 50% to 100%.

For sensitivity analysis, we optionally disable various optimization components in SFS, including IO handling, overhead handling, and time slice configuration. We provide details about how to disable each component on Github. For IO handling analysis, we use FaaSBench to generate IO-intensive function workload and evaluate performance under 3 levels of polling intervals: 1ms, 4ms, and 8ms. For overhead handling, we evaluate queuing delays with and without the hybrid strategy. To evaluate the efficiency of the adaptive time slice mechanism, we compare adaptive time slices with a fixed time slices ranging from 50ms, 100ms, to 200ms.

When evaluating SFS-port OpenLambda (OL), we focus on end-to-end efficiency and overhead. We evaluate an SFS-ported OL on a 96-vCPU VM (m5.metal), where we set up 64 OL workers for high concurrency. To reduce task request overhead, we build an HTTP client to submit function requests. We assign each OL worker a specific port ranging from 5003 to 5066. The client stores a map that maps the OL workers to ports and submits function jobs as configured in a provided trace file. Each OL worker is responsible for the same amount of tasks that run asynchronously. We configure 72 vCPU to be used by OL and set the IAT ratio from 1 to 3 to achieve an average CPU utilization between 50% and 100%. During the workload, we run trace-cmd to profile scheduled events. Overhead is monitored by go pprof tools.

AUTHOR-CREATED OR MODIFIED ARTIFACTS:

Artifact 1

Persistent ID: `https://github.com/ds2-lab/SFS`
Artifact name: SFS
Citation of artifact: fishercht1995. (2022). ds2-lab/SFS: test0.0 (test0.0). Zenodo. https://doi.org/10.5281/zenodo.6588689

Reproduction of the artifact with container: In the repository, we provide detailed instructions to build the experiment environment and reproduce evaluation experiments.

To get started, SFS requires the following dependencies:

(1) `Operating System: Ubuntu 20.04`
(2) `go version == 1.17.2`
(3) `schedtool version == v1.3.0`

Grant SFS scheduler the permission to access large-amount of files simultaneously by `ulimit -n 1024000` We provide a docker image that executes a sample of workloads under CFS and SFS, you would simple run the container by

```
docker run -privileged -name test -mount
type=bind,source="$(pwd)"/result,target=/result
fuyuqi1995/sfs
```

We also provide local SFS built by

```
go build && ./test.sh
```

Local SFS-port OpenLambda requires three sessions worked together

(1) install openlambda
```
make imgs/lambda  make install
./create.sh && python cp_default.py && python
replace.py
```
(2) initial SFS scheduler
```
go build  go run main.go
```
(3) submit a workload by HTTP client
```
go build  go run run.go
```

We have lots of sensitivity analysis on evaluation section, here we provide how we make configurations.

(1) Adaptive time slice. To disable Adaptive time slice you can comment out line 351 in SFS-standalone/sfs.go and you could configure a fix time slice by change initial value on line 337.
(2) Overhead handling. To disable hybrid strategy, you can comment out line 348 of SFS-standalone/sfs.go.
(3) IO handling. To disable IO handling, you can comment out line 352 in SFS-standalone/sfs.go
(4) IAT rate. You could configure IAT rate on line 45 in SFS-standalone/readTrace.go

Details of installing local SFS-standalone and SFS-port OpenLambda are available at

```
https://github.com/ds2-lab/SFS
```