A Case of Multi-Resource Fairness for Serverless Workflows

Amit Samanta
University of Utah
Salt Lake City, Utah, USA

Ryan Stutsman
University of Utah
Salt Lake City, Utah, USA

ABSTRACT

Serverless platforms have exploded in popularity in recent years, but, today, these platforms are still unsuitable for large classes of applications. They perform well for batch-oriented workloads that perform coarse transformations over data asynchronously, but their lack of clear service level agreements (SLAs), high per-invocation overheads, and interference make deploying online applications with stringent response time demands impractical.

Our assertion is that beyond the glaring issues like cold start costs, a more fundamental shift is needed in how serverless function invocations are provisioned and scheduled in order to support these more demanding applications. Specifically, we propose a platform that leverages the observability and predictability of serverless functions to enforce multi-resource fairness. We explain why we believe interference across a spectrum of resources (CPU, network, and storage) contributes to lower resource utilization and poor response times for latency-sensitive and high-fanout serverless application patterns. Finally, we propose a new distributed and hierarchical function scheduling architecture that combines lessons from multi-resource fair scheduling, hierarchical scheduling, batch-analytics resource scheduling, and statistics to create an approach that we believe will enable tighter SLAs on serverless platforms than has been possible in the past.

CCS Concepts

• **Computer systems organization → Cloud computing**.

Keywords

Serverless workflows, multi-resource management, fairness, SLA.

ACM Reference Format:
Amit Samanta and Ryan Stutsman. 2023. A Case of Multi-Resource Fairness for Serverless Workflows (Work In Progress Paper). In *Companion of the 2023 ACM/SPEC International Conference on Performance Engineering (ICPE '23 Companion), April 15–19, 2023, Coimbra, Portugal.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3578245.3585033

1 INTRODUCTION

In the past few years, serverless computing [22, 26, 31] has exploded. All major cloud providers offer serverless platforms that automatically deploy and scale stateless serverless functions in response incoming workload requests. It is popular, in large part, because it frees application developers from having to handle scaling, fault-tolerance, and resource provisioning problems. However, serverless hasn't been a panacea; these platforms are a step backward for applications that need strong performance guarantees, leaving many applications unable to use the serverless paradigm. Serverless platforms deploy tenants' applications on spare resources co-located with existing workloads and virtual machines, which leaves serverless functions to suffer performance interference and unfairness. Hence, serverless platforms must evolve before they can be used for applications with hard deadlines [1, 2], where job completion time is a priority, or which must generate real user-facing responses.

We make the case that the first step in doing that requires re-architecting the scheduling at the heart of serverless platforms to make fairness, resource allocation and interference, and resource-aware placement central. Our view is that fair and resource-aware scheduling need not come at the expense of efficiency and utilization. Instead, we speculate that cloud providers ignore key information that serverless applications expose that could lead to better placement decisions, strong and tighter service level agreements (SLAs) for serverless functions, and improved resource utilization. To that end, we describe a new platform for serverless workflows (Figure 1) that rethinks the flow of information between worker containers and the scheduler and the data that the scheduler uses in decision-making. Our design combines three key ideas.

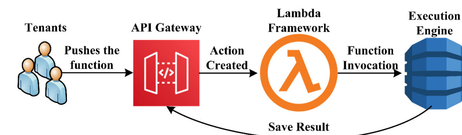


Figure 1: An overview of serverless workflow.

Observability. Serverless applications are inherently more observable than conventional cloud applications. Cloud providers can profile individual functions to understand their invocation patterns, execution times, and resource needs.

Predictability. Functions are short and mainly focus on a single task; combined with observability this means functions are predictable, often in their execution time, but also in their resource demands. By monitoring functions and placing them intelligently, a serverless scheduler can reduce interference, improve utilization and SLAs. Beyond that, in our design, the scheduler can use this predictability to determine whether a function's recent placements have led to new SLA violations and correct for mistakes.

Multi-resource Fairness. Finally, with these pieces, we believe it is time for serverless functions to guarantee their fair share of resources. Serverless applications should be ensured their share of

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
ICPE '23 Companion, April 15–19, 2023, Coimbra, Portugal
© 2023 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 979-8-4007-0072-9/23/04...$15.00
https://doi.org/10.1145/3578245.3585033

Resource Type	AWS	Google	Azure	Alibaba
CPU Bound	Worst	Average	Average	Best
Memory Bound	Average	Average	Worst	Best
I/O Bound	Worst	Average	Average	Best
Network Bound	Worst	Best	Average	Worst

Table 1: Performance of serverless functions with differing types of resource demands on cloud [27].

CPU and network resources, both locally on a machine and globally as a tenant. Hence, we describe how we enforce fairness with a multi-level fair queuing approach that enforces dominant resource fairness (DRF) among tenants; we also describe how the scheduler optimizes utilization subject to fairness while meeting SLAs.

In summary, this paper makes the following contributions:

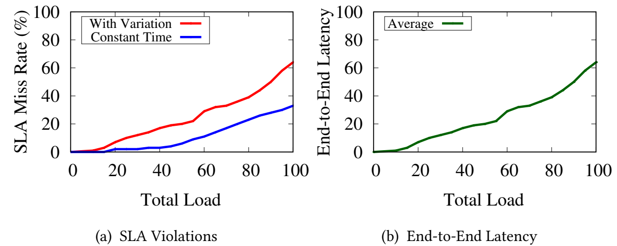
1) First, we demonstrate that variation in requirements for multiple resources leads to unfairness, SLA violations, and overall unsuitability for serverless applications.
2) Second, we outline the important challenges for multi-resource scheduler that is practically implementable and reduces SLA violations.
3) Finally, we outline a design for multi-resource scheduler for serverless workloads.

2 MOTIVATION

Many recent works have shown that the performance interference that is prevalent in cloud platforms [15, 21] impacts serverless workloads as well [12, 14]. Work that optimizes function placement across different cloud providers [7, 16] or across serverless instance types or regions [26] further reinforces that performance isolation and unfairness affect serverless workloads. For example, one recent work characterized serverless function performance on several cloud providers' serverless platforms and showed that performance and cost-effectiveness varies highly depending on which resources functions use most aggressively (CPU, memory capacity, storage I/O, and network I/O) [27]. Table 1 gives a high-level summary of their findings. Their results showed that no single cloud provider works best for all classes of functions. Depending on whether a function is compute, memory, I/O, or network bound, different platforms could give the best performance.

On some of these platforms, performance of one resource is correlated with provisioning decisions for another: for example, on Google and AWS platforms allocating more memory to a function also improves its CPU performance. Similar relationships hold for I/O; Google's I/O performance is better than Alibaba's, but only if functions are provisioned with at least 1,024 MB of memory.

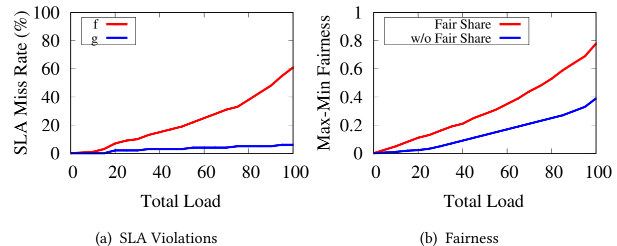
We suspected that this variation combined with a lack of a multi-resource scheduling approach in serverless leads to unfairness, violations, and serverless's overall unsuitability for many types of applications. To show this for ourselves, we designed a set of simulations that highlight how variation leads to unfairness and SLA violations. In these simulations, we simulate a scheduler that schedules serverless functions invocations at a worker. The worker runs its assigned invocations in turn. Our simulations assume each function that is invoked is warm, so the simulations include no cold start costs; all slow downs that an invocation experiences are due to queueing and contention for (CPU) resources.



(a) SLA Violations (b) End-to-End Latency

Figure 2: Analysis of SLA violations and latency.

Figures 2(a) and 2(b) show SLA violations for functions invocations as load scheduled to the worker increases (from 0% CPU utilization to 100%). As expected, violations increase as load increases, but the distribution of the runtime of the scheduled functions also plays a role. The red line in Figure 2(a) shows a case where the invoked functions vary between 5 to 10 ms CPU run time and the blue line shows when runtime is fixed to 8 ms for all functions. Variation in invocation CPU use increases the number of violations.



(a) SLA Violations (b) Fairness

Figure 3: Analysis of SLA violations and fairness.

However, are those violations distributed fairly among tenants? As one would suspect they are not. Figure 3(a) shows the fraction of SLA violations for two different functions, which use different amounts of CPU time per invocation: f uses more per invocation than g. Scheduled naively this leads to f consuming more than its fair CPU share resulting in increased SLA violations for g. In general, there is CPU unfairness among all of the functions that result in additional violations even when some functions aren't able to use their fair share. Figure 3(b) shows mix-max fairness [4] among all of the functions, which is low even as load increases. Using a fair CPU scheduler (e.g. proportional share) helps. The red line shows that when a function-aware fair scheduler is used, CPU unfairness due to function run time variation is reduced, resulting in less SLA violations for functions that consume less resources than others (like g).

Crucially, here we are showing this trend just for CPU resources; however, this same trend in Figure 3(a) bears out in all resources: CPU time, network transmit/receive, and storage I/O. Heavy-hitters for each of these types of resources are likely to interfere with functions that use these resources less. Worse, this interference is

additive with respect to tail latency: a stall to g due to h's network use compounds with a stall to g due to f's CPU use for some invocations. This leaves short, low-resource, latency-sensitive functions at a disadvantage, especially since their SLA is likely to be tight and defined in terms of a tail percentile (e.g. 99th-percentile).

Our assertion is that for serverless platforms to be more efficient, more fair, and viable for new classes of applications, schedulers for these platforms must be function- and invocation-aware, and they must enforce multi-resource fair policies that ensure that applications with sensitive SLAs are well isolated from resource-intensive functions.

3 KEY GOALS AND POSSIBLE APPROACHES

Our goal is to design a new serverless scheduler architecture that enables a wider variety of applications with more complex SLAs than are possible on today's serverless platforms. It should support strict latency bounds, including on tail percentiles, and high elasticity. It should be efficient both in terms of resources to the provider and in terms of cost to tenants, and it should centrally feature strong resource isolation via fair scheduling. However, building a scalable platform that provides multi-resource fair scheduling while supporting diverse sets of serverless applications is challenging, since applications have diverse SLA needs and metrics, programming models [32], inter-invocation strategies, task dependencies, and placement constraints.

Our design centers around three key principles:

1) Improved SLAs via Scalable Multi-resource Fair Scheduling. Different functions have different CPU, network, and I/O resource needs, and functions that have stringent SLAs must receive their fair share. Serverless schedulers must apply multi-resource fair principles like Dominant Resource Fairness [9] to avoid violations due to unfairness. This is challenging because functions can have rapid fan-out, scale-out, and scale-in, making tracking precise aggregate resource consumption at scale difficult. This is especially true of online workloads, which are inherently unpredictable and which crucially depend on tight tail response times to meet SLAs. Despite this, tenants should not have to significantly over-provision resources, and costs should be consistent regardless of the workload volume (tenants should be charged based on the work done).

Approach: To facilitate multi-resource scheduling, our plan is to use a multi-level approach to scheduling [23]. A centralized scheduler must make good global placement decisions among worker nodes, ensuring that the different resources of each worker are well-utilized and that the potential for interference between scheduled invocations is low. Internally, workers must also enforce fairness locally preventing short-term unfairness, and they must eventually report aggregate resource use back to the global scheduler. The key tension is that for scalability neither the global nor the local schedulers can afford to have perfect knowledge.

2) Function-resource-demand-aware Placement. Serverless applications are inherently decomposed into small units that can be observed and profiled independently. This extra visibility can be exploited to improve global placement if invocation resource needs can be reliably predicted. A key challenge is that not all functions behave predictably.

Approach. Serverless workers can observe and aggregate invocation resource usage, propagating that into a fuller picture of function behavior at a global function scheduler. Even treating functions as a black box, collecting empirical cumulative distribution functions (CDFs) of the resource consumption of different functions can help in scheduling by giving expected resource needs along with variance measures. Similarly, statistical tools (e.g. Kruskal–Wallis [17]) can help the scheduler determine if a function's resource needs are stationary or if, for example, recent interference has changed the runtime distribution of a function. Similar to Monotasks [20] but at a much shorter timescale, we expect it may make sense to break functions into "phases" where we model each invocation's resource uses as being one resource at-a-time and broken into discrete phases with individual CDFs. Using this information at the global scheduler, resource contention can be avoided at workers. This information also helps at the local worker level since invocations with long, heavy-hitter phases for specific resources can be isolated from invocations with short, bursty resource phases.

3) Optimized Latency. Many functions are small and executed in seconds or even milliseconds, so environment startup latency can dominate. We mainly focus on the startup latency when cold start happens. Reducing cold start time is a key challenge in serverless computing [6, 13, 18]. Also, the platform specific languages, workloads, package size, and memory affect the cold start time of function instance, which increases the latency.

Approach: The serverless workloads need stable average and tail latency, regardless of workload volume. Infrequent or sporadic workloads should not suffer from high cold-start latency, as the stateless services require stringent latency to meets SLAs. Further, by matching CDFs of different resource response times at workers to the CDFs of resource needs of the different phases of a function, scheduling can place invocations where they are likely to hit their SLA without resorting to minimizing queueing for all invocations, which can lead to later regret in scheduling decisions. Hence, we expect our approach will be able to meet SLAs than approaches that blindly minimize queuing or which only consider CPU resources when trying to meet deadlines.

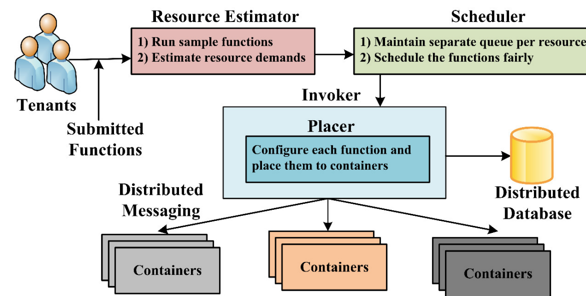


Figure 4: High level vision of resource framework.

4 A MULTI-RESOURCE FAIR ARCHITECTURE

Next, we describe our multi-resource fair serverless architecture (Figure 4). As multiple serverless functions can reside on a worker machines, hence the functions will contend for fair amount of resources. To enforce fairness, the platform enforces fair queuing

to schedule the functions according to each functions local weight. When a function is invoked, the invoker adds the function to a queue specific to a particular tenant. After the admission to a queue, each tenant gets its local and global weight to determine the fair share across multiple resources domains (i.e., CPU, memory, I/O and network). In each round, the tenants can get their weighted fair share of total available resources. By restricting the available resources per round, the scheduler can provide fairness within a timeframe. The weights should be determined based on the SLA of tenants. To enable SLA guarantees, the platform can translate a stated rate to a resource chunk, considering the total capacity of the system. According to the requirement of tenants, the platform can also reset the resource requirements of functions. To achieve the above goals, the system should (1) place a barrier between multiple resources to provide fair allocation, (2) enforce local weights to maintain global fairness, (3) optimize for good resource utilization, and (4) queue requests to enforce fairness.

4.1 Enabling Max-Min Fairness

To enable multi-resource fairness at each local worker, we should first compute the appropriate Dominant Resource Fair (DRF) [8, 9] shares, the scheduler (i.e., resource estimator) estimates the resource requirements of each tenant per worker machine. The estimator updates resource requirements periodically to provide good performance over time. It also tracks the per-function dominant resource to normalize the resource utilization and estimates the restricting DRF shares while computing the minimum value by taking the inverse of weighted sums. Our main aim is to provide DRF on a per-worker-machine level; however, we must also provide global max-min fair shares to tenants. The lower bound of a resource is set by the platform to ensure the fair share of a tenant's dominant resource. Hence, if there is variation in dominant resources across different workers, the accumulated share would provide equal or higher max-min fair share of the system's total resource capacity. The framework would allow us to consider a sole weighting factor to determine the global and local resource shares of tenants, alternately relying on complex weighted factors.

Hence, using the global max-min fairness, we would get better resource utilization while improving the predictability with near-optimal and effective resource sharing. Following the max-min fairness, none of the serverless function would gain any unfair advantage over other serverless services in the presence of higher system load, i.e., each serverless function would get its weighted fair share from the available resource pool. However, given the work-conserving nature, if some serverless functions utilize less resources than their allocated share, then underutilized resources should be divided across other functions to increase the utilization.

4.2 Resource Management

The resource management framework is comprised of three main pieces: a resource estimator, a near-optimal scheduler, and an adaptive function placer. First, the framework uses the resource estimator to get the characteristics of different serverless functions including their resource requirements. The main work of the scheduler is to manage which function to be scheduled next and to place it on a worker machine. Lastly, the placer should execute the functions scheduled in the system adaptively.

Resource Estimator. Our resource estimator design works in an online fashion where it knows the number of times a function has been invoked. The resource estimator estimates each function's resource demand and its execution times. The completion time on each resource is approximated by taking an average over all the samples collected for different functions. The completion times of functions are comparatively similar for same kinds of serverless applications. The estimator also estimates the memory demands of functions while observing the memory usage of their correlated samples. Current serverless platforms do not have any fine-grained resource allocation scheme for multiple serverless functions. Hence, we consider a fine-grained resource allocation process for serverless functions. The estimator considers the maximum number of cores in a CPU to maximize the resource utilization.

Scheduler. Current resource allocation schemes [10, 24, 25] and OpenWhisk [19] do not support fair resource allocation; hence, we cannot use them directly. Instead, we plan to implement a scheduler from scratch. As invocation requests arrive at the scheduler, they are routed through multiple queues for different resources. Among the queued invocations, the scheduler decides which invocation should be scheduled on which worker. The scheduler also aggregates resource usage statistics (including per-function per-resource resource use histograms) from the workers and execution time on CPU from the estimator. The scheduler tracks per-resource max-min fairness, and it considers the progress of all tenants over time. The scheduler applies admission control in case there are insufficient resources for some of the invocations, holding these invocations in a waiting queue awaiting notification of newly available resources (either due to a change in the worker pool or due to invocation completions at workers). We also cache the execution location of different functions to help meet SLA guarantees for frequently invoked functions. When a function is admitted, the scheduler checks the recent location cache for that function, and it returns the location if it is cached and the existing worker is not oversubscribed.

Finally, on completion the scheduler checks if a function gets its fair resource and finishes in time or not. If a function receives its allocated resources, the progress of that tenant is increased subsequently. However, if a function is executing with unsuitable configuration (with longer execution time), its position is discounted in the queue. If a function finishes in time, it skips this queue.

Function Placement. The resource management architecture has a central controller (CC) and several service placement units placed in worker nodes. Meta-operations are handled by CC, which provides detailed information about functions' run time and resource use. Execution of serverless functions are handled by containers spread across many worker machines. The service placement unit deals with both allocating resources (e.g. CPUs) to individual functions (function *placement*) and scheduling multiple functions over time. The function placer invokes theses functions in appropriate containers within workers and executes them following the scheduling strategy. In OpenWhisk, the containers are configured before invoking the functions to run on CPU; therefore, they do not require any kind of placer. However, we need an adaptive placer to maximize resource utilization. For optimal placement of serverless

functions, we might need to change some application-level configurations. A configuration function in their runtime must also monitor the available resources using our scheduler.

Finally, the placer will also implement preemption. Preemptive scheduling will be necessary to satisfy tight SLAs in many cases, since it allows the scheduler and placer to correct decisions as invocations arrive and to prevent starvation and unfairness. However, most textbook preemptive scheduling policies are merely designed for fair sharing, not minimizing SLA violations. Similarly, most schedulers aren't designed to have knowledge of resource needs, since, in general, it is difficult to predict the actual resource requirement of programs. A key goal is to design an SLA and resource-need-aware policy that combines ideas from fair scheduling with deadline-based and omniscient approaches like earliest deadline first and shortest remaining time first.

Flow and Objectives. The lifecycle from a tenant's perspective is to (1) upload a function to the system, then (2) send execution requests. The system performs function execution when requests are received, and transparently scales based on the workload demand. It distributes functions to one or more workers. It has three primary objectives (one tenant-centric and the other two provider-centric): *1) Minimize SLA violations:* SLAs should be met in aggregate for each function. *2) High resource utilization:* All the available resources should be utilized. *3) Fairness and Starvation freedom:* Functions should not starve or interfere with other functions beyond what their fair share allows.

Aside from those core objectives, the scheduler will also address these secondary concerns: *(i) Optimized Completion Time:* The execution time of serverless function is known to the scheduler; therefore, it can decrease the average completion time with good placement. *(ii) Efficient function consolidation:* Serverless functions should always be consolidated. While it is true that consolidated placement of a function may minimize its communication time, some serverless functions are insensitive to placement. *(iii) Overhead:* The current scheduling framework does not preempt functions because of large time overhead. Hence, it is important to design the scheduling policy in a way by which we can minimize the preemption overhead.

5 RELATED WORK

Among all the scheduling framework for serverless platforms, Caerus [30] is a task-scheduling framework for serverless analytics workloads which minimizes the execution cost and job execution time. Gsight [33] is an incremental learning predictor, which provides high precision by harnessing the spatial-temporal overlap codes and profiles the serverless functions for the scheduling algorithm. Similarly, FaasRank [28] provides a learning algorithm to schedule functions in serverless platforms. PASch [3] is a novel package-aware scheduling algorithm which tries to find package affinity so that worker nodes can re-use the execution environments. Skedulix [5] is a hybrid cloud scheduling framework for cost-efficient execution of serverless applications. Fifer [11] is an adaptive framework which is responsible for tackling the resource underutilization on serverless platforms. However, Fifer assumes only one resource type, while serverless functions require multiple resource types to be executed. Zhou *et al.* [34] proposed QoS-and-uncertainty-aware resource management framework for serverless workflows. Yu *et al.* [29] proposed a framework to accelerate serverless computing for harvesting idle resources. Unlike prior works, our system considers SLA-awareness and resource-aware placement, which are crucial for meeting serverless workloads objectives.

However, there is no prior work (as per our knowledge) that explores the potential of multi-resource scheduling for serverless workflows. Further, there is no existing serverless platform that supports multi-resource management for serverless workflows. Thus, there is a scope for designing multi-resource scheduler for serverless workflows which improves resource efficiency and reduces SLA violations.

6 FINAL THOUGHTS

Our initial simulator and prototype using OpenWhisk shows that the promise of muti-resource framework to enable high performance for challenging serverless applications and workloads. Following the prototype, we conducted some basic measurements to identify the variability and predictability in terms of execution time, resource usage and end-to-end latency for different functions. In the future, we aim to improve the prototype and implement our policies to answer the open research questions.

The main goal of this paper is to start dialogue on the importance of multi-resource fair resource allocation, and the unique problems serverless presents for fairness. We believe multi-resource fair scheduling is an important step forward to provide scalable performance for serverless platforms. We propose a hierarchical approach built around the observability and predictability of serverless functions that we believe can scale while providing more stringent SLAs than today's platforms, and we believe building such a system would open up new avenues for research.

REFERENCES

[1] Ahsan Ali, Riccardo Pinciroli, Feng Yan, and Evgenia Smirni. 2020. Batch: Machine learning inference serving on serverless platforms with adaptive batching. In *ACM/IEEE SC*. 1–15.
[2] Ahsan Ali, Riccardo Pinciroli, Feng Yan, and Evgenia Smirni. 2022. Optimizing inference serving on serverless platforms. *VLDB* 15, 10 (2022), 2071–2084.
[3] Gabriel Aumala, Edwin Boza, Luis Ortiz-Avilés, Gustavo Totoy, and Cristina Abad. 2019. Beyond load balancing: Package-aware scheduling for serverless platforms. In *IEEE/ACM CCGRID*. 282–291.
[4] Emilie Danna, Avinatan Hassidim, Haim Kaplan, Alok Kumar, Yishay Mansour, Danny Raz, and Michal Segalov. 2017. Upward Max-Min Fairness. *J. ACM* 64, 1, Article 2 (March 2017), 24 pages.
[5] Anirban Das, Andrew Leaf, Carlos A Varela, and Stacy Patterson. 2020. Skedulix: Hybrid cloud scheduling for cost-efficient execution of serverless applications. In *IEEE CLOUD*. 609–618.
[6] Dong Du, Tianyi Yu, Yubin Xia, Binyu Zang, Guanglu Yan, Chenggang Qin, Qixuan Wu, and Haibo Chen. 2020. Catalyzer: Sub-Millisecond Startup for Serverless Computing with Initialization-Less Booting. In *ACM ASPLOS*. 467–481.
[7] T. Elgamal. 2018. Costless: Optimizing Cost of Serverless Computing through Function Fusion and Placement. In *IEEE/ACM SEC*. 300–312.
[8] Ali Ghodsi, Vyas Sekar, Matei Zaharia, and Ion Stoica. 2012. Multi-Resource Fair Queueing for Packet Processing. In *ACM SIGCOMM*. 1–12.
[9] Ali Ghodsi, Matei Zaharia, Benjamin Hindman, Andy Konwinski, Scott Shenker, and Ion Stoica. 2011. Dominant Resource Fairness: Fair Allocation of Multiple Resource Types. In *USENIX NSDI*. 323–336.
[10] Jashwant Raj Gunasekaran, Prashanth Thinakaran, Nachiappan C. Nachiappan, Mahmut Taylan Kandemir, and Chita R. Das. 2020. Fifer: Tackling Resource Underutilization in the Serverless Era. In *ACM Middleware*. 280–295.
[11] Jashwant Raj Gunasekaran, Prashanth Thinakaran, Nachiappan C Nachiappan, Mahmut Taylan Kandemir, and Chita R Das. 2020. Fifer: Tackling resource underutilization in the serverless era. In *ACM Middleware*. 280–295.
[12] M. R. HoseinyFarahabady, A. Y. Zomaya, and Z. Tari. 2018. A Model Predictive Controller for Managing QoS Enforcements and Microarchitecture-Level Interferences in a Lambda Platform. *IEEE TPDS* 29, 7 (2018), 1442–1455.
[13] Eric Jonas, Johann Schleier-Smith, Vikram Sreekanti, Chia-Che Tsai, Anurag Khandelwal, Qifan Pu, Vaishaal Shankar, Joao Carreira, Karl Krauth, Neeraja Yadwadkar, Joseph E. Gonzalez, Raluca Ada Popa, Ion Stoica, and David A. Patterson. 2019. Cloud Programming Simplified: A Berkeley View on Serverless Computing. arXiv:1902.03383 [cs.OS]
[14] D. Kelly, F. Glavin, and E. Barrett. 2020. Serverless Computing: Behind the Scenes of Major Platforms. In *IEEE CLOUD*. 304–312.

[15] Ang Li, Xiaowei Yang, Srikanth Kandula, and Ming Zhang. 2010. CloudCmp: Comparing Public Cloud Providers. In *ACM IMC*. 1–14.
[16] Nima Mahmoudi, Changyuan Lin, Hamzeh Khazaei, and Marin Litoiu. 2019. Optimizing Serverless Computing: Introducing an Adaptive Function Placement Algorithm. In *ACM CASCON*. 203–213.
[17] Patrick E McKight and Julius Najab. 2010. Kruskal-wallis test. *The corsini encyclopedia of psychology* (2010), 1–1.
[18] Edward Oakes, Leon Yang, Dennis Zhou, Kevin Houck, Tyler Harter, Andrea Arpaci-Dusseau, and Remzi Arpaci-Dusseau. 2018. SOCK: Rapid Task Provisioning with Serverless-Optimized Containers. In *USENIX ATC*. 57–70.
[19] OpenWhisk. [n. d.]. https://openwhisk.apache.org/.
[20] Kay Ousterhout, Christopher Canel, Sylvia Ratnasamy, and Scott Shenker. 2017. Monotasks: Architecting for Performance Clarity in Data Analytics Frameworks. In *ACM SOSP*. 184–200.
[21] Thomas Ristenpart, Eran Tromer, Hovav Shacham, and Stefan Savage. 2009. Hey, You, Get off of My Cloud: Exploring Information Leakage in Third-Party Compute Clouds. In *ACM CCS*. 199–212.
[22] Mohammad Shahrad, Rodrigo Fonseca, Inigo Goiri, Gohar Chaudhry, Paul Batum, Jason Cooke, Eduardo Laureano, Colby Tresness, Mark Russinovich, and Ricardo Bianchini. 2020. Serverless in the Wild: Characterizing and Optimizing the Serverless Workload at a Large Cloud Provider. In *USENIX ATC*. 205–218.
[23] David Shue, Michael J. Freedman, and Anees Shaikh. 2012. Performance Isolation and Fairness for Multi-Tenant Cloud Storage. In *USENIX OSDI*. 349–362.
[24] A. Suresh, G. Somashekar, A. Varadarajan, V. R. Kakarla, H. Upadhyay, and A. Gandhi. 2020. ENSURE: Efficient Scheduling and Autonomous Resource Management in Serverless Environments. In *IEEE ACSOS*. 1–10.
[25] Ali Tariq, Austin Pahl, Sharat Nimmagadda, Eric Rozner, and Siddharth Lanka. 2020. Sequoia: Enabling Quality-of-Service in Serverless Computing. In *ACM SoCC*. 311–327.
[26] Liang Wang, Mengyuan Li, Yinqian Zhang, Thomas Ristenpart, and Michael Swift. 2018. Peeking Behind the Curtains of Serverless Platforms. In *USENIX ATC*. 133–146.
[27] Jinfeng Wen, Yi Liu, Zhenpeng Chen, Yun Ma, Haoyu Wang, and Xuanzhe Liu. 2021. Understanding Characteristics of Commodity Serverless Computing Platforms. arXiv:2012.00992
[28] Hanfei Yu, Athirai A Irissappane, Hao Wang, and Wes J Lloyd. [n. d.]. FaaSRank: Learning to Schedule Functions in Serverless Platforms. In *IEEE ACSOS*. 31–40.
[29] Hanfei Yu, Hao Wang, Jian Li, Xu Yuan, and Seung-Jong Park. 2022. Accelerating serverless computing by harvesting idle resources. In *ACM WWW*. 1741–1751.
[30] Hong Zhang, Yupeng Tang, Anurag Khandelwal, Jingrong Chen, and Ion Stoica. 2021. Caerus:{NIMBLE} Task Scheduling for Serverless Analytics. In *USENIX NSDI*. 653–669.
[31] Tian Zhang, Dong Xie, Feifei Li, and Ryan Stutsman. 2019. Narrowing the gap between serverless and its state with storage functions. In *ACM SoCC*. 1–12.
[32] Wen Zhang, Vivian Fang, Aurojit Panda, and Scott Shenker. 2020. Kappa: A Programming Framework for Serverless Computing. In *ACM SoCC*. 328–343.
[33] Laiping Zhao, Yanan Yang, Yiming Li, Xian Zhou, and Keqiu Li. 2021. Understanding, predicting and scheduling serverless workloads under partial interference. In *ACM SC*. 1–15.
[34] Zhuangzhuang Zhou, Yanqi Zhang, and Christina Delimitrou. 2022. AQUATOPE: QoS-and-Uncertainty-Aware Resource Management for Multi-stage Serverless Workflows. In *ACM ASPLOS*. 1–14.