Centralized Core-granular Scheduling for Serverless Functions

Kostis Kaffes
kkaffes@stanford.edu
Stanford University

Neeraja J. Yadwadkar
neeraja@cs.stanford.edu
Stanford University

Christos Kozyrakis
kozyraki@stanford.edu
Stanford University
Google

ABSTRACT

In recent years, many applications have started using serverless computing platforms primarily due to the ease of deployment and cost efficiency they offer. However, the existing scheduling mechanisms of serverless platforms fall short in catering to the unique characteristics of such applications: burstiness, short and variable execution times, statelessness and use of a single core. Specifically, the existing mechanisms fall short in meeting the requirements generated due to the combined effect of these characteristics: scheduling at a scale of millions of function invocations per second while achieving predictable performance.

In this paper, we argue for a cluster-level *centralized* and *core-granular* scheduler for serverless functions. By maintaining a global view of the cluster resources, the centralized approach eliminates queue imbalances while the core granularity reduces interference; together these properties enable reduced performance variability. We expect such a scheduler to increase the adoption of serverless computing platforms by various latency and throughput sensitive applications.

CCS CONCEPTS

• **Computer systems organization → Cloud computing**; *n-tier architectures*; • **Software and its engineering → Scheduling**.

KEYWORDS

cloud computing, serverless computing, scheduling, resource allocation

ACM Reference Format:
Kostis Kaffes, Neeraja J. Yadwadkar, and Christos Kozyrakis. 2019. Centralized Core-granular Scheduling for Serverless Functions. In *ACM Symposium on Cloud Computing (SoCC '19), November 20–23, 2019, Santa Cruz, CA, USA.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3357223.3362709

1 INTRODUCTION

Serverless computing platforms, such as AWS Lambda [2], Azure Functions [3], and Google Cloud Functions [4], simplify deployment of applications to cloud environments [21]. Users write functions, specify events as execution triggers, and pay only for the actual runtime of functions. Serverless computing platforms automate resource provisioning, scaling, and isolation.

The scalability and cost-efficiency of these systems has led to the development of frameworks that spawn a massive number of tasks for varied workloads, such as video processing [6, 16] and machine learning [20, 41]. Serverless functions exhibit the following unique properties: burstiness, short and variable execution times, statelessness and single-core execution. Despite their rise in popularity, existing serverless platforms lack deployment and scheduling mechanisms tailored to these characteristics of serverless applications.

As Wang et al. [45] pointed out, one limitation of existing serverless platforms is unpredictable performance that primarily arises due to their server-level scheduling granularity. Today, functions are assigned to servers as opposed to individual cores. Such coarse-grain function placement policies result in sharing of physical and virtual resources among function invocations, thereby causing performance variabilities [45]. Figure 1a depicts a coarse-grain server-level scheduling mechanism and Figure 1c shows a finer-grain core scheduling mechanism.

The next limitation of serverless platforms is their inefficient scalability [45] that arises because short lived functions are invoked in bursts by applications [15, 20, 25]. The scalability challenge stems from latency of scheduling, the queueing delay of functions waiting for resources to become available, and latency for deploying and initializing the required function binaries. This challenge of scale makes the use of distributed task-scheduling mechanisms [12, 31, 34, 37] appealing (Figure 1b). However, since such distributed mechanisms do not maintain a global view of the cluster resources they are forced to rely on task migration to handle any work imbalance. This renders them unsuitable for serverless functions that can have high start-up overheads [14]. Section 2 details our analysis of existing scheduling mechanisms.

Finally, the combination of extreme parallelism and large scale required by today's serverless applications, such as video analytics [6, 16], and ML inference [39, 46], makes shorter end-to-end tail latency a key requirement [10, 15, 25]. Performance variability and limited scalability hinders the adoption of serverless computing for emerging and existing latency sensitive applications.

To alleviate these issues, we argue for a cluster-level scheduler that is *centralized* and operates at *core-granularity* as shown in Figure 1c. Being *core-granular*, the scheduler assigns functions directly to individual cores. It thus improves performance predictability by explicitly managing the sharing of individual cores among simultaneously executing functions. The *centralized* design of the cluster-level scheduler maintains a global view of cluster resources that enables it to assign work to any core in the cluster. This design thus improves scalability and elasticity by eliminating overloading of cores and any queue imbalances. It also eradicates the need for heavy-weight function migrations away from busy cores.

SoCC '19, November 20–23, 2019, Santa Cruz, CA, USA

ACM ISBN 978-1-4503-6973-2/19/11…$15.00
https://doi.org/10.1145/3357223.3362709

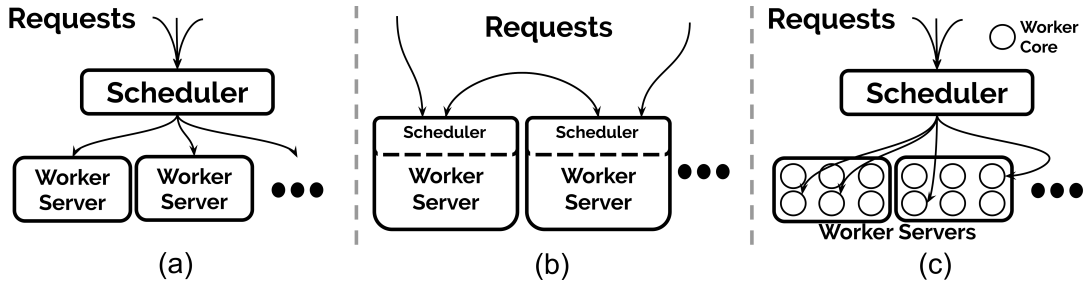

Figure 1: (a) Centralized server-granular scheduling (b) Distributed server-granular scheduling (c) Proposed *centralized core-granular* scheduling.

In this paper, we propose the design of such a centralized and core-granular scheduler that enables millions of function invocations per second. In the last section, we discuss how such a scheduler enables further optimizations in various aspects of serverless computing.

2 MOTIVATION

This section identifies the gap between what existing schedulers for serverless functions offer and the expectations of applications as the need for managed services continues to evolve. We focus on unique characteristics of serverless applications that motivate the need for a cluster-level centralized and core-granular scheduler.

2.1 Typical serverless functions

Common workloads deployed on serverless platforms, such as video transcoding, compilation, and machine learning, display the following characteristics that together motivate the need to revisit the scheduler design.

Burstiness: Workloads can fluctuate their degree of parallelism by several orders of magnitude in a matter of seconds [15, 20, 25].

Short but variable execution times: Functions typically live in the range of a few hundreds of milliseconds to a few minutes, and are billed at sub-second granularity. AWS Lambda limits function executions to 15 minutes and bills users at a 100ms-granularity [2].

Low or no intra-function parallelism. Applications exploit parallelism offered by serverless platforms by launching multiple functions simultaneously. However, each function executes on a single or at most two CPUs.

Statelessness: Application state is usually stored remotely and hence locality does not matter.

There are other platforms that have some of these characteristics; however, none combines all of them. For example, scientific computing [37] and analytics [34] frameworks usually break jobs down to small tasks similar to serverless functions. Being data-intensive, achieving data locality and leveraging intra-instance parallelism is critical to performance. Hence, a scheduler that queues tasks or moves them between machines so that they execute close to their corresponding data is beneficial. However, there is no reason for stateless serverless functions to suffer delays caused by the wait involved for specific execution environments. Typical long-running cloud workloads, such as web search, may experience bursts but can afford to react to them by dynamically provisioning resources. Thus, start-up and scheduling latencies affect such workloads less severely compared to the short-lived serverless functions.

It is through these characteristics that important performance metrics for serverless functions emerge. The most important is *elasticity;* any serverless system must be able to spawn a large number of functions in a short period of time. Second, function *start-up latency*, including scheduling and instantiation overheads, must be kept low and predictable. Finally, *high cost efficiency* is always necessary in a cloud provider setting.

2.2 Limitations of existing schedulers

We now discuss how existing mechanisms used by general purpose task schedulers and cloud providers fall short in accommodating the needs of serverless functions.

2.2.1 General purpose distributed schedulers. Centralized scheduling approaches are proven to perform better than distributed scheduling algorithms [24]. For instance, an M/M/n queue is provably better than n×M/M/1 queues in terms of tail latency. However, in practice, a centralized scheduler often becomes the bottleneck [34] at large scale. Existing centralized schedulers avoid this bottleneck either by opting for coarse granularity [44] or by making static scheduling decisions based on prior knowledge of task execution graphs [43]. To avoid these limitations, most modern frameworks have adopted a distributed approach [30, 31, 34, 37].

Sparrow [34] achieves low latency scheduling for data analytics frameworks by balancing load across machines using power-of-2 choices. Canary [37] claims that synchronous controller-worker communication is the hurdle for achieving low-latency scheduling. Canary proposes an asynchronous control plane where a central controller tells worker nodes how to redistribute data and therefore work, while workers locally decide when to redistribute. To support fine-grain computations (10 milliseconds) at high throughput (1.8M tasks per second) for reinforcement learning workloads, Ray [31] uses a two-level scheduler; per-node local schedulers that can also forward tasks to a global scheduler. Despite the differences, these frameworks share a common architecture: Tasks are assigned to a server using a simple load balancer while imbalances are detected and handled by per-machine scheduling agents that migrate tasks away from busy servers.

However, such hierarchical scheduling mechanisms are not suitable for serverless functions for two key reasons. First, traditional

task scheduling frameworks assume that the execution environment is already set-up on all servers of a cluster. In serverless systems, source code, binaries and other dependencies need to be fetched remotely, and an isolated execution environment needs to be initialized on each function invocation. Thus, serverless functions tend to incur a start-up cost [45] that makes migration unappealing. Second, the range of execution times of serverless functions can be large, from 100ms to 15min, i.e., 4 orders of magnitude. Such variability leads to queue imbalances that (a) are hard to detect, and (b) may lead to spurious task migrations. For example, even the lower variability associated with exponential execution times has been shown to lead to migration rates as high as 50% [35]. Distributed frameworks cannot handle such migration rates primarily due to the cost associated with finding a candidate machine and migrating a serverless function to it [34].

2.2.2 Existing schedulers for serverless platforms. Details of schedulers implemented by existing serverless platforms are not available publicly. However, open-source platforms, such as Apache OpenWhisk [1], OpenLambda [5], and efforts to reverse engineer AWS Lambda, Google Cloud Functions, and Azure Functions [45], reveal some of the internal workings of these platforms. We note some of these findings motivating the need for a new scheduling mechanism.

Incoming function invocations are assigned to individual servers for execution by a load balancer. We refer to such server-level scheduling as coarse-gain scheduling. Each server informs the load balancer of its readiness to receive more work at a regular time interval. This approach suffers from issues of performance isolation. For example, AWS Lambda packs containers running function invocations on VMs with the aim of improving resource utilization. This coarse-grain scheduling tends to introduce contention for shared resources, and hence results in *performance variability* [45].

Moreover, such load balancers limit function instantiations [45]. Azure can only scale to 200 function instances and Google Functions enqueue functions and delay instantiation even for lower concurrency levels. This degree of parallelism is not sufficient for several embarrassingly-parallel workloads such as compilation [15], analytics [20], and video encoding [16]. AWS Lambda scales to a higher number of concurrent invocations but can take a few seconds to spawn a few thousand function instances [20]. Such *high start-up latency* severely limits the ability of applications to react to naturally occurring burstiness or flash crowds [32].

3 SCHEDULER FOR SERVERLESS SYSTEMS

The analysis of existing scheduling frameworks in Section 2 has revealed two key requirements for a serverless function scheduler. It needs to prevent: (a) load imbalances and (b) inter-function interference. These issues have been studied in operating systems research [22, 33, 36]. *Centralized scheduling* has shown to reduce queueing and load imbalances [22]. *Core-granular scheduling* can significantly reduce interference [33, 36].

Benefits and implications of core-granular scheduling: Core-granular scheduling enables us to choose a point in the interference-utilization trade-off space. For example, it can avoid (a) scheduling delays due to time-sharing and (b) high-level cache pollution, by ensuring that no two functions are co-located on the same core. [1] On the other hand, to achieve higher utilization, the scheduler can decide to pack multiple functions on a single core. We could also adopt a more complex approach where latency-tolerant batch functions can be packed together and are susceptible to temporary delays while latency-sensitive functions are assigned to their own dedicated core.

Benefits and implications of centralized schedulers: Global view of cluster resources can enable centralized schedulers to harvest unused resources and offer higher elasticity to users, both in terms of increasing the number of concurrent requests and reducing the start-up latency. Moreover, a centralized design can easily adapt to changing requirements, such as support for new hardware, different types of resource allocations, and varying function execution times.

Synergy of centralization and core-granularity: A global core-granular scheduler can assign function instances directly to available cores, thereby reducing queueing time. This also eliminates the need to migrate tasks from one queue to another. However, the proposed centralized scheduler will be operating at a coarser time granularity than the OS-level CPU scheduler. This can potentially lead to resource under-utilization. We discuss how to address this issue and harvest spare resource capacity in section 4.4.

The challenge of scale: To understand the requirements imposed by the underlying infrastructure, let us consider an example of a cluster of 20,000 servers with 32 cores each. Assuming the billing granularity of 100 milliseconds, we need to schedule 20,000×32×10= 6.4M function instances per second. Achieving such a high request rate using a single scheduler is particularly challenging. The time spent on each scheduling decision must be very short while the scheduling logic needs to be parallelizable. That is why the traditional approach to meeting such requirements is to use distributed mechanisms [34, 37] whose inherent need for migrations makes them unsuitable for serverless platforms.

3.1 A centralized core-granular scheduler

In this section, we propose a design for a centralized core-granular scheduler for serverless platforms. Figure 2 summarizes the key components of the scheduler and its workflow. The centralized scheduler runs on a multicore server. We distinguish between cores on the scheduler server, called scheduler cores, and cores on worker servers, called worker cores. Each scheduler core maintains a list of references to idle worker cores. Incoming requests are assigned to one of the scheduler cores **(i)** using a load balancing mechanism, e.g., RSS [40]. The scheduler core takes an idle worker core out of its local list **(ii)** and schedules the function instantiation request to that core **(iii)**. When the function finishes execution, the scheduler is notified and the worker core is added to a scheduler core's list **(iv)**; not necessarily the same which originally scheduled the function. If a request arrives to a scheduler core whose worker core list is empty **(v)**, it can steal a worker core from the list of a different scheduler core **(vi)** and schedule the function for execution there **(vii)**.

[1] For complete performance isolation, mechanisms orthogonal to ours [28] can be used to mitigate memory bandwidth, last-level cache, and network interference.

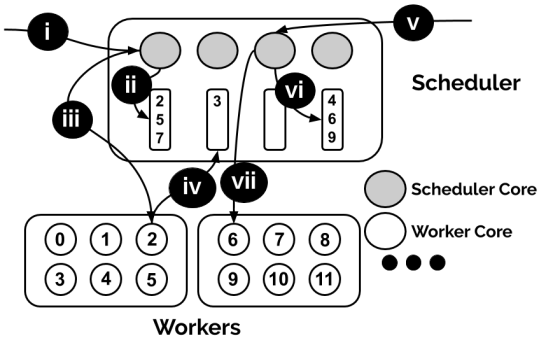

Figure 2: Scheduler design.

This design is work-conserving, i.e., functions ready for execution are directly assigned to idle worker cores avoiding unnecessary queueing, eliminating queue imbalances, and reducing start-up latency. Instead of handling imbalances by work stealing and function migrations between machines, we avoid them altogether by stealing references to idle cores within a single machine that executes the scheduling logic. It thus avoids overloading a core when other idle cores exist. Moreover, this design can overcome the scalability challenges and spawn a large number of functions in a short period of time. Scheduling decisions are reduced to the removal of an element from a list while the lack of need for inter-thread communication in the common case allows scaling to a large number of scheduler cores.

3.2 Discussion

Implementation: A cluster-scale centralized scheduler needs to scale to millions of requests per second. To enable scheduling at this scale, we propose leveraging the mechanisms developed for high-performance dataplane operating systems [9, 23]. To the best of our knowledge, this is the first attempt to use such mechanisms for a middle-tier service, such as a centralized scheduler.

We focus on two of these mechanisms. IX [9] and eRPC [23] show that the distribution of packets from the NIC to the cores using RSS can scale to high request rates and core counts. This scalability is achieved by ensuring that the work performed by a scheduler core for each function invocation is homogeneous; it does not depend on the type of the invoked function or have any variability. The other performance-critical operation is the stealing of worker cores from scheduler cores' queues. This translates to the need for high-performance single-producer, multiple-consumer queues. Such data structures are capable of handling tens of millions of requests per second, satisfying our requirements [13, 38, 48].

Fault Tolerance: The system by design is not fault-tolerant. We leave the implementation of fault tolerance to frameworks running atop the scheduler [15, 20]. The number of function requests affected by the failure of any worker server is at most equal to the number of cores of the worker. If the scheduler server fails, the only state a new scheduler server needs to recover is the list of idle worker cores. This can be achieved by exchanging a few bytes with each worker server.

Multicore Functions: Commercial offerings allow serverless functions to use up to 2 cores [2, 4]. While our scheduler is designed for single core functions, it can be extended to support multicore ones with minor modifications, such as maintaining a second list to keep track of pairs of idle worker cores on the same machines at each scheduler core. A scheduler core uses this list for multicore function requests. Using this design, we can accommodate future serverless functions with higher core counts by increasing the number of lists per scheduler core.

4 RESEARCH DIRECTIONS

The scheduler's centralized design and per-core scheduling capabilities offer ways to handle some of the open problems of serverless computing platforms.

4.1 Security

Multi-tenant environments need to provide isolation between different users and simultaneously executing instances [11]. The traditional choices for isolation mechanisms have been VMs and containers. Containers are light-weight but face security issues [17, 19], while VMs have higher instantiation cost and memory footprint [29] than containers. There is a large body of work that attempts to achieve the performance of containers and the security offered by VMs [14, 18, 29]. gVisor [18] implements a large part of the Linux kernel's functionality in userspace. While this approach minimizes the attack surface compared to regular containers, it incurs high start-up and runtime overheads [47]. Amazon's Firecracker [14] can launch a very simple virtual machine in as little as 125ms and is used for AWS Lambda today. Still, if our eventual goal is to move to 10ms tasks, start-up time needs to be one to two orders of magnitude shorter than that. LightVM [29] achieves start-up times as low as 2.3ms by optimizing the Xen supervisor's control plane and by building minimalistic application-specific VMs based on unikernels. While it achieves the required performance, the need to build a custom VM image for each different cloud function makes it impractical for serverless platforms.

Our core-granular scheduling approach simplifies the design of a secure runtime system on the worker servers. The fact that the scheduler explicitly controls how many function instances can run concurrently on a worker core eliminates many attack vectors. For example, in the case where only one function runs on each core we can create a single long-running VM per worker core, use that as a mechanism to do resource assignments (e.g., cpumask, memory), and achieve inter-function isolation. Furthermore, we can enforce isolation between the different functions that execute in that VM over time by keeping a clean "copy" of the VM and reverting back to it after each function invocation finishes execution.

We could also explore the use of hardware virtualization extensions and implement functionality similar to Dune's sandbox [8]. Isolation would be enforced through privilege modes by executing the runtime in ring 0 and the untrusted function code in ring 3. However, there still are open questions regarding the vulnerability of such an isolation mechanism to side-channel attacks.

4.2 Coldstart latency

A significant performance bottleneck for existing serverless platforms is the time taken to deploy the function along with any dependencies it may have. We refer to this time as the coldstart

latency. Given the lack of publicly available details about how the cloud providers deploy user-defined functions to their clusters, we can only speculate using the findings in recently published literature [32, 42, 45]. Existing platforms are assumed to run a container image hub and pull the image corresponding to the user's function on the VM that is scheduled to run the function. As a result of this, the median (110ms - 3.6sec) and the maximum (1.4sec - 45.7sec) coldstart latencies are high [45]. This has led to attempts to reduce this latency either by trimming down the image size [32] or by fetching dependencies on-demand at runtime [42].

With the proposed centralized core-granular scheduling, function code can run as a regular process, sandboxed using hardware virtualization extensions (see Section 4.1). Therefore, we do not need to fetch a full operating system image; we just need the function code and dependencies. The function code can thus efficiently be shipped to each server on-demand. Moreover, the set of most commonly used libraries for various supported languages can be cached on each server, and the rest can be fetched on-demand. Deciding which dependencies should be cached, and for how long, is an open question. Exploring a peer-to-peer approach may provide a scalable solution for dependency fetching.

The per-core scheduling approach allows for additional optimizations. The cost of initializing the language runtime contributes significantly to coldstart latency [32]. Following an approach similar to the one we discussed in Section 3.2, each scheduler core can have multiple worker core lists. Each list contains cores with "warm" runtime environments for a different language or framework, e.g., Javascript, Python, or Java cores. Function instances are then scheduled to the appropriate core types that do not require runtime initialization. The same approach can be extended to implement existing optimizations such as per-application "warm" containers [7]. This way most of the remaining coldstart latency can be eliminated.

4.3 Naming and addressing system

The emergence of stateful workloads in serverless platforms leads to the generation of data, both ephemeral [25] and long-term [31], that need to be transferred between function instances. Existing serverless platforms do not support direct dynamic communication between function instances; user code can only initiate but not listen for network connections. Moreover, discovering the IP address of a specific function instance and communicating with it directly is challenging for an external service and/or a different function instance. Multi-stage jobs that require data sharing between different function instances need to use paid services such as S3, ElastiCache, or Pocket [26]. Since direct communication between function instances is cheaper and provides lower latency than using an intermediate storage system, some frameworks have used rendezvous servers to side-step these networking limitations [16].

A centralized scheduler simplifies the issue of naming and addressing for cloud functions. The scheduler saves the function-to-core mapping each time a function is scheduled to a core. Function instances can query this mapping using the id of the function instance they want to communicate with and get its IP address. However, such point-to-point inter-function communication cannot accommodate all communication patterns. It is unclear how to implement more complex group communication primitives such as broadcast or gather [27].

4.4 Resource sharing

Sharing resources between serverless functions and other workloads is the key to achieving cost-efficient cloud computing. There is a need for mechanisms that can determine the partitioning of cores between serverless functions and other cloud workloads.

Such mechanisms can use history [49] or simple feedback mechanisms [28]. A mechanism can communicate the core allocation decision to the scheduler as follows: if a core is assigned to the serverless platform, the serverless runtime system on that core is activated and sends a notification to the scheduler that the core is idle and waiting for work. If a core executing a serverless function is assigned to some other workload, the runtime can either wait for the function instance to finish executing or preempt it. Then, it notifies the scheduler that the core is no longer available and sleeps.

5 CONCLUSION

The design and implementation of existing serverless platforms does not allow them to unleash the full potential of serverless computing. In this paper, we propose a new cluster-level scheduler for serverless functions which, departing from existing paradigms, is both centralized and core-granular. In addition to improving elasticity and reducing interference, this scheduler opens up exciting new research avenues in serverless computing.

ACKNOWLEDGMENTS

We thank our shepherd, Raul Castro Fernandez, and the anonymous SoCC reviewers for their helpful feedback. We also thank Francisco Romero, Qian Li, and Ana Klimovic for providing feedback on early versions of this paper. This work was supported by the Stanford Platform Lab.

REFERENCES

[1] 2019. Apache OpenWhisk. (2019). Retrieved June 2, 2019 from https://openwhisk.apache.org/
[2] 2019. AWS Lambda. (2019). Retrieved June 5, 2019 from https://aws.amazon.com/lambda/
[3] 2019. Azure Functions. (2019). Retrieved June 5, 2019 from https://azure.microsoft.com/en-us/services/functions/
[4] 2019. Google Cloud Functions. (2019). Retrieved June 5, 2019 from https://cloud.google.com/functions/
[5] 2019. OpenLambda. (2019). Retrieved June 2, 2019 from https://open-lambda.org/
[6] Lixiang Ao, Liz Izhikevich, Geoffrey M. Voelker, and George Porter. 2018. Sprocket: A Serverless Video Processing Framework. In *Proceedings of the ACM Symposium on Cloud Computing (SoCC '18)*. ACM, New York, NY, USA, 263–274. https://doi.org/10.1145/3267809.3267815
[7] AWS Lambda 2018. A Serverless Journey: AWS Lambda under the hood. (2018). Retrieved September 24, 2019 from https://youtu.be/QdzV04T_kec
[8] Adam Belay, Andrea Bittau, Ali Mashtizadeh, David Terei, David Mazières, and Christos Kozyrakis. 2012. Dune: Safe User-level Access to Privileged CPU Features. In *Presented as part of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*. USENIX, Hollywood, CA, 335–348. https://www.usenix.org/conference/osdi12/technical-sessions/presentation/belay
[9] Adam Belay, George Prekas, Ana Klimovic, Samuel Grossman, Christos Kozyrakis, and Edouard Bugnion. 2014. IX: A Protected Dataplane Operating System for High Throughput and Low Latency. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*. USENIX Association, Broomfield, CO, 49–65. https://www.usenix.org/conference/osdi14/technical-sessions/presentation/belay

[10] Jeffrey Dean and Luiz AndrÃĬ Barroso. 2013. The Tail at Scale. *Commun. ACM* 56 (2013), 74–80. http://cacm.acm.org/magazines/2013/2/160173-the-tail-at-scale/fulltext

[11] Christina Delimitrou and Christos Kozyrakis. 2017. Bolt: I Know What You Did Last Summer... In The Cloud. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '17)*. ACM, Xi'an, China, 599–613. https://doi.org/10.1145/3037697.3037703

[12] Christina Delimitrou, Daniel Sanchez, and Christos Kozyrakis. 2015. Tarcil: Reconciling Scheduling Speed and Quality in Large Shared Clusters. In *Proceedings of the Sixth ACM Symposium on Cloud Computing (SoCC '15)*. ACM, Kohala Coast, Hawaii, 97–110. https://doi.org/10.1145/2806777.2806779

[13] Aleksandar Dragojević, Rachid Guerraoui, and Michal Kapalka. 2009. Stretching Transactional Memory. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '09)*. ACM, Dublin, Ireland, 155–165. https://doi.org/10.1145/1542476.1542494

[14] Firecracker 2018. Firecracker - Lightweight Virtualization for Serverless Computing. (2018). Retrieved May 23, 2019 from https://aws.amazon.com/blogs/aws/firecracker-lightweight-virtualization-for-serverless-computing

[15] Sadjad Fouladi, Francisco Romero, Dan Iter, Qian Li, Shuvo Chatterjee, Christos Kozyrakis, Matei Zaharia, and Keith Winstein. 2019. From Laptop to Lambda: Outsourcing Everyday Jobs to Thousands of Transient Functional Containers. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*. USENIX Association, Renton, WA, 475–488. https://www.usenix.org/conference/atc19/presentation/fouladi

[16] Sadjad Fouladi, Riad S. Wahby, Brennan Shacklett, Karthikeyan Vasuki Balasubramaniam, William Zeng, Rahul Bhalerao, Anirudh Sivaraman, George Porter, and Keith Winstein. 2017. Encoding, Fast and Slow: Low-Latency Video Processing Using Thousands of Tiny Threads. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*. USENIX Association, Boston, MA, USA, 363–376. https://www.usenix.org/conference/nsdi17/technical-sessions/presentation/fouladi

[17] Aaron Grattafiori. 2016. Understanding and Hardening Linux Containers. (2016). Retrieved May 28, 2019 from https://www.nccgroup.trust/us/our-research/understanding-and-hardening-linux-containers

[18] gVisor 2018. Open-sourcing gVisor, a sandboxed container runtime. (2018). Retrieved May 23, 2019 from https://cloud.google.com/blog/products/gcp/open-sourcing-gvisor-a-sandboxed-container-runtime

[19] Jesse Hertz. 2016. Abusing Privileged and Unpriviliged Linux Containers. (2016). Retrieved May 28, 2019 from https://www.nccgroup.trust/uk/our-research/abusing-privileged-and-unprivileged-linux-containers

[20] Eric Jonas, Qifan Pu, Shivaram Venkataraman, Ion Stoica, and Benjamin Recht. 2017. Occupy the Cloud: Distributed Computing for the 99Computing *(SoCC '17)*. ACM, Santa Clara, CA, USA, 445–451. https://doi.org/10.1145/3127479.3128601

[21] Eric Jonas, Johann Schleier-Smith, Vikram Sreekanti, Chia-che Tsai, Anurag Khandelwal, Qifan Pu, Vaishaal Shankar, Joao Carreira, Karl Krauth, Neeraja Jayant Yadwadkar, Joseph E. Gonzalez, Raluca Ada Popa, Ion Stoica, and David A. Patterson. 2019. Cloud Programming Simplified: A Berkeley View on Serverless Computing. *CoRR* abs/1902.03383 (2019). arXiv:1902.03383 http://arxiv.org/abs/1902.03383

[22] Kostis Kaffes, Timothy Chong, Jack Tigar Humphries, Adam Belay, David Mazières, and Christos Kozyrakis. 2019. Shinjuku: Preemptive Scheduling for Îijsecond-scale Tail Latency. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*. USENIX Association, Boston, MA, 345–360. https://www.usenix.org/conference/nsdi19/presentation/kaffes

[23] Anuj Kalia, Michael Kaminsky, and David Andersen. 2019. Datacenter RPCs can be General and Fast. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*. USENIX Association, Boston, MA, 1–16. https://www.usenix.org/conference/nsdi19/presentation/kalia

[24] Leonard Kleinrock. 1975. *Queueing Systems. Volume 1: Theory*. John Wiley & Sons, New York.

[25] Ana Klimovic, Yawen Wang, Christos Kozyrakis, Patrick Stuedi, Jonas Pfefferle, and Animesh Trivedi. 2018. Understanding Ephemeral Storage for Serverless Analytics. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*. USENIX Association, Boston, MA, 789–794. https://www.usenix.org/conference/atc18/presentation/klimovic-serverless

[26] Ana Klimovic, Yawen Wang, Patrick Stuedi, Animesh Trivedi, Jonas Pfefferle, and Christos Kozyrakis. 2018. Pocket: Elastic Ephemeral Storage for Serverless Analytics. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. USENIX Association, Carlsbad, CA, 427–444. https://www.usenix.org/conference/osdi18/presentation/klimovic

[27] Collin Lee and John Ousterhout. 2019. Granular Computing. In *Proceedings of the Workshop on Hot Topics in Operating Systems (HotOS '19)*. ACM, Bertinoro, Italy, 149–154. https://doi.org/10.1145/3317550.3321447

[28] David Lo, Liqun Cheng, Rama Govindaraju, Parthasarathy Ranganathan, and Christos Kozyrakis. 2015. Heracles: Improving Resource Efficiency at Scale. In *Proceedings of the 42Nd Annual International Symposium on Computer Architecture (ISCA '15)*. ACM, Portland, Oregon, USA, 450–462. https://doi.org/10.1145/2749469.2749475

[29] Filipe Manco, Costin Lupu, Florian Schmidt, Jose Mendes, Simon Kuenzer, Sumit Sati, Kenichi Yasukata, Costin Raiciu, and Felipe Huici. 2017. My VM is Lighter (and Safer) Than Your Container. In *Proceedings of the 26th Symposium on Operating Systems Principles (SOSP '17)*. ACM, Shanghai, China, 218–233. https://doi.org/10.1145/3132747.3132763

[30] Omid Mashayekhi, Hang Qu, Chinmayee Shah, and Philip Levis. 2017. Execution Templates: Caching Control Plane Decisions for Strong Scaling of Data Analytics. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*. USENIX Association, Santa Clara, CA, 513–526. https://www.usenix.org/conference/atc17/technical-sessions/presentation/mashayekhi

[31] Philipp Moritz, Robert Nishihara, Stephanie Wang, Alexey Tumanov, Richard Liaw, Eric Liang, Melih Elibol, Zongheng Yang, William Paul, Michael I. Jordan, and Ion Stoica. 2018. Ray: A Distributed Framework for Emerging AI Applications. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. USENIX Association, Carlsbad, CA, 561–577. https://www.usenix.org/conference/osdi18/presentation/moritz

[32] 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 *2018 USENIX Annual Technical Conference (USENIX ATC 18)*. USENIX Association, Boston, MA, 57–70. https://www.usenix.org/conference/atc18/presentation/oakes

[33] Amy Ousterhout, Joshua Fried, Jonathan Behrens, Adam Belay, and Hari Balakrishnan. 2019. Shenango: Achieving High CPU Efficiency for Latency-sensitive Datacenter Workloads. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*. USENIX Association, Boston, MA, 361–378. https://www.usenix.org/conference/nsdi19/presentation/ousterhout

[34] Kay Ousterhout, Patrick Wendell, Matei Zaharia, and Ion Stoica. 2013. Sparrow: Distributed, Low Latency Scheduling. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles (SOSP '13)*. ACM, Farminton, Pennsylvania, 69–84. https://doi.org/10.1145/2517349.2522716

[35] George Prekas, Marios Kogias, and Edouard Bugnion. 2017. ZygOS: Achieving Low Tail Latency for Microsecond-scale Networked Tasks. In *Proceedings of the 26th Symposium on Operating Systems Principles (SOSP '17)*. ACM, Shanghai, China, 325–341. https://doi.org/10.1145/3132747.3132780

[36] Henry Qin, Qian Li, Jacqueline Speiser, Peter Kraft, and John Ousterhout. 2018. Arachne: Core-Aware Thread Management. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. USENIX Association, Carlsbad, CA, 145–160. https://www.usenix.org/conference/osdi18/presentation/qin

[37] Hang Qu, Omid Mashayekhi, Chinmayee Shah, and Philip Levis. 2018. Decoupling the Control Plane from Program Control Flow for Flexibility and Performance in Cloud Computing. In *Proceedings of the Thirteenth EuroSys Conference (EuroSys '18)*. ACM, Porto, Portugal, Article 1, 13 pages. https://doi.org/10.1145/3190508.3190516

[38] Sepideh Roghanchi, Jakob Eriksson, and Nilanjana Basu. 2017. Ffwd: Delegation is (Much) Faster Than You Think. In *Proceedings of the 26th Symposium on Operating Systems Principles (SOSP '17)*. ACM, New York, NY, USA, 342–358. https://doi.org/10.1145/3132747.3132771

[39] Francisco Romero, Qian Li, Neeraja J. Yadwadkar, and Christos Kozyrakis. 2019. INFaaS: Managed & Model-less Inference Serving. (2019). arXiv:cs.DC/1905.13348

[40] RSS 2017. Receive Side Scaling. (2017). Retrieved June 2, 2019 from http://msdn.microsoft.com/library/windows/hardware/ff556942.aspx

[41] Vaishaal Shankar, Karl Krauth, Qifan Pu, Eric Jonas, Shivaram Venkataraman, Ion Stoica, Benjamin Recht, and Jonathan Ragan-Kelley. 2018. numpywren: serverless linear algebra. *CoRR* abs/1810.09679 (2018). arXiv:1810.09679 http://arxiv.org/abs/1810.09679

[42] Jörg Thalheim, Pramod Bhatotia, Pedro Fonseca, and Baris Kasikci. 2018. Cntr: Lightweight OS Containers. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*. USENIX Association, Boston, MA, 199–212. https://www.usenix.org/conference/atc18/presentation/thalheim

[43] Shivaram Venkataraman, Aurojit Panda, Ganesh Ananthanarayanan, Michael J. Franklin, and Ion Stoica. 2014. The Power of Choice in Data-Aware Cluster Scheduling. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*. USENIX Association, Broomfield, CO, 301–316. https://www.usenix.org/conference/osdi14/technical-sessions/presentation/venkataraman

[44] Shivaram Venkataraman, Aurojit Panda, Kay Ousterhout, Michael Armbrust, Ali Ghodsi, Michael J. Franklin, Benjamin Recht, and Ion Stoica. 2017. Drizzle: Fast and Adaptable Stream Processing at Scale. In *Proceedings of the 26th Symposium on Operating Systems Principles (SOSP '17)*. ACM, New York, NY, USA, 374–389. https://doi.org/10.1145/3132747.3132750

[45] Liang Wang, Mengyuan Li, Yinqian Zhang, Thomas Ristenpart, and Michael Swift. 2018. Peeking Behind the Curtains of Serverless Platforms. In *Proceedings of the 2018 USENIX Conference on Usenix Annual Technical Conference (USENIX ATC '18)*. USENIX Association, Boston, MA, USA, 133–145. http://dl.acm.org/citation.cfm?id=3277355.3277369

[46] Neeraja J. Yadwadkar, Francisco Romero, Qian Li, and Christos Kozyrakis. 2019. A Case for Managed and Model-less Inference Serving. In *Proceedings of the Workshop on Hot Topics in Operating Systems (HotOS '19)*. ACM, Bertinoro, Italy,

184–191. https://doi.org/10.1145/3317550.3321443

[47] Ethan G. Young, Pengfei Zhu, Tyler Caraza-Harter, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2019. The True Cost of Containing: A gVisor Case Study. In *11th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 19)*. USENIX Association, Renton, WA. https://www.usenix.org/conference/hotcloud19/presentation/young

[48] Yang Zhan and Donald E. Porter. 2016. Versioned Programming: A Simple Technique for Implementing Efficient, Lock-Free, and Composable Data Structures. In *Proceedings of the 9th ACM International on Systems and Storage Conference (SYSTOR '16)*. ACM, Haifa, Israel, Article 11, 12 pages. https://doi.org/10.1145/2928275.2928285

[49] Yunqi Zhang, George Prekas, Giovanni Matteo Fumarola, Marcus Fontoura, Inigo Goiri, and Ricardo Bianchini. 2016. History-Based Harvesting of Spare Cycles and Storage in Large-Scale Datacenters. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. USENIX Association, Savannah, GA, 755–770. https://www.usenix.org/conference/osdi16/technical-sessions/presentation/zhang-yunqi