Understanding Ephemeral Storage for Serverless Analytics

Ana Klimovic, Yawen Wang, and Christos Kozyrakis, ***Stanford University;***
Patrick Stuedi, Jonas Pfefferle, and Animesh Trivedi, ***IBM Research***

https://www.usenix.org/conference/atc18/presentation/klimovic-serverless

This paper is included in the Proceedings of the 2018 USENIX Annual Technical Conference (USENIX ATC '18).

July 11–13, 2018 • Boston, MA, USA

ISBN 978-1-939133-02-1

Open access to the Proceedings of the 2018 USENIX Annual Technical Conference is sponsored by USENIX.

Understanding Ephemeral Storage for Serverless Analytics

Ana Klimovic[1], Yawen Wang[1], Christos Kozyrakis[1],
Patrick Stuedi[2], Jonas Pfefferle[2], and Animesh Trivedi[2]

[1]Stanford University
[2]IBM Research

Abstract

Serverless computing frameworks allow users to launch thousands of concurrent tasks with high elasticity and fine-grain resource billing without explicitly managing computing resources. While already successful for IoT and web microservices, there is increasing interest in leveraging serverless computing to run data-intensive jobs, such as interactive analytics. A key challenge in running analytics workloads on serverless platforms is enabling tasks in different execution stages to efficiently communicate data between each other via a shared data store. In this paper, we explore the suitability of different cloud storage services (e.g., object stores and distributed caches) as remote storage for serverless analytics. Our analysis leads to key insights to guide the design of an ephemeral cloud storage system, including the performance and cost efficiency of Flash storage for serverless application requirements and the need for a pay-what-you-use storage service that can support the high throughput demands of highly parallel applications.

1 Introduction

Serverless computing is an increasingly popular execution model in the cloud. With services such as AWS Lambda, Google Cloud Functions, and Azure Functions, users write applications as collections of stateless functions which they deploy directly to a serverless framework instead of running tasks on traditional virtual machines with pre-allocated resources [8, 14, 19, 2]. The cloud provider schedules user tasks onto physical resources with the promise of automatically scaling according to application demands and charging users only for the fine-grain resources their tasks consume.

While already popular for web microservices and IoT applications, the elasticity and fine-grain billing advantages of serverless computing are also appealing for a broader range of applications, including interactive data analytics. Several frameworks are being developed which leverage serverless computing to exploit high degrees of parallelism in analytics workloads and achieve near real-time performance [13, 17, 10].

A key challenge in running analytics workloads on serverless computing platforms is efficiently sharing data between tasks. In contrast to simple event-driven applications that consist of a single task executed in response to an event trigger, analytics workloads typically consist of multiple stages and require intermediate results to be shared between stages of tasks. In traditional analytics frameworks (e.g., Spark, Hadoop), tasks buffer intermediate data in local storage and exchange data between tasks directly over the network [25, 24]. In contrast, serverless computing frameworks achieve high elasticity and scalability by requiring tasks to be stateless [15]. In other words, a task's local file system and child processes are limited to the lifetime of the task itself. Furthermore, since serverless platforms do not expose control over task scheduling and placement, direct communication between tasks is difficult. Thus, the natural approach for inter-task communication is to store intermediate data in a common, remote storage service. We refer to data exchanged between tasks as *ephemeral data*.

There are several storage options for data sharing in serverless analytics jobs, each providing different cost, performance and scalability trade-offs. Managed object storage services like S3 offer pay-what-you-use capacity and bandwidth for storage resources managed by the provider [7]. Although primarily intended for long term data storage, they can also be used for ephemeral data. In-memory key-value stores like Redis and Memcached offer high performance, at the high cost of DRAM [21, 4]. They also require users to manage their own storage VMs. It is not clear whether existing storage options meet the demands of serverless analytics or how we can design a storage system to rule them all.

In this paper, we characterize the I/O requirements for data sharing in three different serverless applications

including MapReduce sort, distributed software compilation, and video processing. Using AWS Lambda as our serverless platform, we analyze application performance using three different types of storage systems. We consider a disk-based, managed object storage service (Amazon S3), an in-memory key value store (ElastiCache Redis), and a Flash-based distributed storage system (Apache Crail with a ReFlex Flash backend [1, 23, 18]). Our analysis leads to key insights for the design of distributed ephemeral storage, such as the use of Flash to cost-efficiently support the throughput, latency and capacity requirements of most applications and the need for a storage service that scales to meet the demands of applications with abundant parallelism. We conclude with a discussion of remaining challenges such as resource auto-scaling and QoS-aware data placement.

2 Serverless Analytics I/O Properties

We study three different serverless analytics applications and characterize their throughput and capacity requirements, data access frequency and I/O size. We use AWS Lambda as our serverless platform and configure lambdas with the maximum supported memory (3 GB) [8]. Figure 1 plots each job's cumulative storage bandwidth usage over time. Figure 2 shows the I/O size distribution.

Parallel software build: We use a framework called gg to automatically synthesize the dependency tree of a software build system and coordinate lambda invocations for distributed compilation [12, 3]. Each lambda fetches its dependencies from ephemeral storage, computes (i.e., compiles, archives or links depending on the stage), and writes an output file. Compilation stage lambdas read source files which are generally up to 10s of KBs. While 55% of files are read only once (by a single lambda), others are read hundreds of times (by many lambdas in parallel), such as glibc library files. Lambdas which archive or link read objects up to 10s of MBs in size. We use gg to compile `cmake` which has 850 MB of ephemeral data.

MapReduce Sort: We implement a MapReduce style sort application on AWS Lambda, similar to PyWren [17]. Map lambdas fetch input files from long-term storage (S3) and write intermediate files to ephemeral storage. Reduce lambdas merge and sort intermediate data read from ephemeral storage and write output files to S3. Sorting is I/O-intensive. For example, we measure up to 7.5 cumulative GB/s when sorting 100 GB with 500 lambdas. Each intermediate file is written and read only once and its size is directly proportional to the dataset size and inversely related to the number of workers.

Video analytics: We use Thousand Island Scanner (THIS) to run distributed video processing on lambdas [20]. The input is an encoded video that is divided into batches and uploaded to ephemeral storage. First

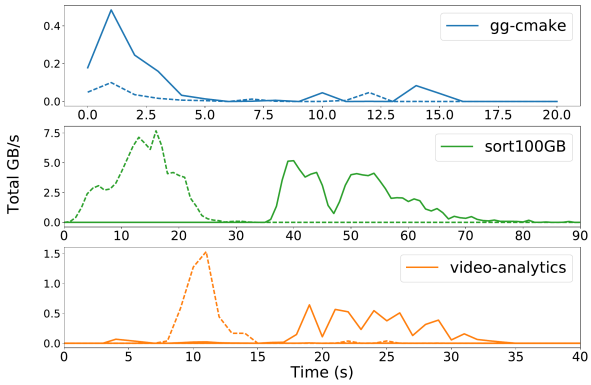


Figure 1: Cumulative throughput over time.

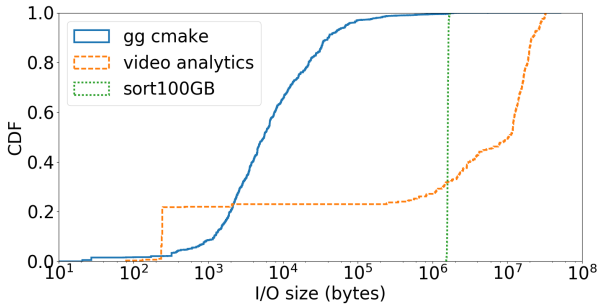


Figure 2: I/Os range from 100s of bytes to 100s of MBs.

stage lambdas read a batch of encoded video frames from ephemeral storage and write back decoded video frames. Each lambda then launches a second stage lambda which reads a set of decoded frames from ephemeral storage, computes a MXNET deep learning classification algorithm and outputs a classification result. We use a video consisting of 6.2K 1080p frames and tune the batch size to optimize runtime (62 lambdas in the decode stage and 310 lambdas for classification). The total ephemeral storage capacity is 6 GB.

3 Remote Storage for Data Sharing

We consider three different categories of storage systems for ephemeral data sharing in serverless analytics: fully managed cloud storage (e.g., S3), in-memory key-value storage (e.g., Redis), and distributed Flash storage (e.g., Crail-ReFlex). We focus on ephemeral storage as the original input and final output data of analytics jobs typically has long-term availability requirements that are well served by various existing cloud storage systems.

Simple Storage Service (S3): Amazon S3 is a fully managed object storage system that achieves high availability and scalability by replicating data across multiple nodes with eventual consistency [9]. Users pay only for the storage capacity and bandwidth they use, without ex-

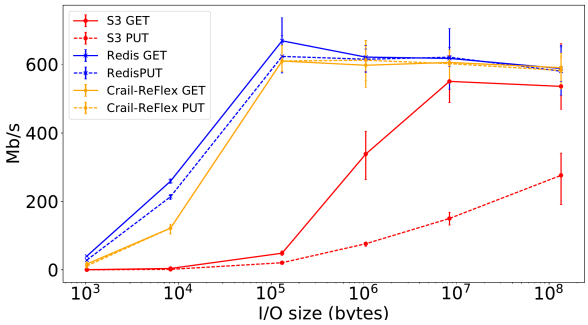


Figure 3: Peak storage throughput per lambda

	Read	Write	Metadata lookup
S3	12.1 ms	25.8 ms	–
Redis	230 μs	232 μs	–
Crail-ReFlex	283 μs	386 μs	185 μs

Table 1: Average unloaded latency for 1KB requests.

plicitly managing storage resources. S3 has significant overhead, particularly for small requests. As shown in Table 1, it takes on average over 25 ms to write 1KB. For requests smaller than 100 KB, Figure 3 shows that a single lambda achieves less than 5 MB/s (40 Mb/s) throughput. For requests 10 MB or larger, throughput goes up to 70 MB/s. With up to 2500 concurrent lambda clients, S3 scales to 80 GB/s with each client achieving approximately 30 MB/s (not shown in the figure).

Elasticache Redis: DRAM is a viable storage media for ephemeral data, which is short-lived. We use ElastiCache Redis with cluster mode enabled as an example in-memory key-value store [21, 6]. Table 1 shows that Redis latency is ~240 μs, two orders of magnitude lower than S3 latency. We find that AWS Lambda infrastructure introduces some overhead as the same `c4.8xlarge` Redis cluster has ~ 115μs lower round-trip latency from a `r4.8xlarge` EC2 client (10 GbE). We also confirm the 640 Mb/s peak per-lambda throughput in Figure 3 is an AWS Lambda limitation; the EC2 client achieves up to 5 Gb/s for the same, single TCP connection test. Since we occasionally observe lambda throughput burst above 640 Mb/s, we suspect AWS throttles a 1 Gb/s link.

Crail-ReFlex: Finally, we consider a Flash-based distributed storage system as Flash offers a medium ground between disk and DRAM for both performance and cost. In particular, NVM Express (NVMe) Flash devices are becoming increasingly popular in the cloud, offering high performance and capacity per dollar [5]. We choose to use the Apache Crail distributed storage system as it is designed for high performance access to data with low durability requirements, which matches the properties of ephemeral data. While Crail is originally designed for RDMA networks which are not available on AWS, its modular architecture supports pluggable storage tiers. We implement a NVMe Flash storage tier for Crail based on ReFlex, an open-source software system for low-latency, high throughput access to Flash over commodity networks [18]. We deploy Crail-ReFlex on `i3` EC2 nodes. Table 1 shows that from a lambda client, remote access to Flash (using Crail-ReFlex) has similar read latency as remote access to DRAM (using Redis). However, while Redis uses a simple hash to assign keys to storage servers, Crail relies on metadata servers to route client requests and manage data placement across nodes for more control over load balancing and quality of service. Thus, Crail requires an extra round-trip for metadata lookup which takes 185 μs.

4 Serverless Analytics Storage Analysis

We compare three different storage systems for ephemeral data sharing in serverless analytics and discuss how application latency sensitivity, parallelism, and I/O intensity impact ephemeral storage requirements.

Latency-sensitive jobs: We find that jobs in which lambdas mostly issue fine-grain I/O operations are latency-sensitive. Out of the applications we study, only `gg` shows some sensitivity to storage latency since the majority of files accessed are under 100 KB. Figure 4 shows the runtime for a parallel build of `cmake` as a function of the number of concurrent lambdas (`gg` allows users to set the maximum lambda concurrency, similar to `-j` in `make`). The job benefits from the lower latency of Redis storage compared to S3 with up to 100 concurrent lambdas. The runtime with S3 and Redis converges as we increase concurrency because the job eventually becomes compute-bound on AWS Lambda.

Jobs with limited parallelism: While serverless platforms allow users to exploit high application parallelism by launching many concurrent lambdas, individual lambdas are wimpy. Hence, we find that jobs with inherently limited parallelism (e.g., due to dependencies between lambdas) are likely to experience lambda resource bottlenecks (e.g., memory, compute and/or network bandwidth limits) rather than storage bottlenecks. This is the case for `gg` . The first stage of the software build process has high parallelism as each file can be pre-processed, compiled and assembled independently. However, subsequent lambdas which archive and link files depend on the outputs of earlier stages. Figure 5 plots the per-lambda read, compute and write times when using `gg` to compile `cmake` with up to 650 concurrent lambdas (650 is the highest degree of parallelism in the job's dependency graph). Using Redis (Figure 5b) compared to S3 (Figure 5a) reduces the average time that lambdas spend on I/O from 51% to 11%. However, the job takes approx-

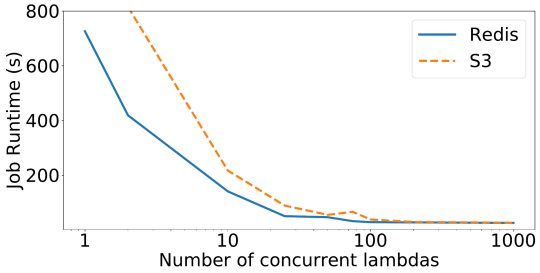
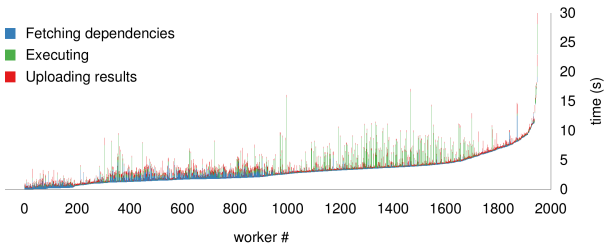


Figure 4: Distributed compilation of `cmake` is latency-sensitive at low concurrency and becomes compute-bound when run with ∼100 or more concurrent lambdas.

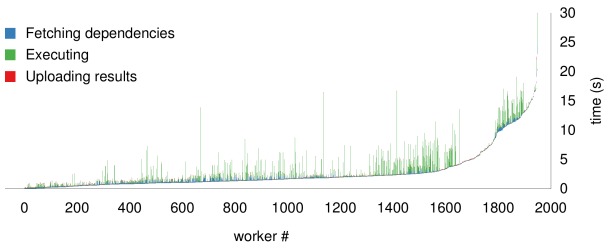
imately 30 seconds to complete, regardless of the storage system. This is because optimizing I/O does not affect the lambdas with particularly high compute latency which become the bottleneck.

Throughput-intensive jobs: MapReduce sort is an I/O-intensive application with abundant parallelism. Figure 6 shows the average time each lambda spends on I/O and compute to sort a 100 GB dataset [16]. We use S3 for input/output files and compare performance with S3, Redis (12 `cache.r4.2xlarge` nodes), Crail-ReFlex (12 `i3.2xlarge` nodes) as ephemeral storage. Storing ephemeral data in remote DRAM (Redis) or remote Flash (Crail-ReFlex) gives similar end-to-end performance, since we provision sufficient bandwidth in the storage clusters and the bottleneck becomes lambda CPU usage. Performance scales linearly as we increase the number of lambdas. S3 achieves lower throughput than Redis and Crail-ReFlex with 250 lambdas, leading to higher execution time. However, S3 outperforms a *single* node Redis or Crail-ReFlex cluster since a single node's network link becomes a bottleneck (not shown in the figure). Using S3 for ephemeral data shuffling with more than 250 lambdas in the 100 GB sort job results in I/O rate limit errors, preventing the job from completing.

Video analytics is another application with abundant parallelism. Figure 7 shows the average time lambdas in each stage spend reading, computing, and writing data. Reading and writing ephemeral data to/from S3 increases execution time compared to Redis and Crail-ReFlex. Stage 2 read time is higher with Crail-ReFlex than Redis due to read-write interference on Flash. Some lambdas in the first stage complete and launch second stage lambdas sooner than others. Thus read I/Os for some second stage lambdas interfere with the write requests from first stage lambdas that are still running. This interference can be problematic on Flash due to asymmetric read-write latency [18]. However, this does not noticeably affect overall performance as stage 2 lambdas are compute-bound. Stage 2 has low write time as its output (a list of objects detected in the video) is small.



(a) gg `cmake` with up to 650 concurrent workers and S3 storage



(b) gg `cmake` with up to 650 concurrent workers and Redis storage

Figure 5: Redis reduces I/O time compared to S3, but compute is the bottleneck. Based on Figure 6 from [12].

5 Discussion

Our analysis leads to several insights for the design of ephemeral storage for serverless analytics. We summarize the properties an ephemeral storage system should provide to address the needs of serverless analytics applications, make recommendations for the choice of storage media, and outline areas for future work.

Desired ephemeral storage properties: To meet the I/O demands of serverless applications, which can consist of thousands of lambdas in one execution stage and only a few lambdas in another, the storage system should have *high elasticity*. The system should also support *high IOPS* and *high throughput*. Since the granularity of data access varies widely (Figure 2), storing both small and large objects should be cost and performance efficient. To relieve users from the difficulty of managing storage clusters, the storage service should *auto-scale resources* based on load and charge users for the bandwidth and capacity used. This effectively extends the serverless abstraction to storage. Finally, the storage system can leverage the unique characteristics of ephemeral data. Namely, ephemeral data is short-lived and can easily be re-generated by re-running a job's tasks. Thus, unlike traditional long-term storage, an ephemeral storage system can provide low data durability guarantees. Furthermore, since the majority of ephemeral data is written and read only once (e.g., a mapper writes intermediate results for a particular reducer), the storage system can optimize capacity usage with an API that allows users to hint when data should be deleted right after it is read.

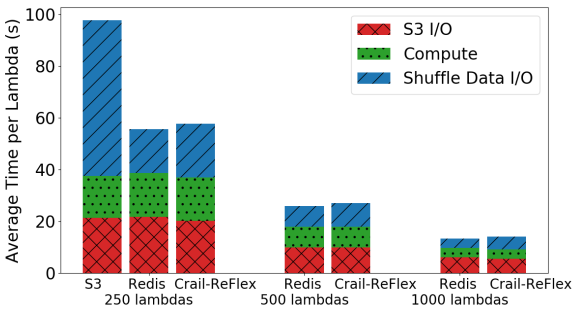


Figure 6: Average time per lambda for 100GB sort. S3 gives I/O rate limit errors with over 250 lambdas.

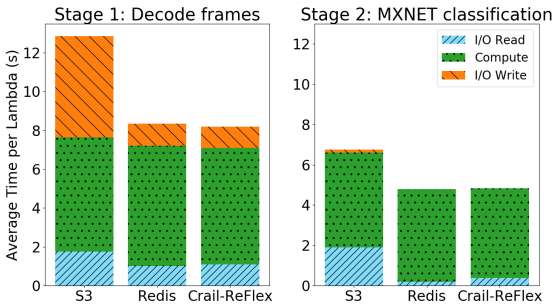


Figure 7: Video analytics I/O vs. compute breakdown, storing ephemeral data in S3, Redis and Crail-ReFlex.

Choice of storage media: A storage system can support arbitrarily high throughput by scaling resources up and/or out. The more interesting question is which storage technology allows the system to cost effectively satisfy application throughput, latency and capacity requirements. Figure 1 plots cumulative GB/s over time for gg cmake, sort, and video analytics which have 0.85, 100, and 6 GB ephemeral datasets, respectively. Considering typical throughput-capacity ratios for each storage technology (DRAM: $\frac{20GB/s}{64GB} = 0.3$, Flash: $\frac{3.2GB/s}{500GB} = 0.006$, HDD: $\frac{0.7GB/s}{6TB} = 0.0001$), we conclude that the sort application is best suited for Flash-based ephemeral storage. The throughput-capacity ratios of the gg-cmake and video analytics jobs fall into the DRAM regime. However we observed that using Flash gives similar end-to-end performance for these applications at lower cost per GB, as lambda CPU is the bottleneck. I/O intensive applications with concurrent ephemeral read and write I/Os are likely to prefer DRAM storage as Flash tail read latency increases significantly with concurrent writes [18].

Future research directions: The newfound elasticity and fine resource granularity of serverless computing platforms motivates many systems research directions. Serverless computing places the burden of resource management on the cloud provider, which typically has no upfront knowledge of user workload characteristics. Hence, building systems that dynamically and autonomously rightsize cluster resources to meet elastic application demands is critical. The challenge involves provisioning resources across multiple dimensions (e.g., compute resources, network bandwidth, memory and storage capacity) in a fine-grain manner to find low cost allocations that satisfy application performance requirements. With multiple tenants sharing serverless computing infrastructure to run jobs with high fan-out, another challenge is providing predictable performance. Interference often leads to high variability, yet a job's runtime often depends on the slowest lambda [11]. Developing *fine-grain* isolation mechanisms and QoS-aware resource sharing policies is an important avenue to explore.

6 Related Work

Fouladi et al. leverage serverless computing for distributed video processing and overcome the challenge of lambda communication by implementing the mu software framework to orchestrate lambda invocations with a long lived coordinator that is aware of each worker's state and execution flow [13]. While their system uses S3 to store ephemeral data, we study the suitability of three different types of storage systems for ephemeral data storage. Jonas et al. implement PyWren to analyze the applicability of serverless computing for generic workloads, including MapReduce jobs, and find storage to be a bottleneck [17]. We build upon their work, provide a more thorough analysis of ephemeral storage requirements for analytics jobs, and draw insights to guide the design of future systems. Singhvi et al. show that current serverless infrastructure does not support network intensive functions like packet processing [22]. Among their recommendations for future platforms, they also identify the need for a scalable remote storage service.

7 Conclusion

To support data-intensive analytics on serverless platforms, our analysis motivates the design of an ephemeral storage service that supports automatic and fine-grain allocation of storage capacity and throughput. For the three different applications we studied, throughput is more important than latency and Flash storage provides a good balance for performance and cost.

Acknowledgements

We thank the ATC reviewers for their feedback. We thank Sadjad Fouladi and Qian Li for the insightful technical discussions. This work is supported by the Stanford Platform Lab, Samsung and Huawei. Ana Klimovic is supported by a Stanford Graduate Fellowship.

References

[1] Apache crail (incubating). http://crail.incubator.apache.org/, 2018.

[2] Apache openwhisk (incubating). https://openwhisk.apache.org/, 2018.

[3] gg project. https://github.com/stanfordsnr/gg, 2018.

[4] Memcached – a distributed memory object caching system. https://memcached.org/, 2018.

[5] AMAZON. Amazon EC2 I3 instances, next-generation storage optimized high I/O instances. https://aws.amazon.com/about-aws/whats-new/2017/02/now-available-amazon-ec2-i3-instances-next-generation-storage-optimized-high-i-o-instances, 2017.

[6] AMAZON. Amazon ElastiCache. https://aws.amazon.com/elasticache/, 2018.

[7] AMAZON. Amazon simple storage service. https://aws.amazon.com/s3, 2018.

[8] AMAZON. AWS lambda. https://aws.amazon.com/lambda, 2018.

[9] AMAZON. Introduction to Amazon S3. https://docs.aws.amazon.com/AmazonS3/latest/dev/Introduction.html, 2018.

[10] DATABRICKS. Databricks serverless: Next generation resource management for Apache Spark. https://databricks.com/blog/2017/06/07/databricks-serverless-next-generation-resource-management-for-apache-spark.html, 2017.

[11] DEAN, J., AND BARROSO, L. A. The tail at scale. *Commun. ACM 56*, 2 (Feb. 2013), 74–80.

[12] FOULADI, S., ITER, D., CHATTERJEE, S., KOZYRAKIS, C., ZAHARIA, M., AND WINSTEIN, K. A thunk to remember: make -j1000 (and other jobs) on functions-as-a-service infrastructure (preprint). http://stanford.edu/~sadjad/gg-paper.pdf.

[13] FOULADI, S., WAHBY, R. S., SHACKLETT, B., BALASUBRAMANIAM, K. V., ZENG, W., BHALERAO, R., SIVARAMAN, A., PORTER, G., AND WINSTEIN, K. Encoding, fast and slow: Low-latency video processing using thousands of tiny threads. In *Proc. of the 14th USENIX Symposium on Networked Systems Design and Implementation* (2017), NSDI'17, pp. 363–376.

[14] GOOGLE. Cloud functions. https://cloud.google.com/functions, 2018.

[15] HENDRICKSON, S., STURDEVANT, S., HARTER, T., VENKATARAMANI, V., ARPACI-DUSSEAU, A. C., AND ARPACI-DUSSEAU, R. H. Serverless computation with openlambda. In *8th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 16)* (2016).

[16] HIGGS, E. Terasort benchmark for spark. https://github.com/ehiggs/spark-terasort, 2018.

[17] JONAS, E., PU, Q., VENKATARAMAN, S., STOICA, I., AND RECHT, B. Occupy the cloud: distributed computing for the 99%. In *Proceedings of the 2017 Symposium on Cloud Computing* (2017), SOCC'17, pp. 445–451.

[18] KLIMOVIC, A., LITZ, H., AND KOZYRAKIS, C. Reflex: Remote flash == local flash. In *Proc. of the 22nd International Conference on Architectural Support for Programming Languages and Operating Systems* (2017), ASPLOS '17, pp. 345–359.

[19] MICROSOFT. Azure functions. https://azure.microsoft.com/en-us/services/functions, 2018.

[20] QIAN LI, JAMES HONG, D. D. Thousand island scanner (THIS): Scaling video analysis on AWS lambda. https://github.com/qianl15/this, 2018.

[21] REDIS LABS. Redis. https://redis.io, 2018.

[22] SINGHVI, A., BANERJEE, S., HARCHOL, Y., AKELLA, A., PEEK, M., AND RYDIN, P. Granular computing and network intensive applications: Friends or foes? In *Proc. of the 16th ACM Workshop on Hot Topics in Networks* (2017), HotNets-XVI, pp. 157–163.

[23] STUEDI, P., TRIVEDI, A., PFEFFERLE, J., STOICA, R., METZLER, B., IOANNOU, N., AND KOLTSIDAS, I. Crail: A high-performance i/o architecture for distributed data processing. *IEEE Data Engineering Bulletin 40*, 1 (2017), 38–49.

[24] WHITE, T. *Hadoop: The Definitive Guide*. O'Reilly Media, Inc., 2012.

[25] ZAHARIA, M., CHOWDHURY, M., FRANKLIN, M. J., SHENKER, S., AND STOICA, I. Spark: Cluster computing with working sets. In *Proceedings of the 2Nd USENIX Conference on Hot Topics in Cloud Computing* (2010), HotCloud'10, pp. 10–10.