Towards Persistent Memory based Stateful Serverless Computing for Big Data Applications

Yuze Li[1*], Kevin Assogba[2*], Abhijit Tripathy[1], Moiz Arif[2], M. Mustafa Rafique[2], Ali R. Butt[1], Dimitrios Nikolopoulos[1]
Virginia Tech[1], Rochester Institute of Technology[2]

ABSTRACT

The Function-as-a-service (FaaS) computing model has recently seen significant growth especially for highly scalable, event-driven applications. The easy-to-deploy and cost-efficient fine-grained billing of FaaS is highly attractive to big data applications. However, the stateless nature of serverless platforms poses major challenges when supporting stateful I/O intensive workloads such as a lack of native support for stateful execution, state sharing, and inter-function communication. In this paper, we explore the feasibility of performing stateful big data analytics on serverless platforms and improving I/O throughput of functions by using modern storage technologies such as Intel Optane DC Persistent Memory (PMEM). To this end, we propose *Marvel*, an end-to-end architecture built on top of the popular serverless platform, Apache OpenWhisk and Apache Hadoop. *Marvel* makes two main contributions: (1) enable stateful function execution on OpenWhisk by maintaining state information in an in-memory caching layer; and (2) provide access to PMEM backed HDFS storage for faster I/O performance. Our evaluation shows that *Marvel* reduces the overall execution time of big data applications by up to 86.6% compared to current MapReduce implementations on AWS Lambda.

1 INTRODUCTION

Problem The Function-as-a-service (FaaS) [17] or serverless computing model is gaining popularity in the cloud environment. The model offers attractive features such as ease of deployment, function-driven execution, and cost-efficiency. Thus, the use of FaaS in supporting big data applications that make up a significant portion of cloud applications [1, 2, 20] can provide an efficient solution. In modern FaaS frameworks, data storage is decoupled from the computation instances, e.g., in object stores such as S3 [6], or managed in-memory cache instances such as Elasticache[4]. However, functions have to load data from either local or remote storage, perform computation, and store the results back to the storage tier. The distributed and isolated nature of serverless functions puts a lot of strain on the storage sub-system that can quickly become a performance bottleneck for large amounts of data such as that used in emerging deep learning [23] and video encoding [19] applications.

Big data application processing frameworks such as Apache Hadoop [8] require stateful operations, typically supported by a distributed filesystem such as HDFS [21]. It is natural to integrate these frameworks for their programming advantages and FaaS for its cost efficiency, and researchers have begun to explore such integration. However, using stateless serverless to run stateful big data applications is challenging. The stateless nature of serverless limits any inter-function communications [18]. Big data applications, however, require data processing and storage to be co-located in order to avoid the network performance bottleneck. Previous solutions [14, 25, 26, 28] are stateless in nature and uses remote storage media to read input data, cache intermediate data, and write output data. For example, Corral [14] and AWS Serverless Reference Architecture [28] use AWS S3 as storage media for intermediate state. These solutions decouple/distribute the compute and storage and thus are slow; they require at least four I/O calls to read and write data to either local or remote network storage entailing higher latency. In Hadoop, for example, mappers read input from remote storage, write back intermediate state, reducers read from remote storage, and then write back the final output. In such cases, the network quickly becomes the bottleneck and reduces overall performance.

Another limitation of existing solutions is that they rely on commercial frameworks that only provide serverless platforms with no native support for Big Data applications. These applications would require custom implementations of data-parallel processing frameworks, particularly MapReduce. As a result, existing solutions offer very simple serverless implementations of MapReduce with minimal features and no support for data-parallel processing [26]. These systems basically take the ad-hoc approach of running serverless functions with remote storage services for MapReduce workloads to share state. Such I/O strategies scale poorly in data-intensive workloads. This is because during the shuffle phase of MapReduce, each mapper sends its intermediate data to reducers that may not be scheduled on the same host. Thus, the number of remote I/O requests increase leading to observable

[*] Equal Contribution.

,

performance degradation due to slower I/O. In our experiments we found that Corral's Lambda and S3 based invocations fail due to the maximum data transfer limit of 15 GB. Moreover, such solutions require isolated, long-running functions that need to coordinate the state information, and thus create another potential bottleneck of coordinating the transition from the map stage to the reduce stage. Being stateful, any function failure will result in loss of computation, state and data resulting in application-level failure.

Solution Recent hardware trends offer a promising substrate for integrating FaaS and Big Data processing frameworks. New memory technologies such as Intel's Optane DC Persistent Memory (PMEM) [15] can offer a stateful operations substrate in a serverless setting at near DRAM speed, thus mitigating the need for slower local or remote storage.

In this paper, we propose an architecture that integrates serverless platforms with Hadoop to launch stateful functions for big data processing. We optimize the I/O performance of this architecture by co-locating compute and storage capability via persistent memory. The overarching goal is to leverage the ease-of-deployment and cost-efficiency of serverless platforms and the data analytics capabilities of Hadoop. We design *Marvel*, which to the best of our knowledge is the first solution to support stateful Big Data function execution in serverless platforms. We enable the serverless platform to launch stateful functions, coordinate the data and state sharing amongst MapReduce components, and we integrate PMEM with Hadoop HDFS to provide fast intermediate storage and a storage backend for input and output data. *Marvel* uses Apache OpenWhisk [10], to launch MapReduce actions that interact with the Apache Hadoop[8] core components. Hadoop core components, e.g., NameNode, DataNode, and NodeManager, are deployed in OpenWhisk containers. The NameNode provides data storage services to MapReduce functions through optimized DataNodes that use PMEM as the underlying storage media. This allows us to achieve better I/O performance, avoid the bottlenecks associated with slow storage tiers, and minimize data transfers over the network. We use Apache Ignite[9] as an in-memory accelerator to store the intermediate data. This allows us to store intermediate data produced by Hadoop functions in a fast storage tier accessible to all functions of the application.

Contributions Specifically, we make the following contributions:

- We design an architecture that enables serverless platforms to run stateful big data applications on Hadoop framework while abstracting away complicated infrastructure management, and providing highly scalable, and cost-effective FaaS.
- We provide a fast in-memory data store for maintaining function states and storing intermediate data produced by MapReduce components to enable a fast inter-function communication path.
- We improve the performance of big data applications by utilizing PMEM as the storage backend for HDFS to provide high throughput and low latency data access.
- We present an architecture that requires minimal changes to the underlying frameworks and to the user applications, this making it easy to use and deploy in a distributed cluster.

2 MOTIVATION AND BACKGROUND

In the following, we provide motivational experiments for *Marvel*, and discuss related work and background on persistent memory and serverless computing frameworks.

Dataset sizes in MapReduce applications Infrastructure-as-a-Service (IaaS) providers such as AWS EC2 [3], have proved efficient in supporting big data applications and providing the needed performance and scalability. However, additional steps have to be taken to design and deploy the infrastructure, launch, configure and maintain compute instances. This is an additional burden on the application developer in terms of time, effort and cost. In contrast, FaaS or serverless computing, aims at making infrastructure transparent to the application developers which is highly desirable. FaaS frameworks support fast auto-scaling that allocates compute resources dynamically to fulfill application requirements at runtime [16]. However, current FaaS implementations do not support stateful function execution and rely on cloud storage services for state sharing. The remote nature of such storage results reduces I/O and application performance. Therefore, access to a fast storage layer is crucial for running big data applications on stateful serverless platforms.

To better understand the above challenge we studied the size of input, intermediate, and output data generated by different MapReduce workloads as shown in Table 1. We observed a high degree of variation in the size of data for big data workloads at each step. During the MapReduce shuffle phase, mapper functions send data to the reducers. Therefore, as the input size increases, the data I/O from local or remote storage increase. This leads to the storage and network bottlenecks, e.g., when using S3, that puts a limit on the IOPS and charges a premium per I/O request. [24].

Role of storage in serverless performance To observe the impact of data locality and storage types on stateful application's performance we run MapReduce application on AWS Lambda [31] that uses AWS S3 [6] for storage. We compared the performance of this setup with an on-premise serverless deployment that utilizes different storage backends.

Table 1: Dataset sizes at different MapReduce phases.

Workloads	Input Size (GB)	Intermediate Data Size (GB)	Output Size (GB)
Scan Query	0.54	0.76	0.1
	1.2	1.3	0.16
	5.7	6.7	0.81
Aggregation Query	10.5	17.4	0.01
	26.3	32	0.03
	58	74	0.03
Join Query	12.5	49.6	9.7
	27.5	103	22.6
	63.7	242	51
Word Count	1	5.5	0.01
	5	28	0.03
	10	56	0.1
	50	291	0.4

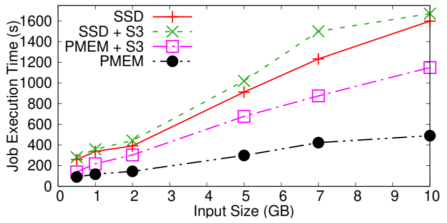

Figure 1: Wordcount performance using serverless with varying storage layer provided by S3, local SSD, and PMEM.

Figure 1 shows the job completion time for a wordcount application that uses the Corral MapReduce library. We compare the performance by running the application backed by SSD, SSD and S3, PMEM and S3, and finally just PMEM. The results show that for an input size of 7 GB the PMEM performed the best, with SSD performance slightly slower and S3 backed experiments performed the worse. Hence, leveraging new persistent memory technologies is an effective way to improve the I/O performance of serveless frameworks.

Serverless and big data platforms Integrating serverless and big data platforms is becoming increasingly important due to the benefits of both frameworks [35]. Public cloud providers such as AWS and NetApp are developing serverless big data solutions, e.g., AWS EMR on serverless[20] and Ocean for Apache Spark[30]. However, at the time of writing this paper AWS EMR Serverless is under development, whereas NetApp's ocean is available for demo use only. Moreover, based on our research, there are no opensource projects that integrate modern storage technologies, such as PMEM, with serverless platforms.

Persistent memory Intel has recently launched DC class of PMEM, a type of Phase change memory based on the 3D XPoint memory media [22]. PMEM can be configured as: (1) a volatile main memory while using DRAM as an L4 cache (*Memory mode*), (2) a persistent byte-addressable storage device (*AppDirect mode*), (3) a combination of *Memory mode* and *AppDirect mode* called *Mixed mode.* Researchers have investigated the use of PMEM in multiple scenarios, e.g. handling disaggregated PMEM consistency [34, 36], PMEM file system [13, 27, 29], distributed PMEM Pool [33], and PMEM-embedded network [32]. Based on a recent study [37], many popular PMEM-based file systems have shown I/O performance improvement over other slower storage devices. Considering the high performance of PMEM, we include PMEM for handling data I/O in *Marvel.*

Table 2: IOPS, Bandwidth, Latency for PMEM vs. SSD.

Benchmarks		IOPS (K)	Bandwidth (GiB/s)	Latency
Seq. Read	PMEM	10700.0	41.0	0.6 us
	SSD	108.0	0.4	4.7 ms
Seq. Write	PMEM	3314.0	13.6	1.9 us
	SSD	118.0	0.5	5.0 ms
Random Read	PMEM	1166	4.6	0.6 us
	SSD	82.3	0.3	0.8 ms
Random Write	PMEM	335.0	1.4	2.3 us
	SSD	66.2	0.3	1.0 ms

To demonstrate the benefits of PMEM over SSDs, we used FIO [12] microbenchmark. We configure PMEM in AppDirect Mode, and mount a DAX-enabled EXT4 filesystem on it. We use a 4 KB block size, and up to eight parallel streams for sequential reads and writes. Moreover, we use *libaio* and *libpmem* as I/O engines for SSD and PMEM, respectively. Table 2 shows $10\times - 100\times$ speed-up in IOPS, bandwidth, and latency for PMEM.

3 *MARVEL* DESIGN

3.1 Design Goals

The main goal of this paper is to design a system that supports stateful function execution of big data applications and to improve the application performance. The key objectives driving the design of *Marvel* are as follows: (1) Enable serverless platform to run stateful functions by storing function states. This will enable the execution of stateful big data analytics applications to leverage the benefits of serverless infrastructure. (2) Enable access to an in-memory distributed database and PMEM-backed storage for hosting input, output, and intermediate data, thus improving the throughput and reducing the total application execution time.

3.2 Challenges

Communication between OpenWhisk actions and Hadoop components All OpenWhisk components and actions are deployed as Docker containers. Hadoop does not include native support for Docker. Therefore, there is a need to containerize Hadoop and make the components accessible to OpenWhisk over the network to both frameworks.

Integrating Ignite with Hadoop In containerized Hadoop, the intermediate data produced by mappers is stored in HDFS

,

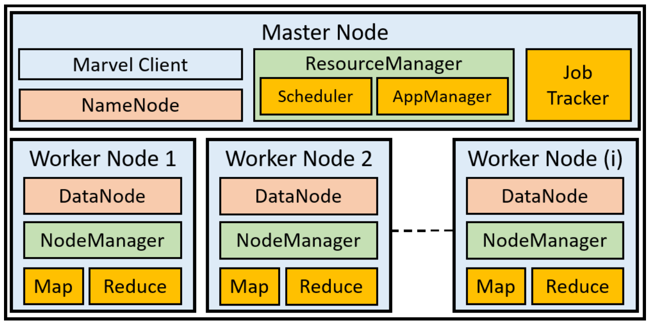

Figure 2: Proposed architecture of *Marvel*.

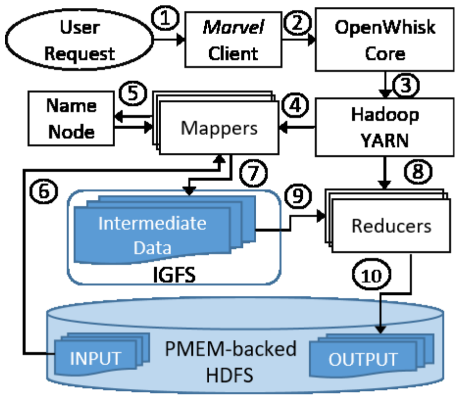

Figure 3: Job Execution Workflow in *Marvel*.

backed by local storage media. To speed up I/O, mapper functions must be configured to store intermediate data in Ignite, and reducer functions to read that data from the database while writing the final output to HDFS.

Ease of deployment To enable quick, easy and automated deployment, all OpenWhisk, Hadoop, and Ignite components, inter-component integration, configuration of PMEM needs to be automated to support and end-to-end deployment.

3.3 Design Overview

Figure 2 shows the high-level architecture of *Marvel*. We use OpenWhisk as the serverless platform, Hadoop as the big data framework, PMEM for storing data, and Ignite for storing intermediate data. In *Marvel*, OpenWhisk Invoker executes user requests inside the Hadoop runtime that provide the necessary libraries to launch the use code.

All Hadoop components are deployed as Docker containers, and communicate through a Docker overlay network accessible to all OpenWhisk components. In a distributed setup, NameNode and ResourceManager are deployed along with OpenWhisk core components on the master node, while each worker node runs an instance of NodeManager, DataNode and OpenWhisk Invoker. All core components are backed by persistent docker volumes mounted on top of PMEM. The developer submits a job through the client which coordinates the launch of OpenWhisk actions, data provisioning from the DataNodes via NameNode and providing endpoints to mappers and reducers to access the intermediate data. Finally, we use YARN [11] for determining the appropriate number of Mappers/Reducers needed per job.

3.4 *Marvel* Architecture

3.4.1 Apache OpenWhisk. Apache OpenWhisk manages the serverless platform on the user's behalf and handles the lifecycle of the action containers. The OpenWhisk controller launches actions that contain the user code, communicates with the resource manager, the NameNode and DataNodes for accessing input and storing the final output data.

3.4.2 Apache Hadoop. Apache Hadoop provides a big data analytics platform and is deployed on top of the serverless infrastructure. It is integrated with the OpenWhisk core for scheduling MapReduce functions. We use HDFS to store data, where computation is performed on the same nodes that host the data, thus achieving data co-location. The HDFS DataNode containers use persistent volumes mounted on top of PMEM to speeds up I/O performance. To ensure that all components from all cluster nodes can communicate, the entire stack is deployed within an overlay network. However, OpenWhisk is designed to deploy containers in the default docker bridge network. To ensure that OpenWhisk actions communicate with Hadoop, we modified OpenWhisk to deploy all containers, including OpenWhisk core components, within the same overlay network. To enable the communication between OpenWhisk and Hadoop we built a Docker-based function runtime with necessary packages exported.

3.4.3 Apache Ignite File System (IGFS). Apache Ignite provides a high-speed database that enables stateful execution and stores intermediate function data. As described in Section 2, intermediate data requires significant I/O during the application execution. Thus, we deploy Ignite File System as a distributed in-memory cache, to allow low-latency access to intermediate data. To enable Hadoop components to communicate with Ignite for reading and writing intermediate data, the mappers and reducers use the ignite library that is included in the runtime image created as a part of *Marvel*.

3.5 Application Workflow

Figure 3 shows the job execution workflow in *Marvel*. Users submit jobs to the *Marvel* client, which serves as an entry point to the stateful serverless big data framework (1). The client coordinates application execution with OpenWhisk core (2) which sends the execution requests to YARN (3) and includes the job's metadata and the submitted JAR files. YARN schedules mappers (4) on the workers where OpenWhisk invokers are deployed. The mappers fetch locations of the input data from the NameNode (5) and read it from the PMEM-backed HDFS DataNodes (6). The mappers perform the map step, and stores the shuffled output as intermediate data into IGFS (7). After the Map phase completes, YARN spawns reducers functions (8) which read intermediate data from IGFS (9), runs the reduce phase and stores the final output to the PMEM-backed HDFS (10). The entire application execution is managed and monitored by OpenWhisk.

4 EVALUATION AND DISCUSSION

4.1 Evaluation Setup

Marvel supports distributed cluster deployment, where Hadoop and OpenWhisk core components are deployed on a cluster of machines. To have a fair comparison with Lambda-supported Corral [14] we test *Marvel* on a single server. Our server comprises of 32 Intel Xeon Silver 4215 CPU @ 2.50GHz CPUs, 360 GB of DRAM, 700 GB of PMEM in App Direct Mode, and runs CentOS 8. We use Apache Hadoop 3.2.1 with Apache Ignite 2.6.1 as the distributed database to store intermediate function data. For Corral, we configure the maximum 10 GB memory per function instance running on Lambda. The memory and vCPU configuration is kept consistent across all experiments. We run tests on three system configurations: (1) Lambda with S3 and Corral's MapReduce library; (2) *Marvel* with HDFS where the DataNodes are mounted on PMEM; and (3) *Marvel* with IGFS that is similar to the previous configuration, except that the intermediate data is cached in Ignite. We run the experiments 5 times and report the average.

4.2 Performance Results

4.2.1 Impact of data locality and Ignite memory database on workload performance. We study the performance of *Marvel* with well-known WordCount and Grep benchmarks. Mappers in WordCount scan input files word by word, and emit key-value pairs, like <word, 1>. WordCount reducers sum up the frequency of each word and emit the final <word, count> key-value pairs. Grep mappers work similarly to WordCount, matching each word against a specific regular expression, while the reducers only count those words that meet the regular expression. We show a zoomed-in version of the WordCount experiment where input size is between 0.5 GB and 11 GB. From figures 4 and 5, we make the following observations. (1) The Corral Lambda solution operating on AWS architecture reaches its concurrency quota at 15 GB of input size. This is due to AWS's rate-limiting criteria, that puts a limit on S3 I/O and function invocation requests. [5, 7] (2) For medium-sized workloads, *Marvel* with HDFS achieves comparable performance as AWS Lambda, which also depends on the type of workloads. As a benefit of data co-location, where data is directly fetched from HDFS, the network bandwidth constraint on *Marvel* is reduced. The result shows that *Marvel* reduces the job execution time by 86.6 %, compared to Lambda, while allowing stateful execution for serverless functions. (3) *Marvel* with IGFS where intermediate data is in memory shows the best performance.

4.2.2 I/O throughput for HDFS Vs. IGFS. Next, we examine the throughput for the various studied cases. Figure 6 shows the I/O throughput for HDFS mounted on PMEM, and IGFS, while running the WordCount workload. Here, we can observe that while using IGFS, the I/O throughput increases with an increase in data size, and reaches a peak throughput of 12 Gbps at 10 Gb input size. This shows that using an in-memory cache like IGFS can boost the I/O performance of MapReduce applications.

4.3 Discussion and Future Work

We have taken the first steps to design *Marvel* and integrate serverless and MapReduce in a performance-aware fashion. Our evaluation indicates that the approach has promise. However, a number of challenges remain.

First, we need to explore using Ignite as a distributed database on top of PMEM. By doing that, intermediate data could be persisted while making it available on DRAM for fast I/O performance. This will enable us to develop a checkpoint-based fault-tolerant mechanism for applications to resume execution in the event of machine or network failures, while allowing access to data on a fast storage tier.

Second, it is uncertain how serverless platforms can interact with the resource managers of big data frameworks. In this paper, we have explored the interaction between the scheduling policies of OpenWhisk and Hadoop YARN [11]. Two potential approaches that we will explore in our ongoing work are: (1) Allow YARN to make placement decisions for OpenWhisk Action containers; or (2) Make OpenWhisk aware of the Hadoop cluster topology, data placement, and resource usage, and let it schedule the MapReduce actions in a manner similar to YARN.

5 CONCLUSION

In this paper, we presented *Marvel*, a stateful FaaS architecture that enables and optimizes the performance of stateful function execution of big data applications by utilizing persistence memory. We investigate the use of PMEM in various architectural settings to maximize the I/O throughput of stateful functions of the Hadoop framework. Our evaluation shows that *Marvel* reduces job execution time by 86.6%, compared to the default serverless platform.

REFERENCES

[1] [n.d.]. Apache Hadoop open source ecosystem | Cloudera. https://www.cloudera.com/products/open-source/apache-hadoop.html. (Accessed on 03/29/2022).

,

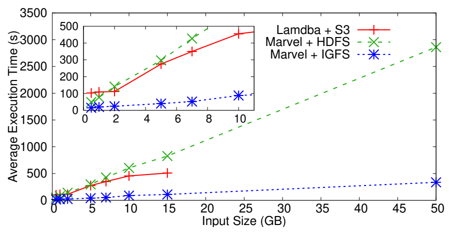

Figure 4: WordCount execution time.

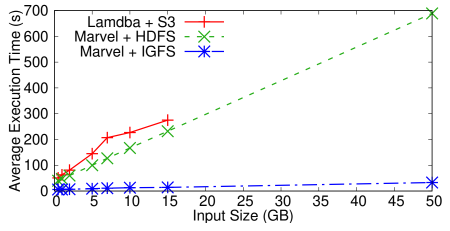

Figure 5: Grep execution time.

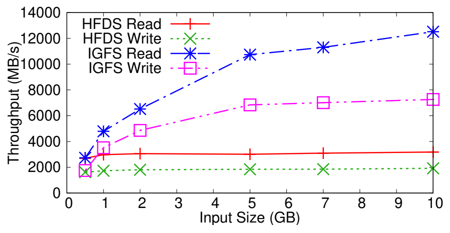

Figure 6: Throughput comparison.

[2] [n.d.]. Hortonworks Data Platform | Cloudera. https://www.cloudera.com/products/hdp.html. (Accessed on 03/29/2022).
[3] Amazon. [n.d.]. AWS EC2. https://aws.amazon.com/ec2/
[4] Amazon. [n.d.]. AWS Elasticache. https://aws.amazon.com/elasticache/
[5] Amazon. [n.d.]. AWS Lambda Quotas. https://docs.aws.amazon.com/lambda/latest/dg/gettingstarted-limits.html
[6] Amazon. [n.d.]. AWS S3. https://aws.amazon.com/s3/
[7] Amazon. [n.d.]. AWS S3 Request limit throttling. https://aws.amazon.com/premiumsupport/knowledge-center/s3-request-limit-avoid-throttling/
[8] Apache. [n.d.]. Apache Hadoop Framework. https://cwiki.apache.org/confluence/display/HADOOP/
[9] Apache. [n.d.]. Apache Ignite Documentation. https://ignite.apache.org/docs/latest/.
[10] Apache. [n.d.]. Apache/openwhisk: Apache OpenWhisk is an open source serverless cloud platform. https://github.com/apache/openwhisk
[11] Apache. [n.d.]. Hadoop YARN. https://hadoop.apache.org/docs/stable/hadoop-yarn/hadoop-yarn-site/YARN.html
[12] Jens Axboe. 2022. *Flexible I/O Tester.* https://github.com/axboe/fio
[13] Youmin Chen, Jiwu Shu, Jiaxin Ou, and Youyou Lu. 2018. HiNFS: A persistent memory file system with both buffering and direct-access. *ACM Transactions on Storage (ToS)* 14, 1 (2018), 1–30.
[14] Ben Congdon. 2020. Introducing corral: A serverless mapreduce framework. https://benjamincongdon.me/blog/2018/05/02/Introducing-Corral-A-Serverless-MapReduce-Framework/
[15] Intel Corporation. 2019. Intel® optane™ persistent memory product brief. https://www.intel.com/content/www/us/en/products/docs/memory-storage/optane-persistent-memory/optane-dc-persistent-memory-brief.html
[16] IBM Cloud Education. [n.d.]. What is Serverless Computing? | IBM. https://www.ibm.com/cloud/learn/serverless. (Accessed on 03/26/2022).
[17] IBM Cloud Education. 2019. What is FaaS (Function-as-a-Service)? | IBM. https://www.ibm.com/cloud/learn/faas. (Accessed on 03/29/2022).
[18] 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)*. 475–488.
[19] 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)*. 363–376.
[20] J. B. Gilmour, A. W. Lui, and D. C. Briggs. 1986. Serverless EMR. https://docs.aws.amazon.com/emr/latest/EMR-Serverless-UserGuide/emr-serverless.html
[21] IBM. [n.d.]. Hadoop Distributed File System. https://www.ibm.com/topics/hdfs
[22] Intel. [n.d.]. Intel Optane Technology. https://newsroom.intel.com/press-kits/introducing-intel-optane-technology-bringing-3d-xpoint-memory-to-storage-and-memory-products/
[23] Jiawei Jiang, Shaoduo Gan, Yue Liu, Fanlin Wang, Gustavo Alonso, Ana Klimovic, Ankit Singla, Wentao Wu, and Ce Zhang. 2021. Towards demystifying serverless machine learning training. In *Proceedings of the 2021 International Conference on Management of Data*. 857–871.
[24] 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. https://doi.org/10.48550/ARXIV.1902.03383
[25] Youngbin Kim and Jimmy Lin. 2018. Serverless data analytics with flint. In *2018 IEEE 11th International Conference on Cloud Computing (CLOUD)*. IEEE, 451–455.
[26] Hang Li Li, Robert Chatley, and Giuliano Casale. 2020. *Serverless Data Pipelines*. Ph.D. Dissertation. Master's thesis. Imperial College London.
[27] Ruibin Li, Xiang Ren, Xu Zhao, Siwei He, Michael Stumm, and Ding Yuan. 2022. ctFS: Replacing file indexing with hardware memory translation through contiguous file allocation for persistent memory. In *Proceedings of the 20th Usenix Conference on File and Storage Technologies, FAST*, Vol. 22.
[28] Bryan Liston. 2016. Ad Hoc Big Data Processing Made Simple with Serverless MapReduce. https://github.com/awslabs/lambda-refarch-mapreduce
[29] Youyou Lu, Jiwu Shu, Youmin Chen, and Tao Li. 2017. Octopus: an {RDMA-enabled} Distributed Persistent Memory File System. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*. 773–785.
[30] Jean-Yves Stephan Senior Product Manager, Jean-Yves Stephan, and Senior Product Manager. 2022. Orchestrate spark pipelines with airflow on ocean for Apache Spark. https://spot.io/blog/orchestrate-spark-pipelines-with-airflow-on-ocean-for-apache-spark/
[31] DAVID MUSGRAVE. 2022. Lambda. https://docs.aws.amazon.com/lambda/index.html
[32] Korakit Seemakhupt, Sihang Liu, Yasas Senevirathne, Muhammad Shahbaz, and Samira Khan. 2021. PMNet: in-network data persistence. In *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 804–817.
[33] Jiwu Shu, Youmin Chen, Qing Wang, Bohong Zhu, Junru Li, and Youyou Lu. 2020. Th-dpms: Design and implementation of an rdma-enabled distributed persistent memory storage system. *ACM Transactions on Storage (TOS)* 16, 4 (2020), 1–31.
[34] Shin-Yeh Tsai, Yizhou Shan, and Yiying Zhang. 2020. Disaggregating Persistent Memory and Controlling Them Remotely: An Exploration of Passive Disaggregated {Key-Value} Stores. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*. 33–48.
[35] Sebastian Werner, Jörn Kuhlenkamp, Markus Klems, Johannes Müller, and Stefan Tai. 2018. Serverless Big Data Processing using Matrix

Multiplication as Example. In *2018 IEEE International Conference on Big Data (Big Data)*. 358–365. https://doi.org/10.1109/BigData.2018.8622362

[36] Ming Zhang, Yu Hua, Pengfei Zuo, and Lurong Liu. 2022. FORD: Fast One-sided RDMA-based Distributed Transactions for Disaggregated Persistent Memory. In *20th USENIX Conference on File and Storage Technologies (FAST 21). USENIX Association.*

[37] Guangyu Zhu, Jaehyun Han, Sangjin Lee, and Yongseok Son. 2021. An empirical evaluation of nvm-aware file systems on intel optane dc persistent memory modules. *Electronics* 10, 16 (2021), 1977.