2019 19th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)

Optimizing Performance and Computing Resource Management of in-memory Big Data Analytics with Disaggregated Persistent Memory

Shouwei Chen*‡, Wensheng Wang*, Xueyang Wu*, Zhen Fan*, Kunwu Huang†,
Peiyu Zhuang†, Yue Li†, Ivan Rodero‡, Manish Parashar‡, Dennis Weng*
*JD.com Inc., †MemVerge Inc., ‡Rutgers Discovery Informatics Institute

***Abstract*—The performance of modern Big Data frameworks, e.g. Spark, depends greatly on high-speed storage and shuffling, which impose a significant memory burden on production data centers. In many production situations, the persistence and shuffling intensive applications can suffer a major performance loss due to lack of memory. Thus, the common practice is usually to over-allocate the memory assigned to the data workers for production applications, which in turn reduces overall resource utilization. One efficient way to address the dilemma between the performance and cost efficiency of Big Data applications is through data center computing resource disaggregation. This paper proposes and implements a system that incorporates the Spark Big Data framework with a novel in-memory distributed file system to achieve memory disaggregation for data persistence and shuffling. We address the challenge of optimizing performance at affordable cost by co-designing the proposed in-memory distributed file system with large-volume DIMM-based persistent memory (PMEM) and RDMA technology. The disaggregation design allows each part of the system to be scaled independently, which is particularly suitable for cloud deployments. The proposed system is evaluated in a production-level cluster using real enterprise-level Spark production applications. The results of an empirical evaluation show that the system can achieve up to a 3.5-fold performance improvement for shuffle-intensive applications with the same amount of memory, compared to the default Spark setup. Moreover, by leveraging PMEM, we demonstrate that our system can effectively increase the memory capacity of the computing cluster with affordable cost, with a reasonable execution time overhead with respect to using local DRAM only.**

I. Introduction

The increasing generation of data at high rates is creating new requirements for large-scale enterprise applications from different business areas. For example, faster Big Data analytics are critical for supporting speedier business intelligence and for leveraging the recent advances in machine learning. This trend is pushing Big Data analysis systems toward high-performance in-memory processing solutions. However, in-memory Big Data processing systems impose a significant memory burden on the data center enterprise, especially in terms of capital and operation costs.

To deliver high performance, the computing infrastructure must provide sufficient DRAM memory to hold large amounts of intermediate data. In addition, it is common to see a data skew in production applications as the data size increases. A common practice in production environments is to over-allocate the memory assigned to the data workers. This lowers overall memory utilization and increases the data center's memory requirements. At the same time, real large-scale production data centers are facing several challenges, such as uneven resource utilization levels between CPU and memory and the expensive cost of DRAM memory. As a result, these factors are limiting how data centers can deliver high-performance in-memory Big Data applications efficiently.

Both academia and industry have devoted significant efforts to design and implement data center architectures to optimize the use of novel hardware technologies. However, advances in processor, memory, storage, and network technology show different growth and demand trends. Therefore, data center architectures that are built based on the vision of server-centric resources face significant challenges with adopting the latest hardware innovations quickly [1], [2]. Conversely, disaggregated data center designs allow different resources to be scaled independently, enabling the faster adoption of novel hardware developments at a lower cost. In turn, enterprise applications can effectively obtain full value from an advanced disaggregated infrastructure and its associated investments. This is especially beneficial for supporting in-memory Big Data frameworks. However, to address the challenges associated with upgrading data center infrastructures at affordable cost, co-designing applications with innovative memory system architectures is essential.

To address the challenges described above, we design and implement an in-memory Big Data processing system using disaggregated memory resource. The system is based on a state-of-the-art in-memory Big Data framework and a novel in-memory distributed file system. Existing research efforts [3]–[6] have explored the potential of remote page caching; however, unlike existing approaches, we increase the memory capacity of the data processing cluster with large-volume DIMM-based persistent memory (PMEM) [7] and the abstraction provided by the proposed in-memory distributed file system co-designed with PMEM. With our approach, we implement a disaggregated memory pool without modifying the kernel. Furthermore, instead of remote page caching, our implementation is capable of transparently employing disaggregated memory resources in a data processing cluster using Spark [8] and the proposed in-memory distributed file system. We also propose external shuffle/persistence storage in a pool of disaggregated memory resources using the proposed in-memory distributed file system, thereby significantly

978-1-7281-0912-1/19/$31.00 ©2019 IEEE
DOI 10.1109/CCGRID.2019.00012

improving the performance and resource utilization of Spark. This is especially important for large-scale enterprise data centers running Big Data frameworks, as computing resource utilization is typically uneven, and with the large volume of data (i.e., TBs to tens of TBs of data per application), it is difficult to achieve a high performance without significant and complex tuning. To the best of our knowledge, this is the first work addressing these issues in a real production enterprise data center.

In this paper, we present several scientific and engineering contributions. We first characterize the computing resource efficiency of a real production enterprise data center. We address the uneven utilization of memory and CPU, and then we address the bottleneck of the current in-memory Big Data framework within an existing data center. We also characterize the remote read and write performances of PMEM and DRAM, enabling us to explore the potential of PMEM as an affordable replacement for DRAM, used in a disaggregated memory pool. Next, we present the design and implementation of the proposed in-memory Big Data processing system using disaggregated memory with Spark and an in-memory distributed file system called Distributed Memory Objects (DMO). We present the design and implementation of an external shuffle service with DMO, leading to a savings of up to 72% in execution time with shuffle-intensive Spark applications having the same memory consumption. Furthermore, we present the design and implementation of external extended storage for Spark data shuffling and persistence with DMO and large-volume PMEM. Finally, we empirically evaluate the performance of our system with real production workloads. We demonstrate that the system can increase memory capacity at affordable cost and with low overhead, compared to using DRAM exclusively. Finally, we discuss the impacts of our system on real production data processing systems.

The rest of the paper is organized as follows. Section II studies the computing resource utilization of a real production enterprise data center. Section III discusses current Spark memory management and shuffling. Section IV characterizes the remote read/write performance of PMEM and Section V presents the design and implementation of the proposed shuffling storage and persistence system using DMO and PMEM. Section VI provides the experimental evaluation of the proposed system using real production workloads. Finally, section VII presents an overview of the literature and section VIII concludes the paper and outlines directions for future work.

II. Analysis of the Efficiency of Data Center Computing Resources

Before discussing the details of our proposed approach, in this section, we provide a detailed utilization analysis of computing resources in a real production data center with $3,700$ servers from a large E-Commerce company. This characterization will enable us to understand the bottlenecks of current production computing clusters. We first analyze the utilization of computing resources (both CPU and memory) during one month (August 2018), and then we analyze the performance pattern of real production applications.

We collected profiling data through an internal cluster monitoring tool. The data includes multiple system metrics from the data center's computing cluster running Big Data analytics using Spark. As the focus of this work is on computing resource management, we use the allocation information of the YARN scheduler collected through the monitoring tool, to analyze the utilization of the CPU and memory, which are the most important computing resources in the cluster.

In order to illustrate the computing resource utilization in a data processing cluster, we define cluster memory overhead $\text{MEM}_{\text{Overhead}}$ as:

$$\text{MEM}_{\text{overhead}} = \text{CPU}_{\text{util}} - \text{MEM}_{\text{util}}$$

where CPU utilization CPU_{util} is the ratio between the number of allocated CPU cores and the number of allocable CPU in the cluster, and memory utilization MEM_{util} is the ratio between the amount of allocated memory and the amount of total allocable memory in the cluster.

Fig. 1 shows the memory overhead of the computing cluster during the data collection period. The average memory utilization is mostly above that of the CPU utilization for the data center. Therefore, the Big Data processing frameworks that run on the computing cluster cannot fully utilize the available CPU cores because of limited memory resources. In the following section, we discuss the current memory architecture and memory management of Spark, and explain the high memory utilization of the computing cluster.

III. Background of Spark Memory Management and Shuffle Service

Spark, which is one of the most powerful Big Data frameworks [8], uses memory to speed up large-scale data processing. In this section, we describe the latest memory management and shuffle service used in Spark (version 2.3) and provide cost analysis based on its current memory management and architecture.

A. Spark memory management optimization

In Spark, data is abstracted as resilient distributed datasets (RDD) [9], which represents a collection of objects partitioned across a set of compute nodes. RDDs include the lineage information, which can help Spark applications recompute the RDDs to ensure data reliability upon task failures. Spark tries to keep all RDDs in memory to ensure fast access to the data and uses the disk when memory space is insufficient.

Spark divides memory into two main regions, execution and storage memory. The execution memory is mainly used for storing the objects required during the execution of Spark tasks. The intermediate data of shuffle is stored in execution memory. Execution memory will not be evicted for storage memory purposes. The storage memory is a region of memory used for caching and for intermediate serialized data. We are facing the following challenges regarding to optimize the performance and computing resource utilization in Spark:

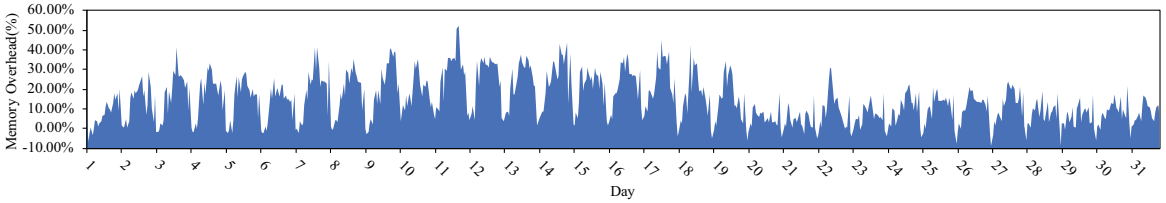


Fig. 1. Memory overhead of a production computing cluster with 3,700 servers

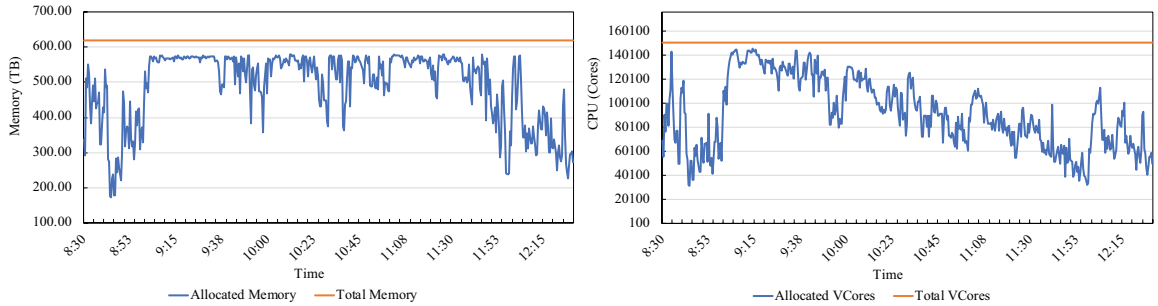


Fig. 2. Computing resources utilization of a production cluster with 3,700 servers from 8:30 AM to 12:30 PM

Tuning Challenge: Although computing resource tuning can help adjusting the amount of memory to the Spark application, it is hard to set the optimal configuration for every application because (i) enterprise data centers can run tens of thousands of applications in the computing cluster every day; and (ii) the size of input data vary every day.

Uneven utilization of computing resources: Current cluster is not typically designed for running in-memory big data frameworks and, as mentioned in previous sections, the utilization of the CPU is lower than the utilization of the memory. As a result, there is significant under-utilized CPU resource in the computer cluster while memory may not be sufficient to persist data in memory.

Spark has several optimization strategies, such as dynamic memory tuning at run time, which is called *"Dynamic Resource Allocation"*. In this strategy, Spark applications request computing resources back to the computing cluster if they are no longer being used, and they request computing resource again when needed [10]. However, we found that dynamic resource allocation cannot solve this problem in a production system for the following two reasons.

1) *Dynamic Resource Allocation* can only kill or start an executor to release currently unused resource early, but cannot address the waste of processing resources due to an imbalance between CPU and memory utilization. Fig. 2 shows the memory and CPU utilization of the production computing cluster from 8:30 AM to 12:30 PM during a work day. The memory utilization of the entire cluster can remain close to 100% for a long period, while utilization of the CPU ranges from 30% to 70%, representing a large waste of CPU resource.

2) In production environments, once a resource is freed, the scheduler may take a long time to get enough executors for the same Spark application because of computing resource racing. The memory utilization of the entire computing cluster reached 100%, so it cannot offer enough executor memory to the Spark applications in the following stages on time.

B. Spark shuffle management

The shuffle operation can re-distribute data to certain tasks [11], and it is involved in a large number of Spark operations. Shuffle is one of the most expensive operations in Spark, as it involves disk I/O, data serialization, and network I/O. We divide the main I/O costs of Spark shuffle into two parts:

Data spill cost: The output data from an individual mapper is kept in the memory until there is insufficient memory in JVM. Spark will spill data onto disks once the execution memory is insufficient. Moreover, Spark sorts the shuffling results based on the target partitions. Thus, with sort based shuffling, spill data can significant decrease the performance of Spark.

Shuffle read/write cost: Spark task writes/reads shuffle data to/from disks, which could introduce significant I/O cost.

C. Spark memory requirements

A Spark application is executed in stages. Specifically, it considers all operations before the shuffle phase to be one stage and all operations after to be another. If an operation does not require shuffling, it is considered one stage. Spark can suffer significant performance loss if it does not have sufficient memory to be allocated to every executor. On the other hand, in production it is possible to have large variability in the size of the input data for different tasks at the same stage. As a result, it is hard to balance the trade-off between performance and memory usage efficiency.

Fig. 3 shows the data distribution of one of our typical production Spark workloads at different stages. The input data size is organized into six layers, from 0 MB to 1 GB.

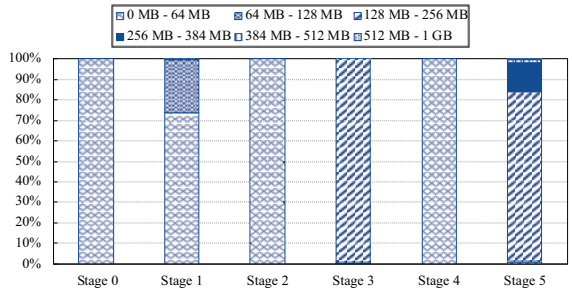


Fig. 3. Input data size at different stages of a production Spark application

The figure shows that the input data size of every task differs, while the allocated computing resources are the same in every executor. Thus, the allocated memory for every executor will be dominated by the largest partition in its tasks, which leads to memory under-utilization during data processing. An effective way to address this imbalanced memory requirement issue is to have a large shared memory pool to satisfy different executor's memory requirement.

We observed in our production system that most of the executor memory resource is used for two purposes: persistence and shuffling. Based on the observation, we focus on optimizing the shuffle and persistence mechanisms in Spark with in-memory distributed file system DMO and persistent memory (PMEM). DMO helps to improve the memory utilization while PMEM helps to increase the cluster's memory capacity with affordable cost. In the next section, we show that the IO throughput of PMEM can meet the requirements of our system.

IV. Characterizing Remote PMEM

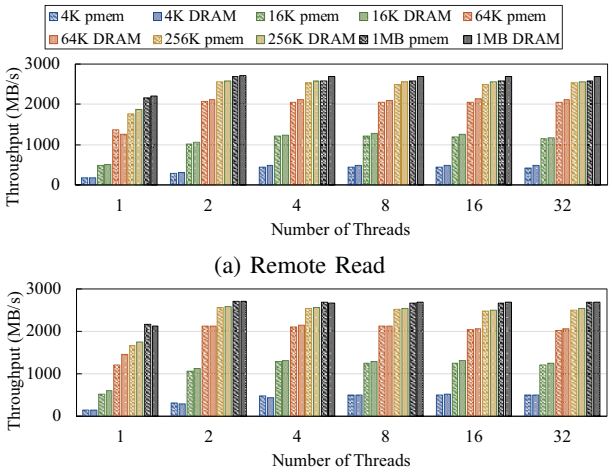
Recent architecture developments feature DIMM-based PMEM, which utilizes novel storage media to acheive data density that is much higher than DRAM. Moreover, the per GB cost of PMEM has also been projected to be much lower than that of DRAM. In most modern computer architectures, there is no layer between memory and storage, but there is a huge performance gap between memory and disk. PMEM has been proposed recently both by industry and academia to fulfill such performance gap. Therefore, PMEM can be considered as larger but slower memory or faster persistent storage. In this work, we empirically characterize and compare the read/write performance of PMEM and DRAM remotely. Furthermore, we explore whether the read/write throughput of PMEM is sufficient to be used as a remote disaggregated resource via RDMA technology.

A. Experimental setup

As our evaluation investigates the performance of using a disaggregated memory pool for Spark workloads, we mimick the scenario where Spark executors read from and write to remote PMEM: we set up a two-node cluster within a 25 GbE network (this is the current network bandwidth used across the whole data center). Among these two nodes, one node serves as the external pmem server, and the other emulates the Spark compute node. We let the external pmem server be equipped with two Intel Xeon Gold 6252 CPUs @ 2.10 GHz with 24 physical cores, 192 GB of DDR4 memory and 1.5 TB PMEM samples in DIMM form factor from a major vendor. The compute node uses two Intel Xeon E5-2650 v4 @ 2.2 GHz with 12 physical cores and 256 GB DDR4 memory. Both nodes use Mellanox ConnectX-4 Ethernet cards. The operating system is CentOS 7.5 with default 3.10 kernel that comes with PMEM support. PMEM in our system can be configured in different modes, allowing user applications to treat it as either volatile memory or persistent block or character device. In this work, we use the latter approach and configure all the PMEM as device DAX [12]. This further allows us to build a highly customized distributed in-memory storage system on top of the PMEM as we show in section V.

B. Remote PMEM performance

We further build an internal remote memory profiling tool that measures the throughput of single-sided RDMA read and write operations sent to the external memory node, and compare the remote access performance between PMEM and DRAM. To better understand the sequential read/write performance of remote PMEM, we use different data transfer sizes, from 4 KB to 1 MB, with the number of threads ranging from 1 to 32. Fig. 4 shows the remote read and write throughput of PMEM and DRAM, respectively.



(a) Remote Read

(b) Remote Write

Fig. 4. Remote RDMA read/write throughput for PMEM and DRAM using 25GbE network

The results indicate that both remote PMEM and DRAM access are able to almost saturate the 25Gb network bandwidth at higher I/O sizes. Moreover, PMEM is able to offer the same remote read/write throughput compared to DRAM as throughput starts being bottlenecked by the network as opposed to the bandwidth of PMEM itself.

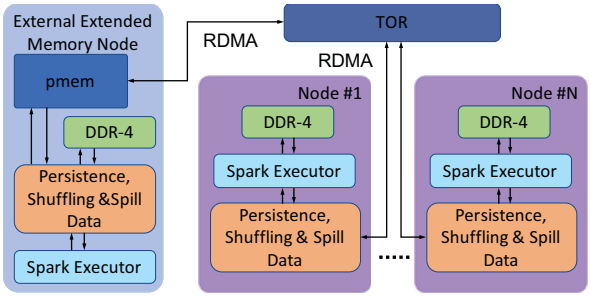


Fig. 5. Extended memory design with remote PMEM

C. PMEM viability for remote memory implementation

Although existing data centers deploy servers with different types of resources (e.g., CPU, DRAM, GPU, HDD, SSD, etc.) that are typically used within a server, our observations discussed above indicate that the server-centric architecture cannot fully utilize the computing resources available in the data processing clusters. For example, in current data centers, usually only a limited number of servers feature advanced storage devices and large volumes of memory while most servers are equipped with standard hardware configurations. Because of this, it is not realistic to approach a high performance and cost effective solution based on the traditional server-centric data center architecture. The advances and continuous cost reduction in communication networks (e.g., RDMA) enables computing nodes in modern data center to leverage remote resources efficiently. The analysis above clearly show that PMEM is a solid candidate for implementing extended remote memory solutions for enterprise data centers.

In this paper, we design and implement disaggregated memory pool with DRAM and PMEM. Fig. 5 shows the basic architecture of our system, co-designed with Spark. Our system stores all spill, persistence and shuffling data in the disaggregated memory pool. To solve the problem discussed in Sections II and III, we use a server featuring large volume of PMEM as supplemental memory for the disaggregated memory pool. Moreover, to ensure the high availability of remote PMEM, we connect the servers via fast fabric interconnection. In the next section, we present the detailed design and implementation of our system.

V. System Implementation

We present the design and implementation of two core components in our system: (1) a distributed in-memory storage system named distributed memory objects (DMO), and (2) integration of DMO with Spark via memory-speed RDD storage and a newly designed Spark shuffle manager.

A. Distributed Memory Objects

DMO is designed as next generation data infrastructure software optimized for memory-centric computing and high bandwidth networks. It aims at providing memory-speed data persistence and very fast data exchange that are highly demanded by distributed in-memory computation frameworks such as Spark. DMO offers a large PMEM pool by aggregating PMEM resources from its member nodes. For Spark, the pool becomes a disaggregated memory resource that extends the capacity of the off-heap memory for every Spark executor. Current Spark utilizes off-heap memory for accelerating RDD caching/storage as well as storing data used by shuffle tasks. With the help of PMEM as well as our RDMA-based networking framework, accessing any data in the pool has been made even faster than accessing local disks.

TABLE I
Major DMO client side APIs.

API	Description
`connect`	Establish connections with DMO backend.
`disconnect`	Drop an existing connection with DMO backend.
`create`	Create an empty object in DMO.
`write`	Write data to some offset of an existing object.
`read`	Read some offset of an existing object.
`get_attr`	Retrieve the attributes of an existing object.
`delete`	Delete an object from DMO.
`create_dir`	Creating an empty directory for holding objects.
`remove_dir`	Remove a directory.

At high level, DMO consists of client and storage backend. The client integrates with user applications through a set of APIs that performs data and metadata operations on storage backend. Storage backend is a remote cluster of multiple PMEM-equipped servers where each node executes one or more of the following DMO components: name server, and object store. Across nodes communications within DMO are purely through RDMA over Converged Ethernet (RoCE). In particular, we use single-sided RDMA write/read for data transfer, and use double-sided RDMA send/receive for RPC messages. We briefly describe each of the DMO components in the following paragraphs.

DMO client exposes a set of APIs for data and metadata operations. Currently, Java, C and C++ versions of object APIs have been implemented. Table I describes the major APIs used in this work. These APIs are sufficient for supporting the integration of DMO with Spark.

Name server records the locations of a DMO object's metadata and some other attributes. In DMO, metadata are stored in object store (described below) and records the address information of the object's data chunks. The name server further maintains another map that tracks information of directories such as what objects are contained inside a directory. Multiple name servers can be deployed inside a DMO cluster to handle a large number of objects. In such scenario, entries in name servers are sharded and replicated using consistent hashing [13].

A DMO object consists of metadata chunks and data chunks. All these chunks are held by object store. Metadata chunks mainly hold the address information of each data chunk, including node index, PMEM device index and the starting offset of the data chunk in the PMEM device. As all the data are stored in memory, data operations are done through load and store using memory copy. To guarantee data persistency, we use `clflush` instruction to flush CPU cache back to PMEM region [14].

Caching is still needed to achieve close-to-DRAM performance as PMEM is still lower in performance compared to DRAM. We chose to cache an object's metadata in DRAM when the object is accessed for the first time. Furthermore, when an object's data is accessed by a remote node, a copy of the related data chunks is cached on the remote node after the first read. Cached chunks are evicted either upon a timeout or when the available space for caching in the node is approaching a limit specified by user when configuring DMO.

B. Integrating DMO with Spark

We made two major efforts to integrate DMO with Spark: a modification of Spark to allow persisting RDD into DMO, and a Spark shuffle manager based on DMO.

1) Persisting RDD to DMO: In Spark, RDD can be cached in memory or disk in order to avoid recomputation of lineage [9]. By default, RDD is cached using the `persist` API. The caching policy specifying the media RDD will be placed on can be passed in as an input parameter. We modified Spark to allow RDD to be persisted directly into the external PMEM pool of DMO. This is done by adding a new RDD persist policy, while augmenting the implementation of the `persist` function of Spark's block manager. To further enable RDD retrieval from DMO, we modified the function for reading RDD data blocks in block manager. Besides enabling the use of remote PMEM for RDD storage, our modification further replaced the TCP/IP based Netty communication framework used for cross-node communication with our RDMA framework used by DMO.

2) Shuffle with DMO: We implement a customized shuffle manager based on DMO. Compared to default Spark shuffle manager, mappers write shuffle outputs directly to remote PMEM of DMO instead of to local DRAM and disks; reducers read shuffle output by pulling data directly from DMO instead of each Spark mapper node. All the remote data operations are efficiently carried out with our RDMA-based networking layer instead of the TCP/IP based Netty framework.

Building a pluggable shuffle manager requires us to implement the shuffle reader, the shuffle resolver and the shuffle writer components specified by the `ShuffleManager` trait of Spark. Fig. 6 illustrates the structure of our implementation. A shuffle task is performed in two consecutive stages: map and reduce. During map stage, a mapper uses `DMOShuffleWriter` to produce a shuffle output. The output is made of an index file and a data file, which are stored as separate objects by DMO. Data file contains multiple partitions, and each partition stores the data to be read by one reducer. Offsets marking the starting positions of the partitions in data file are kept in index file. To allow reducer to be able to retrieve needed partitions correctly, mapper registers with `DMOShuffleBlockResolver`, assigning the partitions of the mapper to corresponding reducers. In reduce stage, each reducer uses `DMOShuffleReader` to retrieve all the partitions belonging to the reducer following the guidance of `DMOShuffleBlockResolver`. The reducer then merges all the partitions together.

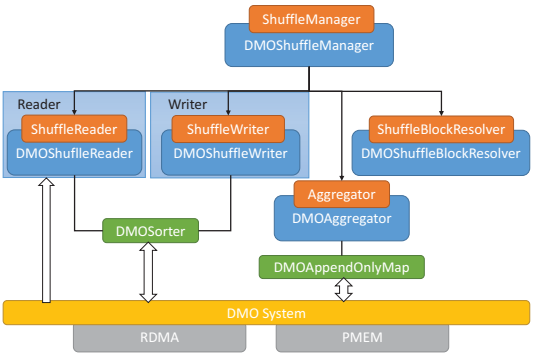


Fig. 6. The structure of DMO-based shuffle manager

TABLE II
INPUT SIZE AND CHARACTERISTIC OF WORKLOADS

Workload	Input Size	Characteristic
Terasort	600 GB	I/O intensive
Warehouse application	200 GB	I/O intensive
price protection application	726.9 GB	I/O, CPU intensive

Both `DMOShuffleWriter` and `DMOShuffleReader` utilize `DMOSorter`. A sorter receives partitions written by mapper or read by reducer, and insert them into an in-memory collection for fast computation. During the insertion, it performs aggregation and sorting if specified. However, when memory assigned to a sorter is not sufficient for holding all the data, spilling part of the data to disk is required to avoid out-of-memory (OOM) failure. In our implementation, `DMOSorter` starts spilling part of the in-memory data by serializing and writing them to DMO. Therefore, DMO objects holding spilled data are treated as part of the spilled collection of a `DMOSorter`.

An aggregator is used when shuffle is triggered by "group by" computations. Depending on the characteristics of input data, aggregator of default Spark shuffle manager may also spill in-memory data to local disk when there is not enough memory. Similar to `DMOSorter`, we implement our own `DMOAggregator` to speed up data spill by directly writing spilled data into DMO. We also implemented some custom logic to avoid out of memory issues when the result of aggregation consumes too much memory.

VI. Experimental Evaluation

A. Workloads and experimental setup

We selected three different workloads to evaluate the performance, memory efficiency, and scalability of our system. Table II shows the input size and characteristics of the workloads, and Table III shows the shuffling size and persistence RDD size of the workloads.

Terasort: We select TeraSort from HiBench [15]. TeraSort is a standard shuffling intensive benchmark well suited to evaluate the I/O performance (especially shuffling performance) of Big Data frameworks, such as Spark.

TABLE III
SHUFFLING SIZE AND PERSISTENCE RDD SIZE OF WORKLOADS

Workload	Shuffling Size	Persistence Size
Terasort	334.2 GB	N/A
Data warehouse application	234.7 GB	N/A
Price protection application	57.6 GB	349 GB

Core service of data warehouse application: We selected a typical data warehousing scenario in the E-commerce company that provides core data joining and aggregation services to various businesses, including user information, order information, shipping information, storage information, etc. It consists of more than 150 Spark SQL-based applications. As it provides core data to a large amount of downstream customers with explicitly defined SLA, reducing its execution time is critical. This workload is I/O-intensive (both disk and network), as it is based on the *select*, *insert*, and $fullouterjoin$ operations.

Price protection service: This is a core service of large E-commerce companies that typically suffers frequent price DDoS attacks, such as coordinated product price crawling. The price protection application can find abnormal information, e.g., IP addresses, using several different strategies. With these information, the price protecting strategy can help data scientists make better decisions regarding product price to minimize the loss from price DDoS. Because the price protection application reuses RDDs tens of times, it is both I/O intensive and computing intensive.

The experimental setup includes one server featuring PMEM technology and 10 regular production enterprise servers. The regular server is equipped with two Intel(R) Xeon(R) CPU E5-2650 v4 @ 2.20GHz with 24 cores leveraging hardware threads, and 256 GB of DDR-4 RAM memory. What's more, Spark version 2.3 was deployed using standalone mode, CentOS 7.5 and HDFS version 2.7.

B. Performance evaluation results

We evaluate the performance of the three workloads with 60 Spark executors, each using five cores and a different amount of memory varying from 1 GB to 20 GB. We determine the total memory utilization from the allocated executor memory in Spark and the storage memory in DMO, i.e.,

$$\text{Mem}_{\text{total}} = \text{Mem}_{\text{Spark}} + \text{Mem}_{\text{DMO}}$$

Note that the PMEM usage is also accounted into the total memory utilization.

Based on the above experimental setup, we evaluate the performance of our system with four configurations: (1) Spark, (2) Spark with DMO Local (all DMO data chunks written/read to/from local DRAM, no remote PMEM), (3) Spark with DMO Remote (all DMO data chunks written/read to/from remote PMEM), and (4) Spark with DMO Local with caching.

Overall performance. Fig. 7 shows the execution time of Spark with different executor memory and DMO memory using different workloads . For TeraSort, default Spark cannot successfully finish until the executor memory is more than 10

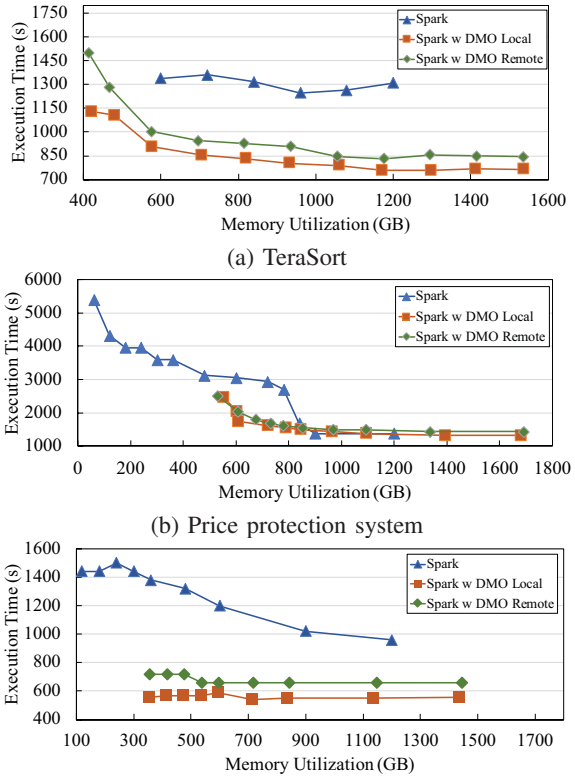


Fig. 7. Total memory - Execution time of TeraSort, price protection system and data warehouse application

GB (total memory is 600 GB). This is because (1) the tasks cannot obtain enough local memory which leads to significant Java GC, even OOM, and (2) executor connections are closed because the bandwidth of local disk is too low. In consequence, as we can see from Fig. 7, the minimum required memory for default Spark is 600 GB, while our proposed system is 410 GB. The most expensive operation in TeraSort is shuffling, because Spark must write/read large amounts of shuffling data on disk. With our optimization of shuffling (i.e., external shuffling memory instead of disk), our system can reduce the execution time by up to 40% compared to default Spark with the same memory consumption. The figures also show that the performance is similar comparing using DMO with local DRAM and with a disaggregated remote PMEM pool.

The price protection system first generates the intermediate RDDs, which are re-used tens of times in the subsequent stages. We store persistent RDDs with serialized Java objects in Spark. Our system stores shuffling data, spills, and persistence data in DMO, which is backed by a distributed DRAM and PMEM memory pool, which can offer sufficient storage memory to Spark. On the other hand, the re-computation strategy enables default Spark to finish all stages with insufficient storage memory for the JVM. However significant execution overhead was introduced because of re-computation. With a greater executor memory capacity, default Spark can achieve

the best performance with more than 14 GB memory per executor, with a total of 840 GB of allocated memory. The best performance of our proposed system offers a 4.3% execution time reduction with the same memory utilization. Although we do not optimize the persistence strategy of Spark, the system provides performance improvement from garbage collection reduction. Note that comparing to default Spark, the proposed system with DMO needs larger minimum total memory to finish the workload, as the additional memory is used to persist all intermediate RDDs with disaggregated memory pool in these experiments.

The core service of the data warehouse application is a typical Spark SQL application that involves a large volume of data. This workload profile is similar to the TeraSort application. As shown in Fig. 7, our proposed system can reduce the execution time up to 59.5% with same amount of memory compared with default Spark.

Computing resource efficiency. As discussed in Section II, the data center memory utilization is usually higher than the CPU utilization in data center. As each regular production server features 256 GB of DRAM already, increasing the memory size further may not be possible.

We use one server featuring large-capacity PMEM to increase the overall memory size and therefore optimize the use of computing resources. Specifically, we increase by 66.5% the overall memory capacity of the cluster with 10 regular nodes, by adding one server with 192 GB of DRAM and 1.5 TB of PMEM. The corresponding performance overhead of TeraSort, price protection, and data warehouse application is only 10.5%, 9.1%, and 18% at the worst scenario: all DMO read from/written to remote PMEM, comparing to all DMO read from/written to local DRAM.

Additionally, because of the disaggregated design, the extended memory is shared across different executors in a Spark application for persistence and shuffling. This eases the performance tuning in the production scenario, and improves the overall memory utilization, especially for imbalanced data partition cases.

Quality of service discussion. To meet Service-level Agreements (SLAs) in enterprise data centers, system engineers tend to assign large volumes of memory to Spark applications; although in some cases, Spark can finish jobs with lower memory requirements. This usually leads to memory resource waste. For example, assuming a price protection application is required to finish in less than 2,000 seconds, an engineer would allocate 840 GB of memory to the application using default Spark, and still worry about that occasional OOM due to input data change can ruin the SLA. As the comparison, the memory requirement of our proposed system is only around 600 GB, and is more scalable to data change.

Impact of caching. Fig. 8 shows that with caching optimization, TeraSort can further reduce its execution time by 52% with the same amount of overall memory, which is up to 3.5-fold performance improvement comparing to default Spark. However, for the price protection and data warehouse applications, the caching does not show obvious benefits. This

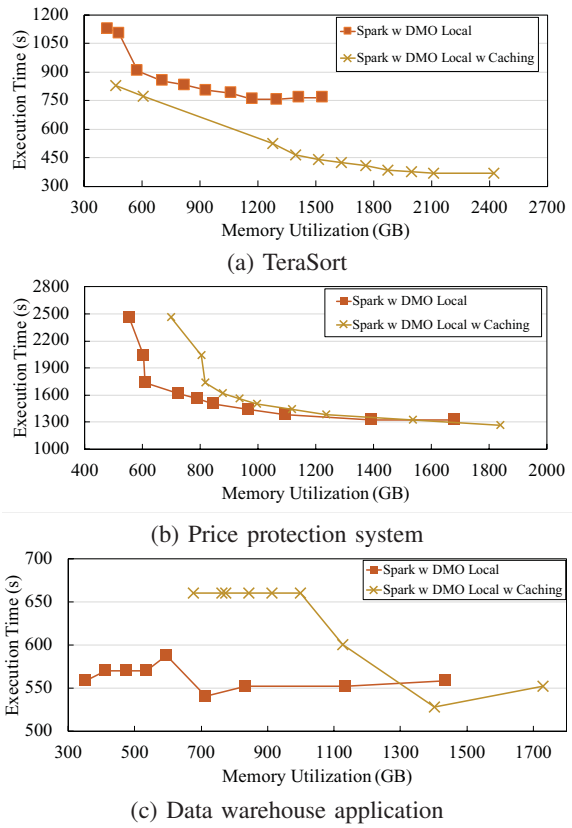
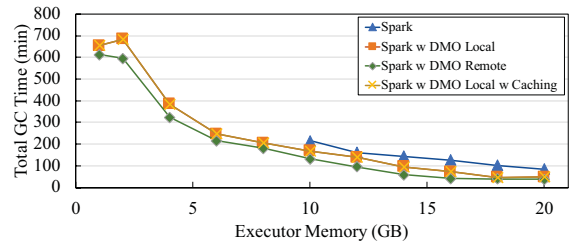


Fig. 8. Total memory - Execution time of TeraSort, price protection system and data warehouse application

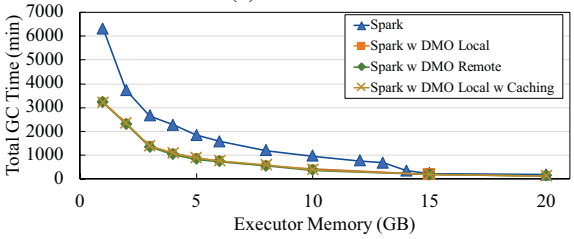
is because TeraSort does the sort operation at reduce side, which produces heavy IO loads. Thus, TeraSort can benefit from caching while the other two workloads can not.

Impact of garbage collection (GC). We observed in production environments that the GC of a Spark application can represent a significant fraction of its execution time. The most common approach to avoid performance degradation due to GC is to increase the size of executor memory in Spark. Fig. 9 shows the total GC time of default Spark and Spark with DMO. As shown in the figure, our proposed system can save up to 42% of the total GC time for TeraSort and price protection applications with the same amount of executor memory, because the persistence and shuffling data are offloaded to DMO. However, for the data warehouse application with low memory utilization, the GC time in Spark with DMO could be larger than default Spark. This is because we do not implement the `BypassMergeSortShuffleWriter` in DMO, which is used when number of partitions is no more than 200 by default, which is the case for the data warehouse application. In general, with the same executor memory size, DMO (external extended memory) is expected to decrease the GC time.

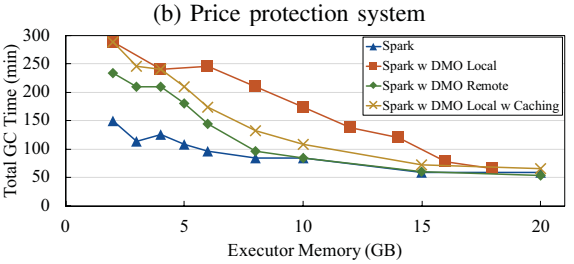
Discussion of N+1 architecture. We propose a system with a disaggregated memory pool, which can utilize the extended PMEM with N+1 architecture (N regular nodes with



(a) TeraSort



(b) Price protection system



(c) Data Warehouse application

Fig. 9. Executor memory - GC time of TeraSort, price protection system and data warehouse application

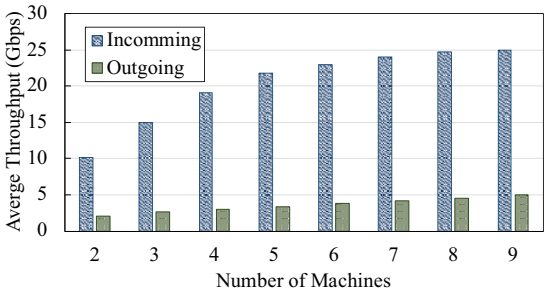


Fig. 10. Average network throughput of PMEM-based server with TeraSort

one PMEM node). We evaluated the network throughput of the server featuring PMEM in our system, with TeraSort application. We increase the number of executors and the size of data accordingly as the number of regular nodes used for computation increases. Fig. 10 shows the average incoming throughput for shuffle write phase and outgoing throughput for shuffle read phase on the single remote PMEM server. We aim at finding a trade-off between the bandwidth of the computing cluster and the data center's computing resource efficiency. As shown in Fig. 10, as the application load increases, the shuffle write throughput can reach the available network bandwidth in our system. This also provides meaningful data points for identifying efficient enterprise data center design choices.

VII. Related Work

Characterization of Big Data data center: Complementing existing research that analyzed the resource utilization of compute clusters and data centers [16], [17], our work provides a characterizations of a large scale production data center in a large E-commerce company running big data applications and addresses relevant bottlenecks that limit the capabilities of in-memory Big Data frameworks.

Memory efficiency and disaggregated data center: Current research address disaggregated resources with advanced infrastructure to increase the performance and compute resource efficiency for Big Data frameworks. Existing research [18]–[21] has explored the potential of RDMA for Spark. Gao et at. [22] address the network requirements of disaggregated data centers, and compare the performance loss of different network latency and bandwidth. Rao et al. [23] compare the performance of Spark SQL with different memory bandwidth. Industry has already proposed disaggregated data center architectures with PCIe and fast fabric interconnects such as the Intel Rack Scale Design (RSD) for rack-scale disaggregation [24], HP's "The Machine" concept [25], and Facebook's proposed disaggregated data center [26].

Optimizing the shuffle service: Zhang et al. [27], [28] optimize Spark with disaggregated storage (disk) and external shuffle service in Facebook. To avoid overfit into the local disk, they store temporary data into distributed file system. Google has also optimized the shuffle service with the Cloud Dataflow service.

Burst buffers for Big Data framework: Existing literature has explored the potential of HPC burst buffers for Big Data frameworks. For example, Yildiz et al. [29] propose Eley, which leverage burst buffers and data prefetching for accelerating Big Data applications. Chaimov et al. [20] use NVMe SSD buffers between compute nodes and Lustre file systems in order to improve the scalability of the Spark framework. Islam et al. [21] propose using Memcached as a burst buffer system to integrate HDFS with the Lustre file system in a performant and efficient way and evaluate the performance of NVMe SSD based HDFS [30].

VIII. Conclusion and Future Work

In this paper, we present the design and implementation of a disaggregated memory system with persistent memory for an in-memory Big Data processing framework. Our proposed system optimizes the memory efficiency of the data center with external extended persistent memory and a disaggregated memory pool. By leveraging shuffle and persistence optimization, we demonstrate that our system can effectively reduce the execution time by up to 72% for shuffle-intensive applications. Moreover, we incorporate shuffle and persistence mechanisms into Big Data frameworks using an in-memory distributed file system. The results of the experimental evaluation from empirical executions show that with remote large-volume persistent memory and a disaggregated memory pool, our proposed system can also increase the overall memory capacity by 66.5% with much lower overheads. While the experimental

evaluation has been conducted using a 25 GbE commodity network, we expect our proposed system to provide significantly better results using faster networks (e.g., 40/100 GbE or Intel Omnipath technology). This actually shows the viability of PMEM for implementing bare-metal software-defined infrastructures in production enterprise environments.

Based on this proof of concept work, the large E-commerce company is looking at deploying the solution with persistent memory in production for shuffle/persistence intensive Spark applications with high SLA requirements. The company is also deploying Spark on container and Kubernetes based private cloud environment, mixed with other workloads to improve the machine resource utilization. The solution fits nicely to the deployment as it gets rid of the dependency on the local disks which is difficult to virtualize and manage in the environment. With Spark executor in cloud, the solution provides a truly elastic way to run Spark application.

ACKNOWLEDGMENTS

We thank Jie Li for his help with our experimental setup. We are indebted to Yue Zhao, Wei Kang, Ning Xu, and Haiyan Wang for their help with developing and some of our changes to DMO. Finally, we thank JD cloud and Charles Fan for their support of this project. The research at Rutgers was conducted as part of the Rutgers Discovery Informatics Institute (RDI^2) and is supported in part by NSF via grants numbers OAC 1640834, OAC 1826997, OAC 1835692, and OCE 1745246.

REFERENCES

[1] Nvidia. Gpu. https://www.nvidia.com/en-us/about-nvidia/ai-computing/. Accessed March, 2019.

[2] Intel. Optane ssd. https://www.intel.com/content/www/us/en/architecture-and-technology/intel-optane-technology.html. Accessed March, 2019.

[3] Sandhya Dwarkadas, Nikolaos Hardavellas, Leonidas Kontothanassis, Rishiyur Nikhil, and Robert Stets. Cashmere-vlm: Remote memory paging for software distributed shared memory. In *International Symposium on Parallel and Distributed Processing*, pages 153–159, 1999.

[4] Michail D Flouris and Evangelos P Markatos. The network ramdisk: Using remote memory on heterogeneous nows. *Cluster Computing*, 2(4):281–293, 1999.

[5] Shuang Liang, Ranjit Noronha, and Dhabaleswar K Panda. Swapping to remote memory over infiniband: An approach using a high performance network block device. In *IEEE Cluster Computing*, pages 1–10, 2005.

[6] Evangelos P Markatos and George Dramitinos. Implementation of a reliable remote memory pager. In *USENIX Annual Technical Conference*, pages 177–190, 1996.

[7] Intel. Persistent memory. https://software.intel.com/en-us/persistent-memory. Accessed March, 2019.

[8] Matei Zaharia, Mosharaf Chowdhury, Michael J Franklin, Scott Shenker, and Ion Stoica. Spark: Cluster computing with working sets. *HotCloud*, 10(10-10):95, 2010.

[9] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauly, Michael J. Franklin, Scott Shenker, and Ion Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proceedings of the 9th USENIX Symposium on Networked Systems Design and Implementation*, pages 15–28, San Jose, CA, 2012.

[10] Spark. Spark dynamic resource allocation. https://spark.apache.org/docs/latest/job-scheduling.html#dynamic-resource-allocation. Accessed March, 2019.

[11] Spark. Shuffle operations. https://spark.apache.org/docs/latest/rdd-programming-guide.html#shuffle-operations. Accessed March, 2019.

[12] Dan Williams. "Device DAX" for persistent memory. https://lwn.net/Articles/687489/. Accessed March, 2019.

[13] David Karger, Eric Lehman, Tom Leighton, Rina Panigrahy, Matthew Levine, and Daniel Lewin. Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the world wide web. In *Proceedings of the Twenty-ninth Annual ACM Symposium on Theory of Computing*, STOC '97, pages 654–663, 1997.

[14] Andy Rudoff. Persistent memory: The value to hpc and the challenges. In *Proceedings of the Workshop on Memory Centric Programming for HPC*, pages 7–10, New York, NY, USA, 2017. ACM.

[15] Shengsheng Huang, Jie Huang, Jinquan Dai, Tao Xie, and Bo Huang. The HiBench benchmark suite: Characterization of the mapreduce-based data analysis. In *Data Engineering Workshops (ICDEW), 2010 IEEE 26th International Conference on*, pages 41–51. IEEE, 2010.

[16] Abhishek Verma, Luis Pedrosa, Madhukar Korupolu, David Oppenheimer, Eric Tune, and John Wilkes. Large-scale cluster management at google with borg. In *Proceedings of the Tenth European Conference on Computer Systems*, page 18. ACM, 2015.

[17] Yunqi Zhang, George Prekas, Giovanni Matteo Fumarola, Marcus Fontoura, Íñigo Goiri, and Ricardo Bianchini. History-based harvesting of spare cycles and storage in large-scale datacenters. In *Proceedings of the 12th USENIX conference on Operating Systems Design and Implementation*, pages 755–770, 2016.

[18] Xiaoyi Lu, Dipti Shankar, Shashank Gugnani, and Dhabaleswar K DK Panda. High-performance design of apache spark with rdma and its benefits on various workloads. In *Big Data (Big Data), 2016 IEEE International Conference on*, pages 253–262. IEEE, 2016.

[19] Xiaoyi Lu, Md Wasi Ur Rahman, Nusrat Islam, Dipti Shankar, and Dhabaleswar K Panda. Accelerating spark with rdma for big data processing: Early experiences. In *High-performance interconnects (HOTI), 2014 IEEE 22nd annual symposium on*, pages 9–16. IEEE, 2014.

[20] Nicholas Chaimov, Allen Malony, Shane Canon, Costin Iancu, Khaled Z Ibrahim, and Jay Srinivasan. Scaling spark on hpc systems. In *Proceedings of the 25th ACM International Symposium on High-Performance Parallel and Distributed Computing*, pages 97–110. ACM, 2016.

[21] Nusrat Sharmin Islam, Dipti Shankar, Xiaoyi Lu, Md Wasi-Ur-Rahman, and Dhabaleswar K Panda. Accelerating i/o performance of big data analytics on hpc clusters through rdma-based key-value store. In *Parallel Processing (ICPP), 44th International Conference on*, pages 280–289. IEEE, 2015.

[22] Peter Xiang Gao, Akshay Narayan, Sagar Karandikar, Joao Carreira, Sangjin Han, Rachit Agarwal, Sylvia Ratnasamy, and Scott Shenker. Network requirements for resource disaggregation. In *OSDI*, volume 16, pages 249–264, 2016.

[23] Pramod Subba Rao and George Porter. Is memory disaggregation feasible?: A case study with spark sql. In *Proc. of the 2016 Symp. on Architectures for Networking and Communications Systems*, pages 75–80, 2016.

[24] Intel. Intel rack scale design. https://www.intel.com/content/www/us/en/architecture-and-technology/rack-scale-design-overview.html. Accessed March, 2019.

[25] hp. Hp the machine. http://www.labs.hpe.com/research/themachine/. Accessed March, 2019.

[26] Jason Taylor. Facebooks data center infrastructure: Opencompute, disaggregated rack, and beyond. In *Optical Fiber Communication Conference*, pages W1D–5. Optical Society of America, 2015.

[27] Haoyu Zhang, Brian Cho, Ergin Seyfe, Avery Ching, and Michael J Freedman. Riffle: optimized shuffle service for large-scale data analytics. In *Proceedings of the Thirteenth EuroSys Conference*, page 43. ACM, 2018.

[28] Brian Cho. Taking advantage of a disaggregated storage and compute architecture. https://databricks.com/session/taking-advantage-of-a-disaggregated-storage-and-compute-architecture. Accessed March, 2019.

[29] Orcun Yildiz, Amelie Chi Zhou, and Shadi Ibrahim. Improving the effectiveness of burst buffers for big data processing in hpc systems with eley. *Future Generation Computer Systems*, 86:308–318, 2018.

[30] Nusrat Sharmin Islam, Md Wasi-ur Rahman, Xiaoyi Lu, and Dhabaleswar K Panda. High performance design for hdfs with byte-addressability of nvm and rdma. In *International Conference on Supercomputing*, page 8, 2016.