2018 30th International Symposium on Computer Architecture and High Performance Computing

Exploring the Potential of Next Generation Software-Defined In-Memory Frameworks

Shouwei Chen, Ivan Rodero
Rutgers Discovery Informatics Institute (RDI^2), Rutgers University
{shouwei.chen, irodero}@rutgers.edu

***Abstract*—As in-memory data analytics become increasingly important in a wide range of domains, the ability to develop large-scale and sustainable platforms faces significant challenges related to storage latency and memory size constraints. These challenges can be resolved by adopting new and effective formulations and novel architectures such as software-defined infrastructure. This paper investigates the key issue of data persistency for in-memory processing systems by evaluating persistence methods using different storage and memory devices for Apache Spark and the use of Alluxio. It also proposes and evaluates via simulation a Spark execution model for using disaggregated off-rack memory and non-volatile memory targeting next-generation software-defined infrastructure.**

Experimental results provide better understanding of behaviors and requirements for improving data persistence in current in-memory systems and provide data points to better understand requirements and design choices for next-generation software-defined infrastructure. The findings indicate that in-memory processing systems can benefit from ongoing software-defined infrastructure implementations; however current frameworks need to be enhanced appropriately to run efficiently at scale.

I. Introduction

The rapid growth of the sheer quantity of digital data generated every day through the internet has motivated the science, engineering, and industry communities to develop big data processing frameworks. Apache's Hadoop is one of the most popular big data frameworks and is based on the MapReduce model for distributed processing of large scale datasets using an affordable infrastructure. However, the increasing volume and rate of data [1], along with the associated costs in terms of latency, quickly limit the ability of data analytics applications to leverage this data in an effective and timely manner. For example, new requirements in different areas of science and engineering for supporting quick response analytics (e.g., machine learning algorithms) of data being produced or collected at high rates (e.g., real-time streaming data) are pushing application formulations for big data toward in-memory processing solutions. Apache Spark has become increasingly popular due to its in-memory and directed acyclic graph (DAG) execution engine. Further, Spark provides the abstraction of resilient distributed datasets (RDD), which is essential to delivering high-performance capabilities while remaining compatible with existing Hadoop ecosystems, e.g., the Hadoop distributed file system (HDFS) and a number of high-level APIs in Scala, Java, and Python. Another example is Alluxio (formerly known as Tachyon [2]), which delivers a distributed file system for reliable in-memory data sharing. However, the increasing amount of memory and power required to run complex data-centric applications and workflows using in-memory processing systems are pushing the community to explore novel system architectures (e.g., deeper memory hierarchies) and new application formulations. Furthermore, DRAM memory represents a large portion of the investment for building datacenter IT infrastructure, especially when they target in-memory processing systems.

Understanding the limitations of current in-memory processing frameworks and the ability to trade off response time, power consumption, and infrastructure cost is an important concern; however, exploring system design choices for enabling next generation infrastructure to effectively support these platforms and, in turn, co-designing new software frameworks is paramount. Ongoing processor architectures, non-volatile technologies such as Intel Optane NVMe, and advances in integrated silicon photonics promise systems capable of delivering off-node non-volatile memory latency and bandwidth comparable to PCIe-based in-node access [3], which is essential for the implementation of software-defined infrastructures. Software-defined infrastructure is based on the concept of resource desegregation. Previous work has explored requirements for building disaggregated datacenters [4]–[7], taking into consideration flash [8] and DRAM memory [9], [10] resources as well as RDMA technologies [11]. Although network fabric latency and bandwidth has been a major blocker for the adoption of software-defined infrastructures, recent technological advances such as NVMe over fabrics [12], Intel Omni-Path, and Mellanox Quantum 200G HDR exemplify a step forward toward this vision. Rack scale design (RSD) architecture is one of the realizations of software-defined infrastructure. Intel RSD uses a high-performance PCIe to reduce data transmission time within the same rack. However, the data transmission time across different racks is still challenging. Next-generation SDI is expected to deliver lower latency and higher bandwidth interconnects for desegregated resources (e.g., compute, memory, and storage) within datacenters. Our previous research [13] explored the potential of software-defined infrastructure for HDFS targeting different storage technologies (e.g., NVMe); however, exploring the potential of next generation software-defined infrastructure for in-memory frameworks has become a critical concern.

This paper focuses on understanding the behaviors and limitations of current in-memory frameworks, thus leading to insights regarding design choices toward next-generation

978-1-5386-7769-8/18/$31.00 ©2018 IEEE
DOI 10.1109/SBAC-PAD.2018.00042

software-defined in-memory frameworks. Our empirical experimental evaluation focuses on persistence methods for Spark and the use of Alluxio. The contributions of this paper revolve around an empirical evaluation of Spark persistence methods using different storage technologies and Alluxio, which answers questions related to behaviors and limitations of current in-memory processing systems, provides meaningful data points to better understand requirements and design choices for next-generation software-defined infrastructure and explores the use of disaggregated off-rack memory and NVMe via simulation due to the limitations of current network fabric interconnects.

The rest of the paper is organized as follows. Section II discusses persistence methods for in-memory processing frameworks. Sections III and IV describe the methodology used to evaluate Spark persistence and the results, respectively. Section V presents the simulation evaluation results considering off-rack NVMe and memory. Section VI presents an overview of the literature and section VII concludes the paper and describes future work.

II. Understanding In-Memory Big Data Frameworks

A. Spark Persistence

Spark RDD persistence (or caching) is one of Spark's key capabilities, allowing it to deliver low latency and high performance. It allows users to store intermediate RDD into memory, disk, or a combination of memory and disk and reuse them in other actions on that dataset. Spark persistence technology can dramatically increase the speed of Spark applications (often by more than 10x [14]) with proper configuration and using programming best practices, especially in iterative applications.

Spark uses several approaches to RDD persistence. Memory can be used to store RDDs as de-serialized Java objects when they have enough memory space to store intermediate RDD datasets. Because RDD datasets are stored as de-serialized Java objects in memory, the required memory space has to be estimated based on an "expansion Index" and the size of the original dataset. Using memory with de-serialized Java objects is the fastest Spark persistence mechanism when sufficient memory space is available. Spark provides three different approaches to persist RDDs when insufficient memory space is available. The most straightforward way is to use block storage (e.g., disk) as an addition to memory. Spark can also store a whole RDD into block storage; however, in most cases, this approach is slow. RDDs can also be stored as serialized Java objects, which is less CPU-efficient compared to using de-serialized Java objects. Furthermore, Spark can also use off-heap memory (i.e., memory that is outside of executors) as storage space for persistent RDD.

B. Spark Persistence Memory Management

Spark memory management is based on Java Virtual Machine (JVM) memory management. Spark divides the memory into execution and storage, based on the different memory usages. The memory used for computation in shuffles, joins, sorts, and aggregations is considered execution memory and the memory used for caching (Spark RDD persistence with memory) and internal data transfers within clusters is considered storage memory. Former versions of Spark (version <1.6) implemented persistence using static memory management, which used differentiated memory spaces for execution and storage memory. With this approach, storage memory cannot use execution memory even when the execution memory is not utilized. To resolve this drawback, Spark introduced the current unified memory management, which merges execution and storage memory.

Important aspects of unified memory management in Spark include the following: (1) Spark reserves 300 MB of memory for the system as default, (2) unified memory represents the execution and storage memory, which can be customized ($spark.memory.fraction$, default 0.6), and (3) the rest of the heap memory includes user data structures, internal metadata in Spark, and safeguarding against out-of-memory errors (default 0.4). In comparison with static memory management, unified memory management allows storage memory to use more memory when demand for execution memory is not high. More importantly, execution memory can evict storage memory if there is not enough memory in the unified space, while storage memory has minimum space protection.

C. Spark Data Locality and Delay Scheduler

Data locality plays an important role in Spark. It specifically organizes the data into three main levels based on locality: (1) data can be fetched from the same JVM as the running executor, (2) data can be fetched from the same node (data from other executors in the same node, e.g., HDFS and Alluxio data in the same node), and (3) data can be fetched from the same rack of the cluster. The first option is clearly the fastest, the second a bit slower, and the last one is the slowest because datasets must be sent over the network.

Spark checks whether there are any available datasets in the same JVM as the running executer. If Spark cannot find datasets in the same JVM as the running executor, then it checks for data within the same node; otherwise, it checks for data in the same rack.

D. Alluxio

Alluxio (formerly known as Tachyon) is a distributed in-memory storage system; it has an API similar to HDFS but aims at accelerating big data frameworks by using memory technology as the main substrate for implementing the distributed file system. In this paper, we use Alluxio to accelerate Spark executions and implement persistency models that have the potential for supporting data coupling between multiple Spark applications. However, there are some important differences between Alluxio and Spark persistence models. On the one hand, predictability (e.g., estimating execution time) is more complex in Spark as its persistence mechanisms, such as unified memory management, evict storage memory when possible. On the other hand, Spark cannot easily share intermediate data (i.e., RDDs) with other applications and platforms. In this paper, we evaluate and compare the performance of Alluxio and Spark persistence models to understand software design issues that should be considered in future implementations targeting next-generation software-defined infrastructure.

III. Evaluation Methodology

This section describes three aspects of the experimental evaluation methodology. First, we evaluate Spark RDD persistence using different methods and storage technologies: memory ($MEMORY\_ONLY$), memory with serialized Java objects ($MEMORY\_ONLY\_SER$), and disk only ($DISK\_ONLY$), using NVMe and hard disk. Please note that storing RDD datasets with serialized Java objects requires trading off CPU utilization and I/O speed. We explore the need for using serialized Java objects when using NVMe technology. Second, we compare the traditional disk-based distributed file system that uses HDFS with an in-memory virtual distributed storage system (Alluxio). Alluxio is not intended to replace persistent distributed storage systems; rather, it provides a faster intermediate storage layer that interfaces with other file systems (e.g., HDFS, Amazon S3) and big data frameworks (e.g., Spark) to speed up data access. However, we compare these two approaches to understand the potential tradeoffs between cost and performance. Third, we study Spark RDD persistence with Alluxio. We use $MEMORY\_ONLY$ and $MEMORY\_ONLY\_SER$ as storage levels for Spark, and Alluxio uses memory as its only storage device. Finally, based on the results obtained from the empirical evaluation, we explore via simulation different scenarios using remote memory and NVMe for Spark data persistence.

A. Testbed and System Configuration

The empirical executions were conducted on the NSF-funded research instrument computational and data platform for energy-efficiency research (CAPER). CAPER is an eight-node cluster based on a SuperMicro SYS-4027GR-TRT system with a flexible configuration. The servers have two Intel Xeon Ivy Bridge E5-2650v2 (16 cores/node) and the configuration used in this work includes 128GB DRAM, 1TB Flash-based NVRAM (Fusion-io IoDrive-2), 2TB SSD, 4TB hard disks (as a RAID with multiple spindles, as recommended by best practices), and both 1GbE and 10GbE and Inbiniband FDR network connectivity. This platform mirrors key characteristics of datacenter infrastructure, which will allow us to extrapolate our models to larger systems and make projections.

We configured the big data framework and distributed storage file system as baselines using commonly used and balanced configurations without specific optimization. The characteristics of the system configuration are described as follows. Spark version 2.0 was deployed using YARN. One server was configured as Master and six servers were configured as Slaves. Hadoop version 2.7 (with HDFS) was deployed using YARN. One server was configured as the Name Node and six servers were configured as Data Nodes. Alluxio version 1.5 was deployed on seven servers, with one server configured as Master and six nodes configured as Workers. The system was configured with 24 executors, using 4 cores each. The JVM memory was set to 20GB and 16GB for Spark Driver and Executor, respectively.

B. Workloads and Data Set

The evaluation is focused on three types of benchmarks: 1) LineCounter (I/O intensive application), 2) WordCount-reduceByKey (I/O and network intensive application), and 3) WordCount-groupByKey (network intensive application). All of these three workloads are based on real data from Wikipedia in text format. Furthermore, LineCounter is the most I/O intensive and WordCount-groupByKey is the most network intensive of these benchmarks. The executions are conducted using data sets of different sizes for different benchmarks, from 10GB to 250GB (50GB-250GB for LineCounter, 10GB-250GB for WordCount-reduceByKey and 10GB-40GB for WordCount-groupByKey).

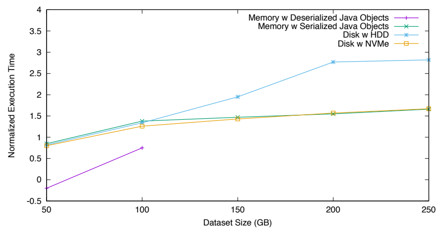
IV. Experimental Results

A. Spark RDD Persistence

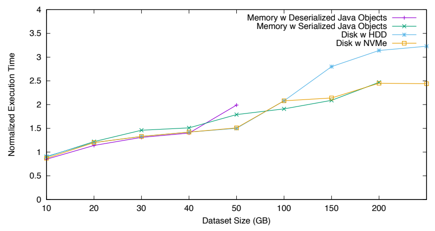
This section explores the impact on Spark RDD persistence performance of using different storage technologies. Although Spark RDD persistence can be configured using different storage levels, including memory, Spark clusters cannot always provide sufficient memory capacity for co-locating the application and persisted datasets. When insufficient memory is available in the system, Spark uses additional storage levels such as the hard disk to store RDDs as serialized Java objects, i.e., it extends the data memory space with storage devices. Because the serialization of Java objects requires significant CPU resources, there is a tradeoff between memory space and application execution performance.

There is an "expansion Index" for de-serialized Java objects, which means that the required storage space for RDDs is bigger for de-serialized Java objects than for serialized Java objects. For the particular case of LineCounter, the expansion index is 2.09. We assigned 480GB of memory to Spark executors, thus providing Spark with 288GB of available unified memory. As a result, the experiments discussed in this section using Spark persistence on disk persists up to 250GB of data; however, only 100GB of data are persisted in experiments using Spark RDD persistence with de-serialized Java objects in memory. According to the results shown in Figure 1a, the execution time of LineCounter using RDD persistence with de-serialized Java objects is the shortest among the tested methods because Spark does not de-serialize the RDD datasets during the execution. The average task execution time for each task is about 20 ms; however, Figure 2 presents the normalized average task execution time.

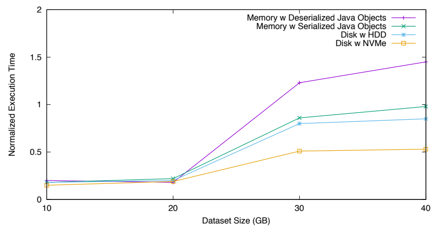
Figures 1b and 1c show the execution time of WordCount-reduceByKey and WordCount-groupByKey. Both of them have same result with same input data, but WordCount-groupByKey has much more shuffle data than WordCount-reduceByKey, which is showed in Table I.



(a) LineCounter



(b) WordCount-reduceByKey



(c) WordCount-groupByKey

Figure 1: Normalized execution time of different benchmarks using Spark RDD persistence and different storage technologies

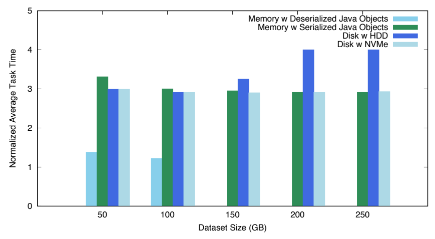


Figure 2: Normalized AVG LineCounter task execution time using Spark RDD persistence and different storage choices

As shown in Figure 1b, WordCount-reduceByKey with persisted deserialized Java Objects provides the best performance. However, as the data size increases, the cluster can not offer enough storage memory. This causes performance degradation with 50 GB and larger data sets. Furthermore, the experimental evaluation shows that NVMe provides high performance, which can match the performance of memory with serialized Java Objects.

Compared to WordCount-reduceByKey, WordCount-groupByKey shows quite low performance. This is not only because of the shuffle size. GroupByKey is a quite expensive task because it sends data to the assigned executors, which consumes most of the memory and network resources. Once the data set increases to 30GB, Spark evicts part of the data from memory. Thus, Spark needs to calculate the required data block again. As a result, it is easy for GroupByKey to trigger the delay scheduler. Table II shows the number of off-node blocks that are needed to be fetched for WordCount-groupByKey.

While the execution of LineCounter using Spark RDD persistence with serialized Java objects in memory is slower than with de-serialized Java objects, its execution with serialized Java objects on NVMe has performance similar to that using memory for persisting the serialized Java objects. Although these results might be counterintuitive, the de-serialization of Spark RDD uses most of the CPU resources in Spark tasks. In other words, the I/O throughput is not the bottleneck for persisting RDD datasets with serialized Java objects on NVMe (or memory).

The execution of workloads using Spark RDD persistence with serialized Java objects on hard disk drives is the slowest method, as expected. However, this is not necessarily only because hard disk drives are slower than memory and NVMe.

Data Size	WC-reduceByKey (GB)	WC-groupByKey (GB)
10 GB	0.76	2.4
20 GB	1.53	4.6
30 GB	2.3	6.7
40 GB	3.1	8.7

Table I: Shuffle r/w size of WordCount (WC)

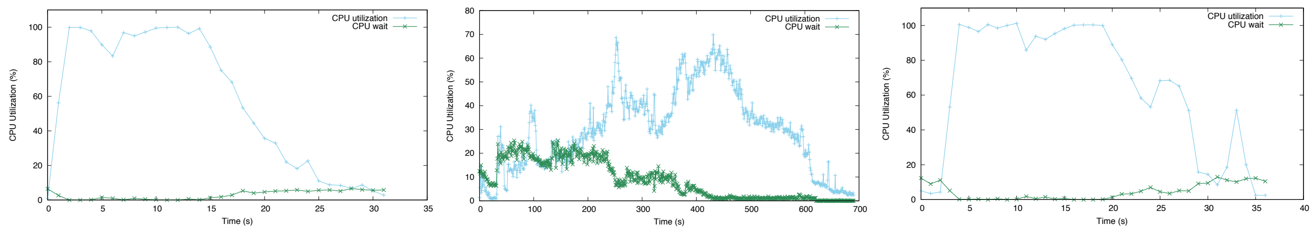
Data Size	Memory	Memory w Ser	HDD	NVMe
10 GB	0	0	0	0
20 GB	0	0.4	0.4	3
30 GB	21.2	1.4	3	3
40 GB	31	4.6	1.2	4.2

Table II: Off node data fetching of WordCount-groupByKey

As shown in Figure 2, the processing time for each task increases significantly as the dataset size increases from 50GB to 250GB (specifically from under 1 second to about 10 seconds). There are two main reasons for this workload slowdown: (1) as datasets become bigger and bigger, the operating system cannot offer sufficient memory buffer to accelerate the data fetching, and (2) as tasks require longer and longer processing time, it eventually exceeds the threshold of the delay scheduler, which means that it will fetch data from a remote node instead of a local one.

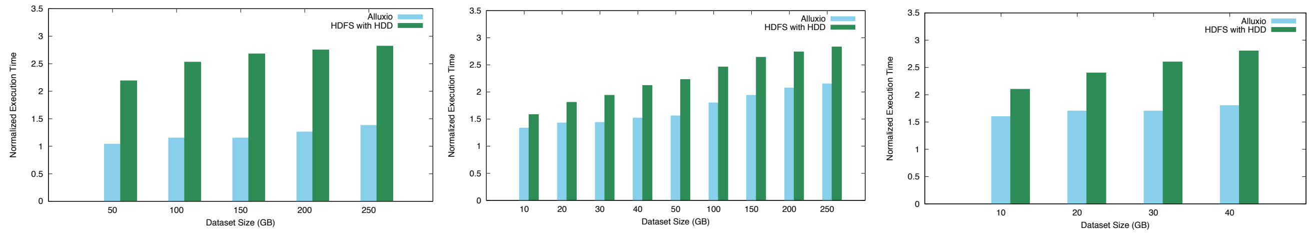
We choose LineCounter here to analyze different persistence technologies. Figures 3a and 3c show that the CPU utilization of the cluster is almost 100% at the beginning and middle stages of the workload execution. This is because Spark consumes most of the CPU resources in de-serializing the RDD datasets. Thus, the bottleneck of Spark persistence is not the I/O throughput with NVMe and memory (with serialized Java objects). This indicates that NVMe is a good candidate to replace traditional storage when the amount of memory capacity is constrained. Furthermore, Spark has almost the same performance with memory and NVMe combined with compression (columnar compression in Spark) while NVMe has a lower cost than memory.

Figure 3b shows that the workload CPU utilization ranges from 20% to 60% using Spark RDD persistence with a hard disk drive. This indicates that the I/O throughput is the bottleneck when using the hard disk, while the bottleneck is the processor when using NVMe or memory. We conclude that Spark RDD persistence using NVMe with compression as an intermediate data buffer can achieve almost the same performance as using memory and, therefore, using remote non-volatile memory has a large potential for supporting in-



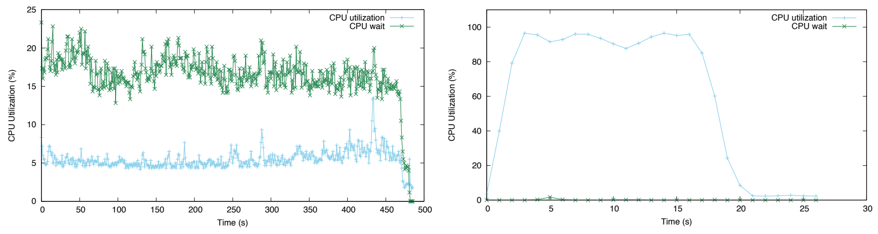
(a) Memory (Java serialized objects) (b) Hard disk (c) NVMe

Figure 3: CPU utilization using Spark RDD persistence with different storage technologies and 200 GB data sets



(a) LineCounter (b) WordCount-reduceByKey (c) WordCount-groupByKey

Figure 4: Execution time of different benchmarks using Alluxio and HDFS (hard disk-based)



(a) HDFS (b) Alluxio

Figure 5: CPU Utilization of LineCounter using HDFS and Alluxio with 200GB data sets

memory processing data persistency using NVMe over fabrics and/or current generation software-defined infrastructure.

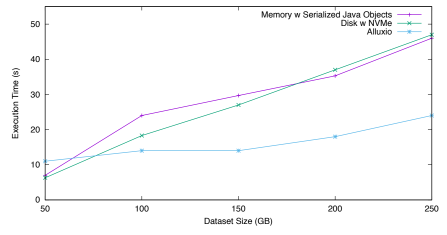
B. Comparative Study of HDFS and Alluxio

Compared to HDFS, the most important difference when using Alluxio is in the memory utilization approach. As shown in Figures 4a, 4b and 4c, Alluxio is much faster than HDFS for all dataset sizes. Because HDFS is based on a hard disk and Alluxio is based on an in-memory storage layer, these two distributed file systems have different bottlenecks. We also choose LineCounter to show the detail insights. Figures 5a and 5b show that the I/O throughput is the bottleneck for HDFS while the CPU performance (deserialization speed) is the bottleneck for Alluxio. However, although Alluxio is much faster than HDFS, it is not expected to replace HDFS in the short term because memory is still a very expensive resource in datacenters. Alluxio seems a very promising solution not only as an intermediate layer between HDFS and Spark but also as a means to provide mechanisms for data coupling in data-centric workflows as discussed below. Furthermore, Spark can pre-fetch required data into Alluxio, which can improve the overall performance of Spark applications. This is especially relevant if we expect low-latency access to remote non-volatile memory, which is a current trend in architecture.

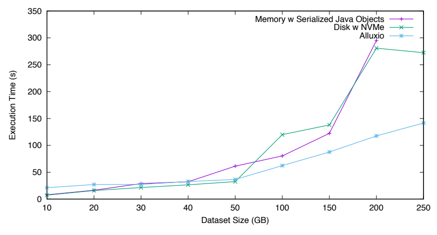
C. Spark RDD Persistence vs. Alluxio

In this section, we evaluate the performance of Alluxio with two different Spark persistence technologies. Because Alluxio stores the serialized dataset in its own system, we store Spark RDDs as serialized Java objects in Alluxio using memory and NVMe. As shown in Figures 6a, 6b and 6c, Spark persistence technology shows better performance than Alluxio using both memory and NVMe with small datasets. However, Alluxio shows better performance with large datasets compared to Spark persistence. This is because Alluxio has better load balancing and high-throughput optimization than Spark persistence technology. As a result, in order to improve the execution of Spark applications, Spark persistence is more appropriate for small datasets and Alluxio is more appropriate for large datasets.

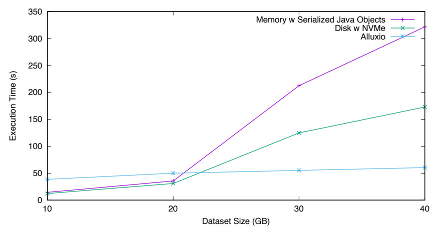
We also choose LineCounter here as an example to give detailed insights. As shown in Figure 7, the tasks' execution time (i.e., execution time of the different executors) using Spark persistence and Alluxio are almost the same. However,



(a) LineCounter



(b) WordCount-reduceByKey



(c) WordCount-groupByKey

Figure 6: Execution time of different benchmarks using on Spark RDD Persistence and Alluxio

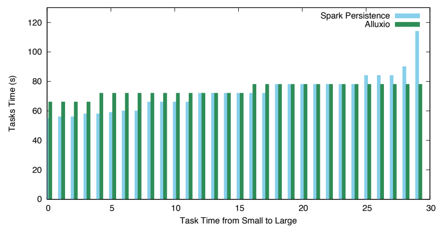


Figure 7: Task execution time of different executors using Spark Persistence in memory and Alluxio, 200 GB data sets

Alluxio is more stable than Spark persistence. Because Spark application execution time is dominated by the longest executor, Alluxio provides better overall performance than Spark persistence. This indicates that new models for data persistency might be needed for optimizing in-memory processing systems using next-generation software-defined infrastructure.

V. Simulation-Based Evaluation

In order to set a baseline configuration for our software-defined infrastructure model, we divide the Spark model into two parts: (1) a model for estimating Spark's application execution time, and (2) a model for estimating the datacenter interconnect transmission time. Our model also considers key aspects of current software-defined infrastructure (i.e., RSD). Specifically, in this architecture the storage node is connected to compute nodes using high-performance PCIe, which requires a new data locality strategy in Spark. Based on this architecture, we define data locality using two approaches, rack and off-rack. We assume that our Spark cluster is based on the computing nodes that are on the same rack. Because of the resource pool design of RSD, we assume that Spark can access remote memory and disk while there are insufficient resources in the local Spark cluster.

We build the Spark execution time model based on the experimental evaluation results discussed above, which provides a baseline performance characterization of current software-defined infrastructure[1]. Our model is built upon the performance prediction model by Wang et al. [15]. The execution time of a Spark stage is described as follows:

$$Time_{Stage} = Time_{start} + \max_{c=1}^{P} \sum_{i=1}^{K_c} Time_{Task} + Time_{cleanup}$$

where P is the total number of processor cores available in the cluster. In Spark, the executors fetch tasks from a task pool once an existing task is completed, and thus different processor cores deal with different numbers of tasks in one stage. K_c is the number of finished tasks for a given core c.

Based on the assumptions described above, a Spark cluster can fetch persisted RDD datasets from a remote storage device through a fast (i.e., low-latency and high-throughput) datacenter interconnect, which means that data can be fetched from remote memory or remote NVMe (e.g., via RDMA). In order to build the Spark execution time model targeting next-generation software-defined infrastructure, we must consider several parameters in our design choices, including network bandwidth, network latency, and the size of the RDD dataset.

Different models have been proposed in the existing literature to estimate the cost of data transmission over networks. For example, Thakur et al. [16] proposed the $\alpha + n\beta$ model to estimate the cost of one message sent between two nodes. In this model, α is the latency of each message, β is the cost of transferring one byte, and n is the number of bytes in the message. Thus, we can drive our model for estimating data transfer cost as follows:

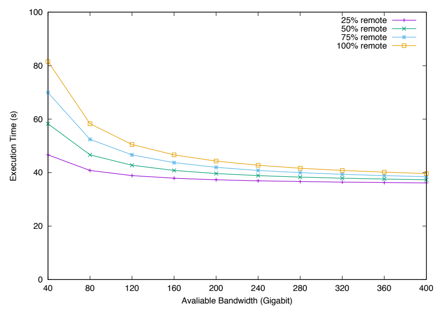
$$Time_{network} = \alpha + \frac{S_{Block} \times N_{tasks}}{B}$$

where α is the latency of each message, S_{Block} is the size of each block in Spark, and B is the available bandwidth between two nodes. Because Spark runs multiple tasks at the same time, N_{task} represents the number of tasks running at the same time.

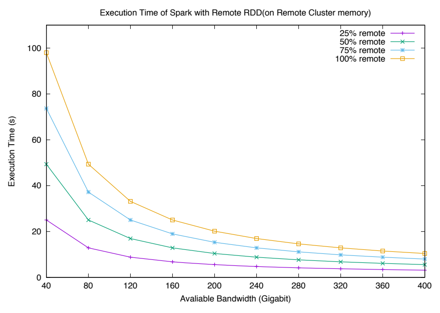
In the simulations, we assume that the Spark cluster does not have enough memory capacity to store the intermediate Spark RDD datasets. We also assume that remote resources in the same datacenter can provide additional memory and NVMe as storage for Spark persistence.

Because different Spark clusters share the datacenter's network bandwidth, we target the execution time of Spark applications for different network design choices in terms of bandwidth. We assume that next-generation software-defined infrastructure will be capable of delivering off-node bandwidth comparable to PCIe-based infrastructure but with a baseline of

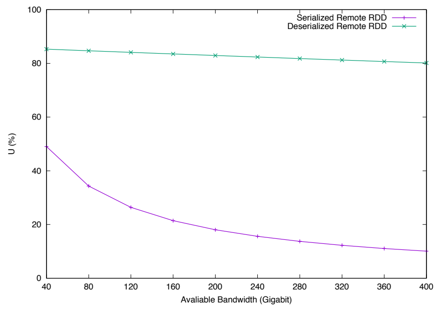
[1]For reproducibility, the experiments to build the model are provided in GitHub at https://github.com/HelloHorizon/SBAC-PAD-18



(a) Execution time, remote RDDs using NVMe



(b) Execution time, remote RDDs using RAM



(c) Network bandwidth utilization

Figure 8: Simulation results using RDDs in remote (i.e., off-node) resources

400Gbps, which vendors have advertised as being available in the market in the near future [3]. We consider the datasets from section IV, i.e., 200GB in serialized format. The input parameters for the simulations can be summarized as follows:

- 20-100% RDD data sets stored in remote resources
- 10-100% network bandwidth available for the cluster
- 400 Gbps network bandwidth with RDMA technology
- RDD data sets persisted in NVMe (serialized format)
- RDD data sets persisted in memory (de-serialized format)

As shown in Figure 8a, if a Spark cluster can monopolize the datacenter's bandwidth, then the execution time becomes shorter as available bandwidth increases. Spark applies the columnar compression to decrease I/O pressure on the storage devices. Because the network bandwidth is sufficiently provisioned in the simulated system, we simulate the Spark execution time with RDD datasets in de-serialized format. Figure 8a shows that Spark with de-serialized RDD datasets performs better than Spark with serialized RDD datasets when available bandwidth exceeds 280Gbps. Further, when the larger RDD datasets are persisted on remote resources, Spark with de-serialized RDD datasets has worse performance. In summary, Spark performance with de-serialized RDD datasets is better than its performance with serialized RDD datasets when there is sufficient network bandwidth available or when the dataset is small.

As shown in Figure 8c, Spark cannot fully utilize the network bandwidth using the targeted software-defined infrastructure. According to our model and Figure 8c, Spark with de-serialized RDDs can decrease the execution time by leveraging current high-bandwidth network fabrics; however, we conclude that the current standard implementation of Spark is not expected to fully exploit future interconnects (e.g., technologies based on silicon photonics).

VI. Related Work

Existing studies in the literature have aimed at improving the performance of MapReduce and in-memory big data processing frameworks, such as Spark, using different network technologies and optimization strategies. Sur et al. [17] evaluated the impact of high-speed interconnects for datacenters such as Infiniband and 10Gb Ethernet for supporting Hadoop distributed file systems (HDFS). This work also revealed that Infiniband and 10Gb Ethernet is more suitable for systems based on solid-state drives than spinning hard disks. Islam et al. [18] addressed the design of HDFS using remote direct memory access (RDMA) over Infiniband. They evaluated the performance of Gigabit Ethernet and IP-over-Infiniband on the QDR platform, showing that InfiniBand has much better performance than Gigabit Ethernet. Lu et al. [19], [20] proposed a high-performance RDMA approach based on accelerating the shuffle stage for Spark and evaluated the performance of RDMA with InfiniBand for different workloads in Spark. Kamburugamuve et al. [21] accelerated Apache Heron using InfiniBand and Intel Omni-path, which can increase the speed of network throughput for real-time big data frameworks. Gupta et al. [22] used Intel Omni-Path to accelerate big data frameworks such as Spark and Hadoop.

There are also proposals in the existing literature of optimizations for Spark using GPU and FPGA. Manzi et al. [23] explored the potential of GPU to accelerate different workloads (WordCount, K-Means and Sort) for the Spark framework. Li et al. [24] developed HereroSpark, which can utilize the GPU to increase the speed of machine learning algorithms for Spark. Rathore et al. [25] designed a GPU-based Spark system for processing real-time large-size city traffic video data. Gupta et al. [22] proposed a design using Xeon and FPGA to increase the speed of Spark and Hadoop.

Current research has also addressed algorithm optimization to improve the performance of Spark. Davidson et al. [26] used RDD compression to increase the speed of Spark, which increased the CPU utilization while reducing the I/O pressure. Chaimov et al. [27] developed a file pooling layer to improve metadata performance in Spark. The utilization of various storage technologies to improve the performance of big data frameworks has been previously evaluated. Our previous work [13] explored and evaluated performance, power, and tradeoffs of using different storage technologies such as solid state drive (SSD) and non-volatile memory for Spark and Hadoop with a focus on HDFS and extrapolating current tradeoffs to next-generation software-defined infrastructure. Kambatla et al. [28] and Moon et al. [29] explored the potential of using SSD in HDFS for accelerating Hadoop deployments. Moon et al. also studied how to accelerate Hadoop using multiple disks. To the best of our knowledge, this is the first work aimed at exploring key co-design aspects of in-memory big data frameworks for next-generation software-defined infrastructure.

VII. Conclusion and Future Work

In this paper, we analyzed the performance of Spark persistence technology using different storage technologies and compared native Spark persistence with the Alluxio distributed in-memory file system. We observed that NVMe with columnar compression is a good candidate for complementing memory in Spark clusters. We also concluded that Alluxio has better performance than Spark persistence technology for large datasets and found that Alluxio has better load balancing than Spark persistence. Results indicate that software-defined infrastructure can be a viable solution for provisioning bare-metal disaggregated datacenter resources and provide meaningful data points to illuminate the requirements of in-memory systems to efficiently scale next-generation software-defined infrastructure implementations (e.g., 400G MSA vs. 400/800G embedded optics vs. PCIe 5.0). While this work represents the foundation or a segment in the most ambitious path towards software-defined in-memory frameworks, the insights obtained from this work are critical for our undergoing work on a caching system for combining Spark and Alluxio more efficiently. Our future work includes developing memory emulation mechanisms to evaluate full-stack in-memory frameworks with different software-defined infrastructure design choices. As power requirements to run workloads in-memory may have different resource utilization patterns using off-rack resources, our current work also includes studying power-related issues.

Acknowledgments

This work is supported in part by National Science Foundation via grants numbers ACI-1464317 and CNS-1305375, and was conducted as part of the Rutgers Discovery Informatics Institute (RDI^2).

References

[1] C. Lynch, "Big data: How do your data grow?" Nature, 2008.

[2] H. Li, A. Ghodsi, M. Zaharia, S. Shenker, and I. Stoica, "Tachyon: Reliable, memory speed storage for cluster computing frameworks," in Proceedings of the ACM Symposium on Cloud Computing. ACM, 2014, pp. 1–15.

[3] L. Liao, "Intel silicon photonics: from research to product," IEEE Components, Packaging and Manufacturing, 2017.

[4] S. Han, N. Egi, A. Panda, S. Ratnasamy, G. Shi, and S. Shenker, "Network support for resource disaggregation in next-generation datacenters," in Proceedings of the Twelfth ACM Workshop on Hot Topics in Networks. ACM, 2013, p. 10.

[5] J. Lee, Y. Turner, M. Lee, L. Popa, S. Banerjee, J.-M. Kang, and P. Sharma, "Application-driven bandwidth guarantees in datacenters," in ACM SIGCOMM Computer Communication Review, vol. 44, no. 4. ACM, 2014, pp. 467–478.

[6] P. X. Gao, A. Narayan, S. Karandikar, J. Carreira, S. Han, R. Agarwal, S. Ratnasamy, and S. Shenker, "Network requirements for resource disaggregation." in OSDI, vol. 16, 2016, pp. 249–264.

[7] S. Legtchenko, H. Williams, K. Razavi, A. Donnelly, R. Black, A. Douglas, N. Cheriere, D. Fryer, K. Mast, A. D. Brown et al., "Understanding rack-scale disaggregated storage," in Proceedings of the 9th USENIX Conference on Hot Topics in Storage and File Systems. USENIX Association, 2017, pp. 2–2.

[8] A. Klimovic, C. Kozyrakis, E. Thereska, B. John, and S. Kumar, "Flash storage disaggregation," in Proceedings of the Eleventh European Conference on Computer Systems. ACM, 2016, p. 29.

[9] K. Lim, J. Chang, T. Mudge, P. Ranganathan, S. K. Reinhardt, and T. F. Wenisch, "Disaggregated memory for expansion and sharing in blade servers," in ACM SIGARCH Computer Architecture News, vol. 37, no. 3. ACM, 2009, pp. 267–278.

[10] K. Lim, Y. Turner, J. R. Santos, A. AuYoung, J. Chang, P. Ranganathan, and T. F. Wenisch, "System-level implications of disaggregated memory," in High Performance Computer Architecture (HPCA), 2012 IEEE 18th International Symposium on. IEEE, 2012, pp. 1–12.

[11] J. Gu, Y. Lee, Y. Zhang, M. Chowdhury, and K. G. Shin, "Efficient memory disaggregation with infiniswap." in NSDI, 2017, pp. 649–667.

[12] Z. Guz, H. H. Li, A. Shayesteh, and V. Balakrishnan, "Nvme-over-fabrics performance characterization and the path to low-overhead flash disaggregation," in Proceedings of the 10th ACM International Systems and Storage Conference. ACM, 2017, p. 16.

[13] S. Chen and I. Rodero, "Understanding behavior trends of big data frameworks in ongoing software-defined cyber-infrastructure," in Proceedings of the Fourth IEEE/ACM International Conference on Big Data Computing, Applications and Technologies. ACM, 2017, pp. 199–208.

[14] Spark, "Rdd persistence," https://spark.apache.org/docs/2.0.0/programming-guide.html#rdd-persistence, 2017.

[15] K. Wang and M. M. H. Khan, "Performance prediction for apache spark platform," in Proceedings of HPCC-CSS-ICESS'15. IEEE, 2015, pp. 166–173.

[16] R. Thakur, R. Rabenseifner, and W. Gropp, "Optimization of collective communication operations in mpich," The International Journal of High Performance Computing Applications, vol. 19, no. 1, pp. 49–66, 2005.

[17] S. Sur, H. Wang, J. Huang, X. Ouyang, and D. K. Panda, "Can high-performance interconnects benefit hadoop distributed file system," in Workshop on Micro Architectural Support for Virtualization, Data Center Computing, and Clouds (MASVDC). Held in Conjunction with MICRO. Citeseer, 2010.

[18] N. S. Islam, M. W. Rahman, J. Jose, R. Rajachandrasekar, H. Wang, H. Subramoni, C. Murthy, and D. K. Panda, "High performance rdma-based design of hdfs over infiniband," in Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis. IEEE Computer Society Press, 2012, p. 35.

[19] X. Lu, M. W. U. Rahman, N. Islam, D. Shankar, and D. K. Panda, "Accelerating spark with rdma for big data processing: Early experiences," in High-performance interconnects (HOTI), 2014 IEEE 22nd annual symposium on. IEEE, 2014, pp. 9–16.

[20] X. Lu, D. Shankar, S. Gugnani, and D. K. D. 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. IEEE, 2016, pp. 253–262.

[21] S. Kamburugamuve, K. Ramasamy, M. Swany, and G. Fox, "Low latency stream processing: Apache heron with infiniband &#38; intel omni-path," in Proceedings of the10th International Conference on Utility and Cloud Computing, 2017, pp. 101–110.

[22] P. Gupta, "Accelerating datacenter workloads," in 26th International Conference on Field Programmable Logic and Applications (FPL), 2016.

[23] D. Manzi and D. Tompkins, "Exploring gpu acceleration of apache spark," in Cloud Engineering (IC2E), 2016 IEEE International Conference on. IEEE, 2016, pp. 222–223.

[24] P. Li, Y. Luo, N. Zhang, and Y. Cao, "Heterospark: A heterogeneous cpu/gpu spark platform for machine learning algorithms," in Networking, Architecture and Storage (NAS), 2015 IEEE International Conference on. IEEE, 2015, pp. 347–348.

[25] M. M. Rathore, H. Son, A. Ahmad, A. Paul, and G. Jeon, "Real-time big data stream processing using gpu with spark over hadoop ecosystem," International Journal of Parallel Programming, pp. 1–17, 2017.

[26] A. Davidson and A. Or, "Optimizing shuffle performance in spark," University of California, Berkeley-Department of Electrical Engineering and Computer Sciences, Tech. Rep, 2013.

[27] N. Chaimov, A. Malony, S. Canon, C. Iancu, K. Z. Ibrahim, and J. Srinivasan, "Scaling spark on hpc systems," in Proceedings of the 25th ACM International Symposium on High-Performance Parallel and Distributed Computing. ACM, 2016, pp. 97–110.

[28] K. Kambatla and Y. Chen, "The truth about mapreduce performance on ssds," in 28th Large Installation System Administration Conference (LISA14), 2014, pp. 118–126.

[29] S. Moon, J. Lee, and Y. S. Kee, "Introducing ssds to the hadoop mapreduce framework," in Cloud Computing (CLOUD), 2014 IEEE 7th International Conference on. IEEE, 2014, pp. 272–279.