High Performance Design for HDFS with Byte-Addressability of NVM and RDMA *

Nusrat Sharmin Islam, Md. Wasi-ur-Rahman, Xiaoyi Lu, and Dhabaleswar K. (DK) Panda
Department of Computer Science and Engineering
The Ohio State University
{islamn, rahmanmd, luxi, panda}@cse.ohio-state.edu

ABSTRACT

Non-Volatile Memory (NVM) offers byte-addressability with DRAM like performance along with persistence. Thus, NVMs provide the opportunity to build high-throughput storage systems for data-intensive applications. HDFS (Hadoop Distributed File System) is the primary storage engine for MapReduce, Spark, and HBase. Even though HDFS was initially designed for commodity hardware, it is increasingly being used on HPC (High Performance Computing) clusters. The outstanding performance requirements of HPC systems make the I/O bottlenecks of HDFS a critical issue to rethink its storage architecture over NVMs. In this paper, we present a novel design for HDFS to leverage the byte-addressability of NVM for RDMA (Remote Direct Memory Access)-based communication. We analyze the performance potential of using NVM for HDFS and re-design HDFS I/O with memory semantics to exploit the byte-addressability fully. We call this design NVFS (NVM- and RDMA-aware HDFS). We also present cost-effective acceleration techniques for HBase and Spark to utilize the NVM-based design of HDFS by storing only the HBase Write Ahead Logs and Spark job outputs to NVM, respectively. We also propose enhancements to use the NVFS design as a burst buffer for running Spark jobs on top of parallel file systems like Lustre. Performance evaluations show that our design can improve the write and read throughputs of HDFS by up to 4x and 2x, respectively. The execution times of data generation benchmarks are reduced by up to 45%. The proposed design also reduces the overall execution time of the SWIM workload by up to 18% over HDFS with a maximum benefit of 37% for job-38. For Spark TeraSort, our proposed scheme yields a performance gain of up to 11%. The performances of HBase *insert*, *update*, and *read* operations are improved by 21%, 16%, and 26%, respectively. Our NVM-based burst buffer can improve the I/O performance of Spark PageRank by up to 24% over Lustre. To the best of our knowledge, this paper is the first attempt to incorporate NVM with RDMA for HDFS.

*This research is supported in part by National Science Foundation grants #CNS-1419123, #IIS-1447804, and #ACI-1450440.

ICS '16, June 01-03, 2016, Istanbul, Turkey

DOI: http://dx.doi.org/10.1145/2925426.2926290

1. INTRODUCTION

Wide adoption of Big Data Analytics in the current financial services has transformed the way of operation in the leading industries. According to IDC [6] financial insights report [24] in 2015, financial institutions spent more than 25% of IT budgets on just three contemporary and transformative technologies - mobility, cloud, and Big Data Analytics. The report also forecasts that this rate will rise to 30% by the year of 2019. The vast growth of data with the technological complexity in processing and storage is the major reason behind this trend. According to [2], 2.5 exabytes of data were generated each day throughout the year 2014. This proliferation of data necessitates the usage of the most advanced storage technologies in the next-generation Big Data middleware.

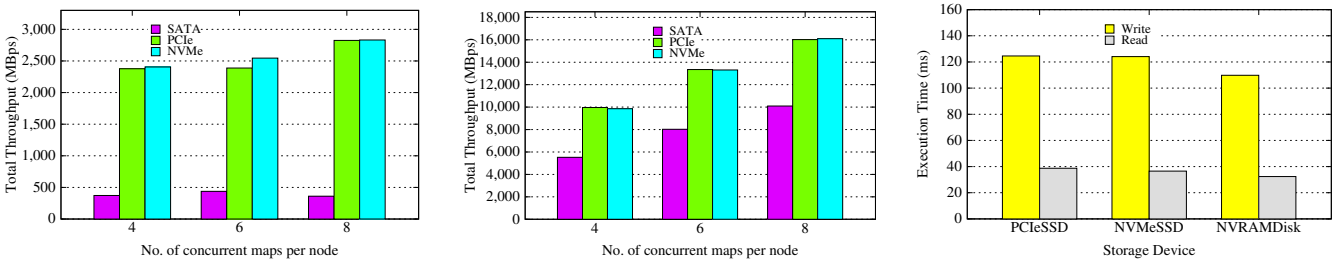
For Big Data Analytics, Apache Hadoop [37] has become the de-facto distributed processing platform. The current eco-system of Hadoop contains the legacy batch processing framework like MapReduce [37], as well as the in-memory based DAG execution framework like Spark [41] for iterative and interactive processing. Hadoop Distributed File System (HDFS), which is well-known for its scalability and reliability, provides the storage capabilities to these processing frameworks. HDFS is also used as the underlying file system of Hadoop database, HBase [1, 13]. HDFS, along with these upper-level middleware, is being increasingly used on HPC platforms for scientific applications. HPC clusters are usually equipped with high performance interconnects and protocols, such as RDMA. Non-volatile memory (NVM) technology is also emerging and making its way into the HPC systems. Since the initial design considerations for HDFS were focusing on the technologies mostly used in the commodity clusters, extracting performance benefits through the usage of advanced storage technologies, such as NVMs along with RDMA, requires reassessment of those design choices for different file system operations.

1.1 Motivation

HDFS stores the data files in local storage on the DataNodes. For performance-sensitive applications, in-memory storage is being increasingly used for HDFS on HPC systems [3, 20]. Even though HPC clusters are equipped with large memory per compute node, using this memory for storage can lead to degraded computation performance due to

Table 1: Existing Studies on NVM and File Systems

	HDFS	Replication over RDMA	Enhanced Read	Block access	Byte-addressability	Usage policies
[42]		√		√	√	
[15, 16, 40]			√	√	√	
[19, 21]	√	√				
This paper	√	√	√	√	√	√



(a) TestDFSIO Write (b) TestDFSIO Read (c) I/O Time for an HDFS Block

Figure 1: Performance Characteristics of NVM and SSD

competition for physical memory between computation and I/O. The huge memory requirements by applications from diverse domains, such as, deep-learning, neuroscience, astrophysics make the memory contention a critical issue to rethink the deployment of in-memory file systems on HPC platforms. Moreover, many researches [12, 36] have studied the impact of caching to accelerate HDFS Read performance. But as indicated in a recent trace from Cloudera [14], 34% of jobs have their outputs as large as the inputs. The performances of such write-intensive jobs are bound by the data placement efficiency of the underlying file system. Recent studies [3, 20, 27] propose in-memory data placement policies to increase the write throughput. But in-memory data placement makes the task of persistence challenging. NVMs, being non-volatile, promise new opportunities for persistence and fault tolerance. NVMs can not only augment the overall memory capacity, but also provide persistence while bringing in significant performance improvement.

As shown in Table 1, even though researches like [15, 16, 40] utilize byte-addressability of NVM, they concentrate on unreplicated single machine systems. Mojim [42] focuses on RDMA-based replication with NVM but does not consider read performance in the presence of high-level of parallelism or cost-effective NVM usage policies that are critical for Big Data applications. Thus, none of these studies fully address the needs of Big Data file systems like HDFS.

Moreover, the use of NVMs is becoming popular in the NVMe SSDs. Figure 1(a) and Figure 1(b) show the performance comparisons of NVMe and traditional PCIe SSD with those of SATA SSD. As observed from the figures, both NVMe and PCIe SSD significantly improve the performances over SATA SSD. On the other hand, even though NVMe standardizes the interface of SSDs, there is little performance increase over traditional PCIe ones. Figure 1(c) illustrates the I/O times for a single HDFS block across different storage devices. The figure clearly shows that, the I/O time reduces by 12% when the block is stored to RAMDisk backed by NVM (which we call NVRAMDisk). But even this is not the most optimal use of NVM as this cannot fully exploit the byte-addressability and obtain peak performance.

All these lead us to the following broad challenges:

1. What are the alternative techniques to use NVM for HDFS?
2. Can RDMA-based communication in HDFS leverage NVM on HPC systems?
3. How can we re-design HDFS to fully exploit the byte-addressability of NVM to improve the I/O performance?
4. Can we propose advanced, cost-effective accelerating techniques for Spark and HBase to utilize NVM in HDFS? Can the NVM-based HDFS also be used as a burst buffer layer for running Spark jobs over parallel file systems like Lustre?

1.2 Contributions

To answer these questions, this paper proposes a novel design of HDFS, which we call **NVFS** (NVM- and RDMA-aware HDFS), to leverage the byte-addressability of NVM. Our design minimizes the memory contention of computation and I/O by allocating memory from NVM for RDMA-based communication. We analyze the performance potential of using NVM for HDFS I/O and re-design the HDFS storage architecture to exploit the memory semantics of NVM for I/O operations (HDFS *Write* and *Read*). In this study, we also propose advanced, cost-effective techniques for accelerating Spark and HBase by intelligently utilizing NVM in the underlying file system for Spark job outputs and HBase Write Ahead Logs (WALs) only, respectively. We also present enhancements to use the NVM-based HDFS as a burst buffer for running Spark jobs on top of parallel file systems like Lustre.

Due to the limited availability of the NVM hardware, we simulate the proposed designs using DRAM while conservatively adding extra latency to mimic the behavior of NVM as done in [18, 42].

In-depth performance evaluations show that, our design can increase the read and write throughputs of HDFS by up to 2x and 4x, respectively over PCIe SSD, 2x and 3.5x over

NVMe SSD, 1.6x and 4x over SATA SSD across different HPC clusters. It also achieves up to 45% benefit in terms of execution time for data generation benchmarks. Our design reduces the overall execution time of the SWIM workload by 18% while providing a maximum gain of 37% for job-38. The proposed design also reduces the execution time of Spark TeraSort by 11% over PCIe SSD by storing only the job output in NVM. The performances of HBase *insert*, *update*, and *read* operations are improved by 21%, 16%, and 26%, respectively over SATA SSD by writing only the HBase WALs to NVM, while all other data are stored to SATA SSD. Our burst buffer design reduces the I/O time of Spark PageRank by up to 24% over Lustre. To the best of our knowledge, this paper is the first attempt to incorporate NVM with RDMA for HDFS.

The following describes the organization of the paper. Some background knowledge is discussed in Section 2. In Section 3 we discuss our design scope. Section 4 describes our designs. In Section 5, we present our evaluations. In Section 6, we discuss the related work. We conclude the paper in Section 7 with future works.

2. BACKGROUND

In this section, we provide some background information for the technologies used in this paper.

2.1 Hadoop Distributed File System

Hadoop Distributed File System (HDFS) is used as the primary storage for a Hadoop cluster [4]. An HDFS cluster consists of two types of nodes: NameNode and DataNode. The NameNode manages the file system namespace. It maintains the file system tree and stores all the meta data. The DataNodes store the HDFS files. HDFS divides large files into fixed-sized blocks. Each block is stored independently in a separate file in the the DataNode. HDFS guarantees fault tolerance by replicating each block to multiple DataNodes [34]. The default replication factor is three.

2.2 RDMA-based HDFS Design

In [21], the authors propose an RDMA-enhanced HDFS design that uses RDMA for HDFS write and replication via the JNI interface. The JNI layer bridges Java-based HDFS with communication library written in native C.

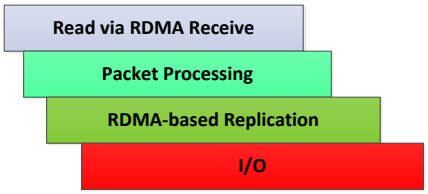


Figure 2: Overlapping among different stages of HDFS write [19]

In [19], the authors re-design the HDFS architecture to maximize overlapping among different stages of HDFS write operation with a SEDA-based approach [39]. Figure 2 illustrates the overlapped execution in the proposed design. In the default HDFS architecture, the different stages (data read, packet processing, replication, and I/O of the data packets) of HDFS write happen sequentially and for each incoming data block, a new thread is created (one thread per block). With, RDMA-based communication, data arrive at the DataNodes at a much faster speed. Therefore, it is inefficient to wait for the I/O of the previous packet before going to receive the next packet. In [19], the read, processing, replication, and I/O for HDFS write can overlap with RDMA-based communication as well as one another. Thus, the authors achieve overlapping in packet-level within a block in addition to the block-level overlapping in the existing architecture. This is achieved by assigning the task of each of the stages to a dedicated thread pool. Each thread pool is in charge of processing events on the corresponding queue. By performing admission control on each event queue, the whole system with multiple thread pools can maximally overlap various stages of HDFS write.

2.3 Non-volatile Memory (NVM)

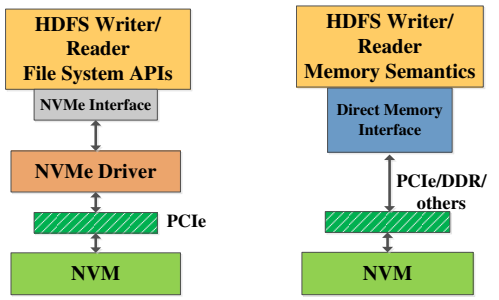
Non-volatile Memory (NVM) or NVRAM [30] provides high data reliability and byte-addressability. Therefore, they are closing the performance, capacity and cost-per-bit gaps between volatile main memory (DRAM) and persistent storage. NVMs are, thus, excellent contenders to co-exist with RAM and SSDs in large scale clusters and server deployments. The performance of NVM is orders of magnitude faster than SSD and disk. This implies that it is the software overheads that cause performance bottlenecks when using NVM to replace the disk-based storage systems.

2.4 SSD

Storage technology has been rapidly growing over the recent years. Due to low power consumption and high random access bandwidth, SSD or flash devices have paved their way into HPC clusters. SATA SSDs are being widely used due to their low cost and high performance as compared to disk-based hard drives (HDDs). But the SATA-based interface limits the capacity of the bus that transfers data from the SSDs to the processor. The current generation of PCIe flash SSDs have thus become very popular in modern HPC clusters due to their high bandwidth and fast random access. While they provide excellent performance, the adoption of PCIe SSDs is inhibited by different implementations and unique drivers. NVM Express standard (NVMe) [8] enables faster adoption and interoperability of PCIe SSDs. The benefits of NVMe with PCIe over traditional SATA SSD drives are reduced latency, increased IOPS and lower power consumption, in addition to being durable.

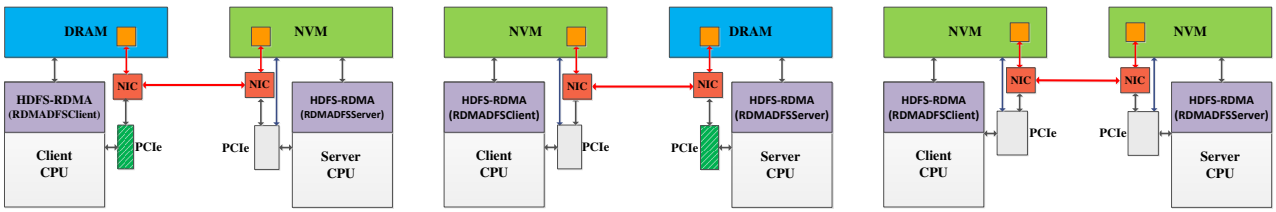
3. DESIGN SCOPE

In this section, we discuss our design scope.



(a) Block Access (b) Memory Access

Figure 3: NVM for HDFS I/O



(a) D-to-N over RDMA (b) N-to-D over RDMA (c) N-to-N over RDMA

Figure 4: NVM for HDFS over RDMA

3.1 NVM as HDFS Storage

In this section, we discuss different design alternatives to incorporate NVMs for HDFS storage. Typically, NVM can be accessed in either of two modes:

Block Access: Figure 3(a) shows how NVMs can be used in block access mode for HDFS. NVM can be mounted as an HDFS data directory in the DataNode side. After the data is received, it can be written to the corresponding HDFS block file via HDFS I/O APIs by the writer threads. Similarly, the readers can read data blocks using the HDFS file *Read* APIs. These I/O operations go through the NVMe interface in fixed blocks, just the way flash or disk storages are accessed. RAMDisk can be backed by NVM instead of DRAM. HDFS can access NVM in block mode (via RAMDisk) in such case.

Memory Access (NVRAM): Figure 3(b) depicts the way to use NVM in memory access mode for HDFS. The NVM card usually supports a direct memory interface with memory mapping and addressing. In this mode, the writer/reader threads on HDFS DataNodes can use memory semantics like direct load and store to access the data in NVRAM. But this requires the HDFS storage engine to be re-designed using memory semantics instead of file system APIs.

3.2 NVM for RDMA

In this section, we discuss different design alternatives to incorporate NVM in HDFS over RDMA. In RDMA-based HDFS designs [19, 21], the memory for RDMA communication between *RDMADFSClient* and *RDMADFSServer* is allocated from DRAM. The communication library provides APIs to register memory from DRAM in both the client and server side. Communication also occurs among the DataNodes for HDFS replication. Since these data transfers happen in a non-blocking fashion, a large amount of memory (allocated from DRAM) is needed for communication. Moreover, many concurrent clients and *RDMADFSServer* can co-exist in a single machine, which can lead to huge memory-contention. Thus, RDMA communication takes memory from the DRAM which otherwise could be used by heavy-weight applications.

The presence of NVMs in the HDFS DataNodes (we assume that each HDFS DataNode has an NVM) offers the opportunity to utilize them for RDMA communication. This requires the NVM to be used in memory access mode (NVRAM). Figure 4 depicts the three possible alternatives to use NVRAM for RDMA.

DRAM to NVRAM (D-to-N) over RDMA: In this approach, the communication buffers in *RDMADFSClient* are allocated from DRAM; the server side registers memory from NVRAM. After the *RDMADFSServer* receives data, the data is replicated to the next DataNode in the pipeline. In this case, the first DataNode acts as a client and allocates memory from DRAM for the sender side buffers. The receiver side (*RDMADFSServer*) receives data in buffers allocated in NVRAM.

NVRAM to DRAM (N-to-D) over RDMA: In this approach, the communication buffers in *RDMADFSClient* as well as the buffers for replication are allocated in NVRAM; the server side registers memory for RDMA communication from DRAM.

NVRAM to NVRAM (N-to-N) over RDMA: This approach allocates memory from NVRAM in both *RDMADFSClient* and *RDMADFSServer*. RDMA-based replication also uses buffers allocated in NVRAM.

The (N-to-N) over RDMA approach leads to more penalty compared to traditional RDMA as well as the other two approaches using NVRAM. The performance characteristics of D-to-N and N-to-D are similar. Moreover, D-to-N does not need NVMs to be present in the client side. Since we want to use more of NVM space for storage in the DataNode (server) side and each *RDMADFSServer* has to serve multiple concurrent *RDMADFSClient*s, we follow the **D-to-N over RDMA** approach in our design.

3.3 Performance Gap Analysis

In order to make the opportunities for performance gain with NVM concrete, we evaluated HDFS block I/O times over different memory and storage configurations. For this, we designed a benchmark (in Java) that mimics the HDFS file write/read APIs. This benchmark was run locally on a single node to measure the time required to write/read one HDFS block (128MB) over different storage (or memory) configurations. The block is written in 512KB chunks (HDFS packet size). The NVM is accessed in memory mode (NVRAM).

For performance characterization over DRAM, the chunks are written to (read from) a *hash map*. Here, we assumed that NVRAM is 10x slower than DRAM in write performance and has similar read performance [9, 30]. Therefore, while calculating the HDFS block I/O times over NVRAM, we followed the same approach as in DRAM and added extra latency for simulation. On the other hand, for writing (reading) to (from) RAMDisk, PCIe SSD, or NVMe SSD, we used HDFS file I/O APIs. We also measured the time to write (read) to (from) NVRAMDisk using the file system APIs, but added extra latency to mimic the NVRAM behavior.

As observed from Figure 5, NVRAM is 20.8x faster than RAMDisk, 24.9x faster than PCIe SSD, and 24.8x faster

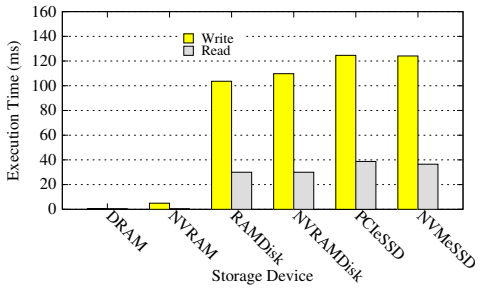


Figure 5: Performance gaps among different storage devices

than NVMe SSD for HDFS write. For HDFS read, NVRAM is 83.4x faster than RAMDisk, 107.67x faster than PCIe SSD, and 101.53x faster than NVMe SSD. On the other hand, NVRAMDisk is 1.13x faster than both PCIe and NVMe SSD for HDFS write; for HDFS read, it is 1.29x and 1.21x faster than PCIe and NVMe SSD, respectively.

We also calculate the delay (extra latency) to be added per block for HDFS I/O for NVRAM (compared to DRAM). The block write time for DRAM is 0.499ms; for NVRAM, it is 4.99ms (NVRAM is 10x slower than DRAM). Therefore, the extra latency to be added for NVRAM during HDFS write is (4.99 - 0.499) = 4.49ms per block. No extra latency is added for HDFS block read as NVRAM has similar read performance as DRAM. We followed this approach because, to the best of our knowledge, none of the available NVM simulators can work with HDFS so far.

4. PROPOSED DESIGN AND IMPLEMENTATION

4.1 Design Overview

In this section, we present an overview of our proposed design, NVFS. Figure 6 presents its architecture.

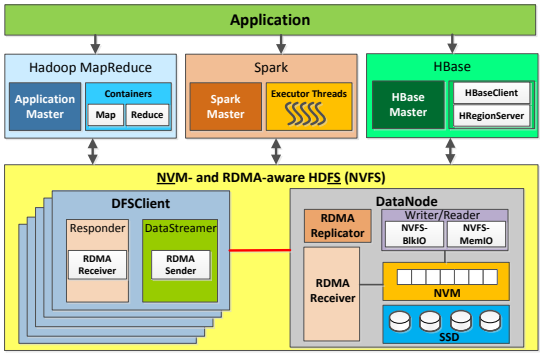


Figure 6: Architecture of NVFS

NVFS is the high performance HDFS design using NVM and RDMA. For RDMA-based communication between DFSClient and DataNode and replication among the DataNodes, we use the **D-to-N over RDMA** approach as discussed in Section 3.2. The *RDMASender* in the DFSClient sends data over RDMA and uses DRAM for buffer allocation. On the other hand, the *RDMAReceiver* in the DataNode receives the data in buffers allocated in NVRAM. The data is replicated by the *RDMAReplicator* over RDMA. After the data is received by the *RDMAReceiver*, it is given to the HDFS Writer which can access NVM either in block access mode (NVFS-BlkIO) using the HDFS file system APIs or in memory access mode (NVFS-MemIO). In this paper, we present designs for HDFS I/O using both of these modes.

We also present advanced and cost-effective acceleration techniques for Spark and HBase by using NVM (with SSD through a hybrid design) for performance-critical data only. For this, we co-design HDFS with HBase and Spark so that these middleware can utilize the NVM in a cost-effective manner. We further propose enhancements to use the NVM-based HDFS design as a burst buffer for running Spark jobs over parallel file systems, such as Lustre.

4.2 Design Details

In this section, we discuss our proposed designs in detail. We propose designs for utilizing NVM both in block and memory access modes for HDFS while considering workload characteristics (read-/write-intensive).

4.2.1 Block Access (NVFS-BlkIO)

In this section, we propose to use NVM in block access mode for HDFS storage on the DataNodes. For RDMA communication, the buffers are allocated from NVRAM in the HDFS DataNode side when the DataNodes start up. These buffers are used in a round-robin manner for receiving data by the *RDMADFSServer*. In order to preserve the HDFS block I/O structure, the data received in the NVRAM buffers via RDMA, are then encapsulated into JAVA I/O streams and written to the block files created in NVM using the HDFS file system APIs. Figure 7 describes this design.

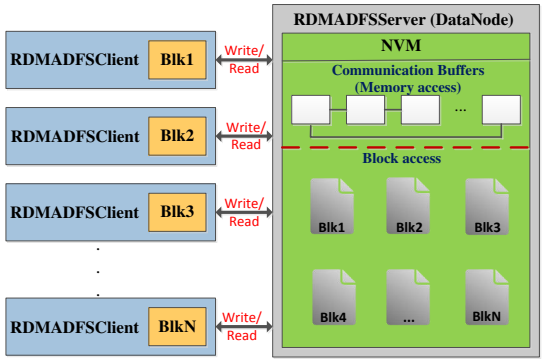


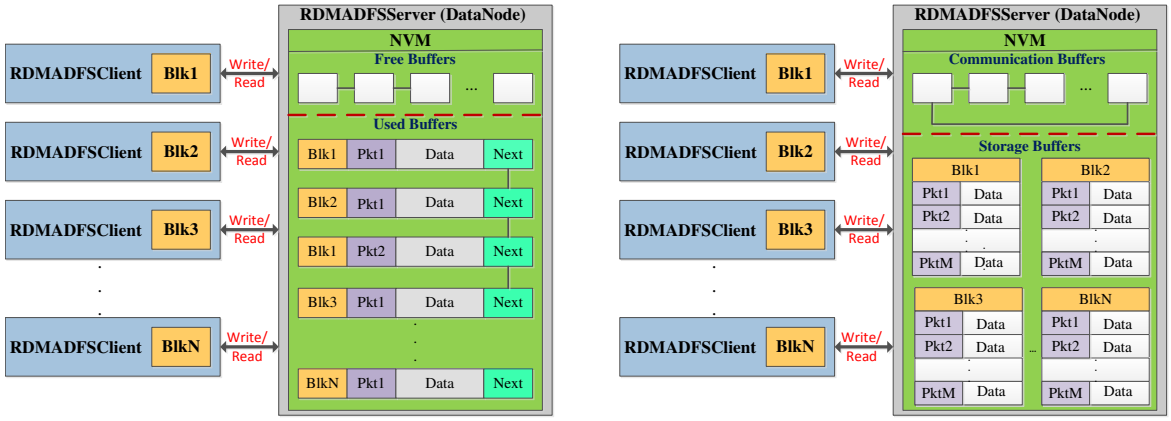
Figure 7: Design of NVFS-BlkIO

In order to accelerate the I/O, we propose to *mmap* the block files during HDFS write over RDMA. In this way, the NVM pages corresponding to the block files are mapped into the address space of *RDMADFSServer*, rather than paging them in and out of the kernel buffer-cache. *RDMADFSServer* can then perform direct memory operations via ByteBuffer put/get to the *mmap*ed region and syncs the block file when the last packet of the block arrives.

4.2.2 Memory Access (NVFS-MemIO)

In this section, we present our design to use NVM in memory access mode (NVRAM) for HDFS. We consider two different architectures for HDFS to incorporate NVRAM.

Flat Architecture: Figure 8(a) shows the *Flat Architecture*. In this design, we allocate a pool of buffers in the NVM



(a) Flat Architecture (b) Block Architecture

Figure 8: Design of NVFS-MemIO

on the HDFS DataNode. The buffers are allocated in a single linked list. We maintain two pointers: `head` and `tail` for the buffer pool. The `head` pointer always points to the first used buffer. The first free buffer is pointed to by the `tail` pointer. Initially, all the buffers in the buffer pool are free for use and `head` and `tail` point to the same buffer. As the *RDMADFSServer* receives data, the `tail` pointer keeps on moving forward to point to the first free buffer. The major ideas of this design are:

1. *Shared buffers for communication and storage*: The buffers allocated in NVM are used for both RDMA-based communication and HDFS I/O. The *RDMADFSServer* receives the data packets sent by the *RDMADFSClient* or upstream DataNodes. Each packet is identified by the corresponding `block id` and `packet sequence number` in the packet header. These packets, after being received, remain in those buffers and the buffers are marked as `used`. In this way, whenever a new packet arrives, the first free buffer pointed to by the `tail` pointer is returned and this buffer acts as the RDMA receive buffer as well as the storage buffer for that packet. As shown in Figure 8(a), in this architecture, a packet from $Block_2$ can reside in the buffer next to a packet from $Block_1$ when multiple clients send data concurrently. Therefore, this architecture does not preserve HDFS block structures and hence, is called the *Flat Architecture*.

2. *Overlap between Reader and Writer*: Even though the `tail` pointer keeps on changing when data comes to the *RDMADFSServer*, in the presence of concurrent readers and writers, the *Flat Architecture* does not need any synchronization to access the buffer pool. Because, read request for a block comes only when it is entirely written to HDFS. Therefore, scanning through the pool of buffers can continue without any synchronization from `head` to *tail* as it was at the time of the arrival of the read request. Therefore, this architecture can guarantee good overlap between HDFS write and read.

3. *Scan through the buffer pool during read*: Due to the *Flat Architecture*, when a read request for a block arrives, the reader thread in the DataNode side gets the first used buffer in the linked list pointed to by the *head* pointer and has to scan through the entire pool of buffers (up to the *tail* pointer in the worst case) to retrieve the packets belonging to that block in sequence. For large buffer pool, this scan is very inefficient which results in poor read performance.

This design has fast write performance, as the communication buffer for a piece of data also acts as the storage buffer. During HDFS write, the first free buffer can be retrieved in $O(1)$ time as it is pointed to by the *tail* pointer. So, the time complexity to write a block with M packets is $O(M)$. But in order to achieve this, all the storage (communication also) buffers have to be registered against the RDMA connection object, and, therefore, depends on the underlying communication library. Moreover, this architecture needs to scan the entire pool of buffers to retrieve the required data packets. If the number of blocks received by the *RDMADFSServer* is N when the read request arrives, and each block contains at most M packets, then, the time to read an entire block is $O(M * N)$. This makes read performance very slow for large data sizes (large N). Considering all these, we propose another architecture, which we call *Block Architecture* that offers modularity between storage and communication library with enhanced read performance.

Block Architecture: Figure 8(b) shows the *Block Architecture*. In this design, we allocate two different buffer pools in the NVM: one for RDMA communication, another for storage. Each of the buffer pools is pointed to by a `head` pointer that returns the first free buffer. The reason to separate out these two buffer pools is to eliminate the need to register all the buffers against the RDMA connection object. In this way, storage operation does not depend on the communication library and offers better modularity than the *Flat Architecture*. The main ideas behind this design are:

1. *Re-usable communication buffers*: The communication buffers are used in a round-robin manner. The *RDMADFSServer* receives the data in a free communication buffer returned by the `head` pointer. After this, data is written to one of the free storage buffers. The communication buffer is then freed and can be re-used for subsequent RDMA data transfers.

2. *HDFS Block Structure through the storage buffers*: Figure 9 shows the internal data structures for this design. The DataNode has a `hash table` that hashes on the `block id`s. Each entry in the `hash table` points to a linked list of buffers. This linked list contains the data packets belonging to the corresponding block only and data packets belonging to the same block are inserted sequentially in the linked list. In this way, this design preserves the HDFS block structures. This is why, we name this architecture *Block Architecture*.

3. *Overlapping between communication and storage*: The encapsulation of data from the communication to the storage buffers happens in separate threads. The receiver thread in *RDMADFSServer*, after receiving a data packet, returns the buffer id to a worker thread. The worker thread then performs the transfer of data to a storage buffer; while the receiver thread can continue to receive subsequent data over RDMA. In this way, the *Block Architecture* ensures good overlap between communication and I/O.

4. *Constant time retrieval of blocks during read*: As the DataNode maintains a `hash table` of block ids, the list of data packets can be retrieved in constant time (amortized cost) for each block during read. The subsequent data packets for a block are also obtained it constant time as they are in adjacent entries of the linked list.

5. *Synchronization*: Since separate buffer pools are used for communication and storage, no synchronization is needed between the communication and storage phases. The readers read from the linked lists corresponding to their respective blocks and the writers write to the free storage buffers. So, no synchronization is needed for concurrent readers and writers. Concurrent writers need to synchronize the pool of storage buffers to retrieve the first free buffer.

In this architecture, the amortized time complexity to read a block is $O(1) + O(M)$, where M is the number of packets per block. This time is independent of the total number of blocks received by the DataNode and therefore, the read performance is much faster than that of the *Flat Architecture*. On the other hand, the amortized time complexity to write a block is $M * (O(1) + O(W))$. The first free buffer for RDMA is retrieved in constant time and W is the number of concurrent writers competing for a free storage buffer pointed to by the `head` pointer. Moreover, $W << N$ and receiving the data is overlapped with I/O. Thus, the write overhead is negligible compared to the *Flat Architecture*. Therefore, the overall performance of *Block Architecture* is better than *Flat*.

In our design, we do not make any changes to HDFS client side APIs. Even though data is stored to NVM using memory semantics by the DataNode, this happens in a transparent manner and upper layer frameworks (HDFS clients) can still use HDFS file system APIs to invoke the I/O operations.

4.3 Implementation

In this section, we discuss our implementation details.

NVFS-BlkIO: We assume that the RAMDisk is backed by NVRAM and simulate HDFS I/O using RAMDisk as a data directory while adding additional latency for the write operations. As obtained in Section 3.3, the extra latency added per block is 4.49ms (0ms) during HDFS write (read). Similarly, for RDMA communication for HDFS write and replication, we add extra latency of 4.49ms per block.

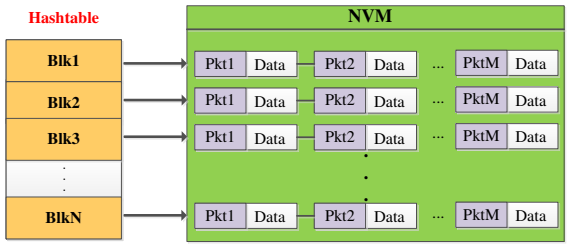


Figure 9: Internal data structures for Block Architecture

NVFS-MemIO: The *Block Architecture* offers better read performance and modularity over the *Flat Architecture*. Therefore, we implement this design in this work. We consider two types of memory allocation here.

1. Memory Allocation from JVM (NVFS-MemIO-JVM): Allocates memory from JVM.

2. Off-Heap Memory Allocation (NVFS-MemIO-JNI): Allocates off-heap memory through Java nio direct buffers using JNI.

In both the cases, some buffers are pre-allocated to avoid the overhead of memory allocation. The rest of the buffers are dynamically allocated on demand. We overlap the dynamic buffer allocation with communication so that this cost does not come into the critical path of the application.

For both NVFS-BlkIO and NVFS-MemIO, the NVM behavior is obtained by the following equations for each block:

$$t_{comm}^{D-N} = t_{comm}^{D-D} + delay_{comm}$$

Here, t_{comm}^{D-N} represents communication time for D-to-N over RDMA, t_{comm}^{D-D} is communication time for RDMA over DRAM (traditional RDMA), and $delay_{comm} = 4.49ms$ represents the added overhead.

$$t_{io}^{NVM} = t_{io}^{DRAM} + delay_{io}$$

Similarly, t_{io}^{NVM} represents I/O time for NVM, t_{io}^{DRAM} is I/O time for DRAM, and $delay_{io} = 4.49ms$ represents the added overhead for write. For read, $delay_{io} = 0$.

4.4 Proposed Cost-effective Performance Schemes for using NVM in HDFS

NVMs are expensive. Therefore, for data-intensive applications, it is not feasible to store all the data in NVM. We propose to use NVM with SSD as a hybrid storage for HDFS I/O. In our design, NVM can replace or co-exist with SSD through a configuration parameter. As a result, cost-effective, NVM-aware placement policies are needed to identify the appropriate data to go to NVMs. The idea behind this is to take advantage of the high IOPS of NVMs for performance-critical data; all others can go to SSD. Moreover, the performance-critical data can be different for dif-

ferent applications and frameworks. Consequently, it is challenging to determine the performance-critical data for different upper-level middleware over HDFS. In this section, we present optimization techniques to utilize NVMs in a cost-effective manner for Spark and HBase.

4.4.1 Spark

Spark jobs often run on pre-existing data that are generated previously and stored in disk. For such jobs, we propose to write only the job output in NVM. To do this, HDFS client identifies the output directory of a job provided by the application. The client then appends a flag bit to indicate buffering with the header of each block to be written to the output directory. In other words, all the headers of all the blocks belonging to the job output is updated with the buffering info. The DataNode parses the block header and writes the corresponding data packets to NVM. Periodically, these data are moved from NVM to SSD to make free buffers in the NVM. By default, Spark aggressively uses memory to store different types of data. We present an orthogonal design to write the job outputs to NVM in HDFS, so that Spark can use the available memory for other purposes like computation, intermediate data, etc.

4.4.2 HBase

With large amount of data insertions/updates, HBase eventually flushes its in-memory data to HDFS. HBase holds the in-memory modifications in MemStore which triggers a flush operation when it reaches its threshold. During the flush, HBase writes the MemStore to HDFS as an HFile instance. High operational latency for HDFS write can affect the overall performance of HBase *Put* operations [17, 19]. HBase also maintains HLog instances that are the basis of HBase data replication and failure recovery, and thus, must be stored in HDFS. HBase periodically flushes its log data to HDFS. Each HBase operation has a corresponding log entry and the logs need to be persistent. Therefore, in this paper, we propose to store only the Write Ahead Logs (WALs) in NVM. All other HBase files along with the data tables are stored in SSD. When a file is created in HDFS from HBase, the file path is distinguished based on its type. For example, a data file is always written under the directory */hbase/data*. On the other hand, WAL files are written in */hbase/WALs*. While a block is sent from the DFSClient to a DataNode, the client appends the file path (the file the block belongs to) to the block header. In the DataNode side, the block is stored to NVM according to Algorithm 1.

Algorithm 1 Algorithm to Store HBase WALs in NVM

Input: Block b

```
b_h ⇐ getHeader(b)
filePath ⇐ parse(b_h)
if /hbase/WALs ∈ filePath then
    Write b to NVM
else
    Write b to SSD
end if
```

4.5 NVFS-based Burst Buffer for Spark over Lustre

In order to handle the bandwidth limitation of shared file system access in HPC clusters, burst buffer systems [29] are often used; such systems buffer the bursty I/O and gradually flush datasets to the back-end parallel file system. The most commonly used form of a burst buffer in current HPC systems is dedicated burst buffer nodes. These nodes can exist in the I/O forwarding layer or as a distinct subset of the compute nodes or even as entirely different nodes close to the processing resources (i.e., extra resources if available). A burst buffer can also be located inside the main memory of a compute node or as a fast non-volatile storage device placed in the compute node (i.e., NVRAM devices, SSD devices, PCM devices, etc.). BurstFS [38] is an SSD-based distributed file system to be used as burst buffer for scientific applications. It buffers the application checkpoints to the local SSDs placed in the compute nodes, while asynchronously flushing them to the parallel file system.

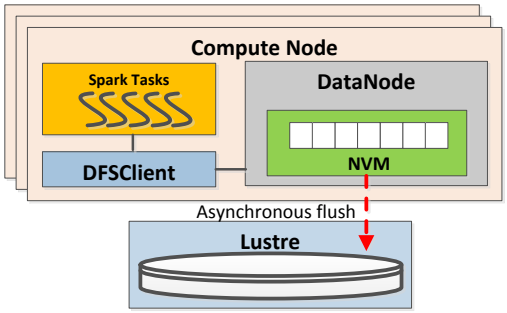
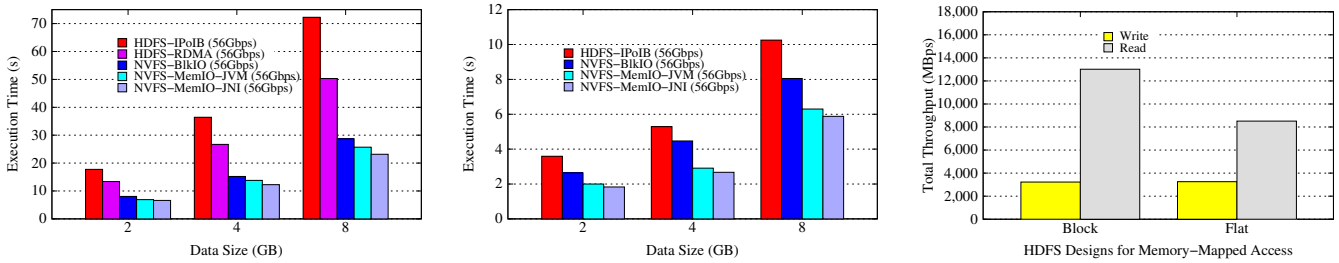


Figure 10: Design of NVFS-based Burst Buffer for Spark over Lustre

NVFS can, therefore, be used as the burst buffer for running Spark jobs over parallel file system Lustre. For this, we propose to deploy NVFS cluster with replication factor of one. The tasks from Spark jobs are co-located with the NVFS DataNodes in the compute partition of an HPC cluster. As a result, these tasks send their data to the DataNodes. These data are buffered in NVM and the DataNodes asynchronously flush the data to Lustre. To equip the NVFS DataNodes with the burst buffer functionalities, we add a pool of threads in the DataNode side to take care of the asynchronous flush to the parallel file system. This helps improve the I/O performance of Spark jobs by reducing the bottlenecks of shared file system access. Figure 10 shows the proposed design.

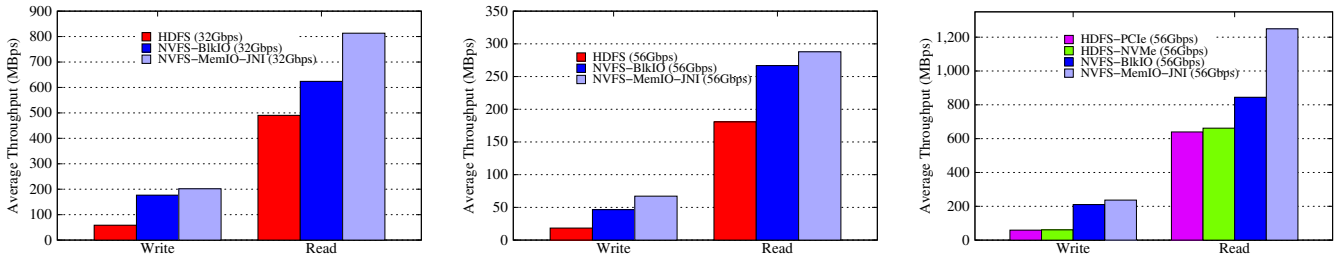
5. PERFORMANCE EVALUATION

In this section, we present a detailed performance evaluation of NVFS design in comparison to that of the default HDFS architecture. In this study, we perform the following sets of experiments: (1) Performance Analysis, (2) Evaluation with MapReduce, (3) Evaluation with Spark, (4) Evaluation with HBase. In all our experiments, we use RDMA for Apache Hadoop 2.x 0.9.7 [5] released by The Ohio State University and JDK 1.7.0. We also use OHB Microbenchmark 0.8 [5], YCSB-0.6.0, HBase 1.1.2, and Spark-1.5.1 for our evaluations. First we analyze our designs NVFS-BlkIO and NVFS-MemIO using the HDFS Microbenchmarks from OHB and then demonstrate the effectiveness of both of these designs using different MapReduce, Spark, and HBase work-



(a) SWL (b) SRL (c) Flat vs Block Architecture

Figure 11: HDFS Microbenchmark



(a) Cluster A (b) Cluster B (c) Cluster C

Figure 12: TestDFSIO

loads.

5.1 Experimental Setup

We use three different clusters for our evaluations.

(1) Intel Westmere Cluster (Cluster A): This cluster has five nodes. Each node has Xeon Dual quad-core processor operating at 2.67 GHz. Each node is equipped with 24 GB RAM, two 1TB HDDs, single 300GB OCZ VeloDrive PCIe SSD, and 12 GB RAMDisk with Red Hat Enterprise Linux Server release 6.1. Nodes on this cluster have MT26428 QDR ConnectX HCAs (32 Gbps data rate) and are interconnected with a Mellanox QDR switch.

(2) SDSC Comet (Cluster B): Each compute node in this cluster has two twelve-core Intel Xeon E5-2680 v3 (Haswell) processors, 128GB DDR4 DRAM, and 320GB of local SATA-SSD with CentOS operating system. The network topology in this cluster is 56Gbps FDR InfiniBand with rack-level full bisection bandwidth and 4:1 over-subscription cross-rack bandwidth. Comet also has seven petabytes of Lustre installation.

(3) Intel Sandy Bridge Cluster (Cluster C): We choose four nodes with Intel P3700 NVMe-SSD from this cluster. Each of them has dual eight-core 2.6 GHz Intel Xeon E5-2670 (Sandy Bridge) processors, 32GB DRAM, 16GB RAMDisk and is equipped with Mellanox 56Gbps FDR InfiniBand HCAs with PCI Express Gen3 interfaces. The nodes run Red Hat Enterprise Linux Server release 6.5. Each of these nodes also have a 300GB OCZ VeloDrive PCIe SSD and INTEL SSDSC2CW12 SATA SSD.

We allocate 1.5GB NVM space for the communication buffers that are used in a round-robin manner for non-blocking RDMA transfers. For storage buffers, we allocate 6GB NVM space on Cluster A and C; on Cluster B, it is 10GB.

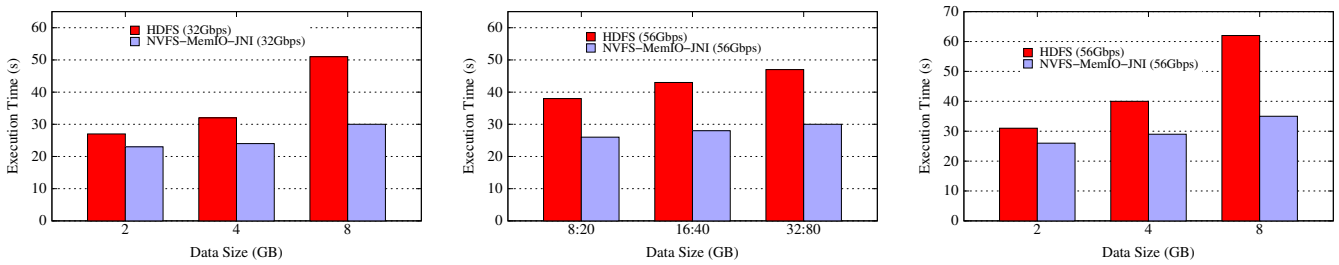
5.2 Performance Analysis

In this section, we analyze the performances of our proposed designs using HDFS Microbenchmarks from OHB Microbenchmark 0.8. These experiments are performed on three DataNodes on Cluster C. NVMe SSD is used as HDFS data directory on each of the DataNodes.

Performance comparison using HDFS Microbenchmarks: We evaluate our designs using SWL and SRL from OHB Microbenchmarks. As observed from Figure 11(a), NVFS-MemIO-JNI reduces the execution time by up to 67% over HDFS-IPoIB (56Gbps) and 53.95% over HDFS-RDMA (56Gbps) for 8GB data size. The performance of NVFS-MemIO-JNI is better by up to 20% over NVFS-BlkIO. As observed from Figure 11(b), NVFS-MemIO-JNI reduces the execution time of SRL by up to 43% over HDFS-IPoIB (56Gbps) and 27% over NVFS-BlkIO. NVFS-MemIO-JNI also shows better performance than NVFS-MemIO-JVM. Therefore, unless otherwise stated, we use NVFS-MemIO-JNI for our subsequent experiments.

Figure 11(c) shows the performance comparisons between our two proposed architectures for memory access mode of NVM. In this experiment, we use SWT and SRT for 8GB data size with four concurrent clients per node. The figure clearly shows that, the read performance for the *Block Architecture* is 1.5x of the *Flat Architecture*. Both the designs offer similar performances for write. This explains the benefit of the *Block Architecture* over *Flat*. In the presence of concurrent readers, read in the *Block Architecture* is faster, as each block and the corresponding data packets can be retrieved in constant time. Due to the overlap between communication and storage, the write performances are similar in both the architectures.

Performance comparison over RAMDisk: We evaluate NVFS-BlkIO and NVFS-MemIO-JNI and compare the



(a) Cluster A (b) Cluster B (c) Cluster C

Figure 13: Data generation benchmark, TeraGen

performances with HDFS running over RAMDisk using the SWL benchmark for 4GB data size. Table 2 shows the results for this experiment.

Table 2: SWL performance over RAMDisk

IPoIB	RDMA	NVFS-BlkIO	NVFS-MemIO-JNI
22.72 sec	18.3 sec	15.21 sec	12.26 sec

The table clearly shows that, NVFS-MemIO-JNI reduces the execution time by 46%, 33%, and 20% over HDFS-IPoIB, HDFS-RDMA, and NVFS-BlkIO, respectively. NVFS-BlkIO also gains by 33% and 17% over HDFS-IPoIB and HDFS-RDMA, respectively. Moreover, RAMDisk cannot ensure tolerance against process failures; which our designs can, due to the non-volatility of NVM. NVFS-BlkIO gains due to memory speed write using *mmap*, while NVFS-MemIO gains due to elimination of software overheads of the file system APIs.

Comparison of communication time: We compare the communication times of HDFS over RDMA using DRAM vs. NVRAM. For this, we design a benchmark that mimics the HDFS communication pattern. The benchmark transfers data from the client to a server which replicates it in a pipelined manner (no processing or I/O of the received data).

Table 3: Comparison of Communication Time

IPoIB	RDMA	RDMA with NVRAM
10.73 sec	5.10 sec	5.68 sec

As shown in Table 3, the communication time for a 4GB file is 5.10s over RDMA using DRAM. On the other hand, when NVRAM is used for RDMA, this time increases to 5.68s due to the overheads of NVRAM. Compared to the communication time over IPoIB (56Gbps), the proposed **D-to-N over RDMA** approach has 47% benefit.

Comparison of I/O time: In this study, we compare the I/O times needed by our designs with that of NVMe SSD using the SWL benchmark for 4GB data size.

Table 4: Comparison of I/O Time

NVMe SSD	NVFS-BlkIO	NVFS-MemIO-JNI
4.27 sec	3.14 sec	1.83 sec

The I/O times are profiled on the first DataNode in the replication pipeline. First, we measure the I/O time per block and report the cumulative I/O time for the entire file. As observed from Table 4, NVFS-MemIO-JNI is 42% and 57% faster than NVFS-BlkIO and HDFS with NVMe SSD, respectively. NVFS-BlkIO is also 27% faster than HDFS with NVMe SSD in terms of I/O time. Thus, our designs guarantee much better I/O performance than default HDFS with NVMe SSD as they leverage the byte-addressability of NVM.

5.3 Evaluation with Hadoop MapReduce

In this section, we evaluate MapReduce benchmarks.

5.3.1 TestDFSIO

Figure 12 shows the results of our evaluations with the TestDFSIO benchmark on three different clusters. For these experiments, we use both NVFS-BlkIO and NVFS-MemIO-JNI. On Cluster A, we run this test with 8GB data size on four DataNodes. As observed from Figure 12(a), NVFS-BlkIO increases the write throughput by 3.3x over PCIe SSD. The gain for NVFS-MemIO-JNI is 4x over HDFS using PCIe SSD over IPoIB (32Gbps) for 8GB data size. For TestDFSIO read, our design NVFS-MemIO-JNI has up to 1.7x benefit over PCIe SSD on Cluster A. The benefit for NVFS-BlkIO is up to 1.4x. As shown in Figure 12(b), on Cluster B with 32 DataNodes for 80GB data, our benefit is 2.5x for NVFS-BlkIO and 4x for NVFS-MemIO-JNI over SATA SSD, for TestDFSIO write. Our gain for read is 1.2x. Figure 12(c) shows the results of TestDFSIO experiment for 8GB data on three DataNodes on Cluster C, where we have 4x benefit over NVMe SSD for TestDFSIO write. Our gain for read is 2x.

5.3.2 Data Generation Benchmark (TeraGen)

Figure 13 shows our evaluations with the data generation benchmark TeraGen [11]. We use our design NVFS-MemIO-JNI here. Figure 13(a) shows the results of experiments that are run on four HDFS DataNodes on Cluster A. As observed from the figure, our design has 45% gain over HDFS using PCIe SSD with IPoIB (32Gbps). On Cluster B, we vary the data size from 20GB on eight nodes to 80GB on 32 nodes. As observed from the Figure 13(b), our design reduces the execution time of TeraGen by up to 37% over HDFS using SATA SSD with IPoIB (56Gbps). As shown in Figure 13(c), on Cluster C with three DataNodes, our design has a gain of up to 44% over HDFS using NVMe SDD with IPoIB (56Gbps). TeraGen is an I/O-intensive benchmark. The enhanced I/O-level designs using NVM and communication with the **D-to-N over RDMA** approach, results in

these performance gains across different clusters with different interconnects and storage configurations.

5.3.3 SWIM

In this section, we evaluate our design (NVFS-BlkIO) using the workloads provided in Statistical Workload Injector for MapReduce (SWIM) [10]. The MapReduce workloads in SWIM are generated by sampling traces of MapReduce jobs from Facebook. In our experiment, we generate one such workload from historic Hadoop job traces from Facebook. We run this experiment on eight DataNodes on Cluster B and generate 20GB data. The generated workload consists of 50 representative MapReduce jobs. As observed from Figure 14, 42 jobs are short duration with execution times less than 30s. For such jobs, NVFS-BlkIO has a maximum benefit of 18.5% for job-42. Seven jobs have execution times greater than (or equal to) 30s and less than 100s. For such jobs, our design has a maximum gain of 34% for job-39. NVFS-BlkIO gains a maximum benefit of 37% for job-38; for this job, the execution time of HDFS is 478s, whereas for NVFS-BlkIO, it is 300s. The overall gain for our design is 18%.

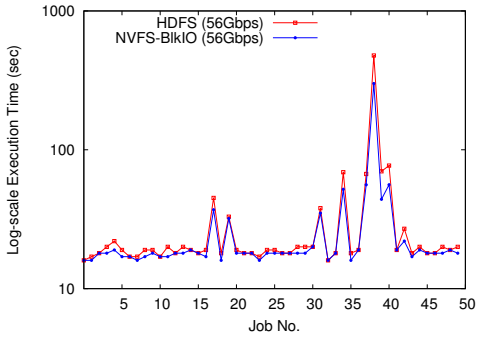


Figure 14: SWIM (Cluster B)

Table 5 summarizes the results for SWIM. The MapReduce jobs in SWIM read (write) data from (to) HDFS. Therefore our design, with enhanced I/O and communication performances gain over HDFS with SATA SSD over IPoIB (56Gbps). Job-38, reads and writes large amount of data from (to) HDFS ($hdfs\_bytes\_read$ = 1073750104, $hdfs\_bytes\_written$ = 17683184696) compared to the others. Therefore, the gain is also higher for this job.

Table 5: Summary of benefits for SWIM

Job Duration	Execution Time (sec)	Number of Jobs	Benefit
Short	<30	42	18.5%
Medium	$\geq$ 30 and < 100	7	34%
Long	$\geq$ 100	1	37%

5.4 Evaluation with Spark

5.4.1 TeraSort

In this section, we evaluate our design (NVFS-MemIO-JNI) along with the cost-effective scheme using the Spark TeraSort workload. We run these experiments on four DataNodes on Cluster A using the Spark over YARN mode. The data size is varied from 10GB to 14GB. The values of Spark executor and worker memory are set to 2GB each.

As observed from Figure 15(a), by buffering only the output of TeraSort to NVM, our design can reduce the execution time by up to 11% over HDFS using PCIe SSD with IPoIB (32Gbps). The input data as well as other control information are stored to PCIe SSD. The reason behind this gain is the reduction of I/O and communication overheads while writing the TeraSort outputs to HDFS. Besides, TeraSort is shuffle-intensive and involves computation to sort the data. In our design, we use DRAM to mimic NVRAM behavior which could otherwise be used for computation as in HDFS, if real hardware was available.

5.4.2 PageRank

In this section, we evaluate our NVFS-based burst buffer design using the Spark PageRank workload. We perform these experiments on eight DataNodes on Cluster B and compare the I/O times of PageRank when run over NVFS-based burst buffer vs. Lustre. For this, we use our design NVFS-MemIO-JNI and PageRank is launched with 64 concurrent maps and reduces. Figure 15(b) illustrates the results of our evaluations.

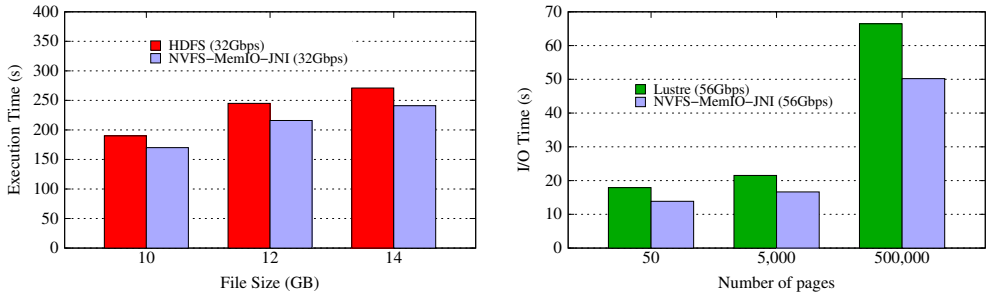
As observed from the figure, our burst buffer design can reduce the I/O time (time needed to write the output of different iterations to the file system) of Spark PageRank by up to 24% over Lustre. This is because, when Spark tasks write data to NVFS-based burst buffer, most of the writes go to the same node where the tasks are launched. As a result, unlike Spark running over Lustre, the tasks do not have to write data over the network to the shared file system, rather, data is asynchronously flushed to Lustre. Besides, studies have already shown that, in terms of data access peak bandwidth, SSD is much faster than Lustre [20, 22, 38]. NVM being faster than SSD, has much higher performance than Lustre. Therefore, our proposed burst buffer design can improve I/O performance of Spark jobs over Lustre.

5.5 Evaluation with HBase

In this section, we evaluate our design (NVFS-BlkIO) with HBase using the YCSB benchmark. We perform these experiments on Cluster B. We vary the cluster size from 8 to 32 and the number of records are varied from 800K on 8 nodes to 3200K on 32 nodes.

Figure 16 shows the performance of our design for the 100% *insert* workload. As observed from the figure, our design reduces the average insert latency by up to 19% over HDFS using SATA SSD with IPoIB (56Gbps); the increase in operation throughput is 21%. Figure 17 shows the performance of our design for the 50% *read*, 50% *update* workload. As observed from the figure, our design reduces the average read latency by up to 26% over HDFS using SATA SSD; the average update latency is improved by up to 16%; the increase in operation throughput is 20%. The proposed design flushes the WAL files to NVM during the course of these operations. The WALs are also replicated using the **D-to-N over RDMA** approach. The reduction of I/O and communication times through the proposed cost-effective solution helps improve the performance of HBase *insert*, *update*, and *read*.

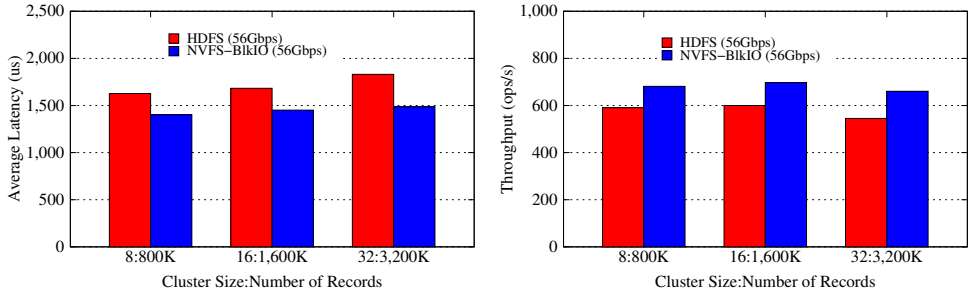
6. RELATED WORK



(a) Spark TeraSort with cost-effective scheme (Cluster A)

(b) Spark PageRank with NVFS-based burst buffer (Cluster B)

Figure 15: Evaluation with Spark workloads



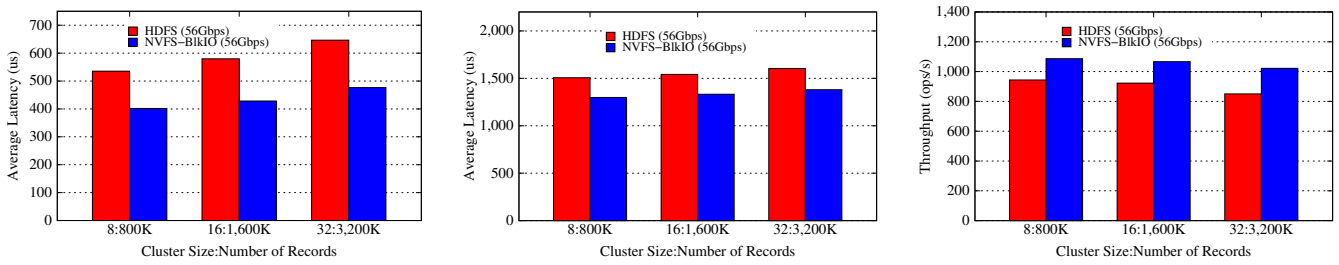
(a) Average Latency

(b) Throughput

Figure 16: YCSB 100% insert (Cluster B)

There have been many researches to accelerate I/O performance through NVM. The authors in [15] present a file system BPFS and a hardware architecture that are designed around the properties of persistent, byte-addressable memory. In order to transmit atomic, fine-grained updates to persistent storage, BPFS proposes a new technique called short-circuit shadow paging that helps BPFS provide strong reliability guarantees and better performance than traditional file systems. Pelley et al. [30] propose to leverage NVRAM as main memory to host all of the data sets for in-memory databases. The authors in [16] propose a new, light-weight POSIX-compliant file system, PMFS, that exploits the byte-addressability of NVRAM to avoid overheads of block I/O. PMFS also leverages hardware features like processor paging and memory ordering. Fusion-io's NVMFS, which was formerly known as DFS [23] uses the virtualized flash storage layer for simplicity and performance. DFS ensures a much shorter datapath to the flash memory by leveraging the virtual to physical block allocation in the virtualized flash layer. SCMFS [40] is a file system for storage class memory and uses the virtual address space for implementation. In [31], the authors propose a hybrid file system to resolve the random write issues of SSDs by leveraging NVRAM. In [33], the authors perform systematic performance study of various legacy Linux file systems with real world workloads over NVM. In this work, the authors demonstrate that most of the POSIX-compliant file systems can be tuned to achieve comparable performance as PMFS. All these NVM-based file system studies concentrate on workloads over unreplicated single machine. In Mojim [42], the authors propose RDMA and NVM-based replication for storage subsystems in data centers. But this work focuses only on the write and replication part, whereas, a lot of Big Data applications involve significant amount of reads. None of the above-mentioned studies consider the special needs of a Big Data file system like HDFS. Besides, HDFS has to support a variety of upper layer frameworks and middleware. Therefore, it is very important to maintain backward compatibility for HDFS even when exploiting the byte-addressability of NVMs. Moreover, the high expense of NVMs make cost-effectiveness a big issue to consider for Big Data applications. In this study, we consider all these aspects while designing HDFS over NVM and RDMA.

Both Big Data and HPC community have put significant efforts to improve the performance of HDFS by leveraging resources from HPC platforms. In [21], the authors present RDMA-enhanced HDFS to improve the performance of write and replication. But these designs kept the default HDFS architecture intact. In [19], a SEDA (Staged Event-Driven Architecture) [39] based approach has been proposed to re-design HDFS architecture. Authors in [12] have proposed in-memory data caching to maximize HDFS read throughputs. The Apache Hadoop community [36] has also proposed such kind of centralized cache management scheme in HDFS, which is an explicit caching mechanism that allows users to specify paths to be cached by HDFS. Some recent studies [20, 25, 26, 35] also pay attention to incorporate heterogeneous storage media (e.g. SSD and parallel file system) in HDFS. Authors in [25] deal with data distribution in the presence of nodes that do not have uniform storage characteristics; whereas, [26] caches data in SSD. Researchers in [35] present HDFS-specific optimizations for PVFS and [32] propose to store cold data of Hadoop cluster to network-attached file systems. In [20], the authors



(a) Average Read Latency (b) Average Update Latency (c) Throughput

Figure 17: YCSB 50% read, 50% update (Cluster B)

propose advanced data placement policies for HDFS to efficiently utilize the heterogeneous storage devices on modern HPC clusters. But none of these researches have explored NVM technology for HDFS. In this paper, we, therefore, analyze the performance potential for incorporating NVM in HDFS and finally propose designs to leverage the byte-addressability of NVMs for HDFS communication and I/O.

7. CONCLUSION

This paper proposes a novel design of HDFS to leverage the byte-addressability of NVM. Our design minimizes the memory contention of computation and I/O by allocating memory from NVM for RDMA-based communication. We re-design the HDFS storage architecture to exploit the memory semantics of NVM for I/O operations. We also propose advanced and cost-effective techniques for accelerating Spark and HBase by intelligently utilizing NVM in the underlying file system for only job outputs and Write Ahead Logs (WALs), respectively. We further propose enhancements to use our proposed NVFS as a burst buffer layer for running Spark jobs over parallel file systems, such as Lustre. In-depth performance evaluations show that, our design can increase the read and write throughputs of HDFS by up to 2x and 4x, respectively over default HDFS. It also achieves up to 45% benefit in terms of job execution time for data generation benchmarks. Our design reduces the overall execution time of the SWIM workload by 18% while providing a maximum gain of 37% for job-38. The proposed design can improve the performance of Spark TeraSort by 11%. The performances of HBase *insert*, *update*, and *read* operations are improved by 21%, 16%, and 26%, respectively. Our burst buffer design can also improve the I/O performance of Spark PageRank by up to 24% over Lustre.

In the future, we would like to apply the proposed designs for Tachyon and Kudu [7, 28] and evaluate how these storage systems influence the performance of different applications.

References

[1] Apache HBase. http://hbase.apache.org.

[2] Big data needs a new type of non-volatile memory. http://www.electronicsweekly.com/news/big-data-needs-a-new-type-of-non-volatile-memory-2015-10/.

[3] Hadoop 2.6 Storage Policies. https://hadoop.apache.org/docs/stable/hadoop-project-dist/hadoop-hdfs/ArchivalStorage.html.

[4] HDFS. https://hadoop.apache.org/docs/r1.2.1/hdfs_design.html.

[5] HiBD. http://hibd.cse.ohio-state.edu/.

[6] IDC. www.idc.com.

[7] Kudu. https://blog.cloudera.com/blog/2015/09/kudu-new-apache-hadoop-storage-for-fast-analytics-on-fast-data/.

[8] NVMe. http://www.nvmexpress.org/.

[9] NVRAM. http://www.enterprisetech.com/2014/08/06/flashtec-nvram-15-million-iops-sub-microsecond-latency/.

[10] Statistical Workload Injector for MapReduce. https://github.com/SWIMProjectUCB.

[11] TeraGen. http://hadoop.apache.org/docs/r0.20.0/api/org/apache/hadoop/examples/terasort/TeraGen.html.

[12] G. Ananthanarayanan, A. Ghodsi, A. Wang, D. Borthakur, S. Kandula, S. Shenker, and I. Stoica. PACMan: Coordinated Memory Caching for Parallel Jobs. In *9th USENIX Conference on Networked Systems Design and Implementation (NSDI)*, 2012.

[13] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber. Bigtable: A Distributed Storage System for Structured Data. In *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2006.

[14] Y. Chen, S. Alspaugh, and R. Katz. Interactive Analytical Processing in Big Data Systems: A Cross-industry Study of MapReduce Workloads. *Proc. VLDB Endow.*, 2012.

[15] J. Condit, E. B. Nightingale, C. Frost, E. Ipek, D. Burger, B. Lee, and D. Coetzee. Better I/O Through Byte-Addressable, Persistent Memory. In *Symposium on Operating Systems Principles (SOSP)*, 2009.

[16] S. R. Dulloor, S. Kumar, A. Keshavamurthy, P. Lantz, D. Reddy, R. Sankaran, and J. Jackson. System Software for Persistent Memory. In *Proceedings of the Ninth European Conference on Computer Systems (EuroSys)*, 2014.

[17] T. Harter, D. Borthakur, S. Dong, A. Aiyer, L. Tang, A. Arpaci-Dusseau, and R. Arpaci-Dusseau. Analysis of HDFS Under HBase: A Facebook Messages Case Study. In *12th USENIX Conference on File and Storage Technologies (FAST)*, 2014.

[18] J. Huang, K. Schwan, and M. Qureshi. NVRAM-aware Logging in Transaction Systems. In *41st International Conference on Very Large Data Bases (VLDB)*, 2015.

[19] N. S. Islam, X. Lu, M. W. Rahman, and D. K. Panda. SOR-HDFS: A SEDA-based Approach to Maximize Overlapping in RDMA-Enhanced HDFS. In *23rd International ACM Symposium on High-Performance Parallel and Distributed Computing (HPDC)*, 2014.

[20] N. S. Islam, X. Lu, M. W. Rahman, D. Shankar, and D. K. Panda. Triple-H: A Hybrid Approach to Accelerate HDFS on HPC Clusters with Heterogeneous Storage Architecture. In *15th IEEE/ACM Intl. Symposium on Cluster, Cloud and Grid Computing (CCGrid)*, 2015.

[21] 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 *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 2012.

[22] N. S. Islam, M. W. Rahman, X. Lu, D. Shankar, and D. K. Panda. Performance Characterization and Acceleration of In-Memory File Systems for Hadoop and Spark Applications on HPC Clusters. In *2015 IEEE International Conference on Big Data (IEEE BigData)*, 2015.

[23] W. K. Josephson, L. A. Bongo, K. Li, and D. Flynn. DFS: A File System for Virtualized Flash Storage. *Trans. Storage*, 2010.

[24] K. Massey. Worldwide Financial Services 3rd Platform IT Spending, 2014 - 2019 - Opportunities Abound. http://www.idc.com/getdoc.jsp?containerId=US40697215.

[25] K. R. Krish, A. Anwar, and A. Butt. hatS: A Heterogeneity-Aware Tiered Storage for Hadoop. In *14th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*, 2014.

[26] K. R. Krish, S. Iqbal, and A. Butt. VENU: Orchestrating SSDs in Hadoop Storage. In *2014 IEEE International Conference on Big Data (IEEE BigData)*, 2014.

[27] H. Li, A. Ghodsi, M. Zaharia, S. Shenker, and I. Stoica. Tachyon: Reliable, Memory Speed Storage for Cluster Computing Frameworks. In *ACM Symposium on Cloud Computing (SoCC)*, 2014.

[28] T. Lipcon, D. Alves, D. Burkert, J. Cryans, A. Dembo, M. Percy, S. Rus, D. Wang, M. Bertozzi, C. P. McCabe, and A. Wang. Kudu: Storage for Fast Analytics on Fast Data. http://getkudu.io/kudu.pdf.

[29] N. Liu, J. Cope, P. Carns, C. Carothers, R. Ross, G. Grider, A. Crume, and C. Maltzahn. On the Role of Burst Buffers in Leadership-Class Storage Systems. In *2012 IEEE Conference on Massive Data Storage*, 2012.

[30] S. Pelley, T. F. Wenisch, B. T. Gold, and B. Bridge. Storage Management in the NVRAM Era. *Proc. VLDB Endow.*, 2013.

[31] S. Qiu and A. L. N. Reddy. NVMFS: A Hybrid File System for Improving Random Write in Nand-Flash SSD. In *IEEE 29th Symposium on Mass Storage Systems and Technologies (MSST)*, 2013.

[32] K. R, A. Khasymski, A. Butt, S. Tiwari, and M. Bhandarkar. AptStore: Dynamic Storage Management for Hadoop. In *International Conference on Cluster Computing (CLUSTER)*, 2013.

[33] P. Sehgal, S. Basu, K. Srinivasan, and K. Voruganti. An Empirical Study of File Systems on NVM. In *IEEE 31st Symposium on Mass Storage Systems and Technologies, (MSST)*, 2015.

[34] K. Shvachko, H. Kuang, S. Radia, and R. Chansler. The Hadoop Distributed File System. In *IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)*, 2010.

[35] W. Tantisiriroj, S. Patil, G. Gibson, S. Son, S. Lang, and R. Ross. On the Duality of Data-intensive File System Design:Reconciling HDFS and PVFS. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 2011.

[36] The Apache Software Foundation. Centralized Cache Management in HDFS. http://hadoop.apache.org/docs/r2.3.0/hadoop-project-dist/hadoop-hdfs/CentralizedCacheManagement.html.

[37] The Apache Software Foundation. The Apache Hadoop Project. http://hadoop.apache.org/.

[38] T. Wang, K. Mohror, A. Moody, W. Yu, and K. Sato. BurstFS: A Distributed Burst Buffer File System for Scientific Applications. In *The International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 2015.

[39] M. Welsh, D. Culler, and E. Brewer. SEDA: An Architecture for Well-Conditioned, Scalable Internet Services. In *18th ACM Symposium on Operating Systems Principles (SOSP)*, 2001.

[40] X. Wu and A. L. N. Reddy. SCMFS: A File System for Storage Class Memory. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 2011.

[41] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica. Spark: Cluster Computing with Working Sets. In *Proceedings of the 2Nd USENIX Conference on Hot Topics in Cloud Computing (HotCloud)*, 2010.

[42] Y. Zhang, J. Yang, A. Memaripour, and S. Swanson. Mojim: A Reliable and Highly-Available Non-Volatile Memory System. In *20th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2015.