Exploration of Memory Hybridization for RDD Caching in Spark

Md Muhib Khan
Florida State University
Florida, USA
khan@cs.fsu.edu

Muhammad Ahad Ul Alam
Florida State University
Florida, USA
alam@cs.fsu.edu

Amit Kumar Nath
Florida State University
Florida, USA
nath@cs.fsu.edu

Weikuan Yu
Florida State University
Florida, USA
yuw@cs.fsu.edu

Abstract

Apache Spark is a popular cluster computing framework for iterative analytics workloads due to its use of Resilient Distributed Datasets (RDDs) to cache data for in-memory processing. We have revealed that the performance of Spark RDD cache can be severely limited if its capacity falls short to the needs of the workloads. In this paper, we have explored different memory hybridization strategies to leverage emergent Non-Volatile Memory (NVM) devices for Spark's RDD cache. We have found that a simple layered hybridization approach does not offer an effective solution. Therefore, we have designed a flat hybridization scheme to leverage NVM for caching RDD blocks, along with several architectural optimizations such as dynamic memory allocation for block unrolling, asynchronous migration with preemption, and opportunistic eviction to disk. We have performed an extensive set of experiments to evaluate the performance of our proposed flat hybridization strategy and found it to be robust in handling different system and NVM characteristics. Our proposed approach uses DRAM for a fraction of the hybrid memory system and yet manages to keep the increase in execution time to be within 10% on average. Moreover, our opportunistic eviction of blocks to disk improves performance by up to 7.5% when utilized alongside the current mechanism.

CCS Concepts • **Information systems → MapReduce-based systems**; **Storage class memory**.

Keywords hybrid memory, apache spark, big data

ISMM '19, June 23, 2019, Phoenix, AZ, USA

ACM ISBN 978-1-4503-6722-6/19/06…$15.00
https://doi.org/10.1145/3315573.3329988

ACM Reference Format:
Md Muhib Khan, Muhammad Ahad Ul Alam, Amit Kumar Nath, and Weikuan Yu. 2019. Exploration of Memory Hybridization for RDD Caching in Spark. In *Proceedings of the 2019 ACM SIGPLAN International Symposium on Memory Management (ISMM '19), June 23, 2019, Phoenix, AZ, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3315573.3329988

1 Introduction

In recent years, big data analytics has seen wide adoption in many places, ranging from industry leading enterprises to small startups and research labs. In-memory distributed processing frameworks, e.g., Apache Spark [28], M3R [19] have seen tremendous success and growth in use because of their incredible speed and performance that outclass well-established frameworks like Hadoop [6]. The impressive success of these in-memory cluster computing frameworks can largely be attributed to their ability of processing data in the memory without involving the disk, thus avoiding serialization cost and reducing a significant amount of I/O overhead [13, 26].

On the other hand, the advent of "big data" has put tremendous pressure on the existing data analytics systems that has to process terabytes or even petabytes worth of data routinely [29]. The speed at which the datasets are getting bigger outpaces the drop in DRAM price, and memory remains a scarce resource in these data analytics clusters [27] [3]. Apache Spark uses Resilient Distributed Datasets (RDDs) to store data in memory that can be reused efficiently and recomputed using the dependency graph (RDD lineage) in a fault-tolerant manner. Utilization of RDDs to avoid recomputation of frequently required data in iterative workloads significantly improves the application execution time but puts a huge burden on the memory system. When working with large datasets, memory constraint can force Spark to either drop the RDDs from memory or write them to disk. Dropping critical RDDs from memory can lead to a large drop in performance which diminishes the appeal of using in-memory frameworks.

Non-Volatile Memory (NVM) has been touted as a solution to the ever growing need for memory. NVM can be coupled with Dynamic Random Access Memory (DRAM) to create heterogeneous memory systems having higher capacity and lower cost. Although NVM offers much better I/O capabilities, treating it like a disk will incur software overheads that will introduce a performance bottleneck [7]. When incorporating NVM in the memory system to achieve hybridization, cluster computing frameworks, e.g., Apache Spark needs to redesign their software stack responsible for handling in-memory data to take advantage of the benefits of NVM (e.g., byte addressability, higher capacity) while minimizing the disadvantages (e.g., higher access latency).

In this paper, we explore different hybridization approaches for incorporating NVM in the caching component of Apache Spark. We first consider a simple layered hybridization approach which is attractive due to its simplicity of implementation and integration. Due to inheriting current mechanisms that are not optimized for handling hybrid memory, the layered approach suffers from poor performance. We closely inspect the mechanisms involved in caching to identify the problems and propose a flat hybridization approach along with three optimizations to the caching mechanism. Our proposed optimizations include dynamic memory allocation for block unrolling (dynamic unrolling) that avoids stalling due to synchronous transfer by switching between memory regions as required. We also propose a technique termed "asynchronous migration with preemption" that works with dynamic unrolling to avoid synchronous transfer of RDD blocks between memory regions. Finally, we propose a mechanism that monitors the RDD cache to detect imminent disk eviction and writes blocks to the disk ahead of time to achieve better I/O and computation overlap, which we term as opportunistic eviction to disk. Our flat hybridization approach proves to be robust in handling different scenarios of DRAM/NVM ratio and NVM bandwidth as the increase in execution time compared to a DRAM-only system is limited to 8%, 6% and 16% respectively for the tested workloads of *PageRank*, *ConnectedComponent*, and *PregelOp*.

The rest of the paper is organized as follows - Section 2 introduces some background on Apache Spark and hybrid memory architecture. In Section 3, we explore the possible hybridization strategies and present insights achieved from our implementation of layered hybridization of the RDD cache. Section 4 presents the design for a flat hybridization of the RDD cache along with our architectural optimizations. Section 5 gives details of our implementation and experimental results are presented in Section 6. We discuss the related works in Section 7 and finally conclude the paper in Section 8.

2 Background and Motivation

2.1 Apache Spark

Apache Spark is a framework which enables in-memory data processing to gain scalability over MapReduce and thus offers strong support for iterative workloads [18], such as K-Means and PageRank. Spark provides APIs in different languages (e.g., Java, Scala, Python, and R); however, internally it is written in Scala and Java and runs atop JVMs. Spark deals with the fault tolerance of the large scale data processing system by introducing the abstraction of lazily computed Resilient Distributed Datasets (RDDs). An RDD is a collection of objects that are partitioned across the nodes of the cluster. These RDD partitions are also referred to as RDD blocks. In Spark, there are several types of RDDs (e.g., ShuffledRDD, MapPartitionsRDD) and each RDD can be (re)computed from its parent RDDs using a dependency graph or lineage. A stage is a physical unit of execution in Spark where an RDD is transformed from one type to another through a series of transformations. An RDD is divided into multiple partitions so that each task operates on one partition in parallel. Spark applications can choose to persist any RDD in memory to avoid recomputation when needed. Spark internally controls the allocation of memory for different purposes (e.g., caching of RDD blocks, execution memory for join operation) through a MemoryManager module where one single instance exists per JVM. The BlockManager module provides the interface for putting and getting RDD blocks from different storage options (e.g., memory, disk) and manages metadata for the cached RDD blocks.

Memory Requirements from RDD: While memory resident RDDs greatly benefit iterative processing, they do pose a hefty requirement on the available memory. A Spark-based cluster typically consists of a driver node and several worker nodes where multiple executors will launch the actual Spark tasks. Aside from a small fraction of memory (typically 300 MB) for system use, each executor splits its memory between user created data structures and the Spark internal functions (storage and execution related to RDD computation, shuffling and caching). By default, the latter in total is allocated 60% of the available JVM heap memory. In turn, half of this allocation is reserved for caching RDD blocks where they are immune from eviction due to the execution memory pressure. Evidently, this amounts to nearly 30% of memory being reserved for caching of RDD blocks. At run time, memory demands from user tasks and other Spark execution activities can fluctuate, resulting in constant competition for memory. When there is not enough memory for all RDD blocks, some will simply be dropped from the RDD Cache, or evicted to disk if Spark RDD blocks are allowed to reside on both memory and disk, i.e., `MEMORY_AND_DISK`.

2.2 Performance Impact of Limited RDD Cache

When an application requires more memory for RDD caching than what Spark has provisioned, severe performance degradation can occur. To quantify the impact of limited RDD cache, we ran *PageRank (PR)* and *ConnectedComponent (CC)* applications with 16 GB of input data on a Spark cluster with four workers and a driver node, each with a JVM heap size of 56 GB. By changing the configuration parameter `spark.memory.fraction`, we vary the amount of memory for Spark. We measure the resulting execution time and normalize them against the execution time when 90% of the JVM heap is allocated for Spark.

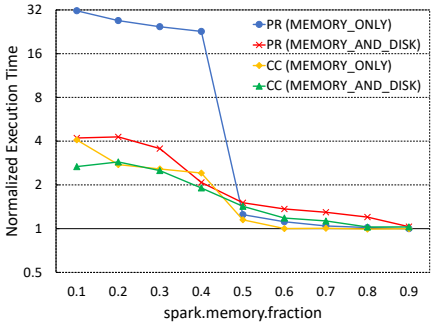

Figure 1. Spark Execution with Limited RDD Cache.

As shown in Figure 1, when RDD blocks are only cached in memory (`MEMORY_ONLY`), workload execution time dramatically increases as we reduce the allocated memory to be less than 50%, in which case many critical RDD blocks need to be repeatedly recomputed using the dependency graph. When the allocation is only 10%, the execution time can be as much as 32x and 3x, respectively for *PageRank* and *ConnectedComponent*. Even if eviction to disk is allowed for RDD blocks, i.e., when the cache level is `MEMORY_AND_DISK`, Spark execution time still increases significantly. When the allocation is only 10%, the degraded execution time can be as much as 4x and 2.8x, respectively for *PageRank* and *ConnectedComponent*.

We also observe that when the allocated memory is sufficient, i.e., ($\geq$ 50%), eviction of non-critical RDDs to disk actually hurts the execution time (caching level is set to `MEMORY_AND_DISK`). As documented by Zhang et al. [30], unnecessary movement of RDD blocks to and from disk can cause performance degradation due to the (de)serialization cost and the disk I/O.

Our experiments demonstrate that, without sufficient memory, the performance of iterative workloads can degrade severely. Even with the help of disk to store evicted blocks, the performance degradation is quite alarming.

2.3 NVM and Hybrid Memory Architectures

Non-Volatile Memory (NVM) is an emerging technology that has already attracted both industry and academia due to its potential in meeting the need for cheaper byte-addressable memory. Due to being denser and cheaper, NVM can be a feasible alternative to DRAM. However, NVM has several disadvantages (e.g. higher read/write latency, higher write power consumption) that require considerable attention and prevents a direct substitution of DRAM by NVM in most systems. NVM read latency can vary from being similar to DRAM to being up to 4x higher [3, 24]. According to Dulloor et al. [3], NVM's slower write translates to lower bandwidth to NVM, as writes to write-back cacheable memory are posted. This bandwidth can be as low as 1/10th [8] of DRAM or even lower [23], depending on the NVM technology. Hence, hybrid memory architectures utilizing a small fraction of DRAM and a large amount of NVM have been touted to be more promising which exploit the cost and capacity benefits of NVM while minimizing the disadvantages of NVM with DRAM [1, 5, 23].

Existing hybrid memory architectures can generally be categorized into two types, layered hybrid memory systems, and flat-addressable hybrid memory systems. **Layered hybrid memory systems** commonly use the DRAM as a cache to the NVM [9, 11, 14, 15]. The DRAM is transparent to the OS in this case and is entirely managed by the hardware. Layered hybrid memory systems require extra storage space to store the metadata for tracking the data blocks in the DRAM and also usually employ on-demand cache fetching policies. **Flat-addressable hybrid memory systems** form the main memory by organizing the DRAM and NVM in the same address space [10, 16, 31]. These hybrid systems overcome the disadvantages of the NVM by migrating frequently accessed pages to the DRAM for avoiding higher access latency. However, page migrations involve multiple read/write operations, which can be costly as NVM has a much higher write latency than DRAM. Flat-addressable hybrid memory systems employ additional hardware in the memory controller to track hot pages. In the rest of the paper, we explore both strategies of memory hybridization in the context of Spark to leverage the benefits provided by NVM for RDD caching.

3 Exploration of Hybridization Strategies for RDD Cache

Spark currently provides a layered architecture for the caching of RDD blocks [30] using both the memory and the disk. RDD blocks cached at the memory level are managed under the `MemoryStore` and those in the disk are handled by the `DiskStore`. In this section, we explore two main strategies, layered and flat hybridization, to leverage the NVM as an additional source of RDD cache. Due to the challenge posed by the JVM in emulating NVM, we use a simple methodology for emulation of NVM in our system. The details about the methodology are described in Section 5.2. Based on our exploration, we offer some observations about the issues that need to be tackled when hybridizing the RDD cache.

3.1 Comparison of Hybridization Strategies

Layered Hybridization: A straightforward and non-intrusive strategy is to introduce NVM as a separated and intermediate level between memory and disk, as shown by the left portion in Figure 2. Due to its layered composition, its is termed as *Layered Hybridization*. This strategy retains the modularity of the current implementation of RDD cache in Spark. Memory, NVM and disk-based RDD stores are all managed through discrete modules. Each level is opaque to the internal intricacies of the other ones, while they still coordinate together to allow the migration of RDD blocks across them through their caching and eviction mechanisms.This modularity leads to a straightforward implementation for the stores, as the DRAM store (`MemoryStore`) can be left as it is and the NVM store can be inserted between the `MemoryStore` and the `DiskStore`. This architecture allows for a completely pluggable interface that should be able to integrate NVM with minimal changes in the implementation. The blessings of simplicity and modularity of Layered Hybridization can easily become its curses when it comes to the need of transparency for block management and migration across multiple levels. The modular and opaque design offers only a local visibility of RDD blocks at each level, preventing a globally optimal decision on the placement and mobility of blocks across multiple levels.

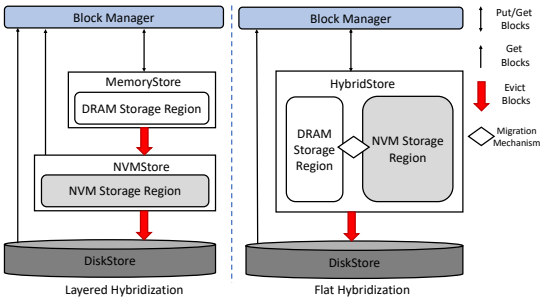

Figure 2. Comparison of Layered and Flat Hybridization for RDD Cache.

Flat Hybridization: In contrast to the layered strategy, another strategy is to leverage NVM as an expansion of DRAM, and integrate NVM and DRAM seamlessly together for RDD caching. This is shown as the *Flat Hybridization* scheme in Figure 2. Under the flat scheme, both DRAM and NVM will be managed within the same store. This scheme can allow for much better visibility across both memory regions and better migration strategy of RDD blocks across DRAM and NVM for performance gains. However, flat hybridization requires significant architectural changes in the design of `MemoryStore` for DRAM and NVM regions to be used in a coordinated manner. We will elaborate the flat hybridization scheme in Section 4.

3.2 Assessment of Layered Hybridization

For its simplicity and modularity, we have developed a conceptual implementation of layered hybridization which is achieved by introducing an independent block store for NVM (`NVMStore`), and interfacing it as the intermediary between the `MemoryStore` and the `DiskStore`. In this way, we can achieve a hybridization with minimal changes to the relevant Spark modules. We evaluate the performance impact of this layered hybridization, using a combination of DRAM and NVM, which we emulate by introducing appropriate read/write latencies.

Impact of NVM latency and bandwidth: We use *PageRank* in our tests to measure the impact of NVM latency. In our test, we configure a hybrid RDD cache with 1/8th as DRAM and the rest NVM, and compare its performance to an RDD cache composed of only DRAM. We first measure the impact of higher read latency while keeping the write bandwidth fixed. We have observed a minor performance degradation of 2-4% when the read latency of NVM ranges from 2x and 4x to 8x of the DRAM (results omitted for brevity). This test indicates that higher read latency of NVM does not influence the performance of our workloads to a great extent. On the other hand, the performance impact of NVM bandwidth exhibits a different trend. Lower NVM bandwidth has a severe effect on the performance of the *PageRank* workload. As shown in Figure 3, when the NVM bandwidth is set to be 1/16 of the DRAM, the application execution time increases significantly, by up to almost 1.5 times compared to the DRAM-only case. Since the performance is so closely correlated with the NVM bandwidth, we further examine the block management process in Spark to pinpoint the underlying issues.

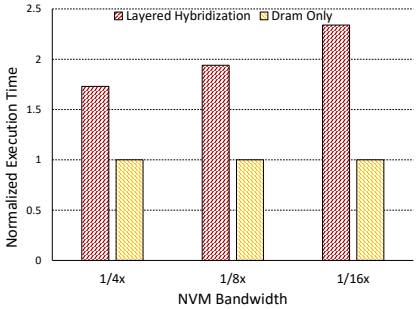

Figure 3. Impact of NVM write bandwidth for the layered organization of RDD cache. NVM write bandwidths are expressed as fractions of the DRAM write bandwidth.

3.3 Block Management Issues in Spark RDD Cache

In Spark, the *BlockManager* is responsible for interacting with the underlying block stores, e.g., `MemoryStore`, `DiskStore` for putting a block or reading from them. When a task requests an RDD block from the `BlockManager` which is marked for caching, but currently is not present in any of the stores, the `MemoryStore` tries to compute and store the

block if possible. The developers of Spark combined the computation and caching process of a block and termed it as the *Unrolling* process. An un-computed block is passed as an iterator that can be looped through to compute the values of the RDD block one by one. The `MemoryStore` periodically checks whether it should acquire more memory for the temporary vector that is storing the computed values. If all the values are successfully computed and put into the vector, it is converted into an array and stored as a block in the `MemoryStore`. This process of periodically checking whether more memory is needed, allocating space for to be computed values and finally registering the block with the `BlockManager` is termed as the *Unrolling* process. When there is not enough memory for expanding the unroll vector, the `MemoryStore` evicts blocks to make space for unrolling.

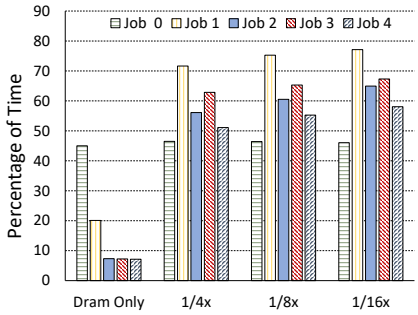

Figure 4. The percentage of time spent on average for allocating memory in the unrolling process of tasks of the *PageRank* workload. Only a few jobs are shown here for brevity. The rest of the jobs continue to show this pattern of increased allocation time. 1/4x, 1/8x and 1/16x denote the NVM bandwidth used by the layered hybridization approach where *x* denotes DRAM bandwidth. 1/8th of the hybrid memory is DRAM here.

We instrument the code of the unrolling process to detect the bottleneck when the stores are organized in a strictly layered manner. The unrolling process can be thought to have two components, one is computing new values and the other is allocating memory for expanding the unroll vector. From Figure 4, we see that in most cases the percentage of time spent allocating memory in the layered hybridization takes up more than 50% of time in the unrolling process, while in the DRAM only case, it usually takes less than 10%.

In Figure 5, where we depict the process of freeing up space for allocating unroll memory, we see that the process is synchronized and only a single task thread gets to enter the critical block. In the face of memory contention, one task thread frees up space for itself while the computation of new values stalls. If other task threads require more memory during the process, they need to wait for acquiring the lock for evicting LRU blocks. This can lead to increased waiting time for allocating memory in the unrolling process, which is what we see in Figure 4.

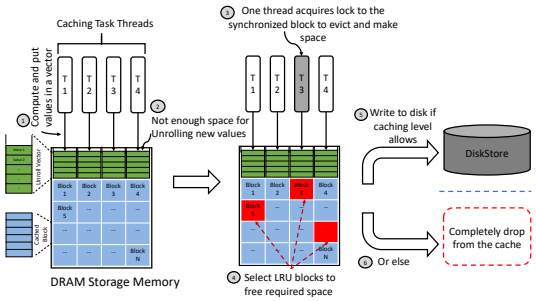

Figure 5. The process of freeing up memory for unrolling of an RDD block.

Here we summarize our observations from the examination of Spark RDD cache.

Observation 1: Memory contention during unrolling leads to increased execution time. When we reduce the amount of DRAM space and organize the caching layers in a strict hierarchy, if we try to unroll a new block on the DRAM, the time taken by the memory allocation routine goes up due to increased wait time while the DRAM is being freed by the synchronized transfer mechanism.

Observation 2: The synchronized transfer mechanism is insufficient. The current mechanism processes a single RDD block at a time, while the other task threads wait. This is suitable when we completely drop the block, which takes no time or write to disk, where sequential write is preferable. For transferring across memory regions, where random access is not a problem, we can employ multiple threads to transfer blocks asynchronously.

Observation 3: NVM read latency does not influence the execution time to any noticeable degree. We do not need to move frequently accessed blocks to the DRAM as the effect of read latency is very minimal. RDD blocks are read in parallel and the increase in read latency is amortized across all the task executions.

Observation 4: Low NVM bandwidth can degrade performance if it is on the critical path. Lower bandwidth of NVM has a noticeable effect because execution of task threads can stall while a block is being transferred.

4 Flat Hybridization of RDD Cache

Based on our assessment of layered hybridization and revelation of drawbacks in the Spark RDD cache, we propose to design flat hybridization for addressing them through a combination of three techniques, namely dynamic memory allocation for block unrolling, asynchronous migration with preemption, and opportunistic eviction to disk. We describe these techniques in detail in the rest of this section.

4.1 Dynamic Memory Allocation for Block Unrolling

From our observations, we know that memory contention during the unrolling process leads to increased execution

time. We design our hybrid memory system as a flat-addressable space to avoid placing all newly computed blocks entirely in either the DRAM or NVM. We switch between memory regions to allocate memory for block unrolling depending on a couple of indicators.

We term our proposed technique as **Dynamic Memory Allocation for Unrolling** or in short dynamic unrolling. If the required memory amount exceeds the free memory space in DRAM, the requesting task thread must invoke the synchronous transfer mechanism to make space and ensure allocation. As an invocation of the synchronous mechanism has much overhead associated with it, we avoid it by monitoring the free space left in the DRAM region. We keep a rolling average of the size of cached RDD blocks in a stage (a phase of execution in Spark). As blocks cached during a particular stage are very similar in size (blocks of the same RDD), we can use this as an indicator or reference to predict how big the unrolling vector of the requesting thread may become. We calculate a threshold using the average cached block size. If the requested memory amount or the calculated threshold is greater than the current free memory in the DRAM, we switch to NVM as the region for allocating memory for the requesting thread. We have found that using a multiplier of two with the average block size to calculate the threshold is conservative enough to handle the variation in the amount of memory required per task.

Using dynamic unrolling to switch between DRAM and NVM region helps eliminate the problem of stalling during the unrolling process and we pair this with the following two techniques to avoid memory contention during unrolling.

4.2 Asynchronous Migration with Preemption

From our observations, we know that the current block transfer or eviction mechanism in Spark is designed for either writing blocks to the disk in a sequential manner or completely drop blocks from the memory. To better utilize the random access capability and byte-addressability of NVM, we design a mechanism that synergizes with dynamic unrolling to ensure computation of RDD blocks without stalling.

Generally, in a hybrid system the amount of DRAM is a small fraction of the total memory, thus in a point of time during the execution of a workload, we may find that the DRAM is full with RDD blocks. If we place new blocks in the completely occupied DRAM, the synchronous transfer mechanism will be invoked to ensure that the task gets enough memory for unrolling, which we want to avoid for the associated overhead. If we use the dynamic unroll routine to avoid DRAM, but no mechanism removes blocks from DRAM after it becomes full, every newly created block after that point in time will be placed in the NVM. We remedy this potential problem by preemptively migrating blocks from the DRAM to NVM in the background while dynamic unrolling switches between regions depending on the availability of memory.

In our proposed mechanism, we aim to avoid serial transfer of blocks and opt for concurrent transfer using multiple threads. As a transfer between DRAM and NVM regions does not require CPU-intensive serialization and deserialization operation, these threads are only bound by the I/O capability of the NVM device.

Algorithm 1 Monitor memory usage and Initiate migration

Require: Caching is ongoing in the current Stage

```
1:  procedure MonitorMemoryUsage
2:      dramUsage ← getDramUsage()
3:      if dramUsage ≥ InitiatingThreshold then
4:          multiplier ← Min(multiplier, waveWidth)
5:          tctSize is the total concurrent transfer amount
6:          tctSize ← avgBlockSize × multiplier
            ▷ Increase multiplier for later invocations
7:          multiplier ← Min(multiplier + 1, waveWidth)
            ▷ curTrans is the current transfer amount
8:          targetSize ← tctSize − curTrans
9:          while size < targetSize do
10:             selectedBlock ← getLruBlock()
11:             size += getSize(selectedBlock)
12:             blocks += selectedBlock
13:         end while
14:         submitToMigrartionService(blocks)
15:     else
            ▷ Reset migration multiplier
16:         multiplier ← Max(waveWidth ÷ 2, 1)
17:     end if
18: end procedure
```

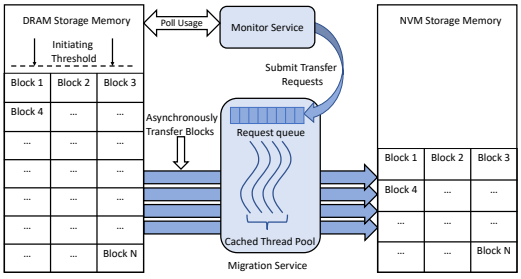

Figure 6. Asynchronous Migration with Preemption

Combining these ideas, we design a mechanism named **Asynchronous Migration with Preemption (AMP)**. Figure 6 depicts the two components of AMP, one is the monitoring service, and another is the migration service. The migration service uses a pool of cached threads to handle block transfer requests that are submitted by the monitoring service. We show the process of monitoring and submitting transfer requests to the migration service in Algorithm 1. The monitoring service invokes this procedure repeatedly in intervals given that the caching of RDD blocks is ongoing

in the current stage. If the procedure detects that free memory in the DRAM has gone below an *InitiatingTheshold*, it calculates a target transfer amount and selects LRU (Least Recently Used) blocks to submit them to our asynchronous migration service. Several factors are used to determine the transfer size. We keep a rolling average of the cached block size of the current stage and update it whenever a new block is cached. As the wave width or the number of available cores to the application is generally the number of threads concurrently caching, we use half of that number as a multiplier to the average cached block size to calculate the initial transfer amount conservatively. If we detect that the DRAM usage by RDD caching continues to be greater than the configured threshold, we increase the transfer amount by gradually incrementing the multiplier to the average cached block size. We ramp up until the multiplier reaches the available core count of the application to aggressively migrate blocks to the NVM from the DRAM.

4.3 Opportunistic Eviction to Disk

The **AMP** mechanism is sufficient for the default caching level (`MEMORY_ONLY`), where we only need to migrate blocks between two memory regions. However, for caching level `MEMORY_AND_DISK`, where blocks evicted from memory must be written to disk, we propose further optimizations in the system. In this optimization, we aim for an overlap of computation and disk I/O to make eviction to disk faster. However, this requires careful consideration of different factors.

Unlike NVM, the random write capability of traditional disk drives is quite limited and reading RDD blocks back from the disk involves the costly process of deserialization. If we are too aggressive in evicting blocks to the disk, application performance can get degraded for several reasons. If the blocks are needed after they were aggressively evicted, reading them back from disk introduces unnecessary overhead. Also if we constantly evict blocks to disk too frequently, it can cause the disk to perform poorly, as it struggles to keep up with the write requests. Considering these factors, we propose to opportunistically write a limited number of blocks to the disk when eviction due to memory pressure is imminent. Using a monitoring service, we detect when eviction to disk is imminent and write a limited number of cached blocks using a dedicated daemon thread. The written blocks are kept in memory for the time being and are tracked through a list of processed blocks. When the memory is full, and the eviction mechanism kicks in, the processed blocks are given priority for eviction from memory. As the processed blocks are already in the disk when their eviction request is handled, all the eviction mechanism has to do is to clear out the space used by them and skip the disk writes.

The monitor service uses an eviction window threshold for detecting whether disk write is imminent or not. We calculate the threshold value using the average amount of eviction per event multiplied by a tunable configuration parameter. The disk eviction service uses the LRU policy to select blocks with their size totaling up to the eviction window amount and writes them to the disk. The monitor service detects the use of processed blocks and submits new requests if the amount of free memory is shrinking and disk eviction window has space for more blocks.

5 Implementation

In this section, we discuss our implementation details. We have implemented the proposed flat hybrid architecture in `HybridStore`, which is a pluggable block store for Spark. The default `MemoryStore` implementation and our `HybridStore` is extended from a high-level abstract class and thus can be instantiated and plugged in interchangeably depending on the configuration. Besides the standard methods for interacting with the other modules of Apache Spark, we have implemented the proposed mechanisms inside our `HybridStore`. We also implemented an `NvmEmulator` that can be used by other modules, e.g., `HybridStore` to calculate and add delays for emulating reads and writes to the NVM region. The Key APIs of our implementation are listed in Table 1.

5.1 HybridStore

The **Monitoring Service** is implemented as an executor service with a single daemon thread, which is scheduled to run periodically with a fixed delay. Currently, we use a few milliseconds as the fixed delay period after which the monitoring service submits transfer requests to the migration service if proper conditions are met.

We implement the **Migration Service** as an executor service with a cached daemon thread pool. Caching the migration threads allows for reuse without having to invoke thread creation routines excessively. We also limit the maximum number of threads in the executor service to the number of cores allocated for the corresponding Spark application.

Dynamic Unrolling is incorporated in our modified memory allocation for unrolling routine. Depending on the passed configuration, our `HybridStore` can switch between the default or the dynamic allocation method.

The **Disk Eviction Service** is implemented as a single threaded executor service with a limited queue size. When the number of requests for disk eviction exceeds the queue size, instead of running it, the monitoring service is handed back the work which serves as a feedback mechanism to slow the request rate down if necessary.

5.2 NVM Emulation

Although NVM is a technology that is on the cusp of becoming mainstream, the required hardware is still not that widely available. We have thus evaluated several approaches to emulate the read and write characteristics of NVM devices. We have explored the choice of introducing latency using emulation frameworks against instrumenting the code of

Table 1. Key APIs of Our Implementation

API	Description
`putBlockInNvm`	The `HybridStore` moves the specified block from DRAM to NVM. The transfer time is calculated and emulated through the `NvmEmulator`.
`updateAvgCached-BlockSize`	The `HybridStore` updates the average size of the cached blocks for the current active stage. Called after each block is successfully cached.
`monitorUsage(x)`	Called periodically by the `monitor service`. Checks DRAM or NVM usage and submits requests to the migration service or disk eviction service respectively.
`reserveUnrollMemory-Dynamic`	Dynamically allocate memory for the reqeusting task.

Apache Spark. Lightweight NVM emulation frameworks like Quartz [21] and HME [2] provide options to applications for emulating NVM and hybrid memory. Unfortunately, they are geared towards languages like C and C++ and do not provide support for JVM languages, e.g., Java and Scala. Also, they make use of remote NUMA nodes to emulate NVM access and do not make use of all the CPU cores in the remote NUMA nodes. All these obstacles preclude us from using an established emulation framework, i.e., Quartz in our work. So, we have chosen to instrument the code of Apache Spark to introduce the required latency by injecting software induced delays.

Software-induced delays can be injected during the execution of a program in many ways. Researchers have used RDTSC [24][12] instruction of the processor to introduce a specific amount of delay by spinning until the processor timestamp counter reaches the intended delay to emulate access to NVM during the execution of their software. For JVM-based research works, a common approach for introducing latency is to utilize Java `Thread.sleep()` function. For example, Islam et al. [8] and Rahman et al. [22] used this approach to introduce latency in the write operation of an HDFS block and shuffle output file respectively. In both cases, the data size written to NVM is large enough such that the added latency is expected to be at the level of several milliseconds. However, RDD blocks can be as small as a few megabytes, which requires the injected latency to be at a finer granularity.

In our experiments, we have seen (Table 2) that Java thread sleep is unable to introduce a latency of length shorter than 1 ms (Section 6.1 gives details about the testbed). For cases where the required latency is shorter than 1 ms, using `Thread.sleep()` will introduce a significant amount of error to our emulation procedure. To tackle this problem, we implement a shared library that uses RDTSC instruction to loop for the specified amount of time, and we invoke the required functions of this library from Spark code through a JNI interface. In this way, we can introduce latency far more accurately when the time is shorter than 1 ms.

Table 2. Comparison of time taken by RDTSC and Thread Sleep. Unit of measurement is nanoseconds.

Latency	RDTSC Avg.	Thread Sleep Avg.
800	902	1110185
1600	1705	1111296
2000000	2000273	2107764
4000000	4000433	4110156

6 Evaluation

6.1 Experimental Setup

Cluster deployment: We deploy our implementation on our in-house cluster. Each of our used nodes is equipped with two 8-core 2.1 GHz Intel Xeon(R) E5-2620 CPUs, 64 GB of memory and a 7200-RPM 1 TB Seagate ST1000NM0033 SATA hard disk. Nodes are connected by a 10-Gigabit Ethernet network. Our implementation is based on Spark 2.1.1, and HDFS 2.7.3 is used for storing input and output data. All experiments are done using five nodes (one driver and four workers) of our system. The used JVM in our experiment is the OpenJDK 64-Bit Server VM with version 1.8.0_171. For all the experiments, we use a JVM heap size of 56 GB to allow resources for HDFS and other system activities to continue when our workloads are running.

Workloads: In our experiments we use *PageRank (PR)*, *ConnectedComponent (CC)* and *PregelOp* to test and analyze the implemented techniques. The selected workloads are heavily dependent on caching [27] and thus suitable for evaluating our implemention. We use SparkBench [20] to generate the different datasets used in the experiments. The workloads are modified and adapted from SparkBench for accepting our custom parameters. We use an input dataset of 16 GB for both *PageRank* and *ConnectedComponent* and a dataset of size 8 GB for *PregelOp*.

Unless otherwise specified, we allocate 90% of the heap for storage and execution purpose and use the caching level of `MEMORY_ONLY` for RDDs. We evaluate for a scenario where all the blocks fit in memory due to having a larger hybrid memory system. If no block is evicted due to memory pressure, `MEMORY_ONLY` and `MEMORY_AND_DISK` performs the same. In which case, `MEMORY_ONLY` is a good baseline for evaluating the impact of our hybridization techniques as it is the default caching level for RDDs. We use an *InitiatingThreshold* of 60% for the AMP mechanism. We consider NVM read latency to be comparable to DRAM and don't evaluate for other cases as the impact is very minimal and emulate as such.

6.2 Sensitivity to NVM bandwidth

In our first experiment, we evaluate the sensitivity of our hybridization approaches to NVM bandwidth for three workloads. In our tests, we use the same amount of total memory for both types of hybrid organization and the DRAM only case. We normalize the performance of the hybrid organizations against the DRAM-only baseline to see how well they perform using a lesser amount of DRAM and lower bandwidth of NVM. In the hybrid systems, we configure 1/8th of the total memory to be DRAM. As NVM still is a developing technology, we use a range of NVM bandwidths to evaluate the performance.

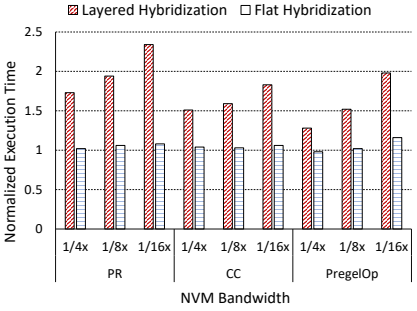

Figure 7. Sensitivity of hybridization approaches for different NVM bandwidths. Here DRAM is 1/8th of the total available memory. NVM bandwidth is varied from 1/4x of DRAM bandwidth to 1/16x of it.

Figure 7 shows the results of our evaluations. We see that with the reduction of NVM bandwidth, all three workloads experience a significant increase in execution time for the layered hybridization approach. The trend of increased execution time is correlated with the decrease in NVM bandwidth, as we recall from earlier discussion how invocation of the synchronized mechanism places the transfer of blocks in the critical path. With the flat hybridization approach, we see much better performance. For all the workloads, the execution time is very close to the baseline. In the case of *PageRank*, the increase in execution time is only 2-8%, for *ConnectedComponents* the increase is 4-6% and for *PregelOp* we see that the increase is within 16%. The performance of the flat-hybridization approach stems from avoiding the bottleneck of synchronous block transfer using dynamic allocation of memory for block unrolling and asynchronous migration of blocks between the DRAM and NVM region. As dynamic unrolling places some of the newly created blocks in the NVM directly, we do see an increase in execution time but that is much less compared to the layered approach. Thus, our experiments reveal that the flat hybridization approach copes very well with different NVM bandwidths and is an excellent choice for a hybrid RDD cache.

6.3 Sensitivity to DRAM/NVM ratio

We evaluate how the different hybridization approaches react to different DRAM/NVM ratios, as in a hybrid system, only a fraction of the total memory is expected to come from DRAM. For this set of experiments, we fix the NVM bandwidth to be 1/8th of the DRAM and vary the amount of DRAM available to the hybrid systems. We normalize our results with respect to a baseline DRAM-only system.

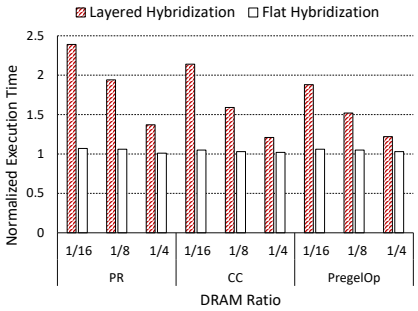

Figure 8. Sensitivity of hybridization approaches for different DRAM/NVM ratios. The NVM bandwidth is set to 1/8x of DRAM, while the DRAM/NVM ratio is varied.

Figure 8 depicts our results. As the amount of DRAM shrinks in a hybrid system, due to increased memory contention during the unrolling process, the layered hybridization approach suffers significantly. We see when the DRAM amount is 1/16th of the total amount, the execution time is 1.88 to 2.39 times of the DRAM-only system. If we increase the amount of DRAM to 1/4th of the total memory, we see a lot of improvement compared to the case when it is 1/16th, but still, the execution time ranges from 1.21 to 1.37 times of DRAM-only, which is not that good. The flat hybridization approach performs excellently for all three workloads. For *PageRank*, we see at most 7% degradation and *ConnectedComponent* and *PregelOp* sees an increase in execution time of at most 6%. So, the flat hybridization approach shows very little sensitivity to the reduction of DRAM amount in the hybrid system, which is very encouraging from a cost benefit standpoint.

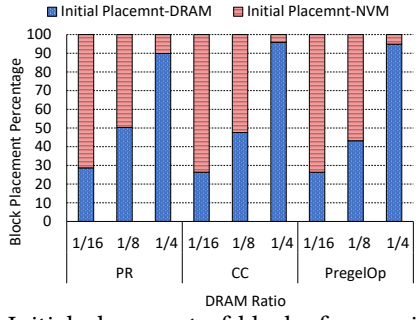

Figure 9. Initial placement of blocks for varying DRAM ratio.

We analyze the results for the flat hybridization approach to better understand the impact of our implemented techniques. Figure 9 shows the initial placement of newly computed blocks and Table 3 shows the percentage of transfer

in terms of the number of blocks using the AMP mechanism. Results show that synchronous transfer from DRAM to NVM is almost eliminated. Also, the percentage of blocks initially placed on the DRAM is much higher than the actual DRAM ratio. We observe that our dynamic unrolling technique was able to switch between allocation regions seamlessly to avoid triggering synchronous transfers. Since the synchronous transfer of blocks was the main bottleneck, by avoiding that we were able to achieve almost DRAM-like performance. Also, we note that placing newly computed blocks on the NVM does not cause a bottleneck and only influences the performance modestly.

Table 3. Percentage of blocks transferred using the AMP mechanism. NVM bandwidth is 1/8x of DRAM here.

Workload	1/16 DRAM	1/8 DRAM	1/4 DRAM
PR	96.5%	93.97%	99.87%
CC	97.58%	99.01%	100%
PregelOp	99.23%	99.43%	99.98%

6.4 Layered Hybridization with AMP

We also explore the performance impact of enhancing the layered approach with the AMP mechanism to see whether that makes it viable. The AMP mechanism should reduce the memory contention during unrolling in the layered hybridization by preemptively moving blocks. We use a comparatively higher NVM bandwidth of 1/4x to assess whether in a favorable configuration, the AMP mechanism can make the layered approach comparable to the flat one.

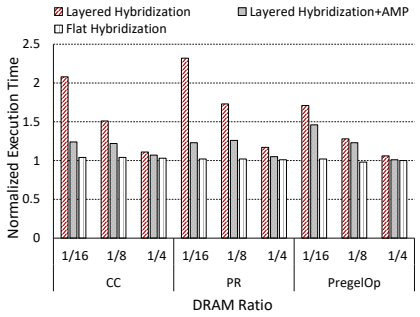

Figure 10. Performance of layered hybridization enhanced with AMP.

In Figure 10, we see that indeed the layered approach benefits from utilizing AMP. For the AMP-enhanced layered hybridization (Layered Hybridization + AMP), when DRAM is 1/4th of the total memory, the increase in execution time from the baseline DRAM-only system is at most 7%. In this case, the AMP-enhanced layered approach is very much comparable to the flat one, albeit a little worse. However, as we decrease the amount of DRAM in the system, we see degradation in the performance of this approach. The increase in

Table 4. Method of block transfer for the AMP-enhanced layered hybridization approach with *PageRank* workload. NVM bandwidth is 1/4x of DRAM here.

DRAM Ratio	Synchronous Transfer	Asynchronous Transfer
1/4	44.94%	55.06%
1/8	75.84%	24.16%
1/16	82.76%	17.24%

execution time for *PageRank (PR), ConnectedComponent (CC)* and *PregelOp* is as much as 26%, 24% and 46% respectively.

We examine the method of block transfer for the *PageRank* workload with AMP-enhanced layered hybridization. Table 4 shows, when the amount of DRAM shrinks, the percentage of blocks asynchronously transferred with the AMP mechanism decreases. This is expected as NVM bandwidth is lower than the DRAM, even a constantly working AMP mechanism fails to keep up with the speed of writing blocks in the DRAM.

From the results of Figure 10 and Table 4, it becomes evident that, even though the AMP-enhanced layered approach is comparable to the flat one in some cases, it is not as robust. Thus we reiterate that the flat hybridization approach is the most sensible option.

6.5 Impact of Opportunistic Eviction of Blocks to Disk

For testing the opportunistic disk eviction technique, we use the workload *PregelOp* with our flat hybridization approach. To quantify the impact of opportunistic eviction and the window multiplier, we focus on the flat hybridization approach and do not include other variations of implementation. We reduce the available memory for caching to 20% of the heap size to force the eviction of blocks to disk and set the NVM write bandwidth to be 1/4th of the DRAM. We vary the tunable window size multiplier parameter to see how smaller or bigger windows affect the performance of opportunistic disk eviction. We test for both the `MEMORY_AND_DISK` and `MEMORY_AND_DISK_SER` caching levels of RDD.

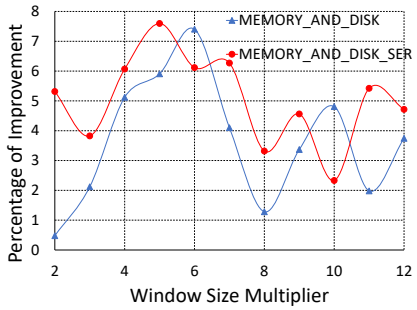

Figure 11. Performance Improvement trend for different window sizes.

From Figure 11, we see that with a suitable multiplier, a performance improvement of around 7.5% happens for both

types of caching levels. From the results, we can infer that, when the disk eviction window size is relatively small the overlap of I/O with computation is not that much; hence the improvement is lower. When we increase the window size, it gets better until the point where opportunistic disk eviction triggers too early and causes eviction of blocks that are again needed shortly, which somewhat diminishes the benefits. We see that our mechanism handles suboptimal multiplier size quite well, as there is no performance degradation. Depending on the memory capability, input dataset size, and workload characteristics the optimal window size will vary but as our mechanism is robust, we expect the user can use this to tune and improve the performance of the caching mechanism.

7 Related Work

There have been some research works that tackle different aspects of caching in Apache Spark and other DAG based cluster-computing frameworks. Least Reference Count (LRC) [27] analyzes the application DAG to evict the RDD blocks which have the least number of child blocks that are yet to be computed. LCS [4] or Least Cost Strategy evicts the blocks that will lead to minimum recovery cost in the future. Neutrino [25] proposes a fine-grained caching mechanism that allows serialized or deserialized caching for different blocks of the same RDD depending on the runtime characteristics of the cluster.

Clash of the Titans [18] explored the impact of storage levels and memory constraints for iterative workloads (e.g., PageRank). Zhang et al. [30] explored the impact of different types of storage devices and data abstractions for disk-based caching in Apache Spark and show that serialization cost dominates rather than disk I/O bandwidth and suppressing the excessive movement of blocks between disk and memory can lead to improved performance.

Quartz [21] is an emulator of persistent memory that utilizes DRAM thermal control to emulate bandwidth and uses hardware performance counters to calculate the required latency to be added after fixed time intervals to emulate latency of NVMs. HME [2] is another emulator for hybrid memory that exploits feature available in NUMA architectures to emulate NVM by injecting software added latency after memory accesses to remote NUMA nodes.

There have been efforts to utilize the byte-addressability, size, and bandwidth of NVM in other data-intensive frameworks. NVFS [8] is a proposed design of HDFS that implements HDFS I/O with memory semantics to exploit the byte-addressability of NVM and reduces DRAM memory contention by allocating buffers for RDMA-based communication from NVM. Rahman et al. [22] proposed to speed up the map-phase of RDMA-enabled Hadoop by spilling shuffle data to NVM SSDs instead of traditional disk storage.

Prominent research on hybrid memory design include works like DBUFF [15] and RaPP [17]. In DBUFF, the DRAM is used as a buffer and is not visible to the OS for allocation. The DRAM buffer along with a write queue is used to mitigate the write latency of PCM by enabling a lazy-write mechanism. RaPP organizes DRAM and PCM memory in a flat manner and migrates pages between them using a modified multi-queue (MQ) algorithm. RaPP ranks the memory pages depending on both access frequency and recency and dynamically places them depending on their popularity. These works are focused on the system level memory controller while our research focuses on a higher level caching mechanism i.e., caching in cluster computing framework.

8 Conclusion

In this paper, we have examined the key components of RDD caching used by Spark to support iterative in-memory data processing. We found that the performance of Spark RDD caching is not only affected by the capacity of available memory but also hindered by the suboptimal management of block allocation and eviction, due to its tightly coupled process of unrolling which combines computation and caching. Accordingly, we have explored two different strategies, layered and flat hybridization, to leverage emergent non-volatile memory devices for the Spark RDD cache. A simple layered hybridization could not address many drawbacks of the existing RDD cache in Spark. In contrast, flat hybridization can be designed to decouple the unrolling and block transfer decisions, and effectively leverage NVM devices for the RDD cache. Our experimental results demonstrate that flat hybridization can use DRAM as only a fraction of an expanded hybrid memory system and retain the degradation of execution time within a small percentage.

Acknowledgments

We thank the anonymous reviewers and our shepherd Yufei Ding for their valuable feedback. This work is supported in part by the National Science Foundation awards 1561041, 1564647, 1744336, 1763547, and 1822737. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.

References

[1] Gaurav Dhiman, Raid Ayoub, and Tajana Rosing. 2009. PDRAM: A Hybrid PRAM and DRAM Main Memory System. In *Proceedings of the 46th Annual Design Automation Conference (DAC '09)*. ACM, New York, NY, USA, 664–469. https://doi.org/10.1145/1629911.1630086

[2] Z. Duan, H. Liu, X. Liao, and H. Jin. 2018. HME: A lightweight emulator for hybrid memory. In *2018 Design, Automation Test in Europe Conference Exhibition (DATE)*. 1375–1380. https://doi.org/10.23919/DATE.2018.8342227

[3] Subramanya R. Dulloor, Amitabha Roy, Zheguang Zhao, Narayanan Sundaram, Nadathur Satish, Rajesh Sankaran, Jeff Jackson, and Karsten Schwan. 2016. Data Tiering in Heterogeneous Memory Systems. In

Proceedings of the Eleventh European Conference on Computer Systems (EuroSys '16). ACM, New York, NY, USA, Article 15, 16 pages. https://doi.org/10.1145/2901318.2901344

[4] Yuanzhen Geng, Xuanhua Shi, Cheng Pei, Hai Jin, and Wenbin Jiang. 2017. LCS: An Efficient Data Eviction Strategy for Spark. *International Journal of Parallel Programming* 45, 6 (01 Dec 2017), 1285–1297. https://doi.org/10.1007/s10766-016-0470-1

[5] M. Giardino, K. Doshi, and B. Ferri. 2016. Soft2LM: Application Guided Heterogeneous Memory Management. In *2016 IEEE International Conference on Networking, Architecture and Storage (NAS)*. 1–10. https://doi.org/10.1109/NAS.2016.7549421

[6] Hadoop [n.d.]. Apache Hadoop. https://hadoop.apache.org/.

[7] Jian Huang, Karsten Schwan, and Moinuddin K. Qureshi. 2014. NVRAM-aware Logging in Transaction Systems. *Proc. VLDB Endow.* 8, 4 (Dec. 2014), 389–400. https://doi.org/10.14778/2735496.2735502

[8] Nusrat Sharmin Islam, Md. Wasi-ur Rahman, Xiaoyi Lu, and Dhabaleswar K. Panda. 2016. High Performance Design for HDFS with Byte-Addressability of NVM and RDMA. In *Proceedings of the 2016 International Conference on Supercomputing (ICS '16)*. ACM, New York, NY, USA, Article 8, 14 pages. https://doi.org/10.1145/2925426.2926290

[9] Benjamin C. Lee, Engin Ipek, Onur Mutlu, and Doug Burger. 2009. Architecting Phase Change Memory As a Scalable Dram Alternative. In *Proceedings of the 36th Annual International Symposium on Computer Architecture (ISCA '09)*. ACM, New York, NY, USA, 2–13. https://doi.org/10.1145/1555754.1555758

[10] Haikun Liu, Yujie Chen, Xiaofei Liao, Hai Jin, Bingsheng He, Long Zheng, and Rentong Guo. 2017. Hardware/Software Cooperative Caching for Hybrid DRAM/NVM Memory Architectures. In *Proceedings of the International Conference on Supercomputing (ICS '17)*. ACM, New York, NY, USA, Article 26, 10 pages. https://doi.org/10.1145/3079079.3079089

[11] Justin Meza, Jichuan Chang, HanBin Yoon, Onur Mutlu, and Parthasarathy Ranganathan. 2012. Enabling Efficient and Scalable Hybrid Memories Using Fine-Granularity DRAM Cache Management. *IEEE Comput. Archit. Lett.* 11, 2 (July 2012), 61–64. https://doi.org/10.1109/L-CA.2012.2

[12] Bao Nguyen, Hua Tan, and Xuechen Zhang. 2017. Large-scale Adaptive Mesh Simulations Through Non-volatile Byte-addressable Memory. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC '17)*. ACM, New York, NY, USA, Article 27, 12 pages. https://doi.org/10.1145/3126908.3126944

[13] Kay Ousterhout, Ryan Rasti, Sylvia Ratnasamy, Scott Shenker, and Byung-Gon Chun. 2015. Making Sense of Performance in Data Analytics Frameworks. In *Proceedings of the 12th USENIX Conference on Networked Systems Design and Implementation (NSDI'15)*. USENIX Association, Berkeley, CA, USA, 293–307. http://dl.acm.org/citation.cfm?id=2789770.2789791

[14] H. Park, S. Yoo, and S. Lee. 2011. Power management of hybrid DRAM/PRAM-based main memory. In *2011 48th ACM/EDAC/IEEE Design Automation Conference (DAC)*. 59–64.

[15] Moinuddin K. Qureshi, Vijayalakshmi Srinivasan, and Jude A. Rivers. 2009. Scalable High Performance Main Memory System Using Phase-change Memory Technology. In *Proceedings of the 36th Annual International Symposium on Computer Architecture (ISCA '09)*. ACM, New York, NY, USA, 24–33. https://doi.org/10.1145/1555754.1555760

[16] Luiz E. Ramos, Eugene Gorbatov, and Ricardo Bianchini. 2011. Page Placement in Hybrid Memory Systems. In *Proceedings of the International Conference on Supercomputing (ICS '11)*. ACM, New York, NY, USA, 85–95. https://doi.org/10.1145/1995896.1995911

[17] Luiz E. Ramos, Eugene Gorbatov, and Ricardo Bianchini. 2011. Page Placement in Hybrid Memory Systems. In *Proceedings of the International Conference on Supercomputing (ICS '11)*. ACM, New York, NY, USA, 85–95. https://doi.org/10.1145/1995896.1995911

[18] Juwei Shi, Yunjie Qiu, Umar Farooq Minhas, Limei Jiao, Chen Wang, Berthold Reinwald, and Fatma Özcan. 2015. Clash of the Titans: MapReduce vs. Spark for Large Scale Data Analytics. *Proc. VLDB Endow.* 8, 13 (Sept. 2015), 2110–2121. https://doi.org/10.14778/2831360.2831365

[19] Avraham Shinnar, David Cunningham, Vijay Saraswat, and Benjamin Herta. 2012. M3R: Increased Performance for In-memory Hadoop Jobs. *Proc. VLDB Endow.* 5, 12 (Aug. 2012), 1736–1747. https://doi.org/10.14778/2367502.2367513

[20] SparkBench [n.d.]. Spark-Bench. https://github.com/SparkTC/spark-bench.

[21] Haris Volos, Guilherme Magalhaes, Ludmila Cherkasova, and Jun Li. 2015. Quartz: A Lightweight Performance Emulator for Persistent Memory Software. In *Proceedings of the 16th Annual Middleware Conference (Middleware '15)*. ACM, New York, NY, USA, 37–49. https://doi.org/10.1145/2814576.2814806

[22] Md. Wasi-ur Rahman, Nusrat Sharmin Islam, Xiaoyi Lu, and Dhabaleswar K. (DK) Panda. 2016. Can Non-volatile Memory Benefit Mapreduce Applications on HPC Clusters?. In *Proceedings of the 1st Joint International Workshop on Parallel Data Storage & Data Intensive Scalable Computing Systems (PDSW-DISCS '16)*. IEEE Press, Piscataway, NJ, USA, 19–24. https://doi.org/10.1109/PDSW-DISCS.2016.7

[23] Kai Wu, Yingchao Huang, and Dong Li. 2017. Unimem: Runtime Data Managementon Non-volatile Memory-based Heterogeneous Main Memory. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC '17)*. ACM, New York, NY, USA, Article 58, 14 pages. https://doi.org/10.1145/3126908.3126923

[24] Fei Xia, Dejun Jiang, Jin Xiong, and Ninghui Sun. 2017. HiKV: A Hybrid Index Key-Value Store for DRAM-NVM Memory Systems. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*. USENIX Association, Santa Clara, CA, 349–362. https://www.usenix.org/conference/atc17/technical-sessions/presentation/xia

[25] Erci Xu, Mohit Saxena, and Lawrence Chiu. 2016. Neutrino: Revisiting Memory Caching for Iterative Data Analytics. In *8th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 16)*. USENIX Association, Denver, CO. https://www.usenix.org/conference/hotstorage16/workshop-program/presentation/xu

[26] L. Xu, M. Li, L. Zhang, A. R. Butt, Y. Wang, and Z. Z. Hu. 2016. MEMTUNE: Dynamic Memory Management for In-Memory Data Analytic Platforms. In *2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. 383–392. https://doi.org/10.1109/IPDPS.2016.105

[27] Yinghao Yu, Wei Wang, Jun Zhang, and Khaled Ben Letaief. 2017. LRC: Dependency-aware cache management for data analytics clusters. *IEEE INFOCOM 2017 - IEEE Conference on Computer Communications* (2017), 1–9.

[28] Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, and Ion Stoica. 2010. Spark: Cluster Computing with Working Sets. In *Proceedings of the 2Nd USENIX Conference on Hot Topics in Cloud Computing (HotCloud'10)*. USENIX Association, Berkeley, CA, USA, 10–10. http://dl.acm.org/citation.cfm?id=1863103.1863113

[29] Haoyu Zhang, Brian Cho, Ergin Seyfe, Avery Ching, and Michael J. Freedman. 2018. Riffle: Optimized Shuffle Service for Large-scale Data Analytics. In *Proceedings of the Thirteenth EuroSys Conference (EuroSys '18)*. ACM, New York, NY, USA, Article 43, 15 pages. https://doi.org/10.1145/3190508.3190534

[30] K. Zhang, Y. Tanimura, H. Nakada, and H. Ogawa. 2017. Understanding and improving disk-based intermediate data caching in Spark. In *2017 IEEE International Conference on Big Data (Big Data)*. 2508–2517. https://doi.org/10.1109/BigData.2017.8258209

[31] W. Zhang and T. Li. 2009. Exploring Phase Change Memory and 3D Die-Stacking for Power/Thermal Friendly, Fast and Durable Memory Architectures. In *2009 18th International Conference on Parallel Architectures and Compilation Techniques*. 101–112. https://doi.org/10.1109/PACT.2009.30