Early Experience with Optimizing I/O Performance Using High-Performance SSDs for In-Memory Cluster Computing

I. Stephen Choi
Memory Solutions Lab.
Samsung Semiconductor Inc.
San Jose, CA
stephen.ch@samsung.com

Weiqing Yang
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA
weiqingy@andrew.cmu.edu

Yang-Suk Kee
Memory Solutions Lab.
Samsung Semiconductor Inc.
San Jose, CA
yangseok.ki@samsung.com

***Abstract*—This paper describes our experience with storage optimization that utilizes cost-effective PCIe solid-state drives (SSDs) to improve the overall performance of a Spark framework. A key problem we address is the limited memory system performance. In particular, we adopt high-performance SSDs to alleviate the saturated DRAM bandwidth and its limited capacity. We utilize SSDs to store shuffle data and persisted RDDs. As a result, the overall performance improves due to the larger capacity of SSDs and the increased bandwidth provided by SSDs while alleviating memory contentions. Our experiments show that we can improve the performance of data-intensive applications by 23.1% on average, compared to the performance of the memory-only approach. To our knowledge, this is the first work to demonstrate performance optimizations using PCIe SSDs on Spark.**

***Keywords*-SSD; Spark; I/O performance; Memory bandwidth; Storage**

I. Introduction

This is an era of Big Data. We generate exa-/zetta-byte of data and such data is driving radical changes in compute-centric platforms. To cope with the ever-increasing data, cluster computing is widespread and data-intensive frameworks, e.g., MapReduce and Dryad, are proposed and largely deployed in data-centers. Such frameworks allow users to perform data mining, machine learning and sophisticated data analytics, seamlessly scaling up to thousands of cluster nodes. However, such frameworks commonly utilize file-based data communications hence generate heavy I/Os. As a result, the performance of distributed file systems, such as HDFS, becomes a bottleneck to achieve high performance oftentimes. To solve this problem, there have been many studies to improve the performance of distributed file systems [21], [12], [13], [14], [22].

While high-performance distributed file systems can improve the overall performance of data-intensive frameworks, making the best use of commodity servers is still an open question. In fact, most of the time data-intensive frameworks under-utilize abundant memories in the clusters [2]. Utilizing such memories by storing intermediate data inside and reusing it can improve the performance significantly. To implement such in-memory data caching, Spark was recently introduced and achieves much higher performance than the performance of Hadoop [10] by placing data inside memory with an awareness of data locality [27].

Although Spark achieves higher performance than that of Hadoop, limited bandwidth and capacity of memory are limiting factors toward the realization of *memory-speed* computation. This is because core scaling continues while memory bandwidth does not, due to the pin count limitation per chip [1]. Moreover, because iterative workloads keep generating intermediate data that is larger than the input data, storing it inside memory is practically impossible.

Meanwhile, modern high-performance solid-state drives (SSDs) provide substantial bandwidth and larger capacity compared to those of DRAM memories. For example, latest cost-effective PCIe SSDs provide up to 20% of the bandwidth that single DDR3 DIMM can provide. The measured latency of PCIe SSDs to access files is also only 2.5 times slower than the latency of DRAM despite of the greater differences in the raw latencies. Moreover, its unit cost per capacity is more than 15 times cheaper than that of memory. As a result, SSDs can be used to store intermediate data while alleviating the memory bandwidth and capacity requirements. As such, utilizing SSDs will result in performance improvements. Therefore, it is crucial to understand how to alleviate the bandwidth and capacity requirements of in-memory cluster computing by coordinating the memory and high performance SSDs.

In this paper, we optimize Spark's RDD caching and shuffling mechanism by utilizing cost effective, high performance PCIe SSDs. We compare different caching/shuffling schemes to optimize its performance with representative data-intensive applications.

The key contributions of this paper include followings:

- We study big-data analytics from the computer architecture perspective and show that memory bandwidth and capacity requirements are main bottlenecks in clusters with commodity servers
- We enhance the current Spark implementation to coordinate DRAM and high performance SSDs. In particular, we separate caching and shuffling locations

978-1-4799-9926-2/15/$31.00 ©2015 IEEE

- We evaluate energy and cost efficiencies as well as performance improvements of different caching/shuffling configurations to coordinate high performance SSDs and memory. We can improve the performance by 23.1% on average. We also show that the energy efficiency can be improved 12% and 22.9% for the cost efficiency by utilizing high performance SSDs to store persisted RDDs along with shuffle data compared to memory-only approach.

II. Background and Motivation

A. Big Data and I/O

The coordination of software tools and architectures was common in high-performance computing, e.g. MPI or OpenMP and specially designed clusters or large-scale shared-memory machines. However, in the cluster computing for data-intensive applications, programming and execution frameworks provide a single view of data while orchestrating commodity servers. Virtually all of these frameworks heavily rely on data parallelism and have been built for large-scale parallel executions [3]. Some of widely used frameworks are MapReduce [8] and Dryad [11]. Such frameworks come with important commonalities in the granularity of their parallelism. Each *job*, computational work, can be submitted to a framework and a job is defined as a directed acyclic graph (DAG) of *phases*, where each phase consists of many fine-grain *tasks* utilizing data parallelism. The input and output of each task consists of one or more blocks of file and tasks of a phase have no dependencies among them.

Hadoop, an open source implementation of MapReduce for example, divides input data into many chunks and each worker executes jobs on the split for parallelization. This approach is particularly effective not only for exploiting the cost-effectiveness of commodity servers, but for parallel execution of distributed data to exploit its data locality. However, this involves explicit data movements via files for its communications, utilizing distributed file systems oftentimes.

One of the most important characteristics from the architectural perspective is found here: communications between tasks and data synchronizations are performed by utilizing file systems. That is, many parallel tasks read and write data for communications between them. Moreover, data should be committed to storage for its synchronization. As a result, MapReduce-like frameworks introduce a new challenge of heavy I/Os that might cause I/O bottlenecks in commodity servers despite of its cost-effectiveness.

B. Spark Framework

While such cluster computing is pervasive in large-scale data processing, memory capacities in cluster nodes are underutilized [2]. To address this problem, Spark was recently introduced. Spark is compatible with Hadoop while providing up to 100X speedup compared to Hadoop [27].

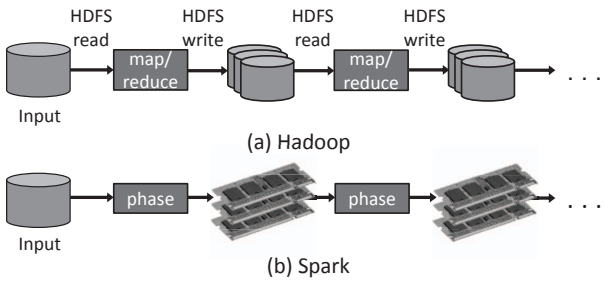

Figure 1. Comparison between Apache Hadoop and Spark with conceptual work-flows. Spark reduces HDFS read/write latencies by maintaining data in memory for future references.

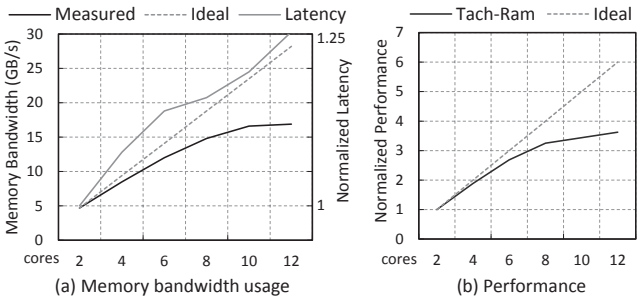

Figure 2. Performance scaling by adding more cores per data node with PageRank and measured memory bandwidth usage per node.

The key improvement of Spark from Hadoop is the better utilization of memory with its programming model and execution environment.

Spark makes the use of memory locality to speed-up data-intensive jobs with its DAG engine to support data flow while storing intermediate data between tasks in memory. On the other hand, Hadoop has to store the intermediate data between different map/reduce phases (known as tasks in Hadoop) on HDFS. The conceptual differences are shown in Figure 1. For example, there is an I/O intensive phase where it processes the input data, e.g. *map* in Hadoop. During this phase at every task, reading the raw or intermediate data happens and it can easily take significant portion of the total execution time (similarly in *reduce* for writing data). By eliminating these phases by utilizing memory, Spark achieves significant performance improvements especially for iterative computations which can easily exploit the data reuse.

C. Bandwidth and Capacity Limits

Although the better memory utilization of Spark achieves substantial performance improvements, Spark exacerbates the current memory-wall problem [26]; the memory performance will restrain the overall performance. In particular, inter-thread interferences in memory system due to bank/bus/row-buffer conflicts are well known [18]. Figure 2-(a) shows memory bandwidth usage by adding more cores

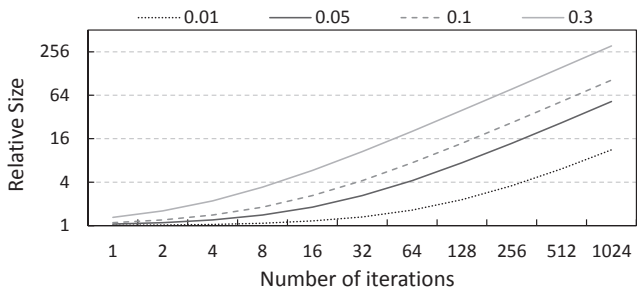

Figure 3. The growth of accumulated intermediate data size with iterative applications with different ratios by comparing the size of intermediate data to the size of original input data.

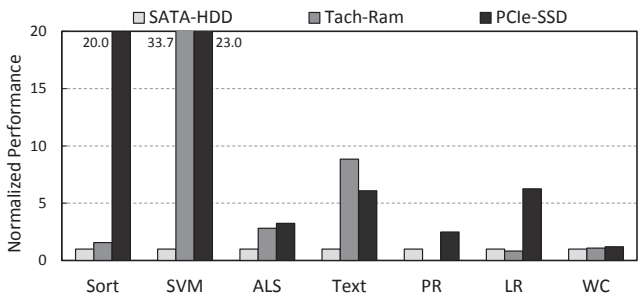

Figure 4. Performance gain with data-intensive applications on Spark framework by using higher-bandwidth storage devices.

	DDR-3 DRAM	PCIe SSD
Latency (4KB random read, μs)	78 (ramfs)	196 (EXT4)
Bandwidth (GB/s)	5.8	1.1
Capacity per dollar (MB/$)	67	1150

Table I
MEMORY VS. HIGH-PERFORMANCE SSD.

on our Spark clusters while running PageRank [6]. We use Tachyon for RDD caching and *ramfs* for RDD shuffling in this experiment (details will be discussed in Section IV). We measure the memory bandwidth usages and latencies with different number of active cores by varying the number of cores from 2 to 12 per data node. As shown, the bandwidth usage does not scale well and the latency increases as adding more cores. In particular, 12-core configuration achieves only 60% of the bandwidth with ideal scaling and shows 25% longer latency. These results indicate that a limited memory-system performance restrains the overall performance. As a result, the overall performance degrades as shown in Figure 2-(b). The 12-core configuration achieves only 60% of the performance of ideal scaling. Therefore, alleviating the limited memory performance is a major challenge to achieve high performance.

On the other hand, such approach–storing RDDs and shuffle data inside memory– will encounter capacity limit, too. As shown in Figure 3, iterative applications generate large amount of accumulated intermediate data while iterating. For example, an application might generate intermediate data that is 300 times bigger than the size of its original input data. Despite of the large aggregated memory capacities of clusters, maintaining increasing intermediate data inside memory is practically impossible.

D. Resolving Limits with High-Performance SSDs

To alleviate such limited memory performance and satisfy large capacity requirements, we consider high performance SSDs. Due to the great performance improvements recently, SSDs are bridging the performance gap between memory and HDDs. Note that SSDs' superior performance, energy efficiency, and energy proportionality [23] over HDDs have catalyzed broad and fast adoption. Although SSDs are slower than DRAM in the latencies and bandwidths as shown in Table I, its much larger capacity per unit cost is attractive. Having said that, latest PCIe SSD's bandwidth is almost 20% of the single-channel DRAM bandwidth (see Section IV for details). Despite of the difference in the raw latencies by 3 orders of magnitude in general, the measured latency throughout a file system shows only 2.5 times slower latencies in the PCIe SSD to access 4KB data. Moreover, its capacity per dollar is more than 15X of DRAM's. These relatively high bandwidth and larger capacity per dollar make high-performance SSDs as supplements for a higher bandwidth and a larger capacity, which are the current keys of in-memory cluster computing to achieve high performance.

In fact, storing RDDs and shuffle data in storage devices can outperform the way that stores these inside memory. Although we cannot avoid inter-thread interferences in the memory system on chip multiprocessors (CMPs), we can at least reduce interferences caused by exploiting the memory system both for execution and storage purposes. Figure 4 shows a limit study result of performance gain on Spark clusters by utilizing high performance SSDs to store RDDs and shuffle data. We evaluate the performance of 7 workloads with SATA HDD and PCIe SSD and performances are normalized to the performance of SATA HDD. We assume clusters that experience poor page cache hit rates (e.g. data nodes commonly serve HDFS and/or other distributed DB causing page-cache interferences). For example, Sort, ALS, LR, and WC, PCIe SSD outperforms the memory-only approach, Tach-Ram, and this shows great potential performance improvements by alleviating the problem of the limited memory performance. It is notable that clusters with PCIe SSDs can improve the overall performance of Sort as much as 13X of the performance of the clusters with Tach-Ram. Moreover, we were not able to run PR with memory-only due to the limited memory capacity (details will be discussed in Section IV and V).

III. I/O Optimization on Spark

As shown, Spark introduces a nontrivial challenge of optimizing I/O performance due to bandwidth and capacity requirements. We will optimize the overall performance of Spark utilizing high performance SSDs throughout this paper. In this section, we explain our approach to optimize performances on Spark. We first explain Spark's data management mechanisms with an example of PageRank.

A. Spark RDD

Spark provides a programming and execution model via *resilient distributed datasets* (RDD). RDDs provide general programming models because it can be created by transforming data and processed by using data flow operators. One of the unique characteristics of RDD is that it is immutable. Once created, RDDs will not be updated thereafter, hence there is no concern for data coherence. As a result, caching data with RDDs locally can be simplified and each node can maintain its own cache without considering coherency problems.

B. Spark Caching Mechanism

To achieve high performance, all data should be kept in memory for fast data retrievals. However, caching data inside JVM heap memory space is challenging due to the Java memory management. Memory allocation on a Java virtual machine (JVM) is handled automatically and garbage collector (GC) takes care of it. Since Java GC normally adopts generational caching mechanism, it is hard to maintain data, which will be used in future, in memory due to being flushed out by GC. To overcome this mismatch between the programming/execution model and the caching mechanism in Java, Spark adds caching semantics with *persist()*. RDDs can be persisted in JVM heap memory by passing *StorageLevel* object with *MEMORY_ONLY*.

Unfortunately, persisting RDDs in memory causes significant overhead in GC time. Keeping persisted data in JVM heap space is costly because Java GC marks live data rather than marking dead data. Thus the cost of GC gets more expensive as the size of live dataset increases. We will discuss the increased GC time in Section IV.

Tachyon is introduced to overcome this GC overhead when live data is kept inside Java heap space. Tachyon is a memory-centric distributed file system [16]. Along with other advantages, it reduces GC overhead significantly from the performance perspective despite its (de)serialization overhead. We will discuss such GC overheads with experimental results in Section V by comparing the results of this alternative mechanism, Tachyon.

C. Spark Data Shuffling

Spark keeps data distributed across cluster nodes while data being partitioned. During the execution, each partition of data will be fed into corresponding worker nodes. Because RDDs are immutable, all data write and modification is considered as *transformation* and it is directed by DAG. Given that, the input data will be transformed to new RDDs as a job progresses while staying in the same partition as much as possible to avoid data shuffling across machines.

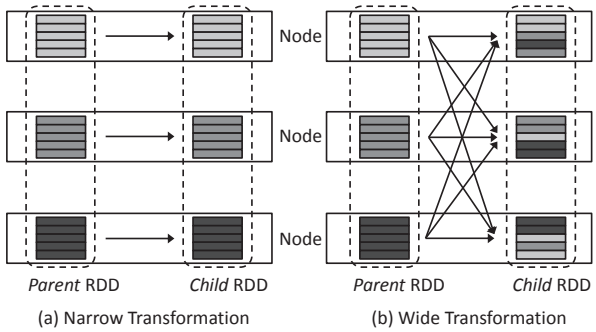

Figure 5. Data shuffling across machines. There are narrow/wide transformation of RDDs and only wide transformation requires data shuffling.

Spark has two types of such data transformation as shown in Figure 5. *Narrow transformation* means the all partitions of a parent RDD will be consumed by the same node and form a new child RDD and thus, requiring no data shuffling across machines. Examples include *Map, FlatMap, Filter*, and *Sample*. On the other hand, *wide transformation* requires data shuffling across machines because a parent RDD will be split into smaller elements and fed into remote nodes so that these forms new partitions in a child RDD as shown in Figure 5-(b). Wide transformation includes *SortByKey, ReduceByKey, GroupByKey*, and *Join*.

Data shuffling is another important parameter to tune as well as RDD caching because it involves lots of I/Os. We will discuss performance and energy efficiency implications in Section V by varying the shuffling/caching location utilizing both memory and SSDs.

D. RDDs in PageRank

PageRank is an algorithm used by Google Search [6] to rank websites and commonly used as benchmark application. Its Scala implementation for Spark is shown in List. 1. For the initialization, following Spark operators are used:

- distinct(): return a new RDD containing the distinct elements in this RDD
- groupByKey(): return a RDD grouped by the same keys
- cache(): persist this RDD with the default storage level
- mapValues(): pass each value in the key-value pair RDD

For the main computation part, it uses a *for-loop* to iterate.

- join(): when called on datasets, returns a dataset of merged Key-Value based on all pairs of elements for each key.

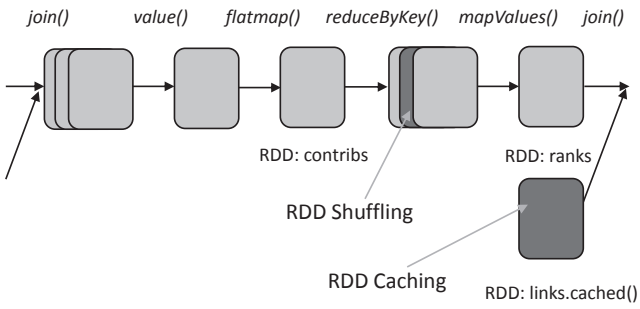

Figure 6. Work-flow in Scala implementation of PageRank in Spark. There are RDD shuffling and RDD caching, involved in *reduceByKey()* and *join()*, respectively.

- values(): extracts the values from all contained tuples and returns in a new RDD
- flatMap(*func*): return a new RDD by passing each element of the source through a function described in the *func*
- reduceByKeys(*func*): returns a dataset of Key-Value pairs where the values for each key are aggregated using the given function *func*.
- mapValues(*func*): passes each value in the Key-Value pair RDD through a map function.

Figure 6 illustrates RDD transformations and there are one wide transformation, *reduceByKey()*, thus resulting in RDD shuffling across machines and one narrow transformation which involves RDD caching. From the memory and storage perspectives, these two RDDs are critical because it generates remote data accesses and local data accesses due to RDD shuffling and RDD caching, respectively.

Listing 1. PageRank implementation in Scala.

```
/** Initialization
  */
val lines = ctx.textFile(args(0), 1)
val links = lines.map{ s =>
  val parts = s.split("\\s+")
  (parts(0), parts(1))
/** RDD caching
  */
}.distinct().groupByKey().cache()
var ranks = links.mapValues(v => 1.0)

/** Main iteration
  */
for (i <- 1 to iters) {
  val contribs = links.join(ranks).values.flatMap{
    case (urls, rank) =>
    val size = urls.size
    urls.map(url => (url, rank / size))
  }
  ranks = contribs.reduceByKey(_ + _).mapValues \\
    (0.15 + 0.85 * _)
}

val output = ranks.collect()
```

Nodes	node#0 runs a namenode node#1-#8 run datanodes
CPU	two Intel Xeon E5-2630 2.3 GHz 6 cores per chip and 12 cores in total per node
Memory	64 GB DRAM (DDR3-1600 with ECC)
Storage	Dell SSD (256 GB, MLC, SATA) Samsung 840 Pro SSD (512 GB, MLC, SATA) Samsung XP 941 SSD (512 GB, MLC, PCIe/AHCI)
Software	Ubuntu 12.04 desktop OpenJDK 1.6.0_27, 64bit Spark 1.0.1 Tachyon 0.4.1 Hadoop 1.0.4 for HDFS

Table II
CLUSTER CONFIGURATIONS.

IV. EXPERIMENTAL METHODOLOGY

A. Experimental Environment

1) Cluster Configuration: The commodity cluster system we use in this work consists of 8 data nodes and one name node. Each node is a Dell Precision T7600 with two 2.3 GHz Xeon E5-2630 hexa-core processor (12 cores in total). Hyper-threading and Intel Turbo Boost are disabled for consistent results. Each node comprises 64 GB of memory, a 250 GB Dell SATA SSD, a 512 GB Samsung SATA SSD, and 512 GB PCIe SSD. The Dell SSD is used by the OS. Two HDDs are used for HDFS and another HDD is used for the RDD caching device. All nodes are connected via 10 Gigabit Ethernet using Intel X520 NICs so that the network bandwidth does not limit the overall performance. Table II summarizes these cluster configurations.

2) Spark Framework: We use Ubuntu 12.04 LTS, OpenJDK 1.6.0_27, Spark 1.0.1, Tachyon 0.4.1, and Hadoop 1.0.4 for HDFS. In this study, we first tune and optimize Spark for good performance of the baseline (detailed information will be described in the following section). We use Tachyon [16] to evaluate in-memory caching and shuffling while keep GC time low by avoiding placing data in JVM heap space which generates substantial GC times. Finally, we use a JVM instance per core to reduce multi-Java threads overheads.

3) Workloads and Datasets: We use 7 representative Spark applications to evaluate storage bandwidth limits. We use ported TeraSort – originally written for Hadoop – (Sort), Support Vector Machine (SVM), Alternating Least Squares (ALS), in-memory text search (Text), PageRank (PR), Logistic Regression (LR), and word count (WC). We use Spark MLlib [20] implementations of SVM and ALS. Other than these, we use Spark's native functions to implement algorithms.

We generate synthetic input data of 100GB for TeraSort using TeraGen. We use *mnist8m* dataset, size of 12GB and 8.1 million data of 784 features [7], for SVM. We use Amazon movie reviews of 100GB, generated from the model of 8 million reviews [24], for ALS and Text. We use Twitter's social graph dataset of 22GB size consists of 41.7

Component	Description	Unit Price ($)
CPU	Intel Xeon E5-2630 2.3 GHz	605
Motherboard	Intel Workstation Board W2600CR	625
Case	Intel SC5650DPNA 600W	248
NIC	Intel X520-DA2	341
Memory	Samsung DDR3-1600 4GB ECC Registered Dual Rank CL11	60
Storage	WD Black 2TB HDD SATA	125
	Samsung 840 PRO SSD 512 GB SATA	270
	Samsung XP 941 SSD 512 GB PCIe	445

Table III
PRICES OF CLUSTER COMPONENTS.

million users profiles, 1.47 billion social relations [15] for PageRank and WC. Lastly, we generate 128GB synthetic input for LR consists of random numbers.

4) Page Cache Flush and Shuffle Data Sync: For a limit study purpose, we assume poor page caching environment by running multiple services on data nodes. For this purpose, we drop the page cache every second. Next, we set *spark.shuffle.sync* to represent common configurations in Spark. In this experiment, all shuffle data should be written to storage devices to proceed. This option is used to provide better reliability by guaranteeing the shuffle data after successful task completion. So, a node fails after completing certain task, the shuffle data after the completion should be persisted in the storage. Without this option, the shuffle data could be lost if power failure happens before synchronizing page caches down to the storage.

5) Storage Bandwidth: We configure Spark so that RDD caching and shuffling data is written to a storage device. Spark provides an API for this and we use StorageLevel.DISK_ONLY for RDD caching in a storage device. The measured storage bandwidths of SATA HDD and PCIe SSD are around 162/100 MB/s and 1204/1096 MB/s (r/w), respectively (we use *Linux fio* with 128K block size, 32 IO depth, and 12 concurrent jobs considering number of cores per node). We also use the *fio* to measure the average latency shown in Table I.

6) Memory Bandwidth: We utilize Tachyon and *ramfs* to place RDDs and shuffle data, respectively, in memory at the same time. Note that the number of job iteration is limited due to the increasing size of shuffle data while the physical memory capacity is constant. We control the parameters not to cause page swap happening for the baseline configuration. We use performance counters of CPUs to measure memory bandwidth [9]. The reported numbers in Figure 2 are the average number of gathered information across the data nodes.

7) Power Consumption: The power consumption of Spark clusters is measured with a Watts Up Net device. This power meter is connected between AC and the power supply of each cluster node. We randomly select a data node and measure, then we report the average power consumptions.

8) Node Component Cost: We use SSDs for both RDD caching and shuffling locations. Such SSDs are dedicated for either caching/shuffling or both only. Currently high performance or enterprise-level PCIe SSDs are relatively more expensive than consumer market SSDs, *i.e.* Samsung 840 Pro 512 GB SSD costs around 270$ while XS 1715 400 GB SSD costs as high as 2098$. To cope with this premium price of enterprise SSDs, we use more cost effective XP 941 SSDs which still provides relatively high performance. Table III shows unit prices of memory and SSDs used in this work.

B. Tuning Spark Framework

In order to evaluate the performance and energy efficiency implications of memory and storage devices, we need a reasonably tuned Spark platform for fair comparisons. For example, SSDs are pervasive in these days thus having a Spark platform using HDDs for the local storage is neither representative nor fair. In this paper, our baseline utilizes sata SSDs for local storages. We first tune and optimize Spark for good performance in the baseline. Such tuning and optimization include JVM tuning, specifically for a garbage collection type and a number of JVM instances, and memory caching scheme. Although configuring JVM to achieve the best performance is challenging and non-trivial process, we address this problem by comparing few possible configurations to get reasonable baseline in the performance evaluations in Section V.

1) Java Garbage Collection: First of all, we use *parallel old GC* with a -XX:UseParallelOldGC flag. Comparing the default GC type, -XX:UseParallelGC which employs parallel GC for young generation and serial GC for old, *parallel old GC* adopts parallel GC for old generation. The reason behind selecting parallel GC is that old generation takes large portion of heap space as JVM caches RDDs. For example, we observed eden space (EC) of 1.3GB comprising survivor space 0 capacity (S0C) of 590MB and survivor space 1 capacity (S1C) of 570MB and old space capacity (OC) of 4.9GB. Under such heap space allocation, young generation GC time (YGCT) took 132ms and full GC time (FGCT) 4s with the default GC type. With *parallel old GC*, YGCT and FGCT took 161ms and 0.9s, respectively. The overall performance of the PageRank was improved up to 2X (PageRank stage 12 took 3 min while taking 6.3 min previously).

2) RDD Caching: Second, we use Tachyon for RDD caching in the memory. We compare the performance of caching inside JVM heap space and off-heap (Tachyon). Figure 7 summaries the performance comparison. This figure shows average execution time of PageRank iterations by varying an effective JVM heap space, *i.e.* excluding JVM heap space used for RDD caching from the total JVM heap space so that we can compare to off-heap. We run the PageRank with 3G, 7G and 15G effective JVM heap space.

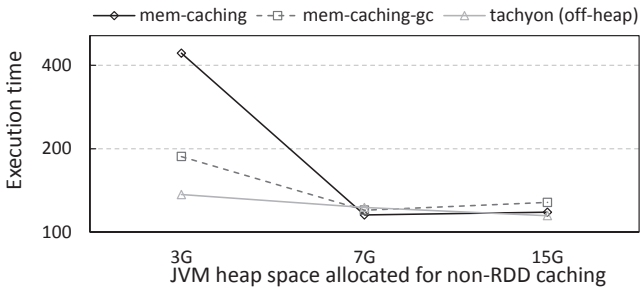

Figure 7. Execution time with memory caching, memory caching with parallel GC and off-heap caching.

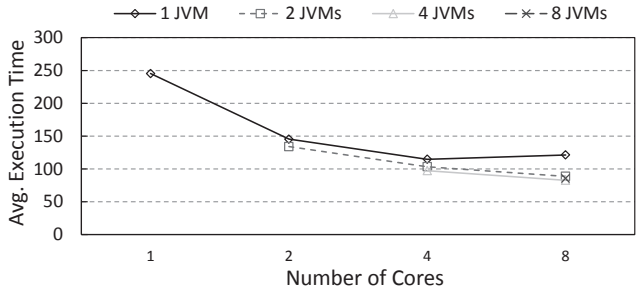

Figure 8. Execution time with 1, 2, 4, and 8 JVM instances.

In addition, we also consider the GC type as well. This comparison was executed in a four-node cluster utilizing 8 cores running in a single JVM instance sharing the memory space. As shown in the figure, off-heap shows consistent performance across different heap capacities. Meanwhile, *mem-caching-gc*, having parallel old GC, shows best performance at 7G, but exhibits degraded performance both at 3G and 15G JVM heap capacities. Moreover, *mem-caching* shows severe performance degradation at 3G, 2-3X slower execution compared to *mem-caching-gc* and *off-heap*, and thus being eliminated from further considerations. As such, we use Tachyon as a default in-memory RDD caching scheme in the rest of this paper.

3) Number of JVM Instances: Lastly, we compare the performances by varying the number of JVM instances. Running multiple Java threads on the same JVM can cause non-trivial implication in the performance. For example, it may increase GC overhead or data interference inside the JVM heap space. We vary the number of threads per JVM by adding active cores. For example, there are four different configurations for 8 cores: one JVM running 8 threads, two JVMs running four threads each, four JVMs running two threads each, and 8 JVMs running single thread each. As Figure 8 shows, deploying separated JVM instances generally result in better performance. Single JVM sharing multiple Java thread shows decreasing performances as number of cores (tasks) increases. Note that we keep the total capacity of JVM heap space as same, *i.e.* (*2 JVMs*) has the half of the (*1 JVM*)'s JVM heap space per instance, resulting in the same total capacity.

C. Coordinating Memory and SSD

For RDD caching, we change the Spark configuration file along with source codes of our workloads so that RDD can be cached either in memory or storage devices. RDD being cached in memory, we use Tachyon as a caching layer on top of the existing HDFS. We specify a RDD persistence type with *OFF_HEAP* parameter, *i.e. rdd.persist(StorageLevel.OFF_HEAP)*. To utilize SSDs for caching location, we specify the type with *DISK_ONLY* and direct the corresponding mounting point in the Spark configuration file.

For RDD shuffling, we modify Spark source code slightly to separate its location from the caching location. By default, Spark uses a unified location for both caching and shuffling. In the rest of this paper, we allocated RDD caching and shuffling locations independently utilizing our patch to Spark while the changes are minimal and thus negligible performance implications due to the changes. To store shuffle RDDs in memory, we exploit *ramfs* to utilize memory space as a virtual storage device. Although *tmpfs* might be more appropriate candidate for RDD shuffling for *ramfs* being vulnerable for memory overflow, we use *ramfs* for the simplicity due to the fact that Tachyon is using it with a careful consideration of memory usage.

In addition, we focus on and monitor compute iterations excluding the very first compute phase causing cold cache misses in the RDD caching. Many Spark workloads consist of iterative computation and thus, repeating compute phases over and over. So observing the compute phases are more representative than observing all phases of the workloads.

V. Experimental Results

A. Performance

1) Limit Study with Page Cache Flush: We compare performance improvements by changing locations for persisted RDDs and shuffle data. In particular, we assume when page cache performs poorly. Figure 9-(a) shows the results and our baseline is the *SSD-all*, RDD caching and shuffling data in a dedicated SATA SSD.

HPSSD-all, storing shuffle data and persisted RDDs in high performance SSDs, shows the best performance in Sort, ALS, and LR with performance improvements of 1.81X, 1.27X and 1.09X, respectively. Moreover, across all workloads, it exhibits relatively high performance gains. That is because utilizing SSDs do not experience performance degradations caused by frequent page swapping found in some Tachyon usages. As a result, HPSSD-all achieves the best average performance gain, 1.34X. It is also notable that HPSSD-all shows 1.20X performance gain over Tach-Ram.

Tach-HPSSD, utilizing memory for persisted RDDs and high-performance SSDs for shuffling data, exhibits the best

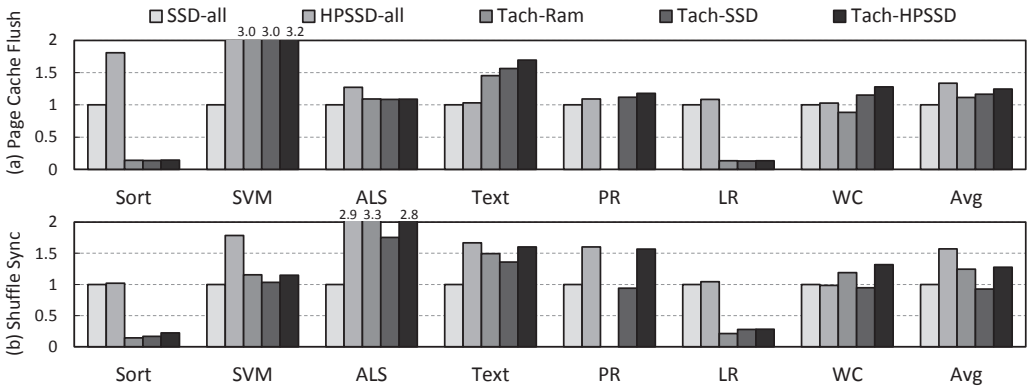

Figure 9. Normalized performances with different caching/shuffling schemes.

performance gains in SVM, Text, PR, and WC, with the gains of 3.19X, 1.70X, 1.18X, 1.28X, respectively. Most probably, Tach-HPSSD relaxes memory contentions by utilizing high performance SSDs while providing additional bandwidth of it. Thus, it achieves the best performance. Moreover, across all workloads, Tach-HPSSD shows better performance than Tach-Ram implicating that alleviating the limited memory system performance is always helpful.

On the other hand, Tach-Ram, utilizing memory for both shuffle data and persisted RDDs, did not show the best performance gain in any workloads. Moreover, in Sort and LR, frequent page swapping results in severe performance degradations, up to 7 times slower execution. This problem is caused by not enough physical memory space and PR fails to run for the same reason with Spark *timeout* error.

Using dedicated memory space shows performance degradation regardless of the purposes. On top of the Sort, PR, and LR that we discussed, we had to check the working capacities for Tachyon and *ramfs* to finish the runs. This is not only significant efforts, but also a key blocker towards seamless deployments of applications using scalable clusters.

2) Shuffle Data Sync: HPSSD-all shows the best performances in 5 workloads including Sort, SVM, Text, PR, and LR with performance gains of 1.02X, 1.78X, 1.67X, 1.60X, and 1.04X, respectively. In overall, HPSSD-all exhibits 1.57X performance gain compared to SSD-all which is larger than the gain in the previous experiments, 1.34X with flushing page caches. This is because blocking IOs for shuffle-data writes get more benefits from PCIe SSDs that have twice better write performances compared to SATA SSDs. Lastly, HPSSD-all shows the performance gains of 1.26X over Tach-Ram.

For Tach-HPSSD and Tach-Ram, they show mixed performance gains; Tach-HPSSD wins over Tach-Ram in Sort, SVM, LR, and WC, and vice versa in ALS and Text. As a result, 1.28X and 1.24X performance gains were achieved for Tach-HPSSD and Tach-Ram, respectively. However, it is notable that Tach-Ram fails to run in PR same to previous experiments.

Configuration	Total Cost ($)
SSD-all/Tach-SSD	4820
HPSSD-all/Tach-HP-SSD	4995
Tach-Ram	4550

Table IV
SYSTEM TOTAL COST.

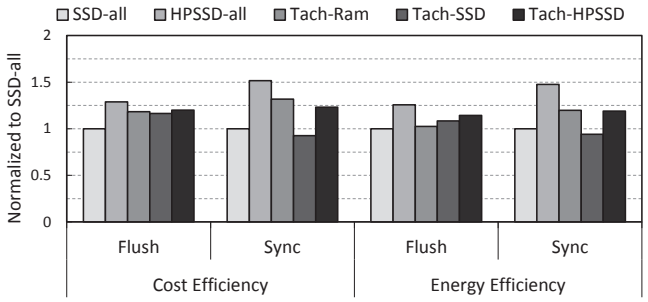

Figure 10. Average cost efficiency (Perf/$) and energy efficiency (Perf/Watt).

For Tach-SSD, it exhibits lower performance gains than Tach-Ram in SVM, ALS, Text, and WC. Moreover, severe performance degradation happens in Sort and LR due to frequent page swapping. As a result, Tach-SSD shows poorer performance than SSD-all on average with 0.92X.

Unlike the previous experiments, neither SATA SSDs nor PCIe SSDs along with Tachyon shows significant performance improvements compared to Tach-Ram. In fact, Tach-SSD is slower than SSD-all on average.

B. Cost and Energy Efficiency

1) Cost Efficiency: We evaluate cost efficiencies of different configurations to understand the cost effectiveness of adopting high performance SSDs to Spark clusters. When we adopt a cost-effective PCIe SSD which was used in our experiments, the total cost of each data node increases 445$

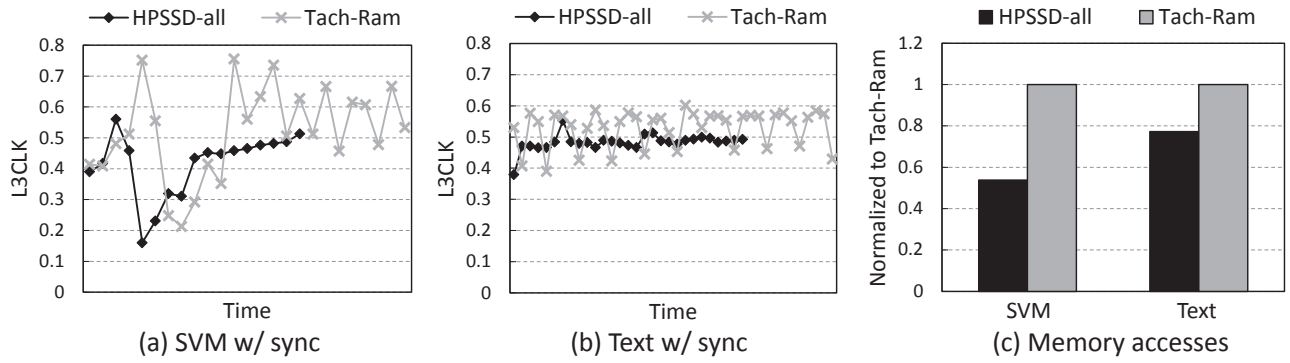

Figure 11. Ratio of CPU cycles lost due to memory accesses.

or 9.8% as shown in Table IV. As a result, the overall cost efficiency, performance per dollar, of HPSSD-all is 1.29 and 1.52 for *Flush* and *Sync*, respectively as shown in Figure 10. These cost efficiencies show 9.1% and 15.0% improvements compared to the efficiencies of Tach-Ram. On average, HPSSD-all shows 12.0% better cost efficiency compared to Tach-Ram.

2) Energy Efficiency: We use performance-per-watt to compare energy efficiencies. Our PCIe SSD consumes around 6W when active and this is relatively small overhead by considering the power consumption of each data node, around 200W. HPSSD-all shows 1.26 and 1.48 for *Flush* and *Sync*, respectively as shown. These energy efficiencies show 22.6% and 23.2% improvements compared to the efficiencies of Tach-Ram. On average, HPSSD-all shows 22.9% better energy efficiency compared to Tach-Ram.

C. Discussion

At the beginning of this section, we had a hypothesis in that high-performance SSDs would reduce the average memory access latency by alleviating the saturated memory bandwidth. To shed some light on this, we conduct experiments to evaluate the effectiveness of utilizing high-performance SSDs in reducing an average memory-access latency. In this experiment, we measure L3CLK, memory stall clock due to L3 cache misses, which represents a relative-but-real memory access time. We pick SVM and Text with shuffle sync for the applications because these showed speedups by using the HPSSD-all scheme compared to Tach-Ram.

Figure 11 (a) and (b) show the memory access time with HPSSD-all and Tach-Ram. As shown in Figure 11-(a), both lines of SVM show similar trends caused by program phase changes while Tach-Ram fluctuating with much higher peaks. Likewise, for Text, as shown in Figure 11-(b), Tach-Ram shows more fluctuations with higher peaks compared to HPSSD-all. Such reduced memory-access latencies were mainly due to decreased total memory references. Figure 11-(c) shows the corresponding total memory access counts of SVM and Text shown in Figure 11-(a) and (b). For SVM, HPSSD-all reduces the total memory accesses as much as 46.2%.

Putting it all together, HPSSD-all not only showed the best performance of 1.45X on average, but also required least efforts to run all workloads. Although both Tach-Ram and Tach-HPSSD are good candidates to achieve best performances, it requires workload specific information as well as the input-data dependent characteristics, making these approaches less attractive. As a result, utilizing high-performance SSDs for local storages to store persisted RDDs and shuffling data is a promising solution to achieve high performance in most cases.

From the perspective of Spark framework, utilizing SSDs for storage engines is a good separation between execution and storage engines. That is, execution and storage engines can be tuned independently. For example, JVM memory tuning does not need to care about neither space allocation in the heap space, nor physical memory space for Tachyon.

VI. Related Work

There have been many studies that investigate the performance improvement by adopting SSDs in Hadoop framework. Previous studies have worked on improving performance in HDFS by adopting SSDs either as replacements of HDDs or an addition to HDDs along with high-performance networks. For example, Sur et al. [21] evaluate the impact of high-performance interconnects and SSDs on HDFS. This work showed that the combination of advanced interconnects with SSD improve the HDFS performance significantly. Islam et al. [12] optimize HDFS using Remote Direct Memory Access (RDMA) over InfiniBand. However, these studies are limited in its scope for optimizing distributed file systems by adopting better storage devices and network. On the other hand, we investigate the usage of SSDs in Spark and our purpose is to speed up execution framework rather than distributed file systems.

There also have been studies from the distributed-computing perspective of Hadoop. Appuswamy et al. [4]

compare cost-effectiveness of scale-up and scale-out approaches by utilizing SSDs. Moon et al. [17] evaluate the effectiveness of using SSDs in Hadoop framework by comparing its performance and cost-effectiveness. This study shows that using SSDs improve both performance and cost-effectiveness in Hadoop framework. Kambatla et al. [13] evaluate the performance of using PCIe SSDs either in place of or in addition to HDDs. Krish et al. [14] evaluate an effective use of SSDs in Hadoop framework by using it as cache layer for the slower HDDs. It shows that using SSDs for caching devices is as fast as a system with SSDs only while achieving higher cost effectiveness. Lastly, Tan et al. [22] investigate the effectiveness of using SSD in Hadoop and show best practices of data placement in a SSD-HDD storage hierarchy. Having said that, to the best of our knowledge, ours is the first study to investigate in Spark framework with high-performance PCIe SSDs.

Recently, there are couple of studies that characterize the performance bottlenecks in Spark framework. For example, Ousterhout et al. [19] investigate on performance bottlenecks based on blocked time analysis. In this study, they claim that CPU is the bottleneck in most cases while disk bottleneck is a minor problem. But we show that local storages can affect the performance significantly and we can achieve the performance improvement as much as 1.5X compared to highly-tuned memory-only approach by utilizing high performance SSDs. Awan et al. [5] characterize the performance in Spark framework from the micro-architecture perspective. This study shows that thread-level load imbalance prohibits an ideal scalability in single scale-up node. Similarly, Wang et al. [25] explore an adoption of Spark on HPC systems by optimizing storage architectures. They introduce enhanced load balancer and congestion-aware task dispatching to improve the performance on the system. However, we show that memory bandwidth is one of the bottleneck in commodity clusters and providing additional bandwidth by using high-performance SSDs can improve the overall performance significantly.

VII. Conclusion

In this paper, we showed that high performance SSDs can improve the overall performance in Spark framework by alleviating the limited memory system performance. In-memory cluster computing generates more I/O to utilize cost-effective commodity servers. Unfortunately, current DRAM memory cannot handle these I/Os due to (1) its limited capacity and (2) limited bandwidth. Meanwhile, modern high performance SSDs provide relatively high bandwidth and larger capacity at lower price than before. Therefore, we propose to use such high performance SSDs for the heavy I/Os while alleviating bandwidth and capacity requirement at memory. Our experimental results show that using high performance SSDs can improve the overall performance 23% on average compared to the performance of memory-only approach, exploiting memory to store both persisted RDDs and shuffle data. Moreover, our approach can improve the cost and energy efficiency by 12% and 22.9%, with the metrics of performance per dollar and performance per watt, respectively. Lastly, it is notable that utilizing high performance SSDs not only showed the best performance and efficiencies and is a better solution for Spark clusters in that it does not require further memory tunings by isolating Spark execution and storage engines.

References

[1] J. Ahn, S. Hong, S. Yoo, O. Mutlu, and K. Choi, "A scalable processing-in-memory accelerator for parallel graph processing," in *Proceedings of the 42nd Annual International Symposium on Computer Architecture*. ACM, 2015, pp. 105–117.

[2] G. Ananthanarayanan, A. Ghodsi, A. Wang, D. Borthakur, S. Kandula, S. Shenker, and I. Stoica, "Pacman: coordinated memory caching for parallel jobs," in *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*. USENIX Association, 2012, pp. 20–20.

[3] G. Ananthanarayanan and I. Menache, "Big data over networks."

[4] R. Appuswamy, C. Gkantsidis, D. Narayanan, O. Hodson, and A. Rowstron, "Scale-up vs scale-out for hadoop: Time to rethink?" in *Proceedings of the 4th annual Symposium on Cloud Computing*. ACM, 2013, p. 20.

[5] A. J. Awan, M. Brorsson, V. Vlassov, and E. Ayguade, "Performance characterization of in-memory data analytics on a modern cloud server," *arXiv preprint arXiv:1506.07742*, 2015.

[6] S. Brin and L. Page, "Reprint of: The anatomy of a large-scale hypertextual web search engine," *Computer networks*, vol. 56, no. 18, pp. 3825–3833, 2012.

[7] C.-C. Chang and C.-J. Lin, "Libsvm: A library for support vector machines," *ACM Transactions on Intelligent Systems and Technology (TIST)*, vol. 2, no. 3, p. 27, 2011.

[8] J. Dean and S. Ghemawat, "Mapreduce: simplified data processing on large clusters," *Communications of the ACM*, vol. 51, no. 1, pp. 107–113, 2008.

[9] R. Dementiev, T. Willhalm, O. Bruggeman, P. Fay, P. Ungerer, A. Ott, P. Lu, J. Harris, P. Kerly, and P. Konsor, "Intel performance counter monitor," 2012.

[10] Hadoop.apache.org, "Apache hadoop," 2015. Available: https://hadoop.apache.org/

[11] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly, "Dryad: distributed data-parallel programs from sequential building blocks," in *ACM SIGOPS Operating Systems Review*, vol. 41, no. 3. ACM, 2007, pp. 59–72.

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

[13] K. Kambatla and Y. Chen, "The truth about mapreduce performance on ssds," in *Proc. USENIX LISA*, 2014.

[14] K. Krish, M. S. Iqbal, and A. R. Butt, "Venu: Orchestrating ssds in hadoop storage," in *Big Data (Big Data), 2014 IEEE International Conference on*. IEEE, 2014, pp. 207–212.

[15] H. Kwak, C. Lee, H. Park, and S. Moon, "What is Twitter, a social network or a news media?" in *WWW '10: Proceedings of the 19th international conference on World wide web*. New York, NY, USA: ACM, 2010, pp. 591–600.

[16] H. Li, A. Ghodsi, M. Zaharia, E. Baldeschwieler, S. Shenker, and I. Stoica, "Tachyon: Memory throughput i/o for cluster computing frameworks," *memory*, vol. 18, p. 1, 2013.

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

[18] O. Mutlu and T. Moscibroda, "Parallelism-aware batch scheduling: Enhancing both performance and fairness of shared dram systems," in *ACM SIGARCH Computer Architecture News*, vol. 36, no. 3. IEEE Computer Society, 2008, pp. 63–74.

[19] K. Ousterhout, R. Rasti, S. Ratnasamy, S. Shenker, B.-G. Chun, and V. ICSI, "Making sense of performance in data analytics frameworks," in *Proceedings of the 12th USENIX Symposium on Networked Systems Design and Implementation (NSDI) Oakland, CA*, 2015, pp. 293–307.

[20] Spark.apache.org, "Mllib — apache spark," 2015. Available: https://spark.apache.org/mllib/

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

[22] W. Tan, L. Fong, and Y. Liu, "Effectiveness assessment of solid-state drive used in big data services," in *Web Services (ICWS), 2014 IEEE International Conference on*. IEEE, 2014, pp. 393–400.

[23] D. Tsirogiannis, S. Harizopoulos, and M. A. Shah, "Analyzing the energy efficiency of a database server," in *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*. ACM, 2010, pp. 231–242.

[24] L. Wang, J. Zhan, C. Luo, Y. Zhu, Q. Yang, Y. He, W. Gao, Z. Jia, Y. Shi, S. Zhang *et al.*, "Bigdatabench: A big data benchmark suite from internet services," in *High Performance Computer Architecture (HPCA), 2014 IEEE 20th International Symposium on*. IEEE, 2014, pp. 488–499.

[25] Y. Wang, R. Goldstone, W. Yu, and T. Wang, "Characterization and optimization of memory-resident mapreduce on hpc systems," in *Parallel and Distributed Processing Symposium, 2014 IEEE 28th International*. IEEE, 2014, pp. 799–808.

[26] W. A. Wulf and S. A. McKee, "Hitting the memory wall: implications of the obvious," *ACM SIGARCH computer architecture news*, vol. 23, no. 1, pp. 20–24, 1995.

[27] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica, "Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing," in *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*. USENIX Association, 2012, pp. 2–2.