StoreApp: Shared Storage Appliance for Efficient and Scalable Virtualized Hadoop Clusters

Yanfei Guo, Jia Rao, Dazhao Cheng
Dept. of Computer Science
University of Colorado at Colorado Springs, USA
yguo@uccs.edu

Changjun Jiang
Department of Computer Science & Technology
Tongji University, Shanghai, China
cjjiang@tongji.edu.cn

Xiaobo Zhou
Department of Computer Science
University of Colorado at Colorado Springs, USA
xzhou@uccs.edu

ABSTRACT

Virtualizing Hadoop clusters provides many benefits, including rapid deployment, on-demand elasticity and secure multi-tenancy. However, a simple migration of Hadoop to a virtualized environment does not fully exploit these benefits. The dual role of a Hadoop worker, acting as both a compute and a data node, makes it difficult to achieve efficient IO processing, maintain data locality, and exploit resource elasticity in the cloud. We find that decoupling per-node storage from its computation opens up opportunities for IO acceleration, locality improvement, and on-the-fly cluster resizing. To fully exploit these opportunities, we propose *StoreApp*, a shared storage appliance for virtual Hadoop worker nodes colocated on the same physical host. To completely separate storage from computation and prioritize IO processing, StoreApp proactively pushes intermediate data generated by map tasks to the storage node. StoreApp also implements late-binding task creation to take the advantage of prefetched data due to mis-aligned records. Experimental results show that StoreApp achieves up to 61% performance improvement compared to stock Hadoop and resizes the cluster to the (near) optimal degree of parallelism.

1. INTRODUCTION

As the amount of data generated by enterprises and organizations has exploded, conventional warehouse systems are unable to efficiently store and analyse the data. MapReduce, a distributed programming model on clusters of commodity hardware, has emerged as the *de facto* standard for processing a large set of unstructured data. Since big data analytics requires distributed computing at scale, usually involving hundreds to thousands of machines, access to such facilities becomes a significant barrier to practising big data processing in small business.

Moving MapReduce into the cloud, either converting the existing in-house computing facility into a private cloud or outsourcing data processing to a public cloud, offers a more efficient and cost-effective model to implement big data analytics [8]. It is believed that the benefits of virtualizing MapReduce include rapid deployment, high availability, on-demand elasticity and secure multi-tenancy [3]. However, a simple migration of MapReduce to a virtualized environment does not fully exploit the flexibility, efficiency and elasticity in the cloud.

Hadoop, the open source implementation of MapReduce, has been increasingly deployed in cloud environments. Leading public cloud vendors, such as Amazon EC2 [7] and Rackspace [22], already provide Hadoop as an online service. Virtualization software vendors, e.g., VMware, also have Hadoop virtualization extensions (HVE) [16] to support Hadoop in a private cloud. Hadoop consists of two subsystems: the MapReduce Engine and Hadoop Distributed File System (HDFS). In conventional clusters, these two subsystems are deployed on the same worker nodes so that computation can be directly applied to its associated data. The coupling of computation and storage requires that *data locality* be preserved as much as possible, which is critical to the performance of Hadoop map tasks. However, such settings pose significant challenges on achieving good Hadoop performance in a virtualized environment.

First, unified compute and data node impedes the scaling of Hadoop clusters. As discussed in [17], matching the cluster size with workload demands is crucial to the performance of MapReduce jobs and the efficiency of Hadoop clusters. However, adding or removing unified Hadoop nodes requires that data be rebalanced and thus incurs significant data movement. The resulted inter-node traffic inevitably limits the scalability of virtual Hadoop clusters and makes it difficult to exploit the elasticity provided by virtualization.

Second, coupling computation and storage leads to inefficient use of resources in virtualized environments. To exploit the parallelism of multicore processors, usually multiple Hadoop nodes are consolidated onto one physical machine. On the other hand, colocated virtual nodes often access disk concurrently, causing interleaved IOs on the host machine. These almost random IO operations can greatly affect the performance of Hadoop jobs that are bottlenecked by disk accesses. There is existing work focusing on exploiting the additional layer of data locality, i.e., *host-local*, in virtual Hadoop clusters [11, 16]. Data stored on colocated Hadoop nodes is considered local on the same host. Thus, Hadoop scheduler is modified to launch "local" task even its data is on a different virtual node. However, this approach does not address the inefficient IO accesses. The separation of data on different virtual nodes prevents Hadoop from coordinating concurrent disk IOs for better efficiency.

Finally, performing task execution and data serving in the same virtual node poses challenges on efficient virtual machine (VM) scheduling. It is believed that IO-intensive VMs should be prioritized as they only consume a small amount of CPU time for IO processing and stay idle waiting for IO completion. Existing virtual machine monitors (VMM) use a simple heuristic, i.e., short CPU

burst, to identify VMs doing mostly IOs and assign them higher priorities. However, the overlapping of computation and IO in the unified Hadoop node renders the heuristic ineffective. We show that (Section 2.2) VMs running a mix of computation and IO operations experience up to 50% IO performance degradations under multi-tenant interferences.

In this work, we find that decoupling the HDFS storage to a separate data node, one per host machine, opens up opportunities for optimized disk accesses, more efficient VM scheduling, and flexible Hadoop scaling. We propose *StoreApp*, a set of Hadoop optimizations that fully uncover the performance potential of using a separate HDFS storage node. StoreApp completely decouples disk accesses from compute nodes by redirecting intermediate data spills to the data node. By consolidating disk accesses into one VM, StoreApp uses existing IO prioritization mechanisms in VMMs to accelerate IO processing. To optimize disk accesses in the storage node, StoreApp takes advantage of the prefetched partial HDFS block due to unaligned records and devises a late-binding task scheduler to reduce the number of disk accesses by launching tasks associated with the prefetched data. We also demonstrate that automate cluster scaling is possible with the help of separated storage node.

We have implemented StoreApp on a 22-node virtual Hadoop cluster and evaluated its benefits using *TestDFSIO* benchmark and the Purdue MapReduce Benchmark Suite (PUMA) [2] with datasets collected from real applications. Experiment results show that StoreApp is able to reduce the job completion time by up to 61% for different benchmarks. We compared the performance of StoreApp running different workloads with that of the stock Hadoop and a recently proposed I/O efficient MapReduce implementation Themis [24]. Experimental results show that StoreApp reduces job completion time by 48% and 37% compared with stock Hadoop and Themis, respectively.

The rest of this paper is organized as follows. Section 2 introduces the background of Hadoop, discusses existing issues, and presents a motivating example. Section 3 elaborates StoreApp's architecture and key designs. Section 4 presents the implementation details of StoreApp. Section 5 gives the testbed setup and experimental results. Related work is presented in Section 6. We conclude this paper in Section 7.

2. BACKGROUND AND MOTIVATION

In this section, we first introduce the basics of the Hadoop MapReduce framework. Then, discuss the major challenges of virtualized Hadoop clusters. We show that moving HDFS into separate data nodes improves the read and write throughput.

2.1 Hadoop MapReduce Framework

Data processing in MapReduce is expressed as two functions: *map* and *reduce*. The map function takes an input pair and produces a list of intermediate key/value pairs. The reduce function processes the intermediate key with the list of its values and generates the final results. In the implementation of Hadoop, the *map* and *reduce* functions are implemented in `MapTask` and `ReduceTask`. Each MapReduce job is divided into multiple map tasks and reduce tasks.

Figure 1 shows the data storage in a Hadoop cluster. Hadoop has two type of storage: the Hadoop Distributed File System (HDFS) and the intermediate storage. HDFS is the persistent storage for input and output data of MapReduce jobs. Data in HDFS is saved as blocks and replicated across all HDFS nodes. The intermediate storage is the place for the storing intermediate result of a MapReduce job. For example, map tasks save their intermediate data to this temporary space when the output buffer is full. Temporary data is save in the local file system of a Hadoop node and deleted after a job is completed. In Hadoop's default setting, each worker node runs `TaskTracker` and `DataNode` at the same time. The coupling between task execution and data allow tasks to run locally avoiding inter-node data movement and communications. Hadoop tasks are created during job initialization and assigned input splits as their associated data. To preserve data locality, Hadoop task scheduler always tries to launch tasks onto nodes that contain their input data. If no such nodes are available, tasks will run on nodes where they access input data remotely.

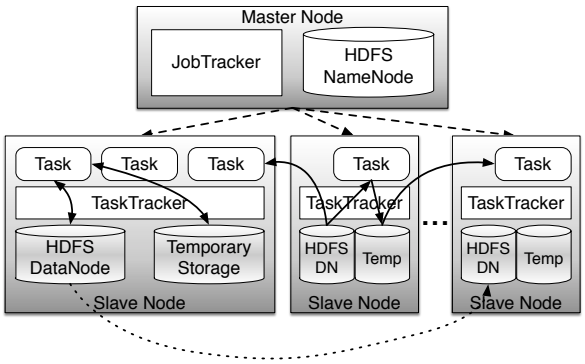


Figure 1: The slot-based task scheduling in Hadoop.

2.2 Challenges in Virtualized Hadoop Clusters

There exist several challenges when running Hadoop in a virtualized cluster. Performance interference in a shared cloud infrastructure and the lack of knowledge on the actual cluster topology are the main reasons for inefficient Hadoop execution.

As reported by [9], there are typically 4-6 VMs consolidated on one physical server in modern datacenters. Therefore, performance interferences from co-located VMs are quite common in virtualized environments. While interference usually come from applications belonging to other cloud tenants, it can also originate from competitions within the same cluster application. Depending on the deployment of VMs, multiple virtual Hadoop nodes can co-locate on the same physical machine. It is also a recommended virtual Hadoop cluster deployment from industry documentations [4, 16]. However, such colocations inevitably incur resource competitions among virtual Hadoop nodes. As most Hadoop jobs are data intensive, their performance is primarily bottlenecked by slow disk accesses. When running in a consolidated scenario, individual nodes can cause interleaved and non-sequential disk IOs. The tight coupling of computation and storage in one Hadoop node makes it difficult to coordinate concurrent disk accesses on different nodes.

The load of input data and intermediate data spills are two main sources of disk IOs in Hadoop. A task reads input data using the `FileInputFormat` interface and transforms unstructured input data that saved in the HDFS into meaningful records. As record lengths can vary, the record at the end of a HDFS data block may be incomplete. When Hadoop loads input data for a task, it will fetch the missing part of the misaligned record from the data block on a different data node. Due to this misalignment between input records and HDFS blocks, creating one input split may result in reading multiple data blocks and issuing multiple disk seek requests to the `DataNode`. Two data blocks that contains consecu-

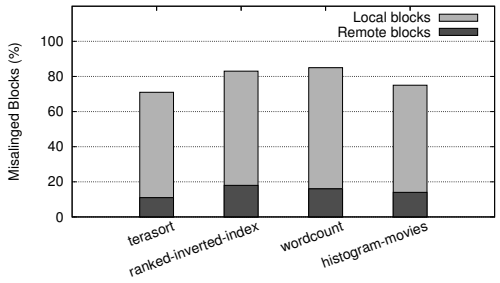


Figure 2: The percentage of blocks with misaligned records.

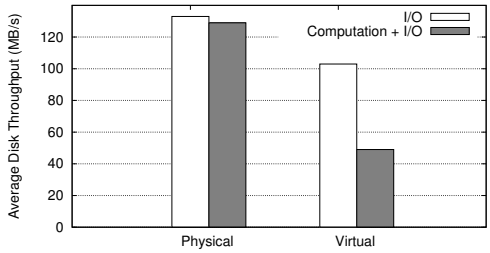


Figure 3: Computation and IO overlapping significantly affects IO performance in virtualized environments.

tive records may not store sequentially on disk, therefore reading them incurs even more random access to the disk and significantly reduces the efficiency of disk IO. More importantly, the fetch of incomplete records requires that a data node read the complete data block into memory. As only part of the loaded block will be sent to the node requesting the record, the remaining loaded data, though is resident in memory, can not be efficiently utilized. In the default Hadoop setting, such in-memory HDFS fragments spread over many independent data nodes, making it even harder to use them.

Next, we show that such misaligned records are prevalent in representative Hadoop workloads. We ran the PUMA benchmarks suite [2] and set the HDFS block size to 64MB. Figure 2 shows the percentage of blocks that have misaligned records at the begin or end of the block. For all four input data sets in PUMA, more than 70% data blocks has misaligned records at the begin or end of the block. Among all these blocks, around $15.5 - 18.7\%$ involved reading the second block from a remote data node. As long as the records have variable lengths, the misalignment between records and blocks is inevitable.

In a virtualized environment, there also exist contentions on CPU resources between co-running applications. Such contentions are particularly damaging to IO workloads because IO processing time can be significantly prolonged. Inspired by traditional operating system design, modern VMMs use a simple heuristic to identify IO intensive workloads and prioritize them to avoid poor IO performance. If a VM spends a significant amount of time idling (probably blocked by IO operations) and occasionally consumes short CPU bursts, it will receive a higher priority. However, Hadoop jobs do not benefit from the IO acceleration. To hide IO latency, Hadoop uses different threads in a task to handle task execution, intermediate data spill, and shuffling. This design works well in a dedicated cluster but is less efficient in a cloud environment with interferences. Due to the overlapping of computation and IO, VMMs fail to prioritize a node when it is performing IO operations.

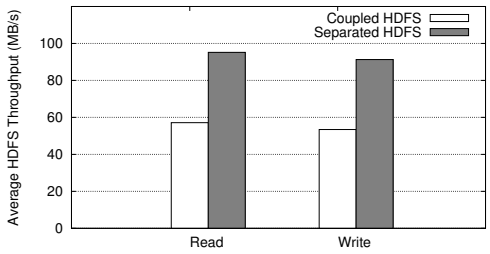


Figure 4: The average HDFS read and write throughput.

We created a micro-benchmark that sequentially accesses 10 GB data on disk and ran it on both a physical and a virtualized machine. We emulate the overlapping of computation and IO by running a busy loop along with the IO benchmark. We ensured that the aggregate CPU demand of the busy loop and the IO benchmark is below the capacity of one CPU so that the competition between the two does not affect IO performance. In the virtualized environment, we put the busy loop and the IO benchmark in a 1-vCPU VM and created another VM on the same host competing for CPU cycles. Figure 3 shows the IO throughput in different settings. Although virtualization adds overheads to IO processing, IO only workloads achieved reasonably good performance (i.e., 103 MB/s) in a virtualized environment even in the presence of interferences. However, when co-running with computation-intensive programs, the IO benchmark suffered a significant performance degradation (i.e., 49 MB/s). Next, we show that decoupling the data node from compute node and use a shared storage node per machine, offers opportunities for more efficient IO processing and more flexible cluster scaling.

2.3 Shared HDFS Storage Appliance

Consolidating data storage on multiple nodes into one data node shared by Hadoop nodes colocated on the same machine, provides a possible solution to the stated issues in virtualized environments. By serving multiple compute nodes, the shared storage node can possibly apply optimizations to a group IO requests. Separating IO from computation allows more effective VM scheduling at the VMM level. Next, we show that the decoupling provides immediate performance improvement to Hadoop HDFS benchmarks.

We built a controlled testbed of two Dell T420 servers, each with two Intel Xeon E5-2420 hex-core CPUs and 16 GB memory. Xen 4.3 and Linux kernel 3.10.0 were used as the VMM and the guest OS. We created an 11-node virtualized Hadoop cluster on the two hosts. One nodes were configured as the `JobTracker` and the `NameNode`. We created two shared storage nodes, one per machine. Each of them had 4 VCPUs and 4 GB memory. They were assigned higher priorities in the Xen scheduler to improve the responsiveness of I/O operations. We set the HDFS block size to its default value 64 MB. The set the number of replications to one to ensure that two storage appliance contain different part of the input data. The rest 8 nodes were configured as compute nodes for task execution. Each compute node was allocated with 2 VCPU and 2 GB memory. We set the per-node map and reduce slots to 4 and 2, respectively.

Figure 4 shows the HDFS read and write throughput using the *TestDFSIO* benchmark. The results show that the separate HDFS storage node achieved 66.5% and 70.9% higher average read and write throughput than the unified Hadoop node, respectively. It shows that using separate HDFS in a virtualized Hadoop cluster leads to higher disk performance.

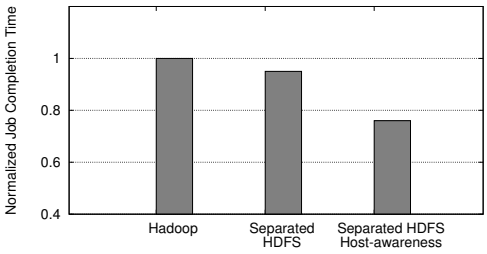


Figure 5: The normalized job completion time of *terasort*.

The separate HDFS node also improves the flexibility in task scheduling. Because no compute node have input data in its local disk, it automatically enables the task scheduler to launch host-local tasks. Figure 5 shows the performance the *terasort* benchmark due to different approaches. The results show that using separated HDFS itself can only reduce the job completion time by 5%. But combine the separated HDFS with host-aware task scheduling can significantly reduce the job completion time by 24%. The performance improvement mainly because the elimination of unnecessary remote tasks.

[**Summary**] We have shown that separating the HDFS from compute nodes improves the performance of HDFS in virtualized Hadoop cluster. To fully exploit the benefit of shared HDFS storage, there are still several issues to be addressed. First, decoupling HDFS doe not remove all IOs from compute nodes. Intermediate data is saved in the local file system of a compute node. Unless all IOs are consolidated onto the storage node, IO processing can not be properly accelerated. Second, there still lacks a mechanism to utilize the in-memory HDFS fragments, which may significantly reduce the number of disk accesses. Finally, there is a need for auto-scaling virtual Hadoop clusters to realize cost-effective hosting. To address these issues, we propose StoreApp, a collection of techniques that optimize Hadoop performance based on shared storage appliance.

3. STOREAPP DESIGN

We propose *StoreApp*, a shared storage appliance for virtualized Hadoop clusters. It consists of a set of storage nodes and a StoreApp manager. Each physical machine has one storage node shared by all the compute nodes residing on the same host. It decouples HDFS data nodes from compute nodes, and provides the accurate information about data locality. To consolidate all IO operations, StoreApp pro-actively pushes intermediate data to the storage node. StoreApp provides efficient HDFS I/O through HDFS prefetching and implements a late-binding task scheduler that schedules tasks onto prefecthed HDFS data. StoreApp further employs an automated cluster resizing technique to determine the optimal cluster size for different jobs.

3.1 Overview

Figure 6 shows the architecture of StoreApp. StoreApp consists of four components *HDFS proxy*, *shuffler*, *StoreApp manager*, and *task scheduler*. The HDFS proxy receives all HDFS requests and forwards them to the HDFS DataNode. The shuffler is a job independent shuffle service that receives the map outputs and proactively pushes them to their destination data nodes. It removes all IOs due to intermediate data spills at the local disks of compute nodes. With these two components, StoreApp fully decouples the storage and computation. The StoreApp manager coordinates the operation of all data nodes and provides the global view of data availability. The task scheduler extends existing Hadoop scheduler

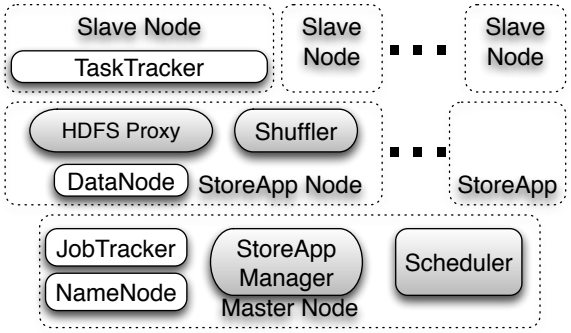


Figure 6: The architecture of StoreApp.

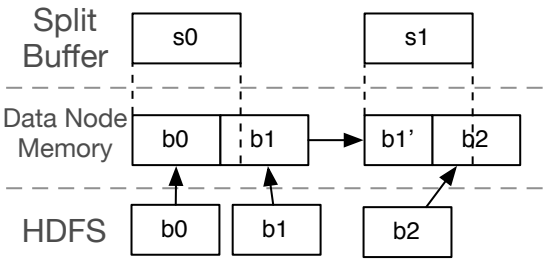


Figure 7: Process of prefetching in HDFS proxy.

to consider the data availability in individual data nodes. We briefly describe major features of StoreApp.

Shared Hadoop Storage - The shared storage provides separated VMs for storage in Hadoop. The storage VM is assigned with higher priority in VMM scheduler to improve the I/O efficiency. It separates the HDFS DataNode from the compute node and implements an independent shuffler to collect the intermediate data. The shared storage is designed to be compatible with current Hadoop API. Any user job can be moved on StoreApp seamlessly.

HDFS Prefetching - The HDFS proxy provides a prefetching service to the HDFS. It prefetches the data adjacent to the partial HDFS block due to misaligned input records and form a complete input split in memory. The new split can later be fed to a newly created task. HDFS proxy exploits the fragmented in-memory block to reduce the number of disk accesses.

Late-binding Task Scheduler - The task scheduler on-the-fly creates tasks and associates them with prefetched data in the data nodes. It also uses the deployment information of shared storage node on each machine and its association with corresponding compute node to schedule map tasks.

Automated Cluster Resizing - StoreApp monitors the processing of individual tasks of a job and searches for the optimal cluster size for the job. By maximizing the parallelism of job execution, it significantly improves the efficiency of virtualized Hadoop cluster.

3.2 HDFS Prefetching

StoreApp uses the HDFS proxy to create the input splits based on fragmented HDFS blocks. We design a prefetching technique that

preloads data when the unaligned read occurs. Figure 7 shows an example of how the data is prefetched. During the creation of an input split `s0`, the prefetcher first reads data block `b0` into the memory buffer. Because the last record at block `b0` is incomplete, the prefetcher needs to read the rest of the record from block `b1`. Instead of only reading the needed partial record, the prefetcher reads the whole block `b1` into the memory buffer. Then, the prefetcher serves the input split `s0` to the task. The unused data in block `b1` is kept in the memory, and the prefetcher reads its consecutive block into the memory to form input split `s1`. Therefore, when the task scheduler assigns a new task to process input split `s1`, the data will be readily available in the memory of the StoreApp node. StoreApp keeps tracking the alignment of record read and performs prefetching if unaligned reads are detected.

Algorithm 1 HDFS Prefetching.

```
1: Variables: Memory buffer area S; Input buffer B;
2:
3: function CREATEINPUTSPLIT(t)
4:     get the input block list L for task t
5:     check S and fetch any block in L and not in S
6:     concatenate all block in L and save in buffer area S[L]
7:     find the start of first record i
8:     find the end of last record j
9:     move data in S[L][i : j] into buffer B[t]
10:    get the last block ID k in L
11:    move data in S[L][j :] into buffer S[k]
12: end function
```

Algorithm 1 shows the pseudo code of the prefetching algorithm. When the `CreateInputSplit` function is invoked by the prefetcher to create the input split for task t, it first extracts the list L of input blocks from the task configuration. Then it checks the buffer area S with all the block IDs in list L and fetches all blocks that is not stored in S. All fetched blocks are concatenated and saved in S. Next, the algorithm identifies the begin and end of the input split and moves the corresponding data into the input buffer for task t. The unused data is kept in the buffer area S and indexed with the ID of the block. Each StoreApp node reports the list of prefetched input splits to the StoreApp manager to prevent multiple data nodes from prefetcing that same blocks.

3.3 Late-binding Task Scheduler

StoreApp provides accurate information of data locality in virtualized Hadoop clusters. But Hadoop task schedulers need modifications to support the functionality of StoreApp. In Hadoop, map tasks are assigned based on the location of its input data. However, StoreApp requires tasks to be scheduled based on the data availability on data nodes, e.g., the availability of prefetched input splits. The task scheduler also needs to check the partition placement on a StoreApp node and schedules the corresponding reduce task. StoreApp implements a late task binding mechanism that dynamically associates a task with the available input data. For map tasks, the scheduler first checks the prefetched blocks of a StoreApp node and associates one map task with the data. The task can be scheduled to any of the compute nodes that co-locates with the StoreApp node. The similar process applies to reduce tasks. A reduce task will not be assigned with a partition until it is being scheduled. Since all shuffle data has been pro-actively pushed to one of the StoreApp nodes, reduce scheduling is deterministic.

For single-user clusters, we modified Hadoop's FIFO scheduler to support the runtime task-data binding. When a compute node with free map or reduce slots requests for new tasks, the `task scheduler` first checks with the StoreApp manager to obtain the list of prefetched data or available partitions that reside on the co-hosted StoreApp node. The scheduler picks the input split or partition in the list and associates its ID with the first map or reduce task in the waiting queue. The selected task is then launched on that compute node. As such, all tasks are guaranteed to have host-local access to their input data and some tasks can even read their input directly from memory.

For multi-user clusters with heterogeneous workloads, we add the support for runtime task-data binding to the Hadoop Fair Scheduler (HFS). The minimum fair share allocated to individual users can negatively affect the efficiency of StoreApp as tasks may be launched on remote nodes to enforce fairness. We disable such fairness enforcement for reduce tasks to support more flexible scheduling. This allows some users to temporarily run more reduce tasks than others. We rely on the data replication of HDFS to minimize the chance of starting remote map tasks. Because the `DataNode` of HDFS is deployed on StoreApp. The default configuration of replication can ensure that each data block is replicated on three different physical machines. Given the fact that each physical machines hosts multiple compute nodes, the possibility that a task cannot find a local compute node is quite small.

3.4 Automated Cluster Resizing

With the separation of the storage and computation, adding compute node to virtualized Hadoop clusters no longer requires data rebalancing. It enables flexible resizing of Hadoop clusters. To maximize the efficiency of virtualized Hadoop cluster, StoreApp implements automated cluster resizing. The goal of automated cluster resizing is to determine the number of compute nodes on each host that yields the shortest completion time of a given job.

Hadoop provides *progress score* to represent the fraction of work completed. It is computed as the ratio of finished input size and the original input size. *Progress rate* [30], which is defined as the change in progress score in unit time, is a good online measure of task performance. By monitoring the progress rate of each map task, StoreApp is able to calculate the expected completion time of a map task as $t = \frac{1}{r}$, where r is the progress rate of the task. Due to the job characteristics and the data distribution, the optimal number of compute nodes on one host can vary from host to host. Therefore, we need to evaluate the choice of number of compute nodes on each host machine separately.

For example, a MapReduce job with n map tasks is running on a tiny cluster within one physical machine, the expected completion time of the job's map phase can be calculated as

$$T_{map} = \frac{n}{p} \times t_{avg}$$

where p is the number of compute nodes per physical machines and t_{avg} is the average expected completion time of tasks. We can evaluate T_{map} with different p and find the one that leads to the minimum t_{map}. Thus we can assert that for this particular physical machine, provisioning p compute nodes leads to the best job performance and resource efficiency. StoreApp uses an iterative approach to search the optimal cluster size through multiple trials.

The pseudo-code of the automated cluster resizing algorithm is shown in Algorithm 2. StoreApp performs the algorithm on each physical machine. The algorithm first increases the number of compute nodes on the physical machine p by one and provisioning new node accordingly. Then it calculates the expected job completion time T_{new} and compares it with the expected job completion time in the previous iteration T_{old}. If T_{new} is larger than T_{old}, the performance degradation due to intra-cluster contention outweighs hav-

Algorithm 2 Automated cluster resizing.

Variables: List of all running tasks on one physical machine M; Number of computation nodes on one physical machine p; Expected completion time of the job in previous iteration T_{old};
while true **do**
  $p \leftarrow p+1$
  provision additional node
  for i in M **do**
    $t_i \leftarrow 1/r_i$
  end for
  calculate average expected completion time for M as t_{avg}
  $T_{new} = n/p \times t_{avg}$
  if $T_{new} > T_{old}$ **then**
    $p \leftarrow p-1$
    kill newly provisioned node
    break
  end if
end while

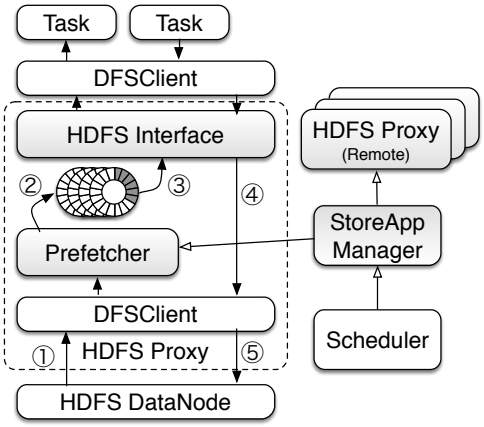


Figure 8: The structure of HDFS proxy.

ing a greater degree of parallelism. Thus the newly provisioned node should be removed.

4. IMPLEMENTATION

In Hadoop, the HDFS is abstracted in class `DistributedFileSystem`. It wraps the `DFSClient` that communicates with the `NameNode` in the master and the `DataNode` in the slave nodes. A task's operations on `DistributedFileSystem` is performed by its internal `DFSClient`. It is responsible for retrieving location of individual blocks from `NameNode` and accessing the data blocks from `DataNode`. The HDFS proxy is deployed with `DataNode` and intercepts the request from `DFSClient`. It listens on the RPC port and passes received requests to the `DataNode`. Therefore, prefetching feature can be easily implemented without change the HDFS.

The structure of the HDFS proxy is shown in Figure 8. It has three main components: `Prefetcher`, `SplitBuffer`, and `HDFS Interface`. The route ①②③ serves the HDFS read operation with prefetching. The HDFS proxy will create the input split before a task is scheduled to the slave node that the StoreApp serves. When a task issues HDFS read, the HDFS proxy will respond the request with buffer input splits. The route ④⑤ serves the HDFS write operation. When a task issues HDFS write, the HDFS proxy will pass the request to the underlying HDFS. In order to create the input splits in the HDFS proxy. HDFS proxy retrieve the `InputFormat` for the job from the StoreApp and use it to find the begin and end of a record in data blocks.

We implement a modified version of *iShuffle* [15] to handle the intermediate data. The `ShuffleServer`, `DataDispatcher`, and `Merger` are implemented as different threads. The grouped map output records is saved in the intermediate queues and waiting for partitioning. StoreApp repetitive performs the balanced partitioning algorithm with latest update of map output record groups. The partitioned and shuffled results are saved as reduce input files. The task scheduler performs runtime task-data binding to associates these data with reduce tasks.

5. EVALUATION

We implement StoreApp on a prototype cloud environment and evaluate its performance in improving HDFS throughput and reducing job completion time. We further study how virtualized Hadoop cluster benefits from automated cluster resizing. We also evaluate the performance of StoreApp in multi-user Hadoop clusters.

5.1 Testbed Setup

Our cloud environment consists of 4 Dell T420 servers. Each server is equipped with 2-way Intel Xeon E5-2420 hex-core CPUs and 16 GB memory. The servers are connected with Gigabit Ethernet. Xen 4.3 with Linux kernel 3.10.0 is used to provide the server virtualization.

The testbed is a 22-nodes Hadoop cluster. We deployed Hadoop stable release version 1.1.1 and each VM run Ubuntu Linux with kernel 2.6.32. Two nodes are configured as the `JobTracker` and `NameNode`, respectively. Four nodes are setup as *StoreApp* nodes. Each of them have 4 VCPUs and 4 GB memory. We set the HDFS block size to its default value of 64 MB. The rest 16 nodes are configured as compute nodes for MapReduce task execution. Each node is allocated with 2 VCPUs and 2 GB memory. Each slave node is configured with 2 map slots and 2 reduce slots. In the experiments of automated cluster resizing, the cluster can be scaled up to have 40 compute nodes.

For comparison, we also implemented Themis [24]. Themis is a system that focuses on I/O efficient MapReduce. It batches the disk I/O to exploit the sequential I/O performance of hard disk. It dynamically allocates memory buffers for the intermediate data and reduce the disk I/O with in-memory merge sort. Themis employed a different workflow for intermediate data processing. It moves the shuffle process to the map side. As the map task running, the output data is copied to different nodes and partitioned into intermediate results. However, the disk virtualization of the slave nodes reduces the effectiveness of I/O batching in Themis. Themis does not has the host-awareness for virtualized Hadoop cluster, which makes it suffers from the inflexible task scheduling that leads to concealed data locality.

5.2 Workloads

We used the *TestDFSIO* to evaluate the performance of HDFS. It is a HDFS specific benchmark that is included as a test job in Hadoop source code. It consists of two parts: *TestDFSIO-write* and *TestDFSIO-read*. It first writes a large amount of data into HDFS, then reads it back. *TestDFSIO* measures the maximum read and write throughput of the HDFS.

We also use the PUMA benchmark suite [2] to test the performance

Table 1: Benchmark details.

Benchmark	Input Size (GB)	Input Data	Output Size (GB)	Shuffle Volume (GB)
tera-sort	150	TeraGen	150	150
ranked-inverted-index	150	multi-word-count output	125	160
term-vector	150	Wikipedia	30	120
wordcount	150	Wikipedia	2	29
grep	150	Wikipedia	3	31
histogram-movies	100	Netflix data	0.0010	0.0012
histogram-ratings	100	Netflix data	0.0013	0.0014

of StoreApp with representative MapReduce jobs. The PUMA benchmark suite contains various MapReduce benchmarks and real-world test inputs. Table 1 shows the benchmarks and their configurations used in our experiments.

These benchmarks are divided into three categories based on the size of their intermediate data. The *terasort*, *ranked-inverted-index* and *term-vector* are I/O intensive benchmarks. They have a have large input data size and a large shuffle volume that is comparable to the input data size. These benchmarks poses great pressure to the host disk. The *wordcount* and *grep* are I/O sensitive benchmarks. They have a large input data size but an relatively small shuffle volume. They are sensitive to the performance disk. But the host disk is not a major bottleneck for them. The *histogram-movies* and *histogram-ratings* are I/O light benchmarks. The performance of HDFS have some impact on their performance, but the major bottleneck is the CPU.

5.3 HDFS I/O Performance

We study the HDFS I/O performance of StoreApp. We use the *TestDFSIO* benchmark to measured the throughput of the HDFS. We configure *TestDFSIO* to write 50 GB data into the HDFS and then read it back. The *TestDFSIO* benchmark does not have intermediate data and shuffling, which allows us to accurately measure the performance of HDFS storage. We compare the HDFS read and write throughput due to different approaches.

Figure 9 shows the average HDFS storage throughput in three cases: the stock Hadoop, StoreApp without prefetching, and StoreApp with prefetching. The results show that StoreApp with prefetching outperformed stock Hadoop by 78.3% and 71.8% higher average throughput in HDFS read and HDFS write, respectively. StoreApp significantly improves the throughput of HDFS in virtualized Hadoop cluster. StoreApp outperformed stock Hadoop for two reasons. First, the shared storage VM has higher priority in VM scheduling, it significantly improves the I/O responsiveness when the interference presents. Second, consolidating the storage into one VM reduces the I/O contentions between compute nodes and allow disk I/O operations to be batched and performed sequentially. As the traces in Figure 9(b) and Figure 9(c) shown, StoreApp not only achieved higher throughput than stock Hadoop in both read and write, it also achieved stabler throughput than stock Hadoop.

The prefetching plays an important role in improving the HDFS throughput. StoreApp with prefetching achieved 9.2% higher read throughput than StoreApp without prefetching. The traces in Figure 9(b) show that StoreApp without prefetching has similar unstable read throughput like stock Hadoop. Without prefetching, the HDFS read operation of Hadoop involves multiple disk seek.

5.4 Performance Impact of Efficient I/O

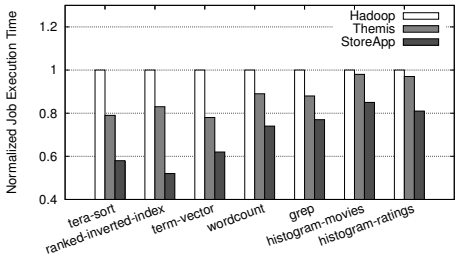


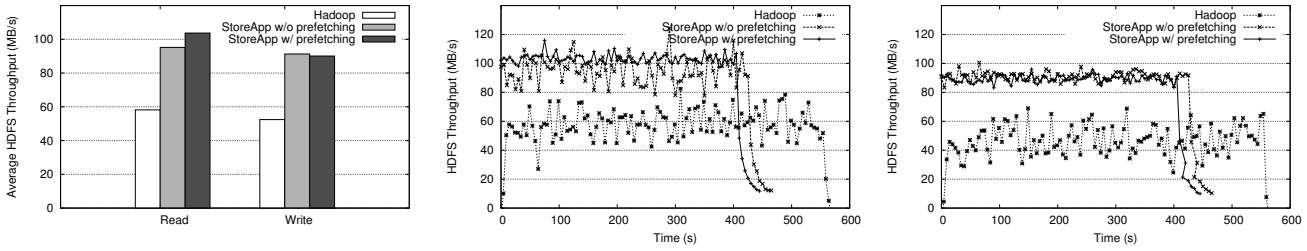
Figure 10: The normalized job completion time due to different approaches.

We have shown the StoreApp is effective in improving HDFS performance for virtualized Hadoop cluster. In this subsection, we study how StoreApp helps improving the overall job completion time. We use the job completion time in the stock Hadoop as the baseline and compare the normalized job completion time of StoreApp and Themis. Figure 10 shows the normalized job completion time of all benchmarks due to these three approaches. The results show that I/O intensive benchmarks with large intermediate data, *e.g.*, *terasort*, *ranked-inverted-index* and *term-vector*, StoreApp outperformed stock Hadoop by 42.2%, 48.1%, and 37.8%, respectively. StoreApp also outperformed Themis by 26.6%, 37.3%, and 20.5% in these benchmarks. Note that StoreApp achieved less improvement on *term-vector* because it has small output data, which means it has less HDFS write than other two benchmarks. This results in less pressure on the HDFS and decreases the performance gain of using StoreApp.

Benchmarks such as *wordcount* and *grep* are I/O sensitive. In the experiments with these benchmarks, StoreApp outperformed the stock Hadoop by 27.8% and 23.2%, respectively. However, the performance improvement of these benchmarks are less then I/O intensive benchmark, because their small intermediate data decreases the performance gain from the shuffle-on-write shuffler. StoreApp achieved 16.9% and 12.5% shorter job completion time than Themis in *wordcount* and *grep* benchmarks, respectively.

For I/O light benchmarks such as, *histogram-movies* and *histogram-ratings*, StoreApp outperformed the stock Hadoop by 15.4% and 19.1%, respectively. StoreApp also outperformed Themis by 13.3% and 16.5% in these benchmarks, respectively. StoreApp is able to reduce the job performance by improving the efficiency of HDFS, while Themis only achieves marginal performance improvement. StoreApp clearly showed a significant advantage compared to Themis.

To further understand how the job completion time is reduced by StoreApp. We pick three representative benchmarks, *terasort*, *wordcount*, and *histogram-movies*, from all three categories of bench-



(a) Average read and write throughput. (b) Read throughput. (c) Write throughput.

Figure 9: The HDFS storage throughput of Hadoop and StoreApp.

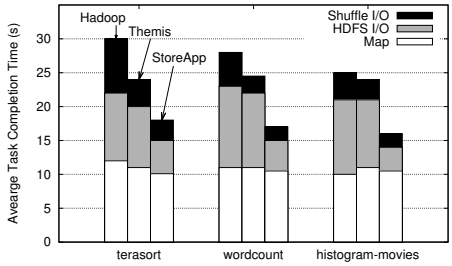


Figure 11: The average task completion time due to three different approaches.

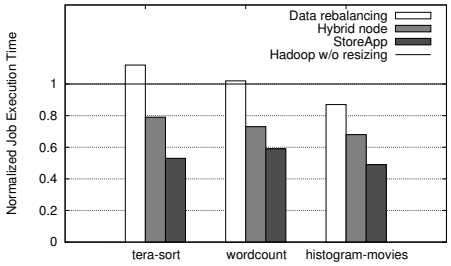


Figure 12: The normalized job completion time due to different storage organization.

marks. We compare the average task completion time due to stock Hadoop, Themis, and StoreApp. We decompose the average task execution time into three parts: the map time, the HDFS I/O time, and the shuffle I/O time. The map time represents the time that a map task spent on executing map function over input records. The HDFS I/O time represents the time spent on HDFS read. Note that in StoreApp, the HDFS read operation is fulfilled by the `SplitBuffer` due to the prefetching, the HDFS I/O time does not include the actual time that the StoreApp node accesses the HDFS. The shuffle I/O time represents the time spent on sending map output records to the shuffler in the case of StoreApp, or to the map output file in the case of Themis and stock Hadoop.

Figure 11 shows the average task completion time for these three benchmarks. The results show that StoreApp achieved 41.4%, 38.7%, and 40.2% shorter task execution time than stock Hadoop in these three benchmarks, respectively. StoreApp also outperformed Themis by 25.1%, 30.6%, and 33.3% in these benchmarks, respectively. StoreApp significantly reduces the task completion time results shorter job completion time.

The decomposition results show that StoreApp, Themis, and stock Hadoop spent similar time in executing map function. The results suggest that the differences in the task completion time is mainly due to the reduction in the time spent on shuffle I/O and HDFS I/O. Both StoreApp and Themis are able to reduce the shuffle I/O time. StoreApp used 62.5%, 60%, and 50% shorter shuffle I/O time than stock Hadoop in *terasort*, *wordcount*, and *histogram-movies* benchmarks, respectively. StoreApp outperformed stock Hadoop for two reasons. First, StoreApp bypasses the map-side combine, which eliminated extra disk access during shuffle. Second, StoreApp uses shuffler to collect and shuffle the map output. This saves the time for a map task by not writing intermediate data into disk. StoreApp also outperformed Themis with around 25% shorter shuffle I/O time in these benchmarks because StoreApp does not involve writing output data into local disk.

StoreApp achieved 51.2%, 62.5%, and 68.2% shorter HDFS I/O time than stock Hadoop in *terasort*, *wordcount*, and *histogram-movies* benchmarks, respectively. StoreApp outperformed Themis by 45.5%, 59.1%, and 65.2% in these three benchmarks, respectively. The improved performance on HDFS read of StoreApp shows significant advantage against Themis, which lacks the optimization on HDFS operations.

5.5 Benefit of Automated Cluster Resizing

StoreApp decouples computation nodes and storage nodes, which allows flexible cluster resizing. StoreApp is able to determine the cluster size for each job to further improve the performance and efficiency of virtualized Hadoop cluster. In this section, we first study how StoreApp improves the scalability of virtualized Hadoop cluster. Then we show how does automated cluster resizing help in reducing job completion time. We use the job completion time in the stock Hadoop as the baseline and compare the normalized job completion time of different approach. Each physical nodes can hold up to 10 slave nodes and the cluster is initialized with 4 nodes per host.

As Section 2 shown, one major challenge of virtualized Hadoop cluster is the scalability. In stock Hadoop cluster, adding compute nodes incurs significantly overhead due to the data rebalancing and can results in performance issue. Using hybrid nodes [10] can avoid data balancing, but breaks the data locality. Here we the evaluate the performance improvement of using automated cluster resizing with data rebalancing, hybrid nodes, and StoreApp. We use three representative benchmarks *terasort*, *wordcount* and *histogram-movies* for the experiment. Figure 12 shows the normalized job completion time of different approaches on all benchmarks. The results show that StoreApp outperforms data rebalancing by 47.3%, 42.1%, and 37.2% in *terasort*, *wordcount*, and

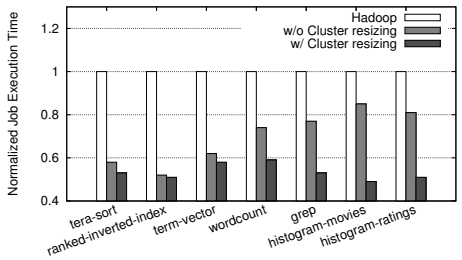


Figure 13: The normalized job completion time due automated cluster resizing.

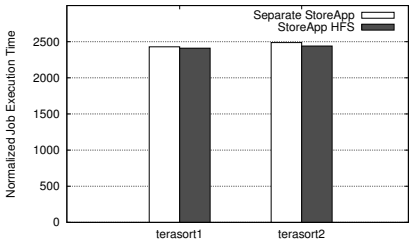


Figure 14: The job completion time of two *terasort* job due to different approaches.

histogram-movies, respectively. It demonstrates that StoreApp has better scalability in cluster resizing than data rebalancing. The overhead of data rebalancing evens out the performance gain due to the increased cluster size. Moreover, in *terasort* and *wordcount* benchmarks, cluster resizing with data rebalancing achieved longer job completion time than Hadoop without any resizing technique. StoreApp also outperforms hybrid nodes by 32.9%, 20.1%, and 27.9% in these benchmarks. It is mainly because the broken data locality and unsolve I/O contentions in hybrid nodes.

Figure 13 shows the normalized job completion time of all benchmarks due to these three approaches. The results show that StoreApp with automated cluster resizing can reduce the job completion time by up to 61%. For I/O intensive benchmarks like *terasort*, *ranked-inverted-index*, and *term-vector*, StoreApp with automated cluster resizing outperformed the stock Hadoop by 53.2%, 51.6%, and 58.4%, respectively. However, employing automated cluster resizing only brings very limited performance improvement for StoreApp in these three benchmarks. StoreApp with automated cluster resizing only achieved 8.9%, 2.1%, and 6.5% shorter job completion time than StoreApp without cluster resizing. The number of slave nodes per host is not changed for these benchmarks. Due to the fact that the disk of host machine is the bottleneck of I/O intensive benchmarks, increasing the number of slave nodes poses even more pressure on the disk. It reduces the performance of individual tasks and counteracts the speedup of increased level of concurrency.

For I/O sensitive benchmarks like *wordcount* and *grep*, StoreApp with automated cluster resizing outperformed the stock Hadoop by 59.7% and 53.5%, respectively. It also outperformed StoreApp without automated cluster resizing by 20.3% and 31.2% in *wordcount* and *grep*, respectively. These benchmarks poses less pressure to the host disk than I/O intensive benchmarks. Therefore, the host machine has extra resource than can be used to further improve the performance of these benchmarks. Thus the automated cluster resizing is able to increase the number of slave nodes per host and speedup the execution of jobs.

For I/O light benchmarks like *histogram-movies* and *histogram-ratings*, StoreApp with automated cluster resizing outperformed the stock Hadoop by 49.2% and 51.3%. Since the host disk is not the bottleneck for these benchmarks, StoreApp is able to provision more slave nodes on each physical machine. Thus, StoreApp with automated cluster resizing achieved 42.4% and 37% shorter job completion time than StoreApp without automated cluster resizing. The results also provide an interesting observation that the storage appliance and automated cluster resizing act in a complementary way, which can significantly improves the performance of different jobs.

Table 2: Job completion time of co-running jobs.

Workload Mix			Sep. StoreApp		StoreApp HFS	
A	+	B	A	B	A	B
terasort	+	grep	2210	1247	2046	1038
term-vector	+	wordcount	2410	1447	2196	1238
terasort	+	histogram -ratings	2308	653	1676	570
terasort	+	histogram -movies	2217	840	1584	790
terasort	+	TestDFSIO	2441	675	1474	721

5.6 Multiple Job Performance

In this section, we evaluate the effectiveness of using StoreApp in a multi-user Hadoop cluster. We created multiple workload mixes, each contained two different MapReduce jobs. We run one workload at a time with two jobs sharing the Hadoop cluster. We modified the Hadoop Fair Scheduler (HFS) (i.e., *StoreApp HFS*) to support runtime task-data binding and incorporate with StoreApp. For comparison, we also study the performance of StoreApp running a single job on a dedicated cluster (i.e., *Separate StoreApp*). We shrink the cluster to half of its original capacity to mimic the fair share of one job in the multi-user cluster.

First, we perform the experiment with two I/O intensive jobs. Figure 14 shows the performance of two *terasort* benchmarks. The results show the difference of job completion time of *terasort* in separate StoreApp and StoreApp HFS is less than 5%. StoreApp HFS is able to enforce fairness between these two benchmarks.

Then, we perform the experiment with the mix of one I/O intensive job and one I/O sensitive job. The first section of Table 2 shows the results of two different job mixes. The results show that StoreApp HFS improved job execution times by 8.6% and 16.8% over separate StoreApp in *terasort* and *grep* benchmarks, respectively. Although the size of input datasets of these two benchmarks are the same, *grep* has a smaller shuffle volume. Therefore, its reduce tasks can be started earlier as their intermediate data required less time to shuffle. Because the fair share of reduce tasks is removed in StoreApp, *grep* is allowed to temporarily use the whole cluster to run its reduce tasks until *terasort* starts reduce tasks. Moreover, *grep* finishes earlier than *tera-sort*, it also allows *terasort* to utilize the entire cluster. The result of *term-vector* and *wordcount* is similar where StoreApp reduces the job completion time of these two benchmarks by 8.9% and 14.4%, respectively.

Third, we test the combination of a I/O intensive job and a I/O light job. The second section of Table 2 shows the results for this case. The results show that StoreApp HFS improved the job execution

time by 27.4% and 12.7% over separate StoreApp in *terasort* and *histogram-ratings* benchmarks, respectively. Like the case of one I/O intensive job and one I/O sensitive job, both jobs benefit from the performance boost of reduce tasks. Because the shuffle workload of *histogram-ratings* is light and it finishes far earlier than *terasort*, *terasort* is able to use the entire cluster for longer time. It further improves the performance of *terasort*. In the case of *terasort* runs with *histogram-movies*, StoreApp HFS outperformed separate StoreApp by by 28.6% and 5.9%, respectively.

Furthermore, we study the performance boost of reduce tasks with an extreme case of heterogeneous workloads. We experiment with one *terasort* job and one *TestDFSIO* job. The last section of Table 2 shows the results show that StoreApp achieved 39.6% shorter job completion time in *terasort* than separate StoreApp. Because *TestDFSIO-write* only have one reduce task and it end instantaneously, *terasort* occupied every reduce slots on all slave nodes. However, the job completion time of *TestDFSIO* in StoreApp HFS is 6.8% longer than it in separate StoreApp. This unexpected result is due to the contention in HDFS. *TestDFSIO* has two parts: *TestDFSIO-write* and *TestDFSIO-read*, which results in intensive HDFS read and write in map tasks. Due to the replication factor of HDFS, *TestDFSIO* can generate triple as much I/O than *terasort* with the same amount of data. However, this is the extreme case of heterogeneous workload. It is very uncommon that a real world job has comparable intensity in HDFS I/O.

6. RELATED WORK

MapReduce is a programming model for large-scale data processing [13]. Hadoop, the open-source implementation of MapReduce, provides a software framework to support the distributed processing of large datasets [1]. YARN [27] is the second generation of Hadoop. It redesigns the resource management of Hadoop cluster and provides the support of different programming models. YARN provides a shuffle plugin interface that allows user to customize the shuffle process. But the shuffler is not an independent service by default. YARN does not provide host-awareness for virtualized Hadoop clusters. The storage and computation are still coupled.

Adaptive processing allows the Hadoop framework to change the execution flow of jobs to improving performance and efficiency. Recent work focuses on balancing the workload distribution by adaptively changing the partition of input data [5, 14, 18, 19, 25]. FLUX [25] splits an operator into mini-partitions. It monitors the utilization of different computation nodes by the time they spent in idle. FLUX moves mini-partitions from the most heavily utilized node to the most lightly utilized one. However, running a large amount of small tasks poses significant overhead. SkewReduce [18] alleviated the computational skew problem by applying a user-defined cost function on the input records. Partitioning across nodes relies on this cost function to optimize the data distribution. SkewTune [19] proposed a framework for skew mitigation by adaptive processing. It repartitioned the long tasks to take the advantage of idle slots freed by short tasks. SkewTune does not pose large scheduling overhead because only straggler tasks will be repartitioned. However, moving rTarazu balances the workload using data repartitioning [5]. It repartitions the intermediate data and distributes the workload of reduce phase to meet the performance difference of heterogeneous clusters. PIKACHU focuses on achieving optimal workload balance for Hadoop [14]. It presents guidelines for the trade-offs between the accuracy of workload balancing and the delay of workload adjustment.

The shuffle and reduce in Hadoop also have a large space for improvement. MapReduce Online [12] proposed a push-based shuffle mechanism to support the online aggregation and continuous queries. MaRCO [6] overlaps the reduce and shuffle. But the early start of reduce generates partial reduces which could be the source of overhead for some applications. Hadoop Acceleration [28] proposed a different approach to mitigate shuffle delay and repetitive merges in Hadoop. It implemented a hierarchical merge algorithm based on remote disk access and eliminated the explicit copying process in shuffle. However, this approach relies on the RDMA feature of Infiniband network, which is not available on commodity network hardware. Wang *et al.* [29] developed JVM-Bypass Shuffling (JBS). JBS avoids the overhead of JVM in data shuffling and enables fast data movement on both RDMA and TCP/IP protocols. CooMR [20] is a cross-task coordination framework for efficient data management in MapReduce. It enables cross-task opportunistic memory sharing and log-structured I/O consolidation, which are designed to facilitate task coordination. Its key-based in-situ merge algorithm allows the sorting/merging of Hadoop intermediate data without actually moving the key-value pairs. Sailfish [23] proposed a novel approach to improve the shuffle performance with I-File. I-File is implemented on Kosmos File System (KFS) to store the intermediate data of jobs. The data is automatically merged to an I-File. It significantly reduces the shuffle delay. However, it does not remove the map-side combine, which still has extra disk accesses.

There are few recent studies focusing on improving the performance of Hadoop with different file systems [21, 26]. QFS [21] is a distributed file system designed to replace HDFS. It improves the efficiency of the distributed file system. It uses erasure coding instead of replication, which enables QFS to provide the same data availability with less disk space. Tantisiriroj *et.al.* proposed to use Parallel Virtual File System (PVFS) to replace HDFS [26]. However, these file systems are not optimized towards the virtualized Hadoop cluster. They shared the same issues that HDFS has.

Themis [24] is a system that focuses on providing I/O efficient MapReduce. It proposed a different design that batches the disk I/O operation and reduces the disk seeks. Themis also provides an adaptive memory allocation to dynamically change the sort buffer to reduce the disk I/O during merge/shuffle. The objective is close to StoreApp, but it lacks of host-awareness for virtualized environment. The virtualization of slave node disks can reduce the efficiency of Themis. Themis focuses on the I/O efficiency of intermediate storage. It does not improve the efficiency of HDFS.

7. CONCLUSIONS

Hadoop provides an open-source implementation of the MapReduce framework. But its design poses challenges to attain the best performance in the virtualized environment due to the coupled computation and storage nodes. Separating the storage from slave nodes enables flexible cluster resizing and allows task scheduler to exploit the host-locality. In this paper, we propose and implement *StoreApp*, a shared storage appliance for virtualized Hadoop cluster. It provides both HDFS and intermediate storage for co-hosted slave nodes on the same physical machine. StoreApp provides an easy way for Hadoop to discover the VM-host topology and exploit the host-locality. StoreApp introduce prefetching to the HDFS and improves the efficiency of HDFS. StoreApp implements automated cluster resizing to determine the optimal cluster size for different jobs. We modified the Hadoop Fair Scheduler with the support to StoreApp and evaluated its effectiveness on a 22-node virtual cluster with various workloads. Experimental results show that StoreApp is able improve the average HDFS throughput by as much as 15% and reduce job completion time by as much as 61% than stock Hadoop. Our future work will be on implementing StoreApp with automated cluster resizing for heterogeneous environments.

8. REFERENCES

[1] Apache Hadoop Project. http://hadoop.apache.org.

[2] PUMA: Purdue mapreduce benchmark suite. http://web.ics.purdue.edu/~fahmad/benchmarks.htm.

[3] A Benchmarking Case Study of Virtualized Hadoop Performance on VMware vSphereÂő 5. http://www.vmware.com/files/pdf/techpaper/VMW-Hadoop-Performance-vSphere5.pdf.

[4] A Benchmarking Case Study of Virtualized Hadoop Performance on VMware vSphereÂő 5. http://www.vmware.com/files/pdf/VMW-Hadoop-Performance-vSphere5.pdf.

[5] F. Ahmad, S. T. Chakradhar, A. Raghunathan, and T. N. Vijaykumar. Tarazu: optimizing mapreduce on heterogeneous clusters. In *Proc. of the Int'l Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2012.

[6] F. Ahmad, S. Lee, M. Thottethodi, and T. N. Vijaykumar. [inpress]mapreduce with communication overlap (marco). *Journal of Parallel and Distributed Computing*, 2012.

[7] Amazon Elastic MapReduce (EMR). https://aws.amazon.com/elasticmapreduce/.

[8] Big data in the cloud: Converging technologies. http://www.intel.com/content/www/us/en/big-data/big-data-cloud-technologies-brief.html.

[9] R. Birke, A. Podzimek, L. Y. Chen, and E. Smirni. State-of-the-practice in data center virtualization: Toward a better understanding of vm usage. In *IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, 2013.

[10] D. E. Bogdan Ghit, Nezih Yigitbasi. Resource management for dynamic mapreduce clusters in multicluster systems (best paper award). In *5th Workshop on Many-Task Computing on Grids and Supercomputers (MTAGS) co-located with Supercomputing (SC)*, 2012.

[11] X. Bu, J. Rao, and C.-Z. Xu. Interference and locality-aware task scheduling for mapreduce applications in virtual clusters. In *Proc. of the ACM Int'l Symposium on High-Performance Parallel and Distributed Computing (HPDC)*, 2013.

[12] T. Condie, N. Conway, P. Alvaro, J. M. Hellerstein, K. Elmeleegy, and R. Sears. MapReduce online. In *Proc. of the USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2010.

[13] J. Dean and S. Ghemawat. MapReduce: simplified data processing on large clusters. In *Proc. of the USENIX Symposium on Operating System Design and Implementation (OSDI)*, 2004.

[14] R. Gandhi, D. Xie, and Y. C. Hu. PIKACHU: How to rebalance load in optimizing mapreduce on heterogeneous clusters. In *Proc. of the USENIX conference on Annual Technical Conference (ATC)*, 2013.

[15] Y. Guo, J. Rao, and X. Zhou. iShuffle: Improving hadoop performance with shuffle-on-write. In *Proc. of the USENIX Int'l Conference on Autonomic Computing (ICAC)*, 2013.

[16] Hadoop Virtualization Extensions on VMware vSphereÂő 5. https://www.vmware.com/files/pdf/Hadoop-Virtualization-Extensions-on-VMware-vSphere-5.pdf.

[17] H. Herodotou, F. Dong, and S. Babu. No one (cluster) size fits all: automatic cluster sizing for data-intensive analytics. In *Proc. of the ACM Symposium on Cloud Computing (SOCC)*, 2011.

[18] Y. Kwon, M. Balazinska, B. Howe, and J. Rolia. Skew-resistant parallel processing of feature-extracting scientific user-defined functions. In *Proc. of the ACM Symposium on Cloud Computing (SOCC)*, 2010.

[19] Y. Kwon, M. Balazinska, B. Howe, and J. Rolia. SkewTune: Mitigating skew in mapreduce applications. In *Proc. of the ACM SIGMOD*, 2012.

[20] X. Li, Y. Wang, Y. Jiao, C. Xu, and W. Yu. CooMR: Cross-task coordination for efficient data management in mapreduce programs. In *Proc. of Int'l Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 2013.

[21] M. Ovsiannikov, S. Rus, D. Reeves, P. Sutter, S. Rao, and J. Kelly. The quantcast file system. *Proc. VLDB Endowment*, 6(11), Aug. 2013.

[22] Rackspace Cloud Big Data Platform. http://www.rackspace.com/cloud/big-data/.

[23] S. Rao, R. Ramakrishnan, A. Silberstein, M. Ovsiannikov, and D. Reeves. Sailfish: A framework for large scale data processing. In *Proc. of the ACM Symposium on Cloud Computing (SOCC)*, 2012.

[24] A. Rasmussen, V. T. Lam, M. Conley, G. Porter, R. Kapoor, and A. Vahdat. Themis: an i/o-efficient mapreduce. In *Proc. of the ACM Symposium on Cloud Computing (SOCC)*, 2012.

[25] M. A. Shah, J. M. Hellerstein, and E. Brewer. Highly available, fault-tolerant, parallel dataflows. In *Proc. of the ACM SIGMOD*, 2004.

[26] W. Tantisiriroj, S. W. Son, S. Patil, S. J. Lang, G. Gibson, and R. B. Ross. On the duality of data-intensive file system design: Reconciling hdfs and pvfs. In *Proc. of Int'l Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 2011.

[27] V. K. Vavilapalli, A. C. Murthy, C. Douglas, S. Agarwal, M. Konar, R. Evans, T. Graves, J. Lowe, H. Shah, S. Seth, B. Saha, C. Curino, O. O, Malley, S. Radia, B. Reed, and E. Baldeschwieler. Apache Hadoop YARN: Yet another resource negotiator. In *Proc. of the ACM Symposium on Cloud Computing (SOCC)*, 2013.

[28] Y. Wang, X. Que, W. Yu, D. Goldenberg, and D. Sehgal. Hadoop acceleration through network levitated merge. In *Proc. of Int'l Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 2011.

[29] Y. Wang, C. Xu, X. Que, X. Li, and W. Yu. Bypass-JVM shuffling for hadoop acceleration. In *Proc. IEEE Int's Parallel Distributed Processing Symposium(IPDPS)*, 2013.

[30] M. Zaharia, A. Konwinski, A. D. Joseph, R. Katz, and I. Stoica. Improving mapreduce performance in heterogeneous environments. In *Proc. of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2008.