Journal of Circuits, Systems, and Computers
Vol. 29, No. 16 (2020) 2050254 (24 pages)
© World Scientific Publishing Company
DOI: 10.1142/S0218126620502540

Application and Storage-Aware Data Placement and Job Scheduling for Hadoop Clusters*

Tao Li
College of Computer Science and Electronic Engineering,
Hunan University, Changsha, P. R. China
litao1310@hnu.edu.cn

Shuibing He†, Ping Chen‡ and Siling Yang§
College of Computer Science and Technology,
Zhejiang University, Hangzhou, P. R. China
†*heshuibing@zju.edu.cn*
‡*zjuchenping@zju.edu.cn*
§*slingzjunet@zju.edu.cn*

Yanlong Yin
Intelligent Computing System Research Center,
Institute of Artificial Intelligence,
Zhejiang Lab, Hangzhou, P. R. China
yyin@zhejianglab.com

Cheng Xu
College of Computer Science and Electronic Engineering,
Hunan University, Changsha, P. R. China
cheng_xu@yeah.net

Received 20 December 2019
Accepted 17 March 2020
Published 21 December 2020

As one of the most popular frameworks for large-scale analytics processing, Hadoop is facing two challenges: both applications and storage devices become heterogeneous. However, existing data placement and job scheduling schemes pay little attention to such heterogeneity of either application I/O requirements or I/O device capability, thus can greatly degrade system efficiencies. In this paper, we propose ASPS, an Application and Storage-aware data Placement and job Scheduling approach for Hadoop clusters. The idea is to place application data and schedule application tasks considering both application I/O requirements and storage device characteristics. Specifically, ASPS first introduces novel metrics to quantify I/O requirements of applications. Then, based on the quantification, ASPS places data of different applications to

*This paper was recommended by Regional Editor Tongquan Wei.
† Corresponding author.

the preferred storage devices. Finally, ASPS tries to launch jobs with high I/O requirements on the nodes with the same type of faster devices to improve system efficiency. We have implemented ASPS in Hadoop framework. Experimental results show that ASPS can reduce the completion time of a single application by up to 36% and the average completion time of six concurrent applications by 27%, compared to existing data placement policies and job scheduling approaches.

Keywords: Hadoop; MapReduce; HDFS; data placement; job scheduling; SSDs.

1. Introduction

Over the past decade, MapReduce[1] has become one of the most popular parallel frameworks for big data analytics processing. Hadoop[2] is the de facto open-source implementation of MapReduce framework, which consists of Hadoop MapReduce, Hadoop YARN, and Hadoop Distributed File System (HDFS).[3] The Apache community has developed the Hadoop ecosystem to address big-data challenges. Previous study[4] has shown that more than 30% of the companies had already deployed Hadoop. Hadoop has attracted much attention not only in industry but also in academia.

Many large-scale data analytics applications are I/O-intensive. For example, workload analysis from Facebook and Microsoft shows that 79% of job execution time is spent on I/O operations.[5] To accommodate the large volumes of data from such applications, Hadoop leverages HDFS to provide efficient data storage services. In addition to Hadoop MapReduce, HDFS is also used in the in-memory data processing framework Spark[6] as well as Hadoop database HBase.[7] However, while HDFS offers promising I/O potential for data-intensive applications, current Hadoop (version 3) is still facing the following challenges due to recent application and storage trends in data centers.

First, Hadoop clusters face multiple heterogeneous applications, which may degrade system performance. For better system resource utilization, Hadoop clusters are commonly shared by multiple applications concurrently.[8–11] Due to their inherent natures, applications may have various I/O requirements to process data (Sec. 2). Unfortunately, current Hadoop cannot intelligently recognize such requirement heterogeneity and still allocates storage resources equally to each application. As a result, some applications may need less bandwidth/capacity than they are assigned while others need more, leading to sub-optimal system performance.

Second, current Hadoop systems host increasingly heterogeneous storage devices, which can also offset system efficiency. To achieve high throughputs with acceptable costs, each node is evolving to heterogeneous storage devices, e.g., hard disk drives (HDDs) and solid-state drives (SSDs).[12,13] However, HDFS is initially designed for homogeneous devices, and thus, it places data uniformly across nodes regardless of device characteristics. Consequently, the tasks of a job on the nodes with faster SSDs will often execute quickly and wait for the tasks on the nodes with HDDs. This task skewness will under-utilize the potential of SSDs. While storage heterogeneity could occur in different fashions, in this paper, we focus on the case where each node hosts an HDD and an SSD.

Several previous approaches address the heterogeneity issues by optimizing data placement of Hadoop systems.[4,13–16] Other efforts improve Hadoop throughput by designing new task schedulers.[17–21] Also some approaches tackle the issues with integrated data placement and job scheduling optimization.[22] However, we note that none of them are efficient for Hadoop clusters where applications have heterogeneous I/O requirements and each node hosts heterogeneous storage devices.

In this study, we propose ASPS, an Application and Storage-aware data Placement and job Scheduling approach for Hadoop clusters. Besides data locality in existing approaches, ASPS places application data and schedules application tasks considering application I/O requirements and storage device characteristics. Specifically, ASPS proposes two metrics to quantify the I/O requirements of the map and the reduce task of an application. Then, based on the quantification, ASPS places application data on the proper devices to make full utilization of storage devices. The optimized data placement involves the input data, the intermediate data, and the output data of an application. Finally, ASPS tries to schedule tasks of an application on the same type of nodes to alleviate task skewness within a job execution. It also preferentially schedules applications with higher I/O requirements on the nodes with faster storage devices to enhance system efficiency for multiple jobs. We have implemented ASPS in Hadoop system and demonstrated its advancements through extensive experimental tests on six benchmark applications. Experimental results show that ASPS can significantly reduce application completion time and achieve better cost-effectiveness on storage-heterogeneous Hadoop clusters.

Specifically, this study makes the following contributions:

- We present two new metrics, which can be measured with a simple sampling approach, to respectively quantify the I/O requirements of the map task (mapper) and the reduce task (reducer) of a MapReduce application.
- Based on the metrics, we design and develop ASPS, an application and storage-aware data placement and task scheduling approach to enhance Hadoop system performance.
- We implemented ASPS in the Hadoop framework. Experimental results with extensive tests on six benchmark applications validate the effectiveness of ASPS.

The remainder of this paper is organized as follows. In Sec. 2, we describe the background and motivation. In Secs. 3 and 4, we propose the design and implementation of ASPS, respectively. Section 5 presents the evaluation results. Section 6 discusses the related work. Finally, we conclude this paper in Sec. 7.

2. Background and Motivation

2.1. *Hadoop data placement and job scheduling*

Hadoop is an open-source implementation of the MapReduce framework. The first version of Hadoop[1] is composed of Hadoop runtime subsystem that executes the MapReduce applications, and HDFS that provides data storage. The runtime system includes one JobTracker and multiple TaskTrackers. The JobTracker schedules tasks onto the TaskTrackers. The Hadoop 2.0/3.0 version has a new module called YARN, which separates the role of JobTracker into two parts: ResourceManager and AppMaster. These two parts manage system resources and monitor task execution, respectively. There are multiple nodes in the Hadoop system. A small number of nodes are connected to form a rack. A Hadoop cluster may consist of one or more racks.

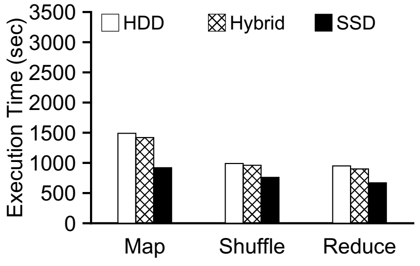
HDFS[3] is composed of one NameNode that manages file metadata and multiple DataNodes that store file data. To provide high throughput, HDFS divides all files into fixed-size blocks (e.g., 64 MB) and places them uniformly across various DataNodes. Moreover, to provide resilience against both node and rack failures, HDFS defaults to maintain three replicas for each block and places them on DataNodes in a *network-aware* fashion: one is placed on the local node where the client is running, one on a node in another rack, and one on a different node in the remote rack. Each input data block corresponds to one map task (mapper). When a job begins to run, the scheduler (Capacity Scheduler) uses a network-aware policy to assign each map task on a DataNode where the nearest replica of the required data resides. These network-aware policies can significantly improve data locality and reduce data transfer time on the network.

As described in Sec. 1, the original Hadoop implementation regards all applications and storage devices to be homogeneous. While recent versions of Hadoop have supported heterogeneous storage devices,[23–25] they only provide interfaces for users to choose the desired device types rather than intelligently determining the preferred devices and scheduling the jobs in the system. When various applications and storage devices occur, existing Hadoop systems may lead to sub-optimal system performance.

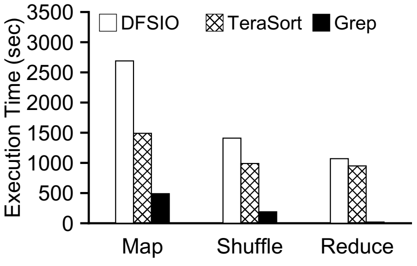
2.2. *Motivating examples*

To highlight these issues, we measure the Hadoop system performance in two different scenarios: homogeneous applications on heterogeneous devices and heterogeneous applications on homogeneous devices. We use three PUMA benchmark applications,[26] i.e., DFSIO, TeraSort, and Grep, each with 40 GB input data.

Heterogeneous device impact. Figure 1(a) shows the completion times of different phases of TeraSort on three storage configurations. The HDD, SSD, and Hybrid configurations refer to 8 HDD DataNodes, 8 SSD DataNodes, and 4 HDD+4



(a) Heterogeneous devices

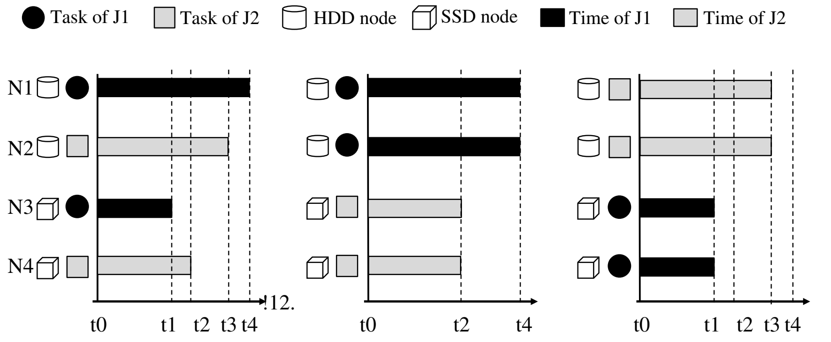


(b) Heterogeneous applications

Fig. 1. The system performance in heterogeneous environments.

SSD DataNodes, respectively. As shown, for all the three phases, i.e., Map, Shuffle, and Reduce phase, Hybrid configuration slightly outperforms the homogeneous HDD cluster and is much worse than the homogeneous SSD configuration. This shows the potential of SSD DataNodes is substantially under-utilized in storage-heterogeneous environments.

The reason is that heterogeneous devices lead to skewed task execution, which offsets application performance. To illustrate this issue, we use Fig. 2(a) as an example to describe job execution on a storage-heterogeneous cluster. There are two jobs, J1 and J2, running on four nodes, N1 to N4. Each job represents an application. For J1, its two input blocks (the nearest replicas) are placed on N1 and N3, which have an HDD and an SSD device, respectively. With the default policy, two tasks of J1 are scheduled on N1 and N3. Because an SSD has higher I/O performance than an HDD, N3 finishes its task more quickly than N1. However, J1's completion time is t_4 because a job is not completed until the slowest task complete.[5] This is called the "*all-or-nothing*" property of job execution, which offsets SSD potential, although it has superior performance. The case is similar for job execution of J2.



(a) Default policy (b) Storage-aware policy (c) Cost-effective policy

Fig. 2. Job execution under different policies in Hadoop system.

To overcome the above issue, an alternative approach is to place data of a job in the same type of devices and schedule its tasks from the same type of nodes. We refer this policy to *storage-aware* policy. As shown in Fig. 2(b), the input blocks of J1 are placed on HDD nodes, namely N1 and N2, and those of J2 are on SSD nodes, namely N3 and N4. During job execution, J1 is scheduled on HDD nodes and J2 on SSD nodes. As a result, the job completion times of J1 and J2 are t_4 and t_2, respectively. Compared to the default policy, the storage-aware policy reduces the job completion time from $t_3 + t_4$ to $t_2 + t_4$.

Heterogeneous application impact. Figure 1(b) shows the completion times of the three heterogeneous applications with the HDD configuration. One can observe that these times in the three phases vary significantly for the three applications. The reason is that applications access the data with different I/O requirements. Usually, each task of an application handles a data block consisting of several records. The number of records, the record size, and the read/write intensity can vary greatly from application to application. This means allocating enhanced resource to different applications results in various performance benefits. For example, the Mapper execution times of TeraSort and Grep are reduced by 1.6X and 1.1X, respectively, when running them on SSD nodes compared to running them on HDD nodes. Therefore, such storage-aware policy may still be sub-optimal because it ignores application characteristics. We will illustrate this in Sec. 3.1 later.

Summary. The above studies show that both heterogeneous devices and heterogeneous applications can substantially impact Hadoop system efficiency. This motivates us to propose ASPS to tackle these challenges.

3. ASPS Design

ASPS aims to enhance Hadoop with a cooperative data placement and job scheduling approach. In this section, we first introduce the idea of ASPS and then describe its architecture, followed by the details of each of its key components.

3.1. *Idea of ASPS*

Since applications own different I/O requirements and storage devices provide various I/O capacities, ASPS places application data and schedules their tasks considering both application and device characteristics. ASPS centers on two key designs: (1) it tries to place an application's data on the same type of devices like in the storage-aware policy. Moreover, as applications with high I/O requirements will bring significant performance benefits from fast devices, it prioritizes to place data of such applications on faster devices, i.e., SSDs; (2) it preferentially schedules the tasks of applications with high I/O requirements on the nodes with the same type of faster devices.

Figure 2(c) shows the idea of ASPS, which we call a *cost-effective* policy. In the example, as J1 has higher I/O requirement than J2, the blocks of J1 are placed on SSD nodes (N3 and N4) and those of J2 are on HDD nodes (N1 and N2). Moreover, by scheduling J1 and J2 on the same type of SSD nodes and HDD nodes, respectively, the overall job completion time is $t_1 + t_3$, which is smaller than $t_2 + t_4$, namely the time achieved by the storage-aware policy in Fig. 2(b).

The proposed approach requires prior knowledge of applications' I/O requirements. Fortunately, periodic jobs make up the majority of cluster workloads, as reported by Refs. 27 and 28. This provides an opportunity to obtain the I/O requirements of application via I/O sampling techniques.

3.2. *System architecture*

Figure 3 shows the system architecture of ASPS. In this system, we assume that each node in the cluster is equipped with homogeneous CPU and memory resource but with heterogeneous storage devices: an HDD and an SSD. There are concurrent jobs (applications) submitted to the cluster. To enable the proposed application and storage-aware data placement and job scheduling approach, ASPS includes the following three components.

- **The I/O quantifier.** It uses a simple but effective sampling technique to quantify the I/O requirements of applications. Based on these metrics, ASPS makes optimized data placement and task scheduling.
- **The data placer.** It places application data on the proper devices based on their I/O requirements. The data include the input data, the intermediate data, and the output data of the application in the I/O path of the job execution.
- **The job scheduler.** It makes efficient scheduling for concurrent jobs by considering application I/O requirements and storage device types.

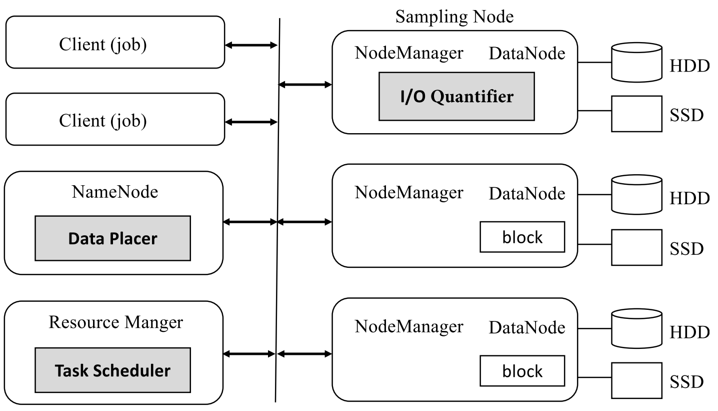


Fig. 3. The system architecture of ASPS.

The above components operate as follows. First, the quantifier recognizes the I/O requirements of applications and then forwards these requirements to the NameNode. Then, when the input data of applications are imported into HDFS, the data placer makes careful data placement on the proper storage devices based on the application's I/O requirements. Finally, when multiple jobs are concurrently submitted to the system, the scheduler allocates the jobs to the proper nodes based on application and device characteristics. This process is repeated until all jobs are completed.

3.3. *I/O requirement quantification*

To enable the proposed data placement and task scheduling approach, we need to quantify the I/O requirements of applications. These requirements are highly application-dependent, thus the measurements change for different applications. Since map tasks and reduce tasks may require various I/O operations (Sec. 2.2), we further quantify the I/O requirements of mappers and reducers, respectively.

Inspired by previous studies that use disk-based sampling technique to learn the features of MapReduce applications,[29] we propose a disk-based sampling approach to measure the I/O requirements of mappers and reducers. We use an in-disk directory as the local HDFS storage and capture the I/O timing information and the input/output data amounts of applications.

For mappers, we use one tailored map task and run it on a dedicated node to process a split that is approximately a block. We measure the split size and the mapper I/O times on different storage devices. Assume the split size is B, the I/O times on an HDD and an SSD device are T_m^h and T_m^s, respectively; then the I/O requirement of the mapper is defined as

$$r_m = \frac{T_m^h - T_m^s}{B} \,. \tag{1}$$

For reducers, they will read input data from the outputs of mappers, which are expected to be materialized in the local file system. Assume the output size of a mapper is D, then the input size of a reducer is $\frac{D\times m}{r}$, where m and r mean the number of mappers and reducers, respectively. We also measure the reducer I/O times on different storage devices. Assume the I/O times on an HDD node and an SSD node are T_r^h and T_r^s, then the I/O requirement of the reducer is

$$r_r = \frac{r \times (T_r^h - T_r^s)}{D \times m} \,. \tag{2}$$

Both r_m and r_r reflect the I/O time reduction brought by per unit of fast storage resource. The larger the values of r_m and r_r, the higher the I/O requirements of the mapper and the reducer.

Note that r_m and r_r are efficient for all mappers and reducers only when they have uniform workloads. For example, all mappers have a uniform number of input and output key-value pairs, and the number of values for each output key is similar. In this case, r_m is relatively stable in terms of different sampling split sizes and mappers, and the mapper completion time is often linearly proportional to the input size.[16] As such, we can quantify the requirements of all mappers with a sampling task and a split. However, this assumption does not always hold true for some nontrivial applications in real-world scenarios. To address this issue, we run the sampling procedure with multiple randomly selected splits and choose the averages as the final metrics of r_m and r_r, respectively.

The quantifier identifies I/O requirements of all applications one by one. Once this process is over, it will store the obtained information into a global requirement table, which is used by the placer and scheduler later. Note that the two metrics only consider limited parameters in characterizing I/O accesses. There are other factors that affect the application's I/O performance. However, such metrics account for the major I/O access features. As shown in the evaluation, they are a reliable indicator to direct data placement and job scheduling.

3.4. *Data placement*

With the proposed metrics in Sec. 3.3, the placer distributes application data on the cluster with the following rules. (1) It tries to place the data of an application from the same type of devices as in the storage-aware policy. (2) As applications with high I/O requirements will bring significant performance benefits from fast devices, it tries to place data of such applications on SSD nodes in a high priority. Specifically, if the application has high I/O requirements (a high value of r_m) and the SSDs have enough space, the placer will store two of the replicas of each block on SSDs and one replica on HDDs. Otherwise, the placer will store all the replicas on HDDs. This policy tries to allocate more space for applications with high I/O requirements so that SSDs can be efficiently used.

The default HDFS places three replicas of a block on DataNodes in a network-aware fashion (Sec. 2). Note that the placer does not break this policy: it first uses the network-aware policy to find the physical node that hosts an HDD and an SSD, and then it adopts the above rules to place the replica on the ultimate device, an SSD or an HDD, according to application characteristics. To shorten the write response time of a client, the placer stores the primary replica of the block on SSDs if it is possible.

Besides the input data of map tasks, the placer also optimizes the data placement of the intermediate data. We do this because the intermediate data materialization is an I/O-intensive procedure and it can substantially impact overall job execution.[30,31] Figure 4 shows the typical I/O path of Hadoop application. We adopt the following rules to optimize the placement of intermediate data on local devices. For map tasks, if the value of r_m is high and the local SSD has enough free space, we store the

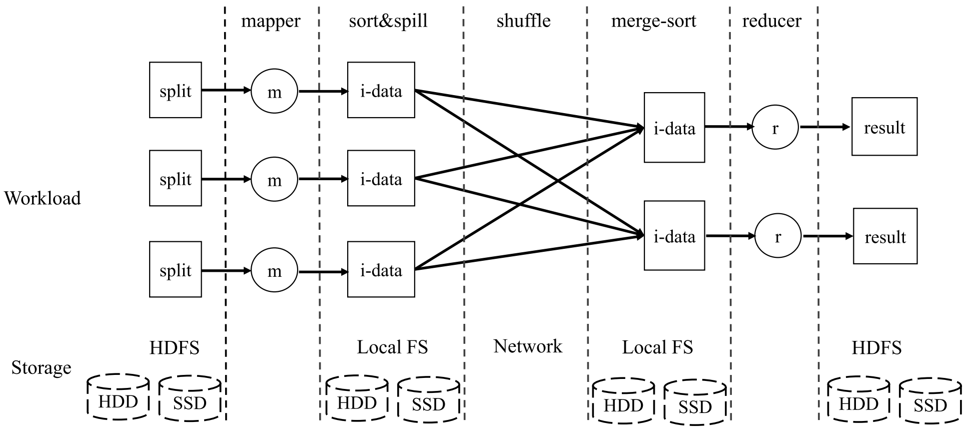


Fig. 4. Typical I/O path in our system.

outputs of the map tasks in the local SSDs. Otherwise, the data will be stored on local HDDs. For reduce tasks, we use similar rules to place their input data except that r_m changes to r_r.

Finally, the placer also optimizes the data placement of the output data of the application. Similar to the rules for the input data of map tasks, we store the output data of reduce tasks in the HDFS file system as following. If the value of r_r is high and SSDs have enough space, we will store two replicas of each block in the output file on SSD nodes and one replica on an HDD node. Otherwise, we will store all the replicas on HDD nodes.

3.5. *Job scheduling*

Based on the proposed data placement policy and I/O metrics, the scheduler assigns multiple jobs to the nodes in the cluster as Algorithm 1. The key idea consists of two aspects. First, considering the *all-or-nothing* property within a single job execution, the algorithm tries to launch tasks of each job on the nodes with the same types of devices. Second, to achieve the cost-effectiveness of multi-job execution, the algorithm schedules the tasks of application with high I/O requirements on SSD nodes to utilize the potential of SSDs.

To this end, the algorithm leverages two helper threads to schedule the tasks of multiple jobs in the system. One thread is responsible for scheduling map tasks and the other for reduce tasks. The two threads are synchronized by a waiting job set ($\mathcal{J}_w$) in the current sliding time window. Each thread uses its own I/O metric, i.e., r_m or r_r, to determine if all tasks of a job in a wave can get the same I/O resource. If yes, the corresponding resources, including CPU, memory, and storage device, are allocated to the job, and this job will be scheduled with these resources. Otherwise, the job is scheduled as the default policy as long as the system has available resources for

Algorithm 1. Cost-effective job scheduler.

```
1:  procedure JOB_SCHEDULING($j$, $R$)
2:      $m \leftarrow$ number of map tasks of job $j$
3:      $r \leftarrow$ number of reduce tasks of job $j$
4:      $\mathcal{J}_w \leftarrow$ waiting job set in current sliding window
5:      $\mathcal{J}_a \leftarrow$ active job set in current sliding window
6:      $r_m \leftarrow$ map task I/O requirement of job $j$
7:      $r_r \leftarrow$ reduce task I/O requirement of job $j$
8:      $\mathcal{J}_w \leftarrow \mathcal{J}_w + j$
9:
10:     while $\mathcal{J}_w \neq 0$ do                                   ▷ Thread 0
11:         pick a job $i$ with the highest $r_m$
12:         if all its map tasks can be assigned on SSD nodes then
13:             allocate the resource on SSD nodes to job $i$
14:         else
15:             allocate the resource as default policy to job $i$
16:         end if
17:         update current available resource $R$
18:         $\mathcal{J}_w \leftarrow \mathcal{J}_w - i$
19:         $\mathcal{J}_a \leftarrow \mathcal{J}_a + i$
20:     end while
21:
22:     while $\mathcal{J}_a \neq 0$ do                                   ▷ Thread 1
23:         pick a job $k$ with the highest $r_r$
24:         if all its reduce tasks can be assigned to SSD nodes then
25:             allocate the resource on SSD nodes to job $k$
26:         else
27:             allocate the resource as default policy to job $k$
28:         end if
29:         update current available resource $R$
30:         $\mathcal{J}_a \leftarrow \mathcal{J}_a - k$
31:     end while
32: end procedure
```

the current wave of job execution. Each thread uses a loop to schedule all tasks in the job set until all jobs are completed.

For a single job, as all of its tasks may be launched onto the nodes with the same type of devices, the task skewness within the job execution will be substantially alleviated. For multiple jobs in the system, as those jobs with high I/O requirements are preferentially executed on the nodes with fast SSDs, the overall system efficiency will be improved as the high-performance SSDs are used in a more efficient fashion.

Note that this algorithm does not break the data locality in traditional network-aware scheduler: it first tries to use the network-aware policy to find the physical node that hosts the required data, then it adopts the above rules to schedule the jobs on the node by accessing the ultimate device, an SSD or an HDD, according to application characteristics. Moreover, the algorithm is not only limited by one-wave task execution, but it also applies to multi-wave workloads, namely large jobs.

4. Implementation

We implemented ASPS under Hadoop version 3.0.3. Figure 5 shows the overview of the system implementation, which includes three phases. In the sampling phase, the map task of each application is profiled on a sampling node to measure application I/O requirements. In the data placement phase, the NameNode guides the data placements of each application on storage devices according to the collected I/O requirements. In the scheduling phase, the scheduler assigns tasks to the proper nodes based on the rating information.

4.1. *Heterogeneous device management*

To manage the heterogeneous storage within a single physical node, we run two DataNodes simultaneously on the same machine. We specify different server addresses and ports in the configuration file to differentiate the DataNodes. Meanwhile, we configure the *dfs.data.dir* to an HDD and SSD-mounted directory to access the HDD and SSD devices, respectively. The start-up script of HDFS is also modified to make the DataNodes start with different configuration files. The NodeManager,

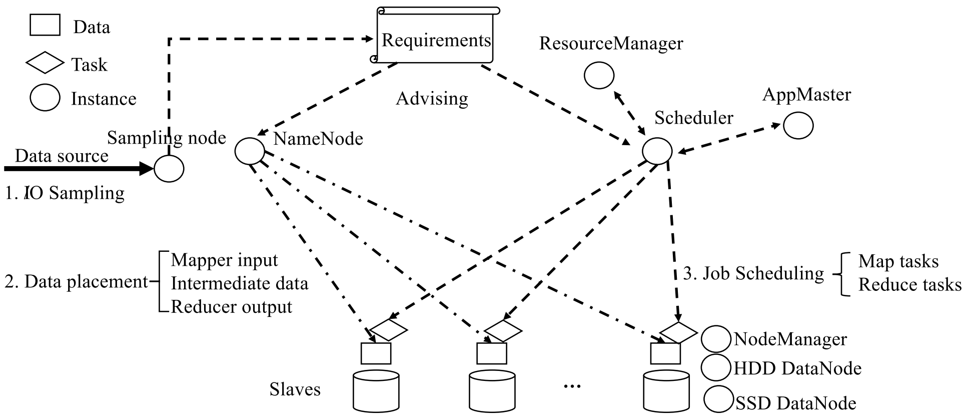


Fig. 5. System Implementation overview of ASPS.

which manages the task running on its hosting node, is the same as the conventional YARN deployment. However, it interacts with both DataNodes.

To enable our proposed schemes, the ResourceManager or Scheduler in YARN needs to know the storage device information in each NodeManager. However, existing computing resources in YARN are abstracted by container, which mainly include CPU and memory resources without storage information. To overcome this issue, we encode the storage information into the port number of the NodeManager, such that the ResourceManager can recognize what the storage device the NodeManager owns according to the port number.

4.2. *I/O metric sampling*

Before importing the input data of applications into HDFS, we use a dedicated machine as a sampling agent to measure their I/O requirements. As we discussed in Sec. 3.3, we use disk-based storage to quantify application I/O features. This can be achieved by specifying the data storage path to an HDD or SSD-based local file system. We tailor the *copyFromLocal* approach to load one or more data blocks into the memory. We run a customized map task to generate the desired metrics, i.e., r_m and r_r. Specifically, we add timing information at the beginning and the ending of the map task execution in MapRunner and record the map output size.

The sampling information is kept in a data structure in NetworkTopology instance. This information will be periodically delivered to the NameNode and the task scheduler via customized RPCs.

4.3. *Data placement*

To place input data blocks on desired devices of DataNodes, we modify the block placement function executed by NameNode. The existing data placer chooses DataNodes in a network-aware fashion by considering the NetworkTopology, where leaf nodes represent data nodes while intermediate nodes denote routers of a rack. To make the data placement policy application- and storage-aware, we hack the NetworkTopology with storage device information. We then intercept the DataNode choosing function and choose the desired DataNode by the IP address and port number. Similar to the default data placement policy, we avoid allocating two replicas of a block on the same physical node.

By default Hadoop stores the intermediate data in the path specified by *hadoop.tmp.dir* in the pre-configured *mapred-site.xml* file. However, all applications in the system share the same configuration. To store the intermediate data on the desired type of devices, we modify the implementation of the AppMaster and the ResourceManager. When the AppMaster executes its ServiceInit procedure, we capture the appName and put it into the PRC issued to the ResourceManager. The ResourceManager determines the data store paths for the intermediate data according to the appName and sends this information to the AppMaster. Once the AppMaster receives the feedback from the ResourceManager, it knows the application-specific path to store the intermediate data for map task outputs and reduce task inputs.

For the outputs of reduce tasks, we use a similar method as in the input placement of map tasks to distribute their data on the specific DataNodes in the HDFS file system.

4.4. *Task scheduling*

A computing resource in YARN is denoted by a container. The ResourceManager manages all resources in the cluster. When a job is submitted to the system, the scheduler in ResourceManager will allocate resources to the AppMaster, which is in charge of executing the tasks of the job. The AppMaster asks the ResourceManager for desired resources, including resource priorities, desired nodes, the number of containers, and resource types. The default scheduler in YARN is Capacity Scheduler, which tries to fulfill the container requests from the AppMaster following a node-local, rack-remote, and rack-remote order. However, it does not consider the storage heterogeneity information.

To schedule tasks cost-effectively, we modify the Capacity Scheduler to meet the I/O requirement of a particular application with a high priority. We add a control switch in the container selection strategy to choose the DataNode with a specific storage device. In the system, each NodeManager periodically reports its resource information to ResourceManager via heartbeats. ResourceManager acknowledges back a heartbeat, including container releasing information. Since the container has application and task information, the ResourceManager learns if it can get the preferred resource according to the port numbers of the NodeManagers.

5. Evaluation

In this section, we first evaluate ASPS in a storage-heterogeneous cluster for a single application. Then, we further prove the efficiency of ASPS in the storage-heterogeneous cluster for multiple heterogeneous applications.

5.1. *Experimental setup*

We conduct experiments on a 17-node Linux cluster, including 1 head node and 16 computing nodes. The head is equipped with dual 2.7 GHz Opteron quad-core processors, 8 GB memory and a RAID5 array. Each computing node has two Opteron quad-core processors, 8 GB memory, a 250 GB HDD, and a 100 GB SSD. Gigabit Ethernet connects all nodes. We use the head node as Master and the 16 computing nodes as Slaves. The operating system on each node is Ubuntu 13.04. The Hadoop version is 3.0.3.

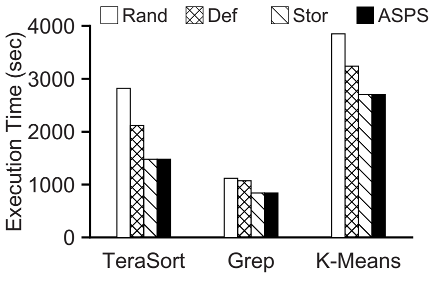
We compare ASPS with three other schemes: Def is the default policy in Hadoop, which only considers network characteristics (or data locality) but ignores storage and application information; Stor accounts for both storage and network but is unaware of application characteristics; Rand is totally random and oblivious of all information. Both Def and Stor are discussed in Sec. 3. In contrast, ASPS considers network, storage, and application factors. By default, the replication factor in HDFS is three and the block size is 64 MB. The input size is 128 GB for each application and the number of reducers is 4.

5.2. *Evaluation with a single application*

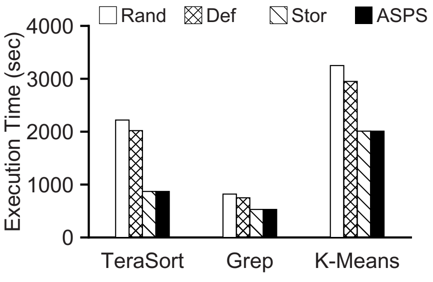
In this set of experiments, we use three MapReduce applications, i.e., TeraSort, Grep, and K-Means from the PUMA benchmark.[26] To evaluate ASPS for a single application, we conduct the experiments by running each benchmark, respectively.

Mapper input placement and scheduling optimization. Figure 6 shows the application performance when only the mapper input data placement and map task scheduling are enabled in ASPS. As shown in Fig. 6(a), Rand has the longest completion times because it does not consider network, storage, and application characteristics. Compared to Rand and Def, ASPS reduces the job completion times of TeraSort, Grep, and K-Means by 47% and 31%, 25% and 21%, and 19% and 16%, respectively. The reason for these improvements is that ASPS places application data on SSDs and launches all mappers from the nodes with the same type of devices, such that the data retrieval of each mapper can be accelerated and the task skewness across mappers can be alleviated. In contrast, Rand and Def fully ignore storage device characteristics. We also observe that ASPS is comparable to Stor since there is only one application in the system thus ASPS defaults to Stor. Figure 6(b) shows the map phase times of the three applications. Similar to Fig. 6(a), we can find ASPS outperforms Rand and Def in reducing the map task completion times of the three applications.

Another observation from Fig. 6 is that ASPS achieves larger performance improvements for Grep and TeraSort than for K-Means. By analyzing the I/O times



(a) Application completion time



(b) Map task completion time

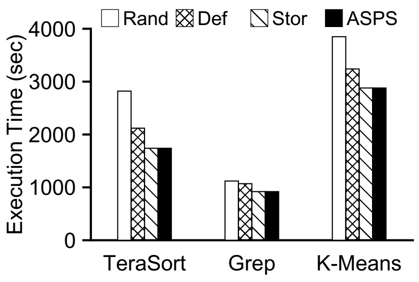
Fig. 6. Application performance when mapper input placement and scheduling optimization are enabled in ASPS.

and overall completion times of the three applications, we find that Grep and TeraSort are I/O intensive workloads while K-Means is a CPU intensive workload. Since ASPS focuses on data storage optimization in the I/O path, it accordingly brings more performance benefits for I/O intensive workloads Grep and TeraSort.

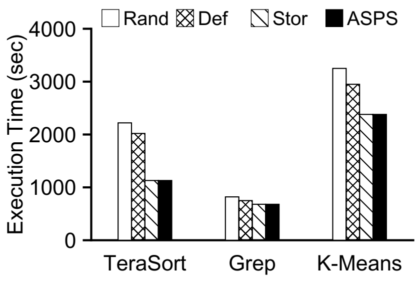
Mapper output placement optimization. Figure 7(a) compares application completion times when only mapper output data placement optimization is enabled in ASPS. This means that ASPS will store the output data of map tasks on local SSDs. One can find that ASPS improves job completion times by 10%–17% against Rand and Def. This shows placing the intermediate data on fast devices can efficiently improve the overall application performance. Figure 7(b) shows the corresponding map task completion times. ASPS contributes 15%–26% performance improvements, which are larger than those improvements of the overall application completion times. Similarly, ASPS has comparable performance with Stor because it defaults to Stor in the single-application case.

Reducer input placement optimization. Figure 8 shows the application execution times when only the reducer input data placement optimization is enabled in ASPS. This means that ASPS will store the input data of reduce tasks on SSDs if the intermediate data are materialized in the local file system. To show the impacts of reduce task data placement on application performance, we set the number of reducers to 16. ASPS improves job completion time by 7%–13% compared to Rand and Def. The reason is that both Rand and Def randomly store the materialized input data of reduce tasks in the local file system on HDDs while ASPS stores them on fast SSDs. This can greatly speed up the I/O materialization procedure.

Reducer output placement and scheduling optimization. Figure 9 plots the application completion times when only the reducer output placement and scheduling optimization are enabled. In this case, ASPS tries to place application data on SSDs and to schedule all reducers on the nodes with the same type of devices. We set the number of reducers to 16. ASPS improves the job completion time of TeraSort by

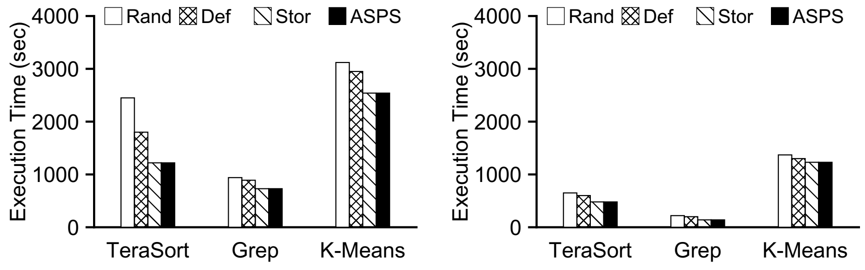


(a) Application completion time



(b) Map task completion time

Fig. 7. Application performance when mapper output placement optimization is enabled in ASPS.

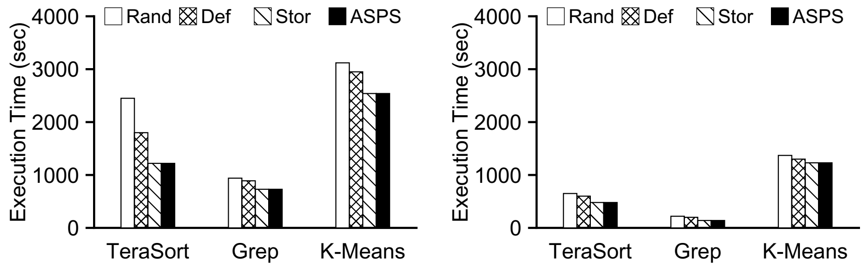


(a) Application completion time (b) Reducer completion time

Fig. 8. Application performance when reducer input placement optimization is enabled in ASPS.

14% and 27% compared to Rand and Def on the heterogeneous cluster, respectively. The reason is that both Rand and Def place data and schedule tasks in a storage-unaware fashion; thus, some reducers are allocated to nodes with HDDs and others are allocated to nodes with SSDs, leading to severe task skewness in the reducer execution. In contrast, ASPS attempts to launch reducers on the nodes with the same type of fast SSDs, which not only speeds up write accesses but also alleviates task skewness across reducers, leading to enhanced system performance.

Comprehensive optimization. Figure 10 plots the application performance when all the optimizations in ASPS are enabled. While it only shows the results of TeraSort with four reducers (4R) and 16 reducers (16R), the results of other applications bring a similar conclusion. Compared to Def, we can see that ASPS improves the overall performance by 36% and 32%, respectively, for four reducers and 16 reducers. This indicates that ASPS can efficiently reduce application completion time when combining all optimizations compared to the case where only one optimization is used. However, the overall improvement cannot be the sum of that of each optimization. This is because some of the execution phases run concurrently. For example,



(a) Application completion time (b) Reducer completion time

Fig. 9. Application performance when reducer output placement and scheduling optimizations are enabled in ASPS.

the reduce phase overlaps with the map phase, which will compromise the performance improvement of each phase.

5.3. *Evaluation with multiple applications*

In this set of experiments, we evaluate ASPS on the storage-heterogeneous cluster with six concurrent applications, i.e., WordCount, GrepSort, GrepSearch, TeraSort, Sorter, and BigramCount. The former four applications are from the PUMA benchmark.[26] The BigramCount is from Cloud[9] MapReduce toolkit[32] and the Sorter is from HiBench benchmark suite.[33]

The input size of each application is 48 GB. As the baseline case, we use "fsck" command to check the data placement of files and ensure that the ratio between the consumed HDD capacity and SSD capacity is approximately 1 to 1, which is consistent to the policy used by the current HDFS data placement policy. We submit the six jobs sequentially but within a short interval, such that each test has the same order of submission for a fair comparison.

Completion time. Figure 11(a) demonstrates the average completion times of the six applications under different placement and scheduling policies. In these tests, we enable all the optimization techniques in ASPS. As shown in the figure, ASPS

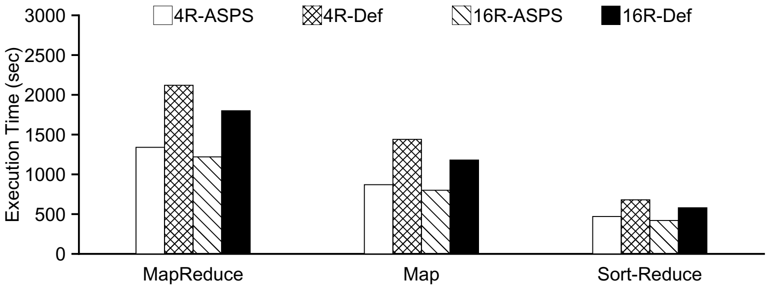
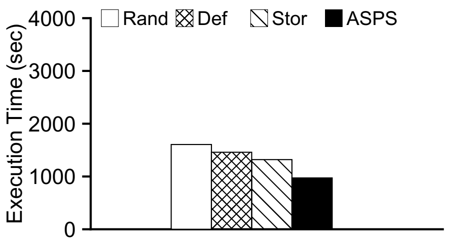
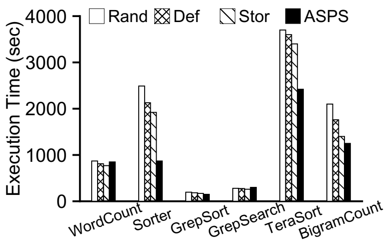


Fig. 10. TeraSort performance when all optimizations are enabled in ASPS.



(a) Average completion time



(b) Individual completion time

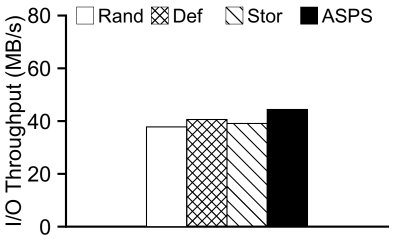
Fig. 11. Completion times of applications.

reduces the average application completion time by 36% and 27% compared to Rand and Def on the heterogeneous cluster, respectively. The reason is that both Rand and Def are storage- and application-unaware; thus, they are obviously inefficient for multiple jobs in a heterogeneous Hadoop cluster.

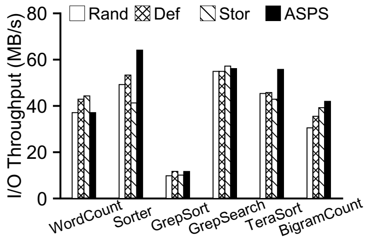
Figure 11(a) also reveals that ASPS outperforms Stor. This is due to the fact that Stor only considers device heterogeneity but it ignores application I/O requirements. Thus, while Stor tries to place data of a job on the same type of devices and schedule the tasks of the job from the same types of devices, it does not consider how to make better utilization of the storage resources for multiple jobs. In contrast, ASPS can carry out cost-effective job execution for multiple applications by trying to place the data of applications with high I/O requirements on high-performance SSDs and to execute the tasks of these applications from the nodes with SSDs.

Figure 11(b) shows the completion time of each individual application. One can find that ASPS does not always outperform Stor in terms of individual application performance in multi-application cases. For example, ASPS increases the completion times of WordCount and GrepSearch compared to Stor. This is because ASPS treats them as applications with low I/O requirements and it leaves the chance of utilizing high-performance SSDs to other applications. According to the I/O metrics in Sec. 3.3, only Sorter and TeraSort are I/O intensive applications and thus, SSDs are reserved for these applications. As the rest of applications are insensitive to storage devices, their data placement and task scheduling prefer to HDDs. However, in terms of overall system performance, ASPS reduces the overall application completion time compared to Stor, Rand, and Def. This result indicates that ASPS is efficient for multi-application Hadoop cluster environments.

I/O throughput. Figure 12(a) demonstrates the average throughput of all applications in the above experiments. Similar to the previous tests, the result shows that ASPS can largely improve the average I/O throughput of the six applications relative to Rand, Def, and Stor. Compared to Def, ASPS improves the average I/O throughput by 9.3%. Figure 12(b) plots the I/O throughput of each individual application. One can find that with ASPS, some applications' throughputs are



(a) Average I/O throughput



(b) Individual I/O throughput

Fig. 12. I/O throughputs of applications.

improved and some applications' throughputs are decreased relative to the other three approaches. However, the overall system throughput is still significantly enhanced.

6. Related Work

Data placement. Researchers have paid attention to optimize data placement of HDFS to improve Hadoop performance. VENU[14] uses SSDs as caches to place preferred data in the Hadoop system incorporating heterogeneous storage media. Triple-H[4] proposes a hybrid architecture to distribute data on heterogeneous storage resources on high-performance computing (HPC) clusters. NVFS[15] places performance-critical data on NVMs in a novel HDFS that utilizes NVM and remote direct memory access (RDMA). hatS[13] proposes a hybrid locality- and tier-aware data placement policy for HDFS with heterogeneous tiered storage devices. In Ref. 16, Xie *et al.* allocated a different number of blocks to nodes according to the node data processing capacity. The recent implementation of HDFS provides heterogeneous storage types in each data node.[23,24] However, these efforts focus on a single application while our work focuses on multiple concurrent applications. We also assume the CPU and memory resource are homogeneous and only the storage devices are heterogeneous in each data node.

Numerous efforts improve parallel program performance by optimizing the data placement of parallel file systems.[34,35] While these techniques are designed for HPC platform, our approach is designed for the Hadoop platform. Given the inherent differences between HPC and Hadoop architectures, such as network topology, data redundancy, etc., existing approaches in the HPC community cannot be simply applied to the Hadoop environment. This study efficiently addresses these challenges.

Task scheduling. Optimizing task scheduling is another effective approach to boost Hadoop performance. LATE scheduler optimizes backup tasks for straggler tasks to improve system efficiency.[17] SkewTune[18] breaks the atomic task into subtasks to gain data processing parallelism. Zacheilas *et al.* scheduled heterogeneous workloads onto heterogeneous nodes.[19] Tarazu[20] uses on-line trace and scheduling techniques to optimize the trade-off between parallelism of computing power and data transfer in a heterogeneous hardware environment. MROnline[21] schedules different resources for different applications according to their resource needs. Morpheus[27] utilizes dynamic resource reservations to mitigate execution variance. However, all these studies focus on CPU and data partition heterogeneity while our research focuses on storage and application heterogeneity. H-Scheduler[36] proposes a storage-aware scheduling approach in Spark clusters, but it is designed for multiple tasks of an application instead of multiple applications. In order to better optimize task scheduling of our research, many works in other heterogeneous environments are also worth studying.[37–40]

For scheduling of dependent jobs, such as workflow systems, task schedulers are proposed to optimize cost and makespan.[41,42]

For traditional scientific workloads, Ucar *et al.* presented a clustering task scheduling approach to assign tasks in heterogeneous computing systems.[43] This method considers task execution costs as well as communication costs between different tasks. However, this scheme focuses on HPC environments, while our approach is designed for Hadoop platforms.

Integrated approach. Some researchers use integrated approaches to boost Hadoop system performance. A closest related work is ϕSched,[22] which is a hardware-aware workflow scheduler for Hadoop. It treats a Hadoop deployment as a collection of multiple heterogeneous clusters; thus, it examines the current cluster load along with data availability information to schedule the job to an appropriate cluster. In addition, ϕSched enhances the data placement of HDFS across a multi-cluster deployment, so that it can handle data locality as well as enable pre-staging of data to appropriate clusters as needed.

Our work differs from ϕSched in that we target one Hadoop cluster, where each node has heterogeneous storage devices but with homogeneous CPU and memory resources, while ϕSched focuses on multiple clusters, where different clusters have heterogeneous hardware. We also assume applications may have different I/O requirements when concurrently accessing a shared cluster.

7. Conclusions

In the big data era, heterogeneous application and storage become the trends in data centers. However, as one of the most popular big data processing frameworks, Hadoop has not efficiently updated in the face of such changes. This paper presents ASPS, cooperative data placement and task scheduling approach for Hadoop clusters. ASPS relies on three key techniques: deriving application I/O requirements from a sampling method, placing application data according to application I/O requirements to better utilize fast devices, and scheduling application jobs based on I/O requirements and device types to achieve cost-effectiveness. We have validated our design and implementation with extensive tests in a Hadoop cluster. In the future, we plan to evaluate ASPS in a large-scale Hadoop cluster that is not currently available to us. We also intend to extend ASPS on other big data platforms and evaluate its effectiveness in practical environments.

Acknowledgments

This work was supported in part by the National Natural Science Foundation of China No. 61772185, the National Natural Science Foundation of China No. 61572377 and the Zhejiang Lab Research Project No. 2019KC0AC01.

References

1. J. Dean and S. Ghemawat, MapReduce: Simplified data processing on large clusters, *Proc. USENIX Symp. Operating Systems Design and Implementation (OSDI)* (San Francisco, CA, USA, 2004), pp. 137–150.
2. Apache Software Foundation, Apache Hadoop Project (2018). Available at http://hadoop.apache.org.
3. K. Shvachko, H. Kuang, S. Radia and R. Chansler, The Hadoop distributed file system, *Proc. IEEE Symp. Mass Storage Systems and Technologies (MSST)* (NW Washington, DC, USA, 2010), pp. 1–10.
4. N. S. Islam, X. Lu, M. Wasi-ur Rahman, D. Shankar and D. K. Panda, Triple-H: A hybrid approach to accelerate HDFS on HPC clusters with heterogeneous storage architecture, *Proc. 15th IEEE/ACM Int. Symp. Cluster, Cloud Grid Computing (CCGrid)* (Shenzhen, China, 2015), pp. 101–110.
5. G. Ananthanarayanan, A. Ghodsi, A. Warfield, D. Borthakur, S. Kandula, S. Shenker and I. Stoica, PACMan: Coordinated memory caching for parallel jobs, *Proc. 9th USENIX Conf. Networked Systems Design and Implementation (NSDI)* (San Jose, CA, USA, 2012), pp. 267–280.
6. Apache Software Foundation, Apache Spark (2018). Available: https://spark.apache.org.
7. Apache HBase, (2018). Available: http://hbase.apache.org.
8. S. Ibrahim, H. Jin, L. Lu, B. He and S. Wu, Adaptive disk I/O scheduling for MapReduce in virtualized environment, *Proc. 40th Int. Conf. Parallel Processing (ICPP)* (Taipei, Taiwan, 2011), pp. 335–344.
9. X. Bu, J. Rao and C.-Z. Xu, Interference and locality-aware task scheduling for MapReduce applications in virtual clusters, *Proc. 22nd Int. Symp. High-Performance Parallel and Distributed Computing (HPDC)* (New York, NY, USA, 2013), pp. 227–238.
10. X. Li, Y. Wang, Y. Jiao, C. Xu and W. Yu, CooMR: Cross-task coordination for efficient data management in MapReduce programs, *Proc. Int. Conf. High Performance Computing, Networking, Storage and Analysis (SC)*, Article No.: 42 (Denver, CO, USA, 2013), pp. 1–11.
11. F. Ahmad, S. T. Chakradhar, A. Raghunathan and T. Vijaykumar, ShuffleWatcher: Shuffle-aware scheduling in multi-tenant MapReduce clusters, *Proc. USENIX Conf. USENIX Annual Technical Conf. (ATC)* (Philadelphia, PA, USA, 2014), pp. 1–12.
12. E. Pettijohn, Y. Guo, and P. Lama, User-centric heterogeneity-aware MapReduce job provisioning in the public cloud, *Proc. 11th Int. Conf. Autonomic Computing (ICAC)* (Philadelphia, PA, USA, 2014), pp. 137–143.
13. K. Krish, A. Anwar and A. R. Butt, hatS: A heterogeneity-aware tiered storage for Hadoop, *Proc. 14th IEEE/ACM Int. Symp. Cluster, Cloud and Grid Computing (CCGrid)* (IEEE, 2014), pp. 502–511.
14. K. Krish, M. S. Iqbal and A. R. Butt, Venu: Orchestrating SSDs in Hadoop storage, *Proc. 2014 IEEE Int. Conf. Big Data (Big Data)* (Washington DC, USA, 2014), pp. 207–212.
15. N. S. Islam, M. Wasi-ur Rahman, X. Lu and D. K. Panda, High performance design for HDFS with byte-addressability of NVM and RDMA, *Proc. 2016 Int. Conf. Supercomputing (ICS)* (Istanbul, Turkey, 2016), pp. 1–14.
16. J. Xie, S. Yin, X. Ruan, Z. Ding, Y. Tian, J. Majors, A. Manzanares and X. Qin, Improving Mapreduce performance through data placement in heterogeneous Hadoop clusters, *Workshops and PhD forum of the Int. Symp. Parallel & Distributed Processing (IPDPSW)* (Atlanta, GA, USA, 2010), pp. 1–9.

17. M. Zaharia, A. Konwinski, A. D. Joseph, R. Katz and I. Stoica, Improving MapReduce performance in heterogeneous environments, *Proc. 8th USENIX Conf. Operating Systems Design and Implementation (OSDI)* (San Diego, CA, USA, 2008), pp. 29–42.
18. Y. Kwon, M. Balazinska, B. Howe and J. Rolia, Skewtune: Mitigating skew in MapReduce applications, *Proc. 2012 ACM Int. Conf. Management of Data (SIGMOD)* (Scottsdale, Arizona, USA, 2012), pp. 25–36.
19. N. Zaheilas and V. Kalogeraki, Real-time scheduling of skewed MapReduce jobs in heterogeneous environments, *Proc. 11th Int. Conf. Autonomic Computing (ICAC)* (Philadelphia, PA, USA, 2014), pp. 189–200.
20. F. Ahmad, S. T. Chakradhar, A. Raghunathan and T. N. Vijaykumar, Tarazu: Optimizing MapReduce on heterogeneous clusters, *Proc. Int. Conf. Architectural Support for Programming Languages and Operating Systems (ASPLOS)* (London, UK, 2012), pp. 61–74.
21. M. Li, L. Zeng, S. Meng, J. Tan, L. Zhang, A. R. Butt and N. Fuller, MRONLINE: MapReduce online performance tuning, *Proc. 23rd Int. Symp. High-Performance Parallel and Distributed Computing (HPDC)* (Vancouver, Canada, 2014), pp. 165–176.
22. K. Krish, A. Anwar and A. R. Butt, ϕSched: A heterogeneity-aware Hadoop workflow scheduler, *Proc. IEEE 22nd Int. Symp. Modelling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS)* (Paris, France, 2014), pp. 255–264.
23. Configuring Heterogeneous Storage in HDFS, (2018). Available at https://www.cloudera.com/documentation/enterprise/5-8-x/topics/admin_heterogeneous_storage_oview.html.
24. Enable Support for Heterogeneous Storages in HDFS (2018). Available at https://issues.apache.org/jira/browse/HDFS-2832.
25. Heterogeneous Storage Phase 2 — APIs to Expose Storage Types (2018). Available at https://issues.apache.org/jira/browse/HDFS-5682.
26. F. Ahmad, PUMA Benchmarks and Dataset Downloads (2018). Available at https://engineering.purdue.edu/puma/datasets.htm.
27. S. A. Jyothi, C. Curino, I. Menache, S. M. Narayanamurthy, A. Tumanov, J. Yaniv, R. Mavlyutov, I. Goiri, S. Krishnan and J. Kulkarni, Morpheus: Towards automated SLOs for enterprise clusters, *Proc. 12th USENIX Symp. Operating Systems Design and Implementation (OSDI)* (2016), pp. 117–134.
28. J. Rasley, K. Karanasos, S. Kandula, R. Fonseca, M. Vojnovic and S. Rao, Efficient queue management for cluster scheduling, *Proc. Eleventh European Conf. Computer Systems (EuroSys)* (London, UK, 2016), pp. 36:1–36:15.
29. H. Herodotou and S. Babu, Profiling, what-if analysis, and cost-based optimization of MapReduce programs, *Proc. VLDB Endow.* **4** (2011) 1111–1122.
30. A. Rasmussen *et al.*, Themis: An I/O-efficient MapReduce, *Proc. Third ACM Symp. Cloud Computing (SOCC)*, Article No.: 13 (San Jose, CA, USA, 2012), pp. 1–14.
31. A. Shinnar, D. Cunningham, V. Saraswat and B. Herta, M3R: Increased performance for in-memory Hadoop jobs, *Proc. VLDB Endow.* **5** (2012) 1736–1747.
32. J. Lin, A Hadoop Toolkit for Working with Big Data, (2014). Available at http://lintool.github.io/Cloud9/.
33. S. Huang, J. Huang, J. Dai, T. Xie and B. Huang, The HiBench benchmark suite: Characterization of the MapReduce-based data analysis, *Proc. IEEE 26th Int. Conf. Data Engineering Workshops (ICDEW)* (Long Beach, CA, USA, 2010), pp. 41–51.
34. S. He, X.-H. Sun and A. Haider, HAS: Heterogeneity-aware selective data layout scheme for parallel file systems on hybrid servers, *Proc. 29th IEEE Int. Parallel and Distributed Processing Symp. (IPDPS)* (Hyderabad, India, 2015), pp. 613–622.

35. S. He, X.-H. Sun, Y. Wang and C. Xu, A migratory heterogeneity-aware data layout scheme for parallel file systems, *Proc. 32nd IEEE Int. Parallel and Distributed Processing Symp. (IPDPS)* (Vancouver, British Columbia, Canada, 2018), pp. 1133–1142.
36. F. Pan, J. Xiong, Y. Shen, T. Wang and D. Jiang, H-Scheduler: Storage-aware task scheduling for heterogeneous-storage spark clusters, *Proc. IEEE 24th Int. Conf. Parallel and Distributed Systems (ICPADS)* (Singapore, 2018), pp. 1–9.
37. J. Zhou, T. Wei, M. Chen, J. Yan, X. S. Hu and Y. Ma, Thermal-aware task scheduling for energy minimization in heterogeneous real-time MPsoC systems, *IEEE Trans. Comput. Aided Design Integr. Circuits Syst.* **35** (2015) 1269–1282.
38. J. Zhou, J. Yan, K. Cao, Y. Tan, T. Wei, M. Chen, G. Zhang, X. Chen and S. Hu, Thermal-aware correlated two-level scheduling of real-time tasks with reduced processor energy on heterogeneous MPSoCs, *J. Syst. Arch.* **82** (2018) 1–11.
39. J. Li, G. Xie, K. Li and Z. Tang, Enhanced parallel application scheduling algorithm with energy consumption constraint in heterogeneous distributed systems, *J. Circuits Syst. Comput.* **28**(11) (2019) 1950190.
40. J. Jiang, W. Li, L. Pan, B. Yang and X. Peng, Energy optimization heuristics for budget-constrained workflow in heterogeneous computing system, *J. Circuit Syst. Comput.* **28** (2019) 1950159.
41. X. Zhou, G. Zhang, J. Sun, J. Zhou, T. Wei and S. Hu, Minimizing cost and makespan for workflow scheduling in cloud using fuzzy dominance sort based HEFT, *Future Gener Comput. Syst.* **93** (2019) 278–289.
42. P. Lu, G. Zhang, Z. Zhu, X. Zhou, J. Sun and J. Zhou, A review of cost and makespan-aware workflow scheduling in clouds, *J. Circuits Syst. Comput.* **28** (2019) 1930006.
43. B. Ucar, C. Aykanat, K. Kaya and M. Ikinci, Task assignment in heterogeneous computing systems, *J. Parallel Distrib. Comput.* **66** (2006) 32–46.