2017 IEEE 31st International Conference on Advanced Information Networking and Applications

Scheduling-Aware Data Prefetching for Data Processing Services in Cloud

Chien-Hung Chen*, Ting-Yuan Hsia*, Yennun Huang†, and Sy-Yen Kuo*
*Department of Electrical Engineering National Taiwan University, Taipei, Taiwan
Email: {d01921025,r03921054,sykuo}@ntu.edu.tw
†Research Center for Information Technology Innovation Academia Sinica, Taipei, Taiwan
Email: yennunhuang@citi.sinica.edu.tw

***Abstract*—Cloud computing services provide flexible computing and storage resources to process large amount of datasets. In-memory techniques keep the frequently used data into faster and more expensive storage media for improving performance of data processing services. Data prefetching aims to move data to low-latency storage media to meet requirements of performance. However, existing mechanisms do not consider how to benefit the data processing applications which do not frequently access the same datasets. Another problem is how to reclaim memory resources without affecting other running applications. In this paper, we provide a Scheduling-Aware Data Prefetching (SADP) mechanism for data processing services in a cloud data center. The SADP includes data prefetching and data eviction mechanisms. It firstly evicts the data from memory to release resources for hosting other data blocks, and then it caches the data that will be used in near future. Finally, real-testbed experiments are performed to show the effectiveness of the proposed SADP.**

I. Introduction

In the era of Big Data, the amount of data on the world will double in size every two years and reach at least 4.4 ZB by 2020 [1]. Cloud data management is one of the key challenges to improve performance of processing large amount of data. Nowadays, multi-tiered storage systems are used in cloud data centers, where different storage media devices have a variety of capacity and performance capabilities. A cloud data management system has to decide which data should be placed on low-latency devices to meet performance requirements. If there are datasets that will not be accessed again, the management system should also evict them to high-latency devices for releasing more valuable resources to meet performance requirements of other datasets.

In-memory techniques keep datasets in random access memory to speed up processing of large amounts of datasets. The techniques are widely used in data processing and data storage systems. The in-memory data processing systems strive to analyze a large amount of data in a small amount of time. By persisting the frequently used data in memory, the execution time of jobs can be significantly improved, especially for the iterative jobs that iteratively reading the same datasets. However, non-iterative jobs are difficult to gain benefits from in-memory techniques, and the memory resources may be wasted on storing the datasets that will not be accessed again. Besides improving read throughput, write workloads are major bottleneck for data-intensive jobs. An in-memory data storage system is able to address such bottleneck. It caches output data in memory and achieves fault-tolerance by leveraging *lineage* [2].

In a large cloud data center, many data-intensive jobs may be running simultaneously. However, each computing node in the cloud has limited memory space to cache input and output data for multiple jobs. When a job j_1 is writing its output datasets, the input datasets of job j_2 may be evicted from the memory due to contention of memory resources. In such case, if the job j_2 cannot read its input datasets from memory, its execution time will be extended. To address this problem, there is a need for management of memory usage for multiple jobs. Most in-memory data processing systems reserve memory space for storing input datasets, intermediate results, and application programs. An in-memory data processing system usually caches data in memory when the data is read or written. Nevertheless, it cannot guarantee that the cached data will be reused. Even if the data blocks will not be accessed again, the data blocks may still be kept in memory until the eviction policy throws them out. If there is not enough space to cache other data blocks, the system will evict some data blocks from memory using Least Recently Used (LRU) policy [3]. In-memory storage systems adopt the same policy to deal with the datasets. Due to that the policy is not aware of which data blocks will be accessed, it may evict the data blocks that will be accessed in near future. In the meantime, it can also keep other unnecessary data blocks in memory.

To deal with these problems, in this paper, we provide a Scheduling-Aware Data Prefetching (SADP) mechanism for data processing services in a cloud data center. The proposed SADP includes data prefetching and data eviction mechanisms. For the eviction mechanism, it aims to release memory space without affecting the access of running applications. When a block of datasets has already processed by a task, the data block will be released from memory space if it will not be accessed again. The data prefetching mechanism is aware of the job scheduling in a cloud data center. It is able to load input data into memory before its corresponding tasks are executed. The proposed mechanisms are implemented by modifying Spark [4] and Alluxio [5]. Apache Spark has become a popular in-memory data processing system. It optimizes the execution of data-intensive applications with features of interactive data exploration and multi-pass analytics. Alluxio is an in-memory

1550-445X/17 $31.00 © 2017 IEEE
DOI 10.1109/AINA.2017.54

data storage system with ability to manage data in tiered storage. Multi-tiered storage is conducive to balance capacity and performance requirements of data storage. It provides more flexible data management in a cloud data center. Currently, the data prefetching and data eviction policies still do not take job scheduling into account. The proposed mechanisms can retrieve scheduling information and then use this information to prefetch and evict data to/from memory.

Overall, this paper makes the following contributions:

- We propose data prefetching and eviction mechanisms which consider multiple jobs running on a large-scale cloud data center, where the cloud data center has limited memory resources.
- We eliminate the gap between computing layer and storage layer in a cloud, such that the storage layer is no longer agnostic to job scheduling. If a data computing service is running on the cloud, the data storage system is aware of which pending tasks will be executed in near future.
- We optimally solve the data prefetch problem by Integer Linear Programming (ILP). Then, we propose an efficient heuristic algorithm, called Scheduling-Aware Data Prefetching (SADP), to solve the data-prefetching problem in a large-scale cloud data center.
- We implement a prototype of our mechanisms on a real-testbed. We show that the proposed mechanisms can achieve about 1.59x faster than default mechanism.

The rest of this paper is organized as follows. The related work is given in Section II. Section III gives system model. Section IV presents our mechanisms. Section V shows the evaluations of the proposed mechanisms. Finally, section VI concludes this paper.

II. Related Work

To improve the performance of data processing, in-memory techniques have been extensively used in data processing systems. Spark [6] is a popular data processing framework for data analysis. It presents a data abstraction, called resilient distributed dataset (RDD), which allows application jobs to cache intermediate results in memory with a fault-tolerance mechanism. Mammoth [7] is an implementation of in-memory techniques based on MapReduce framework. It aims to allocate and reclaim memory resources among computing nodes for enhancing overall performance of application jobs. In the system, each computing node is deployed a special engine to globally manage the memory resources among a cluster. GraphLab [8] is an efficient shared-memory implementation of parallel computing framework for machine learning. A graph-based data model is exploited for representing data and computational dependencies. However, it assumes all data can be stored in memory without the problem of resource contention. SINGA [9] is an open-source platform for distributed deep learning. It is able to support different neural net partitioning schemes and training frameworks. Shared memory resources among the systems are leveraged to store intermediate results for reducing data accessing costs. More data processing systems are proposed for real-time purposes, such as Apache Storm [10] and Yahoo! S4 [11]. Among existing in-memory data processing systems, these systems only take intermediate data of an application job into account, therefore these in-memory data processing systems can only benefit from the jobs iteratively accessing the same datasets. On the other hand, memory resource contention is not considered as well. If there are multiple jobs run on the system, some portions of the intermediate data will be evited from memory, such that these application jobs have to read them from hard disk drives.

In general, there are two types of in-memory storage systems: file systems and database systems, respectively storing unstructured and structured data. Alluxio, formerly called Tachyon [2], is a distributed file system. It can work as a cache system to enhance performance of data accessing for other file systems and databases. Alternatively, it can work as a standalone in-memory file system managing the storage resources. Mercury [12] is a distributed in-memory database for storing structured data. A dedicated hash table is designed for small data sizes of key-value pairs to improve throughput of data analytics jobs. MICA [13] is a key-value in-memory storage system focusing on both read- and write-intensive jobs. It aims to use fewer high-performance computing nodes to reduce latency of data access. New data structure and memory management are designed for optimizing data store and cache by using the properties of the jobs. Pilaf [14] is a distributed in-memory key-value storage system with high-performance networks. It allows application jobs directly access the data stored in memory from remote computing nodes. A self-verifying data structure is also provided to address contention of read-write operations. Including above systems, most existing in-memory storage systems focus on design of memory management based on specific properties of data structures and iterative jobs. However, the job scheduling is not taken into account.

Job scheduling plays an important role in allocating CPU and memory resources for executing data processing jobs among a large-scale cloud. In Spark, it supports three modes of scheduling to deal with multiple jobs: Standalone, Mesos, and YARN. In standalone mode, all jobs in a cluster are run in FIFO (first in, first out) order. The tasks of each job can be allocated to all computing nodes for reaching maximum usage of CPU and memory resources [3]. In Mesos mode, the system allocates resources in accordance with user-defined policy, such as fair sharing and strict priority [15]. It can also share system resources at different granularities based on latency requirements of Spark jobs. In YARN mode, one of simple FIFO, Capacity, and Fair Share schedulers can be selected depending on the user needs [16]. In addition, data locality can have significant impacts on job scheduling. A task of running jobs prefers to be allocated to where its input data stored. Therefore, the job schedulers are designed around the general principle of data locality.

III. Preliminaries

This section presents the system model of in-memory systems used in this paper. The assumptions and definitions of the data prefetching problem are also stated.

A. System Model

Spark is a big data processing framework designed to be fast and general purpose. The resilient distributed dataset (RDD) is the core concept in Spark. It represents a collection of data partitions distributed across many computing nodes. A Spark job can be divided into two or more stages, where each stage consists of a set of tasks. The stages are processed in order defined by a directed acyclic graph (DAG). A central process, called the driver, is responsible for coordinating with a number of executors to run the tasks of the given job. The number of tasks in a stage is the same as the number of data partitions generated in the previous stage. Each task within a job accesses its corresponding data partition, then it performs either *transformation* or *action* operations. If a task performs transformation, its output is constructed as new RDDs. If it performs action, it will return the computing results to the driver process or store the output of the job. When a Spark job is submitted, the driver process firstly asks the cloud manager for resources to launch executors. Tasks of the given job are sent to the executors to perform transformation or action operations. In the first stage of a job, the tasks usually perform transformation operation to load each data block of the input datasets from an external storage system and create RDDs. In the last stage, the tasks perform action operation to save output results to the external storage system. Our proposed mechanisms aim to prefetch and evict the data blocks of the datasets stored in the external storage system, instead of the data partitions of RDDs used in data processing layer.

Alluxio is the external storage system of Spark used in this paper. It supports tiered storage to manage data blocks stored in memory (MEM) and hard disk drive (HDD). When a job is writing new data block to the external storage, the data blocks will be firstly cached to the memory. If there is no enough space to accommodate new data blocks, the system will evict the least recently used data blocks cached memory by default. For the under storages, Alluxio can integrate with various under storages, such as Apache HDFS [17], OpenStack Swift [18], Amazon S3 [19], etc. The HDFS, a popular distributed file system, is used as the under storage of Alluxio. It can be deployed to the same computing ndoes with Spark and Alluxio, so that the system can take advantage of data locality to avoid network transmission delay. The system architecture we used is illustrated in Fig. 1. As shown in Fig. 1, Spark, Alluxio and HDFS are installed among the computing nodes, where Spark is responsible for executing data processing jobs. Alluxio is the external storage system of the Spark. It reserves a part of memory space of each computing node for caching data blocks from its under storage system. HDFS is the under system of Alluxio storing input and output datasets of Spark jobs into hard disk drives.

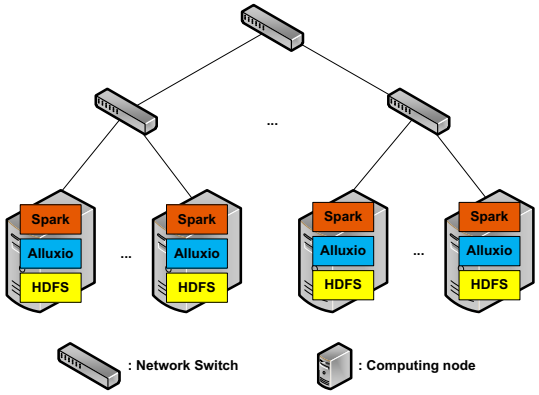


Fig. 1. The architecture of our system.

B. Assumptions and Definitions

Based on the given system model, this paper investigates the Scheduling-Aware Data Prefetching based on Spark Framework, called SADP. Before elaborating our proposed mechanisms, we first give the following assumptions and definitions.

Assumption 1. *Given a large-scale cloud with a set of computing nodes N. In addition to process data, each computing node $n \in N$ can also store data blocks. The computing nodes functionality are similar to the slave nodes in Spark, Alluxio, and HDFS.*

Assumption 2. *Among the computing nodes, there are two sets of data blocks D_h and D_m. The D_h denotes a set of data blocks stored in hard disk drives, where their corresponding tasks are the pending tasks. D_m denotes a set of data blocks cached in memory, where their corresponding tasks are running or completed.*

Assumption 3. *If there is a job writing new data blocks to the external storage system, the new data blocks will be cached in memory first. In the other hand, if there is a job reading data blocks from the external storage, the data blocks will also be cached in memory. It is possible that there are many data blocks cached by other jobs or external storage at the same time.*

Assumption 4. *Each node $n \in N$ has reserved a certain of memory for caching data blocks. Due to space limitation, each node cannot cache too many data blocks for jobs read and write. If the memory space is not enough to accommodate more data blocks, the system will evict data blocks from memory for releasing more memory resources. The evicted data blocks are then stored into hard disk drives of under storages.*

Assumption 5. *The information of both job scheduling and datasets can be obtained from the master node of the system. Each node in the cloud periodically reports the states of the running tasks and the stored data blocks to the master node. The master, therefore, can allocates tasks to the computing nodes based on a specific scheduling manner.*

Assumption 6. *Each data block d_i is associated with a probability p_i and a time c_{ik}, where the p_i denotes the probability that the data block d_i will be reused (referenced), c_{ik} is the benefit time from persisting the data block d_i in memory of node k. Note that, if the data block d_i is persisted in memory and read by a task, the benefit time is the time accessing to the hard disk drive minus the time accessing to the memory.*

Based on the above assumptions, the problems we investigated in this paper are defined as follows.

Definition 1. *The main objective of the prefetch problem is to find an optimal dataset D_{opt} from datasets D_h and D_m, which can maximize the total re-use probability and the total benefit time.*

IV. Proposed Mechanism

A. Optimal Solution

The prefetching problem can be optimally solved by Integer Linear Programming (ILP). ILP is a well-known mathematical technique to solve optimal problems, where its canonical form includes a linear objective function, a number of linear constraints, and an integer solution set. In this section, the canonical form of ILP corresponding to the prefetching problem is expressed as Eq. (1) to Eq. (5). The used notations can be found in Table I.

$$\begin{aligned} Maximize \sum_{\forall i \in D_h} \sum_{\forall k \in N} x_{ik} \times \{\alpha \cdot p_i + (1-\alpha) \cdot c_{ik}\} \\ - \sum_{\forall j \in D_m} \sum_{\forall k \in N} y_{jk} \times \{\alpha \cdot p_j + (1-\alpha) \cdot c_{jk}\}, \end{aligned} \tag{1}$$

$$subject\ to\ \forall i \in D_h, \sum_{\forall k \in N} x_{ik} \leq 1, \tag{2}$$

$$\forall j \in D_m, \sum_{\forall k \in N} y_{jk} \leq 1, \tag{3}$$

$$\forall k \in N, \sum_{\forall i \in D_h} x_{ik} - \sum_{\forall j \in D_m} y_{jk} = a_k, \tag{4}$$

$$\forall i \in D_h \wedge \forall j \in D_m \wedge \forall k \in N, x_{ik}, y_{j_k} \in \{0, 1\}. \tag{5}$$

For the Eq. (1), it is the objective function of the canonical form. There are two maximum terms in the objective function. The first term is the total weight of data blocks prefetching to memory, and the second term is the total weight of data blocks evicting to hard disk drives. The symbol D_h denotes a set of data blocks stored in hard disk drives, where their corresponding tasks are the pending tasks. The D_m denotes a the set of data blocks cached in memory, where their corresponding tasks are running or completed. N is a set of nodes in the cloud. In the Eq. (1), the data prefetching and eviction can be obtained based on the binary variables x_{ik} and y_{jk} respectively. If x_{ik} is 1, the corresponding data block d_i is selected to be cached in memory of node k. If y_{jk} is also 1, the corresponding data block d_j is evicted from memory to hard disk drives of node k. The variables p_i and p_j are the probabilities that the data blocks d_i and d_j will be accessed. The benefit times from caching data blocks d_i and d_j in the memory of node k are respectively denoted by c_{ik} and c_{jk}. The α is a weight coefficient with value from 0 to 1. If the weights are completely depended on the probability of access, the value of α is then set as 1. Conversely, if α is set as 0, the weights are totally depended on the benefit time. Based on the Eq. (1), the canonical form attempts to maximize the total weight by prefetching the data blocks with higher access probability or benefit time. When the memory is full, if there is a data block d_i witch weight is greater than the data block d_j, the objective function will evict d_j and cache d_i by setting x_{ik} and y_{jk} as 1. Note that, if there is a data block that is evicted from memory, the canonical form will cache another data block due to Eq. (4). The reason will be explained later. From the above description, we can know that the canonical form will firstly prefetch data blocks with high weights as many as possible. Then, it replaces low-weight data blocks by high-weight data blocks. Eq. (2) and (3) respectively express that a data block stored in hard disk drives and memory can only be cached or evicted once. In Eq. (4), a_k denotes the available memory space in computing node k. It limits that each node cannot cache too many data blocks to exceed the capacity of its memory space. The number of prefetched data blocks in a node should equal to the number of evicted data blocks plus the free memory space of the node. Due to this constraint, the high-weight data blocks will fill up the free memory space of the system, and the low-weight data blocks in memory can be replaced by high-weight data blocks. Finally, the Eq. (5) is given for setting the solution domain. The variables x_{ik} and y_{jk} can only be 0 or 1, respectively.

TABLE I
Summary of Notations

Notation	Description
D_h	A set of data blocks stored in hard disk drives, where their corresponding tasks are the pending tasks.
D_m	A the set of data blocks cached in memory, where their corresponding tasks are running or completed.
N	A set of computing nodes in the cloud.
a_k	The available memory space in computing node k.
x_{ik}	The 0,1 variable indicates whether a data blocks d_i is prefetched in memory of node k.
y_{jk}	The 0,1 variable indicates whether a data blocks d_j is evicted from memory of node k.
α	A weight coefficient with value from 0 to 1.
p_i (p_j)	The probability that the data blocks d_i (d_j) will be referenced.
c_{ik} (c_{jk})	The benefit time from caching data blocks d_i (d_j) in the memory of node k.

B. Scheduling-Aware Data Prefetching

In the above canonical form, there are $|D_h| \times |N|$ and $|D_m| \times |N|$ binary variables in x_{ik} and y_{jk}. In a cloud, the number of nodes $|N|$ is usually large. Solving ILP is well known to be NP-complete [20]. If $|D_h|$, $|D_m|$ and $|N|$ are large, the above canonical form of ILP will take much computational time to obtain the optimal solution of the prefetching problem. To reduce the computational time, we first make the following assumption. Then a heuristic algorithm is proposed to solve the prefetching problem.

- For the jobs in the system, their task execution sequence and task execution time can be known in advance. Assumptions 5 has defined the capability. With this capability, the heuristic algorithm can find suitable data blocks evicting from memory and then prefetch input data blocks for upcoming tasks.

In this section, we present a new prefetching mechanism, which assumes that the scheduling information is available from computing layer. The mechanism consists of three phases: initial phase, eviction phase, and prefetching phase. To emphasize that the mechanism is aware of the scheduling information, the proposed mechanism is called the Scheduling-Aware Data Prefetching (SADP). The basic idea of the SADP is given in Fig. 2. When a job is submitted, the job is associated with an input dataset, and then it will be put in a ready queue. The input dataset is divided into multiple data blocks. Each task of the job can read or write its corresponding data block from/to the external storage. Before submitting the job, other jobs may have been running simultaneously in the cloud. In such case, the start time and completion time of each tasks can be estimated by retrieving scheduling information, such as running states of tasks and the sequence of task execution. The input data blocks stored in hard disk drives are collected in a dataset D_h, and other input data blocks that have already cached in memory are collected in a dataset D_m. The location of data blocks can be obtained from the external storage system. The initial phase is used to find which data blocks are evictable or prefetchable. Based on the execution sequence of tasks, the mechanism can clearly know which tasks have completed and witch tasks will be launched in near future. If a task has processed its corresponding data, the data block will be considered can be evicted from the memory. In the eviction phase, the proposed mechanism estimates how many data blocks stored in hard disk drives will be accessed. If remaining memory space is not enough to prefetch data blocks, it releases memory resources with considering of system overheads. Finally, in prefetching phase, the mechanism prefetches the corresponding data blocks of the upcoming tasks.

1) Initial Phase: Before a new job is submitted to the cloud, its input datasets have already been stored in the external storage system. When a new job is submitted to the cloud, both of its input and output datasets have been specified. The new job is then put in the job ready queue for waiting execution. While there may already have a number of jobs running simultaneously in the cloud, it is necessary to find the following data blocks before running the pending tasks of the jobs.

Input: A set N of computing nodes, and two sets of data blocks.
Output: Data prefetching and eviction of the datasets.

/* Initial Phase */
for each job j of J **do**
  $D_p \leftarrow$ Find the data blocks that can be prefetched into memory.
  $D_e \leftarrow$ Find the data blocks that can be evicted from memory.
end for
/* Eviction Phase */
$e \leftarrow$ Calculate the number of data blocks that should be evicted from memory.
$E \leftarrow$ Find *un-modified* data blocks from D_e.
if $e > |E|$ **then**
  $E \leftarrow$ Find $e - |E|$ *modified* data blocks from D_e.
end if
/* Prefetching Phase */
for each node n in N **do**
  for data block d_i stored in n and d_i in D_p **do**
    if the corresponding task of d_i will be launched **then**
      $P \leftarrow P \cup d_i$.
    end if
  end for
end for

Fig. 2. Basic idea of the Scheduling-Aware Data Prefetching.

- **Evictable data blocks (The data blocks that can be evicted from memory).** When multiple jobs running in the cloud, several data blocks may have been cached in memory. If a job reads its input dataset from the external storage system, each task of the job only reads one data block of the dataset. It means that the corresponding data block of a completed task will not be accessed again. Evicting such data block will not affect accessing of other running tasks. In this phase, the proposed mechanism first decides which data blocks can be evicted without affecting other running tasks.
- **Prefetchable data blocks (The data blocks that can be prefetched into memory).** Considering multiple jobs running on the cloud with limited computing resources, only a portion of tasks can run simultaneously in the cloud at a time. According to job scheduling, each computing node in the cloud can only launch a specific number of tasks. As mentioned above, each task of a job has only one input data block. Therefore, the number of prefetchable data blocks can be decided in accordance with the number of upcoming tasks.

An example is given in Fig. 3, where there are two jobs running in a cloud. The job j_1 performs reading, and the job j_2 performs writing, respectively. Based on the job scheduling,

the jobs j_1 and j_2 respectively hold two executors to perform their tasks. The task execution sequence is according to the indexes of tasks. The $t_{1,3}$ and $t_{1,4}$ are two tasks of job j_1 respectively reading data blocks $d_{1,3}$ and $d_{1,4}$ from the external storage system, and the tasks $t_{2,4}$ and $t_{2,5}$ of job j_2 respectively write the data block $d_{2,4}$ and $d_{2,5}$ to the external storage system. There are 9 data blocks $d_{1,1}$, $d_{1,2}$, $d_{1,3}$, $d_{1,4}$, $d_{2,1}$, $d_{2,2}$, $d_{2,3}$, $d_{2,4}$, and $d_{2,5}$, that have already cached in memory of the external storage system. We assume that the external storage system can at most cache 10 data blocks in memory. The initial phase firstly determines the number of data blocks that can be evicted from the memory. Based on the task execution sequence, the tasks $t_{1,1}$, $t_{1,2}$, $t_{2,1}$, $t_{2,2}$, and $t_{2,3}$ are finished. Therefore, their corresponding input data blocks $d_{1,1}$, $d_{1,2}$, $d_{2,1}$, $d_{2,2}$, and $d_{2,3}$ are decided to be evictable data blocks. After finding the evictable data blocks, the eviction phase then determines the number of data blocks that can be prefetched into memory. In this phase, all of the pending tasks $t_{1,5}$, $t_{1,6}$, $t_{1,7}$, $t_{1,8}$, $t_{2,6}$, $t_{2,7}$, $t_{2,8}$, and $t_{2,9}$ are the tasks that will be launched in near future. Their corresponding input data blocks $d_{1,5}$, $d_{1,6}$, $d_{1,7}$, $d_{1,8}$, $d_{2,6}$, $d_{2,7}$, $d_{2,8}$, and $d_{2,9}$ are decided to be the prefetchable data blocks. Finally, eight data blocks are decided to be prefetchable data blocks. The above initial phase just decides which data blocks are evictable and prefetchable. The executions of eviction and prefetching are actually performed in the following eviction phase and prefetching phase.

2) Eviction Phase: In above initial phase, the evictable data blocks are found. However, there are two types of evictable data blocks: droppable data blocks and modified data blocks. The droppable data blocks are the data blocks that do not be modified when they are persisting in memory. Conversely, the modified data blocks are the blocks that have been modified. These modified data blocks are maintained by the external storage system. By default, the external storage system achieves data fault tolerance using *lineage* technique [2]. Any changes of data blocks are traced by a logical directed acyclic graph (DAG). It means that any data blocks written by jobs will be persisted in the memory first. If these data blocks are evicted from memory before writing to hard disk drives, the evicted data blocks will be lost. Therefore, if the modified data blocks are selected to be evicted, these data blocks should be written to the hard disk drives before eviction. In the eviction phase, the proposed mechanism calculates how many data blocks will be prefetched, and then it decides which evictable data blocks should be evicted from memory. To avoid affecting other tasks accessing their data blocks, the empty memory space is preferred to be used. Instead of modified data blocks, the droppable data blocks are firstly selected to be evicted from memory. From the example of Fig. 3, the data blocks $d_{1,1}$, $d_{1,2}$, $d_{2,1}$, $d_{2,2}$ and $d_{2,3}$ are the evictable data blocks decided in initial phase, where the $d_{1,1}$ and $d_{1,2}$ are droppable data blocks without modification, the $d_{2,1}$, $d_{2,2}$ and $d_{2,3}$ are modified data blocks. The job scheduling reveals that 4 pending tasks will be launched after completing the running tasks. At least 4 corresponding data blocks of the

Job	Completed tasks
j_1	$t_{1,1}$, $t_{1,2}$
j_2	$t_{2,1}$, $t_{2,2}$, $t_{2,3}$

Job	Running tasks
j_1	$t_{1,3}$, $t_{1,4}$
j_2	$t_{2,4}$, $t_{2,5}$

Job	Pending tasks
j_1	$t_{1,5}$, $t_{1,6}$, $t_{1,7}$, $t_{1,8}$
j_2	$t_{2,6}$, $t_{2,7}$, $t_{2,8}$, $t_{2,9}$

Task and its corresponding data block

Task	$t_{1,1}$	$t_{1,2}$	$t_{1,3}$	$t_{1,4}$	$t_{1,5}$
Data block	$d_{1,1}$	$d_{1,2}$	$d_{1,3}$	$d_{1,4}$	$d_{1,5}$
Task	$t_{1,6}$	$t_{1,7}$	$t_{1,8}$		
Data block	$d_{1,6}$	$d_{1,7}$	$d_{1,8}$		
Task	$t_{2,1}$	$t_{2,2}$	$t_{2,3}$	$t_{2,4}$	$t_{2,5}$
Data block	$d_{2,1}$	$d_{2,2}$	$d_{2,3}$	$d_{2,4}$	$d_{2,5}$
Task	$t_{2,6}$	$t_{2,7}$	$t_{2,8}$	$t_{2,9}$	
Data block	$d_{2,6}$	$d_{2,7}$	$d_{2,8}$	$d_{2,9}$	

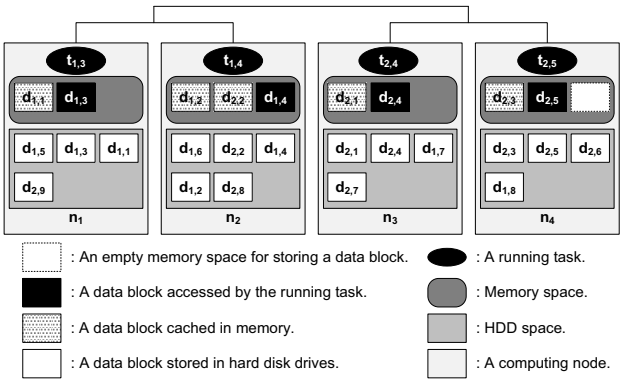
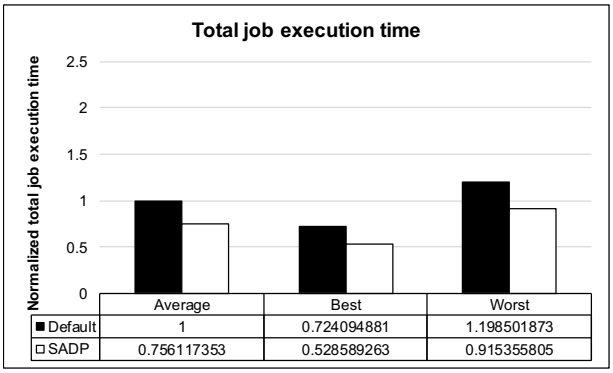


Fig. 3. An example to demonstrate the SADP.

pending tasks will be prefetched in the memory. Based on the memory space, it can accommodate one data block. Totally 3 data blocks among the evictable data blocks have to be evicted. According to the mechanism described above, two droppable data blocks $d_{1,1}$ and $d_{1,2}$ are firstly selected to be evicted, and then the modified data block $d_{2,1}$ is selected. After selecting the droppable and modified data blocks, the selected modified data blocks are written to hard disk drives of the external storage system, and then the selected data blocks are evicted from memory. Comparing to the default eviction mechanism, the data blocks $d_{1,1}$, $d_{2,1}$ and $d_{2,2}$ will be selected by LRU. However, our proposed mechanism can select fewer modified data blocks.

3) Prefetching Phase: Finally, the proposed mechanism turns into the prefetching phase. When the free memory



	Average	Best	Worst
Default	1	0.724094881	1.198501873
SADP	0.756117353	0.528589263	0.915355805

(a)

	Average	Best	Worst
Default	1.982521848	1.622971286	2.247191011
SADP	1.476941323	1.217228464	1.635955056

(b)

Fig. 4. Normalized total job execution time under FIFO scheduling. (a) 20 jobs. (b) 40 jobs.

	Average	Best	Worst
Default	1	0.802588997	1.216828479
SADP	0.683443366	0.561812298	0.815275081

(a)

	Average	Best	Worst
Default	4.113915858	3.365695793	4.71197411
SADP	2.593177994	2.053074434	2.939029126

(b)

Fig. 5. Normalized total job execution time under Fair scheduling. (a) 20 jobs. (b) 40 jobs.

resources are distributed crossing the cloud data center, a pending task may be launched on a computing node different from the location of its corresponding data block. In prefetching phase, the proposed mechanism aims to select which data block should be cached in memory of each computing node. As mentioned in Section II, the scheduler prefers to allocate pending tasks to where their input data blocks located to achieve data locality [16]. Based on data locality and scheduling policy, we can know which pending tasks will be launched on a specific computing node. As shown in Fig. 3, task $t_{1,3}$ runs on computing node n_1. If the running task $t_{1,3}$ is completed, the node n_1 will release an executor to run another pending task. To achieve data locality, the computing node n_1 prefers to run the pending tasks of $t_{1,5}$ and $t_{2,9}$. The scheduler will select the task $t_{1,5}$ to run on node n_1 for reserving computing resources for each job. Therefore, the corresponding data block $d_{1,5}$ of task $t_{1,5}$ is selected to be prefetched into memory of computing node n_1. In the prefetching phase, it will check each computing node for prefetching data blocks. In the example of Fig. 3, the data blocks $d_{1,6}$, $d_{2,7}$ and $d_{2,6}$ are selected to be prefetched in memory of computing nodes n_2, n_3, and n_4, respectively.

V. Evaluation

A. Experimental Environments

We establish our testbed by using the XenServer virtualization software [21] to configure 20 virtual machines on 4 physical machines. The proposed SADP mechanism is implemented by modifying the source code of Spark 1.6.1 and Alluxio 1.2.0 involving classes of TaskSetManager, DAGScheduler, TaskInfo, SparkContext, AlluxioBlockStore, LoadBlockCommand, etc. In the testbed, each physical machine is with quad core 3.30 GHz processor, 32GB of memory, and 2TB of disk. The network connection between machines is Gigabit Ethernet. We configure each virtual machine with 4 vCPU cores, 4 GB of memory, and 256 GB of disk. The proposed SADP mechanism is compared with the default mechanisms in Spark and Alluxio. For the aspect of job workloads, WordCount and TextSearch [22] are adopted as two benchmark jobs to represent different kinds of job workloads. In each testbed run, we submitted 20 to 40 jobs from the benchmark jobs. Each benchmark job is with four jobs by varying the sizes of input datasets as 10 GB, 20 GB, 30 GB, and 40 GB. After the above settings, 50 evaluation runs are performed. We concern the total job execution time as the metric in each evaluation run.

B. Experimental Results

The testbed experimental results are shown in Fig. 4 and Fig. 5. Fig. 4 shows the comparison of the total job execution time in testbed environments with FIFO scheduling based on the average case, best case, and worst case. Here, the total job execution time of each mechanism is normalized against the average total job execution time of the default mechanism. In the average case, the SADP can achieve the improvement of the total job execution time about 24% in 20 jobs. In 40 jobs, the SADP achieves the average improvement of the total job execution time about 26%. The SADP can prefetch input data blocks into memory before the corresponding tasks running on the cloud. Therefore, the running tasks can efficiently read their input data blocks from memory instead of from hard disk drives. As increasing the number of jobs, the proposed mechanism can gain more improvement.

Fig. 5 shows the comparison of the total job execution time with Fair scheduling in 20 jobs and 40 jobs, respectively. The total job execution time is normalized against the average total job execution time of using default mechanism as well. Comparing to default mechanism, the SADP improves 32% of the total job execution time, as shown in Fig. 5(a). With larger job workloads, the jobs will incur longer access time due to resource contention. Even the SADP is aware of the job scheduling, the resource contentions among jobs can still extend the execution time of jobs. However, the SADP tries to achieve better memory usage. In the 40 jobs, the total job execution time of the SADP achieves 37% of that of default mechanism on average, as shown in Fig. 5(b).

VI. Conclusion

We have investigated the data prefetching problem in a large-scale cloud data center. Considering each computing node in the cloud has limited memory resources, we provide two solutions to solve this problem. Firstly, we formulate a canonical form of Integer Linear Programming (ILP) for obtaining optimal solution. To make the data prefetching mechanism accommodates to a large-scale cloud data center, we also propose a heuristic algorithm, called Scheduling-Aware Data Prefetching (SADP), to prefetch and evict data to/from memory in accordance with job scheduling information. The prototype of SADP has been implemented in a real-testbed. The evaluation results show that the proposed mechanisms can efficiently perform the data prefetching for data processing services. The proposed mechanism can achieve 24% reduction in the total job execution time, and its improvement rate increases with the number of jobs.

Acknowledgment

The work is supported by the Ministry of Science and Technology, Taiwan, under Grant MOST 103-2221-E-001-028-MY3 and MOST 105-2221-E-002-119-MY3.

References

[1] V. Turner, J. Gantz, D. Reinsel, and S. Minton, "The Digital Universe of Opportunities: Rich Data and the Increasing Value of the Internet of Things," International Data Corporation (IDC), Tech. Rep., Apr. 2014.

[2] H. Li, A. Ghodsi, M. Zaharia, S. Shenker, and I. Stoica, "Tachyon: Reliable, Memory Speed Storage for Cluster Computing Frameworks," in *Proc. ACM Symposium on Cloud Computing (SOCC'14)*, Seattle, WA, USA, 2014, pp. 1–15.

[3] H. Karau, A. Kowinski, and M. Hamstra, *Learning Spark: Lightning-fast Big Data Analysis*. USA: OReilly Media, Inc., 2015.

[4] (2016) Apache Spark™- Lightning-Fast Cluster Computing. [Online]. Available: http://spark.apache.org/

[5] (2016) Alluxio - Open Source Memory Speed Virtual Distributed Storage. [Online]. Available: http://www.alluxio.org/

[6] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. Franklin, S. Shenker, and I. Stoica, "Resilient Distributed Datasets: A Fault-tolerant Abstraction for In-memory Cluster Computing," in *Proc. 9th USENIX Conference on Networked Systems Design and Implementation (NSDI'12)*, Berkeley, CA, USA, 2012, pp. 2–2.

[7] X. Shi, M. Chen, L. He, X. Xie, L. Lu, H. Jin, Y. Chen, and S. Wu, "Mammoth: Gearing Hadoop Towards Memory-Intensive MapReduce Applications," *IEEE Transaction on Parallel and Distributed Systems*, vol. 26, no. 8, pp. 2300–2315, Jul. 2014.

[8] Y. Low, D. Bickson, J. Gonzalez, C. Guestrin, A. Kyrola, and J. Hellerstein, "Distributed GraphLab: A Framework for Machine Learning and Data Mining in the Cloud," *VLDB Endowment*, vol. 5, no. 8, pp. 716–727, Apr. 2012.

[9] B. Ooi, Y. Wang, Z. Xie, M. Zhang, K. Zheng, K. Tan, S. Wang, W. Wang, Q. Cai, G. Chen, J. Gao, Z. Luo, and A. Tung, "SINGA: A Distributed Deep Learning Platform," in *Proc. 23rd ACM International Conference on Multimedia (MM'15)*, Brisbane, Australia, 2015, pp. 685–688.

[10] A. Toshniwal, S. Taneja, A. Shukla, K. Ramasamy, J. M. Patel, S. Kulkarni, J. Jackson, K. Gade, M. Fu, J. Donham, N. Bhagat, S. Mittal, and D. Ryaboy, "Storm@twitter," in *Proc. 2014 ACM International Conference on Management of Data (SIGMOD'14)*, Snowbird, Utah, USA, 2014, pp. 147–156.

[11] L. Neumeyer, B. Robbins, A. Nair, and A. Kesari, "S4: Distributed Stream Computing Platform," in *Proc. 2010 IEEE International Conference on Data Mining Workshops (ICDMW)*, Sydney, NSW, Australia, 2010, pp. 170–177.

[12] R. Gandhi, A. Gupta, A. Povzner, W. Belluomini, and T. Kaldewey, "Mercury: Bringing Efficiency to Key-value Stores," in *Proc. 6th International Systems and Storage Conference (SYSTOR'13)*, Haifa, Israel, 2013.

[13] H. Lim, D. Han, D. Andersen, and M. Kaminsky, "MICA: A Holistic Approach to Fast In-memory Key-value Storage," in *Proc. 11th USENIX Conference on Networked Systems Design and Implementation (NSDI'14)*, Seattle, WA, USA, 2014, pp. 429–444.

[14] C. Mitchell, Y. Geng, and J. Li, "Using One-sided RDMA Reads to Build a Fast, CPU-efficient Key-value Store," in *Proc. 2013 USENIX Conference on Annual Technical Conference (USENIX ATC'13)*, San Jose, CA, USA, 2013, pp. 103–114.

[15] B. Hindman, A. Konwinski, M. Zaharia, A. Ghodsi, A. Joseph, R. Katz, S. Shenker, and I. Stoica, "Mesos: A Platform for Fine-grained Resource Sharing in the Data Center," in *Proc. 8th USENIX Conference on Networked Systems Design and Implementation (NSDI'11)*, Boston, MA, USA, 2011, pp. 295–308.

[16] V. Vavilapalli, S. Seth, B. Saha, C. Curino, O. O'Malley, S. Radia, B. Reed, E. Baldeschwieler, A. Murthy, C. Douglas, S. Agarwal, M. Konar, R. Evans, T. Graves, J. Lowe, and H. Shah, "Apache Hadoop YARN: Yet Another Resource Negotiator," in *Proc. 4th Annual Symposium on Cloud Computing (SOCC'13)*, Santa Clara, CA, USA, 2013.

[17] (2016) Apache Hadoop 2.7.2 HDFS Users Guide. [Online]. Available: https://hadoop.apache.org/docs/current/hadoop-project-dist/hadoop-hdfs/HdfsUserGuide.html

[18] (2016) Welcome to Swift's documentation! [Online]. Available: http://docs.openstack.org/developer/swift/

[19] (2016) Amazon Simple Storage Service (Amazon S3). [Online]. Available: https://aws.amazon.com/s3

[20] R. M. Karp, *Complexity of Computer Computations*. Springer US, 1972, pp. 85–103.

[21] (2016) XenServer — Open Source Server Virtualization. [Online]. Available: http://xenserver.org/

[22] (2016) Examples — Apache Spark. [Online]. Available: http://spark.apache.org/examples.html