Accelerating MapReduce on Commodity Clusters: An SSD-Empowered Approach

Bo Wang, *Member, IEEE*, Jinlei Jiang, *Member, IEEE*, Yongwei Wu, *Member, IEEE*, Guangwen Yang, *Member, IEEE*, and Keqin Li, *Fellow, IEEE*

Abstract—MapReduce, as a programming model and implementation for processing large data sets on clusters with hundreds or thousands of nodes, has gained wide adoption. In spite of the fact, we found that MapReduce on commodity clusters, which are usually equipped with limited memory and hard-disk drive (HDD) and have processors of multiple or many cores, does not scale as expected as the number of processor cores increases. The key reason for this is that the underlying low-speed HDD storage cannot meet the requirement of frequent IO operations. Though in-memory caching can improve IO, it is costly and sometimes cannot get the desired result either due to memory limitation. To deal with the problem and make MapReduce more scalable on commodity clusters, we present mpCache, a solution that utilizes solid-state drive (SSD) to cache input data and localized data of MapReduce tasks. In order to make a good trade-off between cost and performance, mpCache proposes ways to dynamically allocate the cache space between the input data and localized data and to do cache replacement. We have implemented mpCache in Hadoop and evaluated it on a 7-node commodity cluster by 13 benchmarks. The experimental results show that mpCache can gain an average speedup of 2.09× when compared with Hadoop, and can achieve an average speedup of 1.79x when compared with PACMan, the latest in-memory optimization of MapReduce.

Index Terms—Big data, data caching, MapReduce, scheduling

1 Introduction

1.1 Motivation

The human society has stepped into the big data era where applications that process terabytes or petabytes of data are common in science, industry and commerce. Usually, such applications are termed IO-intensive applications, for they spend most time on IO operations. Workloads from Facebook and Microsoft Bing data centers show that IO-intensive phase constitutes 79 percent of a job's duration and consumes 69 percent of the resources [1].

MapReduce [2] is a programming model and an associated implementation for large data sets processing on clusters with hundreds or thousands of nodes. It adopts a data parallel approach that first partitions the input data into multiple blocks and then processes them independently using the same program in parallel on a certain computing platform (typically a cluster). Due to its scalability and ease of programming, MapReduce has been adopted by many companies, including Google [2], Yahoo [3], Microsoft [4], [5], and Facebook [6]. Nowadays we can see MapReduce applications in a wide range of areas such as distributed sort, web link-graph reversal, term-vector per host, web log analysis, inverted index construction, document clustering, collaborative filtering, machine learning, and statistical machine translation, to name but just a few. Also, the MapReduce implementation has been adapted to computing environments other than traditional clusters, for example, multi-core systems [7], [8], desktop grids [9], volunteer computing environments [10], dynamic cloud environments [11], and mobile environments [12].

Along with the evolution of MapReduce, great progress has also been made with hardware. Nowadays it is common for commodity clusters to have processors of more and more in-chip cores (referred to as *many-core cluster* hereafter) [13], [14]. While MapReduce scales well with the increase of server number, its performance improves less or even remains unchanged with the increase of CPU cores per server. Fig. 1 shows the execution time of *self-join* with varying number of CPU cores per server on a 7-node many-core cluster, where the line with pluses denotes the time taken by Hadoop and the line with squares denotes the time in an ideal world. As the number of CPU cores increases, the gap between the two lines gets wider and wider.

The fundamental reason (refer to Section 2 for a detailed analysis) behind this is that the underlying low-speed HDD storage cannot meet the requirements of MapReduce frequent IO operations: in the Map phase, the model reads raw input data to generate sets of intermediate key-value pairs, which are then written back to disk; in the Shuffle phase, the model reads the intermediate data out from the disk once again and sends it to the nodes to which Reduce tasks are scheduled. In addition, during the whole execution of

• B. Wang and G. Yang are with the Department of Computer Science and Technology, Tsinghua National Laboratory for Information Science and Technology, Ministry of Education Key Laboratory for Earth System Modeling, Center for Earth System Science, Tsinghua University, Beijing 100084, China. E-mail: bo-wang11@mails.tsinghua.edu.cn, ygw@tsinghua.edu.cn.
• J. Jiang and Y. Wu are with the Department of Computer Science and Technology, Tsinghua National Laboratory for Information Science and Technology, Tsinghua University, Beijing 100084, China, Research Institute of Tsinghua University in Shenzhen, Shenzhen 518057, China, and Technology Innovation Center at Yinzhou, Yangtze Delta Region Institute of Tsinghua University, Ningbo 315000, China.
E-mail: {jjlei, wuyw}@tsinghua.edu.cn.
• K. Li is with the Department of Computer Science, State University of New York at New Paltz, NY 12561. E-mail: lik@newpaltz.edu.

Manuscript received 19 Jan. 2016; revised 28 May 2016; accepted 28 July 2016. Date of publication 0 . 0000; date of current version 0 . 0000.
Recommended for acceptance by J. Chen, H. Wang, and M. Parashar.
For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below.
Digital Object Identifier no. 10.1109/TBDATA.2016.2599933

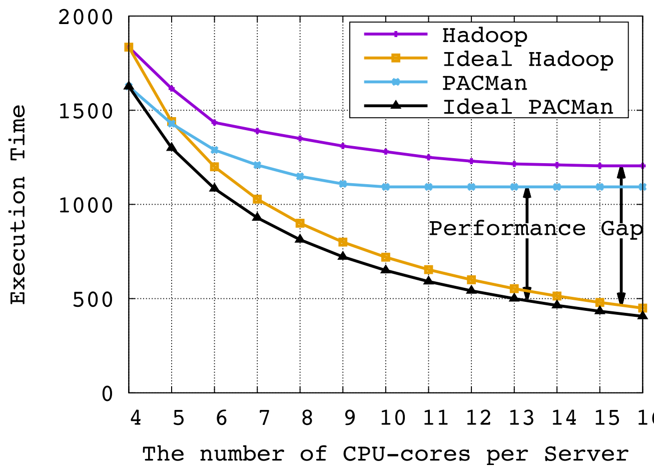


Fig. 1. Execution time of *self-join* with varying number of CPU cores per server using the settings in Section 4. The Input Data is of 60 GB.

jobs, temporary data is also written to local storage when memory buffer is full. Although more tasks can run concurrently in theory as more CPU cores are equipped, the IO speed of the storage system which backs MapReduce remains unchanged and cannot meet the IO demand of high-concurrent tasks, resulting in slightly improved or even unchanged MapReduce performance.

Indeed, the IO bottleneck of hard disk has long been recognized and many efforts have been made to eliminate it. The research work can be roughly divided into two categories.

The first category tires to cache *hot* data in the memory [15], [16], [17], [18]. Since the IO speed of memory is orders of magnitude faster than that of HDD, data in memory can be manipulated more quickly. Only hot data is cached because only limited volume of memory is available due to the high cost (compared with HDD). For parallel computing, memory is also a critical resource. Many parallel computing frameworks (e.g., Apache YARN [19]) use self-tuning technology to dynamically adjust task parallelism degree (TPD, which is the number of concurrent running tasks) according to available CPU-cores and memory. Caching data in memory inevitably occupies memory and makes the available memory for normal tasks operation drop, thus reducing the TPD and the performance. For memory-intensive machine-learning algorithms such as *k-means* and *term-vector*, which consume very large volume of memory during execution, the thing would get even worse—their TPD would drop significantly due to reduced memory for normal operation, leaving some CPU cores idle. Fig. 1 also illustrates this point by the case of PACMan [1], which is the latest work that utilizes memory caching to improve MapReduce. Although adding more memory could alleviate the situation, the volume of data grows even faster, meaning more memory is needed to cache data to get the benefit. Taking cost into consideration, it is not cost-effective to improve IO speed by in-memory caching.

The second category tries to use new storage medium of high-IO speed to replace HDD [20], [21], [22], [23]. Flash-based SSD is such a most popular storage medium. Since SSD does not have mechanical components, it has lower access time and less latency than HDD, making it an ideal storage medium for building high performance storage systems. However, the cost of building a storage system totally with SSDs often goes over the budget of most commercial data centers. Even considering the trend of SSD price dropping, the average per GB cost of SSD is still unlikely to reach the level of hard disks in the near future [24]. Thus, we believe using SSD as a cache of hard disks is a good choice to improve IO speed as did in [25], [26], [27], and [28].

1.2 Our Contributions

Taking both performance and cost into consideration, this paper presents mpCache (a preliminary version has been published in [29]), a solution that tries to accelerate MapReduce on commodity clusters via SSD-based caching. mpCache not only boosts the speed of storage system (thus eliminating the IO bottleneck of HDD) for IO-intensive applications but also guarantees TPD of memory-intensive jobs. The contributions of our paper are as follows.

- We identify the key cause of the poor performance of MapReduce applications on many-core clusters as the underlying low-speed HDD storage system cannot afford high concurrent IO operations of MapReduce tasks.
- We propose mpCache, an SSD-empowered cost-effective cache solution that caches both Input Data and Localized Data of MapReduce jobs in SSD to boost IO operations. In order to get the best benefit of caching, a mechanism is also put forward to dynamically adjust the SSD allocation between Input Cache and Localized Cache.
- We present a cache replacement scheme that takes into consideration not only replacement cost, data set size, and access frequency, but also the all-or-nothing characteristic of MapReduce caching [1].
- Extensive experiments are conducted to evaluate mpCache. The experimental results shows that mpCache can get an average speedup of 2.09× when compared with standard Hadoop and an average speedup of 1.79× when compared with PACMan.

The rest of this paper is organized as follows. Section 2 gives a brief introduction to MapReduce and analyzes the reasons why MapReduce applications perform poorly on many-core clusters. Section 3 describes the key ideas and algorithms of mpCache. Section 4 shows the experimental results. Section 5 reviews the related work and the paper ends in Section 6 with some conclusions.

2 Problem Analysis

In this section, we first give a brief introduction to MapReduce, and then set out to find out the bottleneck of MapReduce applications on many-core clusters.

2.1 Overview of MapReduce

A MapReduce [2] program is composed of a Map function and a Reduce function, where the Map function is used to process the key-value pairs associated with the input data (supplied by a certain distributed file system or database) to generate a set of intermediate key-value pairs and the Reduce function is used to merge all intermediate values associated with the same intermediate key. The program is executed by a runtime, the core part of a MapReduce framework that is in charge of such things as reading in and

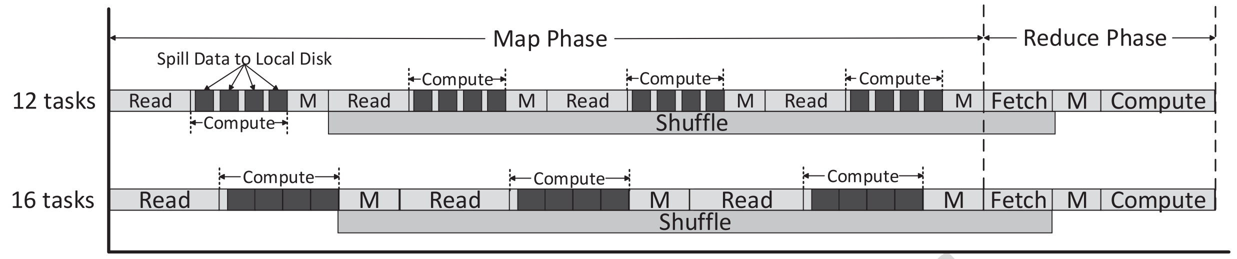


Fig. 2. Diagrammatic sketch of MapReduce performance issues with different concurrent tasks. **M** in the figure denotes *Merge*.

partitioning the input data, scheduling tasks across the worker nodes, monitoring the progress of tasks execution, managing all the communications between nodes, tolerating the fault encountered, and so on.

There are many MapReduce frameworks available and in this paper, we base our prototype on YARN, the latest version of Apache Hadoop, which is probably the most popular open-source implementation of MapReduce model.

The execution of a MapReduce program consists of three phases, that is, the Map phase, the Shuffle phase, and the Reduce phase. Fig. 2 shows the execution of a MapReduce job from the perspective of IO operations. Details are as follows.

In the Map phase, each map task *Read*s in the data block (from the source specified by the job), and runs the user-providing Map function to generate some key-value pairs (called intermediate results) that are stored first in a memory buffer and then flushed to local disk as a file (called *spill* file in Hadoop) when the buffer runs out. The spill procedure repeats until the end of the Map task, generating multiple spill files. After that, spill files of the same Map task are *Merged* (denoted by **M** in the figure) into a single file and written back to local disks for the purpose of fault-tolerance.

In the Reduce phase, the reduce task first *Fetches* input data from all the Map nodes and *Merges* (denoted by **M** in the figure) the fetched data into a single file. Then the user-providing Reduce function is executed to process the data. Since all the temporary results of *Spill* and *Merge* procedures and the outputs of Map function are written to local storage, they are called *Localized Data* in Hadoop.

Between Map phase and Reduce phase is the Shuffle phase that is employed to sort the Map-generating results by the key and pipeline data transfer between Mappers and Reducers. Since Reduce tasks in a MapReduce job will not execute until all Map tasks finish, the pipelining mechanism in the Shuffle phase would save a large part of data transfer time and thus improve performance.

All the three phases involve IO operations multiple times. For example, disk manipulation occurs two times (reading data from and writing data to disks) in the Map phase, while during the Shuffle phase, disk operations will happen at both the Mapper and the Reducer sides—data is read out from the disks of the Mappers, sent over the network, and then written to the disks of the Reducers. Since the speed of hard disks cannot match that of CPU, IO operations are time-consuming and thus limit tasks throughput. With IO operations improved by SSD-based caching at both Mappers and Reducers, the computation process will be accelerated accordingly. That is the basic idea behind our work here.

2.2 Bottleneck Analysis

With the development of hardware technology, many-core servers get common in data centers [13], [14]. For example, each server in our experiment has 16 CPU cores. More cores on a node usually means the node could process more tasks concurrently. In a typical Hadoop configuration, one CPU core corresponds to one Map/Reduce task. Thus, one node could run concurrently as many tasks as the number of CPU cores in theory. We define the number of concurrently running tasks (i.e., TPD) as *wave-width* since tasks in MapReduce are executed wave by wave. Then we have $wave\# = ceil\,(tasks\#/wave\text{-}width)$. Obviously, the bigger the *wave-width*, the smaller the *wave#* and the shorter the job execution time.

We examine the job execution time by varying the wave-width. As shown in Fig. 1, the execution time of the job reaches the minimum value when the wave-width is 12 and this value remains unchanged even if the wave-width increases. Consider a job consisting of 288 Map tasks and running on a many-core cluster of 6 nodes. Obviously, each node should process $288/6 = 48$ Map tasks. When each node is equipped with 12 CPU cores, the number of concurrently running Map tasks (i.e., the *wave-width*) is 12. In this case we get $48/12 = 4$ waves for the job. On the contrary, when each node is equipped with 16 CPU cores, we get *48/16=3* waves for the same job. Ideally, if the node could provide sufficient resources such as CPU, memory, and IO, job execution in 3 waves should take 3/4 time of that running in 4 waves. But as shown in Fig. 2, the Map time remains unchanged when the wave-width increases from 12 to 16. Please note that the execution time of different waves might be different in the real world and here we just use Fig. 2 to simplify the problem description.

The reason for unchanged execution time is that the IO-intensive operations (i.e., *Read*, *Spill*, *Merge*) slow down due to the IO bottleneck of the underlying storage system. For commodity clusters, they are usually equipped with HDDs. Since the MapReduce job performance is bounded by the low IO speed of HDD, it is natural that the performance remains unchanged with the increase of CPU cores. This phenomenon was also reported by PACMan [1]—the authors found that *"the client saturated at 10 simultaneous tasks and increasing the number of simultaneous tasks beyond this point results in no increase in aggregate throughput"*.

In summary, the bottleneck of MapReduce applications running on many-core clusters is the IO speed of storage system. As mentioned in Section 1, caching data in memory and building total SSD storage systems have several disadvantages, impeding them to be used for memory-intensive applications. Therefore, we propose mpCache, an SSD-based cache solution that caches both Input Data and

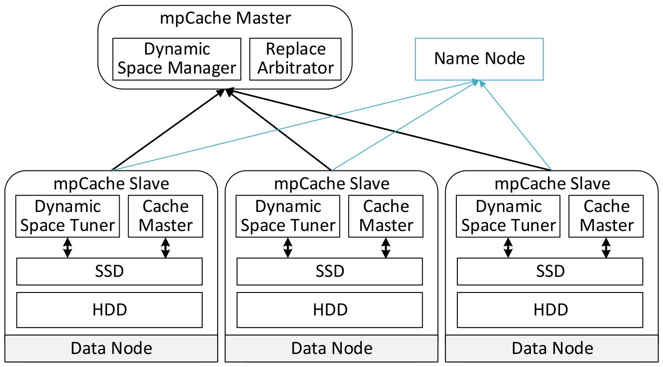


Fig. 3. mpCache architecture. It adopts a master-slave architecture with mpCache Master managing mpCache Slaves locating on every data node. Thin lines represent control flow and thick arrows denote data flow. Such an architecture is in accordance with that of the underlying distributed file system that backs up the MapReduce framework.

Localized Data to provide high IO speed and thus speed up all the critical operations—*Read*, *Spill*, and *Merge*. Besides, mpCache allows dynamically adjusting the space between Input Cache and Localized Cache to make full use of the cache to get the best benefit.

3 mpCache Design

This section details the design of mpCache.

3.1 Architecture

In accordance with the distributed file system that backs the MapReduce framework up, mpCache adopts a master-slave architecture, as shown in Fig. 3, with one mpCache Master and several mpCache Slaves. mpCache Master acts as a coordinator to globally manage mpCache slaves to ensure that a job's input data blocks, which are cached on different mpCache slaves, present in an all-or-nothing manner, for some prior research work [1] found that a MapReduce job can only be speeded up when inputs of all tasks are cached. We can see from the figure that SSD-based cache space locates in each data node of the underlying distributed file system of the MapReduce framework. It is a distributed caching scheme.

mpCache Master consists of two components—*Dynamic Space Manager* and *Replace Arbitrator*. *Dynamic Space Manager* is responsible for collecting the information about dynamic cache space allocation from each mpCache Slave and recording into history the job type and input data set size. *Replace Arbitrator* leverages the cache replacement scheme.

mpCache Slave locates on each data node and also consists of two components, that is, *Dynamic Space Tuner* and *Cache Master*. *Dynamic Space Tuner* is deployed to adjust the space allocation between Input Cache (for caching Input Data) and Localized Cache (for caching Localized Data). *Cache Master* is in charge of serving cached data blocks and caching new ones.

During job execution, *Cache Master* on each data node intercepts the data reading requests of Map tasks, and checks whether the requested data block is cached. If so, *Cache Master* servers the data blocks from cache and informs *Replace Arbitrator*, which resides with mpCache Master, of the block hit. Otherwise, the data block will be cached. In the case that there is no enough cache space, *Cache Master* will send cache replacement request to *Replace Arbitrator* and do cache replacement according to the information returned by *Replace Arbitrator* to make room for the newly requested data block.

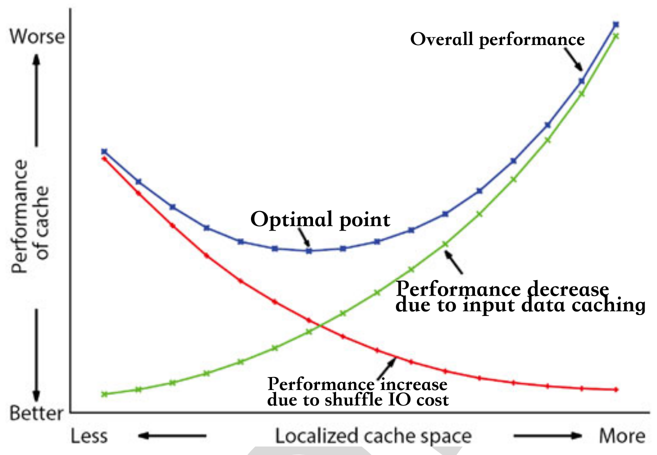


Fig. 4. Balancing the size of the input cache and the localized data cache is necessary to get best benefit of caching.

3.2 Dynamic Space Allocation

As described in Section 2.1, *Read*, *Spill*, and *Merge* are all IO-intensive procedures, imposing restrictions on performance when running on many-core clusters. Since both reading in the job input data and reading/writing *Localized Data* involve IO operation, mpCache caches both data. Because the cache space is limited and different jobs may have different characteristics in terms of input data size and localized data size, we must smartly allocate the space between Input Cache and Localized Cache to get the best benefit. Fig. 4 illustrates this, where the x-axis represents the Localized Cache size and y-axis represents the total benefit of caching.

As shown in the figure, the Input Cache size as well as the corresponding caching benefit decreases as the Localized Cache size increases. With a larger Localized Cache, the cost of writing/reading Localized Data reduces and the cache performance improves. At a certain point, the two lines cross and total benefit of caching reaches the best value. Please note that this figure is just an illustration. In the real world, the optimal point may vary between jobs, for different jobs may produce quite different volumes of Localized Data, according to which jobs can be categorized into *shuffle-heavy*, *shuffle-medium*, and *shuffle-light*. Therefore, we must dynamically adjust the space allocation to ensure the best benefit of caching.

It is in this sense that *Dynamic Space Tuner* is introduced. As shown in Fig. 5, *Dynamic Space Tuner* divides the whole cache space into three parts, that is, Input Cache, Dynamic Pool, and Localized Cache. Since the distributed file systems (e.g., GFS [30] and HDFS [31]) that back MapReduce applications up store data in the unit of block, we also divide Dynamic Pool into many blocks. Blocks in Dynamic Pool

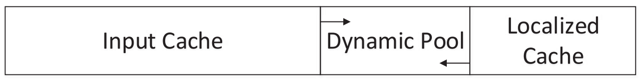


Fig. 5. The whole cache space is divided into three parts, namely Input Cache, Dynamic Pool and Localized Cache. Blocks in Dynamic Pool are used on demand as Input Cache or Localized Cache depending on the workload to get the most benefit of caching.

will be used on demand as Input Cache or Localized Cache. During job execution, *Dynamic Space Tuner* constantly monitors the utilization of the Localized Cache. When the cache space runs out, *Dynamic Space Tuner* checks if there are free blocks in Dynamic Pool. If not, *Dynamic Space Tuner* will remove some cached input data from Dynamic Pool using the same scheme described in Section 3.3. Then the just freed blocks are used as Localized Cache one by one. If the utilization of Localized Cache is below the *guard value*, which is set to 0.5 in our implementation, all blocks used as Localized Cache in Dynamic Pool are reclaimed.

3.3 Input Data Cache Model

Since the cache size is limited, it is necessary to do cache replacement to guarantee the desired benefit. Here we explain the cache model used for input data.

3.3.1 Admission Control Policy

We use an admission control policy in the first place to decide whether or not an object should be cached. Since the cache space is limited and input data size varies job by job, caching input data of one job may mean purging the data of the other jobs from the cache. Too frequent cache replacement may result in the case that some cached data will never be used during the whole lifetime in the cache, reducing the benefit of caching. It is the duty of admission control policy to avoid this happening.

The admission control policy utilizes an auxiliary facility to maintain the *identities* of input data sets of different jobs. For each data set recorded in this facility, its access number and the last access time are also maintained. Each time the data set is accessed, the corresponding access number is increased by 1 and the record is updated. The auxiliary facility is kept in memory, for it just maintains metadata about the data sets rather than the data sets themselves and will not consume too much memory.

Using the admission control policy, we would like to ensure that, at a certain time when some data is accessed, the potential incoming input data jd_i gets popular enough so that it can be loaded into the cache to get more benefit. The process is as follows.

If there is enough free space for jd_i, we simply load jd_i into the main cache. Otherwise, we check to see if jd_i has been recorded by the auxiliary facility. If not, we will record the related information with the auxiliary facility rather than put jd_i itself into the main cache. In the case that jd_i does occur in the auxiliary facility, we proceed to see if some cache replacement is necessary. By necessary we mean the cache replacement is profitable, or in other words, it can bring in some benefit for the performance. This is done by comparing the value $1/(Size(jd_i)\Delta_{jd_i})$ with the sum $\sum_j 1/(Size(jd_j)\Delta_{jd_j})$, where jd_j is the candidate data sets to be replaced that is determined by the replacement scheme described in Section 3.3.2, $Size(jd)$ is the number of blocks in data set *jd*, and Δ_{jd} is the data set access distance, which is defined as the number of data set accesses between the last two times that the data set *jd* was accessed. In the case that *jd* is accessed for the first time, Δ_{jd} is defined as the number of all data set accesses before that, and in the case of successive accesses, Δ_{jd} is set to 0.01. A candidate data set jd_i can be loaded into the main cache if and only if $1/(Size(jd_i)\Delta_{jd_i}) > \sum_j 1/(Size(jd_j)\Delta_{jd_j})$. It is easy to see that the data set access distance defined in such a way ensures those data sets being frequently accessed (and thus, of smaller data set access distance) have greater chance to be loaded into the main cache.

3.3.2 Main Cache Replacement Scheme

We now describe the cache replacement scheme adopted by the main cache. For each data set in the main cache we associate it with a *frequency* $Fr(jd)$, which is the number of times that jd has been accessed since it was loaded into the main cache. Besides, a priority queue is maintained. When a data set of a certain job is inserted into the queue, it is given the priority $Pr(jd)$ using the following way:

$$Fr(jd) = Blocks\_Access(jd)/Size(jd) \quad (1)$$

$$Pr(jd) = Full + Clock + Fr(jd), \quad (2)$$

where *Blocks_Access(jd)* is the total number of times that all blocks of data set *jd* are accessed; *Full* is a constant *bonus* value assigned to the data set whose blocks are all in the main cache (in favor of the *all-or-nothing* characteristic of MapReduce cache [1]); *Clock* is a variable used by the priority queue that starts at 0 and is set to $Pr(jd_{evicted})$ each time a data set $jd_{evicted}$ is replaced.

Once the mpCache Master receives a data access message from an mpCache Slave, Algorithm 1 is used to update $Pr(jd)$ of the corresponding data set indicated by the message. Since *Clock* increases each time a data set is replaced and the priority of a data set that has not been accessed for a long time was computed using an old (hence small) value of *Clock*, cache replacement will happen on that data set even if it has a high *frequency*. This "aging" mechanism avoids the case that a once frequently-accessed data set, which will never be used in the future, unnecessarily occupies the cache and thus degrades performance. *To_Del* in Algorithm 1 is a list of tuples that have the format $<data\_node, blocks_{evicted}>$. It is introduced for *Replace Arbitrator* to record those data blocks on each data node that have already been selected by *Replace Arbitrator* as outcasts but the corresponding *Cache Master* is not notified of.

4 Evaluation

We implement mpCache by modifying Hadoop distributed file system HDFS (version 2.2.0) [3] and use YARN (version 2.2.0) to execute the benchmarks.

4.1 Platform

The cluster used for experiments consists of 7 nodes. Each node has two eight-core Xeon E5-2640 v2 CPUs running at 2.0 GHz, 20 MB Intel Smart Cache, 32 GB DDR3 RAM, one 2 TB SATA hard disk and two 160 GB SATA Intel SSDs configured as RAID 0. All the nodes run Ubuntu 12.04, have a Gigabit Ethernet card connecting to a Gigabit Ethernet switch. Though we have *160 * 2 = 320 GB* SSD on each node, we only use 80 GB as cache in our experiment to illustrate the benefit of mpCache. Such a value is selected because the data sets used for experiments are not large (the maximum volume of data manipulated during our experiments is about 169 GB in the case of *tera-sort*) and too large cache space would hold all data, making cache replacement unnecessary. In the real world, the input data sets of MapReduce may be of terabytes or even petabytes, well beyond the SSD capacity.

Algorithm 1. Main Cache Replacement Scheme

```
1: if the requested block $bk$ is in the cache then
2:   $jd \leftarrow$ the data set to which $bk$ belongs
3:   $Blocks\_Access(jd) \leftarrow Blocks\_Access(jd)$+1
4:   update $Pr(jd)$ using Equations (1)-(2) and move $jd$ accordingly in the queue
5: else
6:   if no cache replacement is necessary then
7:     cache $bk$
8:   else
9:     $mpSlave \leftarrow$ the source of the data access request
10:    $data\_node \leftarrow$ the data node that $mpSlave$ is seated
11:    if $To\_Del$.hasRecord($data\_node$) then
12:      send $blocks_{evicted}$ to $mpSlave$, and replace $blocks_{evicted}$ with $bk$ at $mpSlave$
13:    else
14:      $jd_{evicted} \leftarrow$ the data set with lowest priority in the queue
15:      $Clock \leftarrow Pr(jd_{evicted})$
16:      $blocks_{evicted} \leftarrow$ all the blocks of $jd_{evicted}$
17:      send $blocks_{evicted}$ to $mpSlave$, and replace $blocks_{evicted}$ with $bk$ at $mpSlave$
18:      $allnodes \leftarrow$ all the data nodes that store $blocks_{evicted}$
19:      for $dn \in allnodes$ do
20:        $To\_Del$.addRecord($< dn, blocks_{evicted} >$)
21:      end for
22:    end if
23:  end if
24:  $Blocks\_Access(jd) \leftarrow Blocks\_Access(jd)$+1
25:  if all the blocks of $jd$ are cached then
26:    $Full = BONUS\_VALUE$
27:  else
28:    $Full = 0$
29:  end if
30:  compute $Pr(jd)$ using Equation (2) and put $jd$ into the queue accordingly
31: end if
```

4.2 Benchmarks

We use 13 benchmarks released in PUMA [32], covering shuffle-light, shuffle-medium, and shuffle-heavy jobs. We vary the input data size of each benchmark from 1 to 20 times of the original data set. Input data size of each benchmark is shown in Table 1. *grep*, *word-count*, *inverted-index*, *term-vector*, and *sequence-count* use the same input data, which is a text file downloaded from wikipedia. *histogram-rating*, *histogram-movies*, *classification*, and *k-means* use the same data set, which is classified movie data downloaded from Netflix. *self-join*, *adjacency-list*, *ranked-inverted-index*, and *tera-sort* use data set downloaded from PUMA.

Since the input data size has Zipf-like frequency distribution [33], we associate a probability with each data size as

$$f(k; s, N) = \frac{1/k^s}{\sum_{i=1}^{N} 1/i^s}. \quad (3)$$

TABLE 1
Input Data Size of Benchmarks

Data Source	Data Size	Benchmarks
wikipedia	k*4.3G	grep word-count inverted-index term-vector sequence-count
netflix data	k*3.0G	histogram-rating histogram-movies classification k-means
PUMA-I	k*3.0G	self-join
PUMA-II	k*3.0G	adjacency-list
PUMA-III	k*4.2G	ranked-inverted-index
PUMA-IV	k*3.0G	tera-sort

($k = 1, 2, \ldots, 20$).

Since 20 times of data size are generated, we set N to 20. For the Zipf parameter s, we set it to 1 if not specially mentioned. Table 2 summarizes the characteristics of the benchmarks in terms of input data size (take $k = 10$ for example), data source, the number of Map/Reduce tasks, shuffle size, and execution time on Hadoop.

Shuffle-light jobs, including *grep*, *histogram-ratings*, *histogram-movies*, and *classification*, have very little data transfer in the shuffle phase. Shuffle-heavy jobs, which have a very large data size to be shuffled (as shown in Table 2, almost the same as the input data size), include *k-means*, *self-join*, *adjacency-list*, *ranked-inverted-index*, and *tera-sort*. The shuffle data size of shuffle-medium jobs is between that of shuffle-light and shuffle-heavy ones, including *word-count*, *inverted-index*, *term-vector*, and *sequence-count*.

When submitting a job to the cluster, we randomly select one from the 13 benchmarks, and set the input data size according to the attached probability. Each time we submit a job, we use "echo 1 > /proc/sys/vm/drop_caches" command to clear memory cache and make sure the data is read from mpCache other than memory.

4.3 Experimental Results

Our experiment consists of 5 parts: i) Section 4.3.1 compares mpCache with standard Hadoop and PACMan, the state-of-the-art way of MapReduce optimization by in-memory caching; ii) Section 4.3.2 compares mpCache with traditional cache replacement policies such as LRU (Least Recently Used) and LFU (Least-Frequently Used); iii) Section 4.3.3 shows mpCache behavior with different numbers of CPU cores per server; iv) Section 4.3.4 shows the adaptability of mpCache to the cache size; v) Section 4.3.5 shows the adaptability of mpCache to the Input Data size.

4.3.1 Comparison with Hadoop and PACMan

We compare the execution time of benchmarks on mpCache with that on both Hadoop and PACMan. We run the benchmarks with mpCache, Hadoop, and PACMan respectively and get the average value. PACMan uses memory to cache input data and the bigger the cache size, the more the cached data and thus the faster the Map phase. However, the concurrent running tasks number in YARN is tightly related to

TABLE 2
Characteristics of the Benchmarks Used in the Experiment

Benchmark	Input size(GB)	Data source	#Maps & #Reduces	Shuffle size(GB)	Map&Reduce time on Hadoop(s)
grep	43	wikipedia	688 & 40	$6.9 * 10^{-6}$	222&2
histogram-ratings	30	netflix data	480 & 40	$6.3 * 10^{-5}$	241&5
histogram-movies	30	netflix data	480 & 40	$6.8 * 10^{-5}$	261&5
classification	30	netflix data	480 & 40	$7.9 * 10^{-3}$	286&5
word-count	43	wikipedia	688 & 40	0.318	743&22
inverted-index	43	wikipedia	688 & 40	0.363	901&6
term-vector	43	wikipedia	688 & 40	0.384	1,114&81
sequence-count	43	wikipedia	688 & 40	0.737	1,135&27
k-means	30	netflix data	480 & 4	26.28	450&2,660
self-join	30	puma-I	480 & 40	26.89	286&220
adjacency-list	30	puma-II	480 & 40	29.38	1,168&1,321
ranked-inverted-index	42	puma-III	672 & 40	42.45	391&857
tera-sort	30	puma-IV	480 & 40	31.96	307&481

the available CPU cores and the free memory, and consuming too much memory for data caching would decrease the parallelism degree of the tasks. We set the volume of memory used for cache to 12 GB as did in PACMan [1].

Fig. 6 shows the normalized execution time of the Map and Reduce phase. For shuffle-light jobs such as *grep*, *histogram-movies*, *histogram-ratings*, and *classification*, their execution time is short (about 241s, 253s, 279s, and 304s on Hadoop when *k = 10*) and most time is spent on data IO. Input data caching supplied by mpCache can accelerates the Map phase significantly (2.42× faster on average). In the Reduce phase, the speedup is not notable for three reasons: i) The Reduce phase of shuffle-light jobs is very short (about 2s, 4s, 4s, and 5s when *k = 10*); ii) Shuffle-light jobs have very little shuffle data (less than 10 MB); iii) The localized data size is so small (less than 1 MB) that caching localized data results in little acceleration. In all, mpCache gets a speedup of 2.23 times over Hadoop for shuffle-light jobs. When running the jobs with PACMan, each task performs well with 1 GB memory. PACMan and mpCache get the same parallelism degree of the tasks. Although in-memory caching could provide faster IO than SSD-based caching as mpCache does, the larger cache size provided and cache replacement scheme supplied ensure *a higher hit ratio* of mpCache than that of PACMan does (61.7 percent versus 38.5 percent). Therefore, mpCache performs even better than PACMan.

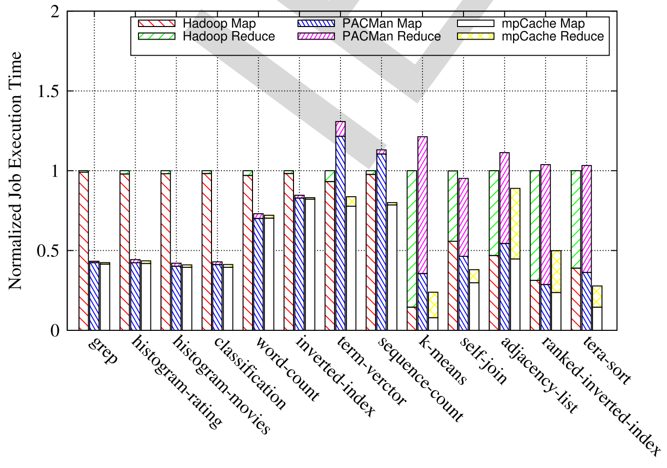


Fig. 6. Job execution time comparison with Hadoop and PACMan.

For shuffle-medium jobs such as *word-count*, *inverted-index*, *term-vector*, and *sequence-count*, their execution time is longer than that of shuffle-light jobs (about 779s, 932s, 1209s, and 1174s), caching Map input data only results in a speedup of 1.25 times averagely. The shuffle data size of these jobs is about 318~737 MB; the size of localized data is 1~3 GB; caching localized data would produce great benefit—the average speedup of the Reduce phase is 1.60 times. In all, mpCache can averagely get a speedup of 1.25 times over Hadoop for shuffle-medium jobs. With PACMan, *word-count* and *inverted-index* run well using 1 GB memory and the speedup got is almost the same as in the case of mpCache. For *term-vector* tasks that need at least 3 GB memory, the parallelism degree is 10 in Hadoop and 6 in PACMan. As a result, the performance of PACMan drops to 0.762 of the performance of Hadoop. The parallelism degree for *sequence-count*, whose task needs at least 2 GB memory, is 16 in Hadoop and 10 in PACMan, making the performance of PACMan drop to 0.868 of the performance of Hadoop.

For shuffle-heavy jobs such as *k-means*, *self-join*, *adjacency-list*, *ranked-inverted-index*, and *tera-sort*, both the shuffle data size and the localized data size are very big. Thus, caching Map input data and localized data reduces the time of Map and Reduce phases greatly. The Map phase time of *k-means*, *self-join*, *ranked-inverted-index*, and *tera-sort* is shorter than that of *adjacency-list* (1168s). Thus the speedup got for the former three jobs is 1.82~2.69 times, whereas the speedup got for the latter job is 1.04 times. Caching localized data also brings in great benefit—a speedup of 3.87 times would be got in the Reduce phase. In all, mpCache results in an average speedup of 2.65 times over Hadoop. For PACMan, the parallelism degree with *self-join*, *adjacency-list*, *ranked-inverted-index*, and *tera-sort*, each task of which needs 2 GB memory, is 10, resulting in the performance of PACMan dropping to 0.981 of the performance of Hadoop. As for *k-means*, the number of Reduce tasks is set to 4 (because it clusters the input data into 4 categories) and each task needs at least 8 GB memory. Since less memory is left for normal operation, PACMan spends 2.46× longer time in the Map phase than Hadoop does. In addition, it does no help to the heavy Reduce phase (2660s, taking about 86.2 percent of the whole job execution time). As a result, the performance of PACMan drops to 0.808 of the performance of Hadoop.

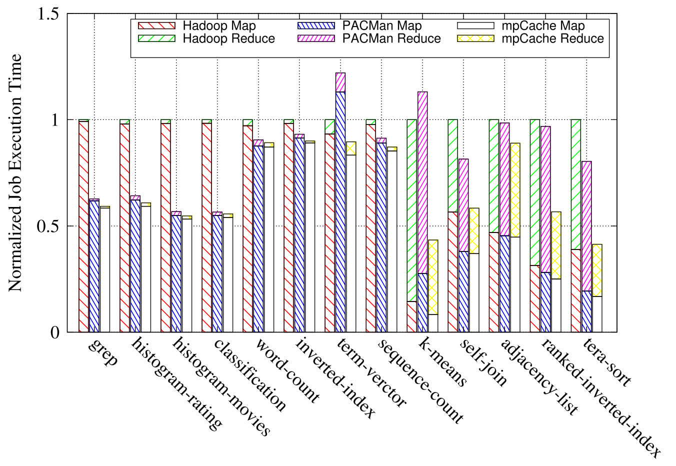


Fig. 7. Job execution time comparison with Hadoop and PACMan on the same cluster of 8 CPU cores.

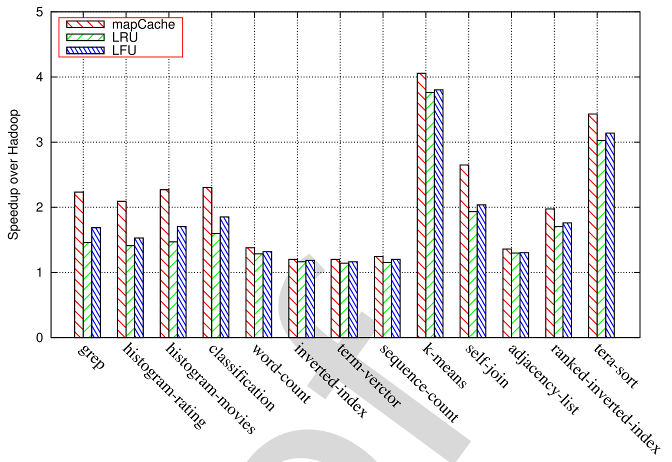


Fig. 8. Performance comparison with LRU and LFU.

PACMan used 12 GB memory for data cache and got considerable performance improvement over Hadoop MapReduce v1 [3], the TPD of which is determined by the "slots" number in the configuration file. Usually it is set to a constant value. Since both Hadoop and PACMan use the same configuration, they are of the same TPD. However, in MapReduce v2 (i.e., YARN [19]), the number of concurrent running tasks is determined by the number of free CPU cores and free memory, allocating memory for data cache inevitably reduces the TPD of some jobs.

In our cluster, each node has 16 CPU cores and 32 GB memory. Since PACMan used 12 GB memory for cache, the memory left for computing is 20 GB. When running "1 GB jobs" (jobs with each task consuming 1 GB memory, including *grep*, *histogram-rating*, *histogram-movies*, *classification*, *word-count*, and *inverted-index*) with PACMan, the TPD is 16, the same as that of Hadoop and mpCache. Therefore, PACMan gets a better performance than Hadoop and mpCache performs almost the same as PACMan. For other jobs, each task needs at least 2 GB memory (3 GB for *term-vector*, and 6 GB for *k-means*), and therefore the TPD of PACMan drops to 10 (6 of *term-vector*, and 3 of *k-means*). Although in-memory caching could significantly speedup the Map phase, the dropping of TPD slows down the job worse: as illustrated in Fig. 6, PACMan performs even worse than Hadoop for these "at least 2 GB" jobs.

For all these benchmarks, mpCache gains an average speedup of 2.09× when compared with the Hadoop, and an average speedup of 1.79× when compared with PACMan. Such improvements come from the speedup of IO operations. Since more data is read from the SSD-based cache rather than hard disks, computing resources waste due to lack of data is lowered and thus tasks can progress faster. Though the speed of SSD is slower than that of memory, the volume of SSD cache is much larger than that of memory cache. As a result, SSD-based cache also shows advantage over memory-based cache. This is why mpCache performs better than PACMan.

In order to better illustrate the in-memory caching effect of PACMan, we also do an experiment where only 8 CPU cores are used on each node for Hadoop, PACMan, and mpCache.

As shown in Fig. 7, for the case of 8 CPU cores, most benchmarks can run with the same TPD on Hadoop, mpCache, and PACMan except *term-vector* and *k-means*. For shuffle-light jobs, mpCache and PACMan run with the same TPD, getting 1.74× and 1.67× speedup over Hadoop respectively. For shuffle-medium jobs, in the 1 GB job case (*word-count* and *inverted-index*), the speedup got over Hadoop is 1.12× and 1.08× respectively; in the 3 GB job case (*term-vector*), Hadoop and mpCache run with TPD=8 whereas PACMan runs with TPD = 6. Thus PACMan has a longer Map phase time than Hadoop and the whole performance of PACMan is even worse than that of Hadoop. For shuffle-heavy jobs, the localized data size is also big. mpCache caches both input data and localized data, resulting in an average speedup of 1.63 times in the Map phase and 2.09 times in the Reduce phase. In contrast, PACMan gets an average speedup of 1.35 times in the Map phase and introduces no benefit in the Reduce phase. Totally, for all the benchmarks, mpCache gets an average speedup of 1.62 times, whereas PACMan gets an average speedup of 1.25 times.

4.3.2 Comparison with Traditional Cache Replacement Policies

We implement two traditional cache replacement policies, namely LRU and LFU. In our settings, mpCache gets an average hit ratio of 61.7 percent, while LRU gets an average hit ratio of 53.9 percent and LFU gets an average hit ratio of 68.3 percent. The resulted performance is shown in Fig. 8. Although LFU gets a higher hit ratio than mpCache does, mpCache takes all-or-nothing characteristic of MapReduce caching into consideration and deploys an auxiliary facility to prevent too frequent replacements, and therefore gets a higher speedup than LFU does. Compared with LRU, mpCache gets both higher hit ratio and speedup. With the cache space better utilized, it is natural that the IO operations and the consequent tasks execution are speeded up.

4.3.3 Adaptability to the Number of CPU Cores per Server

Fig. 9 shows mpCache's adaptability to the number of CPU cores per server, where the line with pluses denotes the execution time of Hadoop, the line with squares denotes the execution time of mpCache, and the line with asterisks denotes the execution time of Hadoop in an ideal world (i.e., with no constraint). mpCache scales well when the

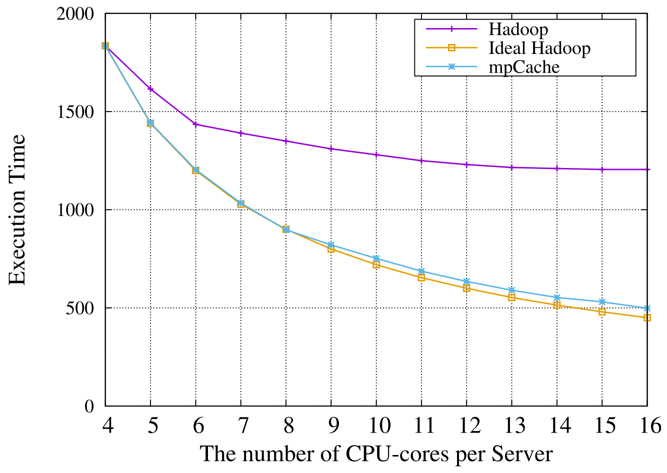


Fig. 9. Execution time of *self-join* varies with the number of CPU cores per server changing.

number of CPU cores per server increase. Its behavior is almost the same as the ideal case.

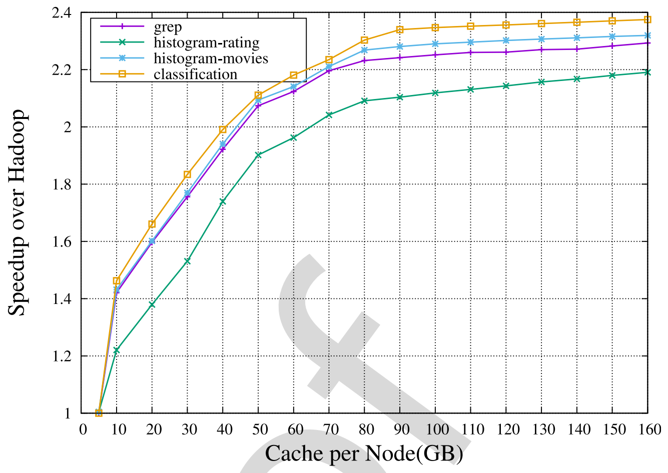
4.3.4 Adaptability to Cache Size

We now evaluate mpCache's adaptability to cache size by varying the available cache size of each mpCache Slave between 5 GB and 160 GB. The experimental results are shown in three sub-figures, i.e., Figs. 10a, 10b, and Fig. 10c, in accordance with the 3 categories of benchmarks.

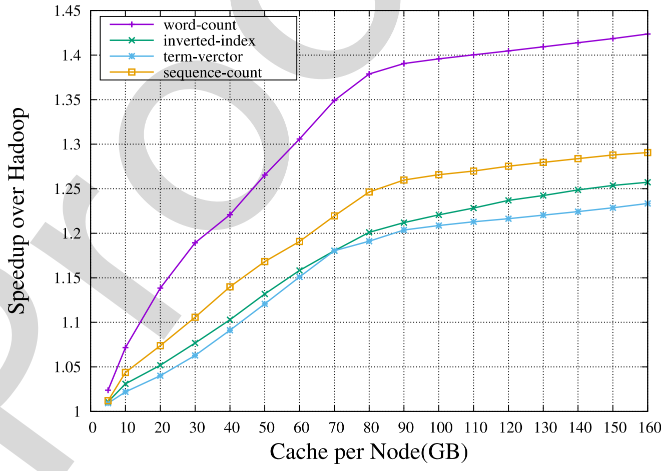
Fig. 10a shows the impact of cache size on *shuffle-light* benchmarks. All these benchmarks have very little shuffle date and very short Reduce phase (the Reduce phase is no greater than 2.1 percent of the whole time). Therefore, the Localized Cache occupies less space and most space is used as Input Space. The speedup of these benchmarks mainly comes from Input Data caching. When the cache size is 5 GB per node, the speedup is very small due to insufficient space to hold Input Data. As the cache size increases, the speedup grows significantly and a maximum value is obtained when the cache size is about 90 GB.

Fig. 10b shows the impact of cache size on *shuffle-medium* benchmarks. These benchmarks have some volume of shuffle data (no more than 1 GB), both Map and Reduce phase could be accelerated by caching. When the cache size per node is 5 GB, all Localized Data is cached, resulting in an average speedup of 59.99 percent in the Reduce phase. However, since the Reduce phase only takes 3.43 percent of the whole time, this only contributes 1.40 percent of the whole job speedup. As the cache size increases, the speedup grows due to the reduction of the Map phase time and a maximum value is reached when the cache size is about 100 GB.

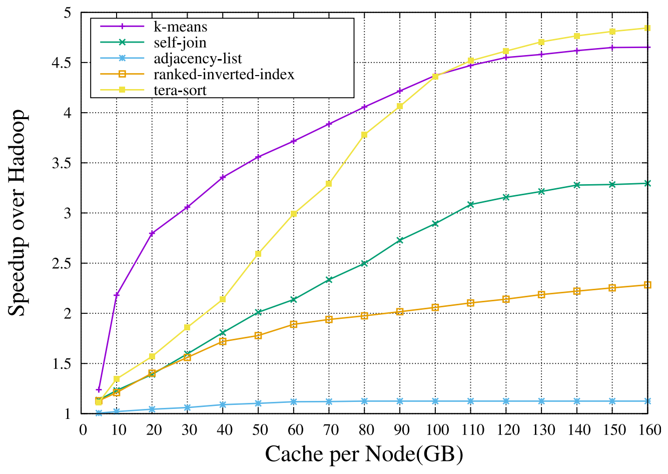
Fig. 10c shows the impact of cache size on *shuffle-heavy* benchmarks. These benchmarks have very large volume of shuffle data. When *tera-sort* runs with 30 GB input data, the localized data occupies as large as 32 GB space. Thus, when the cache size is below 40 GB, most cache is allocated to cache Localized Data, which is the main contribution of the speedup. As depicted in the figure, the *k-means* job gets higher speedup than *tera-sort* does when the cache size is below 100 GB and *tera-sort* gets higher speedup when the cache size is larger than 100 GB. The reason behind this is: the Reduce phase of *k-means* takes a very large portion of the whole execution time (85.53 percent) and larger volume of Localized Data



(a) shuffle-light



(b) shuffle-medium



(c) shuffle-heavy

Fig. 10. The impact of cache size on mpCache performance. For shuffle-light and shuffle-medium jobs, the cache space is mainly used for caching input data. Good benefit can be got when the cache size is 80 GB.

is *spilled* than the case of *tera-sort*. Therefore, caching Localized Data accelerates *k-means* faster than the case of *tera-sort*. When the cache size is below 40 GB, the gradient of *k-means* is bigger than that of *tera-sort*. When the cache size is above 40 GB, the increase of speedup is due to Input Data caching and the reduction of the Map phase time. Since *tera-sort* has smaller Map phase time than *k-means* (as shown in Table 2, when the

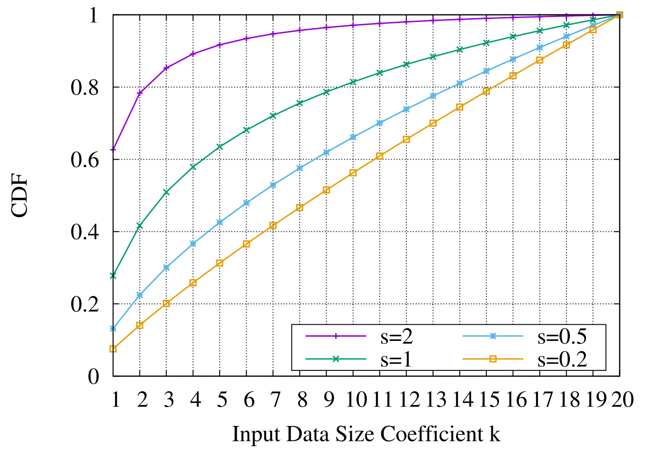


Fig. 11. Input data size distribution varies with Zipf parameter s. The greater the parameter, the larger the input data size.

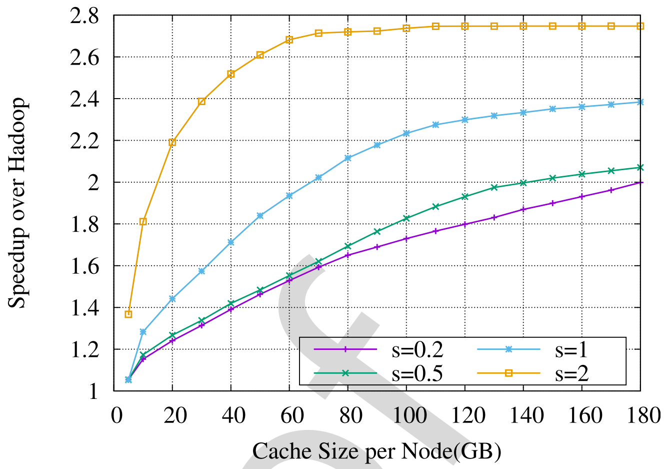


Fig. 12. The impact of Zipf parameter s on mpCache performance.

input data size is 30 GB, the Map phase time of *tera-sort* is 307s, while that of *k-means* is 450s), caching Input Data accelerates *tera-sort* faster than *k-means*, resulting in the same speedup at 100 GB and greater speedup beyond 100 GB. All the shuffle-heavy benchmarks get the maximum speedup when the cache size is about 130~140 GB. Among these benchmarks, the speedup of *adjacency-list* is smallest. The reason behind this is that both Map phase and Reduce phase are compute-intensive and take a long time. Since the critical resource of this benchmark is CPU, accelerating IO only improves the performance a little.

4.3.5 *Adaptability to Input Data Size*

We now evaluate mpCache's adaptability to the input data size by *ranked-inverted-index*. As described in Section 4.2, we attach a selection probability to each input data size using Zipf distribution, which is indicated by parameter s in Equation (3). By varying s between 0.2 and 2, we get different distributions of input data size. Fig. 11 shows input data size distribution with varying Input Data Size Coefficient, where the X-axis represents the Input Data Size Coefficient k and Y-axis indicates the CDF (cumulative distribution function) distribution probability. It can be found that the bigger the s, the higher the probability of small Input Data Size. For example, when $s = 2$, more than 80 percent of the Input Data size coefficient is below 3. In other words, more than 80 percent of the Input Data has a size below 12.6 GB.

Fig. 12 shows the average speedups of the benchmarks with varying s. It is easy to see that mpCache works well in all cases. With the same cache size, the bigger the s, the greater the speedup (a maximum value exists as illustrated by Fig. 4). With Fig. 11 the reason behind this is obvious: a bigger s means small input data size and thus less space is needed to cache all the data to get the same speedup.

5 Related Work

There is a lot of work about MapReduce. Below is the work most related to ours.

5.1 MapReduce Implementations

Due to the high impact, MapReduce, since the first release by Google [2], has been re-implemented by the open-source community [3] and ported to other environments such as desktop grids [9], volunteer computing [10], dynamic cloud [11], and mobile systems [12]. Besides, some MapReduce-like systems [34], [4], [5], [35] and high-level facilities [6], [36], [37] were proposed. In addition, MapReduce has expanded its application from batch processing to iterative computation [38], [39] and stream processing [40], [41]. Our solution can do help to these systems when hard disks are used and many cores are involved.

5.2 MapReduce Optimization on Multi-Core Servers

This can be seen in [7], [42], [8], [43], [44]. All these frameworks are designed for a single server, of which [8], [43], [44] mainly focused on graphics processors and [7], [42] were implemented on symmetric-multiple-processor server. Obviously, a single node with the frameworks could only process gigabytes of data at most and cannot afford the task of handling terabytes or petabytes of data. Besides, they still suffer from the IO bottleneck as could also be seen from Fig. 2 of [7] when the number of cores is greater than 8. Our solution is a distributed caching scheme covering each node of the MapReduce cluster. Therefore, it cannot only accelerate data processing on a single server but also on clusters.

5.3 In-Memory MapReduce

In-memory MapReduce borrows the basic idea of in-memory computing—data put in memory can be processed faster because memory is accessed much more quickly—and places job-related data in random access memory (RAM) to boost job execution. Typical systems include Spark [39], HaLoop [45], M3R [38], Twister [46], and Mammoth [47]. Spark, HaLoop, M3R, and Twister are specially designed for iterative computation and they reduce the IO cost (and thus boost computation) by placing in RAM the data to be processed multiple rounds. Such a way costs more because more memory is needed to hold the data and memory is more expensive than SSD. Mammoth is a comprehensive solution trying to solve inefficiencies in both memory usage and IO operations. To achieve the purpose, it devises various mechanisms to utilize memory more smartly, including rule-based prioritized memory allocation and revocation, global memory scheduling algorithm, memory-based shuffling, and so on. Mammoth can benefit from mpCache especially in a memory-constrained environment where only limited memory can be used for data caching. With mpCache introduced, more memory can be released to support computation and thus the task parallelism degree is improved, which means faster job execution.

5.4 IO Optimization via SSD-Based Cache

With the emergence of NAND (Negative-AND) Flash memory, much research work has been reported that utilized SSD to improve storage performance. Yongseok et al. [48] proposed a way to balance cache size and update cost of flash memory so that better performance can be obtained in the HDD-SSD hybrid storage system. Hystor [25], Proximal IO [26], SieveStore [27], and HybridStore [28] also used SSD as a cache of hard disks as we do. But these methods only focus on a single node, with an aim to boost small files (typical size is below 200 KB) manipulation by caching. mpCache can work across many nodes in a coordinated way. In addition, it devises a relatively complex and efficient cache replacement scheme to better support MapReduce applications.

5.5 MapReduce Optimization via In-Memory Cache

PACMan [1] cached input data in memory to reduce the high IO cost of hard disks so as to improve performance. Since the task parallelism degree of new generation of MapReduce (e.g., YARN) is more concerned with free memory. Caching data in memory, as shown in Section 4.3.1, would cut down the task parallelism and lead to low performance for some memory-intensive jobs (e.g., shuffle-heavy jobs in our benchmarks), for the memory left for normal task operations reduces. It is on account of only limited memory available and the large volume of Localized Data that PACMan only has Input Data cached. As a result, it just improves the Map phase. For those shuffle-heavy MapReduce jobs (e.g., *k-means* and *tera-sort*), they cannot benefit from in-memory caching in the Reduce phase. Unfortunately, the number of shuffle-heavy jobs is large in the real world. Our SSD-based caching solution can solve the problem and accelerate both phases.

6 Conclusion

In this paper we presented mpCache, a solution that utilizes SSD to cache MapReduce Input Data and Localized Data so that all the costly IO operations—*Read*, *Spill*, and *Merge*—are boosted and the whole job is accelerated as a result. Caching in such a way is cost-effective and can solve the performance degradation problem caused by in-memory caching as mentioned in Section 1. Given the fact that data will continue growing exponentially, this is especially important. We have implemented mpCache in Hadoop and evaluated it on a 7-node commodity cluster. The experimental results show that mpCache can get an average speedup of 2.09 times over Hadoop, and 1.79 times over PACMan, the latest work about MapReduce optimization by in-memory data caching.

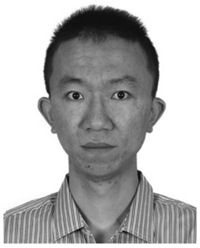
Acknowledgments

This work is co-sponsored by Natural Science Foundation of China (61572280, 61433008, 61373145, and U1435216), National High-Tech R&D (863) Program of China (2014AA01A302), National Key Research and Development Program (2016YFB1000500), and National Basic Research (973) Program of China (2014CB340402).

References

[1] G. Ananthanarayanan, et al., "Pacman: Coordinated memory caching for parallel jobs," in *Proc. 9th USENIX Conf. Netw. Syst. Des. Implementation*, 2012, pp. 20–20.
[2] J. Dean and S. Ghemawat, "MapReduce: Simplified data processing on large clusters," *Commun. ACM*, vol. 51, no. 1, pp. 107–113, 2008.
[3] A. Hadoop, "Hadoop," 2014. [Online]. Available: http://hadoop.apache.org
[4] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly, "Dryad: Distributed data-parallel programs from sequential building blocks," *ACM SIGOPS Operating Syst. Rev.*, vol. 41, no. 3, pp. 59–72, 2007.
[5] Y. Yu, et al., "Dryadlinq: A system for general-purpose distributed data-parallel computing using a high-level language," in *Proc. 8th USENIX Conf. Operating Syst. Des. Implementation*, 2008, pp. 1–14.
[6] A. Thusoo, et al., "Hive: A warehousing solution over a MapReduce framework," *Proc. VLDB Endowment*, vol. 2, no. 2, pp. 1626–1629, 2009.
[7] C. Ranger, R. Raghuraman, A. Penmetsa, G. Bradski, and C. Kozyrakis, "Evaluating MapReduce for multi-core and multiprocessor systems," in *Proc. IEEE 13th Int. Symp. High Perform. Comput. Archit.*, 2007, pp. 13–24.
[8] B. He, W. Fang, Q. Luo, N. K. Govindaraju, and T. Wang, "Mars: a MapReduce framework on graphics processors," in *Proc. 17th Int. Conf. Parallel Architectures Compilation Techniques*, 2008, pp. 260–269.
[9] B. Tang, M. Moca, S. Chevalier, H. He, and G. Fedak, "Towards MapReduce for desktop grid computing," in *Proc. Int. Conf. P2P Parallel Grid Cloud Internet Comput.*, 2010, pp. 193–200.
[10] H. Lin, X. Ma, J. Archuleta, W.-C. Feng, M. Gardner, and Z. Zhang, "Moon: MapReduce on opportunistic environments," in *Proc. 19th ACM Int. Symp. High Perform. Distrib. Comput.*, 2010, pp. 95–106.
[11] F. Marozzo, D. Talia, and P. Trunfio, "P2p-MapReduce: Parallel data processing in dynamic cloud environments," *J. Comput. Syst. Sci.*, vol. 78, no. 5, pp. 1382–1402, 2012.
[12] A. Dou, V. Kalogeraki, D. Gunopulos, T. Mielikainen, and V. H. Tuulos, "Misco: A MapReduce framework for mobile systems," in *Proc. 3rd Int. Conf. Pervasive Technol. Related Assistive Environments*, 2010, Art. no. 32.
[13] L. Seiler, et al., "Larrabee: A many-core x86 architecture for visual computing," *ACM Trans. Graphics*, vol. 27, 2008, Art. no. 18.
[14] A. K. Mishra, J. L. Hellerstein, W. Cirne, and C. R. Das, "Towards characterizing cloud backend workloads: Insights from google compute clusters," *ACM SIGMETRICS Performance Evaluation Rev.*, vol. 37, no. 4, pp. 34–41, 2010.
[15] M. J. Feeley, W. E. Morgan, E. Pighin, A. R. Karlin, H. M. Levy, and C. A. Thekkath, "Implementing global memory management in a workstation cluster," in *Proc. 15th ACM Symp. Operating Syst. Principles*, 1995, pp. 201–212.
[16] M. J. Franklin, M. J. Carey, and M. Livny, "Global memory management in client-server DBMS architectures," in *Proc. 18th Int. Conf. Very Large Data Bases*, 1992, pp. 596–609.
[17] H. Garcia-Molina and K. Salem, "Main memory database systems: An overview," *IEEE Trans. Knowl. Data Eng.*, vol. 4, no. 6, pp. 509–516, Dec. 1992.
[18] J. Ousterhout, et al., "The case for RAMClouds: Scalable high-performance storage entirely in DRAM," *ACM SIGOPS Operating Syst. Rev.*, vol. 43, no. 4, pp. 92–105, 2010.
[19] A. C. Murthy, et al., "Architecture of next generation Apache Hadoop MapReduce framework," 2011. [Online]. Available: https://issues.apache.org/jira/secure/attachment/12486023/MapReduce_NextGen_Architecture.pdf
[20] Y. Lu, J. Shu, and W. Wang, "ReconFS: A reconstructable file system on flash storage," in *Proc. 12th USENIX Conf. File Storage Technol.*, 2014, pp. 75–88.
[21] M. Zheng, J. Tucek, F. Qin, and M. Lillibridge, "Understanding the robustness of SSDs under power fault," in *Proc. 11th USENIX Conf. File Storage Technol.*, 2013, pp. 271–284.
[22] S.-W. Lee, B. Moon, C. Park, J.-M. Kim, and S.-W. Kim, "A case for flash memory SSD in enterprise database applications," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 2008, pp. 1075–1086.
[23] B. C. Lee, E. Ipek, O. Mutlu, and D. Burger, "Architecting phase change memory as a scalable DRAM alternative," *ACM SIGARCH Comput. Archit. News*, vol. 37, no. 3, pp. 2–13, 2009.
[24] J. Handy, "Flash memory versus hard disk drives - which will win?" 2014. [Online]. Available: http://www.storagesearch.com/semico-art1.html

[25] F. Chen, D. A. Koufaty, and X. Zhang, "Hystor: Making the best use of solid state drives in high performance storage systems," in *Proc. Int. Conf. Supercomputing*, 2011, pp. 22–32.
[26] J. Schindler, S. Shete, and K. A. Smith, "Improving throughput for small disk requests with proximal I/O," in *Proc. 9th USENIX Conf. File Storage Technol.*, 2011, pp. 133–147.
[27] T. Pritchett and M. Thottethodi, "Sievestore: A highly-selective, ensemble-level disk cache for cost-performance," in *Proc. 37th Annu. Int. Symp. Comput. Archit.*, 2010, pp. 163–174.
[28] Y. Kim, A. Gupta, B. Urgaonkar, P. Berman, and A. Sivasubramaniam, "Hybridstore: A cost-efficient, high-performance storage system combining SSDs and HDDs," in *Proc. IEEE 19th Int. Symp. Modeling Anal. Simulation Comput. Telecommun. Syst.*, 2011, pp. 227–236.
[29] B. Wang, J. Jiang, and G. Yang, "mpCache: Accelerating MapReduce with hybrid storage system on many-core clusters," in *Network and Parallel Computing*. Berlin, Germany: Springer, 2014, pp. 220–233.
[30] S. Ghemawat, H. Gobioff, and S.-T. Leung, "The Google file system," *ACM SIGOPS Operating Syst. Rev.*, vol. 37, pp. 29–43, 2003.
[31] K. Shvachko, H. Kuang, S. Radia, and R. Chansler, "The Hadoop distributed file system," in *Proc. IEEE 26th Symp. Mass Storage Syst. Technol.*, 2010, pp. 1–10.
[32] F. Ahmad, S. Lee, M. Thottethodi, and T. Vijaykumar, "Puma: Purdue MapReduce benchmarks suite," 2012. [Online]. Available: http://web.ics.purdue.edu/~fahmad/benchmarks.htm
[33] D. E. Knuth, *The Art of Computer Programming*, vol. 3. Reading, MA, USA: Addison-Wesley, 2005.
[34] H.-c. Yang, A. Dasdan, R.-L. Hsiao, and D. S. Parker, "MapReduce-merge: Simplified relational data processing on large clusters," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 2007, pp. 1029–1040.
[35] P. Costa, A. Donnelly, A. Rowstron, and G. O'shea, "Camdoop: Exploiting in-network aggregation for big data applications," in *Proc. 9th USENIX Conf. Netw. Syst. Des. Implementation*, 2012, p. 3.
[36] C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins, "Pig latin: A not-so-foreign language for data processing," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 2008, pp. 1099–1110.
[37] A. F. Gates, et al., "Building a high-level dataflow system on top of MapReduce: The pig experience," *Proc. VLDB Endowment*, vol. 2, no. 2, pp. 1414–1425, 2009.
[38] A. Shinnar, D. Cunningham, V. Saraswat, and B. Herta, "M3r: Increased performance for in-memory Hadoop jobs," *Proc. VLDB Endowment*, vol. 5, no. 12, pp. 1736–1747, 2012.
[39] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, "Spark: Cluster computing with working sets," in *Proc. 2nd USENIX Conf. Hot Topics Cloud Comput.*, 2010, pp. 10–10.
[40] nathanmarz, "Storm," 2014. [Online]. Available: https://github.com/nathanmarz/storm
[41] L. Neumeyer, B. Robbins, A. Nair and A. Kesari, "S4: Distributed stream computing platform," in *Proc. IEEE Int. Conf. Data Mining Workshops*, 2010, pp. 170–177.
[42] J. Talbot, R. M. Yoo, and C. Kozyrakis, "Phoenix++: Modular MapReduce for shared-memory systems," in *Proc. 2nd Int. Workshop MapReduce Appl.*, 2011, pp. 9–16.
[43] W. Fang, B. He, Q. Luo, and N. K. Govindaraju, "Mars: Accelerating MapReduce with graphics processors," *IEEE Trans. Parallel Distrib. Syst.*, vol. 22, no. 4, pp. 608–620, Apr. 2011.
[44] J. A. Stuart and J. D. Owens, "Multi-GPU MapReduce on GPU clusters," in *Proc. IEEE Int. Parallel Distrib. Process. Symp.*, 2011, pp. 1068–1079.
[45] Y. Bu, B. Howe, M. Balazinska, and M. D. Ernst, "HaLoop: Efficient iterative data processing on large clusters," *Proc. VLDB Endowment*, vol. 3, no. 1–2, pp. 285–296, 2010.
[46] J. Ekanayake, et al., "Twister: A runtime for iterative MapReduce," in *Proc. 19th ACM Int. Symp. High Performance Distrib. Comput.*, 2010, pp. 810–818.
[47] X. Shi, et al., "Mammoth: Gearing Hadoop towards memory-intensive MapReduce applications," *IEEE Trans. Parallel Distrib. Syst.*, vol. 26, no. 8, pp. 2300–2315, Jul. 2015.
[48] Y. Oh, J. Choi, D. Lee, and S. H. Noh, "Caching less for better performance: Balancing cache size and update cost of flash memory cache in hybrid storage systems," in *Proc. 10th USENIX Conf. File Storage Technol.*, 2012, pp. 25–25.



Bo Wang received the BS degree in computer science and technology from Tsinghua University, China, in 2008 and the MS degree in computer applications from North China Institute of Computing Technology, in 2011. He is currently working toward the PhD degree in the Department of Computer Science and Technology, Tsinghua University, China, working on Hadoop optimization. His research interests include distributed systems, big data computing, storage and file systems, and virtualization. He is a student member of the IEEE.

Jinlei Jiang received the PhD degree in computer science and technology from Tsinghua University, China, in 2004 with an honor of excellent dissertation. He is currently an associate professor in the Department of Computer Science and Technology, Tsinghua University, China. His research interests include distributed computing and systems, cloud computing, big data, and virtualization. He is currently on the editorial boards of *KSII Transactions on Internet and Information Systems*, *International Journal on Advances in Intelligent Systems*, and *EAI Endorsed Transactions on Industrial Networks and Intelligent Systems*. He is a winner of Humboldt Research Fellowship. He is a member of the IEEE.

Yongwei Wu received the PhD degree in applied mathematics from Chinese Academy of Sciences, in 2002. He is currently a professor of computer science and technology with Tsinghua University, China. His research interests include parallel and distributed processing, mobile and distributed systems, cloud computing, and storage. He has published more than 80 research publications and received two Best Paper Awards. He is currently on the editorial boards of the *IEEE Transactions on Cloud Computing*, the *Journal of Grid Computing*, *IEEE Cloud Computing*, and the *International Journal of Networked and Distributed Computing*. He is a member of the IEEE.

Guangwen Yang received the MS degree in applied mathematics from Harbin Institute of Technology, China, in 1987, and the PhD degree in computer architecture from Harbin Institute of Technology, China, in 1996. He is a professor in the Department of Computer Science and Technology and the director of the Institute of High Performance Computing, Ministry of Education Key Laboratory for Earth System Modeling, Tsinghua University, China. His research interests include parallel and distributed algorithms, cloud computing, and the earth system model. He is a member of the IEEE.

Keqin Li is a SUNY distinguished professor of computer science. His current research interests include parallel computing and high-performance computing, distributed computing, energy-efficient computing and communication, heterogeneous computing systems, cloud computing, big data computing, CPU-GPU hybrid and cooperative computing, multicore computing, storage and file systems, wireless communication networks, sensor networks, peer-to-peer file sharing systems, mobile computing, service computing, Internet of things, and cyber-physical systems. He has published more than 440 journal articles, book chapters, and refereed conference papers, and received several best paper awards. He is currently or has served on the editorial boards of the *IEEE Transactions on Parallel and Distributed Systems*, the *IEEE Transactions on Computers*, the *IEEE Transactions on Cloud Computing*, the *IEEE Transactions on Services Computing*, the *Journal of Parallel and Distributed Computing*. He is a fellow of the IEEE.