mpCache: Accelerating MapReduce with Hybrid Storage System on Many-Core Clusters

Bo Wang, Jinlei Jiang, Guangwen Yang

mpCache: Accelerating MapReduce with Hybrid Storage System on Many-Core Clusters

Bo Wang[1] and Jinlei Jiang[1 2] and Guangwen Yang[1]

[1] Department of Computer Science and Technology
Tsinghua National Laboratory for Information Science and Technology (TNLIST)
Tsinghua University, Beijing 100084, China
Email: bo-wang11@mails.tsinghua.edu.cn {jjlei,ygw}@tsinghua.edu.cn
[2] Technology Innovation Center at Yinzhou
Yangtze Delta Region Institute of Tsinghua University
Zhejiang 314006, China

Abstract. As a widely used programming model and implementation for processing large data sets, MapReduce does not scale well on many-core clusters, which, unfortunately, are common in current data centers. To deal with the problem, this paper: 1) analyzes the causes of poor scalability of MapReduce on many-core clusters and identifies the key one as the underlying low-speed storage (hard disk) can not meet the requirements of frequent IO operations, and 2) proposes mpCache, a SSD based hybrid storage system that caches both Input Data and Localized Data, and dynamically tunes the cache space allocation between them to make full use of the space. mpCache has been incorporated into Hadoop and evaluated on a 7-node cluster by 13 benchmarks. The experimental results show that mpCache gains an average speedup of 2.09 when compared with the original Hadoop, and achieves an average speedup of 1.79 when compared with PACMan, the latest in-memory optimization of MapReduce.

1 Introduction

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% of a job's duration and consumes 69% of the resources [2].

MapReduce [5] is a programming model and an associated implementation for large data sets processing on clusters with hundreds or thousands of nodes. Due to its scalability and ease of programming, MapReduce has been adopted by many companies, including Google [5], Yahoo, Microsoft [9], and Facebook [20].

Although MapReduce scales well with the increase of server number, its performance, however, improves less or even remains unchanged with the increase

of CPU-cores per server. Figure 1 shows the execution time of *self-join* with varied CPU-cores per server on a 7-node cluster, in which 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 plus-line and square-line gets wider and wider. The fundamental reason behind this is that the underlying low-speed storage (hard disk) can not meet the requirements of MapReduce frequent IO operations: in the Map phase, the model reads in raw input data to generate set of intermediate key-value pairs, which are then written back; Shuffle phase, the model reads the intermediate data out from the disk once again and sends to corresponding nodes which Reduce tasks are scheduled on. In addition, during the whole execution of jobs, temporary data is also written to local storage when memory buffer is full. Although more tasks are concurrently running as more CPU-cores equipped, the IO speed of the storage system which backs MapReduce remains unchanged and can not meet the IO demand of high-concurrency tasks, resulting in the unchanged performance of MapReduce. Unfortunately, it is common that servers in data centers are often equipped with a large quantity of CPU-cores (referred to as **many-core**).

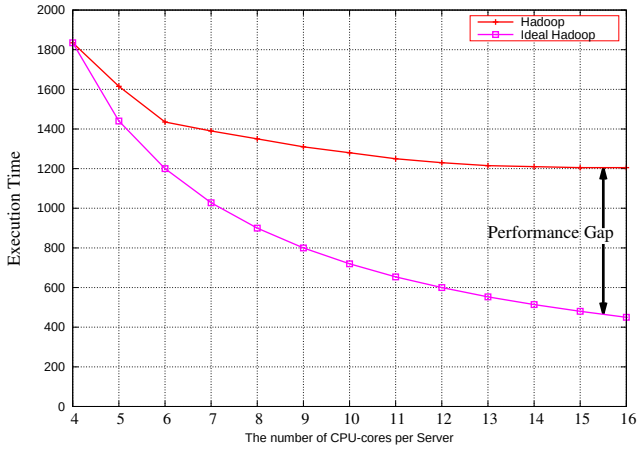


Fig. 1. Execution time of *self-join* running with varied number of CPU-cores per server using settings in Section 3 with 60GB Input Data.

To overcome the bottleneck of low speed storage, caching data in memory is an effective way to improve IO-intensive applications. Indeed many studies have been done on in-memory cache [6] [13]. With the volume of memories scales with hardware technology, it seems more feasible to cache data in memory to provide high IO speed. However, caching data in memory inevitably occupies additional memories and drops down the task parallelism degree (that is the number of concurrent running tasks). What's more, some machine-learning algorithms

such as *k-means* and *term-vector* are memory-intensive that consume very large volume of memories. For these applications, task parallelism degree drops significantly due to insufficient memories, leaving some CPU-cores idle. Although adding more memories could alleviate the situation, the volume of data scales even faster. Taking cost into consideration, it is not cost-effective to provide high IO speed by in-memory caching.

Flash memory based Solid State Drive (SSD), emerges as an ideal storage medium for building high performance storage systems. However the cost of building a storage system completely with SSDs is often above the acceptable threshold in most commercial data centers. Even considering the price-drop trend, the average cost per GB of SSDs is still unlikely to reach the level of hard disks in the near future [8]. Thus, we believe that in most systems, SSDs should not be simply viewed as a replacement for the existing HDD-based storage, but instead SSDs should be a means to enhance it.

Taking all the concerns discussed above into consideration, we proposed mpCache, SSD based hybrid storage system, to support MapReduce scalable on many-core clusters, which not only provides high IO for IO-intensive applications but also maintains task parallelism degree of memory-intensive jobs. The contributions of our paper are as follows.

- We propose a new approach, called mpCache, to cache both Input Data and Localized Data to speedup the IO-intensive phases. We also devise an algorithm to dynamically tune the allocations between Input Cache and Localized Cache to make full use of the cache and provide better performance.
- We propose an algorithm to replace Input Cache efficiently, taking replacement cost, data set size, access frequency, all-or-nothing into consideration for better performance.
- Extensive experiments are conducted to evaluate mpCache. The experiment results shows that mpCache gets an average speedup of 2.09 when compared with original Hadoop and achieves an average speedup of 1.79 when compared with PACMan.

The rest of this paper is organized as follows. Section 2 describes the key ideas and algorithms of mpCache. Section 3 shows the experimental results. Section 4 reviews the related work and the paper ends in Section 5 with some conclusions.

2 mpCache Design

As shown in Figure 2, mpCache adopts a master-slave architecture 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 [2] found that a job is sped up only when inputs of all tasks are cached. mpCache Slave tunes the cache space allocation between Input Cache and Localized Cache, and serves cached blocks.

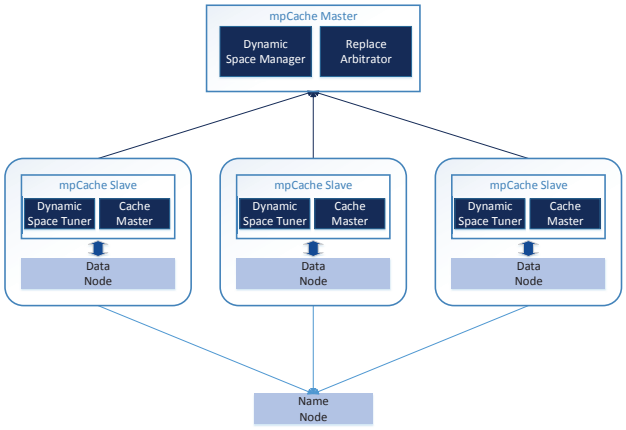


Fig. 2. mpCache architecture.

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

mpCache Slave seats on each data node and consists of two components–*Dynamic Space Tuner* and *Cache Master*. *Dynamic Space Tuner* is corresponding to tuning the space allocation between Input Cache and Localized Cache. *Cache Master*'s role is to serve cached blocks and cache new blocks. *Cache Master* on each data node intercepts the data reading requests from Map task, checking whether the requested data block cached. If does, *Cache Master* servers the data request from cache and request to *Replace Arbitrator* of mpCache Master for the block's hit. If the requested block is not cached and cache space does not have sufficient space to hold it, *Cache Master* will send replace request to *Replace Arbitrator* and evicts cached blocks to make room for new cache according to the return information from *Replace Arbitrator*.

2.1 Optimal Allocation Determination

Dynamic Space Tuner divides the whole cache space into three parts, i.e., Input Cache, Dynamic pool, and Localized Cache. Since the distributed file systems (e.g., GFS [7] and HDFS [17]), which back MapReduce applications up, store data as blocks, we divide Dynamic pool into blocks. When caching input data, free Dynamic pool blocks are allocated to Input Cache when Input Cache is full. As the execution of job going, *Dynamic Space Tuner* constantly monitors the used Localized Cache size. When Localized Data size exceeds Localized Cache size, *Dynamic Space Tuner* checks if there are free blocks in Dynamic pool, if not

Dynamic Space Tuner excludes cached input data from Dynamic pool using the same scheme described in Section 2.2, then allocates blocks of Dynamic pool to Localized Cache one by one. All the Dynamic pool blocks allocated to Localized Cache are withdrawn back when Localized Cache used ratio is below the *guard value* (in our implementation, *guard value* is set to 0.5).

2.2 Input Data Cache Model

Admission Control Policy We use an admission control policy to decide whether or not it is worthwhile caching an object in the first place. We use an auxiliary cache which maintains the *identities* of input data sets from different jobs. For each object in this auxiliary cache we also maintain time-stamps of the last access, measured both in terms of the number of data set accesses and time.

Using the admission control policy, we would like to ensure that at the i^{th} iteration the potential incoming input data jd_i is popular enough to offset the loss of the input data it displaces. So we process as follows: If there is enough free space for jd_i, we simply put jd_i into the main cache. Otherwise, we check if jd_i occurs in the auxiliary cache. If it does not, jd_i is not put into the main cache. However, we put jd_i into the auxiliary cache in accordance with LRU rules. On the other hand, if jd_i does occur in the auxiliary cache, then we determine if the decision which the replacement policy heuristic makes would be profitable. That is we compare the value $1/Size(jd_i)\Delta_{jd_i i}$ ($\Delta_{jd_i i}$ is at the i^{th} iteration the number of accesses since the last time jd_i was accessed) with the sum $\sum_j 1/(Size(jd_j)\Delta_{jd_j i})$ of the set of candidate outgoing data blocks. We admit jd_i only if it is profitable to do so.

Main Cache Replacement Scheme We now describe the data replacement scheme of the main cache. With the data set in the main cache we associate a *frequency* $Fr(jd)$ counting how many times jd was accessed since the last time it entered the main cache. We also maintain a priority queue for the data sets in the main cache. When a data set of a job is inserted into the queue, it is given priority $Pr(jd)$ computed in the following way:

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

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

where *Blocks_Access(jd)* is the number of accesses of all blocks of data set jd; $Fr(jd)$ is the frequency count of data set jd; *Full* is a *bonus* value for the data set which have all the blocks cached in the main cache (due to the *all-or-nothing* characteristic of MapReduce cache [2]); *Clock* is a running queue "clock" that starts at 0 and is updated, for each evicted data set $jd_{evicted}$, to its priority in the queue, $Pr(jd_{evicted})$; and $Size(jd)$ is the number of blocks of data set jd. When mpCache Master receives update message from mpCache Slave, we use Algorithm 1 described below to update $Pr(jd)$ of the data set to which the update message corresponding. To_Del is a list of tuples such as $< data\_node, blocks_{evicted} >$.

Algorithm 1 Main Cache Replacement Scheme.

if the request for the block bk a hit update **then**
  get the data set jd, to which bk belongs.
  $Clock$ do not change.
  $Blocks\_Access(jd)$ increased by one.
  $Pr(jd)$ is update using Equation 1∼2 and jd is moved according in the queue.
else
  if the request does not need replace **then**
    bk is cached.
  else
    identify $mpSlave$ where the request comes from.
    identify $data\_node$ where $mpSlave$ seated on.
    if To_Del list contains $data\_node$ **then**
      return $blocks_{evicted}$ to $mpSlave$, $mpSlave$ evicts $blocks_{evicted}$ and cache bk.
    else
      identify the data set $jd_{evicted}$ to evict, which has the lowest priority
      $Clock$ is set to $Pr(jd_{evicted})$.
      set $blocks_{evicted}$ to all the blocks of $jd_{evicted}$.
      return $blocks_{evicted}$ to $mpSlave$, which evicts $blocks_{evicted}$ and cache bk.
      identify all the data nodes $allnodes$ which store $blocks_{evicted}$.
      for $dn \in allnodes$ **do**
        add $< dn, blocks_{evicted} >$ to To_Del.
      end for
    end if
  end if
  $Blocks\_Access(jd)$ increased by one.
  if all the blocks of jd are cached **then**
    $Full = BONUS\_VALUE$.
  else
    $Full = 0$.
  end if
  $Pr(jd)$ is computed using Equation 2 and jd is enqueued accordingly.
end if

3 Evaluation

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

3.1 Platform

The cluster used for experiments consists of 7 nodes. Each node has two eight-core Xeon E5-2640 v2 CPUs running at 2.0GHz, 20MB Intel Smart Cache, 32GB DDR3 RAM, one 2TB SATA hard disk and two 160GB SATA Intel SSDs configured as RAID0. All the nodes run Ubuntu 12.04, have a Gigabit Ethernet card and connect to a Gigabit Ethernet switch. Since our SSD cache space is *160*2=320GB* on each node, which is large enough to hold most of the input

data set, and in real data centers, the input data of jobs is TB or even PB magnitudes, we only use 80GB cache in our experiment.

3.2 Benchmarks

We use 13 benchmarks released on PUMA [1], covering shuffle-light, shuffle-medium, and shuffle-heavy categories. We vary the input data size of each benchmark to 20 classes. As the input data size has Zipf-like frequency distributions [11], we set a chosen probability to each data size using Equation 3.

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

Table 1 summarizes the characteristics of the benchmarks in terms of input data size (data of the right three column is when *k=10*), data source, the number of Map/Reduce tasks, shuffle size, and execution time on Hadoop.

Shuffle-light cases have very little data transfer in shuffle phase, including *grep*, *histogram-ratings*, *histogram-movies*, and *classification*. Shuffle-heavy cases, the shuffle data size of which is very large (as shown in Table 1, almost the same volume as the input data size), include *k-means*, *self-join*, *adjacency-list*, *ranked-inverted-count*, and *tera-sort*. The shuffle data size of shuffle-medium cases is between shuffle-light and shuffle-heavy, including *word-count*, *inverted-index*, *term-vector*, and *sequence-count*.

Table 1. Input data size of benchmarks. (k=1,2,...,20) and characteristics

Benchmark	Input size(GB)	Data source	#Maps & #Reduces	Shuffle size(GB)	Map&Reduce time on Hadoop(s)
grep	k*4.3	wikipedia	688 & 40	$6.9*10^{-6}$	222&2
histogram-ratings	k*3	netflix data	480 & 40	$6.3*10^{-5}$	241&5
histogram-movies	k*3	netflix data	480 & 40	$6.8*10^{-5}$	261&5
classification	k*3	netflix data	480 & 40	$7.9*10^{-3}$	286&5
word-count	k*4.3	wikipedia	688 & 40	0.318	743&22
inverted-index	k*4.3	wikipedia	688 & 40	0.363	901&6
term-vector	k*4.3	wikipedia	688 & 40	0.384	1114&81
sequence-count	k*4.3	wikipedia	688 & 40	0.737	1135&27
k-means	k*3	netflix data	480 & 4	26.28	450&2660
self-join	k*3	puma-I	480 & 40	26.89	286&220
adjacency-list	k*3	puma-II	480 & 40	29.38	1168&1321
ranked-inverted-count	k*4.2	puma-III	672 & 40	42.45	391&857
tera-sort	k*3	puma-IV	480 & 40	31.96	307&481

When submitting job to the cluster, we randomly select a job from the 13 benchmarks, and we choose input data size according to the attached probability.

3.3 Experimental Results

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 on mpCache, Hadoop, and PACMan separately and get the average value.

PACMan uses memory to cache input data, the bigger the cache size is, the more data is cached in memory, causing the faster Map phase. However, in YARN, the concurrent running tasks number is relative to the available CPU-cores and free memory. Using too much memory for cache will decrease the parallelism degree of the tasks. We set the memory cache size to 12GB as recommended in PACMan [2].

Figure 3 shows the normalized execution time of Map/Reduce phase. For shuffle-light jobs *grep*, *histogram-movies*, *histogram-ratings*, and *classification*, the execution time is short (about 241s, 253s, 279s, and 304s of Hadoop when *k=10*), most of the time is spending on data IO, caching the input data of Map accelerates the execution of Map phase significantly (gets a speedup of 2.42 times of Map phase averagely). The Reduce phase time of mpCache is almost the same as that of Hadoop for three reasons: i) The Reduce phase of shuffle-light jobs is vert 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 very small (less than 1 MB), thus, caching localized data has little acceleration. The job execution time of shuffle-light jobs on mpCache gets a speedup of 2.23 times, averagely. When running on PACMan, each task runs well with 1GB memory, thus PACMan and mpCache gets the same parallelism degree of the tasks. Although PACMan's memory cache provides a fast IO than SSD cache of mpCache, mpCache size is much bigger than PACMan's memory cache size, mpCache's auxiliary cache scheme also prevents too frequent replacement, causing **a higher hit ratio** than PACMan does. Therefore, PACMan gets an average speedup of 2.17 times, which is slightly lower than mpCache.

For shuffle-medium jobs *word-count*, *inverted-index*, *term-vector*, and *sequence-count*, the execution time is longer than shuffle-light jobs(about 779s, 932s, 1209s, and 1174s), the acceleration of caching Map input data is also smaller (gets a speed of 1.25 times of Map phase averagely). The shuffle data size of these jobs is about 318∼737MB, and the localized data size is 1∼3GB, thus, caching localized data has bigger acceleration of Reduce phase than that of shuffle-light jobs, getting a average speed up of 1.60 times of Reduce phase. The job execution time of shuffle-medium jobs on mpCache gets a speed up of 1.25 times, averagely. When running on PACMan, *word-count* and *inverted-index* run well with 1GB memory, thus the speedup is roughly the same as mpCache. *term-vector* task needs at least 3GB memory, thus the parallelism degree is 10 on Hadoop and mpCache, while 6 on PACMan, causing the performance of PACMan drops to 0.762 of Hadoop. *sequence-count* needs at least 2GB memory, thus the parallelism degree is 16 on Hadoop and 10 on PACMan, causing the performance of PACMan drops to 0.868 of Hadoop.

For shuffle-heavy jobs *k-means*, *self-join*, *adjacency-list*, *ranked-inverted-index*, and *tera-sort*, the shuffle data size and localized data size is very big, thus caching Map input data and localized data both reduce the Map&Reduce phase time significantly. The Map time of *k-means*, *self-join*, *ranked-inverted-index*, and *tera-sort* is shorter than that of *adjacency-list*, thus the front three jobs get a speedup of 1.82∼2.69, while, the *adjacency-list* Map time is longer (1168s), thus, getting

a speedup of only 1.04 times. Since the localized data size of shuffle-heavy jobs is the biggest of the three types, caching localized data accelerates the Reduce phase most, getting a speedup of 3.87 times of Reduce phase. The job execution time of shuffle-heavy jobs on mpCache gets a speed up of 2.65 times, averagely. When running on PACMan, *self-join*, *adjacency-list*, *ranked-inverted-index*, and *tera-sort* need 2 GB memory for each task, thus the parallelism degree is 10 on PACMan, and get an average performance of 0.981 of Hadoop. *k-means* benchmark clusters input data into 4 clusters, thus Reduce tasks number is set to 4. The Reduce phase of *k-means* is a heavy part (2660s of 3087s), and needs at least 8GB memory for each task. Therefore, the Map phase time is 2.46 times of Hadoop, and Reduce time is the same as Hadoop, causing a performance of 0.808 of Hadoop.

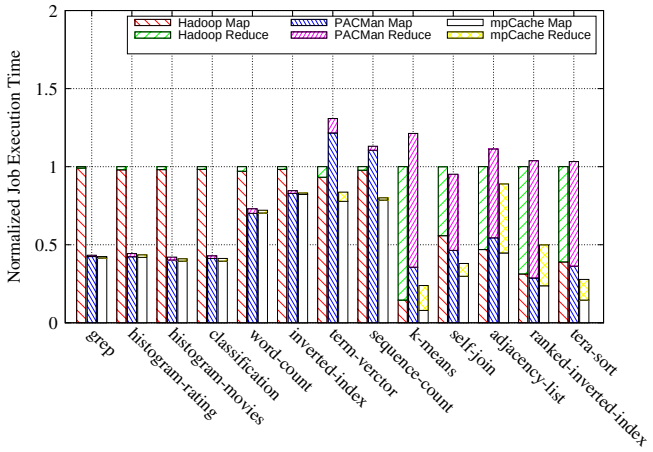


Fig. 3. Job execution time comparison with Hadoop and PACMan.

PACMan used 12GB memory for data cache and got considerable performance using MapReduce v1 of Hadoop, the task parallelism degree of which was configured by "slots" number. And slots number was set as constant value in configuration files, both Hadoop and PACMan used the same configuration, thus the same task parallelism degree. However, in MapReduce v2–YARN, the concurrent running task number is determined by free CPU-cores and free memory, allocating memory for data cache inevitably reduce the task parallelism degree of some jobs. In our cluster, each node contains 16 CPU-cores and 32GB memory, PACMan used 12GB for memory cache, thus the memory left for computing is 20GB. When running "1GB jobs" (which consume 1GB memory for each task, such as *grep*, *histogram-rating*, *histogram-movies*, *classification*, *word-count*, and *inverted-index*) on PACMan, the task parallelism degree is 16, which is the same as that of Hadoop and mpCache. Therefore, PACMan gets a better performance than Hadoop and almost the same as mpCache. For other jobs, each task needs

at least 2GB memory (3GB for *term-vector*, and 6GB for *k-means*), which results in the task parallelism degree of PACMan drop to 10 (6 of *term-vector*, and 3 of *k-means*). Although PACMan's memory cache could significantly speedup Map phase IO, the drop of task parallelism degree slows down the job worse, thus as illustrated in Figure 3, PACMan even performs worse than Hadoop of these "at least 2 GB" jobs.

For all these benchmarks, mpCache gains an average speedup of 2.09 when compared with the original Hadoop, and achieves an average speedup of 1.79 when compared with PACMan.

In order to trade off the influence of memory cache of PACMan, we also do an experiment using only 8 CPU-cores of each node for Hadoop, PACMan, and mpCache. As shown in Figure 4, when using only 8 CPU-cores, most of the benchmarks run with the same task parallelism degree on Hadoop, mpCache, and PACMan(except *term-vector* and *k-means*). For shuffle-light jobs, mpCache and PACMan run with the same task parallelism degree, speedups over Hadoop are 1.74 and 1.67, separately. For shuffle-medium jobs, *word-count* and *inverted-index* is 1GB-task job, getting speedups over Hadoop of 1.12 and 1.08. *term-vector* is 3GB-task job, when running on Hadoop and mpCache, task parallelism degree is 8, while, running on PACMan is 6, causing a high Map phase time than Hadoop. Thus, the whole performance of PACMan is still worse than Hadoop. For shuffle-heavy jobs, the localized data size is also very big, mpCache caches both input data and localized data, resulting in an average speedup of 1.63 times of Map phase, while PACMan gets an average speedup of 1.35 times of Map phase. mpCache's caching localized data also gets an average speedup of 2.09 times of Reduce phase, while PACMan does not affect the Reduce phase. For all the benchmarks, mpCache gets an average speedup of 1.62 times, while, PACMan gets an average speedup of 1.25 times.

Sensitivity to Cache Size We now evaluate mpCache's sensitivity to cache size by varying the available cache size of each mpCache Slave between 5GB and 160GB. The experimental results are shown in 3 sub-figures, i.e., Figure 5(a), Figure 5(b), and Figure 5(c), corresponding to the 3 categories of benchmarks.

Figure 5(a) shows the effect of cache size on *Shuffle-light* benchmarks. These benchmarks all have very little shuffle date and very short Reduce phase (the Reduce phase is no longer than 2.1% of the whole time), thus, the Localized Cache occupies little space and most of the space is allocated to Input Space, the speedup of these benchmarks is mostly due to the caching of Input Data. When the cache size is 5GB per node, the speedup is very small due to insufficient space to hold Input Data. As the cache size increases, the speedup rises significantly and getting the maximum point when the cache size is about 90GB.

Figure 5(b) shows the effect of cache size on *Shuffle-medium* benchmarks. These benchmarks have some volume of shuffle data (no more than 1GB), both Map and Reduce phase could be accelerated by caching Localized Data. When the cache size per node is 5GB, all the Localized Data is cached, thus Reduce phase gets an average speedup of 59.99%. However, the Reduce phase only oc-

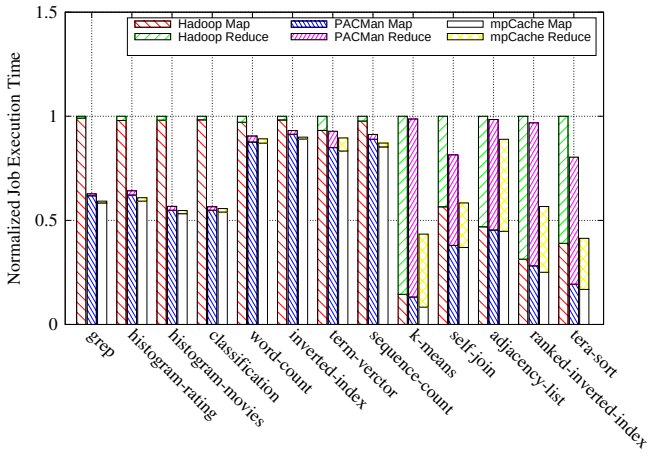


Fig. 4. Job execution time comparison with Hadoop and PACMan on 8 CPU-cores.

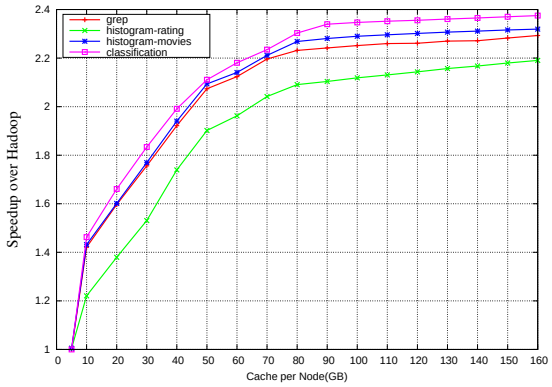
cupies 3.43% of the whole time, resulting in the speedup of the whole job of only 1.40%. As the cache size increases, the speedup increases due to the reduction of Map phase time and getting the maximum speedup when the cache size is about 100GB.

Figure 5(c) shows the effect of cache size on *Shuffle-heavy* benchmarks. These benchmarks have very large volume of shuffle data, resulting in the Localized Data space occupies as large as 32GB when running *tera-sort* with 30GB Input Data. Thus ,when the cache size is below 40GB, most of the cache is allocated to Localized Cache, and the speedup is due to caching Localized Data.

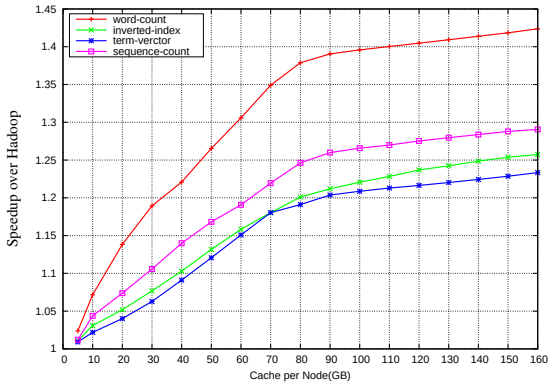
4 Related Work

MapReduce Optimization on Multi-Core Server. With the emerging of multi-core systems, MapReduce frameworks were also proposed and optimized on multi-core server [15][19][18]. All of these frameworks are designed for a single server, of which [18] mainly focused on graphics processors and [15][19] were implemented on symmetric-multiple-processor server. Obviously, these single-node frameworks could only process gigabytes of data at most and are stretched so thin to handle terabytes or petabytes of data.

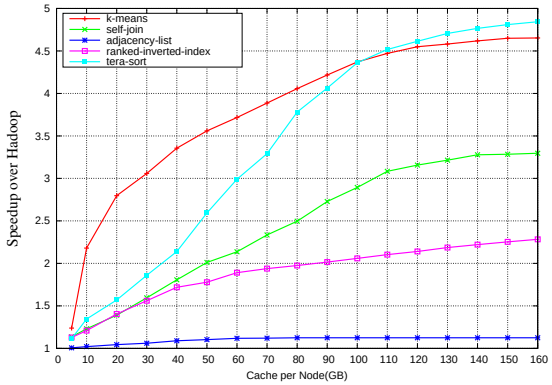
MapReduce Optimization via In-Memory Cache. PACMan [2] and HaLoop [3] cached input data in memory to reduce IO cost of hard disks and optimize performance. Since the task parallelism degree of new generation of MapReduce (e.g., YARN) is more concerns about free memory. Caching data in memory consumes additional memory and cuts down the task parallelism, thus leading to low performance for some memory-intensive jobs. Due to limitation of memories and the large volume of Localized Data, PACMan only cached



(a) Shuffle-light



(b) Shuffle-medium



(c) Shuffle-heavy

Fig. 5. The effect of cache size on mpCache.

Input Data, thus only Map phase was improved. However, many MapReduce applications consist of heavy Reduce phase (e.g., *k-means* and *tera-sort*), our caching Localized Data also significantly improves Reduce phase and gets better acceleration of the whole job.

IO Optimization via SSD Cache. Yongseok et al. [12] proposed balancing data in cache and update cost for optimal performance of SSD. Hystor [4], Proximal IO [16], SieveStore [14], and HybridStore [10] used SSD as the cache of hard disks. However, these works only cache small files (e.g., size below 200KB), and only work for a single node. mpCache works in unison of all the nodes and makes use of relatively complex and efficient eviction scheme to make better support for MapReduce.

5 Conclusion

As a widely used programming model and implementation for processing large data sets, MapReduce does not scale well on many-core clusters due to the IO restriction of storage. Emerging of SSD provides a good trade off between cost and performance and caching data in SSD also prevents the problem of in-memory's degradation of computing parallelism degree. In this paper, we proposed mpCache, an SSD-based universal caching system for MapReduce, which caches both Input Data and Localized Data to speed up all the IO-consuming phases– **Read**, **Spill**, and **Merge**. We implemented mpCache in Hadoop and evaluated it on a 7-node cluster. The results show that mpCache can get a speedup of 2.09 times over Hadoop, and 1.79 times over PACMan.

Acknowledgment

This Work is co-supported by National Basic Research (973) Program of China (2011CB302505), Natural Science Foundation of China (61170210), and National High-Tech R&D (863) Program of China (2011AA01A203).

References

1. F. Ahmad, S. Lee, M. Thottethodi, and T. Vijaykumar, "Puma: Purdue mapreduce benchmarks suite," 2012, http://web.ics.purdue.edu/ fahmad/benchmarks.htm.
2. G. Ananthanarayanan, A. Ghodsi, A. Wang, D. Borthakur, S. Kandula, S. Shenker, and I. Stoica, "Pacman: Coordinated memory caching for parallel jobs," in *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation, NSDI'12*. USENIX, 2012, pp. 20–20.
3. Y. Bu, B. Howe, M. Balazinska, and M. D. Ernst, "Haloop: Efficient iterative data processing on large clusters," *Proceedings of the VLDB Endowment*, vol. 3, no. 1-2, pp. 285–296, 2010.
4. F. Chen, D. A. Koufaty, and X. Zhang, "Hystor: making the best use of solid state drives in high performance storage systems," in *Proceedings of the international conference on Supercomputing, ICS'11*. ACM, 2011, pp. 22–32.

5. J. Dean and S. Ghemawat, "Mapreduce: simplified data processing on large clusters," *Communications of the ACM*, vol. 51, no. 1, pp. 107–113, 2008.
6. 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*. ACM, 1995.
7. S. Ghemawat, H. Gobioff, and S.-T. Leung, "The google file system," in *ACM SIGOPS Operating Systems Review*, vol. 37. ACM, 2003, pp. 29–43.
8. J. Handy, "Flash memory vs. hard disk drives - which will win?" http://http://www.storagesearch.com/semico-art1.html.
9. M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly, "Dryad: distributed data-parallel programs from sequential building blocks," *ACM SIGOPS Operating Systems Review*, vol. 41, no. 3, pp. 59–72, 2007.
10. Y. Kim, A. Gupta, B. Urgaonkar, P. Berman, and A. Sivasubramaniam, "Hybridstore: A cost-efficient, high-performance storage system combining ssds and hdds," in *2011 IEEE 19th International Symposium on Modeling, Analysis & Simulation of Computer and Telecommunication Systems, MASCOTS'11*. IEEE, 2011, pp. 227–236.
11. D. E. Knuth, *The art of computer programming, vol. 3, Addison-Wesley, Reading Mass.* Pearson Education, 2005.
12. 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 *Proceedings of the 10th USENIX Conference on File and Storage Technologies, FAST'12*. USENIX, 2012, pp. 25–25.
13. J. Ousterhout, P. Agrawal, D. Erickson, C. Kozyrakis, J. Leverich, D. Mazières, S. Mitra, A. Narayanan, G. Parulkar, M. Rosenblum *et al.*, "The case for ramclouds: scalable high-performance storage entirely in dram," *ACM SIGOPS Operating Systems Review*, vol. 43, no. 4, pp. 92–105, 2010.
14. T. Pritchett and M. Thottethodi, "Sievestore: a highly-selective, ensemble-level disk cache for cost-performance," in *Proceedings of the 37th Annual International Symposium on Computer Architecture, ISCA'10*. ACM, 2010, pp. 163–174.
15. C. Ranger, R. Raghuraman, A. Penmetsa, G. Bradski, and C. Kozyrakis, "Evaluating mapreduce for multi-core and multiprocessor systems," in *IEEE 13th International Symposium on High Performance Computer Architecture, HPCA'07*. IEEE, 2007, pp. 13–24.
16. J. Schindler, S. Shete, and K. A. Smith, "Improving throughput for small disk requests with proximal i/o." in *Proceedings of the 9th USENIX Conference on File and Storage Technologies, FAST'11*. USENIX, 2011, pp. 133–147.
17. K. Shvachko, H. Kuang, S. Radia, and R. Chansler, "The hadoop distributed file system," in *2010 IEEE 26th Symposium on Mass Storage Systems and Technologies, MSST'10*. IEEE, 2010, pp. 1–10.
18. J. A. Stuart and J. D. Owens, "Multi-gpu mapreduce on gpu clusters," in *2011 IEEE International Parallel & Distributed Processing Symposium, IPDPS'11*. IEEE, 2011, pp. 1068–1079.
19. J. Talbot, R. M. Yoo, and C. Kozyrakis, "Phoenix++: modular mapreduce for shared-memory systems," in *Proceedings of the second international workshop on MapReduce and its applications*. ACM, 2011, pp. 9–16.
20. A. Thusoo, J. S. Sarma, N. Jain, Z. Shao, P. Chakka, S. Anthony, H. Liu, P. Wyckoff, and R. Murthy, "Hive: a warehousing solution over a map-reduce framework," *Proceedings of the VLDB Endowment*, vol. 2, no. 2, pp. 1626–1629, 2009.