EarnCache: Self-adaptive Incremental Caching for Big Data Applications

Yifeng Luo[1,2], Junshi Guo[1], and Shuigeng Zhou[1(✉)]

[1] School of Computer Science, and Shanghai Key Lab of Intelligent Information Processing, Fudan University, Shanghai 200433, China
sgzhou@fudan.edu.cn

[2] School of Data Science and Engineering, East China Normal University, Shanghai 200062, China

Abstract. Memory caching plays a crucial role in satisfying the requirements for (quasi-)real-time processing of exploding data on big-data clusters. As big data clusters are usually shared by multiple computing frameworks, applications or end users, there exists intense competition for memory cache resources, especially on small clusters that are supposed to process comparably big datasets as large clusters do, yet with tightly limited resource budgets. Applying existing on-demand caching strategies on such shared clusters inevitably results in frequent cache thrashing when the conflicts of simultaneous cache resource demands are not mediated, which will deteriorate the overall cluster efficiency.

In this paper, we propose a novel self-adaptive incremental big data caching mechanism, called EarnCache, to improve the cache efficiency for shared big data clusters, especially for small clusters where cache thrashing may occur frequently. EarnCache self-adaptively adjusts resource allocation strategy according to the condition of cache resource competition: turning to incremental caching to depress competition when resource is in deficit, and returning to traditional on-demand caching to expedite data caching-in when resource is in surplus. Extensive experimental evaluation shows that the elasticity of EarnCache enhances the cache efficiency on shared big data clusters, and thus improves resource utilization.

Keywords: Big data · Cache management
Self-adaptive and Incremental caching

1 Introduction

As big data techniques and infrastructures are being applied to facilitate and accelerate the processing of big data with formidable size, people are putting forward eager requests on (quasi-)real-time processing of big datasets yet with exploding volumes, while meeting the (quasi-)real-time processing requests of big datasets is usually held back by the disk-based storage subsystem, because

Y. Cai et al. (Eds.): APWeb-WAIM 2018, LNCS 10988, pp. 379–393, 2018.
https://doi.org/10.1007/978-3-319-96893-3_29

of the expanding tremendous performance gap lying between magnetic disks and processing units. Thus memory caching plays a crucial role in bridging the performance gap between storage subsystems and computing frameworks, and gradually becomes the determinant factor of whether the processing units of big data platforms could work at their wire speed to satisfy the vast and fast data processing requirements. As more and more time-critical applications commence employing memory to cache their big datasets, big data clusters are usually concurrently shared by multiple computing frameworks, applications or end users, just as Fig. 1 shows.

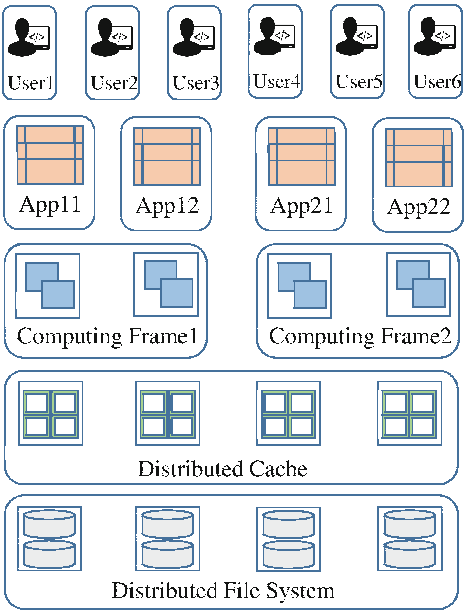


Fig. 1. Big data application hierarchies.

In Web or traditional OLTP database applications, ranges or blocks of the same datasets (or files) usually show vast variance in "hotness" regarding access recency and frequency. While big data applications usually scan their input files as a whole for data processing, and thus all blocks of the same file reveal almost equal hotness. On the other hand, traditional system-level or database-level data caching is executed on small data units (i.e. 8 KB-sized pages), while big data caching is executed on much larger units (i.e. 256 MB-sized blocks). So the cost of caching in/out a data unit in big data scenarios far exceeds that of traditional data caching. Accordingly, traditional caching may have millions of caching slots, which makes hotter data pages less likely to be cached out by colder data pages; while big data caching may only have thousands of slots, which makes comparatively hotter data blocks vulnerable to be cached out by colder data blocks.

Thus there exists intense competition for memory cache resources, especially on small clusters which are supposed to process comparably big datasets as large clusters do, yet with much tightly limited resource budgets.

Intense competition for computational resources would not do much harm to the clusters' running efficiency, while intense competitions for memory cache resources would engender tremendous harm, where frequent cache thrashings would be incurred if cache resource demands are not coordinated, and CPUs may constantly run idle. Applying existing on-demand caching strategies, which cache in data blocks once they are accessed, on small shared clusters inevitably results in frequent cache thrashings, and thus leads to deteriorated overall cluster efficiency. The principal reason behind is that aggressively caching massive numbers of data blocks of big datasets on demand causes constant block replacement in cache, and consequently exacerbates the competition of cache resources when these resources are in strong need. Consequently implementing effective and adaptive management on memory cache resources becomes increasingly important for the efficiency of big data clusters, especially for small/medium enterprises who could only afford non-big (or small) clusters.

Targeting the caching problem existing on non-big clusters, we propose an adaptive cache mechanism, which is named as EarnCache (from s**E**lf-**a**daptive inc**r**emen**t**al **Cache**), to coordinate concurrent cache resource demands to prevent exacerbation of cache efficiency, when intense competitions for memory cache resources occur. As big data applications usually access their input data in the Write-Once-Read-Many (WORM) fashion, we only consider read caching in this paper. Major contributions of this paper include: (1) proposing an incremental caching mechanism which could self-adaptively adjust cache allocation strategies according to the competition condition of cache resources; (2) formulating and solving the cache resource allocation and replacement problem as an optimization problem; (3) implementing a prototype of the proposed mechanism, and performing extensive experiments to evaluate the effectiveness of the proposed mechanism.

With EarnCache, applications or end users do not get their datasets cached once they are accessed, but have to incrementally earn cache resources from other applications or end users by accessing their datasets. A dataset is cached gradually as the upper-level application or end user accesses the dataset, and more blocks of the dataset get cached each time it is accessed. In the rest of this paper, we illustrate the system design and the implementation details of EarnCache in Sect. 2. We provide empirical evaluation results in Sect. 3, and present related work in Sect. 4. We finally conclude the paper in Sect. 5.

2 Framework and Techniques

We illustrate how EarnCache works in this section. Firstly we present the overview about the caching mechanism of EarnCache, and then discuss its architecture design, and finally explain the incremental cache-earning policy and its implementation.

2.1 Overview

On a shared non-big cluster with relatively limited cache capacity, cache resource conflicts would be normal. If the cluster is concurrently used by a moderate number of users and the competition for cache resources is mild, hot blocks is less likely to be cached out by cold blocks, and then applying on-demand caching could expedite hot blocks taking over cache resources from cold blocks. If more and more users need to use the cluster concurrently and competition for cache resources gets wild, applying on-demand caching would leave concurrent cache resource demands unmediated, and making hot blocks more vulnerable to being cached out by cold blocks. Then files which are frequently accessed recently could be totally cached out by files which would rarely be accessed for a second time in the near future, and the flushed-out hot files would require to be cached in soon as their next access should occur in the upcoming future. We consequently need to revisit existing on-demand caching mechanisms for big data caching on small clusters, and propose more effective measures to improve the efficiency of data caching on such clusters.

We believe that a good caching strategy for non-big clusters should be self-adaptive to resource competition conditions, depressing competitions and preventing cache thrashings when cache resources are in desperate deficit. Obviously caching big data files on demand as a whole could not provide such self-adaptivity. Not caching-in files entirely on-demand could provide the elasticity of tuning the amount of cache resources allocated for different files, based on their access recency and frequency.

Ideally, more recently frequently accessed files should be assigned with more cache resources, and less recently frequently accessed ones should be assigned with less cache resources. However, it's not possible to know in advance what files would be frequently accessed in the upcoming future, and we could only make predictions based on historical file access patterns, especially the most recent information. Based on files' historical access information, EarnCache implements an incremental caching strategy, where a user should earn cache resources for its files from other concurrent users via accessing these files. Cache resources are incrementally allocated to a file that becomes more frequently accessed, which gradually takes over cache resources, until all blocks of the file have been cached in. The more a file is accessed, the more cache resources it takes over. The incremental caching strategy ensures that files occupying cache resources are recently frequently accessed, and will not be flushed out by files that are only accessed occasionally or randomly.

2.2 Architecture

Files originally reside in the under distributed file system (e.g. Hadoop File System), and EarnCache coordinately caches files across the whole cluster. EarnCache consists of a central *master* and a set of *workers* residing on storage nodes as shown in Fig. 2. The *master* is responsible for:

1. determining the cache resource allocation plan for a file, concerning how many cache resources should be allocated to the file based on its recent access information;
2. informing workers of cache resource allocation plans via heartbeats;
3. keeping track of metadata of which storage node a cached block resides on;
4. answering clients' queries on cache metadata.

And a *worker* is responsible for:

1. receiving the resource allocation plan from the master;
2. calculating resource composition plans to determine how many cache resources a cached file should contribute to compose the allocated resources depicted in the cache plan;
3. caching in/out blocks according to the calculated resource composition plans;
4. informing the master of cached blocks via heartbeats;
5. serving clients with cached blocks;
6. transferring in-memory blocks to other workers for remote caching.

As illustrated in Fig. 2, a client accesses a block in the following procedures:

1. the client queries the master where the block is cached;
2. the master tells the client which worker the requested block resides on;
3. the client contacts the worker to access the cached block;
4. the worker serves the client with the block data from cache.

One thing worth noting here is that: the client will not contact any worker to access a block if the block is not cached in any worker node, as the master only keeps track of cached blocks. In this situation, the client has to fetch data directly from the under file system.

2.3 Incremental Caching

As we prefer recently frequently accessed files incrementally taking over resources from less recently frequently accessed files, "recently" should be defined quantitatively before we could design the incremental caching strategy, and other related elements should also be clarified. Table 1 presents the definitions of all notations involved in our incremental caching strategy.

We define a function $h_i(x_i)$ to denote the cache profit gain of the ith file to instruct how cache resources should be allocated across all files falling within the observation window. Then we attempt to maximize the total profit gain of all files falling in the observation window with the profit gain function, just as Eq. 1 shows.

$$\sum_{i=1}^{N} f_i \cdot h_i(x_i) \tag{1}$$

According to definitions in Table 1, we can assume that the time it takes to scan the ith file is:

$$time(x_i) = [a \cdot x_i + b \cdot (1 - x_i)] \cdot d_i \tag{2}$$

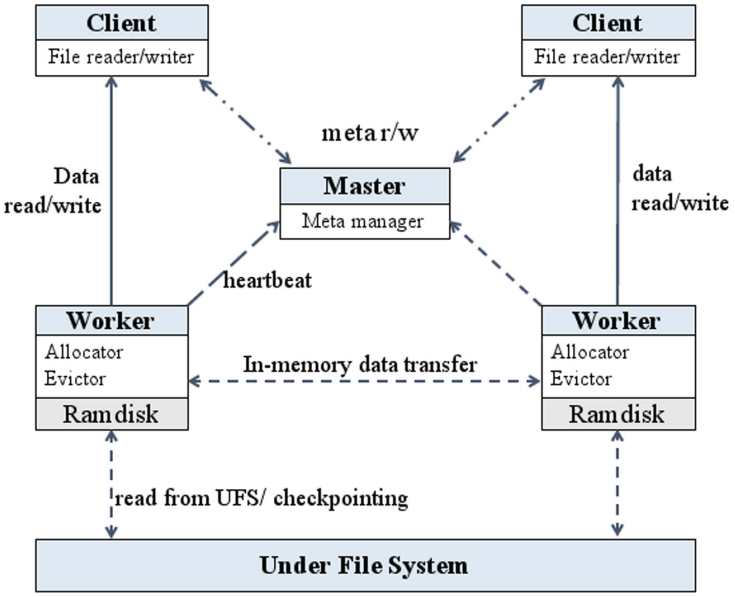


Fig. 2. EarnCache's architecture.

Table 1. Notation definitions

Notation	Definition
W	Predefined window size of the most recently accessed data for observing files falling within
a, b	Scan time per unit data from memory(a) and hdd(b)
N	Total number of files falling in the observation window
d_i	Data size of the ith file
D	Total data size of N files
M	Cache capacity of the whole cluster
f_i	Access frequency of the ith file
F	Total access frequency of N files
x_i	Percentage of data cached for the ith file
$h_i(x_i)$	The ith file's profit gain with x_i data cached

As mentioned above, we use h_i to indicate the ith file's cache profit gain with x_i data cached. For simplicity, we take the file's saved scan time as its cache profit gain, then we can define h_i's deviation at x_i as its gain change over Δx_i, which could be further defined as the percentage of increased saving of the file's scan time with increased cache share at x_i over the total saved scan time at x_i, compared to zero cache share, just formulized as:

$$\frac{\delta h_i}{\delta x_i} = \frac{time(x_i) - time(x_i + \delta x_i)}{time(0) - time(x_i)} = \frac{\delta x_i}{x_i} \tag{3}$$

Thus we can derive that $h_i(x_i) = \ln x_i$, and now our optimization goal becomes:

$$\sum_{i=1}^{N} f_i \cdot \ln x_i \tag{4}$$

subjected to:

$$\sum_{i=1}^{N} x_i \cdot d_i \leq M \tag{5}$$

Note that at any given time, x_i is the only variant contained in the optimization goal, and $f_i \cdot \ln x_i$ is a convex function. After applying Lagrange multiplier method, our optimization goal turns to:

$$L = \sum_{i=1}^{N} f_i \cdot \ln x_i - \lambda(\sum_{i=1}^{N} x_i \cdot d_i - M) \tag{6}$$

Let $\frac{\delta L}{\delta x_i}$ be 0, then we get

$$x_i \cdot d_i = \frac{f_i}{F} \cdot M \tag{7}$$

The above result shows that the amount of memory resources allocated to a file is linear to f_i at a given moment, as all files' access frequencies are determined at that moment, which exactly corresponds to our original intention of incremental caching. One more thing worth noting is that: if the overall size of files falling within the whole observation window is smaller than the cache capacity, and there are cache resources being occupied by files that fall out of the observation window, EarnCache will collect resources from those obsolete files by LRU when there is a caching request, and the requesting file could cache in its blocks once and for all, rather than gradually taking over resources from files falling within the observation window. EarnCache thereby could adaptively devolve to traditional on-demand caching so as to expedite the process of collecting cache resources for actively accessed files when contention for cache resources is light, and evolve to incremental caching to depress competition when resources are in deficit.

2.4 Implementation Details

We implemented EarnCache by implanting our incremental caching mechanism into the modified Tachyon [4]. In EarnCache, we first evenly re-distribute a file's cached data blocks across the whole cluster, so that almost the same amount of blocks are hosted in cache on each cluster node, and all workers can manage their cache resources independently yet still in concert. As uneven data distribution will drag down completion of the whole job, evenly distributing cached data

blocks guarantee that tasks running on each node could ideally finish almost simultaneously.

When the ith file needs caching, EarnCache pre-allocates f_i/F fraction of cache resources on each node to the file based on Eq. 7. If resources pre-allocated to the file are more than its aggregated demands, EarnCache has other files in need of cache resources fairly share the spare cache. Each worker checks its available cache resources, and allocates as many as possible to them directly, which could make full use of cache resources. When there are not enough resources available, the worker calls *BlocksToEvict()*, which implements the eviction algorithm with incremental caching, to determine which blocks should be cached out. As all blocks are cached in from the under file system, cached-out blocks need no more backup and thus workers could discard them directly from cache. When block eviction process is done, the worker will inform the master to update the metadata.

Algorithm 1 describes the process of evicting blocks. EarnCache first checks whether the file requesting cache resources has used up its pre-allocated share in Lines 1–3. In the while loop, files who have overcommitted the most cache resources are selected in Lines 7–14. If no such file exists, EarnCache will reject the cache request (Lines 15–17). Otherwise, blocks of these selected files are added to the candidate block set until enough cache resources have been collected (Lines 18–24). As recency and frequency of all blocks within the same file are identical, workers do not differentiate between blocks of the same file when selecting blocks to cache out.

3 Empirical Evaluation

We deploy an HDFS cluster on Amazon EC2 as the under distributed file system to evaluate EarnCache's performance, on which Spark and EarnCache are deployed as the upper-level application tier and the middle-level caching tier respectively. The cluster consists of five Amazon EC2 m4.2xlarge nodes, one of which serves as the master and the other four serve as slaves. Each cluster node has 32 GB of memory, 12 GB memory is reserved as working memory and the remaining 20 GB of memory is employed as cache resources, summing up to 80 GB of overall cache in total.

We mainly evaluate EarnCache's performance by issuing jobs from Spark to scan files in parallel without any further processing, and compare the performance of EarnCache incremental caching, with LRU and LFU on-demand caching, and MAX-MIN fair caching. We set the size of FILE-1, FILE-2 and FILE-3 equally to 40 GB and unequally to 70 GB, 40 GB and 10 GB respectively, and then evaluate EarnCache with different caching strategies and frequency patterns. We set the observation window size of EarnCache to 1000 GB by default. For each experiment, we issue file scanning jobs on three input files, denoted as FILE-1, FILE-2 and FILE-3, with the following three various frequency patterns, denoted as ROUND, ONE and TWO respectively.

Algorithm 1. Eviction Algorithm: BlocksToEvict()

Input: s, requested cache resources; r, the requesting file id; A=$\{a_1, a_2...a_N\}$, a list of files' pre-allocated memory bytes; C=$\{c_1, c_2...c_N\}$, a list of current consumed memory bytes in local node; M, memory capacity of local node

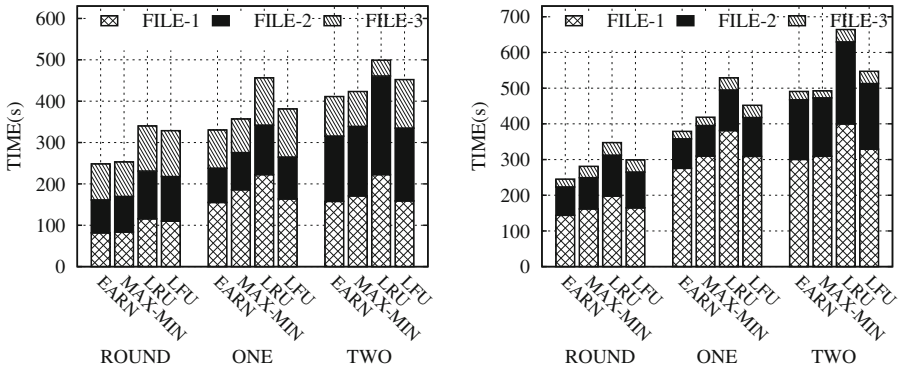
Output: a list of candidate blocks to evict

```
1: if c_r ≥ a_r then
2:     algorithm ends as file r has already consumed all its allocated memory
3: end if
4: candidate ← {}                          ▷ candidate cached out blocks
5: mem ← 0                                 ▷ free resources obtained from evicting candidates
6: while mem < s do
7:     j ← −1
8:     over_j ← 0
9:     for a_i in A and i ≠ r do
10:        if c_i − a_i > over_j then
11:            j ← i
12:            over_j ← c_i − a_i
13:        end if
14:    end for
15:    if j = −1 then
16:        return as request failure
17:    end if
18:    find b_j as a block of file j and not in candidate
19:    candidate ← candidate + b_j
20:    mem ← mem + sizeof(b_j)
21:    c_j ← c_j − sizeof(b_j)
22:    if mem ≥ s then
23:        return candidate
24:    end if
25: end while
```

- ROUND Three files are accessed in pattern: FILE-1, FILE-2, FILE-3, ..., where three files are accessed with equal frequency.
- ONE Three files are accessed in pattern: FILE-1, FILE-2, FILE-1, FILE-3, ..., where one file is accessed more frequently than other two files.
- TWO Three files are accessed in pattern: FILE-1, FILE-2, FILE-1, FILE-2, FILE-3, ..., where two files are accessed more frequently than the other file.

Figure 3(a) and (b) show the averaged overall running time of file scanning jobs. Each group of columns involves the scanning of files contained within the whole period of a frequency pattern, namely 3, 4, and 5 files respectively. We can see that EarnCache yields the best performance, which exceeds that of the LRU and LFU on-demand caching by a large margin, and leads the MAX-MIN fair caching by a smaller margin. The reason of EarnCache achieving the best performance is straight-forward, as it prevents cache thrashings and thus more blocks are accessed from memory. We can see that the performance of EarnCache is only slightly better than the MAX-MIN caching strategy, and sometimes they

achieve similar performance. This is because files receive similar amounts of cache resources from these two caching strategies, as far as our experimental settings are concerned.



(a) Files with equal size

(b) Files with unequal size

Fig. 3. Running time of file scanning jobs.

In the meanwhile, we also observe that the performance of EarnCache is not as mighty as we have expected, especially compared with the LRU and LFU on-demand caching. The reasons are twofold: (1) EarnCache could not hold all blocks in cache, and thus file scanning jobs are sped up partially; (2) cache-locality is not guaranteed, and a prohibitive number of blocks are accessed from remote cache, rather than local cache. We analyze the distribution of blocks accessed from local cache, remote cache, and the under file system respectively in detail, and the results are shown in Fig. 4. We can see that EarnCache has the largest number of blocks accessed from cache, whether locally or remotely, which means that it yields the highest memory efficiency than other caching strategies. However, we observe that EarnCache has the largest number of blocks accessed from remote cache among the four evaluated strategies. This means EarnCache has the largest potential of performance improvement. If cache-aware task scheduling can be integrated into the upper-level task scheduler, more blocks will be accessed from local cache and EarnCache could obtain much better overall performance.

We showcase the change of cache shares of different files during the process of executing file scanning jobs iteratively, and the results are shown in Fig. 5. We can see that the cache shares of different files with EarnCache remain stable across the whole experimental process, while the LRU and LFU on-demand caching strategies witness cache thrashings with huge variance of cache shares. The MAX-MIN caching statically allocates cache resources based on present files, rather than caching blocks on demand, and thus also witnesses no variance of cache shares and avoid cache thrashings. However, we can also see that the

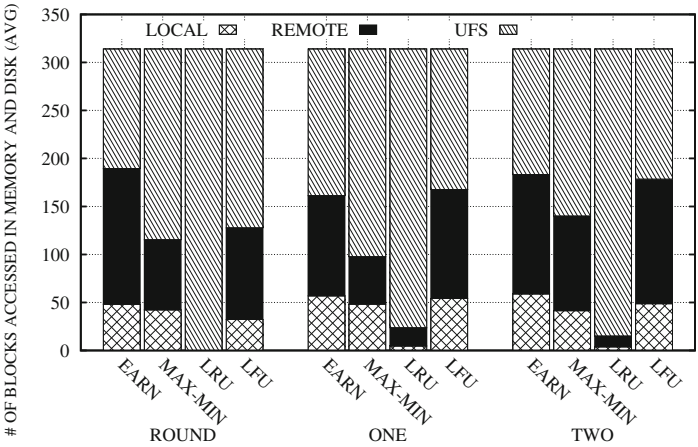


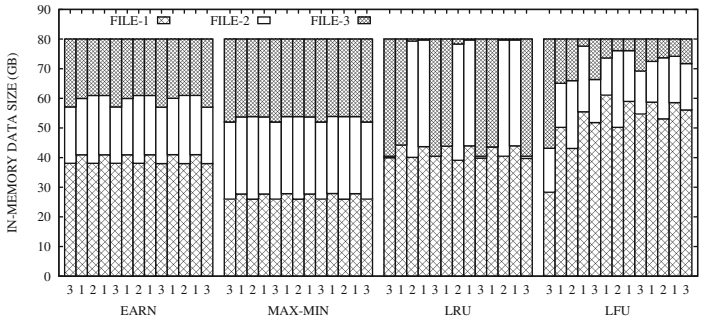
Fig. 4. Distribution of blocks accessed in local cache, remote cache and under file system.

MAX-MIN caching is unable to dynamically re-allocate resources properly when there exist files not receiving any further accesses. To illustrate this, we present the process of resource re-allocation of EarnCache and the MAX-MIN caching in Fig. 6(a) and (b), where two out of the three equal-sized files stop receiving further accesses. We can see that the file remaining accessed gradually takes over cache resources from those obsolete files as time passes, while with MAX-MIN fair caching, the amount of cache resources held by each file does not change. Correspondingly, the running time of each job gradually decreases with EarnCache, yet remains stable with MAX-MIN.

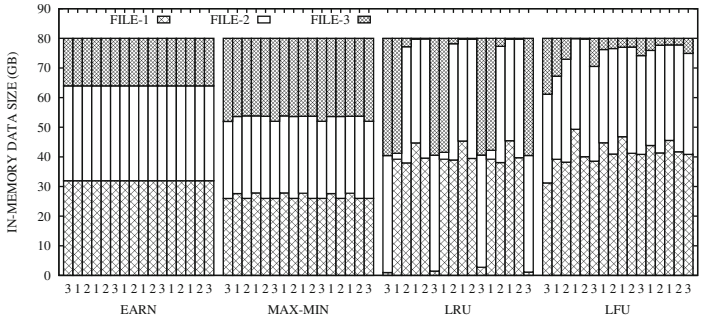
Finally, we experimentally analyze the impact of the predefined observation window size, and the results are presented in Fig. 7. When observation window is set with small sizes, the competition for cache resources could not be coordinated properly, and thus the overall cache efficiency and performance degrades greatly. When the observation window size exceeds 200 GB, which is larger enough compared with the file sizes, EarnCache effectively coordinates cache resources and the performance improves correspondingly.

4 Related Work

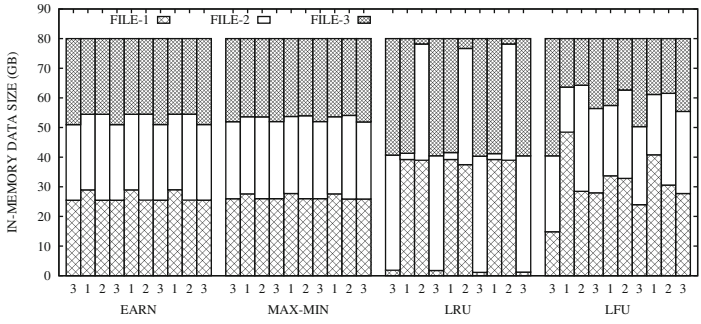
There has been extensive work on memory storage and caching, as more and more time-critical applications [19,22] require to store or cache data in memory to gain improved data access performance, such as Ousterhout et al. proposed RAMCloud [2] to keep data entirely stored in memory for large-scale Web applications, and Spark [9,21] enables in-memory MapReduce [3]-style parallel computing by leveraging memory to store and cache distributed (intermediate) datasets. While caching on distributed parallel systems is tremendously different from traditional centralized page-based file system or database caching, and



(a) One



(b) Two

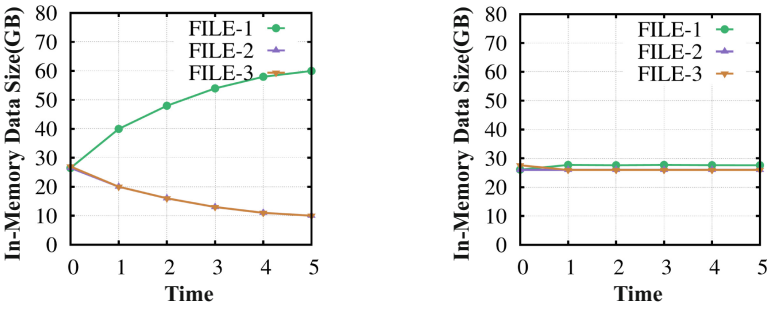


(c) Round

Fig. 5. Change of cache shares as files being accessed.

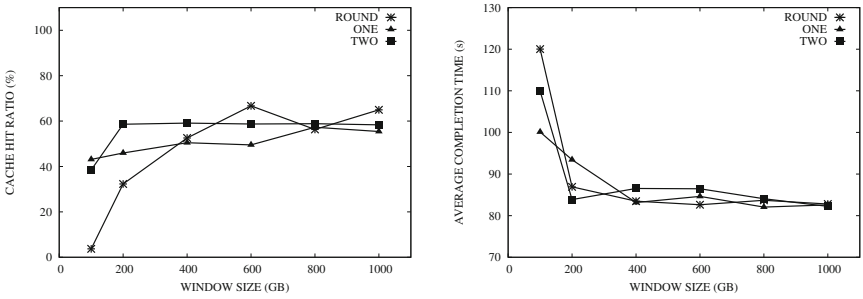
directly applying centralized caching usually does not help much to improve and sometimes even hurts cache efficiency and performance.

Some previous work focuses on implementing an additional layer on existing distributed file system, which enables applications to cache distributed datasets from the underlying distributed file system. Zhang et al. [1] and Luo et al. [11] respectively proposed the HDCache and RCSS distributed cache system based



(a) EarnCache (b) MAX-MIN

Fig. 6. Dynamic resource re-allocation with two files receiving no further accesses.



(a) Cache hit ratio (b) Job running time

Fig. 7. Impact of the observation window size.

on HDFS [6,20], which manages cached data just as HDFS manages disk data. Li et al. [4] further implemented a distributed memory file system for data caching by checkpointing data to the underlying file system. Luo et al. [16] proposed a just-in-time data prefetching mechanism for Spark applications so as to depress the resource demands for caching memory. EARNCache imbeds the incremental caching into Tachyon [4] to coordinate resource competitions and avoid cache thrashings, and improves cache efficiency and resource fairness to a certain degree.

Some work focuses on optimizing data caching for specific frameworks or goals. Zhang et al. [10] proposed to cache MapReduce intermediate data to speed up MapReduce applications. Luo et al. [14,15] optimized cache resource allocations in cloud environments to improve database workload processing efficiency. Ananthanarayanan et al. [7] found the important All-or-Nothing property, which implies that all or none input data blocks of tasks within the same wave should be cached, and then proposed PACMan to coordinate memory caching for parallel jobs. Li et al. [5], Tang et al. [17] and Ghodsi et al. [18] respectively proposed dynamic resource partition strategies to improve fairness, and maximize the

overall performance in the meanwhile. Pu et al. [8] extended the MAX-MIN fairness [12,13] with probabilistic blocking, and proposed FairRide to avoid cheating and improve fairness for shared cache resources.

5 Conclusion

In this paper, we propose the EarnCache incremental big data caching mechanism, which adaptively adjusts resource allocation strategy according to resource competition condition. Concretely, when the resources are in deficit, it adopts incremental caching to depress competition, and turns to traditional on-demand-caching to expedite data caching-in when resources are in surplus. On-demand big data cache usually leads to cache thrashings. With EarnCache, files are not cached on demand. Instead, applications or end users incrementally take over cache resources from others by accessing their datasets. EarnCache manages to achieve improved resource utilization and performance with such an incremental caching strategy. Experimental results show that EarnCache can elastically manage cache resources and yields better performance against the LRU, LFU and MAX-MIN cache replacement policies.

Acknowledgements. This work was supported by National Natural Science Foundation of China (NSFC) (No. U1636205), and the Science and Technology Innovation Action Program of Science and Technology Commission of Shanghai Municipality (STCSM) (No. 17511105204).

References

1. Zhang, J., Wu, G., Hu, X., et al.: A distributed cache for Hadoop distributed file system in real-time cloud services. In: Proceedings of GRID, pp. 12–21 (2012)
2. Ousterhout, J., Agrawal, P., Erickson, D., et al.: The case for RAMCloud. Commun. ACM **54**(7), 121–130 (2011)
3. Dean, J., Ghemawat, S., et al.: MapReduce: simplified data processing on large cluster. In: Proceedings of OSDI, pp. 137–150 (2004)
4. Li, H., Ghodsi, A., Zaharia, M., et al.: Tachyon: reliable, memory speed storage for cluster computing frameworks. In: Proceedings of SOCC, pp. 6:1–6:15 (2014)
5. Li, Y., Feng, D., Shi, Z.: Enhancing both fairness and performance using rate-aware dynamic storage cache partitioning. In: Proceedings of DISCS, pp. 31–36 (2013)
6. Shvachko, K., Kuang, H., Radia, S., et al.: The Hadoop distributed file system. In: Proceedings of MSST, pp. 121–134 (2010)
7. Ananthanarayanan, G., Ghodsi, A., Warfield, A., et al.: PACMan: coordinated memory caching for parallel jobs. In: Proceedings of NSDI, pp. 267–280 (2012)
8. Pu, Q., Li, H., Zaharia, M., et al.: FairRide: near-optimal, fair cache sharing. In: Proceedings of NSDI, pp. 393–406 (2016)
9. Zaharia, M., Chowdhury, M., Das, T., et al.: Resilient distributed datasets: a fault-tolerant abstraction for in-memory cluster computing. In: Proceedings of NSDI, pp. 15–28 (2012)
10. Zhang, S., Han, J., Liu, Z., et al.: Accelerating MapReduce with distributed memory cache. In: Proceedings of ICPADS, pp. 472–478 (2009)

11. Luo, Y., Luo, S., Guan, J., et al.: A RAMCloud storage system based on HDFS: architecture, implementation and evaluation. J. Syst. Softw. **86**(3), 744–750 (2013)
12. Ma, Q., Steenkiste, P., Zhang, H.: Routing high-bandwidth traffic in max-min fair share networks. In: Proceedings of SIGCOMM, pp. 206–217 (1996)
13. Cao, Z., Zegura, W.: Utility max-min: an application-oriented bandwidth allocation scheme. In: Proceedings of INFOCOM, pp. 793–801 (1999)
14. Luo, Y., Guo, J., Zhu, J., Guan, J., Zhou, S.: Towards efficiently supporting database as a service with QoS guarantees. J. Syst. Softw. **139**, 51–63 (2018)
15. Luo, Y., Guo, J., Zhu, J., Guan, J., Zhou, S.: Supporting cost-efficient multi-tenant database services with service level objectives (SLOs). In: Candan, S., Chen, L., Pedersen, T.B., Chang, L., Hua, W. (eds.) DASFAA 2017. LNCS, vol. 10177, pp. 592–606. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-55753-3_37
16. Luo, Y., Shi, J., Zhou, S.: JeCache: just-enough data caching with just-in-time prefetching for big data applications. In: Proceedings of ICDCS, pp. 2405–2410 (2017)
17. Tang, S., Lee, B., He, B., et al.: Long-term resource fairness: towards economic fairness on pay-as-you-use computing systems. In: Proceedings of ICS, pp. 251–260 (2014)
18. Ghodsi, A., Zaharia, M., Hindman, B., et al.: Dominant resource fairness: fair allocation of multiple resource types. In: Proceedings of NSDI, pp. 323–336 (2011)
19. Redis. http://redis.io
20. HDFS. http://hadoop.apache.org/hdfs
21. Spark. http://spark.apache.org
22. Memcached. http://danga.com/memcached