Computer Science and Information Systems 11(1):271–289 DOI: 10.2298/CSIS130127008L

Towards hybrid client-side cache management in network-based file systems

Xiuqiao Li[1,2,3], Limin Xiao[1,2], Ke Xie[2], Bin Dong[2], Li Ruan[1,2], and Dongmei Liu[2]

[1] State Key Laboratory of Software Development Environment, Beihang University
Beijing, China
[2] School of Computer Science and Engineering, Beihang University
Beijing, China
[3] IBM China Systems and Technology Laboratory
Beijing, China
lxiuqiao@cn.ibm.com {xiaolm,kexie,bindong,ruanli,liudm}@buaa.edu.cn

Abstract. Client-side caching is an effective technique to hide network latency and improve I/O performance in network-based file systems. Current methods mainly adopt block-indexed caching structures, which suffer cache inefficiency problems in high concurrency environment. In this paper, we present a hybrid client-side caching scheme (HCCache) to avoid performance degradation caused by the block interleaving problem and increase the cache space efficiency by customizing content addressable levels for files with different sizes. Two new metrics are also proposed to accurately evaluate cache efficiency compared with the metrics of hit rate. Extensive simulations show the I/O performance with HCCache can be improved by 34.2 percent and 6.1 percent in average for read requests and 37.8 percent and 27.8 percent in average for write requests in terms of I/O bandwidth and access latency, respectively. Meanwhile, HCCache can significantly reduce the lookup times of content addressable data blocks and improve the access latency for small files.

Keywords: client-side caching, small files, network-based file system.

1. Introduction

Caching technique is one common approach to alleviate the access latency in many application scenarios, such as web servers [34], storage and file systems [25, 8, 21], databases [38], etc. The rationale is that frequently accessed items have higher probabilities to be served from cache with lower latency than normal accesses. In network-based file systems (such as Lustre [3], PVFS [4]), the functionalities of file system clients and servers are decoupled and interact data among each node over network. Meanwhile, metadata servers (MDSs) in many file systems are also decoupled from data servers (DSs) [40, 42]. Serving an I/O request requires interactions with multiple servers according to complex I/O protocols. Efficient design of client-side caching becomes crucial to hide the I/O latency of both network round-trips and disk accesses.

Nowadays client-side caching is facing more concurrency due to the advent of both hardware and software technologies. Network-based file systems can be shared by thousands of concurrent clients, especially in high performance computing (HPC) facilities

[17]. Moreover, multicore/manycore processors with hundreds of cores are readily available for computing in near future [30]. Hence, even single file system client can experience high I/O concurrency. To benefit from parallelism, multi-thread/process applications programmed by parallel languages or libraries(such as pthread, MPI/MPI-IO [8], OpenMP [27]) become mainstream in many fields, such as scientific computing, image processing, etc. Many of these applications generates concurrent, burst I/O requests with small inter-arrival times shorter than a millisecond [11, 37].

Designing efficient client-side caching is facing a lot of challenges in high concurrency environment. One reason is that I/O workloads exhibit various workload characteristics [9, 10, 13, 25]. For example, previous studies [9, 15] show that many workloads contain large number of small files with an average size under 1 MB or less. Typical small files include tiny web pictures, text documents and output files generated from scientific experiments [9, 32]. Meanwhile, many of them are immutable and dominated with read-exclusive accesses [18]. Hence, the cache performance is critical to serve requests on small files with low latency and high bandwidth. However, current design of client-side caching often employs block-indexed structures to manage cached data. While it is simple and effective for caching data for large files, small files will suffer poor performance in high concurrency environment. With block-indexed structures, the small file data has a high probability of interleaving by the blocks from other files in the *least recently used* (LRU) list. Once a block is evicted, following requests will suffer expensive accesses over the network.

In this paper, we present a hybrid client-side caching scheme (HCCache) to address above issues. HCCache combines the merits of object-indexed and block-indexed structures to distinguish the caching schemes for small and large files. The small file data with HCCache is managed in the LRU list as a whole. Therefore, the probability of partially hit in cache for small files can be significantly reduced. Furthermore, HCCache also can customize the granularity of content addressable cache data to trade off the memory savings and cache performance. Our main contribution can be summarized as follows.

– HCCache allows customizing the per-file caching behaviors (e.g., selecting cache block sizes, operating blocks in LRU list, selecting data compression granularities). Hence, the performance of small files can be optimized without interfering the caching behaviors of large files.
– We analyze the deficiencies of *hit rates* metrics in evaluating cache efficiency for I/O intensive workloads and propose two new metrics to correctly reflect the cache efficiency.
– Extensive simulation results using real-world application traces demonstrate the efficiency of HCCache in aspects of I/O bandwidth and access latency.

The rest of paper starts with the problem statement of our paper in Section 2. The design of HCCache is given in Section 3. Section 4 discusses the evaluation metrics of cache efficiency. Section 5 introduces the implementation of HCCache and the simulation methodology of evaluating our method, and the evaluation results are analyzed in Section 6. Section 7 summarizes the related work, followed by the conclusion and future work in Section 8.

2. Problem Statement

In this section, we introduce the performance problems of caching small files with block-indexed structures in detail by motivating simulation results.

2.1. Cache Block Interleaving Problem of Small Files

To guarantee data consistency and fairness, operating the data in a cache shared by multiple clients requires locks to proceed concurrent requests sequentially. When the concurrent requests are burst, however, the cache blocks of the same file will be interleaved with other blocks in the LRU list. Fig.1 shows an example of the block interleaving phenomenon. Two reasons make this phenomenon to become a common case. The first one lies in the reduction of I/O access size by the file system client. Although applications can specify a large I/O access size with the I/O system call, the file system will split the I/O request into multiple requests with a predefined transfer size. For example, the default transfer size of NFS is 4KB and can be configured up to 32KB [5]. Another reason is that setting too large cache block sizes will introduce *false sharing* problem which significantly degrades performance for concurrent writers [29].

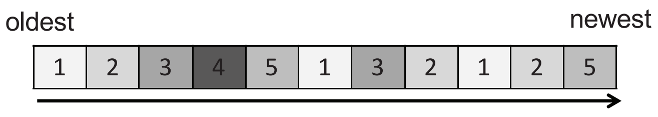


Fig. 1. Example of Cache Block Interleaving phenomenon. The blocks in the LRU list belong to five files. File 1 has 3 blocks in total and interleaved by 6 blocks of other files

To analyze the effect of concurrency on the cache block distribution of small files, we motivate the concept of *average block distance* (ABD), which is defined as dividing the total number of cache blocks between the first and the last block belongs to the file in the LRU list by the total number of cache blocks of the file. Suppose a file ***F*** has *n* data blocks with the storing order in the LRU list. The index number of the *i*th block can be coined as D_i.

Definition 1 (ABD(average block distance)). *The ABD of a file **F** in the client-side cache is defined as:*

$$ABD = (D_n - D_1)/(n-1) \tag{1}$$

, where D_1 and D_n are the index numbers of the first and last data blocks in LRU list, respectively.

We conducted simulations using a real-world trace *httpd* to measure the ABDs of small files in the client-side cache. Fig. 2 shows the results of replaying traces with varying the number of cores, which are indicated by the prefix in the legends, e.g., '256C' means 256 cores configured for the simulations. We can observe that the blocks of small files are interleaved more seriously with large number of concurrent clients. For example, the ABDs with over 256 clients are around 60 and 90 blocks in the LRU list. Therefore, the data of small files have higher probabilities to be partially cached in memory due to block evictions.

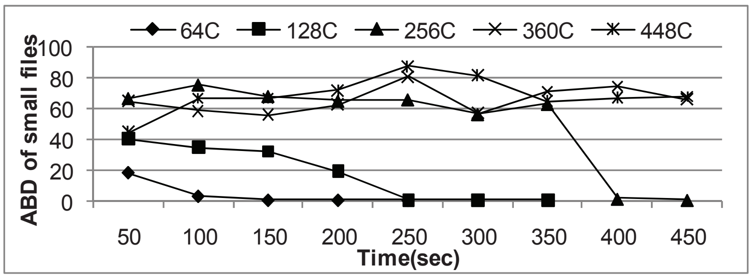


Fig. 2. ABDs of small files in *httpd* traces with varying client scale

2.2. Performance of Partially Cached Small Files

To analyze the effect of block interleaving problem on small files, we measured the read I/O latencies with various transfer sizes. Fig. 3 shows the latency results of performing read requests with the specific size using 100 clients. We can observe that the read latency numbers with the transfer sizes below 64KB are comparable. The performance degradation of partially cached small files is introduced by two reasons. On one hand, the cached blocks of small files bring little performance benefits. Suppose the cache block size is 32KB and the second block of a 33KB file missing in the cache. Although only missing 1KB data in cache, the total latency of reading the file is still about 4 to 5 ms, which is almost the same to the one without cache. Therefore, missing part of blocks of small files in the cache is more expensive than large files. Even worse, on the other hand, the partially cached data is useless to reduce latency and totally wastes the memory, making the cache performance can not be fully exploited.

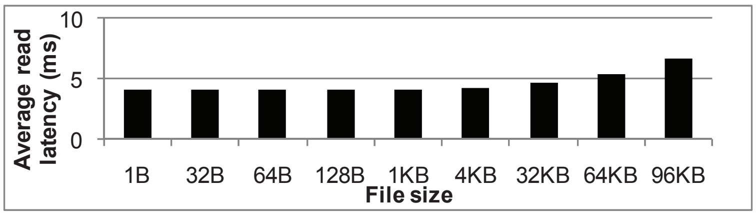


Fig. 3. Average read latency of small files with varying transfer sizes

2.3. Design Goals

To address the above cache performance problems, we aim to design a hybrid client-side caching scheme for I/O intensive workloads and achieve the following design goals: (1) Performance: to improve the aggregate bandwidth and reduce the average latency for small files, (2) Efficiency: to improve the cache space utilization and avoid unnecessary performance overhead incurred by data management techniques, (3) Scalability: to scale caching performance for small files in large scale environment.

3. HCCache Design

In this section, we present the design of proposed HCCache by introducing the architecture overview of our approach and discussing the design details.

3.1. Architecture overview

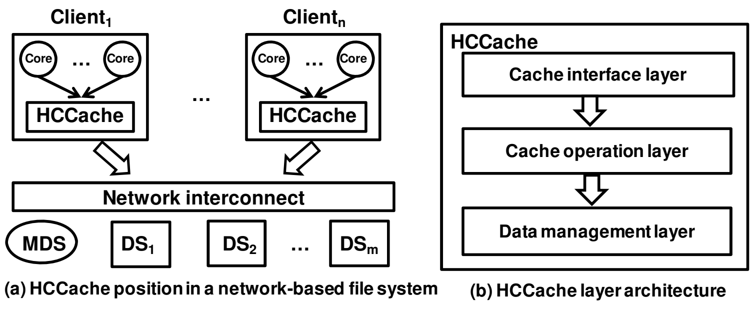


Fig. 4. HCCache architecture in a network-based file system

HCCache is placed on each client to serve I/O requests from memory for frequently accessed files. Requests missed in cache will interact with remote MDSs and DSs over the network. Fig. 4(a) shows the position of HCCache in the network-based file system architecture. All the cores on a client currently share the same HCCache daemon to access data. However, the design can be easily extended to the collective cache, which is shared by multiple clients.

Fig. 4(b) shows the three layers of HCCache to efficiently serve the I/O requests. The functionality of each layer is introduced as follows.

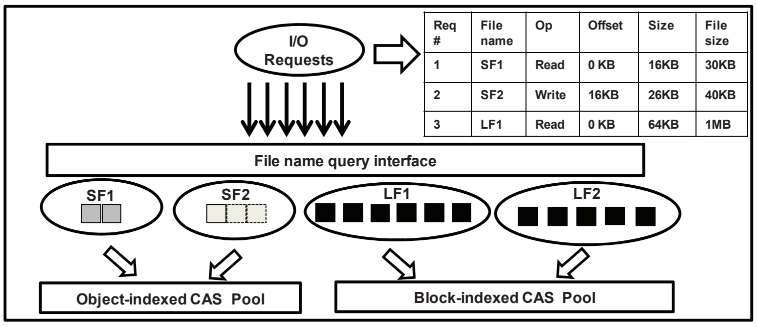
- *The cache interface layer* translates the I/O requests and chooses proper cache operations to serve the requests for small and large files.
- *The cache operations layer* is responsible for managing the cache operations, including data insertion, lookup and eviction.
- *The data management layer* explores the space efficiency by adopting content addressable storage (CAS) method to avoid storing duplicate blocks. The chosen CAS level and granularity is determined based on the file size.

3.2. Cache Interface and Operations

In this section, we present the design details of the cache interface and operations in HCCache.

Cache Structure As shown in Fig. 5, HCCache adopts a three level cache structure to operate file data. Similar with block-indexed structures, HCCache requires serval parameters with I/O requests to operate cache items, such as file name, operation type, access offset and size. The difference is that HCCache needs one more parameter to get the file size of requested file. This information is readily available in practical since the file metadata needs to be retrieved before issuing I/O requests [7, 28]. HCCache relies on a file size threshold to determine whether it is a small file. The file size threshold for small files can be tunable according to workloads characteristics. We chose 256KB as the small file threshold, which worked fine for our simulations.

The data blocks in HCCache are logically arranged into groups indexed by corresponding file names. The top level of cache structure is responsible for indexing these groups and querying the group for specific I/O request. At the second level, the block organizations of each group are versatile for different kinds of files. All the blocks of a small file are treated as a whole in the LRU list when performing cache operations, while the blocks of a large file are processed as individual units to in the LRU list. HCCache maintains the data in different CAS pools at the lowest level to trade off space savings and performance.



Req #	File name	Op	Offset	Size	File size
1	SF1	Read	0 KB	16KB	30KB
2	SF2	Write	16KB	26KB	40KB
3	LF1	Read	0 KB	64KB	1MB

Fig. 5. An example with proposed three-level cache structure

Cache Insertion The data to be inserted may be only part of the small file due to splitting requests or the setting of cache block size smaller than the transfer size. To increase space efficiency, the cache space of a small file is dynamically allocated rather than pre-allocated while inserting data block for the file at the first time. As shown in Fig. 5, although the file size of the file named SF2 is 40KB, the size of cache space for the file is only 16KB before receiving the second request in the table. The blocks with *dashed lines* are not available and do not occupy any space. Only when processing the second request, the occupied space is enlarged to 40KB in the example. The cache insertion algorithm is described in Algorithm 1.

Cache Lookup Looking up data in HCCache requires firstly locating the data group with the requested file name and then finding the blocks with requested offset and I/O

Algorithm 1 Cache insertion algorithm in HCCache

```
Insert $data\_block(F_i, j)$ to LRU list $LRUList$
 1: if $F_i$ is a small file then
 2:     if there is no $DGroup(F_i)$ then
 3:         create data group $DGroup(F_i)$;
 4:         add $data\_block(F_i, j)$ to $DGroup(F_i)$;
 5:         put $DGroup(F_i)$ as the last element in $LRUList$;
 6:     else
 7:         add or update $data\_block(F_i, j)$ to $DGroup(F_i)$;
 8:         put $DGroup(F_i)$ as the last element in $LRUList$;
 9: else
10:     if there is no $DGroup(F_i)$ then
11:         create data group $DGroup(F_i)$;
12:     add $data\_block(F_i, j)$ to $DGroup(F_i)$
13:     put $data\_block(F_i, j)$ as the last element in $LRUList$;
```

size. If only part of requested data hit in the cache, the left region should be requested from remote servers. This procedure is the same for looking up the data of both small and large files. However, the difference is that the whole data group of requested small file is moved to the tail of the LRU list. This way avoids possible block interleaving problems induced by accessing part of file blocks. Algorithm 2 shows the details of cache lookup operation.

Algorithm 2 Cache lookup algorithm in HCCache

```
Lookup $data\_block(F_i, j)$ from LRU list $LRUList$
 1: if $F_i$ is a small file then
 2:     if there is no data group of $F_i$ in $LRUList$ then
 3:         return $CACHE\_MISS$;
 4:     else
 5:         if there is no $data\_block(F_i, j)$ in $DGroup(F_i)$ then
 6:             return $CACHE\_MISS$;
 7:         else
 8:             put $DGroup(F_i)$ as the last element in $LRUList$;
 9:             return $data\_block(F_i, j)$;
10: else
11:     if there is no $data\_block(F_i, j)$ in $LRUList$ then
12:         return $CACHE\_MISS$;
13:     else
14:         put $data\_block(F_i, j)$ as the last element in $LRUList$;
15:         return $data\_block(F_i, j)$;
```

Cache Eviction Evicting a cache item in HCCache is simple and straightforward as shown in Algorithm 3. As the locations of small files data are updated once accessing the file, all the small file data will be erased. Therefore, the cache space occupied by partially cached small files can be reused quickly.

Algorithm 3 Cache eviction algorithm in HCCache

```
Evict $data\_block(F_i, j)$ from LRU list $LRUList$
1:  if $F_i$ is a small file then
2:    if there is no data group of $F_i$ in $LRUList$ then
3:      return $NON\_EXIST$;
4:    else
5:      if there is no $data\_block(F_i, j)$ in $DGroup(F_i)$ then
6:        return $NON\_EXIST$;
7:      else
8:        evict $data\_block(F_i, j)$ from $DGroup(F_i)$;
9:        if there is no $data\_block(F_i, j)$ in $LRUList$ then
10:         remove $DGroup(F_i)$ from $LRUList$;
11:       return $SUCCESS$;
12: else
13:   if there is no $data\_block(F_i, j)$ in $LRUList$ then
14:     return $NON\_EXIST$;
15:   else
16:     evict $data\_block(F_i, j)$ from $LRUList$;
17:     if there is no $data\_block(F_i, j)$ in $LRUList$ then
18:       remove $DGroup(F_i)$ from $LRUList$;
19:     return $SUCCESS$;
```

Cache Operations Runtime Optimization HCCache introduces additional layer to manage data blocks of the same file in a data group. Hence, there is potential impact on the performance of cache operations. The extra steps of operating data groups are as follows.

– *isExist(file_name)*: Query the existence of data group with file name.
– *getDGroup(file_name)*: Retrieve the reference of data group with file name.
– *createDGroup(file_name)*: Create the data group for specific file.
– *add/delBlock2DGroup(data_block,DGroup)*: Add or remove data block from the data group corresponding to its file name.

The runtime of first two operations is both related to the total number of active files in cache. To reduce the overhead of traversing all data groups, we exploit efficient data structures to operate data groups with low overhead. We maintain a *bloom filter* [24, 33] to represent the existence of data group by hashing file names. Queries with *false* results from the structure will be immediately returned as no such data group exists. In contrast, the *true* results are not represented as existence except the data group with the hashed file name found in the list. The query latency and extra space allocation can be traded off to optimize the runtime. The last two operations only need one linked list operation and thus contribute little overhead to cache operations.

3.3. Cache Data Management

Despite the amount of total memory on commodity servers is continued increasing, most of memory space is consumed by applications and the one left for file system caching on client-side is limited [9]. CAS is an effective method to improve the space efficiency for data storage or caching and becomes popular during recent years [14, 16, 26]. The basic idea is that the data is split into small chunks and chunks with duplicate content are only saved one copy. The similarity detection of duplicate chunks is performed using hashing techniques, such as MD5, SHA1. While significant space savings can be achieved, CAS introduces the hash generation cost and the time of looking up chunks.

HCCache takes considerations of workload characteristics into its data management design using CAS methods. Specifically, we choose different CAS levels and granularities for small and large files. As shown in Fig. 5, the small and large file data is stored in the object-indexed and block-indexed CAS pools, respectively. The reasons for this design can be two folds. On one hand, previous studies show that the total data amount of small I/O accesses only accounts for 10 percent of storage space in many applications, although the number of small I/O accesses makes up 90 percent of total accesses[37]. Meanwhile, the access latency is very important for workloads with many small files. For example, the results files of physical experiments or log files recording calls in a telephone company both require to be generated in burst. One the other hand, the data similarity between chunks of different small files is small in many workloads. Take the web workloads contain large number of tiny pictures as an example, these image files are generated and compressed as high-density binary files. The left improvement space of memory space savings is rather small and one practical choice of saving space is by finding completely duplicate images.

4. Metrics of Cache Efficiency

Currently, hit rate is the de-facto metrics to evaluate the cache efficiency [31, 41]. However, this metrics cannot always reflect the real cache efficiency on improving I/O latency and bandwidth. Fig. 6 shows an example of cache block lists with a block-indexed cache (baseline) and the HCCache. There are three files sharing the cache and file A and B both have one data block missed in the baseline cache, while file B completely misses in HCCache. Suppose the cache size is 6 blocks and each block size is 16KB, two I/O requests are issued to request the whole data of file A and B. The hit rate of the baseline cache is 66.7 percent (4 hits in cache and 6 lookup operations), while the one of HCCache is 50 percent (3 hits in cache and 6 lookup operations). However, as both two I/O requests are partially hit in the baseline cache, the file system client with the baseline cache requires to issue two accesses to request data from remote servers. For the case with HCCache, the request on file B completely misses in the cache and needs to retrieve the whole file from server. Although the I/O sizes of two remote accesses with baseline cache are only one cache block size of 16KB, the access latency is similar with the one of accessing the whole file with a file size of 48KB according to the results shown in Section 2. Therefore, the total access latency of the two requests with the baseline cache is appropriate two times larger than the one with HCCache.

The main reason of incorrectly evaluating cache efficiency is that hit rate metrics doesn't take account of the real effect of cached data on I/O performance. We present two

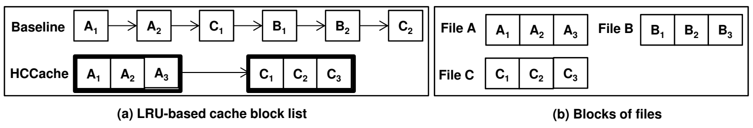


Fig. 6. An example of block lists with different cache schemes

new metrics to evaluate the cache efficiency with HCCache scheme. The definitions of proposed metrics are given as follows.

Definition 2 (ECDAL(Effect of Cache Data Amount on Latency)). *The ECDAL of a client-side cache during a period of time is defined as:*

$$ECDAL = 1/(\sum_{i=1}^{n} L_i * \sum_{j=1}^{m} CDA_j) \tag{2}$$

, where L_i is the latency of completing the ith I/O request, and CDA_j is the data amount of jth cache insert operation.

Definition 3 (ECDAB(Effect of Cache Data Amount on Bandwidth)). *The ECDAB of a client-side cache during a period of time is defined as:*

$$ECDAB = \sum_{i=1}^{n} DA_i/(\max_{k=1}(CT_k) * \sum_{j=1}^{m} CDA_j) \tag{3}$$

, where DA_i is the data amount of the ith I/O request and CT_k is the total I/O time on the kth client.

These evaluation metrics establishes the relations between I/O workloads and cache performance. The effect of the cached data amount on the I/O bandwidth and latency of I/O requests can be quantified. The larger value of these metrics, the higher cache efficiency can be achieved. For instance, the HCCache in our example holds smaller I/O request latency and the same cached data amount compared with the case with the baseline cache. Hence, the proposed metrics can correctly evaluate the cache efficiency as the ECDAL of HCCache is better than the one of baseline cache.

5. Implementation and Simulation Methodology

This section describes the implementation of HCCache on the simulator and the workloads used to evaluate the effectiveness of our method.

5.1. Simulation framework

We extended a file system simulator used in our previous study [20] to evaluate the HC-Cache behaviors on large scale file systems within a short period of time. The simulator is originated from an open-source parallel file system simulator PFSsim [23], which uses

DiskSim [2] to simulate accurate disk response time. As shown in Fig.7(a), there are four kinds of nodes in the simulator, including the client, the data server, the metadata server and the router. These nodes are configured to be connected via networks with specific latency and bandwidth. The simulator is driven by trace files to simulate the process of serving I/O requests and the results are computed and reported by the clients. We made serval modifications on the simulator to support our simulation requirements.

– **Multicore support.** We support simulating multicore architecture on file system clients. The implementation is mainly motivated from HECIOS simulator [22, 29]. As shown in Fig.7(b), each *clientcore* module can replay its own trace file and the requests are forwarded to the *mclient* module, which schedules the requests and sends to the server node via the router node.
– **Various trace formats support.** PFSsim only supports a self-defined trace format, which is not convenient for traces from various sources, such as MAMBO Suite [6], LANL MPI Trace [1] and HECIOS format [29], etc. We extended the input interface of trace files and allowed to configure different traces running on given range of client and server nodes.
– **Better integration of DiskSim.** Accurate disk response time is vital to guarantee the correctness of simulating I/O behaviors. PFSsim relies on TCP connections to communicate between DiskSim and PFSsim. The simulation speed is too slow to make the simulation time intolerable for I/O-intensive workloads. We integrated DiskSim with the simulator more tightly by statically linking DiskSim to the simulator during compilation. The improvement can significantly accelerate simulations while maintaining correctness.

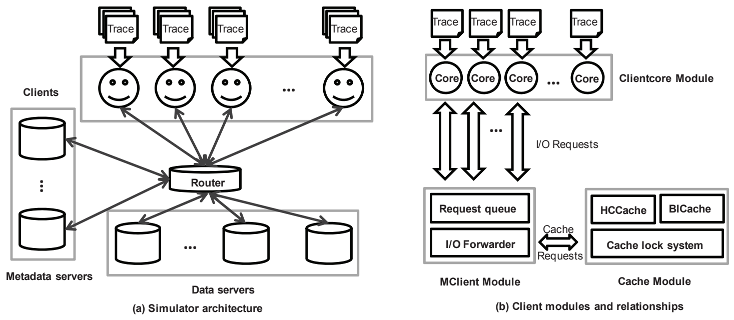


Fig. 7. Simulator architecture overview and its client-side main modules

5.2. HCCache Implementation

The *cache* module on a client in the simulator is shared by multiple cores. The *read* requests issued by *clientcore* module are firstly forwarded to the *cache* module. If the requested data is available, the requests will be directly served. Otherwise, the requests are routed to server nodes along with normal I/O paths. When the client receives the requested data from server nodes, the response message will be forwarded to *cache* module again to update cache status. Finally, the message is forwarded to the *clientcore* module.

The HCCache scheme is implemented in the *cache* module. The most important part of simulating the cache block interleaving phenomenon is the lock mechanisms to generate block insertion sequences for different files. As the simulator is driven by discrete events, each request needs to update the global time and schedule the self event of the *cache* module to proceed a pending request. After a request got the lock, the request will be proceeded using cache operations in HCCache based on the request type. During this period, all received requests are pending in a queue to wait for owning the lock.

The implementation of cache scheme is modular and easy to add new schemes to substitute HCCache scheme. To evaluate the performance benefits, we also implemented a block-indexed cache in our simulator. By default, the simulator was configured with 32 DSs and 1 MDS, which connected with a 1 Gbps switch among each server. The cache on each client node was configured with the cache size of 32 MB and the cache block size of 32 KB.

5.3. I/O Workloads

To evaluate the efficiency of HCCache under I/O intensive workloads, we chose the *httpd* trace [6] which was collected from NASA Kennedy Space Center's web server to perform simulations. After removing the uncorrect records in the traces, the trace files used in our simulation record over 21.5GB read accesses on 10,132 files, which consist of over 9,000 files smaller than 256KB. Table 1 shows the statistics of I/O requests in *httpd* trace. The original traces were stored in 7 files. To simulate future I/O systems with hundreds of cores, we used the similar scale-up approaches in [42] to split the original trace files into 448 subtraces and concurrently replayed them using up to 448 cores.

Table 1. Workload statistics of *httpd* traces

	Small files			Large files		
Trace	File number	Data amount	I/O amount	File number	Data amount	I/O amount
httpd	10118	218.84MB	18.92GB	14	21.8MB	2.56GB

To evaluate the write performance of HCCache, we simulated write-intensive applications with large amount of small and large file accesses. We first generated a synthetic trace to create concurrent requests of writing small files with the request size of 4 KB. The file sizes were varied from 4 KB to 256 KB evenly. Then we simulated the write requests of large files using FLASH I/O traces [29].

6. Performance Evaluation

We evaluated our design by running simulations on a server with AMD Quad-core processor, 8GB RAM, and 1TB Seagate 7200RPM hard drive. This section presents the simulation results in terms of cache performance, efficiency of cache data management and performance sensitivity. We focus on analyzing the benefits of HCCache over the block-indexed cache (BICache).

6.1. Evaluation of cache performance

We present and discuss the cache performance results in aspects of the aggregate bandwidth, the average access latency and the metrics of cache efficiency.

Fig. 8 shows the comparisons of read performance with BICache and HCCache for *httpd* trace. The performance numbers of large files and small files are labeled with “L” and “S”, respectively. The simulations varied the concurrency degrees by replaying different numbers of subtraces. We can observe that HCCache performs better than BICache under all simulation settings. The average improvements of bandwidth and latency for small files are 34.2 and 6.1 percent, respectively. The reason is that the data of a small file with HCCache are operated as a whole, avoiding performance degradation caused by partially hit in BICache. To be noticed, the improvement ratios with different number of cores are not consistent or linearly increasing. The amount of trace records and contents replayed each time are not equal and identical due to workload scale-up reasons. For example, in the case with 256 cores, the tests only use 256 files out of 448 trace files of *httpd* to reduce the simulation time. Large file performance can also be improved by about 3 to 5 percent with HCCache. The reason is the cache space saved by avoiding partially caching small files can cache more large file blocks. Another observation is that the bandwidth numbers are decreasing when the number of clients increases over 256 for both two cache schemes. This is because the system becomes saturated due to high concurrency. However, even in those cases, HCCache is still superior to BICache due to higher efficiency of space utilization.

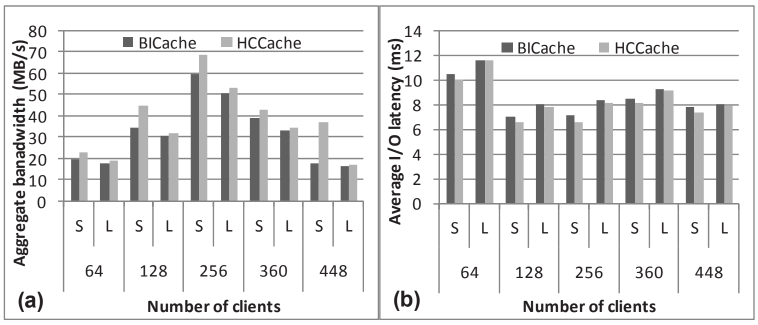


Fig. 8. Read performance for *httpd* trace:(a) aggregate bandwidth,(b) access latency

Fig. 9 shows the comparisons of write performance with BICache and HCCache for the synthetic trace. We configured 256 processes to run the synthetic trace and varied the numbers of concurrent clients running FLASH I/O traces to generate write requests of large files. Compared with BICache performance, we can observe the bandwidth and access latency with HCCache can be improved by 37.8 percent and 27.8 percent in average, respectively. The reason is that HCCache will buffer the small writes in cache and only commit data to disk when the cache block is evicted. The whole small file data is committed using one disk request, while BICache may require more disk requests depending on its data distribution in the evicted cache blocks. The performance impact of HCCache on large files of FLASH I/O traces is not shown in the figure as it is similar to the one we observed for *httpd* trace.

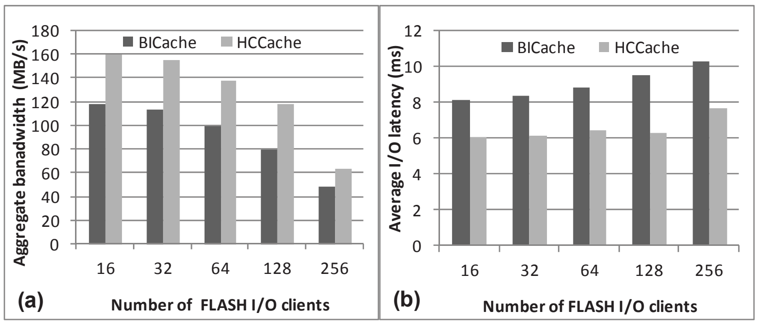


Fig. 9. Write performance for synthetic trace:(a) aggregate bandwidth,(b) access latency

Fig. 10 shows the comparisons of the results of three cache efficiency metrics under above simulations. We can observe that the hit rates with BICache are a little higher than the ones of HCCache. However, the performance results shown in Fig. 8 do not exhibit expected improvements. Instead, both I/O bandwidth and access latency suffers performance problems. Therefore, the metrics of hit rate fails to evaluate the cache efficiency correctly. In contrast, the results of proposed metrics ECDAL and ECDAB can show the effect of cache efficiency on I/O performance correctly.

6.2. Efficiency of Cache Data Management

HCCache adopts CAS method to improve the memory utilization of cache space. However, selecting proper CAS level and granularity is critical to control the overheads inherited to the CAS method. Table 2 summarizes the results of CAS block lookup times and impacts on I/O latency for small files. The baseline results are simulated on HCCache with only a block-indexed CAS pool for both small and large files. We can observe that the lookup times of optimized cases can reduce over 96 percent compared with the baseline. The reason is that the optimized HCCache maintains CAS data at an object granularity and each file request only requires one query operation. The reduction of lookup times also induces lower latency for reading small files due to the same reason.

6.3. Sensitivity Study

Cache size is an important factor potentially influencing the I/O performance. Fig. 11 shows the small file performance with different cache block sizes using 256 clients. In theory, the block interleaving problem will be alleviated with a larger number of cache block size. However, the results show HCCache can achieve better performance for cache

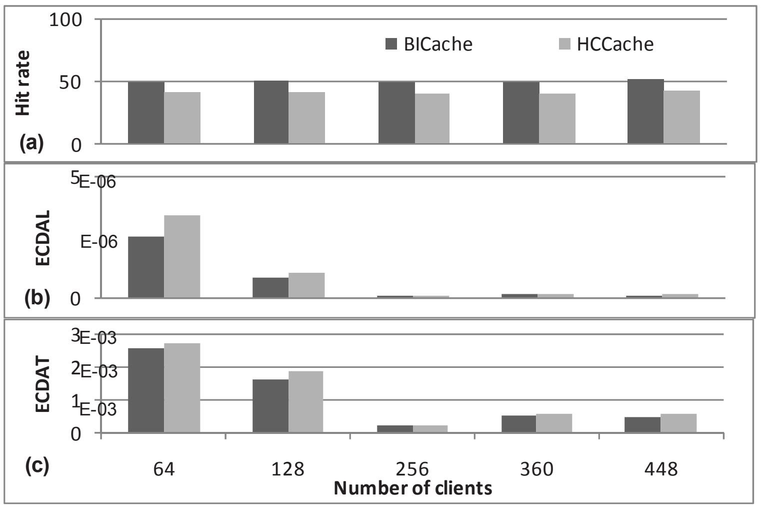


Fig. 10. Results of cache efficiency metrics:(a) Hit rate, (b) ECDAL, (c) ECDAB

Table 2. Comparisons of CAS efficiency of small files in *httpd* trace

# clients	64	128	256	360	448
Baseline lookup times	946592	2075072	3865472	5252608	5952736
Optimized lookup times	29581	64846	112736	164144	186023
Latency improvement	2.47%	1.82%	4.1%	8.78%	5.42%

block sizes smaller than 128KB. Although there is a little performance down for HCCache with the cache block size of 128KB, this block size is not suitable for practical use as it makes the effect of *false sharing* problem worse on cache performance.

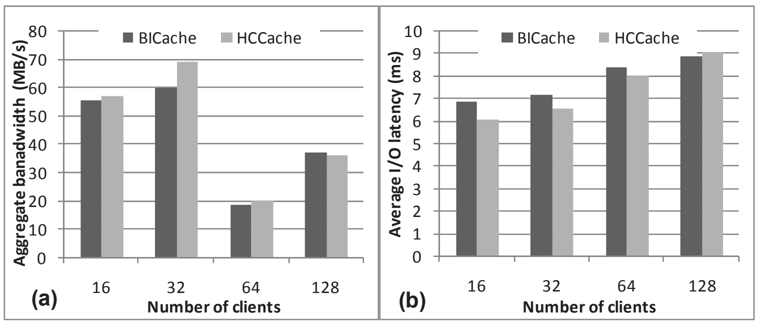


Fig. 11. I/O performance of small files with different cache block sizes: (a) aggregate bandwidth,(b) average latency

7. Related Work

Client-side caching is a common technique adopted by many network-based file systems to improve I/O performance. NFS4 provides fine-grained file region locks to support concurrent cache accesses on the same file. Lustre [3] also uses its distribute lock manager to guarantee cache consistency among concurrent clients. Besides, file systems, such as Ceph [39], PPFS [25], all support client-side caching. However, the existing client-side cache implementations are adopted the block-indexed structures [29]. None of them considers the caching performance of small files in the high concurrency environment.

Many studies have been focused on improving cache performance for I/O workloads. Settlemyer, et. al [29] studied client-side caching techniques to alleviate the effect of *false sharing* problem at costs of time and space. They also motivated a MPI File View combining for collective I/O operations to improve cache performance for MPI applications with large amount of small I/O calls. Vilayannur, et. al [35] proposed a discretionary caching scheme to allow applications configuring caching parameters at compilation stage. Madhyastha, et. al [25] presented two methods to learn access patterns of I/O accesses and utilized the results to make cache policy selection at runtime. Wachs, et. al [36] proposed a workload-aware cache partitioning method to allocate cache space size for concurrent applications on storage server. Frasca, et al. [12] proposed a virtual I/O caching framework to improve the utilizations of cache space by dynamically mapping logical cache pages of each application to physical cache blocks. The main idea is to assign most valued cache blocks higher priority in cache to achieve the same hit rates while consuming a smaller cache size. In contrast, the novelty of our study is to mitigate small files performance degradation induced by block interleaving problem in high concurrency environment. The requests on files with different sizes are operated with different granularities in the client-side cache. Therefore, HCCache scheme can enforce all the data blocks of a small file in the cache at the same time, avoiding the possible performance degradation caused by partially hit in cache. HCCache does not track access information for each cache page and can achieve performance benefits without additional overheads or memory consumptions. Furthermore, we also proposed new metrics to correctly reflect the read cache efficiency.

Many efforts [12, 19, 26] have been focused on reducing cache size without hurting cache performance. These work can be classified by either reducing the number of cache blocks or saving memory space according to the cache block contents. The former one mainly focuses on deciding the necessary blocks in cache and improving the cache space utilization. The latter one often employs compression methods or CAS methods to reduce the cache data size of cache blocks determined by the cache module. Therefore, these methods can further improve cache efficiency after adopting the former kind of methods. However, compression-based methods require large amount of CPU time to process data during I/O operations. In contrast, CAS-based methods are more attractive as they only need one more hash operation before storing data in cache. Nath et al. [26] observed significant savings in network bandwidth by adopting a content addressable cache. However, one shortcoming of CAS-based methods is that getting a cache block requires multiple lookup operations for corresponding CAS blocks. The HCCache scheme trades off the spacing savings and performance improvements by increasing the CAS granularity of small files.

8. Conclusion and Future Work

This paper proposed HCCache to improve cache efficiency by distinguishing caching schemes for small and large files. We studied the block interleaving problem and performance degradation for partially hit small files. Motivated by the problem, we present a three-level cache structure to serve I/O requests according to the characterizes of files with different size. The data of small files is managed in the LRU list and CAS pools at per-object basis. Two metrics of cache efficiency are also defined to correctly reflecting cache performance. The simulation results demonstrate that HCCache can improve the I/O performance of small files in terms of bandwidth and latency compared with the block-indexed cache. In future, we plan to further study HCCache behaviors in the collective cache which shared by clients from multiple nodes.

Acknowledgments. This work is supported by the National Natural Science Foundation of China under Grant No.61370059, 61232009, the Doctoral Fund of Ministry of Education of China under Grant No. 20101102110018, Beijing Natural Science Foundation under Grant No. 4122042 and the fund of the State Key Laboratory of Software Development Environment under Grant No. SKLSDE-2012ZX-07.

References

1. LANL Trace, http://institute.lanl.gov/data/software/.
2. The DiskSim Simulation Environment (v4.0), http://www.pdl.cmu.edu/DiskSim
3. Lustre File System (2010), http://www.lustre.org
4. The Parallel Virtual File System (2010), http://www.pvfs.org
5. Linux NFS-HOWTO (2012), http://tldp.org/HOWTO/NFS-HOWTO/
6. MAMBO I/O Trace Suite (2012), http://www.cs.umd.edu/projects/hpsl/mambo/index.html
7. Ananth, D., Pete, W.: File Creation Strategies in a Distributed Metadata File System. In: IEEE IPDPS. pp. 1–10. IEEE Computer Society (2007)
8. Byna, S., Chen, Y., Sun, X.H., et. al: Parallel I/O prefetching using MPI file caching and I/O signatures. In: ACM/IEEE SC'08. pp. 1–12. Piscataway, NJ, USA (2008)
9. Carns, P., Lang, S., Ross, R., Vilayannur, M., Kunkel, J., Ludwig, T.: Small-file access in parallel file systems. In: IEEE IPDPS. pp. 1–11. Rome, Italy (2009)
10. Chen, J., Roth, P.C., Chen, Y.: Using Pattern-Models to Guide SSD Deployment for Big Data in HPC systems. In: BigData 2013 (2013)
11. Chen, Y., Sun, X.H., Thakur, R., Roth, P.C., Gropp, W.D.: LACIO: A New Collective I/O Strategy for Parallel I/O Systems. In: Proceedings of the 2011 IEEE International Parallel & Distributed Processing Symposium. pp. 794–804. IPDPS '11, IEEE Computer Society, Washington, DC, USA (2011)
12. Frasca, M., Prabhakar, R., Raghavan, P., Kandemir, M.: Virtual i/o caching: dynamic storage cache management for concurrent workloads. In: Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis. pp. 38:1–38:11. SC '11, ACM, New York, NY, USA (2011)
13. He, J., Bent, J., Torres, A., Grider, G., Gibson, G., Maltzahn, C., Sun, X.H.: I/O acceleration with pattern detection. In: HPDC 2013. pp. 25–36. ACM, New York, NY, USA (2013)
14. Koller, R., Rangaswami, R.: I/o deduplication: Utilizing content similarity to improve i/o performance. Trans. Storage 6(3), 13:1–13:26 (2010)
15. Kuhn, M., Kunkel, J.M., Ludwig, T.: Dynamic file system semantics to enable metadata optimizations in PVFS. Concurr. Comput. : Pract. Exper. 21(14), 1775–1788 (September 2009)

16. Kulkarni, A., Manzanares, A., Ionkov, L., Lang, M., Lumsdaine, A.: The design and implementation of a multi-level content-addressable checkpoint file system. In: High Performance Computing (HiPC), 2012 19th International Conference on. pp. 1–10 (2012)
17. Lang, S., Carns, P., Latham, R., Ross, R., Harms, K., Allcock, W.: I/O performance challenges at leadership scale. In: ACM/IEEE SC. pp. 1–12. Portland, Oregon, USA (2009)
18. Lensing, P., Meister, D., Brinkmann, A.: hashFS: Applying Hashing to Optimize File Systems for Small File Reads. In: Proceedings of SNAPI 2010. pp. 33–42. IEEE Computer Society (2010)
19. Li, M., Varki, E., Bhatia, S., Merchant, A.: Tap: table-based prefetching for storage caches. In: Proceedings of the 6th USENIX Conference on File and Storage Technologies. pp. 6:1–6:16. FAST'08, USENIX Association, Berkeley, CA, USA (2008)
20. Li, X., Dong, B., Xiao, L., et. al: Cefls: A cost-effective file lookup service in a distributed metadata file system. In: CCGrid'12. pp. 25–32. IEEE Computer Society (2012)
21. Liao, W.k., Coloma, K., Choudhary, A., Ward, L., Russell, E., Tideman, S.: Collective caching: application-aware client-side file caching. In: HPDC '05. pp. 81–90. IEEE Computer Society, Washington, DC, USA (2005)
22. Limin, X., Ke, X., Guoying, L., Li, R., Xiuqiao, L.: QoSFM: QoS Support for Metadata I/O In Parallel File Systems. In: The 15th IEEE International Conference on High Performance Computing and Communications (HPCC 2013) (2013)
23. Liu, Y., Figueiredo, R., Clavijo, D., Xu, Y., Zhao, M.: Towards simulation of parallel file system scheduling algorithms with PFSsim. In: Proceedings of SNAPI 2011 (2011)
24. Lu, G., Nam, Y.J., Du, D.C.: Bloomstore: Bloom-filter based memory-efficient key-value store for indexing of data deduplication on flash. In: IEEE 28th Symposium on Mass Storage Systems and Technologies (MSST). pp. 1–11 (2012)
25. Madhyastha, T.M., Reed, D.A.: Learning to Classify Parallel Input/Output Access Patterns. IEEE Trans. Parallel Distrib. Syst. 13(8), 802–813 (August 2002)
26. Nath, P., Urgaonkar, B., Sivasubramaniam, A.: Evaluating the usefulness of content addressable storage for high-performance data intensive applications. In: Proceedings of the 17th international symposium on High performance distributed computing. pp. 35–44. HPDC '08, ACM, New York, NY, USA (2008)
27. Osthoff, C., Grunmann, P., Boito, F., et. al: Improving performance on atmospheric models through a hybrid openmp/mpi implementation. In: ISPA '11. pp. 69–74. IEEE Computer Society, Washington, DC, USA (2011)
28. Sadaf, R.A., Hussein, N.E.H., Kristopher, H., Neil, S., Fabio, V.: Parallel I/O and the Metadata Wall. In: 6th Parallel Data Storage Workshop, SC11 (2011)
29. Settlemyer, B.W.: A study of client-based caching for parallel I/O. Ph.D. thesis, ClemsonUniversity (2009)
30. Sun, X.H., Chen, Y.: Reevaluating amdahl's law in the multicore era. J. Parallel Distrib. Comput. 70(2), 183–188 (Feb 2010)
31. Sun, X.H., Wang, D.: Apc: A performance metric of memory systems. SIGMETRICS Perform. Eval. Rev. 40(2), 125–130 (Oct 2012)
32. Swapnil, P., Garth, G.: Scale and Concurrency of GIGA+: File System Directories with Millions of Files. In: FAST '11. San Jose, CA (2011)
33. Tarkoma, S., Rothenberg, C., Lagerspetz, E.: Theory and practice of bloom filters for distributed systems. IEEE Communications Surveys Tutorials 14(1), 131–155 (2012)
34. Vakali, A.: Evolutionary techniques for web caching. Distrib. Parallel Databases 11(1), 93–116 (Jan 2002)
35. Vilayannur, M., Sivasubramaniam, A., Kandemir, M., et. al: Discretionary caching for i/o on clusters. Cluster Computing 9(1), 29–44 (Jan 2006)
36. Wachs, M., Abd-El-Malek, M., et. al: Argon: performance insulation for shared storage servers. In: FAST '07. pp. 5–5. USENIX Association, Berkeley, CA, USA (2007)

37. Wang, F., Xin, Q., Hong, B., et. al: File system workload analysis for large scale scientific computing applications. pp. 139–152. MSST'04 (2004)
38. Wang, X., Malik, T., Burns, R., et. al: A workload-driven unit of cache replacement for mid-tier database caching. In: DASFAA'07. pp. 374–385. Springer-Verlag (2007)
39. Weil, S.A., Brandt, S.A., Miller, E.L., et. al: Ceph: a scalable, high-performance distributed file system. In: OSDI '06. pp. 307–320. USENIX Association (2006)
40. Weil, S.A., Pollack, K.T., Brandt, S.A., Miller, E.L.: Dynamic Metadata Management for Petabyte-Scale File Systems. In: Proceedings of the 2004 ACM/IEEE conference on Supercomputing. pp. 4–. IEEE Computer Society (2004)
41. Zhao, T., March, V., Dong, S., See, S.: Evaluation of a performance model of lustre file system. In: 2010 Fifth Annual ChinaGrid Conference. pp. 191–196 (2010)
42. Zhu, Y., Jiang, H., Wang, J., et. al: HBA: Distributed Metadata Management for Large Cluster-Based Storage Systems. IEEE Trans. Parallel Distrib. Syst. 19, 750–763 (2008)

Xiuqiao Li received his Ph.D. Degree in Computer Architecture at Beihang University, China in 2013. His research interests include high performance file systems, job schedulers and cloud storage.

Limin Xiao is a Professor, Senior membership of China Computer Federation. His main research areas are computer architecture, computer system software, high performance computing, visualization and cloud computing.

Ke Xie is a postgraduate student in Computer Architecture at Beihang University, China. His research mainly focuses on providing I/O QoS in parallel file systems for concurrent metadata-intensive applications.

Bin Dong received his Ph.D. Degree in Computer Architecture at Beihang University, China in 2013. His research interests include data layout optimizations, hybrid storage optimizations in parallel file systems.

Li Ruan is a Ph.D., Lecturer, Senior membership of China Computer Federation. Her main research areas are virtualization and cloud computing, computer system software, high performance computer.

Dongmei Liu received her M.S. Degree in Computer Applications at Beihang University, China in 2010. Her research interests focus on network simulations for distributed systems.

Received: January 27, 2013; Accepted: November 9, 2013.