Hindawi Publishing Corporation
Mathematical Problems in Engineering
Volume 2013, Article ID 867167, 13 pages
http://dx.doi.org/10.1155/2013/867167

Research Article

C-Aware: A Cache Management Algorithm Considering Cache Media Access Characteristic in Cloud Computing

Zhu Xudong,[1,2] Yin Yang,[2] Liu Zhenjun,[2] and Shao Fang[3]

[1] *School of Computer and Information Engineering, Zhejiang Gongshang University, Hangzhou 310018, China*
[2] *Institute of Computing Technology, Chinese Academy of Sciences, Beijing 100080, China*
[3] *Zhejiang University of Technology, Hangzhou 310014, China*

Correspondence should be addressed to Yin Yang; yinyang80@gmail.com

Received 10 August 2013; Accepted 5 September 2013

Academic Editor: Yoshinori Hayafuji

Data congestion and network delay are the important factors that affect performance of cloud computing systems. Using local disk of computing nodes as a cache can sometimes get better performance than accessing data through the network. This paper presents a storage cache placement algorithm—C-Aware, which traces history access information of cache and data source, adaptively decides whether to cache data according to cache media characteristic and current access environment, and achieves good performance under different workload on storage server. We implement this algorithm in both simulated and real environments. Our simulation results using OLTP and WebSearch traces demonstrate that C-Aware achieves better adaptability to the changes of server workload. Our benchmark results in real system show that, in the scenario where the size of local cache is half of data set, C-Aware gets nearly 80% improvement compared with traditional methods when the server is not busy and still presents comparable performance when there is high workload on server side.

1. Introduction

In cloud computing, data congestion and network delay are the important factors that affect performance of systems. Since the hosts for storing data and processing data are often different, the demand for data transmission between tasks is very huge. Zhou et al. [1] shows that the transfer time of data intensive applications accounts for a larger proportion of the overall running time. For example, the system data and user data of PaaS services are usually stored in a centralized way, so when the computing nodes run, they need to handle the data from the storage servers. Obviously, network communication delay becomes the bottleneck of computing performance. What is worse, when multiple tasks need to transmit data across a network at the same time, they would compete for bandwidth, further worsening network delay, and it also increases the complexity of the prediction performance of dynamic model's network congestion.

The traditional way to solve the above question is to schedule the tasks so as to reduce the data transmission. As for data-intensive workflow computing, the main idea is to allocate the calculation task to the nodes that are closer to the data source, thus to reduce the time of network communication and to avoid congestion. Some researches [2] make the dynamic data replication participate as task scheduling, seeking to find the optimal scheduling scheme. It can be proved that the method based on task scheduling is NP-hard problem, which usually uses greedy method or heuristic algorithm to obtain the approximate optimal solution. This kind of method usually applies to data processing and calculation in a short term. The long-term system and service deployment, such as Amazon, Openstack, Sond, is rarely movable once being deployed to computing nodes. Therefore, it is hard to be optimized in scheduling way. Also, because of the centralized storage of data, the computing tasks in different nodes are unable to avoid the storage server bandwidth congestion.

In view of the system and service deployment applications, our group proposes the cache hierarchy architecture based on cheap disk medium [3]. The traditional concept of

cache is to use high speed and relatively expensive, volatile data storage medium to hold hot spots on the upper deck of the access level, thus improving the efficiency of data access. However, it is difficult to meet the demand of data access in the cloud computing system. (1) Computing node uses memory as a cache; the capacity is several GB, but the current cloud computing system has a large data set, of which the volume is usually hundreds of TB even petabytes; also the active data set is greater than the cache capacity. (2) A lot of access to data on a regular basis. Take the system or service startup as an example, the computing node requires access to the storage server to get the startup data, and these data usually will not be used again in a long time. So it is difficult to improve the performance of data access by using the cache. The cheap-disk-based cache hierarchy architecture uses the low-cost and high-capacity disk as the cache media, and the standard block-level interface to provide transparent data caching service for the upper application in the I/O path. The caching system is characterized as low cost, high capacity, the persistent storage, and it can solve internetwork access problems in cloud computing, as well as make it easy to expand to the complex network environment. For example Sun NFS [4], the IBM AFS, Coda [5, 6], xFS, and CAPFS [7, 8] are on the client side using a local disk as a cache to improve system performance, availability, and reliability, so as to alleviate the pressure of the back-end server load and network bandwidth.

The disk-media-based cache architecture contains disk, network, and the cache with different access properties, which the traditional cache management strategy cannot do. For example, under high load condition, the system using large capacity disk medium at low-speed cache data can improve overall system performance, but in low load cases, the emergence of a high-performance network can direct access to the back-end network storage system, and it can obtain higher overall performance than local disk storage. Therefore, the cache effect is not only associated with application access mode, but also the local cache speed characteristic and the overall load of storage system. But most of the current cache management algorithm, such as LRU, ARC [9], 2Q [10], LIRS [11], and UBM [12], each consider different factors in the management, but they are all based on the hypothesis that the local cache performance is much higher than that of medium dielectric properties. It stores the hot data in the cache to pursue high cache hit ratio to improve the performance without considering the performance difference between accessing the data source directly and the cache. In D-Cache [3], it is found that these cache management methods cannot well adapt to the changing of the cache media and data load in some cases and even reduce the overall performance of the system in some cases. How to manage the cache and obtain the overall ideal performance according to the cache medium characteristics, the current load of storage systems and application access mode is an urgent problem.

This paper proposes a storage cache management algorithm called C-Aware as shown in Figure 1. It considers the speed of the cache media and network load conditions, analyzes the access cache and historical information of network data, and also predicts the future access to the cache and storage server performance according to the historical information. Additionally, it decides whether the current request of the data should be stored in the cache. When this algorithm is applied to computing nodes' caches, under the condition of high load, it can use local cheap low-speed cache media to cache data and improve the overall performance of the system; otherwise, it directly accesses the network storage systems to reduce the performance loss caused by access of low-speed media. Through the benchmark test and load trace simulation, the results show that the C-Aware algorithm, compared with other cache algorithms, is able to obtain a better overall performance and better adaptability under different speed of the cache media and storage server load cases.

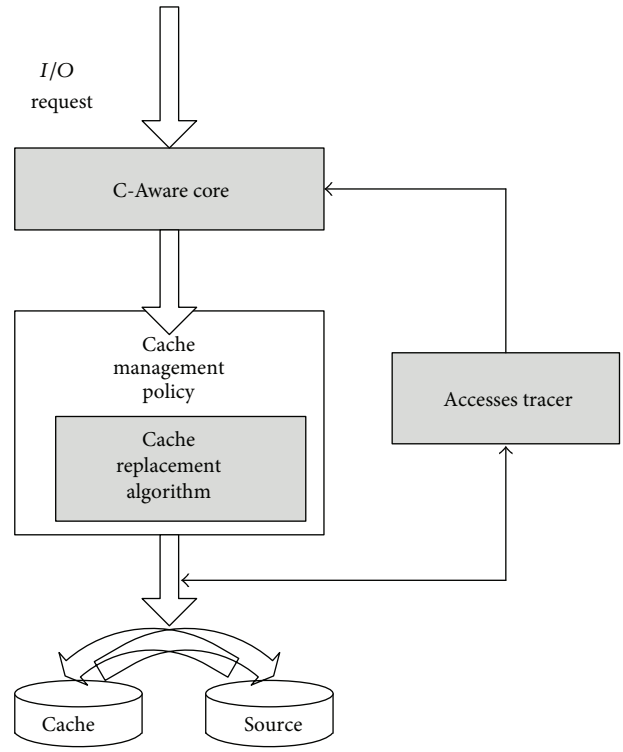


Figure 1: C-Aware framework.

Section 2 briefly introduces the related research on current cache management; Section 3 introduces the basic idea, specific design and implementation of C-Aware algorithm; Section 4 presents the trace simulation test and benchmark assessment results and analysis; the last section summarizes the full text and puts forward the existing problems and the suggestions for future work.

2. Related Work

Cache management mainly consists of two parts [13]: cache replacement strategy and place strategy. The former decides to choose a block of data cache as new cache space when

the old one is full; the latter determines the timing of the block data being stored in the cache. Because the current cache system design is based on the assumption that the performance of accessing the cache media is higher than that of direct access to the data source, the majority of cache management methods use cache placement strategy. That means when accessing the data block, it saves data blocks in the cache, and hot data must be kept in the cache as much as possible. So the current cache management algorithm research mostly focuses on the cache replacement problems.

The current cache replacement algorithm develops from LRU and LFU single strategy to the one that can be adaptive to application access patterns, such as the ARC [9], UBM [12], MQ [1], DULO [14], and so forth. With the development and wide use of network technology and storage system, research on the cache algorithm changes from the single stage to multistage multilevel cache coordination management. According to the cache management strategy, its algorithm can be divided into simple single management strategy, the access-frequency-and-access-time-based balance strategy, application-based strategy, detection-based caching strategies, and so forth. Simple single cache management strategy generally uses a single fixed standard to do replace management, such as LRU, MRU, LFU, and so on. They replace either based on cache block that has recently been used or on the number of cache block that has been accessed. These methods are ancient and simple, but particularly effective for a specific application. And they are simple in the system design and implementation, thus being the most used method at present.

The strategy of access frequency and access time balance is adjusted mainly based on the access frequency and a recent visit time, such as LRFU [15], ARC, CAR [16], MQ [1], and so on, but they use different balance adjustment methods. The characteristic of this strategy is that it can be used in two different access patterns but not for more access modes. The strategy implementation is relatively simple, like the ARC algorithm in IBM's high-end storage systems. Application-based strategy is usually based on some kind of special application cache management optimization or according to some specific information, such as DBMIN [17], Application-Controlled File Caching Policies [18, 19]. This strategy has good effects on the specific applications, and the defect is the poor generality. It adopts the way of clues to display information; application program interface is required to provide support. Cache management strategy which is based on detecting generally tracks the access mode, identifies what the current application accesses belong to, such as the circulation and sequential or random pattern, and then adaptively manages according to the predetermined method. The representative algorithms are DEAR [20], PCC [21], UBM [12], and AMP [22]. This strategy has strong adaptability to different access modes, but the design implementation is more complex; therefore it is seldom used in the actual production system. From the classification of cache management level, it can be divided into single-stage and multistage cache management algorithm. The former is only for the machine system's cache management, not considering the influence of other levels of cache, such as traditional LRU, MRU, and LRU. But along with the development of network, distributed systems and independent storage system, the data access level increases, different levels of cache form a multilevel system. They influence each other, and the single-stage single-level cache management is not able to meet this need; therefore, multistage cache management becomes a hot spot recently. Multistage cache management algorithm can be divided into two categories: one is the radical collaborative; one is the level perceptual. The former uses information displayed or management interface to coordinate between different cache hierarchies, such as TQ [23], DEMOTE [2], ULC [24], and other algorithms. This type of algorithm can more accurately coordinate the cache management. But it may need to change the existing program interface and need additional traffic load. According to the evaluation [25], the actual effect is not ideal. The latter, level perceptual algorithm, usually predicts and judges based on the implicit information left by upper level cache to decide the management in this level. Some studies [26] propose cache algorithms for cloud computing.

These management algorithms try to improve the performance of the system mainly from the influence of application's access modes on the cache hit ratio, and they are all based on the strong premise—visit from cache must be far higher than from direct access to the data source. But for disk or similar common low-speed buffer medium in the distributed system, these algorithms have no specific considerations. Disk can hold data persistently and generally be used as an agent of cache mediation in local or I/O access path in the distributed system [27]. On one hand, it can reduce the performance loss caused by network delay or server under high load; on the other hand, it also reduces the number of requests to concurrently access the storage system, which indirectly improves the response speed of the server. But because of the disk performance limitations, the cost of waiting for cache replacement is very large. With the high-speed network and server providing high performance, the traditional cache management methods that keep the data in the cache and pursue high cache hit ratio approach cannot necessarily get good performance, in some cases even harmful. But these algorithms take no consideration of the performance of caching management and data source itself, and in the current distributed system, these factors are very important to the cache performance. C-Aware algorithm, different from the traditional algorithms, takes the cache speed characteristics of the medium itself and the data source current response performance into consideration in the cache management. C-Aware in the cache placement decision is not based on the access-based placement strategy, but the current cache and data source access conditions to dynamically decide whether to cache data or not. It does not pursue high cache hit ratio as the goal but aims to enhance the overall performance of the system and the adaptability in different media performance and service load. For the distributed cache system and network storage system that use low-speed medium such as disk on the client and the I/O access path, this algorithm has stronger practical significance in improving the overall performance of the multilevel cache.

3. Description of the C-Aware Algorithm

3.1. Basic Idea of C-Aware. The main difference between C-Aware and other traditional algorithms is that C-Aware does not try to cache every request (in this paper, cache means the disk media to store data copy in the computing node) but decides whether to cache data according to current access environment. It gives full consideration to the cache replacement, cache data source, current load, and other factors in cache management. Its idea is simple: C-Aware records the response times of history accesses to cache or source device and guides the cache management decision through a heuristic method based on the past information.

C-Aware framework consists of three components: (1) the original cache management algorithm; (2) C-Aware core, which guides the cache management through heuristic anticipation with history information; and (3) a tracer, which records the processing time for each request.

Based on the history information, C-Aware core heuristically anticipates the future access situation and implements the decision of cache data. C-Aware core is implemented separately for each cache replacement algorithm. It decides whether to cache the data block needed in current access before the replacement algorithm. Once caching the current data block, C-Aware will utilize a traditional replacement algorithm to put the data block into the cache, otherwise, it would not cache the data block. And C-Aware will bypass cache and directly get the data from the network in the next accesses. In short, C-Aware considers access characteristics and workload of cache media, improves the cache performance under heavy server's workload, and tries to eliminate the side effect of caching data on low-speed cache media.

3.2. C-Aware Tracer. In the C-Aware algorithms, the response time of cache and network access is adopted to serve as parameter to assess the network access speed characteristic and the storage network workload. The tracer traces and records every access type and response time and uses them as the basis for C-Aware core decision. Based on different access objects and interface in linux system, the I/O access requests can be divided into six types.

(1) Cache Write Request. This type of request refers to all I/O write requests which are sent to cache by C-Aware.

(2) Direct Cache Write Request. This kind of request refers to the write request which is sent directly to cache media without waiting for cache replacement, write-back, and prereading operation. These requests are a subset of Cache Write Requests.

(3) Source Write Request. This category of request refers to the write request which is sent to source media by C-Aware.

(4) Cache Read Request. This type of request refers to all read requests which are sent to cache media.

(5) Direct Cache Read Request. This kind of request refers to the read request which is sent directly to cache media without waiting for cache replacement, write-back, and prereading operation. These requests are a subset of Cache Read Requests.

(6) Source Read Request. This category of request refers to the read request which is transferred to source media by C-Aware.

C-Aware organizes cache space according to cache block as the unit, and I/O requests are usually less than the size of the cache block size. In order to maintain the cache management, a missing data reads into the cache in accordance with the whole cache blocks. Before the entire cache block of data is read into, although cache blocks have been reported in the current buffer index, the data in the cache block is not immediately available. At this time, other I/O access request needs to wait for the cache block read operation to be completed. Under the simplified D-Cache multilevel model, the read of Cache block in the layer disk needs a process; its time influenced by the response time of the network and lower storage. Therefore, in this case, a cache hit delay is more possible than that of direct access to the cache. So the cache hit access is divided into two cases: (1) cache directly hits and does not need to wait because of the cache management; (2) cache hits but needs to wait for the cache block fetches. The two are different in performance.

In C-Aware, we obtain the average response times to predict the response time of current response. The internal storage for recording the total time and access frequency of every request is limited. However, this would not reflect the current access situation accurately due to the different characteristics at different stages. For example, the office system has periodic fluctuation. In the morning of Monday, it has a vast number of visits, but on weekends, that would be a few. Suppose that the current access characteristics can reflect the current system situation, C-Aware gives different weights to different response time for the access requests at different times. As for the specific design, we refer to the I/O scheduling algorithm in the Linux kernel to count I/O response time. C-Aware tracer records the three parameters of each type of requests:

(1) sample, presents the total requests of each type;

(2) totaltime, records the total handle time until the last request is finished by C-Aware;

(3) meantime, presents average response time for different types of request, the basis for C-Aware to judge the current load.

The formulas are

$$\text{totaltime}_{n+1} = \frac{7 \times \text{totaltime}_n + 256 \times \text{last\_request\_handle\_time}}{8}, \quad (1)$$

$$\text{samples}_{n+1} = \frac{7 \times \text{samples}_n + 256}{8}, \quad (2)$$

$$\text{meantime}_{n+1} = \frac{\text{totaltime}_{n+1} + 128}{\text{samples}_{n+1}}. \quad (3)$$

The last_request_handle_time is the handle time of the last I/O request of this type. It can be seen from (1) that the handle time for the most current request has the biggest weight. However, along with the arriving of new requests, the former request weights are reducing. This statistical method is simple, but it considers the influence of past requests by giving a different weight to history total time and it consumes few resources of the system. Therefore, the C-Aware method is easy to apply and has strong practicality. The experimental results also show its validity.

3.3. C-Aware Core. Through the tracer, C-Aware obtains the average response times of different types of request, predicts the cache and storage situation, and makes management decision. When the current access data hit the cache, C-Aware transfers the request to the cache; otherwise, it handles the request as follows.

Rule 1. If the current request is read and the cache is full, $\text{Meantime}_{\text{Cache\_Read\_Request}} < \text{Meantime}_{\text{Source\_Read\_Request}}$, C-Aware decides to cache data and selects a cache block to replace according to predefined replacement algorithm.

Rule 2. For write request, if cache is full, $\text{Meantime}_{\text{Cache\_Write\_Request}} < \text{Meantime}_{\text{Source\_Write\_Request}}$, C-Aware will make a decision to cache current request and then make a cache block replacement.

Rule 3. If the current request is read, there are still free cache blocks, and $\text{Meantime}_{\text{Direct\_Cache\_Read\_Request}} > \text{Meantime}_{\text{Source\_Read\_Request}}$, C-Aware would not cache the data request.

Rule 4. For write request, there are still free cache blocks, and $\text{Meantime}_{\text{Direct\_Cache\_Write\_Request}} > \text{Meantime}_{\text{Source\_Write\_Request}}$; C-Aware would not cache the data request.

Rule 5. If the cache is full and it does not meet Rules 1 and 2, C-Aware would not cache the data request.

Rule 6. If there are still free cache blocks and it does not meet Rules 3 and 4, then C-Aware allocates a cache block to cache the data request.

C-Aware is very easy to combine with common cache replacement algorithm, as shown in Algorithm 1.

C-Aware renews the access after handling every I/O request. If there are no more requests of the same type, the request information will keep the same. Therefore, the problem emerges: if the response time of the type cannot be renewed, the system load situation will not be true. For example, when the storage network load rises, the value of $\text{Meantime}_{\text{Source\_Write\_Request}}$ may be much bigger than the value of $\text{Meantime}_{\text{Cache\_Write\_Request}}$. So the C-Aware can choose to cache data, and all the requests should be completed in the cache as much as possible. In this case, because the following request reaches the priority access to the cache, $\text{Meantime}_{\text{Source\_Write\_Request}}$ values may not be updated for a long time. When the storage network load drops, the algorithm will not be able to reflect the current situation, still accessing the data from the cache. In order to solve this problem, C-Aware's solution is to regularly update the average response time for each type of request. Specifically, C-Aware sets minimum update interval time. If the average response time of a particular type of requests is longer than that, C-Aware will update automatically. The current practice is to set the average response time half of the old values, so after a certain time, the response time of this type of request will be very low. Therefore, according to the rules set by the C-Aware, it will take the initiative to choose the type of request, thus updating the average response time.

4. Evaluation

We first evaluate the C-Aware algorithms by using caching system simulator based on the trace and test the effectiveness of C-Aware with different applications' traces. Then, we set up a D-Cache system in multiple computing nodes and a web storage server test environment. The D-Cache System executes the prototype C-Aware algorithm. This paper uses iozone test tools to compare the D-Cache system integrating C-Aware algorithms with the one using only LRU algorithm and at the same time compare the performance of IBM dm-Cache [28] caching system.

4.1. Simulation Results. The cache simulator of cloud computing system architecture is realized based on disksim [29]. It simulates disk-based cache in client by disksim and simulates the access pattern between computing nodes and storage servers in cloud computing by setting network delay, access delay of storage server, and other parameters. In the simulation environment, we realize the disk-based cache management algorithm like D-Cache and dm-Cache as the real prototype system. It uses the LRU replacement algorithm to compare simulation results. In simulation, the simulation parameters are shown in Table 1. In order to simplify the design of the simulator, we set the value of write-back cache read data, proofread write cache data, network delay, and storage server access delay as fixed average time.

During the test, we use the block level of OLTP and WebSearch. The OLTP test is a test of read/write hybrid and WebSearch test is a test of read operation, with the write operation rate being extremely low, which is related with the network characteristics of the search request. In simulation test, the average value of load access delay of storage server ranges from 1 ms to 800 ms. Figures 2 and 3 show the simulation results of OLTP and WebSearch trace when the storage server load changes. The test results are in accordance with the various algorithms' average response time per 100 requests.

The "direct" means the test results without accessing the disk Cache storage server. Because other parameters are fixed average, the change of direct response time will reflect the storage server load changes. For each trace, we test the cache performance of two disks: one is higher, disksim simulator sets the average seek time as 2 ms; another performance is low,

```
Cache_Handle (Request){
    Get cache_block from data cache according to current request
    if (cache_block ==NULL){
        if (Cache is full){
            if (Rule 1 or Rule 2 is satisfied)
                Cache_Miss (Request);
            else
                Access the storage network directly;
        } else if (Rule 3 or Rule 4 is satisfied){
            Access the storage network directly;
        } else {
            Cache_Miss (Request);
        }
    } else
        Cache_Hit (Request);
    }
```

Algorithm 1: Algorithm of C-Aware core.

Table 1: Simulation test parameters.

Parameter	Value
Storage server access latency under normal circumstances	0.1 ms
Access delay with increasing of storage server load	According to specific tests
Network delay	0.01 ms~0.1 ms
Write-back cache operation delay	10 ms
Proofread write cache operation delay	10 ms
Disk cache block size	256 K
Cache size	1.25 G, 2.5 G, 12.5 G

the average seek time is 3.5 ms. Specific test results are shown in Figures 2 and 3: (a, b) is the situation when the average seek time is 2 ms. (c, d) is the situation of 3.5 ms. In (a, c) the cache size is 1.25 G, while in (b, d) the cache size is 12.5 G.

It can be seen from the test that when cache access performance is higher (Figures 2(a), 2(b), 3(a), and 3(b)), as D-Cache and dm-cache, the C-Aware algorithm can improve the overall performance of the system. But compared with the D-Cache and dm-Cache, the C-Aware algorithm performance will lose slightly when the cache is larger. This is because the C-Aware will forward requests to the storage server from time to time to detect the storage server's current situation, so as to decide whether to place the data blocks to the local disk cache or not. Therefore, it leads to a higher average response time when the system load is high. On the other hand, when the cache response performance is low (Figures 2(c), 2(d), 3(c), and 3(d)), C-Aware algorithm has stronger adaptability. It can improve the performance in both OLTP and WebSearch's trace tests. But in the reading-dominant WebSearch tests, the average response time of I/O requests is very high when using D-Cache and dm-Cache system. Its performance is much lower than accessing storage system directly.

From Figures 2(c) and 2(d), we can see that C-Aware can significantly improve the speed of the I/O processing at high load with the change of the storage server load. Instead, the speed change of D-Cache and dm-cache is very large without C-Aware algorithm, in many cases, far more than the time required to access storage server directly. The phenomenon is particularly prominent when the cache performance is not high, and the space is little but requires a lot of cache, as shown in Figure 2(c). This is because the D-Cache and dm-Cache use the traditional Cache management algorithm which does not consider cache speed characteristics of the media itself, resulting in a large number of I/O being directed to the low performance cache, thereby reducing the performance of the whole system. C-Aware algorithm will adjust based on the current cache and the storage network access and decide whether to store the following data. Thereby, it obtains a better performance as to the access balance between the cache and the storage server.

In Table 2, we can see that when combined with high-performance cache in the OLTP tests, the cache hit ratio difference in the three types of caching system with different cache size is not big. C-Aware declines a bit 1%~2%, which shows that the C-Aware can achieve better performance by finding cache, thus improving the hit ratio of cache by saving data as far as possible. However, when the performance of the cache is not good, in Table 2, we can observe that the C-Aware will take the initiative to reduce the cache hit ratio and the access to the cache system appropriately, thus improving the overall performance of the system. With decreasing cache size, C-Aware forwards more I/O requests directly to the storage system for processing, and the corresponding cache hit ratio will be lower. But in WebSearch tests, C-Aware also shows the similar phenomenon, which is more obvious. Because the read operation is more time-consuming, in WebSearch tests, when the cache is small, C-Aware will take

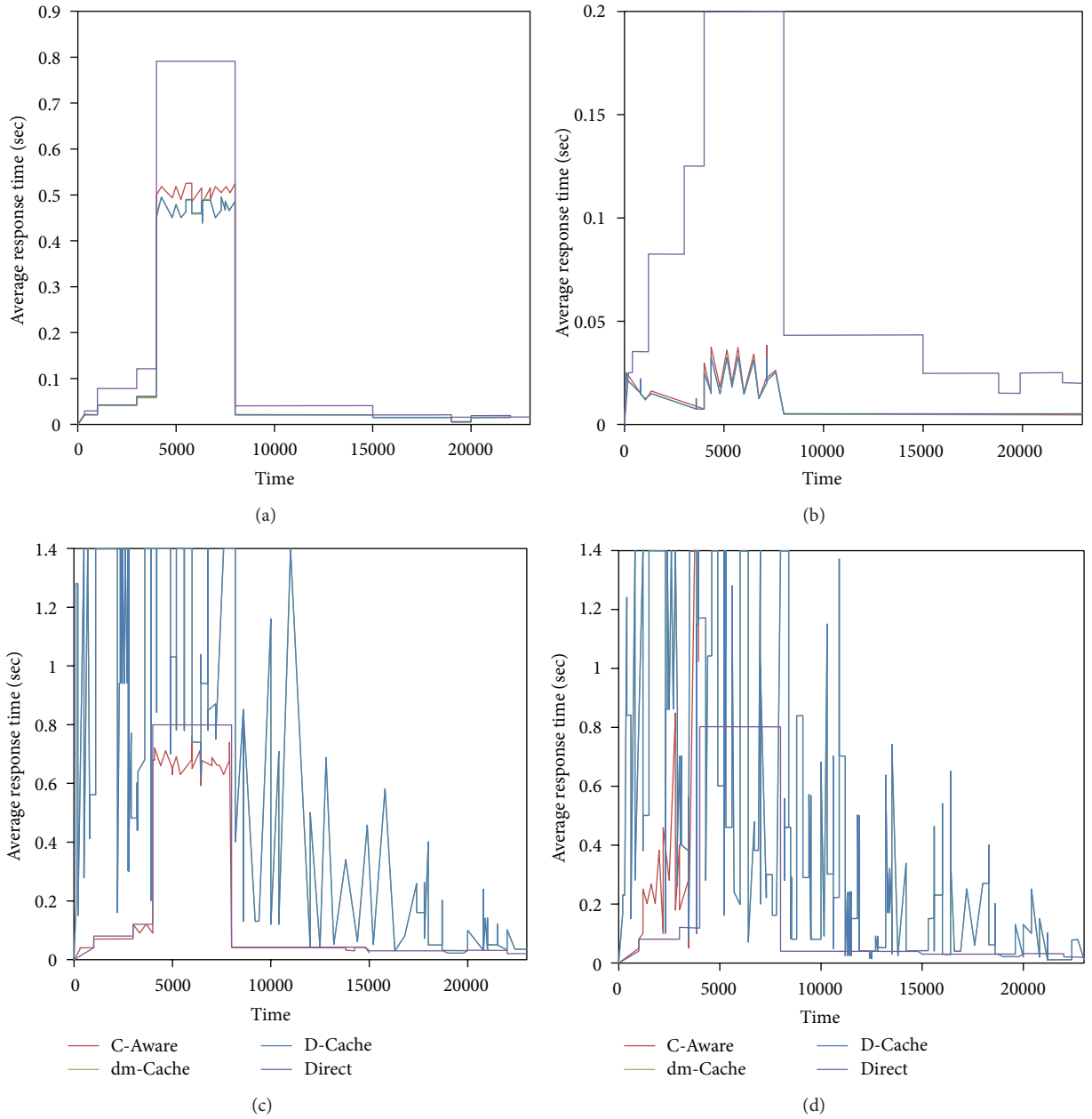


Figure 2: WebSearch test results.

the initiative to reduce the frequency with the cached data and send more I/O requests to the server in order to reduce the load of the disk cache and obtain a better overall performance. Table 3 clearly illustrates the problem with the I/O numbers corresponding to the C-Aware algorithm. When the cache size is 1.25 G, average seek time of cache disk is 3.5 ms, C-Aware has 1130607 requests sent directly to the data source for processing, which account for a quarter of the total number of requests, while under the traditional algorithm there are only 176 requests. Due to the increasing number of requests, the average response time of C-Aware in this test reduces nearly 80%.

We also test the cache storage server under high-load and low-load conditions (for publication reason, we do not list the results here). The test results also show that the C-Aware has better adaptability. Especially when the cache is small and the storage server is able to provide high performance access, C-Aware can significantly reduce the cache data operation and tries to send the request directly to the storage server to complete the processing. In the end, it can achieve better performance. Through the simulation tests, we can see that the C-Aware can dynamically decide whether to cache data or not according to the current access condition and speed properties of the cache media, so that it can in most cases ensure a better overall performance.

4.2. Prototype System Test Results. The following experiments have been done on one storage server and eight computing

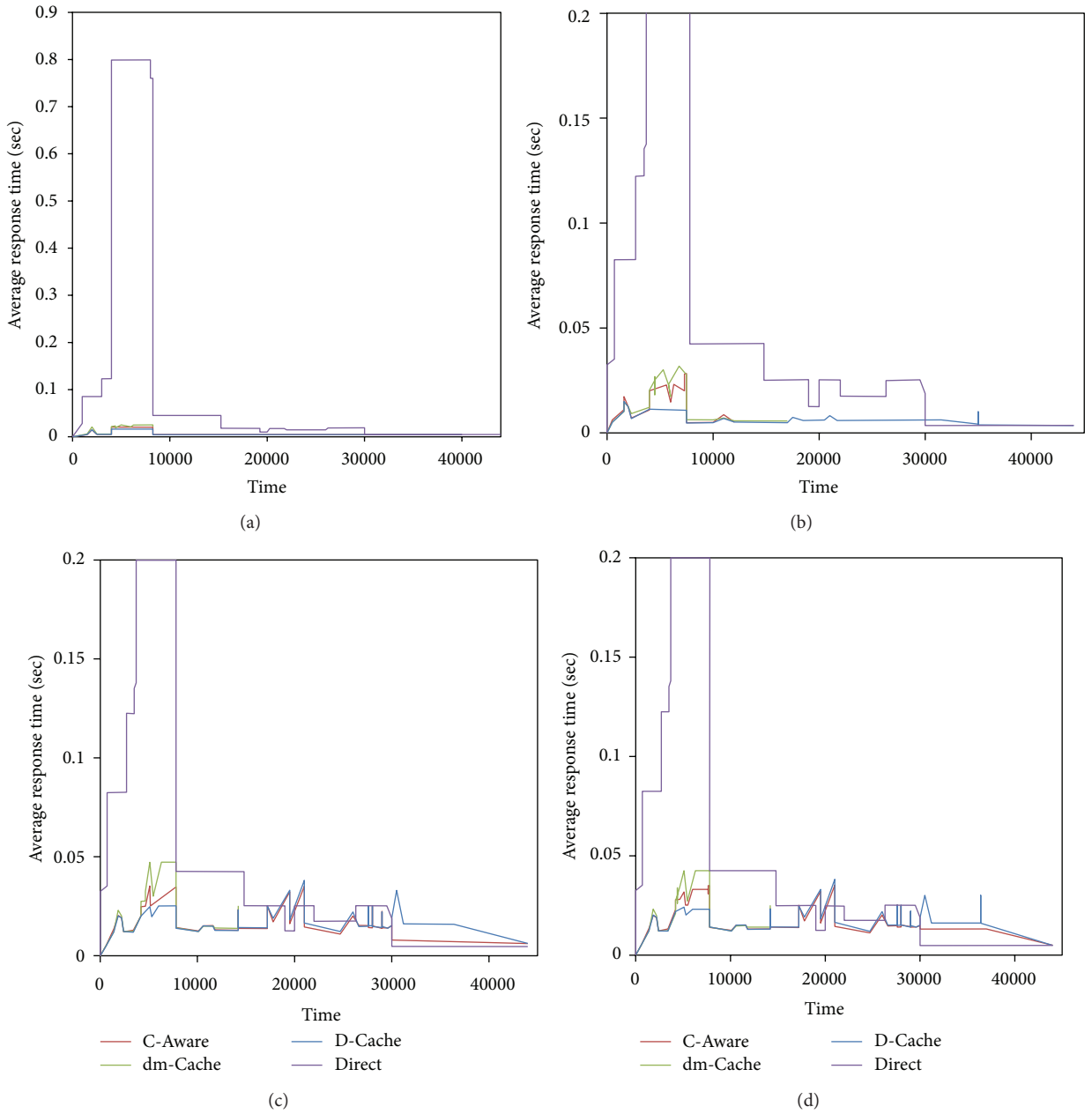


Figure 3: OLTP (finance1) test results.

nodes. The storage server adopts the Intel Xeon(TM) CPU 3.20 GHz × 2.4 GB main memory, Adaptec (formerly DPT) SmartRAID V RAID Controller and six 74 GB SCSI disks to construct a RAID0 storage system, and Intel 82546 GB Gigabit ethernet. The storage server runs 32 bit Fedora Core 4 with 2.6.11-1.1369FC4smp kernel. All AS are equipped with an Intel Xeon(TM) CPU 3.20 GHz × 2.4 GB main memory, Adaptec (formerly DPT) SmartRAID V RAID Controller and three 74 GB SCSI disks to construct a RAID0 storage system, and Intel 82546 GB Gigabit ethernet. Each AS runs 64 bit Fedora Core 4 with x8664 2.6.17-1.2142FC4smp kernel. Both the application servers and storage server are interconnected via Giga-bit Ethernet, and NBD agreement [1] to access the data. We use iozone as the major benchmark tool for this evaluation and 8 G as the size of the test-file, so as to reduce the influence of internal memory on the results. In order to investigate the effects of each algorithm on the scalability of storage system, the benchmark was executed with 1, 2, 4, 8's computing nodes. During the process, each computing node starts the iozone to read/write concurrent access to the storage server at the same time. The tests include:

(1) sequential, random, and mix read/write iozone,

(2) cold cache and warm cache. The test file is already in the cache or not. Before each test, the application server will restart in order to ensure no valid data

Table 2: OLTP trace simulation test results.

		Average seek time 2.0 ms			Average seek time 3.5 ms		
	Direct	C-Aware	D-Cache	dm-Cache	C-Aware	D-Cache	dm-Cache
Cache size 1.25 G							
Average response time (ms)	102.948	7.477	6.852	8.310	13.767	16.004	17.629
Hit ratio (%)		96.57	98.57	98.91	89.19	98.57	98.91
Cache replacement		14283	14989	14989	12502	14989	14989
Direct I/O	5394885	136497	25677	25677	545242	25677	25677
Cache size 2.5 G							
Average response time (ms)		7.225	6.440	7.835	14.268	15.739	17.296
Hit ratio (%)		98.12	99.26	99.46	92.77	99.26	99.46
Cache replacement		6141	6374	6376	5298	6374	6376
Direct I/O		71676	9021	9026	363569	9021	9025
Cache size 12.5 G							
Average response time (ms)		7.304	6.448	7.840	14.979	15.904	17.460
Hit ratio (%)		99.00	99.52	99.69	95.12	99.52	99.69
Cache replacement		0	0	0	0	0	0
Direct I/O		28490	0	0	239552	0	0

Table 3: Web search simulation test results.

		Average seek time 2.0 ms			Average seek time 3.5 ms		
	Direct	C-Aware	D-Cache	dm-Cache	C-Aware	D-Cache	dm-Cache
Cache size 1.25 G							
Average response time (ms)	200.587	162.068	266.522	266.527	171.770	967.141	967.145
Hit ratio (%)		52.24	63.80	63.80	49.59	63.80	63.80
Cache replacement		936005	1217932	1217932	874800	1217932	1217932
Direct I/O	4381687	928748	176	176	1130607	176	176
Cache size 2.5 G							
Average response time (ms)		140.017	242.872	242.873	151.863	941.943	941.943
Hit ratio (%)		63.92	71.94	71.94	61.74	71.94	71.94
Cache replacement		714641	866160	866160	669189	866160	866160
Direct I/O		593114	104	104	753693	104	104
Cache size 12.5 G							
Average response time (ms)		74.244	129.943	129.944	355.029	829.776	829.777
Hit ratio (%)		96.23	98.02	98.02	95.39	98.02	98.02
Cache replacement		8024	9959	9959	7304	9959	9959
Direct I/O		93425	46	46	136094	46	46

in the application server memory, thus reducing the influence of memory cache data on testing. The cache replacement algorithms adopted by D-Cache, dm-Cache, and C-Aware are based on the LRU algorithm, and the size of the Cache block is 128 k. Cache capacity can accommodate 50% of test data.

Figure 4 shows the result of Iozone Sequential Test on warm cache. It is the rewrite, read, and reread test immediately after the finishing of iozone's first write test. From Figure 4, we can see that D-Cache that integrated C-Aware has better performance compared with dm-cache which only has LRU algorithm. In the first read test, when the number of clients is less than 8, the D-Cache performs best. The reason is that the cost of cache replacement leads to the decrease of performance. However, the C-Aware can adjust according to the load situation. At the start of the test, since there is no replacement operation, C-Aware would believe the local cache can improve the performance. With the reading full, the cost of cache replacement can decrease the performance; the C-Aware will forward the subsequent requests directly to the storage devices in order to achieve a good overall performance.

In cold cache test, as for pure write test, the difference between writing the already existing file and writing the new one is not that big. The test result should be similar to Figure 4(a), so we did not test again for this kind of situation, and only tested the read and reread as shown in Figure 5.

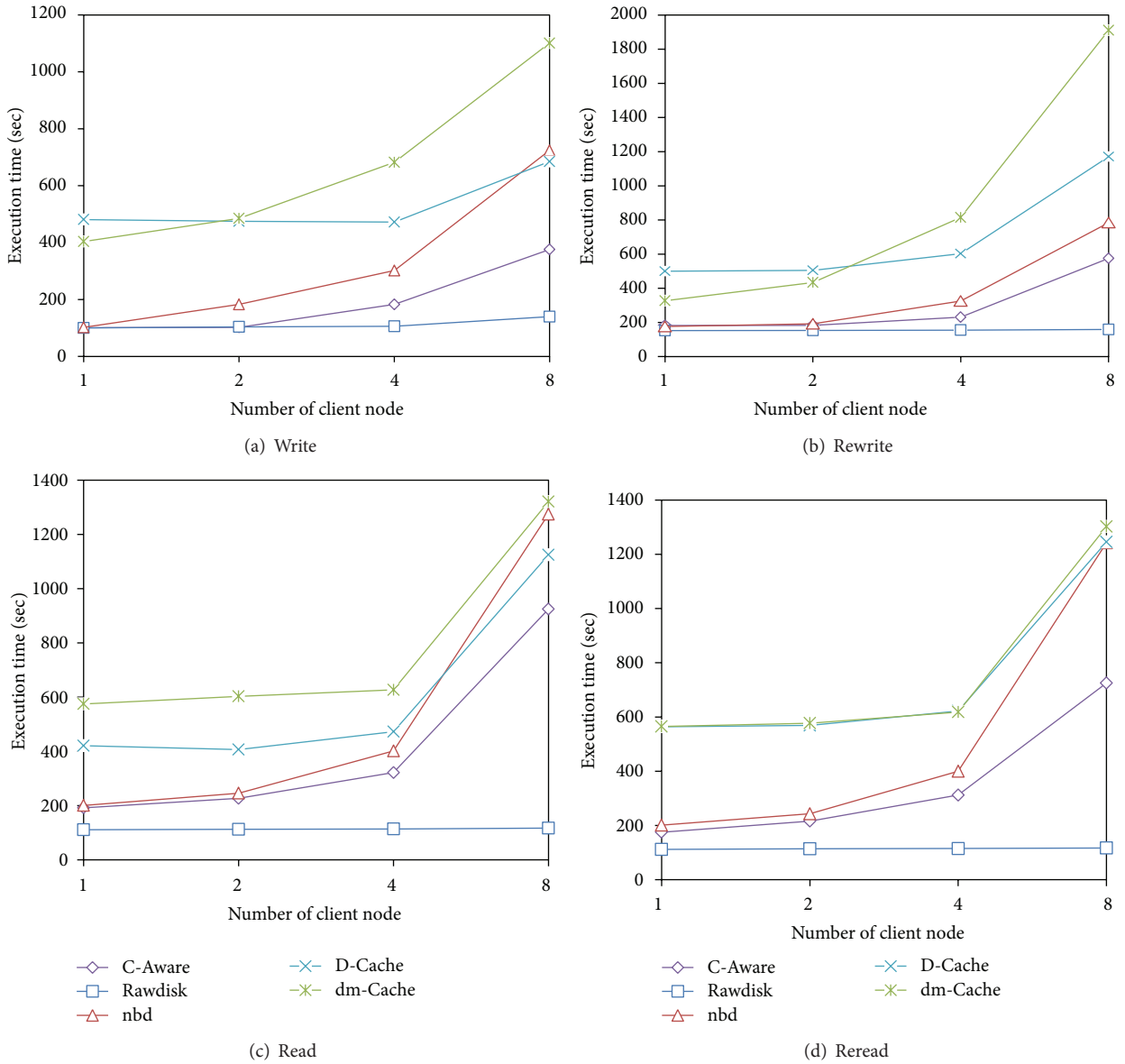
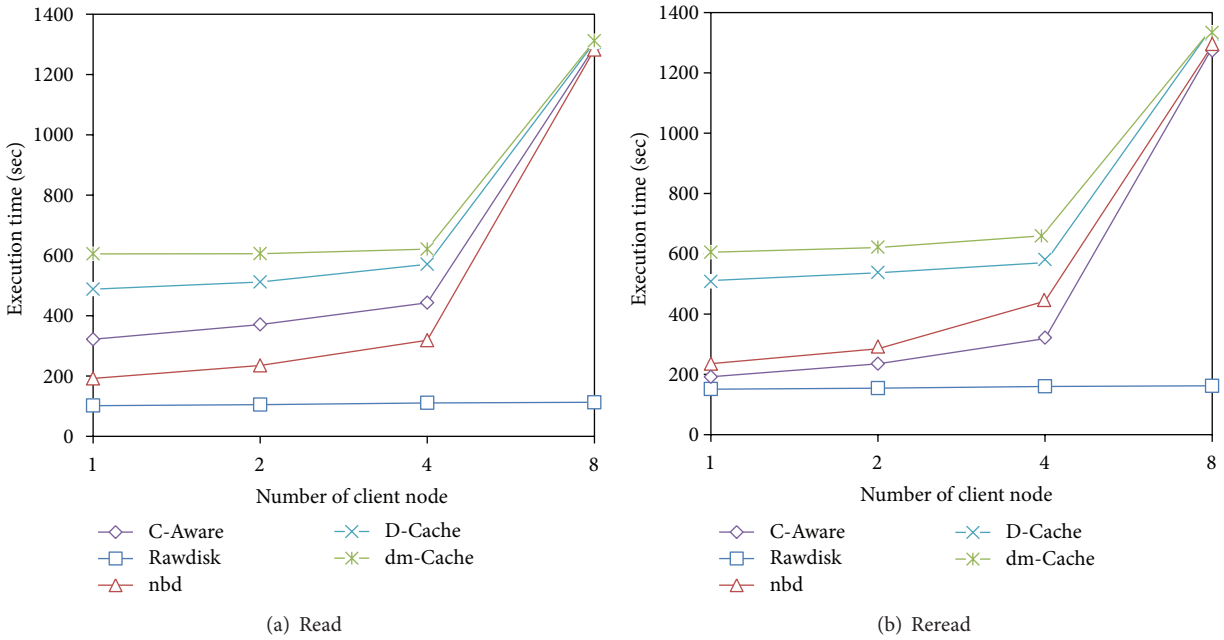


Figure 4: Result of iozone sequential test on warm cache with 50% test data size.

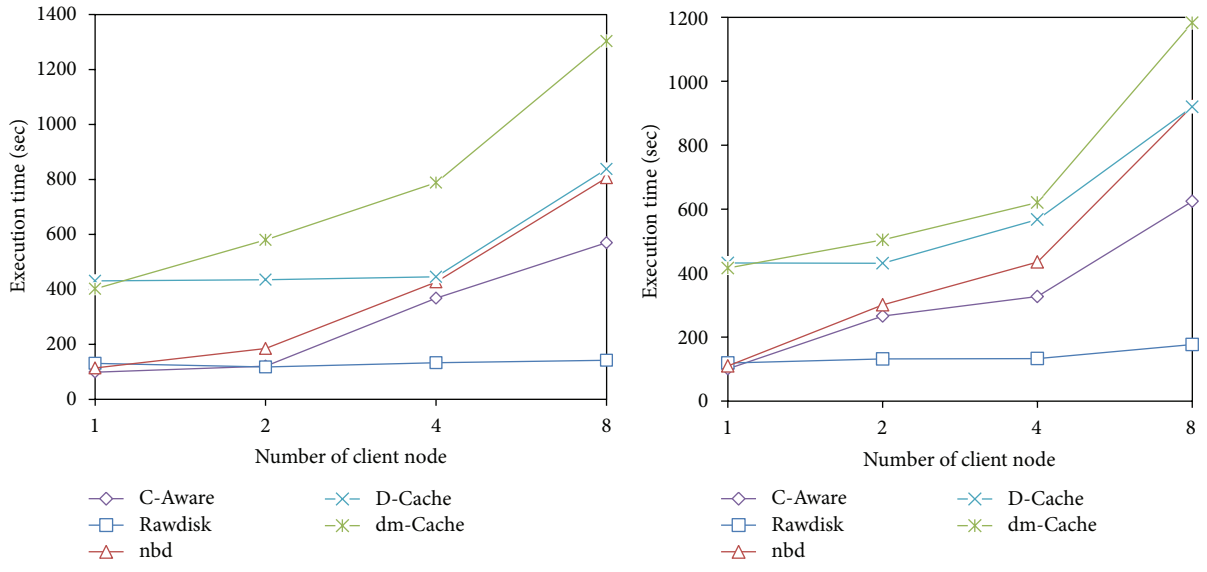
We can see the performance of first read. When the storage server load is low, the performances of dm-cache, D-Cache, and C-Aware are far lower than direct network access. This is because the read request needs to wait for the prereading generated by the cache strategies, and after being written to the cache they can continue to reduce the performance of the read access. Meanwhile, the cache space is limited; cache block replacement further reduces the performance of the system in the reading test process. Compared with the dm-cache and D-cache, the performance of C-Aware is increased by nearly 25%; this is because the C-Aware will try to buffer cache data to reduce the cost and improve the performance. Since it caches part of the data in the cache, it provides a better performance in the subsequent reread. We can also find that with the increase of the storage server system load, the performance difference of dm-cache, D-cache, and C-Aware in the first read data test and direct network access is becoming smaller. This is because the relative continuity and cache prefetching have played an important role in improving performance.

4.2.1. Iozone Mix Read/Write. In the mix tests, we mainly test when the read/write percentage is 30% and 80%. Figure 6 reflects warm cache test results. It can be found that in the tests where read occupies a larger scale, the system performance is better. This is because the read operation needs to be synchronized but write can be asynchronous. Figure 7 reflects test results in cold cache. We can find that, in the case of a cold cache, the cache system achieves relatively better performance. We guess this is because the warm cache,



(a) Read (b) Reread

Figure 5: Result of iozone sequential test on cold cache with 50% test data size.



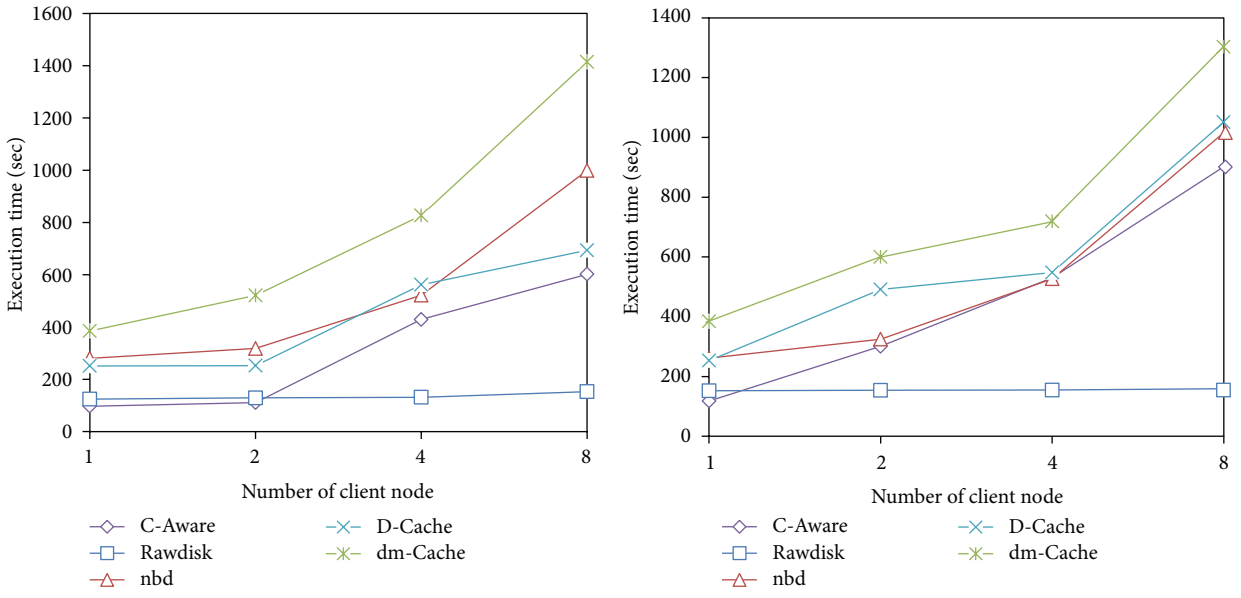
(a) 30% read (b) 80% read

Figure 6: Result of iozone mix test on warm cache with 50% test data size.

after the first write, caches part of the test file, and that will generate cache replacement costs in the mix tests, which can reduce the performance. However, the buffer is empty in the cold cache test at the beginning, thus the write requests can be directly written to the cache, which improves the test performance. From mix test results, it can be seen that the caching system combined with C-Aware algorithm has better performance in most cases.

5. Conclusion

This paper presents a storage cache placement algorithm—C-Aware, which considers the speed characteristics and network access situation of computing nodes' cache media. It adaptively decides whether to cache data block by history access information of data source. It mainly concerns the influence of cache media speed characteristic, prefetch cost,



(a) 30% read

(b) 80% read

FIGURE 7: Result of iozone mix test on cold cache with 50% test data size.

and network transmitting situation on the overall system performance. As a result, it achieves good adaptability to workload difference and cache media characteristics. The benchmark and trace simulation tests have verified the conclusion. Suggestions for future research include first, realize C-Aware at the first document level and further test its validity; second, currently, C-Aware just considers the response time of I/O handling, and more system parameters should be taken into account.

Conflict of Interests

The authors declare that there is no conflict of interests regarding the publication of this paper.

Acknowledgments

This work is supported in part by Natural Science Foundation of China "Research on the snapshot data security storage technology for authorization of release," no. 61100057, and the National Basic Research Program of China, no. 2004CB318205.

References

[1] Y. Zhou, Z. Chen, and K. Li, "Second-level buffer cache management," *IEEE Transactions on Parallel and Distributed Systems*, vol. 15, no. 6, pp. 505–519, 2004.

[2] M. Theodore Wong and J. Wilkes, "My cache or yours? making storage more exclusive," in *Proceedings of the USENIX Annual Technical Conference*, pp. 161–175, 2002.

[3] Y. Yang, *Research on multi-level and low-cost cache for network storage, [Ph.D. thesis]*, Institute of Computing Technology, Chinese Academy of Science, 2009.

[4] R. Sandberg, D. Goldberg, S. Kleiman, D. Walsh, and B. Lyon, "Design and implementation of the sunnetwork file system," in *Proceedings of the Summer USENIX*, pp. 119–130, June 1985.

[5] J. H. Howard, "An overview of the andrew file system," in *Proceedings of the Winter USENIX Conference*, pp. 23–26, Dallas, Tex, USA, February 1988.

[6] M. Satyanarayanan, "Scalable, secure, and highly available distributed file access," *IEEE Computer Society*, vol. 23, no. 5, pp. 9–20, 1990.

[7] T. Anderson, M. Dahlin, J. Neefe, D. Patterson, D. Roselli, and R. Wang, "Serverless network file systems," in *Proceedings of the 15th Symposium on Operating Systems Principles*, ACM Transactions on Computer Systems, 1995.

[8] M. Vilayannur, P. Nath, and A. Sivasubramaniam, "Providing tunable consistency for a parallel file store," in *Proceedings of the 4th USENIX Conference on File and Storage Technologies*, December 2005.

[9] N. Megiddo and D. S. Modha, "ARC: a self-tuning, low overhead replacement cache," in *Proceedings of the USENIX Conference on File and Storage Technologies (FAST '03)*, San Francisco, Calif, USA, March 2003.

[10] T. Johnson and D. Shasha, "2Q: a low overhead high performance buffer management replacement algorithm," Proceedings of the VLDB-20, September 1994.

[11] S. Jiang and X. Zhang, "LIRS: an efficient low inter-reference recency set replacement policy to improve buffer cache performance," in *Proceedings of the ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, pp. 31–42, June 2002.

[12] J. M. Kim, J. Choi, J. Kim et al., "A low-overhead high-performance unified buffer management scheme that exploits sequential and looping references," in *Proceedings of the 4th conference on Symposium on Operating System Design & Implementation*, San Diego, Calif, USA, 2000.

[13] Z. Chen, Y. Zhou, and K. Li, "Eviction-based cache placement for storage caches," in *Proceedings of the USENIX Annual Technical Conference*, pp. 269–282, 2003.

[14] S. Jiang, X. Ding, F. Chen, E. Tan, and X. Zhang, "DULO: an effective buffer cache management scheme to exploit both temporal and spatial locality," in *Proceedings of the 4th USENIX Conference on File and Storage Technologies*, San Francisco, Calif, USA, 2005.

[15] D. Lee, J. Choi, J.-H. Kim et al., "LRFU: a spectrum of policies that subsumes the least recently used and least frequently used policies," *IEEE Transactions on Computers*, vol. 50, no. 12, pp. 1352–1361, 2001.

[16] S. Bansal and D. S. Modha, "CAR: clock with adaptive replacement," in *Proceedings of the USENIX File and Storage Technologies (FAST '04)*, San Francisco, Calif, USA, 2004.

[17] H. -T. i Chou and J. D. David, "An evaluation of buffer management strategies for relational database systems," in *Readings in Database Systems*, pp. 174–188, 1988.

[18] P. Cao, E. W. Felten, and K. Li, "Application-controlled file caching policies," in *Proceedings of the Technical Conference on Summer USENIX*, Boston, Mass, USA, 1994.

[19] R. H. Patterson, G. A. Gibson, E. Ginting, D. Stodolsky, and J. Zelenka, "Informed prefetching and caching," in *Proceedings of the 15th ACM Symposium on Operating Systems Principles*, pp. 79–95, ACM Press, 1995.

[20] J. Choi, S. H. Noh, S. L. Min, and Y. Cho, "An implementation study of a detection-based adaptive block replacement," in *Proceedings of the Annual USENIX Technical Conference*, pp. 239–252, 1999.

[21] C. Gniady, A. R. Butt, and Y. C. Hu, "Program counter based pattern classification in buffer caching," in *Proceedings of the 6th Symposium on Operating Systems Design and Implementation (OSDI '04)*, 2004.

[22] F. Zhou, R. von Behen, and E. Brewer, "Program context specific buffer caching with AMP," Tech. Rep. UCB CSD-05-1379.

[23] X. Li, A. Aboulnaga, K. Salem, A. Sachedina S, and Gao, "Second-tier cache management using write hints," in *Proceedings of the 4th conference on USENIX Conference on File and Storage Technologies*, San Francisco, Calif, USA, 2005.

[24] S. Jiang and X. Zhang, "ULC: a file block placement and replacement protocol to effectively exploit hierarchical locality in multi-level buffer caches," in *Proceedings of the 24th International Conference on Distributed Computing Systems*, pp. 168–177, March 2004.

[25] Z. Chen, Y. Zhang, Y. Zhou, H. Scott, and B. Schiefer, "Empirical evaluation of Multi-level Buffer Cache Collaboration for Storage System," in *Proceedings of the ACM International Conference on Measurement and Modeling of Computing Systems (SIGMETRICS '05)*, June 2005.

[26] H. Lee, "loudCache: Expanding and shrinking private caches," in *Proceedings of the IEEE 17th International Symposium on High Performance Computer Architecture (HPCA '11)*, pp. 219–230, San Antonio, Tex, USA, 2011.

[27] H. Kllapi, E. Sitaridi, M. M. Tsangaris, and Y. Ioannidis, "Schedule optimization for data processing flows on the cloud," in *Proceedings of the ACM SIGMOD International Conference on Management of data*, pp. 289–300, New York, NY, USA, 2011.

[28] E. V. Hensbergen and M. Zhao, "Dynamic policy disk caching for storage networking," Tech. Rep. RC24123, IBM Research Division Austin Research Laboratory, 2006.

[29] http://www.pdl.cmu.edu/DiskSim/.