A Unified Multiple-Level Cache for High Performance Storage Systems [*]

Li Ou, Xubin (Ben) He, Martha J. Kosa
Tennessee Technological University
{lou21,hexb,mjkosa}@tntech.edu

Stephen L. Scott
Oak Ridge National Laboratory
scottsl@ornl.gov

Abstract

*Multi-level cache hierarchies are widely used in high-performance storage systems to improve I/O performance. However, traditional cache management algorithms are not suited well for such cache organizations. Recently proposed multi-level cache replacement algorithms using aggressive exclusive caching work well with single or multiple-client, low-correlated workloads, but suffer serious performance degradation with multiple-client, high-correlated workloads. In this paper, we propose a new cache management algorithm that handles multi-level buffer caches by forming a unified cache (*uCache*) which uses both exclusive caching in L2 storage caches and cooperative client caching. We also propose a new local replacement algorithm, Frequency Based Eviction-Reference (*FBER*), based on our study of access patterns in exclusive caches. Our simulation results show that uCache increases the cumulative cache hit ratio dramatically. Compared to other popular cache algorithms, like LRU, the I/O response time is improved by up to* 46% *for* low-correlated *workloads and* 53% *for* high-correlated *workloads.*

1 Introduction

Caching is a common technique for improving the performance of I/O systems. Researchers have developed many algorithms to manage the buffer cache, such as LRU [6], LFU, 2Q [11], LIRS [9], and ARC [13]. These algorithms were designed for local cache replacement because they do not need any information from other caches. They worked well for a single system. In a distributed I/O environment, buffer caches are mostly organized as multi-level cache hierarchies residing on multiple machines. For example, in a distributed file system, the upper level caches reside on file servers (storage clients), and the lower level caches reside on storage servers. We refer to upper level storage client caches as L1 buffer caches and lower level storage caches as L2 buffer caches [21]. L1/L2 buffer caches are very different from L1/L2 processor caches because L1/L2 buffer caches refer to main-memory caches distributed in multiple machines. The access patterns of L2 caches show weak temporal locality [3, 8, 21] after filtering from L1 caches, which implies that a cache replacement algorithm, such as LRU, may not work well for L2 caches. Additionally, local management algorithms used in L2 caches are inclusive [20], which try to keep blocks that have been cached by L1 caches, and waste aggregate cache space. Thus, though the aggregate cache size of the hierarchy is increasingly larger, the system may not deliver the expected performance commensurate to the aggregate cache size.

Several attempts have been made to improve cache performance of multi-level buffer caches for distributed I/O systems. Recent research [20, 21, 4, 2, 10] characterizes the behavior of accesses to L2 caches, and introduces multiple algorithms based on the characteristics to improve the L2 cache hit ratio. Except for multi-queue replacement [21], all the other algorithms try to achieve exclusive caching [20] through quick eviction of duplicated blocks in L2 caches. Implementing aggressive exclusive caching may get a high hit ratio in case of a single storage client, but multiple-client systems introduce a new complication: the sharing of data among clients. It may no longer be a good idea to discard a recently read block from the L2 cache after it has been sent to a client cache, because the block may be referenced again by other clients in the recent future. Real workloads show behavior between two extremes: disjoint workloads, in which the clients each issue references for non-overlapping parts of the aggregate working set, and conjoint workloads, in which the clients each issue exactly the same references in the same order at the same time [20]. Nearly disjoint workloads are *low-correlated* workloads, and nearly conjoint workloads are *high-correlated*. For low-correlated workloads, aggressive exclusive caching is ef-

[*]This work was supported in part by Research Office under Faculty Research Grants and Center for Manufacturing Research at Tennessee Technological University, ORAU under the Ralph E. Powe Junior Faculty Enhancement Award, and the Mathematics, Information and Computational Sciences Office, Office of Advanced Scientific Computing Research, Office of Science, U. S. Department of Energy, under contract No. DE-AC05-00OR22725 with UT-Battelle, LLC.

fective, but for high-correlated workloads, since the same blocks may be referenced by multiple clients within a relatively short time period, inclusive caching is more attractive. Thus, for a multiple-client system, it is important to design an algorithm which balances between aggressive exclusive caching and inclusive caching according to workload characteristics. Wong and Wikes [20] propose SLRU and an adaptive cache insertion policy to decide how to cache duplicated blocks according to their previous hit ratios. The simulation results show that it could achieve up to a 1.32 speedup for low-correlated workloads and an approximate 1.18 speedup for high-correlated workloads over the LRU algorithm. It trades a hit ratio for low-correlated workloads for a speedup for high-correlated workloads.

In this paper, we propose a new unified cache management algorithm, *uCache*, for multi-level I/O systems to provide high cumulative hit ratios in multiple storage client cache systems, for both high-correlated and low-correlated workloads. We use cooperative client caches [5] to provide inclusive caching for high frequency block reuse among multiple L1 caches with high-correlated workloads, while implementing exclusive caching in L2 caches to improve the hit ratio for low-correlated workloads. We study the access patterns of exclusive caching and find that LRU and other traditional algorithms are not suitable even for local replacement of L2 caches. Based on our study, we propose a new local L2 cache management algorithm, *FBER*, for exclusive caching environments. We compare the *uCache* algorithm with the traditional LRU and other typical multi-level cache management algorithms such as exclusive caching [20, 21], 2Q [11], and SLRU [20], using simulations under different workloads. The results show that compared to LRU, *uCache* can dramatically increase the overall cache hit ratio and improve the average I/O response time by up to 46% for low-correlated workloads and 53% for high-correlated workloads.

The rest of the paper is organized as follows. Section 2 discusses access patterns of L2 caches in exclusive caching environments. Section 3 describes our idea and design issues in detail. Section 4 describes our simulation methodology. We compare our work to previous efforts to improve L2 cache performance in Section 5 and examine related work in Section 6. We draw our conclusions in Section 7.

2 Analysis of access patterns of exclusive caching

Exclusive caching is different from current inclusive caching in several aspects. First, after it is reloaded into the storage cache, and then referenced by a client, a block is quickly discarded by the management algorithm, no matter how many times it has been referenced before, but traditional algorithms try to keep a block with a recently good hit history in the cache as long as possible. Second, the reference sequences of storage caches are totally different from traditional caches. The access sequences of traditional caches consist of continuous references of blocks, and researchers use some metrics, like *reuse distance* [21], *inter reference gap* [15], and *inter reference recency* [9], to describe characteristics of workloads, which are then used to design replacement algorithms to manage buffer caches. In exclusive caching, the access sequence of storage caches consists of two types of continuous operations: *evictions*, which inform storage systems to reload blocks that have been replaced by client caches, and *references*, such as read or write, provided by a standard I/O interface. A typical access sequence of exclusive caching is interleaved randomly with references and evictions. With these differences, we need to analyze the access patterns of exclusive caching, and design a replacement algorithm dedicated for exclusive caching based on those patterns.

Table 1. Characteristics of traces

Trace	Clients	IOs (millions)	Volume
Cello92	1	0.5 per day	10.4GB
HTTPD	7	1.1	0.5GB
DB2	8	3.7	5.2GB

2.1 Traces

To study L2 buffer cache access patterns and evaluate caching algorithms and policies, we use three buffer cache access traces. These traces are chosen to represent different types of workloads: high-correlated and low-correlated. In our study, we use $4KB$ as the cache block size for our access pattern analysis and our experimental evaluation of various algorithms. We have examined other block sizes, with similar results. Table 1 shows the characteristics of traces.

The *HP Cello92* trace was collected at *Hewlett-Packard* Laboratories in 1992 [17]. It captured all L2 disk I/O requests in *Cello*, a timesharing system used by a group of researchers to do simulations, compilation, editing, and e-mail, from April 18 to June 19. We use the trace collected on April 18 as the workload for the single client simulation. *Cello* is an *HP 9000/877* server with one $64MH$ CPU, $96MB$ memory and 8 disks. Since requests of the traces collected in different days access the same data set, we also use them as workloads for the multiple-client simulation: each trace file collected within one day acts as the workload of one client. These workloads are high-correlated.

The *HTTPD* workload was generated by a seven-node *IBM SP2* parallel web server [12] serving a $524MB$ data set. Multiple http servers share the same files, although they seldom read files at the same time. We use the *HTTPD* workload as the high-correlated workloads for the multiple-client simulation.

The *DB2* trace-based workload was generated by an eight-node *IBM SP2* system running an *IBM DB2* database application that performed join, set and aggregation operations on a $5.2GB$ data set. Uysal *et al.* used this trace in their study of I/O on parallel machines [19]. Each *DB2* client accesses disjoint parts of the database. No blocks are shared among the eight clients. We use the *DB2* workload as the low-correlated workload for the multiple-client simulation.

Since L1 buffer cache sizes clearly affect an L2 cache's performance, we carefully set the L1 buffer cache sizes for the three traces to achieve a reasonable L1 hit ratio. The cache size of *HP 9000/877* server is only $10 - 30MB$, which is very small by current standard. The *Cello92* trace and the *HTTPD* trace show high temporal locality, and a small client cache may achieve a high hit ratio. In the simulations, we assume the cache size of each client is $16MB$ for the *Cello92* traces, and $8MB$ for the *HTTPD* trace, providing an L1 hit ratio of approximately 50%. The *DB2* trace shows very low temporal locality, and a $512MB$ client cache just provides an L1 hit ratio of no more than 15%. But if the cache size increases to $600MB$, the L1 hit ratio suddenly increases to 75%, because *reuse distances* [21] of most blocks are less than $150K$ ($600MB$ divided by block size $4K$). To reserve enough cache misses for L2 caches, we assume the cache size of each client for the *DB2* trace is $512MB$. Since the number of compulsory cache misses in the *DB2* trace is large, we use approximately 10% of the requests to warmup the cache space.

2.2 Access patterns of exclusive caching

Because of the uniqueness of reference sequences, the metrics used before may not correctly describe the characteristics of access patterns for exclusive caching. Thus, we need to define new metrics to describe the access pattern. The *Eviction-Reference-Gap (ERG)* indicates the distance (the number of distinct evictions) between an eviction of a block from an L1 cache and the later reference by that cache. *ERG* describes how long a block will stay in the L2 cache space before it is referenced again. The replacement algorithm should keep blocks with small *ERG* values. The *Eviction Frequency* defines how many times a block has been evicted from the L1 caches, and hence reloaded into the storage cache. Not every eviction of a block will be referenced by an L1 cache again within a reasonable *ERG*: some of them are never referenced again, and some of them are referenced, but with an *ERG* that is much larger than a real cache space can provide. These kinds of evictions are *dead evictions*. Evictions referenced by L1 caches again within a reasonable *ERG* are *reusable evictions*. Obviously, a good replacement algorithm should discard dead eviction blocks as quickly as possible, and for reusable eviction blocks, keep those with relatively small *ERGs*.

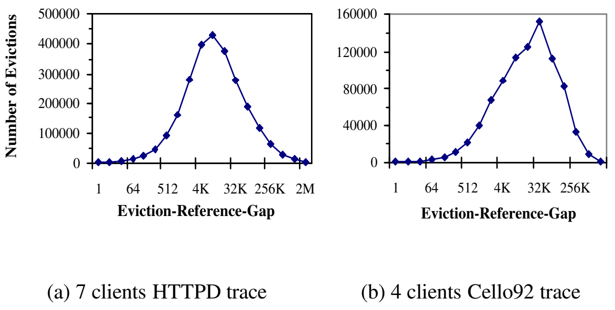


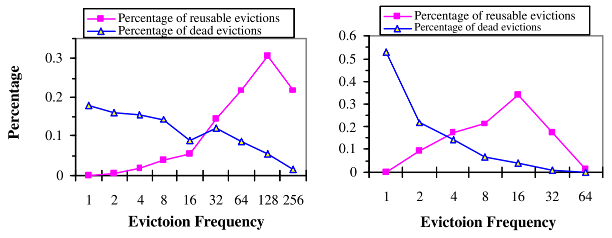
Figure 1. Eviction-Reference-Gap histograms of storage caches

We first study the *Eviction-Reference-Gap* of blocks in storage caches. The data in Fig. 1 shows the distribution of evictions over *ERGs* grouped by powers of two[1]. Significantly, blocks evicted from L1 caches are not referenced quickly: most evictions have relatively large *ERGs* (from $32K$ to $64K$ in the *Cello92* trace, and from $8K$ to $16K$ in the *HTTPD* trace). Furthermore, the curves descend slowly from peak to foot (the largest *ERG* even extends to more than $1M$), which means it is difficult for a replacement algorithm to retain most blocks before they are referenced by clients. A good replacement algorithm for storage caches should at least retain blocks that reside in the hill portion of the histogram for a longer period of time to provide more than a 50% hit ratio. Obviously, the distribution of *ERG* in Fig. 1 shows that LRU is not an appropriate local replacement algorithm for exclusive caching in an L2 cache.

Using the same traces. we have also examined the behavior of storage buffer cache accesses in terms of eviction frequency. The data in Fig. 2 shows the distribution of the percentages of reusable and dead evictions over eviction frequencies grouped by powers of two[2]. It is obvious that blocks with high eviction frequencies result in high percentage of the reusable evictions and low percentage of the dead evictions. The percentage of the dead evictions decreases with the eviction frequency, but the peak of the reusable evictions does not appear at the point of the highest eviction frequency (128 in the *HTTPD* trace and 16 in the *Cello92* trace). That does not mean that blocks with eviction frequencies higher than peak point will reduce the hit ratio, because the dead evictions of those blocks are close to zero, which means that almost all evictions of those blocks

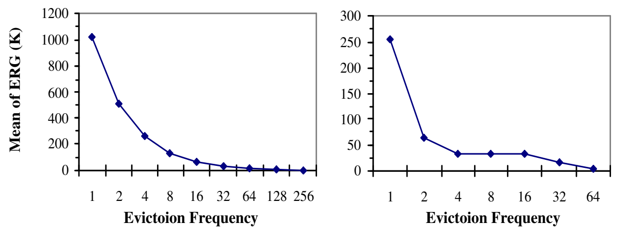
[1]ERGs that are not powers of two are rounded down to the nearest power of two.

[2]Eviction frequencies that are not powers of two are rounded down to the nearest power of two.



(a) 7 clients HTTPD trace (b) 4 clients Cello92 trace

Figure 2. Distribution of reusable and dead eviction among different eviction frequencies for different traces. A point (f, p) on the percentage of reusable (respectively, dead) evictions curve indicates that p percent of total number of reusable (respectively, dead) evictions are to blocks evicted f times.



(a) 7 clients HTTPD trace (b) 4 clients Cello92 trace

Figure 3. Change of Mean of ERGs with the eviction frequencies for the different traces.

are referenced later. Since dead evictions absolutely cause cache misses, caching blocks with high eviction frequencies is helpful for increasing cache hit ratios. We also studied how the average *ERG* distribution changes with the eviction frequency. The data in Fig. 3 shows that the blocks with higher eviction frequencies always have smaller means of *ERGs*, which indicates that those blocks have high probabilities to be hit before they must be discarded by the replacement algorithm. From Fig. 2 and Fig. 3, we conclude that higher eviction frequencies of blocks result in higher contributions to the total cache hits and lower contributions to the total cache misses.

We studied the 8-client *DB2* trace and got similar results. Since the percentage of dead evictions and the average ERGs quickly decrease with the eviction frequency, a good replacement algorithm could retain blocks with a high eviction frequency as long as possible to achieve a high hit ratio.

3 Design of uCache

The basic idea of *uCache* is based on a simple observation. In a multiple-client system, a higher correlation of workloads means that it is more likely that a block requested by one client is found in caches of other clients, because a block used by one client may have been or will be referenced by other clients within a limited time period. From this observation, the *uCache* algorithm implements exclusive caching in L2 caches for low-correlated workloads, but tries to utilize client buffer caches to improve cumulative hit ratios for high-correlated workloads.

In *uCache*, all storage client caches related to a storage server are organized as cooperative client caches [5]. A block is discarded by storage caches after it is sent back to a client, and is loaded again if evicted by that client. The storage cache space is reserved for blocks that cannot be found in the client caches. With a miss in the storage cache, a request may be redirected to an appropriate cooperative client cache if the block can be found in that client, or a hard disk action must be issued.

uCache is inherently adaptive to both low-correlated and high-correlated workloads. For low-correlated workloads, although cooperative client caches have low hit ratios, because of the small number of blocks reused among multiple clients, a high hit ratio is expected in the exclusive storage cache. For high-correlated workloads, similar to previous aggressive exclusive caching, a low hit ratio in the storage cache is predicted, but cooperative client caches provide considerable additional cache hits, according to our earlier observation. Thus the final cumulative hit ratio is still higher than the ratio for traditional inclusive caching, like LRU.

To implement the *uCache* algorithm, we consider three major issues. The first is how clients and storage collaborate to achieve exclusive caching; the second is how the storage system tracks blocks cached by cooperative client caches; and the last is how to replace blocks in storage caches. We discuss the first two issues in 3.1 and the last one in 3.2.

3.1 Collaboration between clients and the storage systems

The storage systems need to collaborate with clients to decide when to reload blocks that have been evicted by client caches, to track which blocks are cached by which clients, and to send a request to an appropriate cooperative client after an access miss in the storage cache. Actually, as long as storage systems know when a block is evicted from a client cache, they can make correct decisions for both reloading blocks and where to redirect requests. Thus, for *uCache*, one of the key design issues is to choose a mecha-

```
/* procedure to be invoked upon a reference to block b */
blockGet(block b)
{
  if b is in cache
      remove b from FIFO queue;
  check HRF;
  if eviction mark of block b was set {
      increase reference frequency of block b by one;
      clear eviction mark of block b;
  }
}

/* procedure to be invoked upon a eviction of block b */
blockPut(block b)
{
  if b is not in cache {
    check HRF;
    if an item is found for block b in HRF
       get reference frequency;
    else {
       add a new item for block b in HRF;
       set reference frequency of block b to 0;
    }
    set eviction mark for block b;
    insertintoFIFO (reference frequency, block b);
  }
}

insertintoFIFO (reference frequency, block b)
{
   insertPoint = log2(reference frequency);
   if insertPoint > m       // m is the point at tail of the queue
      insertPoint = m;
   insert block b into FIFO at insertPoint;
}
```

Figure 4. FBER algorithm

nism for storage caches to learn when a block is evicted by L1 caches.

The most intuitive way is to design a new interface between clients and storage systems to send notifications of block evictions from the L1 to the L2 caches, like the demotion operation [20]. Although this mechanism is the most accurate, client software must be modified, and network overhead between the clients and storage systems is introduced. Another possible mechanism is to guess evictions of clients from access sequences and existing interfaces, without any modification of the L1 software. *uCache* obtains L1 cache replacement information by maintaining a data structure to track client content, similar to the idea proposed in [21]. Chen *et al.* [4] concluded that the performance of the latter design is very close to the former one if appropriate local optimizations are applied. Some distributed I/O systems implement block-level cache consistency algorithms, in which storage servers track blocks cached by clients. From those systems, the *uCache* get enough L1 replacement information, thus does not need to implement the collaboration mechanism itself.

3.2 Local Replacement Algorithm

Based on the study of access patterns of exclusive caching in Section 2, we design a new replacement algorithm, called Frequency Based Eviction-Reference (*FBER*). The main idea of this algorithm is to maintain blocks with different access frequencies for different periods of time in a storage cache. According to Section 2, it is important to retain blocks with high eviction frequency as long as possible. In exclusive caching, once referenced by the L1 caches, blocks are discarded from the L2 cache spaces, so *FBER* maintains a data structure, called history reference frequency (*HRF*) table, to record past reference information of a block evicted by the L1 caches at least once. For each following reference to the block, no matter if it still stays in the cache, *FBER* increases the reference frequency of the block in *HRF*. Each time a block is evicted from the clients and reloaded into a storage cache, *FBER* checks the *HRF* according to the block number and gets the previous reference frequency, then inserts the block into a FIFO queue. The insertion point of a block is determined by its previous reference frequency: the higher the frequency, the closer to the tail of the queue, so a block with high frequency has a longer lifetime than one with low frequency. To achieve this we set m insertion points, from I_0 to I_{m-1}, for the real queue, where m is a tunable parameter. I_{m-1} is the point at the the tail of the queue, and blocks inserted at I_j have a longer lifetime in the cache than those inserted at I_i $(i < j)$. The insertion point I_k of a block is a function of the reference frequency, $insertPoint(f)$. In our current design, $insertPoint(f)$ is defined as $log_2(f)$. Our experiments also show that six insertion points are enough to separate high frequency blocks from others. Fig. 4 outline the *FBER* algorithm.

The highest cumulative hit ratio is provided by totally exclusive caching, since no blocks exist in both the clients and the storage caches, but this configuration degrades the storage hit ratio dramatically for high-correlated workloads. A small inclusive cache in storage is very helpful to increase local hit ratio, but the size of the small cache needs to be tuned carefully. *uCache* use Adaptive Space Allocation algorithm (*ASA*) to manage storage cache and provide optimal inclusive cache space dynamically. LRU algorithm is used to manage the small inclusive cache. Blocks referenced by clients are placed into the LRU cache, either from the *FBER* cache, or from hard disks because of local misses, to provide cache hits for further references. The size of the small LRU cache is determined dynamically by its hit ratio. One hit of the LRU cache will increase its size by one block, and one hit of the *FBER* cache will shrink its size by one block. Since the highest cumulative hit ratio is provided by total exclusiveness, a ghost cache which simulates a totally exclusive storage cache is implemented to provide a reference for each moment of accesses. If the current cumulative hit ratio is too low compared to that of the ghost cache, the LRU cache size will be reduced. The *ASA* algorithm tries to maximum local hit ratio while not sacrifice cumulative hit ratio too much.

Table 2. Access times for different levels in the cache hierarchy

Storage Cache	Remote Client Caches	Storage Disk
250us	360us	9,500us

4 Simulation Methodology

We compare cumulative L2 cache hit ratios and average response times of *uCache* (implementing *HBER* and *ASA* for storage cache) and other algorithms, including LRU, 2Q [11], exclusive caching [20], and SLRU [20].

We use trace-driven simulation to evaluate cumulative hit ratios. We have developed a simulator to simulate two-level buffer cache hierarchies with multiple clients and one storage system. LRU is used as the replacement algorithm in the L1 caches, and multiple algorithms mentioned before are implemented in the L2 cache. Thus in our simulations, when refer to LRU, we talk about *LRU-LRU* (L1-L2 caches). We assume a cache block size of $4KB$. The traces we used for the simulator are described in Section 2.1.

The following formula describes the calculation of the average response times for the L2 caches.

$$T_{mean} = T_s * h_s + T_r * h_r + T_d * miss$$

T_s and T_r are costs of hits in the storage cache and the remote cooperative client caches, respectively. T_d is the cost of reading a block from a storage disk. h_s and h_r are the hit ratios (output by our simulator) of the storage cache and the remote cooperative client caches, respectively, and $miss = 1 - (h_s + h_r)$.

We have designed a program to compute average value of T_s, T_r, and T_d for a $4KB$ block in our lab. The storage server is a *Dell PowerEdge 2500*, with a $1.4GHz$ *Intel Xeon* microprocessor, $1024MB$ memory, and a *Dell PercRaid Raid5* $54.5G$ Disk. The client is a *Dell Dimension 4500*, with a $2.4GHz$ *Intel Pentium-4* microprocessor, $256M$ memory and a $40G$ IDE disk. All machines are equipped with a 32 bit *PCI 100/1000Mbps* network interface card, and connected through a *Dell PowerConnect 5224* Gigabit Ethernet switch. RedHat 9.1 is installed on each machine, with Linux kernel $2.4.20 - 8$. For each access time, we performed 100 experiments and calculated the average value. The results are summarized in Table 2. Note that we do not include any queuing delays in our response time figures. Since the *uCache* algorithm reduces server loads by directing parts of requests to other machines, and popular high performance networks use a switched topology, we do not expect queuing to alter our results significantly.

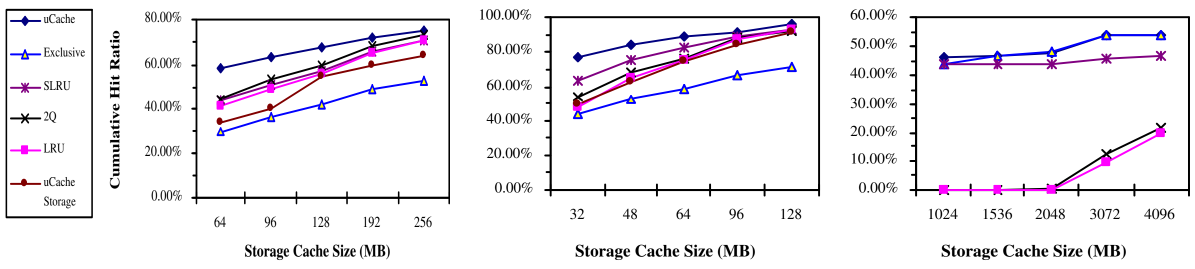
5 Simulation Results

5.1 Low-correlated traces

We use the *DB2* trace as a multiple-client low-correlated workload. Fig. 5(c) shows that *uCache* provides the best hit ratio among all the algorithms. Since no blocks are shared among the eight clients, the additional hit ratio for cooperative caching is zero. The temporal locality of the *DB2* workload is very weak, so the LRU and 2Q algorithms provide very low cache hit ratios, even when the storage cache size increases to $4096MB$. SLRU is much better than LRU, but still lags behind exclusive caching and uCache, because it is designed to be compatible with high-correlated workloads by not completely implementing exclusiveness in the L2 cache. The difference between *uCache* and exclusive caching is not obvious, because the highest eviction frequency of the *DB2* trace is only four, which is not enough for *FBER* to utilize. The *ASA* algorithm successfully allocates all storage cache space to *FBER*, since there are almost no blocks reused among different clients. Fig. 6(c) shows that the average response time follows the same trend of the hit ratio. The biggest improvement from LRU to *uCache* is 46%, with a $1024MB$ storage cache.

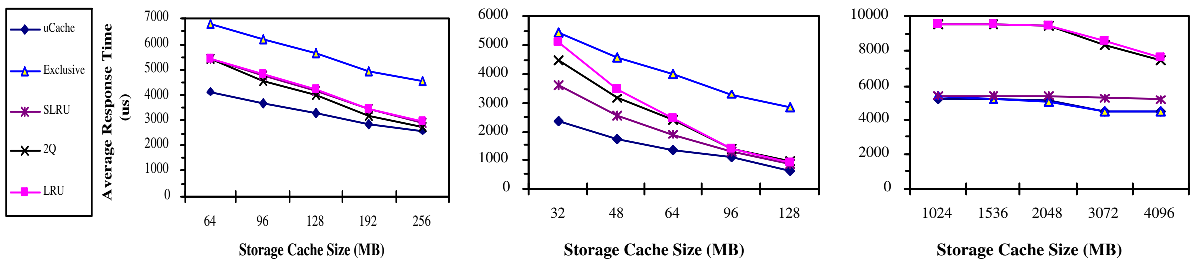
5.2 High-correlated traces

We use the *Cello92* trace and the *HTTPD* trace as multiple-client high-correlated workloads. Fig. 5(a) and Fig. 5(b) shows the hit ratios of different algorithms. The *uCache* always provides the best hit ratio among all the algorithms. LRU provides a relatively high hit ratio because each block in an LRU cache has a long life before it is discarded, and thus has a high possibility to be referenced again and again by different clients with high-correlated workloads. The gain becomes smaller as the storage cache is larger, since a large cache size retains a block for a long enough time, within which it is accessed by most clients. Exclusive caching suffers serious performance degradation even compared to LRU, because discarding a block immediately after it is referenced once causes many cache misses for following references from other clients. We notice that even the storage hit ratios of *uCache*, which does not count the benefits from the cooperative client caches, are much higher than exclusive caching and is very close to the result of three inclusive cache algorithms. The *ASA* algorithm works perfectly to both increase local hits and maintain high cumulative hit ratios. Fig. 6(a) and Fig. 6(b) show that the average response time follows the same trend as the hit ratio. The biggest improvement from LRU to *uCache* is 53%, with a $32MB$ storage cache for the 7-client *HTTPD* trace.



(a) 4 clients Cello92 trace (b) 7 clients HTTPD trace (c) 8 clients DB2 trace

Figure 5. L2 cache hit ratios of various clients under different traces. Hit ratios of uCache Storage does not include hits from cooperative client caches



(a) 4 clients Cello92 trace (b) 7 clients HTTPD trace (c) 8 clients DB2 trace

Figure 6. Average response time of various clients under different traces.

6 Related Work

Muntz and Honeyman [14] and Froese and Bunt [8] showed that L2 caches have poor hit ratios. Further studies show that the poor hit ratio is caused by both weaker temporal locality [3, 21] and duplicated blocks [20]. After studying behavior of NFS servers, Reed and Long [16] found that LRU algorithm may still exploit temporal locality caused by frequent accesses of file system metadata. Many new algorithms have been proposed recently to improve cumulative hit ratios, such as MQ [21], Demotion-based algorithm [20], Global L2 buffer cache management [21], X-Ray [2], and client-controlled cache replacement [10]. Chen *et al.* [4] classified all those algorithms into two types: hierarchy-aware caching, and aggressively-collaborative caching, and compared the performance among typical algorithms belonging to the two types. Ari *et al.* proposed ACME [1] to adaptively select the best replacement policy for each cache-level to achieve high accumulative hit ratios. Our work in multi-level cache hierarchies builds upon but is different from previous studies because the *uCache* algorithm is adaptive to multiple-client systems, with either high-correlated workloads or low-correlated workloads.

Researchers have used metrics such as reuse distance [21], inter reference gap [15], and inter reference recency [9] to analyze access patterns of workloads, but none of them studies the characteristics of reference streams of L2 caches in exclusive caching. Our study shows that the Eviction-Reference Gap is very large and high eviction frequency blocks contribute most to cache hits in exclusive caching. Based on our study, we propose a new algorithm, Frequency Based Eviction-Reference (*FBER*), to improve hit ratios for exclusive caching.

Researchers have considered using cooperative client caching to improve cumulative hit ratios in multi-level cache hierarchies. Dahlin *et al.* [5], proposed four representative cooperative caching algorithms and demonstrated that N-Chance Forwarding can provide the best performance. GMS [7] is more general than N-chance in that it is a distributed shared-memory system, for which cooperative caching is only one possible use. Sarkar et al [18], introduced a hint-based algorithm to reduce overhead of cooperative caches. Our work is related to but different from those previous algorithms, because we use exclusive caching in storage caches to improve hit ratios for low-correlated workloads, while using cooperative client caching to cache blocks reused frequently among clients in high-correlated workloads.

7 Conclusions

In this paper, we propose a new unified buffer cache management algorithm: *uCache*, to improve performance of L2 caches in multi-level cache hierarchies, in multiple client environments. uCache combines both exclusive caching in storage caches to improve hit ratios for low-correlated workloads, and cooperative client caching to improve hit ratios for high-correlated workloads.

We have studied the characteristics of reference streams of exclusive caching. Our results show that the average Eviction-Reference Gap of exclusive caching with multiple clients is very large in that it is difficult for a replacement algorithm utilizing temporal locality of workloads to provide high hit ratios. A frequency based algorithm is highly preferred because high eviction frequency blocks contribute the most to cache hits but cause the least cache misses in exclusive caching. Based on the study, we propose a new local replacement algorithm, Frequency Based Eviction-Reference (*FBER*), and Adaptive Space Allocation (*ASA*), to improve the hit ratios of exclusive caching.

We have evaluated our *uCache* algorithm and other typical multi-level caching algorithms using simulations under both high-correlated and low-correlated workloads. The results show that *uCache* can dramatically increase the cumulative cache hit ratio over LRU and improve the average I/O response time by up to 46% for low-correlated workloads and 53% for high-correlated workloads.

References

[1] I. Ari, A. Amer, R. Gramacy, E. L. Miller, S. A. Brandt, and D. E. Long. ACME: Adaptive caching using multiple experts. In *Proc. in Informatics*, volume 14, page 14158, 2002.

[2] L. N. Bairavasundaram, M. Sivathanu, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. X-RAY: A non-invasive exclusive caching mechanism for RAIDs. In *Proc. 31th Annual International symposium on Computer Architecture*, pages 176–187, June 2004.

[3] R. B. Bunt, D. L. Willick, and D. L. Eager. Disk cache replacement policies for network file servers. In *Proc. IEEE International Conference on Distributed Computing Systems-ICDCS '93*, pages 2–11, June 1993.

[4] Z. Chen, Y. Zhang, and Y. Zhou. Empirical evaluation of multi-level buffer cache collaboration for storage systems. In *ACM SIGMETRICS*, 2005.

[5] M. Dahlin, R. Wang, T. Anderson, and S. Patterson. Cooperative Caching: Using remote client memory to improve file system performance. *Operating Systems Design and Implementation*, 1994.

[6] A. Dan and D. Towsley. An approximate analysis of the LRU and FIFO buffer replacement schemes. In *ACM SIGMETRICS*, pages 143–152, May 1990.

[7] M. J. Feeley, W. E. Morgan, F. H. Pighin, A. R. Karlin, H. M. Levy, and C. A. Thekkath. Implementing global memory management in a workstation cluster. In *Proc. Symp. Operating Systems Principles*, 1995.

[8] K. Froese and R. B. Bunt. The effect of client caching on file server workloads. In *Proc. 29th Hawaii International Conference of System Sciences*, January 1996.

[9] S. Jiang and X. Zhang. LIRS: An efficient low inter-reference recency set replacement policy to improve buffer cache performance. In *Proc. ACM SIGMETRICS*, pages 31–42, 2002.

[10] 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*, Mar 2004.

[11] T. Johnson and D. Shasha. 2Q: A low overhead high performance buffer management replacement algorithm. In *Proc. Twentieth International Conference on Very Large Databases*, pages 439–450, 1995.

[12] E. D. Katz, M. Butler, and R. McGrath. A scalable HTTP server: The NCSA prototype. *Computer networks and ISDN systems*, 27(2):155–164, Nov 1994.

[13] N. Megiddo and D. Modha. ARC: A self-tuning, low overhead replacement cache. In *Proc. Second USENIX Conf. File and Storage Technologies*, 2003.

[14] D. Muntz and P. Honeyman. Multi-level caching in distributed file systems-or-your cache ain't nuthin' but trash. In *Proc. Usenix Winter Technical Conf.*, pages 305–314, 1992.

[15] V. Phalke and B. Gopinath. An inter-reference gap model for temporal locality in program behavior. In *Proc. Joint Int.l Conf. Measurement and Modeling of Computer Systems*, pages 291–300, May 1995.

[16] B. Reed and D. E. Long. Analysis of caching algorithms for distributed file systems. *ACM SIGOPS Operating Systems Review*, 30(3):12–21, 1996.

[17] C. Ruemmler and J. Wilkes. Unix disk access patterns. In *Proc. Winter 1993 USENIX Conf.*

[18] P. Sarkar and J. Hartman. Efficient cooperative caching using hints. In *Proc. Second ACM Symp. Operating Systems Design and Implementation*, 1996.

[19] M. Uysal, A. Acharya, and J. Saltz. Requirements of I/O systems for parallel machines: An application-driven study. Technical Report CS-TR-3802, Dept. of Computer Science, University of Maryland, May 1997.

[20] T. Wong and J. Wilkes. My cache or yours? Making storage more exclusive. In *Proc. USENIX Ann. Technical Conf.*, 2002.

[21] Y. Zhou, Z. Chen, and K. Li. Second-level buffer cache management. *IEEE Transactions on Parallel Distributed Systems*, July 2004.