A Unified Multiple-Level Cache for High Performance Storage Systems

Xubin (Ben) He, Li Ou
Department of Electrical and Computer Engineering
Tennessee Technological University
E-mail: {hexb, lou21}@tntech.edu

Martha J. Kosa
Department of Computer Science
Tennessee Technological University
E-mail: mjkosa@tntech.edu

Stephen L. Scott, Christian Engelmann
Computer Science and Mathematics Division
Oak Ridge National Laboratory
E-mail: {scottsl, engelmannc}@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, such as LRU, the I/O response time is improved by up to 46% for *low-correlated* workloads and 53% for *high-correlated* workloads.

Keywords: Cooperative Cache; Multi-level Cache; Distributed I/O; Storage Systems

Reference to this paper should be: He, X., Ou, L., Kosa,M.J., Scott,S.L., Engelmann, C.(2006) 'A Unified Multiple-Level Cache for High Performance Storage Systems', *Int. J. High Performance Computing and Networking*, Vol. 4, No. 5.

Biographical notes: X. He is an assistant professor of electrical and computer engineering at Tennessee Technological University. He received the PhD degree in electrical engineering from the University of Rhode Island in 2002. His current research interests include computer architecture, storage and I/O systems, and performance evaluation. He is a member of the IEEE Computer Society, Sigma Xi, and ASEE.
L. Ou is a Ph.D candidate of electrical and computer engineering at the Tennessee Technological University. His current research interests include computer architecture, storage and I/O systems.
M. J. Kosa is an associate professor of computer science at Tennessee Technological University. She received the PhD degree in computer science from the University of North Carolina at Chapel Hill in 1994. Her research interests include distributed algorithms and computer science education. She is a member of ACM and MAA.
S. L. Scott is a Senior Research Scientist at the Oak Ridge National Laboratory. His research interest is in experimental systems with a focus on high performance distributed, heterogeneous, and parallel computing. He received his Ph.D. in computer science from Kent State University in 1996. He is a member of ACM, IEEE Computer, and the IEEE Task Force on Cluster Computing.
C. Engelmann is a research staff member at Oak Ridge National Laboratory. He is currently a PhD student at the University of Reading. His research interests include high availability for scientific high-end computing, efficient fault tolerance for extreme-scale systems and flexible, pluggable, component-based runtime environments. He is a member of the IEEE Computer Society and ACM.

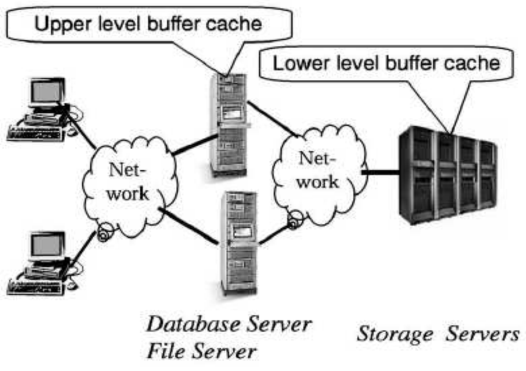


Figure 1: Multi-level buffer cache hierarchy.

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 (Dan and Towsley, 1990), MRU (Denning, 1968), LFU, FBR (Robinson and Devarakonda, 1990), LRU-k (O'Neil et al., 1993), 2Q (Johnson and Shasha, 1995), LIRS (Jiang and Zhang, 2002), and ARC (Megiddo and Modha, 2003). 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, shown in Fig. 1 (Zhou et al., 2004), 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 (Zhou et al., 2004). 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 (Bunt et al., 1993; Froese and Bunt, 1996; Zhou et al., 2004) 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 (Wong and Wilkes, 2002), 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 the cache performance of multi-level buffer caches for distributed I/O systems. Recent research (Wong and Wilkes, 2002; Zhou et al., 2004; Chen et al., 2005; Bairavasundaram et al., 2004; Jiang and Zhang, 2004) 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 (Zhou et al., 2004), all the other algorithms try to achieve exclusive caching (Wong and Wilkes, 2002) through quick eviction of duplicated blocks in L2 caches. Implementing aggressive exclusive caching may get a high hit ratio in the 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 near 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 (Wong and Wilkes, 2002). Nearly disjoint workloads are *low-correlated* workloads, and nearly conjoint workloads are *high-correlated.* For low-correlated workloads, aggressive exclusive caching is effective, 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. For example, the simulation results in Wong and Wilkes (2002) show that the exclusive caching could achieve a 1.50 speedup over LRU for low-correlated workloads, but suffers a 0.55 slowdown for high-correlated workloads. 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 Wilkes (2002) 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 (Dahlin et al., 1994) 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 in 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 (Wong and Wilkes, 2002; Zhou et al., 2004), 2Q (Johnson and Shasha, 1995), and SLRU (Wong and Wilkes, 2002), 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. The background is presented in Section 2. Section 3 discusses access patterns of L2 caches in exclusive caching environments. Section 4 describes our idea and design issues in detail. Section 5 describes our simulation methodology. We compare our work to previous efforts to improve L2 cache performance in Section 6 and examine related work in Section 7. We draw our conclusions in Section 8.

2 Background Review

To improve the hit ratio of buffer caches, researchers have proposed many management algorithms, such as LRU (Dan and Towsley, 1990), MRU (Denning, 1968), LFU, FBR (Robinson and Devarakonda, 1990), LRU-k (O'Neil et al., 1993), 2Q (Johnson and Shasha, 1995), LIRS (Jiang and Zhang, 2002), ARC (Megiddo and Modha, 2003), Cooperative caching (Dahlin et al., 1994; Sarkar and Hartman, 1996), and the Exclusive caching algorithm (Wong and Wilkes, 2002; Zhou et al., 2004). We outline three typical algorithms related to our design below.

2.1 LRU Cache Algorithm

The Least Recently Used (LRU) policy is one of the most effective policies for memory caching. Many current implementations of cache management algorithms also use variants of the LRU policy. The idea of LRU is simple: a block which is least recently used should be the best candidate to be evicted from the cache, if a new block needs to be inserted. In the LRU policy, a block is tagged with a priority measure that is equal to the time elapsed since the block was last accessed. When space needs to be created in the cache, the oldest block, i.e., the one that has been accessed least recently, is removed.

2.2 Exclusive Cache Algorithms

Recent studies (Bunt et al., 1993; Zhou et al., 2004) show that weak temporal locality of L2 cache accesses causes a low hit ratio for the traditional LRU algorithm. Traditional L2 cache algorithms are inclusive (Wong and Wilkes, 2002), which means the same blocks are cached by both the L1 and L2 caches at the same time. Thus duplicated blocks waste aggregate cache space. In exclusive caching, a block is discarded from the L2 caches some time after it is sent back to the L1 caches. If the same block is evicted from the L1 caches, the L2 caches load it again for the next possible access. Exclusive caching algorithms achieve higher hit ratios compared to traditional inclusive caching techniques (i.e., LRU), in single client storage systems, or multiple-client systems with low-correlated workloads. However, they suffer performance degradation in multiple-client systems with high-correlated workloads, because blocks may be referenced again by other storage clients within a limited time after they are sent back to individual clients.

2.3 Cooperative Cache Algorithms

Cooperative cache algorithms (Dahlin et al., 1994) are used to improve the overall cache hit ratio by taking advantage of cache space in client machines.

When a client request is missed in the storage server cache, the traditional way to service the request is to access hard disks. Since the storage server is shared by multiple clients, there is a high probability that the blocks requested by one client and missed in the server cache are kept by other clients. So in cooperative caching, the storage server tracks the blocks cached in each client, and directs a request to a client if there is a cache miss in the server and the corresponding block can be found in that client.

3 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 those of traditional caches. The access sequences of traditional caches consist of continuous references of blocks, and researchers use metrics, such as *reuse distance* (Zhou et al., 2004), *inter-reference gap* (Phalke and Gopinath, 1995), and *inter-reference recency* (Jiang and Zhang, 2002), to describe characteristics of workloads, which are then used to design replacement algorithms to manage buffer caches. In exclusive caching, access sequences of storage caches consist of two types of randomly interleaved operations: *evictions*, which inform storage systems to reload blocks that have been replaced by client caches, and *references*, such as reads or writes, provided by standard I/O interfaces. With these differences, we need to analyze the access patterns of exclusive caching, and design a replacement algorithm based on those patterns.

3.1 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 the traces.

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

The *HP Cello92* trace was collected at *Hewlett-Packard* Laboratories in 1992 (Ruemmler and Wilkes, 1993). 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 $64MHz$ 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 (Katz et al., 1994) 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. (1997) used this trace in their study of I/O on parallel machines. 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 the *HP 9000/877* server is only $10-30MB$, which is very small by current standards. 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 a L1 hit ratio of no more than 15%. But if the cache size increases to $600MB$, the L1 hit ratio suddently increases to 75%, because *reuse distances* (Zhou et al., 2004) of most blocks are less than $150K$. 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.

3.2 Access patterns of exclusive caching

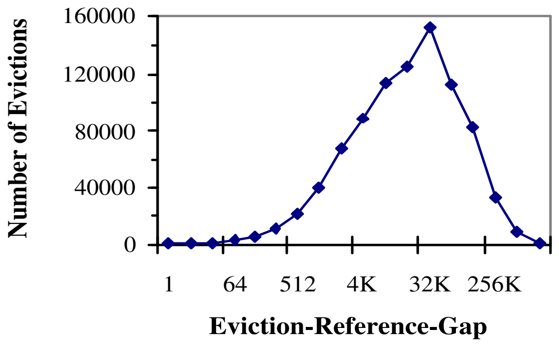
Because of the uniqueness of the reference sequences, the metrics used before may not correctly describe the characteristics of access patterns for exclusive caching. Thus, we need to define new metrics to describe the access pattern. The *Eviction-Reference-Gap (ERG)* indicates the distance (the number 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.*

We first study the *Eviction-Reference-Gap* of blocks in storage caches. The data in Fig. 2 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, from $8K$ to $16K$ in the *HTTPD* trace, and $64K$ in the *DB2* 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. 2 shows that LRU is not an appropriate local replacement algorithm for exclusive caching in an L2 cache.

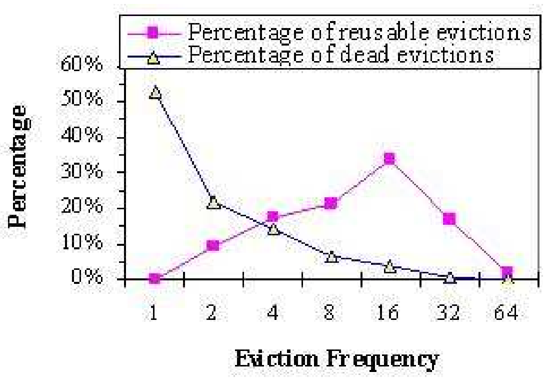
Using the same traces, we have also examined the behavior of storage buffer cache accesses in terms of eviction frequency. The data in Fig. 3 shows the distribution of the percentages of reusable and dead evictions over eviction frequencies grouped by powers of two[2]. It is obvious that blocks with high eviction frequencies result in a high percentage of the reusable evictions and a 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 (16 in the *Cello92* trace, 128 in the *HTTPD* trace, and 2 in the *DB2* trace). That

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

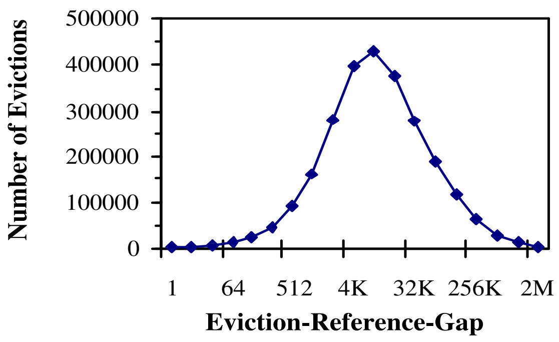
[2]Eviction frequencies that are not powers of two are rounded down to the nearest power of two.



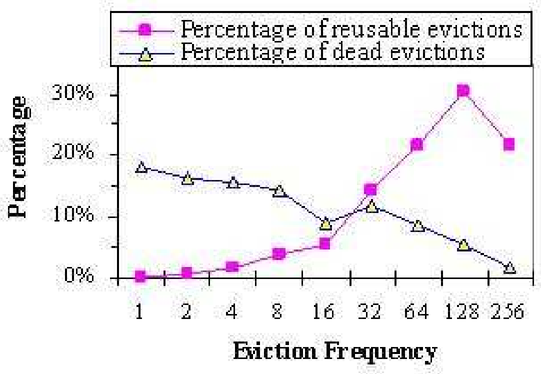
(a) 4 clients under Cello92 trace



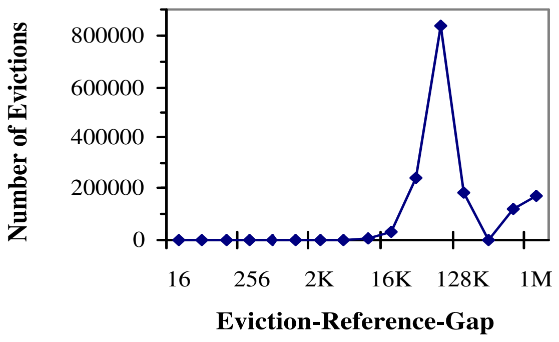
(a) 4 clients under Cello92 trace



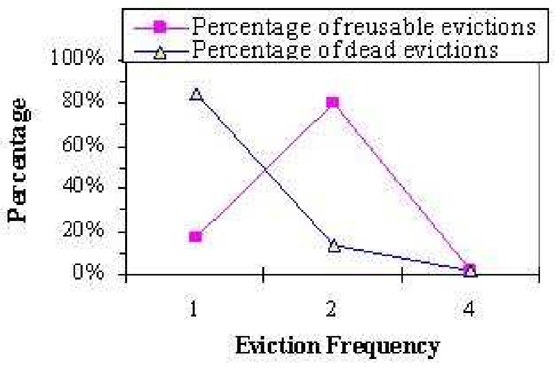
(b) 7 clients under HTTPD trace



(b) 7 clients under HTTPD trace



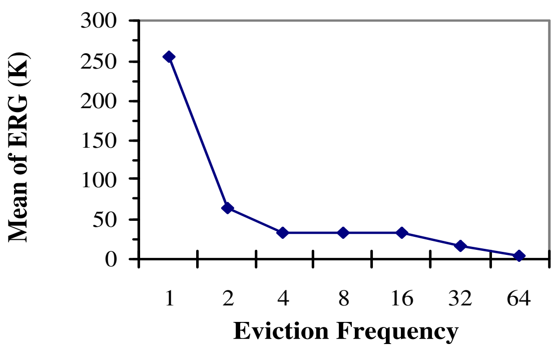
(c) 8 clients under DB2 trace



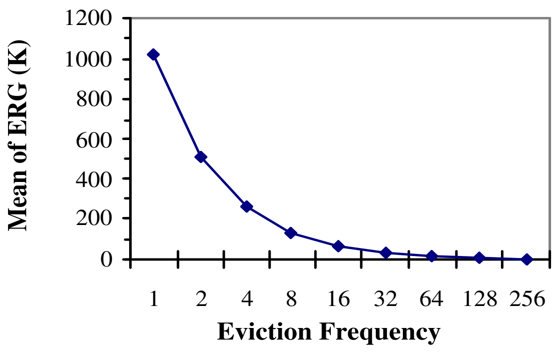
(c) 8 clients under DB2 trace

Figure 2: Eviction-Reference-Gap.

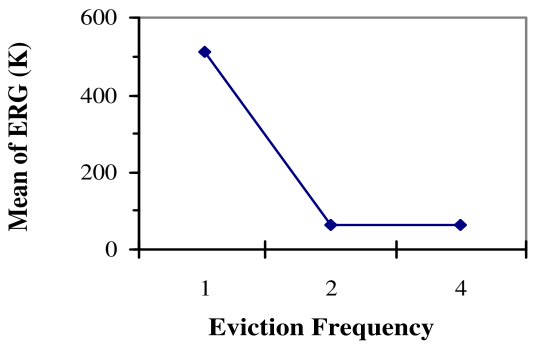
Figure 3: Reusable and dead eviction distribution.



(a) 4 clients under Cello92 trace



(b) 7 clients under HTTPD trace



(c) 8 clients under DB2 trace

Figure 4: Mean of ERGs.

does not mean that blocks with eviction frequencies higher than the peak point will reduce the hit ratio, because the number of dead evictions of those blocks are close to zero, which means that almost all of those blocks are referenced later. Since dead evictions absolutely cause cache misses, caching blocks with high eviction frequencies is helpful for increasing hit ratios. The *DB2* trace shows very low temporal locality, so the highest eviction frequency is only four. We also studied how the average *ERG* distribution changes with the eviction frequency. The data in Fig. 4 shows that the blocks with higher eviction frequencies always have smaller mean *ERGs*, which indicates that those blocks have high hit probabilities before they are discarded by the replacement algorithm. From Fig. 3 and Fig. 4, 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.

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.

4 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 (Dahlin et al., 1994). A block is discarded by storage caches after it is sent back to a client, and is loaded again if evicted by that client. 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. Fig. 5 briefly outlines the uCache algorithm.

uCache is inherently adaptive to both low-correlated and high-correlated workloads. For low-correlated workloads, 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. The final cumulative hit ratio is still higher than the ratio for traditional inclusive caching, such as 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 the blocks cached by the cooperative client caches; and the last is how to replace the blocks in the storage caches. We discuss the first two issues in Section 4.1 and the last one in Section 4.2.

4.1 Collaboration between clients and the storage systems

Since the storage caches are exclusive to the client side, the storage systems need to collaborate with clients to decide when to reload blocks that have been evicted by client caches. On the other side, all client caches are cooperative and it is the storage system's responsibility to redirect requests to clients, so storage systems also need 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 when to reload blocks and where to redirect requests, because with accurate information of load and evictions of blocks, the storage systems can track the content of client caches without problems. Thus, for *uCache*, one of the key design issues is to choose a mechanism for storage caches to learn when a block is evicted by L1 caches.

Unfortunately, with traditional I/O interfaces between clients and storages, only the block loading information is transparent to storages, but not evictions. 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 (Wong and Wilkes, 2002). Although this mechanism is the most accurate, client software must be modified, and network overhead between the clients and storage systems increases. 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 Zhou et al. (2004). Chen et al. (2005) 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, *uCache* gets enough L1 replacement information; thus it does not need to implement the collaboration mechanism.

4.2 Local Replacement Algorithm

Based on the study of access patterns of exclusive caching in Section 2.3, 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 2.3, it is important to retain blocks with high eviction frequencies as long as possible. In exclusive caching, once referenced by the L1

```
/* procedure to be invoked upon a reference to
block b */

blockGet (block b)
{
if block b is available {
   blockGet (block b);  /* call of ASA */
   send back to the client;
 }
else {
  if block b is in a client cache
     redirect request to the client;
  else
     load block b from hard disks;
 }
}

/* procedure to be invoked upon a eviction of
block b */

 blockPut (block b);  /* call of ASA */
```

Figure 5: uCache algorithm.

```
/* 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 6: FBER algorithm.

```
/* procedure to be invoked upon a reference to block b */

blockGet(block b)
{
  if b is in FBER cache {
    blockGet (b);      /* call of FBER */
    increase size of FBER cache by 1;
  }
  else if  b is in LRU cache {
    remove b from LRU cache;
    increase size of LRU cache by 1;
  }
  else
     read b from disk;
  put b to tail of LRU queue;
}

/* procedure to be invoked upon every 1000 references to
cache */

adjustcache()
{
   calculate cumulative hit ratio of real cache and ghost;
   if hit ratio of ghost is much larger than real cache
       decrease size of LRU by n; // n is a tunable parameter.
}

/* procedure to be invoked upon a eviction of block b */

blockPut(block b)
{
  blockPut (b);     /* call of FBER */
  put to ghost cache;
}
```

Figure 7: ASA algorithm.

caches, blocks are discarded from the L2 caches. Thus *FBER* maintains a data structure, called the history reference frequency (*HRF*) table, to record past reference information of a block evicted at least once by the L1 caches. 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 the *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 it 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. 6 outlines 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 the local hit ratio, but the size of the small cache needs to be tuned carefully. *uCache* uses the Adaptive Space Allocation algorithm (*ASA*) to manage the storage cache and provide optimal inclusive cache space dynamically. The LRU algorithm is used to manage the small inclusive cache. Blocks referenced by clients recently are placed into the LRU cache, either from the *FBER* cache, or from hard disks because of local misses, to provide cache hits for future 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 access. 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 maximize the local hit ratio while not sacrificing the cumulative hit ratio too much. Fig. 7 outlines the ASA algorithm.

5 Simulation Methodology

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

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 the aforementioned algorithms are implemented in the L2 cache. Thus in our simulations, when we refer to LRU, we talk about *LRU-LRU* (L1-L2 caches). Since the block size of the machines where the three traces are collected is 4KB, we also assume a cache block size in our simulation of $4KB$. We have examined other block sizes, with similar results. The traces we used for the simulator are described in Section 3.1.

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 the average values of T_s, T_r, and T_d for a $4KB$ block. 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 con-

Table 2: Access times for different level caches

	Storage Cache	Remote Cooperative Client Caches	Storage Disk
Gigabit Ethernet	250us	380us	9,500us
150Mb ATM	1,050us	1,350us	10,500us
10Mb Ethernet	6,900us	7,200us	16,150us

nected 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. Actually, the access time of remote cooperative client caches includes the check time of storage caches, the network overhead among storage and clients, and the service time of the client caches, because the requests are still first checked at storage side, and after a cache miss occurs, they are then sent to clients.

We designed the *ASA* and *FBER* algorithms to improve the local hit ratios of *uCache*, but part of the hit ratio increase still comes from remote cooperative client caches, so the final average response time is sensitive to network latency and host processing speed. To study how the network environment can influence the final result, we also provide average values of T_s, T_r, and T_d in some slower networks. These data are derived from the original study on cooperative caching by Dahlin et al. (1994). Although these measurements are now a few years old and thus likely to be slow when compared to state-of-art equipment, our purpose is to show how *uCache* can tolerate severe environments, since it is obvious that *uCache* works better in high speed networks. In Table 2, the response time of Gigabit Ethernet comes from our measurements, and others are derived from Dahlin et al. (1994). The average response times of the L2 caches for all of the following simulations are calculated from measurements of the Gigabit Ethernet environment, except those in Section 6.4, where we intentionally examine the sensitivity of *uCache* to technology changes.

6 Simulation Results

6.1 Single Client

We use the *Cello92* trace as the workload of a single client system. Since there is only one client, *uCache* works the same as previous exclusive caching algorithms, except that *FBER* and *ASA* are used as the local replacement algorithms in the L2 cache.

Fig. 8 shows that the *uCache* always provides the best hit ratio among all the algorithms. It provides about a 25% higher hit ratio in a 64MB cache and 15% in a 128MB cache, compared to LRU. When the cache size inceases to 128MB, the difference is not obvious, because 128MB is large enough to hold most blocks. LRU is the worst one, because of its inclusiveness. 2Q provides some gain because it evicts cold blocks as soon as possible. *uCache* and

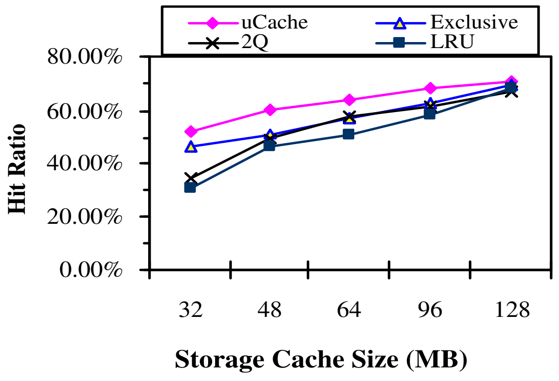


Figure 8: L2 cache hit ratio of single client.

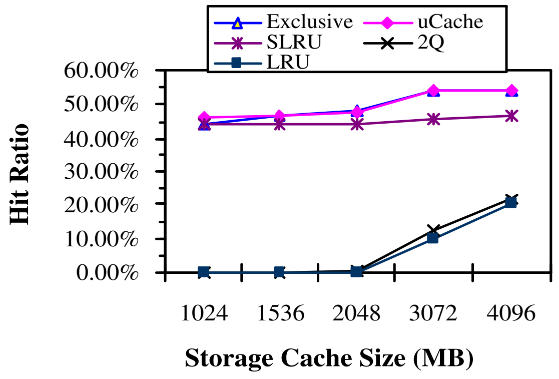


Figure 9: L2 cache hit ratios of 8 clients (DB2 trace).

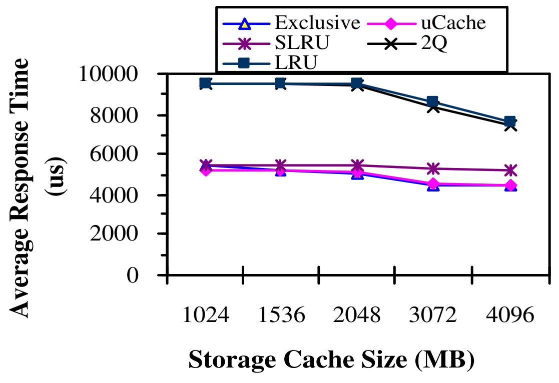


Figure 10: L2 cache response time of 8 clients (DB2 trace).

exclusive caching provide the best results, since duplicated blocks are removed quickly from L2 caches, and the aggregate cache space is utilized efficiently. *uCache* has an obvious improvement, even compared to exclusive caching, because the *FBER* algorithm prefers to retain blocks with high eviction frequencies, which contribute most of the cache hits. The *ASA* algorithm aggressivelly shrinks the small LRU cache to provide all the cache space to *FBER*.

6.2 Low-correlated traces

We use the *DB2* trace as a multiple-client low-correlated workload. Fig. 9 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 almost no blocks are reused among different clients. Fig. 10 shows that the average response time follows the same trend as the hit ratio. The biggest improvement from LRU to *uCache* is 46%, with a $1024MB$ storage cache.

6.3 High-correlated traces

We use the *Cello92* trace and the *HTTPD* trace as multiple-client high-correlated workloads. Fig. 11 shows the hit ratios of the different algorithms under various configurations. 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 by different clients with high-correlated workloads. The gain of *uCache* becomes smaller as the storage cache grows larger, since a large cache size retains a block for a long enough time, to be accessed by most clients. Exclusive caching suffers serious performance degradation, even when compared to LRU, because discarding a block immediately after it is referenced once causes many cache misses for successive 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 those of exclusive caching and are very close to the results of the three inclusive cache algorithms. The *ASA* and *FBER* algorithms work perfectly to both increase local hits and maintain high cumulative hit ratios.

ASA and *FBER* balance well between the local storage hit ratio and the cumulative hit ratio. Fig. 12 compares the cumulative and storage hit ratios of LRU, exclusive caching, and *uCache* under various configurations. We intentionally change the replacement algorithms for exclusive caching and LRU to add cooperative client caches and provide the cumulative hit ratios. We have mentioned before that entirely exclusive caching in storage with cooperative client caches provides the maximum cumulative hit ratio, but very low storage hit ratios in high-correlated workloads. When compared with exclusive caching, we find that the cumulative hit ratio of *uCache* is almost the same, while the local storage hit ratio is much higher. We also find that even with cooperative client caches, LRU cannot provide satisfactory cumulative cache hits, because most blocks in the storage cache and cooperative client caches are the same, and aggregate cache space is wasted. With the *ASA* algorithm, *uCache* provides both satisfactory local hit ratios, almost the same as typical inclusive caching, and high cumulative hit ratios, very close to the maximum values that totally exclusive caching algorithm can reach.

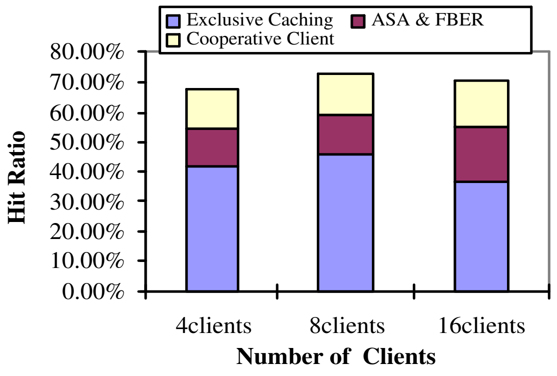


Figure 14: Cumulative hit ratio of L2 cache (Cello92 trace).

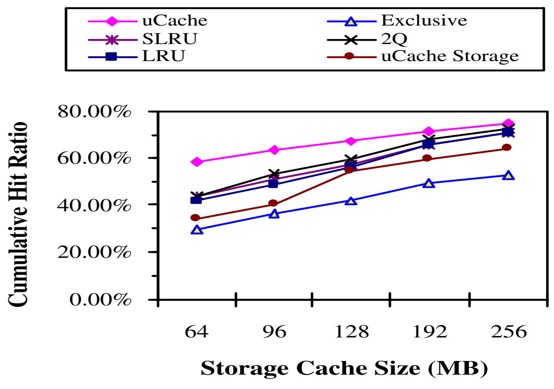
Fig. 13 shows that the average response time follows the same trend as the hit ratio. The biggest improvement is 53% from LRU to *uCache*, with a $32MB$ storage cache for the 7-client *HTTPD* trace.

Fig. 14 uses the *Cello92* trace as an example to show the source of the gain of *uCache*. The base segments are the hit ratios that a pure exclusive cache can provide. We find that *ASA* and *FBER* contribute a lot to the increase of the local hit ratios. Finally, cooperative client caches provide additional cumulative hits.

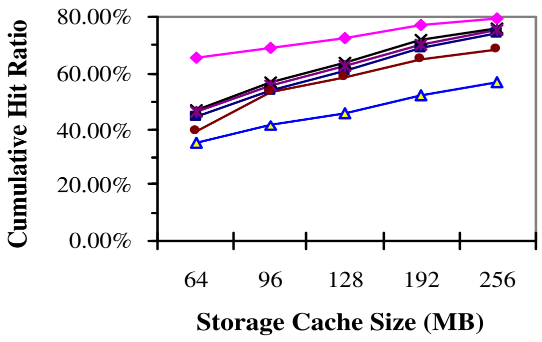
6.4 Sensitivity

To study how the performance of *uCache* is affected by different network technologies with high-correlated workloads (the gains with single client and low-correlated workloads are not from remote cooperative caches), we use the parameters collected in Section 4.2 to calculate the average response times under various environments with the cache hit ratios from the previous simulations.

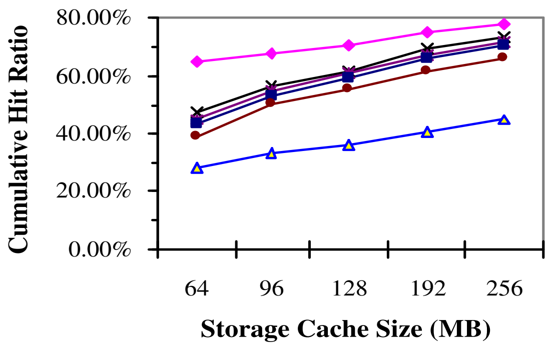
Fig. 15 indicates how the improvement of *uCache* over LRU varies with different network technologies. With com-



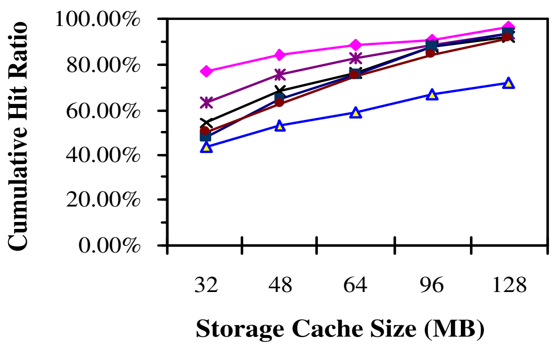
(a) 4 clients under Cello92 trace



(b) 8 clients under Cello92 trace

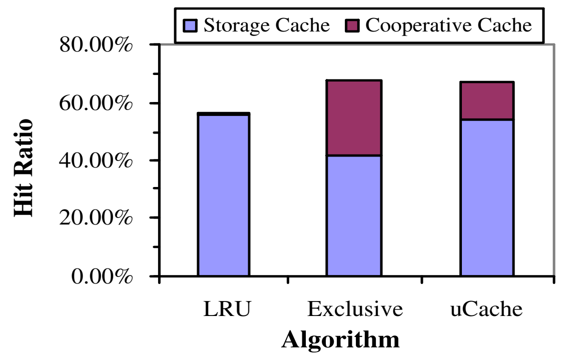


(c) 16 clients under Cello92 trace

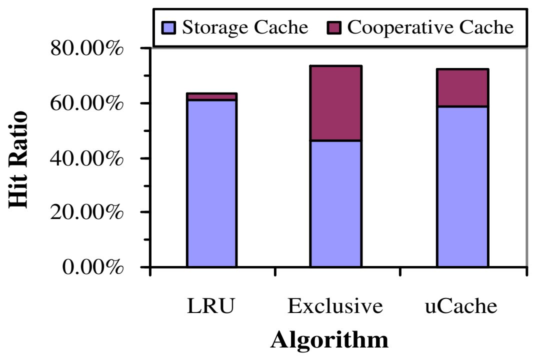


(d) 7 clients under HTTPD trace

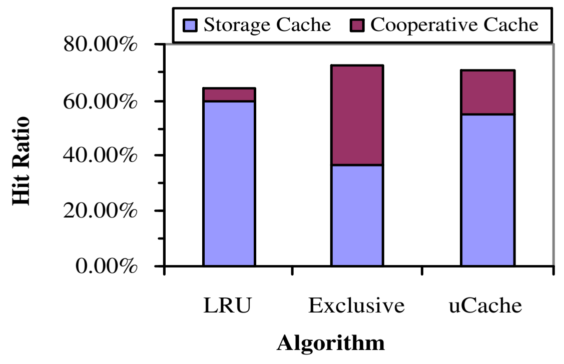
Figure 11: L2 cache hit ratio under Cell92/HTTPD traces. Hits from cooperative client caches are not included.



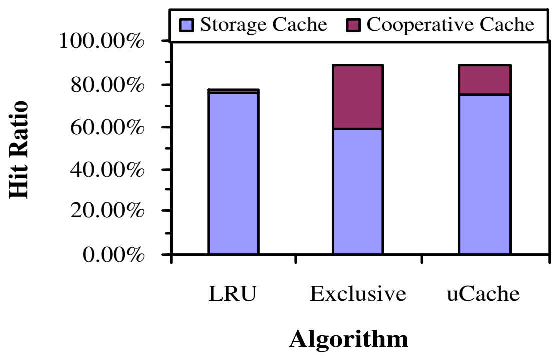
(a) 4 clients under Cello92 trace



(b) 8 clients under Cello92 trace

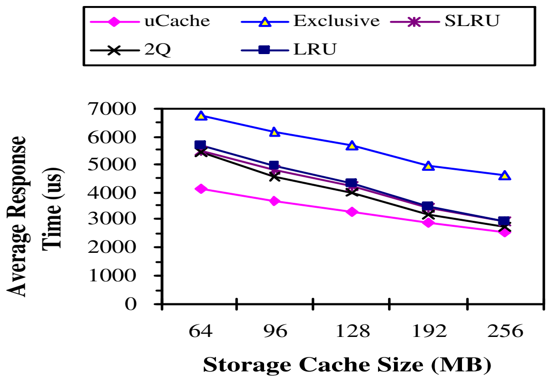


(c) 16 clients under Cello92 trace

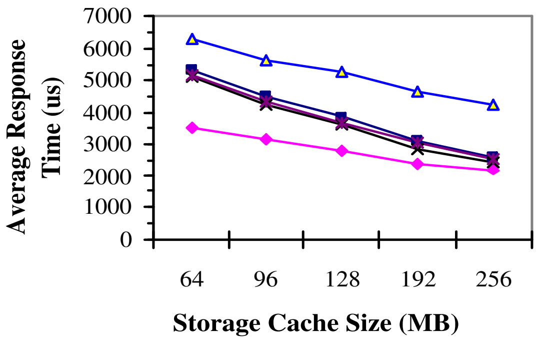


(d) 7 clients under HTTPD trace

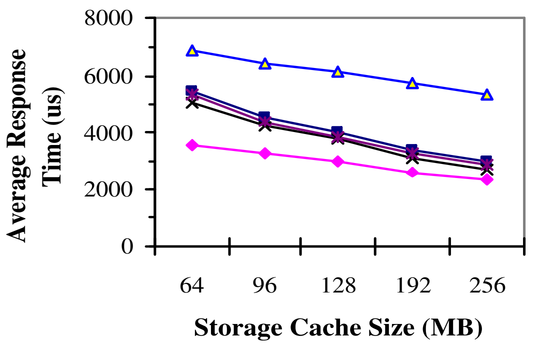
Figure 12: Cumulative hit ratio comparison under Cello92(128MB storage cache)and HTTPD(64MB storage cache) traces.



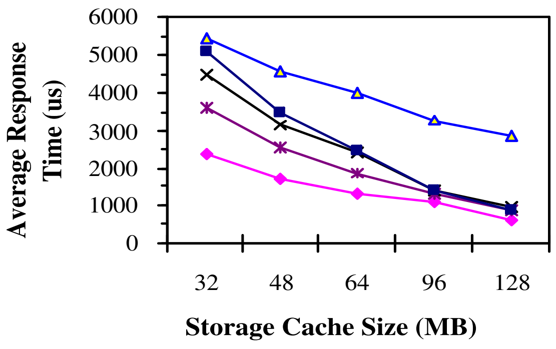
(a) 4 clients under Cello92 trace



(b) 8 clients under Cello92 trace



(c) 16 clients under Cello92 trace



(d) 7 clients under HTTPD trace

Figure 13: Average response time of L2 cache under Cello92 and HTTPD traces.

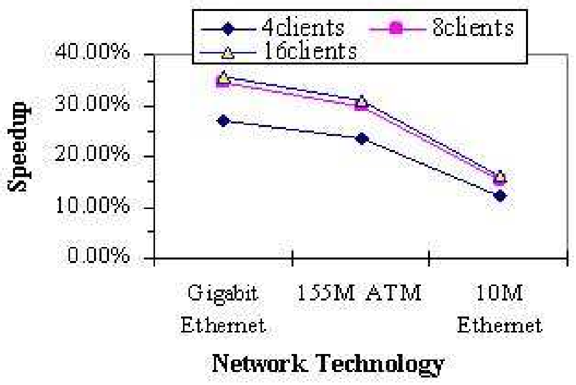


Figure 15: uCache vs. LRU under Cello92 trace.

prehensive comparisons, we find that although *uCache* provides smaller response times in all three settings, the gain decreases when the network speed is slower. *uCache* on Gigabit Ethernet achieves the highest improvement of up to 40%, while the improvement in 10Mb Ethernet is limited to under 13%. Although this result indicates that *uCache* is sensitive to network speeds, current popular networks are fast enough for *uCache* to achieve reasonable speedups.

7 Related Work

L2 caches have poor hit ratios, as demonstrated in Muntz and Honeyman (1992) and Froese and Bunt (1996). Further studies show that the poor hit ratio is caused by both weaker temporal locality (Bunt et al., 1993; Zhou et al., 2004) and duplicated blocks (Wong and Wilkes, 2002). After studying behavior of NFS servers, Reed and Long (1996) found that the 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 (Zhou et al., 2004), Demotion-based algorithm (Wong and Wilkes, 2002), Global L2 buffer cache management (Zhou et al., 2004), X-Ray (Bairavasundaram et al., 2004), and client-controlled cache replacement (Jiang and Zhang, 2004). Chen et al. (2005) 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. (2002) proposed ACME to adaptively select the best replacement policy for each cache-level to achieve high cumulative 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 (Zhou et al., 2004), inter-reference gap (Phalke and Gopinath, 1995), and inter-reference recency (Jiang and Zhang, 2002) 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. (1994) proposed four representative cooperative caching algorithms and demonstrated that N-Chance Forwarding can provide the best performance. GMS (Feeley et al., 1995) 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 and Hartman (1996) 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.

8 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 for multiple clients. *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 rather 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 fewest cache misses in exclusive caching. We propose new local replacement algorithms, Frequency Based Eviction-Reference (*FBER*), and Adaptive Space Allocation algorithm (*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* dramatically increases cumulative cache hit ratios over LRU and improve the average I/O response time by up to 46% for low-correlated workloads and 53% for high-correlated workloads.

In multiple clients with high-correlated workloads, the speedup of *uCache* is sensitive to network technologies. Although slower networks obviously decrease the performance of *uCache*, current popular networks provide enough speed for *uCache* to achieve decent speedups.

9 Acknowledgments

The authors would like to thank the anonymous referees for their insightful and constructive comments. This work was supported in part by the Research Office under a Faculty Research Grant and the Center for Manufacturing Research at Tennessee Technological University. It was also partially supported by 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.

REFERENCES

Ari, I., Amer, A., Gramacy, R., Miller, E. L., Brandt, S. A., and Long, D. E. (2002). ACME: Adaptive caching using multiple experts. In *Proc. in Informatics*, volume 14, pages 143–158.

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

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

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

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

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

Denning, P. J. (1968). The working set model for program behavior. *Communications of the ACM*, 11(5):323–333.

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

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

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

Jiang, S. and Zhang, X. (2004). 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*.

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

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

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

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

O'Neil, E. J., O'Neil, P. E., and Weikum, G. (1993). The LRU page replacement algorithm for database disk buffering. In *Proc. ACM SIGMOD Int'l Conf. Management of Data*, pages 297–306.

Phalke, V. and Gopinath, B. (1995). 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.

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

Robinson, J. T. and Devarakonda, M. V. (1990). Data cache management using frequency-based replacement. In *Proc. ACM SIGMETRICS Conf. Measurement and Modeling of Computer Systems*.

Ruemmler, C. and Wilkes, J. (1993). Unix disk access patterns. In *Proc. Winter 1993 USENIX Conf.*

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

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

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

Zhou, Y., Chen, Z., and Li, K. (2004). Second-level buffer cache management. *IEEE Transactions on Parallel Distributed Systems*.