Empirical Evaluation of Multi-level Buffer Cache Collaboration for Storage Systems

Zhifeng Chen, Yan Zhang, Yuanyuan Zhou
Department of Computer Science
University of Illinois at Urbana-Champaign
{zchen9,zhy,yyzhou}@cs.uiuc.edu

Heidi Scott and Berni Schiefer
IBM Canada
Toronto Laboratory
{hscott,schiefer}@ca.ibm.com

ABSTRACT

To bridge the increasing processor-disk performance gap, buffer caches are used in both storage clients (e.g. database systems) and storage servers to reduce the number of slow disk accesses. These buffer caches need to be managed effectively to deliver the performance commensurate to the aggregate buffer cache size. To address this problem, two paradigms have been proposed recently to *collaboratively* manage these buffer caches together: the ***hierarchy-aware caching*** maintains the same I/O interface and is fully transparent to the storage client software, and the ***aggressively-collaborative caching*** trades off transparency for performance and requires changes to both the interface and the storage client software. Before storage industry starts to implement collaborative caching in real systems, it is crucial to find out whether sacrificing transparency is really worthwhile, i.e., how much can we gain by using the aggressively-collaborative caching instead of the hierarchy-aware caching? To accurately answer this question, it is required to consider all possible combinations of recently proposed local replacement algorithms and optimization techniques in both collaboration paradigms.

Our study provides an empirical evaluation to address the above questions. Particularly, we have compared three aggressively-collaborative approaches with two hierarchy-aware approaches for four different types of database/file I/O workloads using traces collected from real commercial systems such as IBM DB2. More importantly, we separate the effects of collaborative caching from local replacement algorithms and optimizations, and uniformly apply several recently proposed local replacement algorithms and optimizations to all five collaboration approaches.

When appropriate local optimizations and replacement algorithms are uniformly applied to both hierarchy-aware and aggressively-collaborative caching, the results indicate that hierarchy-aware caching can deliver similar performance as aggressively-collaborative caching. The results show that the aggressively-collaborative caching only provides less than 2.5% performance improvement on average in simulation and 1.0% in real system experiments over the hierarchy-aware caching for most workloads and cache configurations. Our sensitivity study indicates that the performance gain of aggressively-collaborative caching is also very small for various storage networks and different cache configurations. Therefore, considering its simplicity and generality, hierarchy-aware caching is more feasible than aggressively-collaborative caching.

SIGMETRICS'05, June 6–10, 2005, Banff, Alberta, Canada.

Categories and Subject Descriptors

C.4 [**Performance Of Systems**]: *Design studies*; C.0 [**General**]: *System architectures*; D.4.8 [**Operating Systems**]: Performance—*Simulation, Measurement*

General Terms

Algorithms, Design, Experimentation, Measurement, Performance

Keywords

Storage system, Database, File system, Collaborative caching

1. INTRODUCTION

The rise of web-centric or service-based computing drives the demand of data centers. Storage is one of the foundational bricks in such computing environments. To improve performance, both storage servers and storage clients usually use large main-memory buffers for caching. For example, EMC Symmetric storage servers contain 4–64GB of memory as their storage caches [11]. A typical commercial database server, such as IBM DB2, also equips with gigabytes of memory [32]. Therefore, caches of storage servers and storage clients make up a two-level cache hierarchy as shown in Figure 1.

Although both storage client caches and storage caches are increasingly large, they do not deliver the performance commensurate to their aggregate buffer cache size. There are two reasons restricting the effectiveness of such a cache hierarchy. First, filtered by client caches, accesses to storage caches have weaker temporal locality. As a result, locality-based caching algorithms, such as LRU, are less efficient for storage caches. Second, data are cached redundantly by both storage client caches and storage caches. Therefore, the effective cache size is smaller than the total amount of caches in the hierarchy [34, 37].

To address this problem, researchers have proposed two paradigms: *hierarchy-aware caching* and *aggressively-collaborative caching*, to collaboratively manage the buffer cache hierarchy together for the purpose of delivering better performance.

Hierarchy-aware caching maintains the same I/O interface and is fully transparent to the storage client software (e.g. databases). The collaboration is achieved by merely exploiting information

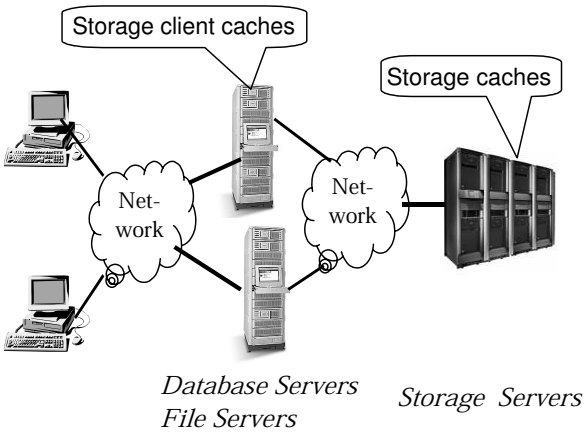


Figure 1: Buffer caches of storage clients (databases or file systems) and storage caches form a two-level cache hierarchy.

available at storage servers without any storage client hints. Examples of hierarchy-aware caching include the MQ replacement algorithm [37], the eviction-based placement [6] and the X-RAY mechanism [1]. Zhou et al. [37] found out that accesses to storage caches usually have poor temporal locality due to filtering effects of storage client caches. Thus, they proposed a Multi-Queue replacement algorithm based on their access pattern study. Chen et al. [6] proposed an eviction-based placement policy, which postpones the block placement from access-time to the time when this block is evicted from storage client caches. They also proposed a method to dynamically estimate eviction information by storage client caches without requiring any hints from storage clients. Bairavasundaram et al. proposed an X-RAY mechanism for storage caches [1] to achieve exclusive storage caching by assuming file system semantics of storage clients and estimating their caching behaviors.

Aggressively-collaborative caching trades off transparency for the possibility of improved performance. It collaboratively manages the storage client-server buffer cache hierarchy by extending the standard I/O interface and modifying storage client software. So far, two main aggressively-collaborative approaches have been proposed. The first one, called ***hint-based***, uses hint information provided by storage clients to improve the performance of storage caches. The DEMOTE scheme proposed by Wong and Wilkes is an example of this approach [34]. The second one, called ***client-controlled***, allows storage clients to control the caching of storage servers. For example, Song and Zhang recently proposed a unified, level-aware caching protocol called ULC, in which a storage client decides whose cache should cache a block [19].

Even though previous studies on aggressively-collaborative caching have shown some good results of aggressively-collaborative approaches for certain workloads, the generality of their evaluation results is quite limited due to two reasons:
(1) Their conclusions are usually drawn from very limited case studies, which focus on comparing a particular collaboration approach with the baseline case, and assumes that the baseline case uses the simple least recently used (LRU) algorithm to manage all buffer caches. However, real systems can use different local replacement algorithms and apply various optimizations to improve caching efficiency. For example, recent work has demonstrated the effectiveness of a new replacement algorithm called ARC [24] in the IBM TotalStorage DS8000 [16]. Recent work has also shown that various optimizations, such as exclusive-caching [6, 34] and cold-block elimination [20], are effective in improving cache performance. Therefore, it is unclear whether aggressively-collaborative caching is still beneficial when these new replacement algorithms and optimizations are uniformly applied to both the baseline and the collaboration cases.
(2) Most previous work on aggressively-collaborative caching does not compare with hierarchy-aware caching that does not require any changes to the I/O interface and storage client software. Therefore, it is unknown whether sacrificing transparency as in aggressively-collaborative caching is really worthwhile, i.e., how much extra performance we can gain by using aggressively-collaborative caching over hierarchy-aware caching, especially if we apply all possible recently-proposed optimizations and local replacement algorithms to both collaboration approaches?

Providing answers to the above questions is very important. Currently, storage vendors such as IBM are in the process of investigating collaborative caching between storage clients and storage servers [15]. Industrial development teams need a guideline to decide whether it is worthwhile to modify the storage client software and I/O interface to support aggressively-collaborative caching.

Our study provides a rigorous empirical evaluation to address the above questions. More specifically, our evaluation has the following characteristics that make our results much more general than previous studies:

- Besides the two aggressively-collaborative approaches (hint-based and client-controlled), we have also designed a new aggressively-collaborative approach called *content-aware* caching, in which a client cache changes its eviction decisions based on the content on the storage server cache. In addition, we also extend the hint-based approach from simple hints like eviction information to general hints such as semantic information (e.g. the importance of a block), which is usually available at storage clients (e.g. IBM DB2).
- We separate the effects of collaborative caching from local replacement algorithms and optimizations, and uniformly apply several recently proposed local replacement algorithms and optimizations to all collaboration approaches. In our comparison of the total 248 combinations, we choose the best local replacement and optimizations for each collaboration approach.
- Our evaluation is conducted using four different types of database/file I/O workloads including online transaction processing (OLTP), decision-support system (DSS) and file system workloads, using traces collected from real commercial systems such as IBM DB2.
- Our above simulation results are also validated using experiments on a real system running OLTP workloads. The system is composed of a database server and a storage server, in which we have implemented both hierarchy-aware and aggressively-collaborative caching approaches, various cache replacement algorithms and local optimizations.
- We have also studied the effects of different storage client and storage cache configurations, and the effects of storage area network (SAN) latency such as IP-storage, Fibre-Channel and future SANs.

Our study indicates that it is more important and effective to apply local optimizations or change the local replacement algorithms in hierarchy-aware caching in order to achieve good performance. Our results show that aggressively-collaborative caching evaluated in our study can only provide less than 2.5% performance improvement on average over hierarchy-aware caching for most workloads and cache configurations. In addition, the end performance (database transaction rate) improvement in real systems

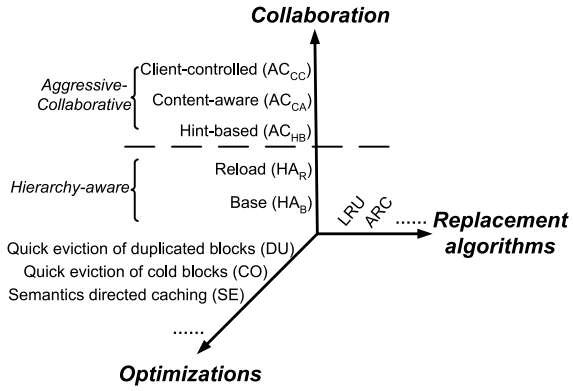


Figure 2: Evaluation space of storage client caches and storage caches.

is only 1.0%. This is because hierarchy-aware caching, with appropriate replacement algorithms and optimizations, can approximately enable both storage client caches and storage caches to work as one global cache without resorting to a centralized algorithm. Furthermore, this observation is also true for different storage networks and various cache configurations. In short, comparing the marginal performance gain of aggressively-collaborative caching with the simplicity and generality of hierarchy-aware caching, hierarchy-aware caching is more practical.

The rest of this paper is organized as follows. Section 2 describes the evaluation space for storage client caches, storage caches and collaboration. Section 3 to section 6 explain in detail of each design dimension. Experimental methods and workloads are described in Section 7, followed by trace-based simulation results in Section 8 and real system evaluation results in Section 9. Section 10 summarizes related work and Section 11 draws conclusions.

2. EVALUATION SPACE

To separate factors that can affect the effectiveness of a storage client-server buffer cache hierarchy, we can envision a large number of specific design combinations in a three-dimensional evaluation space as shown on Figure 2. The three dimensions are: client-server collaboration, local cache replacement algorithms and local optimizations. Along each dimension, there are many different design choices, which have various effects on the performance of the cache hierarchy. Since each design dimension is orthogonal to the others, we should separate their effects. Therefore, to compare different collaborative caching approaches, we need to combine each collaboration strategy with the other two dimensions: replacement algorithms and local optimizations.

The first design dimension, also the focus of our paper, is how a storage client buffer cache and a storage cache collaborate with each other. Different collaboration approaches require different levels of information exchange between a storage client and a storage server. For example, the MQ replacement algorithm [37] and the eviction-based placement [6] require no extra information from a storage client. In contrast, in the client-controlled approach [19], the storage cache is managed entirely by the storage client. In the next section, we will describe different collaboration approaches in more detail.

The second design dimension is cache replacement algorithms which make decisions upon replacement based on certain heuristics. Many replacement algorithms have been proposed in the past. For example, LRU is a commonly used replacement algorithm that replaces the least recently used block when a buffer cache is full. ARC [24] is a recently proposed replacement algorithm that considers both frequency and recency information at replacement. Since both storage caches and storage client caches can independently decide their cache replacement algorithms, we need to consider various combinations.

The third design dimension is local optimizations. Storage clients and storage servers usually exploit workload specific optimizations to improve their performance. For example, databases can identify the root block of B^+-tree and may prefer caching it longer than others. On the other hand, some blocks can be evicted immediately according to workload access patterns. The number of possible optimizations can be quite large depending on various workload characteristics.

Obviously, we cannot explore all choices in each design dimension. Instead, we select those representative ones that are either commonly used, or have recently been proposed with promising good results. In particular, in addition to existing collaboration approaches, we also evaluated a new collaboration approach called content-aware caching, and extended the hint-based collaboration approach with more useful hints such as semantic information.

Our evaluation space which we explore is much larger than previous work, and previous work only examined a limited number of points in our evaluation space (See Section 6). For example, most previous work assumes that storage client caches are managed by simple LRU. They do not consider other replacement algorithms and various workload specific optimizations. In contrast, our work considers three dimensions and evaluates 248 design combinations in total. Therefore, our work can draw more general conclusions about the effectiveness of collaborating caching strategies.

Our study currently focus only on caching and does not consider prefetching. We expect that prefetching can be uniformly applied to all collaboration approaches. Even though it is possible to have collaborative prefetching schemes, its effectiveness is beyond the scope of this paper, and remains as our immediate future work.

Moreover, our study assumes a single storage client and a storage server. This assumption is reasonable because (1) many high-end industrial configurations usually use a dedicated storage system for a single database [36] to avoid storage contention; (2) it is relatively easy to extend collaborative caching to handle multiple clients, using the dynamic partitioning technique [35] proposed by Zhou et al.; (3) Only aggressively-collaborative caching requires changes to deal with multiple clients, while hierarchy-aware caching works for multiple clients directly. As our results indicate that aggressively-collaborative caching is not worthwhile performance-wise even for a single storage client, the issue of handling multiple storage clients would further push the balance towards hierarchy-aware caching.

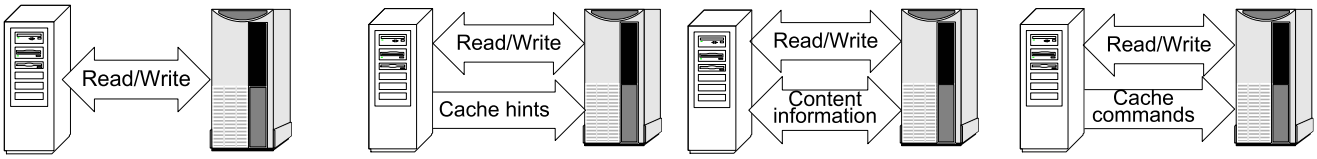
3. COLLABORATIVE CACHING

In order to improve the storage client-server cache hierarchy performance, researchers have proposed several collaboration approaches. According to whether they change the I/O interface, they can be categorized into hierarchy-aware caching approaches and aggressively-collaborative caching approaches.

3.1 Hierarchy-aware Caching Approaches (HA)

Hierarchy-aware caching improves the storage client-server cache hierarchy by exploiting storage server intelligence (Figure 3(a)). Storage servers become aware of the existence of large storage client caches in their front-ends, but do not require changes to the I/O interface and storage client software.

Hierarchy-aware caching improves storage caching performance by transparently estimating the dynamic behavior of storage client caches. For example, storage servers can estimate eviction of stor-



(a) Hierarchy-aware caching approaches

(b) Aggressively-collaborative caching approaches

Figure 3: Hierarchy-aware and aggressively-collaborative caching approaches. Storage clients are on the left and storage servers are on the right in graphs.

age client caches by monitoring I/O buffer addresses [6]. If the storage client tries to read block A into a memory buffer that was previously used to access block B, the storage server can infer that block B is very likely evicted from the storage client buffer cache. Such eviction can also be inferred by exploiting filesystem semantics [1] without explicit hints from storage clients. Such eviction information can be exploited by a storage server to reload those blocks evicted from the storage client cache to the storage cache so that future accesses to these blocks would hit in the storage cache.

Our evaluation investigates two particular hierarchy-aware approaches. One approach is the basic hierarchy-aware approach (HA_B). It knows the existence of storage client caches. Such knowledge facilitates storage caches to efficiently exploit certain optimizations such as exclusive caching (Section 5.1).

The other approach is called HA_R, which estimates storage client evictions and may reload evicted blocks from disks proposed by Chen et al [6]. In HA_R, by monitoring the buffer address of every read access from the storage client, the storage server maps client memory addresses to blocks cached by the storage client. When an address in the storage client is overwritten by another read access, the storage server detects that the corresponding block is evicted. Therefore, the storage server can reload the evicted block into its cache. Although these reloads may incur extra disk accesses, the disk scheduler can schedule reload operations in the background and hide some of these overheads using scheduling algorithms such as FreeBlock scheduling [23].

The above method can be readily used when the storage client accesses one dedicated storage server. In some very large systems where the storage client accesses multiple storage servers, they often access storage servers through a storage virtualization module (e.g. a router) in the storage area network. The above technique can be applied in such a module rather than in every storage server.

3.2 Aggressively-collaborative Caching Approaches (AC)

Different from hierarchy-aware approaches, aggressively-collaborative caching approaches trade off transparency for better performance (Figure 3(b)). They extend I/O interfaces to enable exchange of extra information between storage clients and storage servers. In our study, we evaluate two existing aggressively-collaborative approaches (hint-based and client-controlled) and one new approach (content-aware). Furthermore, we enhance the hint-based approach to pass application semantics information to storage servers. These approaches trade off transparency for performance at different degrees. The hint-based caching is the least aggressive one, while client-controlled caching are the most aggressive one.

3.2.1 Hint-based Caching (AC_{HB})

In the hint-based caching, storage clients provide hints to guide storage server cache management. Storage clients can pass two types of hints: access patterns and application semantics. Both require small changes to the standard I/O interface and the storage client software. Hint information is usually so small that it can be piggybacked along with requests. Such a mechanism of passing hints from the storage client to the storage server is orthogonal to how to exploit hints for caching.

Access patterns at storage clients are useful for improving the performance of storage caches because such patterns are usually lost after filtering through the storage client cache. For example, it is easier for the storage client to find out some sequential access patterns, whereas it is rather difficult for the storage server to find such patterns due to interleaving of requests from multiple access streams of the same client (e.g. different database worker threads), or multiple storage clients. Another example of access patterns is eviction information. For example, the DEMOTE method proposed by Wong and Wilkes requires a storage client to send evicted blocks to storage caches [34].

Semantic hints, which are usually available only at a storage client, can also help improve storage cache performance. For example, storage clients, such as the IBM DB2, know exactly some data blocks are only accessed once. Such storage client can send this information to storage servers. Storage caches can avoid caching such read-once data to save cache space for more important data. Since semantics hints are application specific, storage servers need to understand various hints from different applications. As a result, the change of storage server software would be complex.

This study examines the effects of two kinds of hints. One kind of hints are evicted blocks. We change the I/O interface and enable storage clients to send evicted blocks to storage servers. Upon receiving evicted blocks, storage servers cache them as the most recently accessed blocks. The other kind of hints are semantic hints. Specifically, we use data importance as hints. Data importance tells storage caches how long storage caches should try to keep accessed blocks, or whether they should evict accessed blocks immediately.

3.2.2 Content-aware Caching (AC_{CA})

The content-aware caching makes caching decisions based on the content of other caches. For example, block A is cached by both the storage client cache and the storage cache. Block B is cached by the storage client cache, but is not cached by the storage cache. When the storage client cache needs to replace one block among A and B according to its cache replacement algorithm, it is better to evict A than B so that future accesses to A or B would not incur a disk access. Compared to the hint-based caching, the content-aware caching requires more changes to the storage client cache management. Hence, it is more aggressive.

The content-aware caching may incur extra network traffic between storage clients and storage servers to exchange content information. If there is only one pair of storage client and storage server, the client can emulate changes of the storage server cache. Otherwise, they can piggyback the change of cache content with regular I/O operations. To further reduce the updating traffic, one

Approach	I/O interface change	Storage client change	Extra network traffic	Handle multiple clients	Application specific
HA_B	No	No	No	Yes	No
HA_R	No	No	No	Yes	No
AC_{HB}	Yes	Yes	Yes	Possible	Yes
AC_{CA}	Yes	Yes	Yes	Possible	Yes
AC_{CC}	Yes	Yes	Yes	Difficult	Yes

Table 1: Summary of five collaboration approaches.

cache can use Bloom filters [3] to create a summary of its cache content and send the summary to other caches periodically [12].

Our study examines one particular method of the content-aware caching. This method makes storage clients informed about storage cache contents. Instead of choosing exactly one block to evict directly, a storage client cache selects a group of victim candidates according to its replacement algorithm. The final victim among the candidates is the first one which is also cached by the storage cache. The number of victim candidates is a tunable parameter. In our evaluation, this parameter is manually tuned to achieve the best performance for content-aware caching. It can also be dynamically adjusted based on performance feedback. The dynamic tuning algorithm decreases the number of victim candidates when accesses hit among the candidates and increases the number otherwise.

3.2.3 Client-controlled Caching (AC_{CC})

The client-controlled caching uses storage clients to manage both storage client caches and storage caches as a single unified cache. Storage servers receive control commands from storage clients to update its caches. In this way, a storage client is responsible for making global caching decision. Therefore, it is relatively easy to achieve the global optimal performance.

Although client-controlled caching is intuitively easy to achieve good performance, it faces many complex issues. First, client-controlled caching requires extensive change of storage client software, storage server software and I/O interfaces. Second, when multiple clients are sharing one storage server, client-controlled caching approaches need to coordinate these clients to make a global decision at the storage cache and guarantee no malicious clients can intentionally hurt other clients' performance. Finally, client-controlled caching limits some functionalities of storage caches, such as read-ahead caching, write-back policy etc.

Our evaluation uses the ULC [19] algorithm as the representative method for client-controlled caching, in which each block is either cached by the storage client cache or the storage cache. The storage client maintains the meta-data of a larger cache whose size is the sum of both caches. To reduce moving the data block among storage client caches and storage caches, ULC decides which cache to cache the block according to the reference distance between the last two accesses to this block. If the distance is large, the block will only be cached in the storage cache. Although the original ULC algorithm manages the aggregate cache in an LRU fashion, our evaluation also extends its basic idea to other cache replacement algorithms, such as ARC [24], and combines it with other optimizations, such as using caching hints (see Section 5.3).

3.3 Summary

In Table 1, we compare the hierarchy-aware and aggressively-collaborative caching to analyze whether they need to change the interface and storage client software, whether they incur extra network traffic and how well they can handle multiple clients and support general applications. We also notice that only content-aware caching has a tunable parameter among these approaches.

Hierarchy-aware caching is transparent to storage clients while aggressively-collaborative caching involves both storage clients and storage servers. Therefore, all aggressively-collaborative approaches need to change both the I/O interface and storage client software. Since all aggressively-collaborative approaches exchange extra information between storage clients and storage servers, they incur additional storage area network traffic. While hierarchy-aware caching handles multiple clients directly, aggressively-collaborative caching need special mechanism for storage servers to coordinate storage clients. In addition, while hierarchy-aware caching supports general applications, aggressively-collaborative approaches are specific to certain applications.

Aggressively-collaborative approaches sacrifice different degrees of transparency. The hint-based approach is the simplest one to implement in real systems. It only needs to tag with each I/O operation with a few bits to indicate caching hints. On the other hand, content-aware and client-controlled approaches require storage clients to maintain the meta-data for storage caches, which in turn increases the complexity of storage client software and storage area network traffic. Furthermore, the client-controlled approach needs to address many issues to deal with multiple storage clients, such as controlling malicious clients.

4. CACHE REPLACEMENT ALGORITHMS

In previous work, storage caches and storage client caches are assumed to use LRU as their basic replacement algorithms. However, many replacement algorithms have been recently proposed to improve upon LRU [18, 20, 24, 37]. For example, the adaptive replacement algorithm (ARC) [24] considers both frequency and recency of a block in its replacement. It does this by dividing the cache into two components: the recency component and the frequency component. ARC dynamically adjusts relative sizes of these two components.

Our study chooses both LRU and ARC as our representative replacement algorithms. The rationale for choosing LRU is because LRU is commonly used and it also allows us to validate against previous work.

The reason for choosing ARC is because it is the most recently proposed replacement algorithm and has been shown to outperform many existing algorithms including LRU, LFU, FBR [28], LIRS [18], MQ [37] and 2Q [20], for a variety of workloads, such as OLTP, ERP, SPC1 and NT workstation workloads [24]. Recently, IBM TotalStorage DS8000 uses ARC as its cache replacement algorithm [16].

ARC or LRU can be used for both storage client caches and storage caches. Both of these two replacement algorithms do not have any tuning parameters. Therefore, we can have a total of four combinations: (1) LRU for client and LRU for storage; (2) LRU for client and ARC for storage; (3) ARC for client and LRU for storage; (4) ARC for client and ARC for storage. Our study applies all these four combinations to all collaboration approaches with various local optimizations.

5. LOCAL OPTIMIZATIONS

We study three commonly used cache optimization techniques: quick eviction of duplicated blocks (DU), quick eviction of cold blocks (CO) and semantics directed caching (SE). These optimizations have been shown to be effective in recent caching studies [1, 19, 26]. Optimization DU and SE do not have any tuning parameters. For CO, we set uniformly its tuning parameter to make fair

comparisons. Although DU and CO have been proposed before, previous work only studies their effects on some particular storage client-server cache collaboration approaches. To the best of our knowledge, no previous work has ever studied SE for the storage client-server cache collaboration.

5.1 Quick Eviction of Duplicated Blocks (DU)

This technique avoids the storage cache keeping the same blocks as the storage client cache. Therefore, it is called exclusive caching [1, 34]. This technique minimizes replications in the buffer cache hierarchy and effectively enlarges the amount of available cache spaces. Previous work [1, 34] has demonstrated good performance improvements with this optimization technique.

It is straightforward to implement DU. Whenever a block is fetched from the storage cache, the storage cache will replace it quickly since the storage client cache will cache it.

DU can be applied to only storage caches but can be combined with all collaboration approaches. However, since the client-controlled approach allows each block to be cached by only one cache, exclusiveness is already naturally achieved.

5.2 Quick Eviction of Cold Blocks (CO)

Cold blocks are blocks that are rarely accessed. Caching cold blocks has a negative effect of polluting the cache by evicting some potentially useful blocks. Much previous work [19, 20, 37] has shown that quickly evicting cold blocks can effectively improve buffer cache performance.

One way to identify cold blocks is to extend replacement algorithms with a history buffer. The history buffer records a number of past accesses in an LRU [19] stack or FIFO [20]. If a block is accessed while the history buffer still remembers it, this block is a warm block. Otherwise it is cold. The size of the history buffer is a tuning parameter for CO. To make fair comparison, our evaluation set the history buffer size as same as the cache size for all collaboration approaches.

Different from DU, this optimization can be applied to both storage caches and storage client caches. But combining with other optimizations such as DU is a little more complicated because DU in storage caches already evicts any recently accessed blocks from storage caches.

5.3 Semantics Directed Caching (SE)

Previous work shows that application semantics is helpful in directing cache replacement [7, 27]. For example, a database can identify the root block of a B^+-tree index or a large sequence of data blocks which are accessed by a sequential table scan. Accordingly, database buffer management can decide to cache the index root block for a longer time and quickly evict those data blocks in the sequential scan [7, 31].

In our evaluation, we utilize the data importance value provided in two database workloads from IBM DB2. Through query plans, databases know precisely whether one block will be accessed again. Accordingly, each block accessed is assigned a value indicating its importance. Based on such data importance, storage client caches can selectively avoid caching some blocks and keep others longer.

Although semantics directed caching can be uniformly applied to all collaboration approaches at the storage client cache, only the hint-based approach can explicitly pass semantic information as hints and the storage server can utilize such caching hints. When the client-controlled approach is combined with semantic directed caching, the storage client can use such hints directly to manage both caches globally.

Method	Collaboration	Replacement	Optimizations
EV [6], X-RAY [1]	HA_R	LRU+LRU	DU
DEMOTE [34]	AC_{HB}	LRU+LRU	DU
ULC [19]	AC_{CC}	LRU	CO

Table 2: Previously proposed methods in the evaluation space.

6. COVERAGE OF THE EVALUATION SPACE

The panoramic evaluation space allows us to evaluate much more storage client-server collaborative caching methods including all the previously proposed ones.

The evaluation space covers all previous storage client-server collaborative caching methods. As summarized in Table 2, each previous method is just a particular combination of collaboration approach, a replacement algorithm and some optimizations. For example, the DEMOTE scheme proposed by Wong and Wilkes [34] is just a hint-based collaboration approach combined with LRU for both storage client caches and storage caches and the DU optimization for storage caches.

In our evaluation space, we evaluate *248* different designs of client-server cache hierarchy in total. There are 5 collaboration approaches (HA_B, HA_R, AC_{HB}, AC_{CA} and AC_{CC}), 2 replacement algorithms (LRU and ARC), and 3 optimizations (DU, CO and SE). Each replacement algorithm and most local optimizations can be applied to both storage client caches and storage caches.

7. EVALUATION METHODOLOGY

We evaluate collaborative caching approaches using both trace-based simulation and real system implementation.

7.1 Trace-based Simulation

We first evaluate collaborative caching approaches through trace-driven simulations for multi-level caches. We simulate a system composed of one storage server and a storage client, which are connected through a storage area network.

Our simulation uses four large real system traces. They are either collected by ourselves using industrial benchmarks or by other researchers in large production systems. They represent OLTP (On-Line Transaction Processing), DSS (Decision Support System) and filesystem workloads.

- *OLTP* workload trace is a database buffer access trace collected on an IBM DB2 database running IBM's TPCC benchmark (1000 warehouses). OLTP workload is dominated by small random accesses. The trace has 132 million accesses. The workload accesses 9.7GB data during 12 hours. The block size is 4KB.
- *DSS* workload is another database buffer access trace collected on an IBM DB2 database running IBM's TPCH benchmark. The benchmark has several sequential table scans of one huge table (lineitem). The trace contains 59 million references and accesses 15GB data in 2 hours. The block size is 4KB.
- *Cello99* is a low-level disk I/O trace collected on an HP UNIX server with 2GB memory. The I/O accesses are filtered by the filesystem cache. Therefore, its temporal locality is quite poor. We use one week trace between 12/02/1999 and 12/08/1999 and the block size is 8KB. The data footprint is as large as 117GB.
- *Lair62b* is an NFS server RPC trace collected on an NFS server by SOS project of Harvard University [10]. The original Lair62b trace is an NFS trace. We convert it into a block

Block Size	4KB		8KB	
Access latency (ms)	T_h	T_m	T_h	T_m
VI/Fibre-Channel	0.125	6.352	0.137	6.940
IP Storage	0.692	6.983	0.979	8.101
Future SAN	0.012	6.238	0.014	6.817

Table 3: Access latencies of different storage network configuration for different block sizes (4KB and 8KB). VI/Fibre-Channel network is the default network in our evaluation.

access trace through an FFS-like filesystem simulator, which models i-node and data blocks and ignores other meta-data information. We use one day trace of 2/24/2003 and the block size is 4KB. The data footprint is 6.7GB.

We use average read access time as the major performance metric in our simulation experiments (In contrast, our implementation experiments use end database performance, transaction rates, as the performance metric). An application access to a data block can be a hit in the storage client cache (e.g. the database buffer cache), a hit in the storage cache, or an access to the disk (a miss from both caches). Let H_1, H_2 and M denote the total number of such three types of accesses respectively. Let T_h denote the average access latency of a storage cache hit and T_m denote the average access latency of a disk access. Relatively, the access latency of a hit in the storage client cache is negligible. Accordingly, we estimate the average read access time as follows:

$$T = \frac{T_h \times H_2 + T_m \times M}{H_1 + H_2 + M}$$

To compare the best performance of each collaboration approach, we do not penalize any collaboration approach with extra latency charge imposed by the extra information exchange between the storage client and the storage server in aggressively-collaborative caching, or the extra data reloads from disks as in hierarchy-aware caching. This conservative estimation is reasonable because: (1) Various techniques can be used to hide such overhead. For example, FreeBlock scheduling can enable reload operations to be carried out in the background [23]. In this way, demote operations of hint-based and client-controlled approaches can overlap with the read operations. For content-aware caching, the storage client can obtain states of server caches by piggybacking messages with normal I/O requests. (2) Our real system experimental results (Section 9) that include all these overheads validate our simulation results.

Because both T_h and T_m depend on the storage area network latency, we examine the effect of different networks: VI/Fibre-Channel SAN, IP Storage and future SAN (Table 3). We measure storage cache hit and miss latencies on two real systems. One uses V3 storage servers [36], which is a commercial storage server using VI network. VI networks have similar network latency as Fibre-Channel SANs. The IP storage uses the same server and client platforms as the previous V3 storage. But the network is an Ethernet running IP. To evaluate collaborative caching for even faster storage network in the future, we assume the future network would be 10 times faster than VI/Fibre-Channel network while the disk latency remains the same.

7.2 Real System Implementation

To evaluate collaborative caching approaches in a real system, we conduct experiments in a system composed of a database server and a storage server. Each of the two servers has one 2.4GHz Pentium IV processor with 512KB L2 cache and 1GB of main memory. The storage server runs commercial storage software called V3 [36], which manages its own storage cache and processes I/O requests from the database server. The storage server connects to the database server via a Virtual Interface (VI) network [36] provided by Emulex cLAN network cards. The peak VI bandwidth is about 113MBps and the one-way latency for a short message is 5.5μs. The database server runs the Shore database storage manager [5]. The operating systems of both servers are Windows 2000 Advanced Server. The system is driven by a TPC-C like benchmark developed by CMU [14].

The original Shore and V3 are modified to support collaborative caching. We selectively implemented HA_B, HA_R for hierarchy-aware caching and AC_{HB}, AC_{CC} for aggressively-collaborative caching in the prototype, because our simulation results indicate AC_{CA} does not outperform the other approaches for OLTP workload. The implementation of HA_R only needs to modify the storage server to monitor the eviction of the database and support reload operation. The AC_{HB} approach require extension of I/O interface to send hints from the database buffer to the storage cache. In the prototype, Shore is modified to pass the evicted blocks as the hints to the storage server. Upon receipt of the evicted block, the storage cache keeps it in the cache and does not to write the data block to disks. The implementation of AC_{CC} needs two major I/O interface extensions. One I/O extension is *cache-bypass*, which allows the database to read/write data from/to the storage server and informs the storage server not to cache the data. Another extension is to include storage cache replacement decision in each read/write I/O operation. In the prototype, Shore is modified to manage both the database bufferpool and the storage cache. Using the extended I/O interface, Shore directly controls the storage server whether to cache a data block and which data block to replace on every access to the storage server.

8. SIMULATION RESULTS

In this section, we report our simulation results comparing hierarchy-aware and aggressively-collaborative approaches for the four workloads. We first report the comparison results with all replacement algorithms and optimizations allowed, among which we pick the best combination to represent the best case for each collaboration approach. Since the performance gaps between any two collaboration approaches are the largest with a 1:1 cache distribution (the storage client cache and the storage cache have the same sizes), we use such cache distribution in the discussion of overall results.

We also study the effects of different storage area networks (SAN) besides the default SAN (Fibre-Channel/VI), including both IP SAN and future low-latency SAN. We also study the performance sensitivity to different cache distributions between a storage client cache and a storage cache. At the end of this section, we report the result using LRU without any optimizations, just to compare with previous work and illustrate the importance of exploring the whole evaluation space.

8.1 Overall Results

Our simulation results show that hierarchy-aware caching, with proper cache replacement algorithms and local optimizations, can achieve almost as good performance as aggressively-collaborative caching approaches. The average performance improvement of aggressively-collaborative approaches over hierarchy-aware approaches is less than 2.5%.

Tuned for Each Cache Configuration. To examine the best response time achieved by different hierarchy-aware and aggressively-collaborative caching approaches, we first assume that the system is manually tuned. For each workload, each cache con-

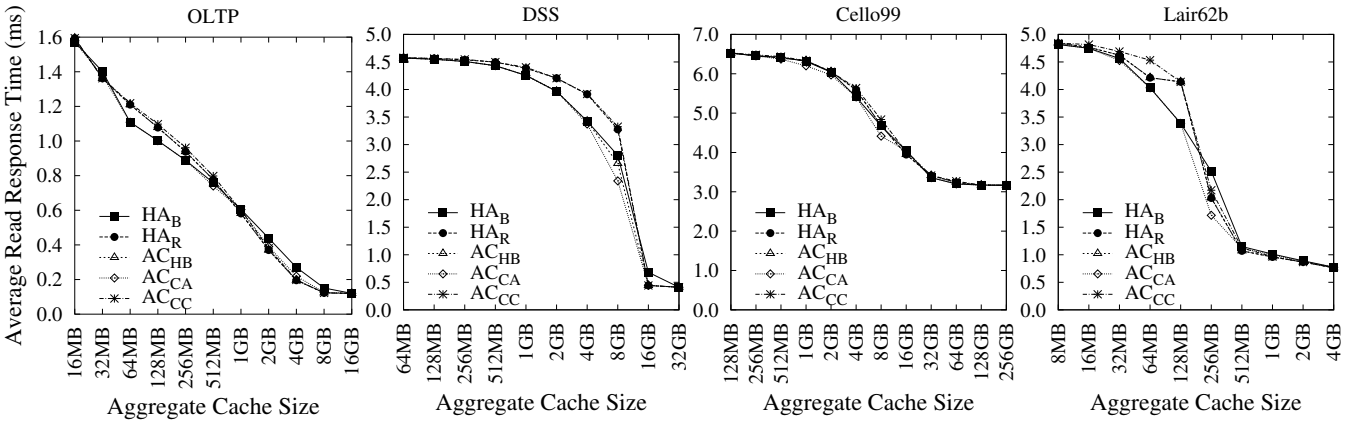


Figure 4: Average read response time of optimized approaches tuned for each cache configuration. The storage client and server have caches of the same size. X-axis shows the aggregate cache size.

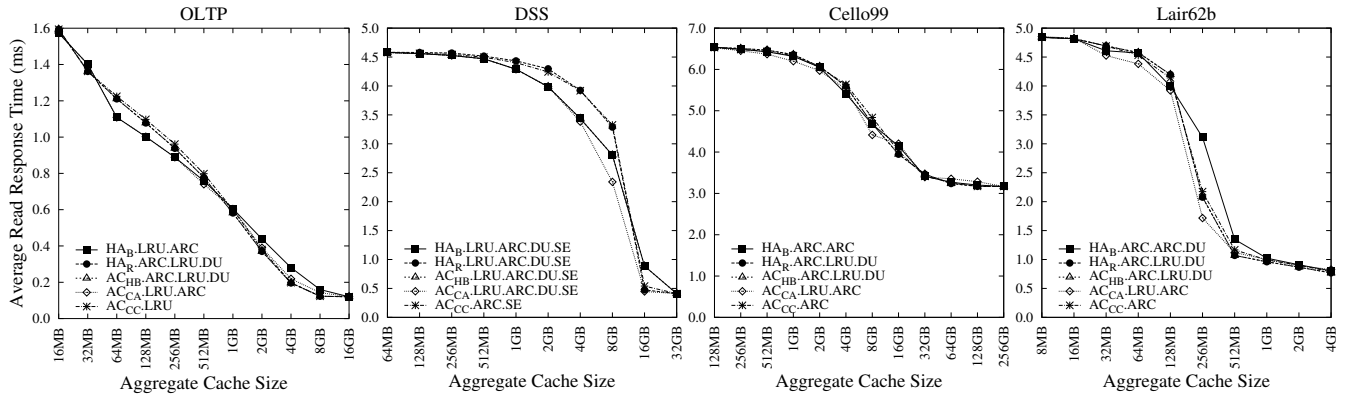


Figure 5: Average read response time of optimized approaches tuned for each workload. X-axis shows the aggregate cache size. For each approach, one particular optimization strategy is chosen. One strategy specifies cache replacement algorithms and whether DU, SE and CO are used. For client-controlled, only one replacement algorithm is needed. For example, $AC_{HB}.LRU.ARC.DU.SE$ means that in the hint-based approach the storage client cache uses LRU, the storage cache uses ARC, DU is applied to the storage cache and SE is applied to both caches.

Response time improvement (%)	OLTP			DSS			Cello99			Lair62b		
	MIN	MAX	AVG	MIN	MAX	AVG	MIN	MAX	AVG	MIN	MAX	AVG
Tuned for cache config	0.0	2.5	0.2	0.0	16.5	1.8	0.0	5.6	0.8	0.0	15.5	1.6
Tuned for workloads	0.0	2.9	0.3	0.0	16.5	2.4	-2.1	5.6	0.7	-0.1	17.4	2.5
IP Storage	0.0	0.4	0.1	0.0	14.5	1.6	0.0	4.5	0.6	0.0	15.0	1.5
Future SANs	0.0	2.9	0.3	0.0	17.0	1.9	0.0	5.7	0.8	0.0	15.6	1.7

Table 4: Summary of the response time improvement. For each workload and each cache configuration, we compare the best aggressively-collaborative approach with the best hierarchy-aware approach and calculate the relative response time improvement in percentage. This table shows the minimum (MIN) and the maximum (MAX) improvement for each workload. In addition, it shows the average (AVG) improvement over all cache configurations for each workload.

figuration and each approach, we choose the best combination of replacement algorithms and optimization techniques. Figure 4 compares the average response time of each approach at its best.

The result shows that hierarchy-aware approaches can achieve close performance to aggressively-collaborative approaches. On average, the response time difference between the best hierarchy-aware approach and the best aggressively-collaborative approach is between 0.2% and 1.8% for four workloads(Table 4). This suggests that the performance gain of aggressively-collaborative caching does not justify the complexity of changing I/O interface between storage servers and storage clients to enable their aggressive collaboration. Instead, with better cache replacement algorithms and local optimizations, hierarchy-aware approaches can achieve performance quite close to the best performance achievable by aggressively-collaborative approaches in real workloads.

For all the workloads, aggressively-collaborative approaches outperform hierarchy-aware approaches by more than 5.6% at only two special points. One is DSS workload at 8GB cache and the other is Lair62b workload at 256MB cache. Such improvements depend on workload characteristics. For example, in the DSS workloads, many tables can fit into an 8GB cache, but not a 4GB buffer cache. Therefore, collaborative caching approaches like

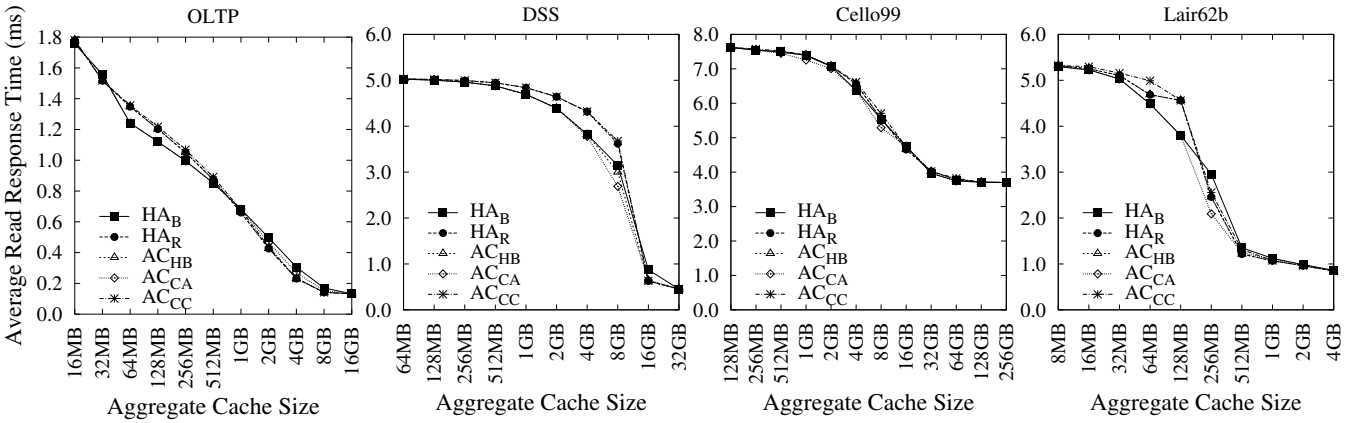


Figure 6: Average read response time of optimized approaches using an IP network. The storage client and server have caches with the same size. X-axis shows the aggregate cache size.

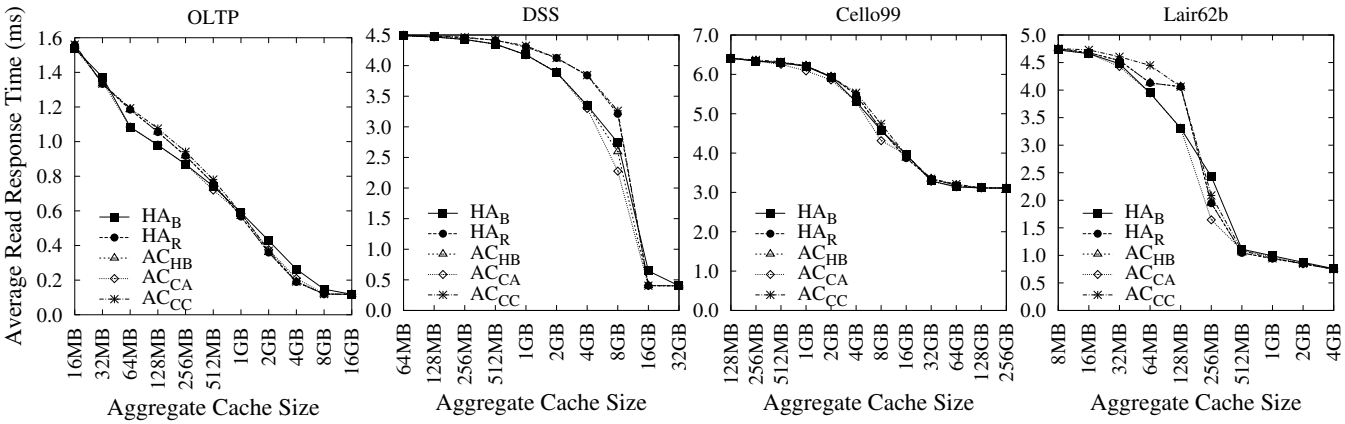


Figure 7: Average read response time of optimized approaches using a 10 times faster network than current storage area network. The storage client and server have caches with the same size. X-axis shows the aggregate cache size.

AC_{CA} that manage both client and storage caches as one large unified cache are more likely to keep these tables in the cache. Therefore, AC_{CA} performs 16.5% better than HA_B.

Tuned for Each Workload. In practice, it may not be possible to tune the system for each cache configuration because of dynamic size of caches. However, it is still feasible to study characteristics of a workload and choose an optimization strategy to achieve reasonably good performance.

Figure 5 shows the best combination of replacement algorithm and optimization for each workload and each collaborative approach. For example, for DSS workload, AC_{CA} performs the best when the storage client cache uses LRU, the storage cache uses ARC, and both DU and SE optimizations are applied. This is because SE and DU effectively avoid the cache pollution of large table scans to the storage client cache and the storage cache respectively. ARC is more effective than LRU for the storage cache, accesses to which have poor temporal locality.

The results still indicate that hierarchy-aware caching can achieve similar response time of aggressively-collaborative caching. On average, the difference between optimums of hierarchy-aware and aggressively-collaborative caching is within 0.3%–2.5%, which is similar to the results tuned for each cache configuration. This further confirms our results that the marginal performance gain is too small to justify the complexity of implementing aggressively-collaborative caching.

8.2 Effects of Storage Area Network Latency

This section examines whether aggressively-collaborative approaches would be more helpful with other kinds of storage area network other than Fibre-channel/VI, which is the default network in above experiments. There are two trends in the storage network. To reduce total cost of storage systems, one trend is to use IP network, which usually has much larger network latency. The other is to use even lower-latency network, such as InfiniBand [17], to achieve high performance. We use two different storage network latency configurations (Table 3) to represent these two trends.

IP SAN. With longer latencies of accessing storage caches, Figure 6 shows the effect of larger network latency of IP network on overall performance of aggressively-collaborative approaches. Because aggressively-collaborative caching increases the network traffic and the cost to access storage cache becomes higher, the benefit of aggressively-collaborative caching can be further offset by such network cost.

Our results validate that the improvement achieved by aggressively-collaborative approaches over hierarchy-aware approaches is even smaller than that of Fibre-Channel network. The performance gain of hierarchy-aware approaches on a network shown

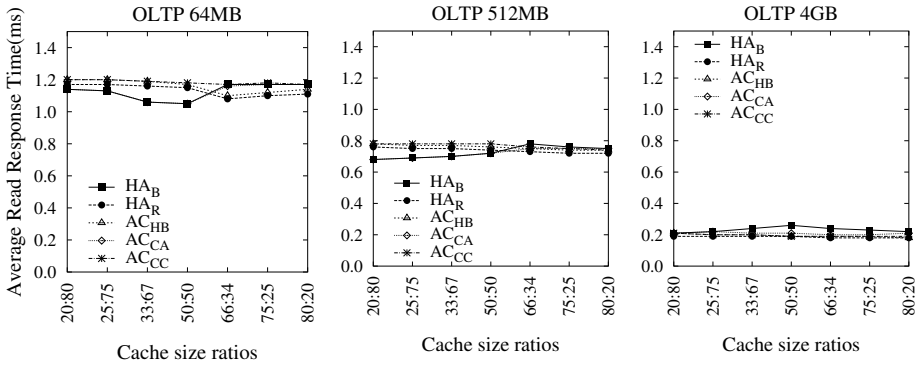


Figure 8: Average read response time of optimized approaches using varying ratios of storage client and server cache sizes. Here, the total cache size for each workload is fixed. X-axis shows the ratio of the storage client cache size versus the storage cache size. Due to the space limitation, we only show the result of the OLTP workload. Other workloads exhibit similar trends.

in Figure 6 is at most 14.5% and on average the gain is between 0.1% and 1.6% (Table 4). Compared to the performance gain on a faster network shown in Figure 4, the benefit of aggressively-collaborative approaches becomes even smaller. The reason for this is that aggressively-collaborative approaches tend to take advantage of storage caches as if they are as fast as its storage client caches. On a fast storage area network, this assumption is approximately true because of the extremely low latency. It is not the case, however, on a slow network such as Ethernet.

In short, as we expected, it is not worthwhile performance-wise for future IP storage systems to sacrifice transparency in order to exploit aggressively-collaborative approaches.

Future Low-latency SANs. In Figure 7, we also examines whether aggressively-collaborative approaches would be worthwhile if the inter-connection between storage clients and servers becomes much faster than current storage area networks. Intuitively, a faster network should give more benefits to aggressively-collaborative approaches.

Unfortunately, our results indicate that such benefit is small even with future low-latency SANs that are 10 times faster. The difference between the best response time of aggressively-collaborative caching and hierarchy-aware caching is at most 17.0% and on average the difference is between 0.3% and 1.9% (Table 4), which is only slightly better than that with current storage area networks.

8.3 Effects of Different Cache Distribution

Fixing the aggregate cache size in a storage client-server cache hierarchy, the relative sizes of storage client caches and storage caches affect the effectiveness of this hierarchy.

Figure 8 shows the OLTP workload performance of different collaborative caching approaches when the aggregate cache size is fixed but the size ratio of the storage client cache and the storage server cache varies.

As we expected, the performance gap between hierarchy-aware and aggressively-collaborative is the largest when the storage cache has the same size as the storage client cache. At other cache distribution, where either the storage cache is larger or smaller than the storage client cache, the difference between hierarchy-aware and aggressively-collaborative is much smaller. The reason is quite obvious: when one cache is larger than the other, the performance of this cache becomes more important. Therefore, collaborating with the other smaller buffer cache becomes less beneficial. This observation also applies to other three workloads.

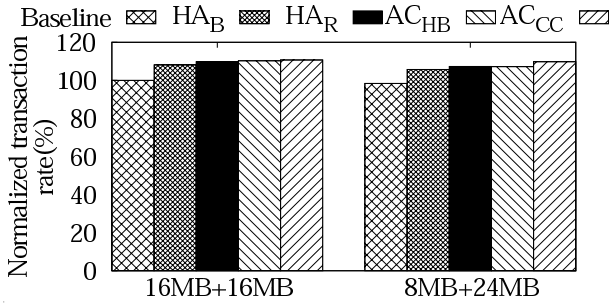


Figure 9: Normalized transaction rate for TPCC-like benchmark on Shore database. The aggregate cache size is kept constant (32MB). 8MB+24MB indicates that the database buffer-pool is 8MB and the storage cache is 24MB. In baseline, both database buffer and storage cache are managed by LRU. All results are normalized to the baseline case of 16MB+16MB.

8.4 Results of Non-optimized Approaches

In Figure 10, we report the results only using LRU and no local optimizations in each collaboration approach, just to compare with previous work and demonstrate the importance of exploring the large evaluation space.

Our results match with those of previous studies. Compared to HA_B, all collaborative caching approaches improve response time. In addition, more aggressive approaches achieve better performance. For example, for the OLTP workload, AC_{HB}, AC_{CA} and AC_{CC} improves the response time over HA_B by up to 17.9%, 22.0% and 39.9% respectively.

Compared to the results shown in Figure 10, the performance improvement of aggressively-collaborative approaches shown in Figure 4 is much smaller. This difference shows that considering cache replacement algorithms and local optimizations are necessary when we study the effect of collaborative caching between storage client caches and storage caches.

9. REAL SYSTEM RESULTS

To validate our simulation results and evaluate the effects of these collaboration approaches on end performance, we implemented these approaches in a real system described in Section 7. Our experiments are conducted using a TPCC-like database using Shore storage manager. The database size is around 4GB. In each experiment, the benchmark program carries out 10,000 transactions. A higher transaction rate, which is the number of transactions finished per minute, indicates a better performance.

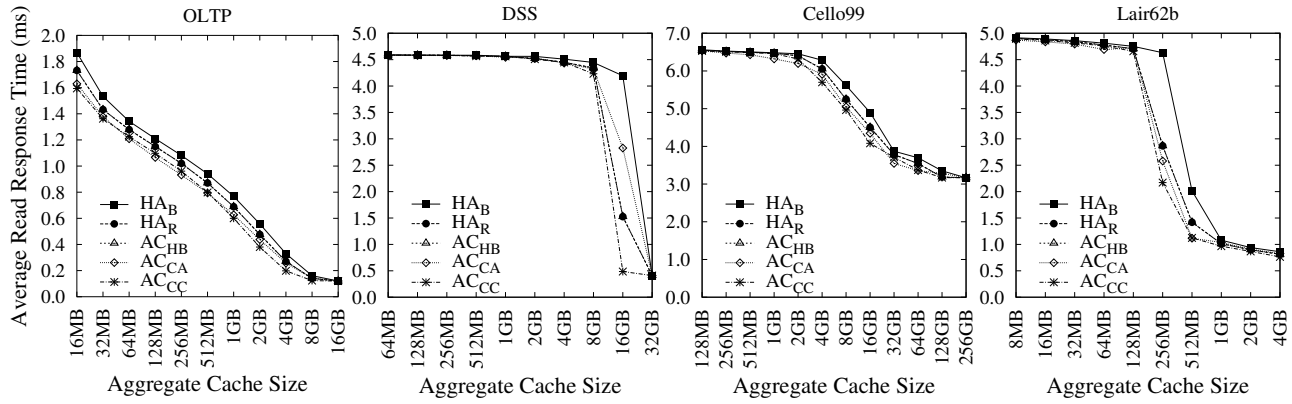


Figure 10: Average read response time of non-optimized approaches. Both the storage server and the storage client use LRU replacement algorithms and no optimizations are applied. The storage client and server have caches with the same size. X-axis shows the aggregate cache size.

Figure 9 shows that the hierarchy-aware caching achieves similar end application performance as the aggressively-collaborative caching can achieve. For example, when both the database buffer and the storage cache are 16MB, AC_{CC} is only 1.0% better than HA_R. Compared to the baseline case, in which only LRU is used and no optimization is applied, all collaborative caching approaches improve the application performance by up to 10.8%. Using ARC as the storage cache replacement algorithm and optimization DU, HA_R improves the baseline case by 9.8%. Therefore, without resorting to aggressively-collaborative approaches, real systems can achieve most of the performance improvement by adopting much simpler methods such as HA_R, with ARC as the storage cache replacement algorithm and the DU optimization.

10. RELATED WORK

Much research has studied cache replacement algorithms over the last forty years. Belady's MIN off-line algorithm gives lower bound on cache miss ratio for a single cache [2]. Most online cache replacement algorithms are based on access recency, frequency or both. LRU, LFU, 2Q [20] and EELRU [30] are classical algorithms representative of each categories. Recently, ARC is proposed to be adaptive to changing workload [24]. In practice, CLOCK and its generalized variant approximates LRU well and commonly used by database buffer management because of its small lock contention in real implementation [26].

Previous research in different environment has noticed the weakness of LRU-like algorithms for lower level buffer cache in a hierarchy and pointed out feasible solutions for different systems. Dan et al. conduct a theoretical analysis of hierarchical buffering in a shared database environment [9]. Muntz and Honeyman investigate multi-level caching a distributed file system, showing that server caches have poor hit ratios because of caching of clients [25]. Willick et al. demonstrate that FBR algorithm performs better for lower level caches than locality based replacement algorithms such as LRU [33]. Cao and Irani show that GreedyDualSize replacement algorithm is superior to other known policies for web caches [4]. Zhou et al. observe different access patterns to the storage cache and propose MQ algorithm for better storage cache management [37].

Wong and Wilkes point out the need for exclusive caching at the storage cache and propose DEMOTE operation [34]. Eviction-based data placement estimates the client eviction through monitoring addresses in each storage accesses and also achieves exclusive storage cache without complex interface change [6]. Assuming the storage client is a file server, X-RAY builds an image of the file server cache based on file system semantic and achieves exclusive storage caching by only caching a different set of blocks from the file server [1].

Various systems harvest caching capability distributed among clients and servers. Cooperative caching enables file system clients to access cached blocks in caches of other clients [8, 29]. Franklin et al. explore database client-server caching and cache consistency problem in object-oriented database context [13]. Summary cache weaves web proxies together to build a bigger web proxy [12]. Content distribution network is composed of many content servers working as one scalable web caching system [21, 22]. All these systems cooperate caching among the same software systems.

Jiang and Zhang recently propose ULC, a client controlled cache placement and replacement protocol for multi-level buffer caches [19]. Their work shares the same motivation of this paper. ULC requires all the levels in the cache hierarchy understand the new protocol. This paper shows this complexity is not necessary. By limiting modification to only one level of cache, we can also achieve similar system performance as a global managed LRU.

11. CONCLUSION

This paper investigates potential benefit of aggressively-collaborative approaches between storage clients and storage servers. Surprisingly, although aggressively-collaborative approaches are effective in improving storage system performance, we find hierarchy-aware caching can achieve similar performance as long as proper optimization techniques are used. In other words, to achieve such improvement, complex approaches such as client-controlled approaches are not necessary in most cases.

Through our rigorous empirical study of various collaborative caching approaches with 248 combinations using real system workloads, we quantitatively demonstrate that the benefit of aggressively-collaborative caching is less than 5.6% for most cases and less than 2.5% on average. Besides, we also find that even if the storage area network changes, say IP storage or future SAN, the benefit of aggressively-collaborative caching is still not significant. Furthermore, our real system results show that the end performance benefit of aggressively-collaborative caching is less than 1.0%, which validates our simulation results.

This work is currently being extended in several aspects. First, even though our evaluation space covers a large design space with

total 248 combinations (many of which are new approaches and new combinations), it is still conceivable to design new collaboration approaches, replacement algorithms or local optimizations. But we expect our general conclusions are very likely to hold, especially if these new approaches are general and not application-specific. Second, we are conducting evaluations on workloads with multiple storage clients. Third, we are in the process of adding collaborative prefetching into our evaluation space. Fourth, we will evaluate more workloads other than database and file systems.

12. REFERENCES

[1] 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 *Proceedings of the 31th Annual International Symposium on Computer Architecture*, pages 176–187, Jun 2004.

[2] L. A. Belady. A study of replacement algorithms for a virtual-storage computer. *IBM Systems Journal*, 5(2):78–101, 1966.

[3] B. H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Commun. ACM*, 13(7):422–426, 1970.

[4] P. Cao and S. Irani. Cost-aware WWW proxy caching algorithms. In *Proceedings of the 1st USENIX Symposium on Internet Technologies and Systems*, pages 193–206, Dec 1997.

[5] M. J. Carey, D. J. DeWitt, M. J. Franklin, N. E. Hall, M. L. McAuliffe, J. F. Naughton, D. T. Schuh, M. H. Solomon, C. K. Tan, O. G. Tsatalos, S. J. White, and M. J. Zwilling. Shoring up persistent applications. In *Proceedings of the 1994 ACM SIGMOD International Conference on Management of Data*, pages 383–394, May 1994.

[6] Z. Chen, Y. Zhou, and K. Li. Eviction-based cache placement for storage caches. In *Proceedings of the 2003 USENIX Annual Technical Conference*, pages 269–282, Jun 2003.

[7] H.-T. Chou and D. J. DeWitt. An evaluation of buffer management strategies for relational database systems. In *Proceedings of the 11th International Conference on Very Large Data Bases*, pages 127–141, Aug 1985.

[8] M. D. Dahlin, R. Y. Wang, T. E. Anderson, and D. A. Patterson. Cooperative caching: Using remote client memory to improve file system performance. In *Proceedings of the 1st Symposium on Operating Systems Design and Implementation*, pages 267–280, Oct 1994.

[9] A. Dan, D. M. Dias, and P. S. Yu. Analytical modelling of a hierarchical buffer for a data sharing environment. In *Proceedings of the 1991 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pages 156–167, May 1991.

[10] D. Ellard, J. Ledlie, P. Malkani, and M. I. Seltzer. Passive NFS tracing of email and research workloads. In *Proceedings of the 2nd USENIX Conference on File and Storage Technologies*, pages 203–216, Mar 2003.

[11] EMC Corporation. Symmetrix 3000 and 5000 Enterprise Storage Systems product description guide. http://www.emc.com/, 1999.

[12] L. Fan, P. Cao, J. Almeida, and A. Z. Broder. Summary cache: a scalable wide-area web cache sharing protocol. In *Proceedings of the 1998 ACM SIGCOMM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, pages 254–265, Sep 1998.

[13] M. J. Franklin, M. J. Carey, and M. Livny. Global memory management in client-server database architectures. In *Proceedings of the 18th International Conference on Very Large Data Bases*, pages 596–609, Aug 1992.

[14] S. Harizopoulos and A. Ailamaki. STEPS towards cache-resident transaction processing. In *Proceedings of the 30th International Conference on Very Large Data Bases*, pages 660–671, Aug 2004.

[15] IBM. Personal communication with IBM, Sep 2003.

[16] IBM Corporation. The datasheet for IBM TotalStorage DS8000 series. http://www-5.ibm.com/storage/europe/uk/disk/ds8000/, 2004.

[17] InfiniBand Trade Association. Infiniband Architecture Specification, Oct 2000.

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

[19] 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.

[20] T. Johnson and D. Shasha. 2Q: A low overhead high performance buffer management replacement algorithm. In *Proceedings of the 20th International Conference on Very Large Data Bases*, pages 439–450, Sep 1994.

[21] D. Karger, A. Sherman, A. Berkheimer, B. Bogstad, R. Dhanidina, K. Iwamoto, B. Kim, L. Matkins, and Y. Yerushalmi. Web caching with consistent hashing. In *Proceeding of the 8th International Conference on World Wide Web*, pages 1203–1213, May 1999.

[22] T. Leighton. The challenges of delivering content on the Internet. In *Proceedings of the 20th ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, page 246, May 2001.

[23] C. R. Lumb, J. Schindler, and G. R. Ganger. Freeblock scheduling outside of disk firmware. In *Proceedings of the 2002 USENIX Annual Technical Conference*, pages 213–226, Jan 2002.

[24] N. Megiddo and D. S. Modha. ARC: A self-tuning, low overhead replacement cache. In *Proceedings of the 2nd USENIX Conference on File and Storage Technologies*, pages 115–130, Mar 2003.

[25] D. Muntz and P. Honeyman. Multi-level caching in distributed file systems -or- your cache ain't nuthin' but trash. In *Proceedings of the Winter 1992 USENIX Conference*, pages 305–314, Jan 1992.

[26] V. F. Nicola, A. Dan, and D. M. Dias. Analysis of the generalized clock buffer replacement scheme for database transaction processing. In *Proceedings of the 1992 ACM SIGMETRICS Joint International Conference on Measurement and Modeling of Computer Systems*, pages 35–46, Jun 1992.

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

[28] J. T. Robinson and M. V. Devarakonda. Data cache management using frequency-based replacement. In *Proceedings of the 1990 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pages 134–142, May 1990.

[29] P. Sarkar and J. Hartman. Efficient cooperative caching using hints. In *Proceedings of the 2nd USENIX symposium on Operating Systems Design and Implementation*, pages 35–46, Oct 1996.

[30] Y. Smaragdakis, S. Kaplan, and P. Wilson. EELRU: simple and effective adaptive page replacement. In *Proceedings of the 1999 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer systems*, pages 122–133, May 1999.

[31] M. Stonebraker. Operating system support for database management. *Communications of the ACM*, 24(7):412–418, 1981.

[32] Transaction processing performance council. http://www.tpc.org.

[33] D. L. Willick, D. L. Eager, and R. B. Bunt. Disk cache replacement policies for network fileservers. In *Proceedings of the 13th International Conference on Distributed Computing Systems*, pages 2–11, May 1993.

[34] T. Wong and J. Wilkes. My cache or yours? Making storage more exclusive. In *Proceedings of the 2002 USENIX Annual Technical Conference*, pages 161–175, Jun 2002.

[35] P. Zhou, V. Pandey, J. Sundaresan, A. Raghuraman, Y. Zhou, and S. Kumar. Dynamic tracking of page miss ratio curve for memory management. In *Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 177–188, Oct 2004.

[36] Y. Zhou, A. Bilas, S. Jagannathan, C. Dubnicki, J. F. Philbin, and K. Li. Experiences with VI communication for database storage. In *Proceedings of the 29th Annual International Symposium on Computer Architecture*, pages 257–268, May 2002.

[37] Y. Zhou, J. Philbin, and K. Li. The multi-queue replacement algorithm for second level buffer caches. In *Proceedings of the 2001 USENIX Annual Technical Conference*, pages 91–104, Jun 2001.