A Locality-Aware Cooperative Cache Management Protocol to Improve Network File System Performance

Song Jiang, Kei Davis
Los Alamos National Laboratory
Los Alamos, NM 87545, USA
{sjiang, kei}@lanl.gov

Fabrizio Petrini
Pacific Northwest National Laboratory
Richland, WA 99352, USA
fabrizio.petrini@pnl.gov

Xiaoning Ding and Xiaodong Zhang
Department of Computer Science and Engineering
Ohio State University
Columbus, OH 43210, USA
{dingxn, zhang}@cse.ohio-state.edu

Abstract

In a distributed environment the utilization of file buffer caches in different clients may vary greatly. Cooperative caching is used to increase cache utilization by coordinating the usage of distributed caches. Existing cooperative caching protocols mainly address organizational issues, paying little attention to exploiting locality of file access patterns. We propose a locality-aware cooperative caching protocol, called LAC, that is based on analysis and manipulation of data block reuse distance to effectively predict cache utilization and the probability of data reuse. Using a dynamically controlled synchronization technique, we make local information consistently comparable among clients. The system is highly scalable in the sense that global coordination is achieved without centralized control.

1. Introduction

I/O performance of file systems in distributed and cluster-based parallel systems has a critical impact on application performance. A central file server may handle requests from hundreds of clients simultaneously and become a serious bottleneck. Distributed storage systems such as Panasas and Lustre Storage Systems can be used to provide a high storage bandwidth [10]. However, on-demand file access latency is not correspondingly reduced.

1.1. Cooperative Caching

The concept of cooperative caching to form a unified level of cache on top of server cache in the file cache hierarchy has been proposed to reduce disk accesses [3]. Systems employing such techniques include GMS and PGMS [6, 13]. In the more than ten years since cooperative caching was first introduced, the technical trends that originally motivated it are more relevant than ever: for example, the performance gap between processor and disk has continued to widen, which makes disk access time increasingly long in terms of CPU operations. While the performance gap is not expected to change dramatically in the foreseeable future, reducing disk access via improved caching remains a desirable goal. In the meantime, however, advances in network technology—vastly improved bandwidth and latency—improve the possibilities for sharing memory space among clients. For example, the use of 10 gigabit Ethernet or Infiniband can make the transfer a data block between clients in a LAN two orders of magnitude faster than fetching the block from a state-of-the-art hard disk. Furthermore, because the aggregate amount of memory in clients scales with the number of clients, cooperative caching improves the system scalability by alleviating the workload pressure on the server as number of clients increases.

1.2. Balanced Cache Utilization

A major objective of cooperative caching is to balance the utilization of caches by moving the data from overloaded caches to underloaded ones. A cooperative caching protocol usually provides mechanisms for *block forwarding* and *block locating*. In block forwarding, a block demoted from a client cache is stored in the cache of a selected remote client rather than being discarded, thus making it more readily available if later requested. Block locating is a mechanism whereby a client can determine the location (hosting client) of a requested block that has been cached either because of a previous forwarding or a local access.

For cooperative caching to achieve efficient and balanced cache utilization, two critical elements of any proposed solution are the determination of *which blocks should be forwarded*; and, *where the blocks should be forwarded*. The algorithms for making these determinations depend on predictions of the *locality strengths* of blocks at each client. Here an accessed block with strong locality will be re-accessed soon; one with weak locality not so soon or not ever. The blocks with weakest locality, i.e. the leastest likely to be re-accessed in the near future, would be the candidates for being forwarded, while blocks with stronger locality would be retained in the local cache. The decision of whether to forward, and if so to where, depends on the relative local and peer *cache utilization*. A cache's utilization is determined by its blocks with weakest locality—a cache containing blocks of weaker locality than another is deemed to have lower utilization. A candidate for forwarding would be sent to another client whose cache utilization is lower than the forwarding client's, if such a client exists, otherwise discarded as in a traditional system. The effectiveness of the system depends on developing a method to predict block locality—a *locality measure*—that exhibits good accuracy, and the ability to make locality quantified at each client *consistently comparable* across the system. If this were achieved, cache utilization across all clients would be balanced, and the aggregate cache at client level more efficiently utilized.

1.3. Locality Measures in Current Cooperative Caching Protocols

Previously proposed cooperative caching protocols generally focus on some organizational issues such as server involvement, cache consistency, and coordination efficiency [3, 6, 11]. The study of the analysis of data access patterns and client cache utilization is still preliminary. For example, researchers have not yet developed an effective locality measure for forwarding decisions and host client selection. Instead, they have simply borrowed the locality measure used in LRU (Least Recently Used) replacement, which is the age of a block. In LRU, a block that has not been used recently (i.e., with a large age) is predicted to be unlikely to be re-accessed (i.e., have weak locality), while a block recently accessed is assumed to have strong locality and therefore should be cached. However, in many situations this locality measure is very inaccurate. For example, if a client sequentially reads a large data file multiple times, blocks with higher age will always be re-accessed sooner than any with lower age. Another example is the one-time scan of files, where all the accessed blocks are of least value for being cached, no matter how recently they were accessed. In these cases, cache performance can be very poor. When the age-based locality measure is employed in cooperative caching, its behavior may jeopardize the performance of not only a single cache but that of peer caches as well—if a client were to rapidly scan a large amount of disk data, it would flush a large number of blocks with small ages, which might never be used again, to other clients whose possibly heavily-used working sets would be displaced. This concern regarding age-based measure has been raised in the context of the GMS system as one of its weaknesses [6]. Generally, the possibility of this inter-client *interference* speaks against the use of cooperative caching despite its potential benefits. In short, what is needed is effective data access pattern analysis and associated locality measures, and a method for consistent comparison of locality (and so utilization) between caches.

1.4. Our Objectives and Solutions

In this paper we propose a *L*ocality-*A*ware *C*ooperative caching protocol, *LAC*, that uses block reuse distance as the locality measure. The reuse distance of a block is the number of other distinct blocks accessed between two consecutive references to the block. In contrast with the age of a block, which only characterizes the number of other blocks subsequently accessed, reuse distance predicts access frequency based on historical precedent. Using the last reuse distance of a block to predict the next reuse distance, we can avoid the disadvantages associated with the measure of age while still being able to exploit the property of stable locality exhibited during a particular phase of program execution. Using the last reuse distance, the locality strength of a block is determined immediately upon its access. Only those blocks with strong locality are cached in memory, and those with relatively weak locality are consequently evicted. For example, because one-time accessed blocks have infinite reuse distance, they are not subject to caching in local memory or forwarding to remote memory. This is in contrast with the use of age, where all blocks recently accessed have small ages and are cached until they grow old.

While each client uses reuse distance to measure block locality strength and keeps those with relatively strong locality in its cache, we must provide a method to make the locality measure consistently comparable across all clients. This is achieved by a periodic synchronization among clients. A block is only allowed to be forwarded from a client with high cache utilization to another one with a low utilization. In this way we can achieve balanced cache utilization across the clients and high performance for this level of the cache hierarchy with minimal adverse interference.

In summary, there are three objectives in the *LAC* protocol to address the weaknesses in current designs.

1. Though current and emerging advanced networking technology enable inexpensive data transfer, local and remote caches are still in a non-uniform caching structure. Thus blocks with strong locality in their host

clients should be cached locally, and only those with weaker locality should be candidates for forwarding.

2. Clients should only forward blocks that, through retention, are predicted to improve aggregate cache performance. This implies that a local block replaced by a forwarded block must have weaker locality on a globally comparable basis to minimize clients unduly degrading the performance of a peer without producing greater global performance improvement.
3. The protocol should be scalable with the size of the system. Our protocol is fully decentralized, with no server involvement, to prevent the server from becoming a performance bottleneck.

2. Related Work

Our work builds on, and synergistically combines, two somewhat distinct areas of research: work on cooperative caching and work on exploiting reuse distance as a measure of locality strength. Related work in both areas is presented.

2.1. Related Work on Cooperative Caching

Around a decade ago, rapid advances in processor and network performance inspired the original work on cooperative caching by Dahlin et al. [3]. They described and evaluated a variety of protocols covering a large design space. They concluded that a protocol named N-Chance Forwarding provided the best performance and relatively low overhead.

N-Chance Forwarding uses the LRU stack as the data structure for managing information about access history. Two factors are considered in deciding whether a block should be forwarded: its age and its redundancy in the system. The last block in the LRU stack, which has the largest age, is forwarded if it is the only copy cached in any client, and is referred to as a *singlet*. No account is taken of the cache utilization of the client that accepts the block. Thus a heavily loaded client may have to accept forwarded blocks of relatively low locality strengths. To limit the lifetime of a cached block, a *recirculation count* is maintained to limit the number of times a block is forwarded. The selection of the target client for forwarding is very simple: it is chosen at random. N-Chance Forwarding maintains a per-block forwarding table on the server to track every block being cached by each client. By consulting the global table a client can determine whether one of its blocks is a singlet, and where to find any remotely cached block. This design minimizes caching redundancy and maximizes block locating performance. However, it also results in a performance bottleneck at the server, which limits the system scalability.

The N-Chance protocol was later integrated into the *xFS* serverless network file system design [1, 2], where central management of the metadata at the server is removed for better scalability and availability. Feeley et al. introduced *Global Memory Service* (GMS), which implemented a distributed shared memory system [6]. GMS deals with all memory activities, including VM paging, memory-mapped files, and explicit file access. To avoid the blindness of target client selection in N-Chance, GMS uses a centralized manager to periodically collect age information of blocks in each client and to disseminate the information to all the clients so that a client caching blocks with large age blocks can be chosen as a forwarding target.

Sarkar et al. designed a decentralized 'hint'-based cooperative caching protocol that removed the need for centralized control at the server and significantly reduced the overhead of centralized coordination [11]. Instead of consulting a central manager to locate requested blocks at remote clients, the protocol uses its own hint to explore the cooperative cache directly. In the hint-based system, the first copy of a block to be cached by any client is designated the *master copy*. The notion of master copy is similar to that of the singlet in N-Chance. To avoid redundancy, only a master copy may be forwarded to a peer client. Any client that has forwarded a master copy will update its own hint to show where the block was sent. If that client later seeks to retrieve the block it will follow the path indicated by the chain of hints to reach the block; if a client without a hint for the block is reached, the request is sent to the server. Like GMS this scheme relies on the age information from peers to select the target with the oldest block for forwarding. However, it does not consult a central manager for the age information. Instead, it uses a mechanism called *best guess replacement*, where each client maintains a local *oldest block list* that contains what the client believes to be identities and locations of the oldest blocks of each client. When a block is forwarded from one client to another, both clients exchange the age information of each of their current oldest blocks, thus allowing each client to update its own list. Our LAC protocol uses similar hint and best-guess techniques to achieve low overhead and scalability. However, we do not use age for a target client selection. Instead, we use reuse distance to make a more effective selection.

2.2. Related Work on Reuse Distance and Locality

Reuse distance has been widely used in recent years to characterize data access locality and to improve program efficiency, CPU cache management, and buffer cache management. For example, reuse distance is used to analyze program locality and its access characteristics [4]. It is used to enhance the locality of programs [15]. Reuse distance can also help improve CPU cache performance [5]. The LIRS buffer cache replacement algorithm, based on a reuse distance analysis, has been proposed to effectively overcome the performance disadvantages of LRU replacement [8]. Reuse distance has also been used to improve virtual memory replacement using the CLOCK-Pro policy [7]. All of the work shows that reuse distance is a powerful and effec-

tive measure for characterizing access locality for improving performance.

Reuse distance has also been introduced into the management of hierarchical multi-level caches in a client-server system using a protocol called Unified Level-aware Caching (ULC) [9]. There are two major differences between the ULC protocol and the LAC protocol presented here. First, the caches being managed have different structures. ULC manages a hierarchy of caches such as client cache, server cache, and RAID disk cache, while LAC manages client caches at the same level. In more detail, the caches in ULC form a tree structure with no breadth-wise communication, while the cache structure in LAC is effectively flat and directly uses breadth-wise (peer to peer) communication. Second, the concerns differ. In the tree-like structure, all the block requests go through the clients before they reach the server, and all fetched blocks go through server before they reach clients, so there exists client filtering issue to address to improve server caching performance. In the latter, all of the caches are at the same level so there is no inherent interaction between them. LAC addresses the problem of how to balance cache utilization in this latter regime to improve overall system performance without introducing undue interference among clients.

3. A Locality-Aware Protocol

Our LAC cooperative caching protocol is built on an effective locality measure. We use block reuse distance to determine access locality. Both local caching and cache space sharing among clients are locality-conscious. That is, only those blocks with relatively weak locality in a client are forwarding candidates, and only those candidates with relatively strong locality evaluated from the view of the target client are allowed to be forwarded to it.

We first describe locality characterization on a single client. We then describe how to make the locality strength characterized at each client comparable across all the clients. Next we describe how a block is forwarded based on consistent locality characterization. Last we describe how a block is located.

3.1. Locality Characterization at One Client

We use an LRU stack to facilitate locality characterization by tracking reuse distance. The LRU stack is a fixed-size data structure used in the implementation of LRU replacement. In LRU replacement, a block is pushed onto the top of the stack when the block is accessed, and is removed when if it reaches the bottom and another block is pushed onto the top.[1] If not reused, a block will move down in the stack as new blocks are accessed. Thus the distance of a block from the top of the stack defines its age or *recency*—how many distinct blocks have been subsequently accessed. The LRU algorithm relies solely on recency for its replacement decision: the block with the largest recency is the candidate for replacement.

The LRU stack can be used to measure reuse distance. When a block in the stack is accessed, its recency represents its reuse distance—the time between its last access and the current access. If a block is accessed for the first time and is not in the stack the reuse distance is taken to be infinite.

The problem with using reuse distance as a measure of locality strength is that a reuse distance becomes outdated after it is collected. For example, suppose block A is accessed when its recency is 5, then its reuse distance is 5. Following that, suppose there are 6 other distinct blocks accessed. Then block A has a recency of 6 but the last reuse distance is still 5. Its current recency is a better measure of the true locality strength. Thus, in fact, after a block is accessed and placed back into the stack top its current recency is the better indicator of its locality strength.

The purpose of evaluating locality strength at the time a block is accessed is to determine whether it should be cached in the local client or be a forwarding candidate. We therefore compare the *reuse distance* of the currently accessed block with the *largest recency* of the cached blocks. If the reuse distance is smaller, the block joins the group of blocks regarded as having a strong locality. Otherwise it goes into the pool of forwarding candidates. Using the LRU stack the comparison is simple and efficient.

We manage the metadata of accessed blocks in the same way as the LRU replacement algorithm does. However, blocks are allowed to leave the cache while their metadata is still in the stack. We refer to those metadata as *shadow blocks*. We also maintain a small LRU stack as a pool to hold the forwarding candidates.

For clarity we use two examples to show how the locality strength of an accessed block is evaluated and how a forwarding candidate block is identified. Figure 1a shows three types of blocks in the LRU stack: cached blocks, which have been evaluated as having strong locality and thus are resident in the cache; forwarding candidate blocks, which have weak locality and are ready to be forwarded; and shadow blocks, which have left the cache and but are kept in the stack for the purpose of calculating reuse distance. When block G is accessed, its recency is R_1. So its new reuse distance is R_1. We need to make a decision whether the block has strong locality by comparing its reuse distance with that of the block with largest recency, which is block F. Because G is originally below F in the stack, its reuse distance is larger than the recency R_2 of block F. So G is deemed as a weak locality block and is sent to the forwarding candidate pool. Note that its metadata is moved to the stack top, as the LRU replacement algorithm does, to measure its next reuse distance. Because the access to block G is a miss, a block (block C) in the forwarding pool is selected for replacement. The locality strength of each forwarding candi-

1 In practice only the metadata of a block is stored in the stack. For simplicity we do not make the distinction in our descriptions.

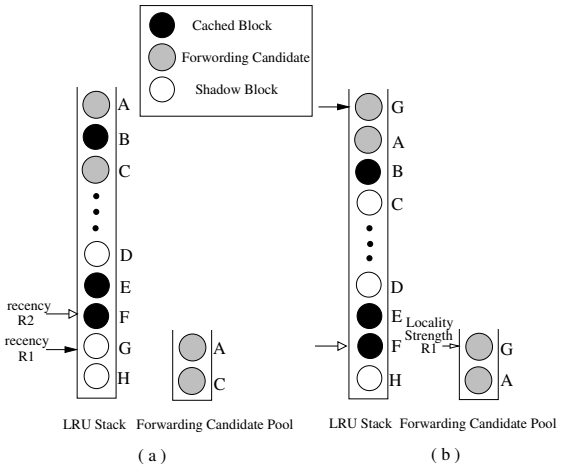

Figure 1. A block reuse distance is generated indicating its WEAK locality. The darker arrow points to the currently accessed block, and the lighter arrow points to the last cached block.

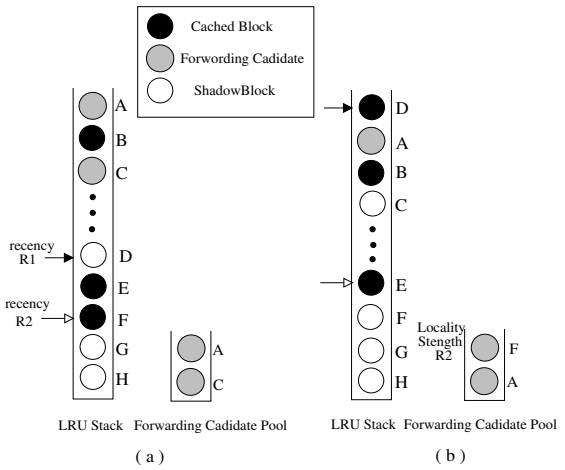

Figure 2. A block reuse distance is generated indicating its STRONG locality. The darkened arrow points to the currently accessed block, and the undarkened arrow points to the last cached block.

date is recorded. In this case the value of the strength for block G is its reuse distance R_1. The result of the access to block G is shown in Figure 1b.

Figure 2a shows a different case, where accessed block D has a reuse distance R_1 that is smaller than the recency R_2 of block F, which has the largest recency among the cached blocks. So block D is deemed a to have strong locality and is cached. Because the cache capacity is fixed, block F is demoted as a forwarding candidate. Its locality strength is set to its recency R_2. The result of the access to block D is shown in Figure 2b.

In summary, by judicious use of reuse distance and recency to define a locality measure, we can dynamically determine which blocks have strongest locality and should be cached, and which are candidates for forwarding.

3.2. Making Locality Comparable across Clients

While we are able to quantify locality at a particular client, the measure is only relative to the blocks accessed by that client, are not comparable between clients. This is because the time is measured by local block access events, not real time. Using these locality strength values for forwarding could cause blocks with weak real-time locality to replace blocks with strong real-time locality in other clients.

To address the issue, we use a global timing mechanism. Instead of relying on the server to collect and broadcast timing information, we let the clients themselves synchronize with each other to avoid the potential server bottleneck. The synchronization is made on a variable time unit we refer to as an *epoch*. Each client maintains an *epoch counter* with the same starting value at each client; this value will remain synchronized across clients. Additionally, each client maintains an *epoch timer*, a count-down timer initialized with a prescribed starting value, the *epoch threshold*. Each time a block is accessed at a client, the value of the epoch counter is assigned to the block and the epoch timer is decremented.

There are two events that end an epoch at a client. One is that its local timer expires (becomes zero). The client then increments its epoch counter by one and broadcasts a message containing the new epoch number to all other clients, instructing them to terminate their epochs. The other event that ends an epoch at a client is when a client receives such a message from another client before its own timer expires, and the epoch number in the message exceeds the client's own. In this case the client increases its epoch counter to match that in the message. In both cases the client starts a new epoch by resetting its timer to the epoch threshold. Thus the length of the epoch for all clients is determined by the client that makes the the most frequent block accesses in real time. Figure 3 shows an example of the accessed blocks at different clients grouped by common epoch number.

Next we observe that the epoch number of a block approximates its recency, reuse distance, and locality strength, and are comparable across clients. For the purpose of comparing locality between clients for making forwarding decisions, the finer distinctions are unnecessary: simple comparison of epoch numbers is sufficient. In the example shown in Figure 3, all the three blocks of client A in epoch n have a recency of 2 because the stack-top blocks of client A have epoch number $n + 2$. If one of those three blocks were then accessed it would generate a reuse distance of 2.

3.3. Forwarding Blocks among Clients

To forward a block from a local cache to a remote cache it is desirable to choose the least-loaded client as the for-

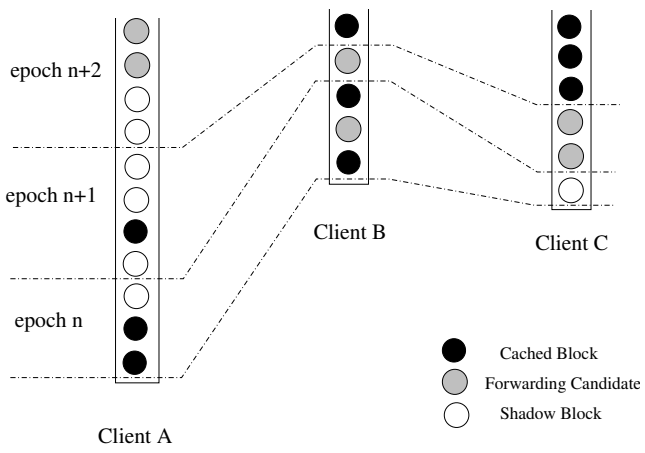

Figure 3. An example illustrating the accessed blocks in the LRU stacks of clients A, B and C are synchronized with their epoch numbers. Note that in practice the number of the cached blocks should be much larger than the number of forwarding candidate blocks.

warding target. There are two considerations, namely *how to evaluate the utilization of the client cache*; and, *how to make the utilization information available to other clients*.

Again the epoch number makes this comparison possible. Because the epoch numbers of the cached blocks represent their locality strengths, we use the smallest epoch number (least locality strength) associated with a cached block in a client to represent the utilization of the client.

The example shown in Figure 3 helps explain the rationale of this choice. In the figure there are three clients, A, B, and C, with their recently accessed blocks ordered in their respective LRU stacks. The blocks in different epochs (epochs n, $n+1$, and $n+2$) are separated by dotted lines. The utilization values of clients A, B, and C are n, n, and $n+2$, respectively. Client A is active and has many accesses. However, most recently accessed blocks have weak locality. They leave the cache soon after they are accessed and become shadow blocks in the stack. Then the epoch numbers of the cached blocks become relatively large. Even though they are still strong locality blocks locally, their locality become weak relative to the cached blocks in client C. Accordingly, the cache utilization of client A is deemed low.

Next we consider client B. It does access its strong locality blocks, so its recently accessed blocks are cached. However, it is less active than client C, which can be observed from the small number of blocks in the recent epochs. Either accessing the weak locality blocks in client A or being inactive in client B causes the spaces held by the cached blocks to be underutilized. Comparatively, client C actively accesses its cached blocks, making its cache spaces more fully utilized. Note that utilization is a relative metric. For example, if all three clients are inactive, the epoch counter will be updated slowly. Then the cache utilization of block B would be considered high.

To make cache utilization information available to other clients we adopt an approach similar to the one used in the hint-based protocol [11]. Each client maintains a list of what it believes to be the current cache utilization value of each of the other clients. The list is sorted by utilization value. A forwarding candidate is then forwarded to the client that has the smallest utilization value in the list if the locality strength value of the candidate block is smaller than the utilization value (both values are represented by epoch numbers). Otherwise, the block is discarded. To maintain the accuracy of the utilization information stored at each client, we allow two clients to exchange their utilization values when one client forwards a block to the other. However, the local utilization at a client could be stale. So the target client could see a forwarded block with its utilization lower than the client utilization. It that case it will abort the forwarding.

3.4. Locating Blocks among Clients

To reduce the control message overhead we use hints to locate blocks in remote clients [11]. Whenever a client successfully forwards a block to another client, it records the target client as the location hint for that block. The next time a client attempts to retrieve the block, it will follow the hints to reach the remote copy.

4. Performance Evaluation and Analysis

We compare LAC with two representative cooperative caching protocols: *N-Chance* and *Hint-Based*, as well as *Baseline* cache management where cooperative caching is not used. We use trace-driven simulations for the evaluation. Our simulator tracks the status of all accessed blocks, monitors the requests and block hits/misses seen at each cache client and server. We assume 8 KB cache block. In the experiments we only consider reads and assume asynchronous writes. The epoch size threshold is set at 50 by default.

We use three large-scale real-life traces to drive the simulator:

1. **httpd** was collected on a 7-node parallel web-server at University of Maryland [12]. The size of the data set served was 524 MB which is stored in 13,457 files. The collection was conducted over 24 hours. A total of about 1.5M HTTP requests were served, delivering over 36GB of data.
2. **db2** was collected at University of Maryland by an eight-node IBM SP2 system running an IBM DB2 database that performed join, set, and aggregation operations [12]. The total data set size was 5.2GB and was stored in 831 files. The total execution time is 7,688 seconds.

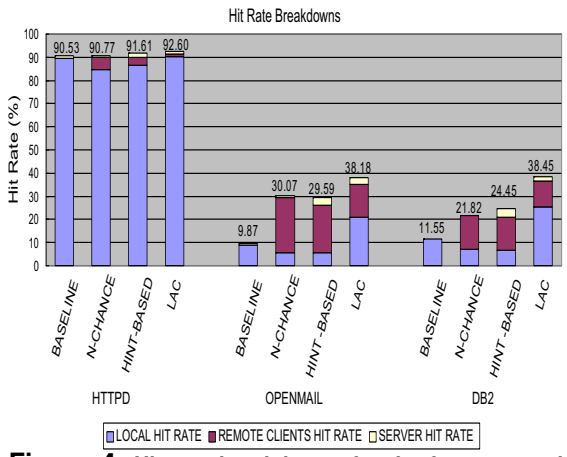

Figure 4. Hit rate breakdowns for the four protocols (*Baseline*, *N-Chance*, *Hint-Based*, and *LAC*) on the the three workloads (*httpd*, *openmail* and *db2*).

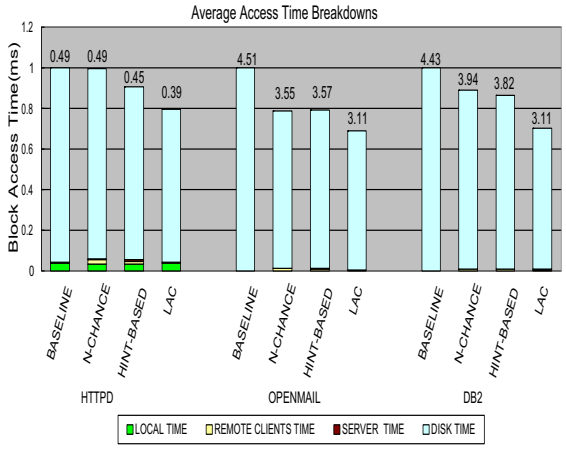

Figure 5. Average block access time breakdowns for the four protocols (*Baseline*, *N-Chance*, *Hint-Based*, and *LAC*) on the the three workloads (*httpd*, *openmail* and *db2*).

3. **openmail** was collected on a production e-mail system running the HP OpenMail application for 25,700 users, 9,800 of whom were active during the hour-long trace [14]. The system has 6 HP 9000 K580 servers running HP-UX 10.20. The size of the data set accessed by all six clients was 18.6GB.

We present hit rates as well as average block access times resulted from the simulations. In calculating the average access times, we assume a 0.02ms for a local block access, 0.2ms for access to a remote client cache or a server cache, and a 5ms for a disk block access. While these values are not specific to any particular system, their relative magnitudes are representative of current technology and do not qualitatively affect the conclusions drawn by comparing the relative performance of the different protocols.

Because of the widely different working set sizes of the access streams of the traces, we adjust the client cache sizes to capture the differences in behavior of the protocols.[2] Specifically, the cache sizes of each client for *httpd*, *db2*, and *openmail* are 80MB, 400MB, and 800MB, respectively. The server cache size is set at half of the total client cache sizes.

We run the four protocols (*Baseline*, *N-Chance*, *Hint-Based*, and *LAC*) on the three workloads (*httpd*, *openmail*, and *db2*), and present their general performance results. Figure 4 shows a breakdown of hit rates (local hit rate, remote client hit rate, and server hit ratio) for each protocol on each workload. Accordingly, Figure 5 shows the breakdowns of the average block access times. To clearly show the results for all the workloads, we normalized the access times to the time of the baseline protocol in Figure 5. From the figures we make the following observations.

(1) *LAC* demonstrates a significant performance improvement over *Baseline*. Its performance advantage is most obvious in the cases of weak locality. For example, because *openmail* and *db2* have weak localities,[3] their hit rates in *Baseline* are especially low (9.9% for *openmail* and 11.6% for *db2*), causing greatly increased block access times (4.5ms for *openmail* and 4.4ms for *db2*). *LAC* is able to significantly increase their hit rates: to 38.2% for *openmail*, and to 38.5% for *db2*, with concomitant reduction in their block access times: 31.0% for *openmail* and 29.8% for *db2*. The ability of *LAC* to effectively handle weak locality is especially important for a workload with many accesses of use-once data: because *LAC* can discard the blocks quickly without forwarding because of their infinite reuse distance, other protocols might allow these blocks to be forwarded among clients several times before they are finally discarded. For the strong locality workload *httpd*, a significant reduction in the block access time is observed with *LAC* (reduced by 20.3%) though the hit rate increase is small (from 90.5% to 92.6%): this is a consequence of the large latency gap between disk and memory accesses. The performance improvements are largely attributable to client cache sharing and the more effective management of local cache by using reuse distance. This is evidenced by the growth of the bars for local hit rate and remote client hit rate in Figure 4.

(2) While *N-Chance* and *Hint-Based* perform better than *Baseline*, there is a conflict between local caching and space

2 For example, for repeated sequential access of a file, there would be no insight to be gained by making client caches as large or larger than the file itself.

3 Detailed analysis of the localities of the chosen traces shows that most client caches have the hit rates less than 20% until the cache sizes increase to 1000MB for *openmail*, and to 600MB for *db2*, and that *httpd* has strong locality [14].

sharing among clients which limits the performance potential of cooperative caching. When we carefully compare the hit rates between *N-Chance* (or *Hint-Based*) and *Baseline*, we observe that the local hit rates of both *N-Chance* and *Hint-Based* are lower than those of *Baseline* for all three workloads. Even though the losses of local hits are compensated by the remote client hits, and the total hit rates are actually increased, this is undesirable and degrades performance for two reasons: (1) the latency to access local cache and remote caches is non-uniform; and, (2) some clients could experience unnecessarily interference because parts of their working sets are forced out by the forwarded-in blocks. This interference is effectively removed in *LAC* because *LAC* uses reuse distance to quantify locality and makes a comparison with the cache utilization of the target client before it decides to cache a block. Only those blocks with relatively strong locality are forwarded, and only truly weak locality blocks leave the cache by forwarding. This is in contrast with the other two protocols that use age information, which allows weak locality blocks to 'dilute' the target cache and to lower its utilization.

(3) A large server cache (half of total client cache sizes in our experiments) does not help much in increasing the hit rates, possibly counter-intuitively. Careful examination of the cached blocks in the server reveals that this is because of redundant caching between the clients and the server—most blocks cached by the server are also cached by clients. The low server cache utilization has been dealt with in client-server protocols such as ULC [9, 14]. Even if the server cache issue were effectively addressed, it would not reduce the demands on cooperating caching at the client side because of its good scalability and the server performance bottleneck concern.

5. Conclusions

We have presented a novel cooperative caching protocol based on the effective evaluation of block locality and cache space utilization. It overcomes the weaknesses of existing protocols in selecting the blocks to forward and of the clients' choice to accept the blocks. The real-life trace-driven simulations show significant improvements over existing representative protocols.

In the experiments with real-life workload traces, we show that LAC can reduce block access time by up to 31.0% with an average of 27.1% over the system without peer cache cooperation, and reduces the time by up to18.6%, with an average of 14.7%, over the best performer of the existing schemes, in both cases using local memory, remote memory, and disk access times with relative magnitudes representative of current technology. In addition, the LAC protocol implements judicious cache sharing in the coordination management, while the existing schemes cause excessive interference among peer clients, which is highly undesirable in a resource sharing system, and so heretofore a strong argument against the use of cooperative caching.

6. Acknowledgement

We thank anonymous reviewers for their comments and suggestions. This research was supported by Los Alamos National Laboratory under grant LDRD ER 20040480ER, and partially supported by the National Science Foundation under grants CNS-0098055 and CNS-0405909.

References

[1] T. Anderson, M. Dahlin, J. Neefe, D. Patterson, D. Roselli, and R. Wang. Serverless network file systems. In *SOSP'95*, 1995.

[2] M. Dahlin, C. Mather, R. Wang, T. E. Anderson, and D. A. Patterson. A quantitative analysis of cache policies for scalable network file systems. In *SIGMETRICS'94*, 1994.

[3] M. Dahlin, R. Wang, T. E. Anderson, and D. A. Patterson. Cooperative caching: Using remote client memory to improve file system performance. In *OSDI'94*, 1994.

[4] C. Ding and Y. Zhong. Predicting whole-program locality through reuse distance analysis. In *PLDI'03*, 2003.

[5] C. Fang, S. Carr, S. Onder, and Z. Wang. Reuse-distance-based miss-rate prediction on a per instruction basis. In *ACM SIGPLAN Workshop on Memory System Performance*, 2004.

[6] M. J. Feeley, W. E. Morgan, F. H. Pighin, A. R. Karlin, H. M. Levy, and C. A. Thekkath. Implementing global memory management in a workstation cluster. In *SOSP'95*, 1995.

[7] S. Jiang, F. Chen, and X. Zhang. CLOCK-Pro: An effective improvement of the clock replacement. In *USENIX'05*, 2005.

[8] S. Jiang and X. Zhang. LIRS: an efficient low inter-reference recency set replacement policy to improve buffer cache performance. In *SIGMETRICS'02*, 2002.

[9] S. Jiang and X. Zhang. ULC: A file block placement and replacement protocol to effectively exploit hierarchical locality in multi-level buffer caches. In *ICDCS'04*, 2004.

[10] D. Nagle, D. Serenyi, and A. Matthews. The panasas activescale storage cluster - delivering scalable high bandwidth storage. In *ACM/IEEE SC'04*, 2004.

[11] P. Sarkar and J. Hartman. Hint-based cooperative caching. *ACM Trans. Comput. Syst.*, 18(4):387–419, 2000.

[12] M. Uysal, A. Acharya, and J. Saltz. Requirements of i/o systems for parallel machines: an application-driven study. Technical report, University of Maryland at College Park, College Park, MD, USA, 1997.

[13] G. M. Voelker, E. J. Anderson, T. Kimbrel, M. J. Feeley, J. S. Chase, A. R. Karlin, and H. M. Levy. Implementing cooperative prefetching and caching in a globally-managed memory system. In *SIGMETRICS'98*, 1998.

[14] T. M. Wong and J. Wilkes. My cache or yours? making storage more exclusive. In *USENIX'02*, 2002.

[15] Y. Zhong, M. Orlovich, X. Shen, and C. Ding. Array regrouping and structure splitting using whole-program reference affinity. In *PLDI'04*, 2004.