New Algorithms for File System Cooperative Caching

Eric Anderson, Christopher Hoover, Xiaozhou Li

HP Laboratories
HPL-2010-69

Keyword(s):

cooperative caching

Abstract:

We present two new cooperative caching algorithms that allow a cluster of file system clients to cache chunks (i.e., fixed-size portions) of files instead of directly accessing them from (potentially remote) origin file servers. The first algorithm, Cooperative-LRU, moves a chunk's position closer to the tail of its local LRU list when the number of copies increases, instead of leaving the position unchanged as in the existing Distributed-LRU algorithm. The second algorithm, RobinHood, explicitly targets chunks cached at many clients for replacement when forwarding a singlet (i.e., a chunk cached at only one client) to a peer, instead of selecting a peer at random as in the existing N-Chance algorithm. We run these algorithms on a variety of workloads, including several publicly available traces. We evaluate these algorithms' loads on different components of the system. We show that the new algorithms significantly outperform their predecessors. Finally, we examine some workload properties that affect how well cooperative caching algorithms can perform.

External Posting Date: July 21, 2010 [Fulltext] Approved for External Publication
Internal Posting Date: July 21, 2010 [Fulltext]
To be published at the 18th IEEE Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS), Miami, Florida, August 17-19, 2010

New Algorithms for File System Cooperative Caching

Eric Anderson Christopher Hoover Xiaozhou Li
Hewlett-Packard Laboratories
{eric.anderson4, ch, xiaozhou.li}@hp.com

Technical Report HPL–2010–69

Abstract

We present two new cooperative caching algorithms that allow a cluster of file system clients to cache chunks (i.e., fixed-size portions) of files instead of directly accessing them from (potentially remote) origin file servers. The first algorithm, Cooperative-LRU, moves a chunk's position closer to the tail of its local LRU list when the number of copies increases, instead of leaving the position unchanged as in the existing Distributed-LRU algorithm. The second algorithm, RobinHood, explicitly targets chunks cached at many clients for replacement when forwarding a singlet (i.e., a chunk cached at only one client) to a peer, instead of selecting a peer at random as in the existing N-Chance algorithm. We run these algorithms on a variety of workloads, including several publicly available traces. We evaluate these algorithms' loads on different components of the system. We show that the new algorithms significantly outperform their predecessors. Finally, we examine some workload properties that affect how well cooperative caching algorithms can perform.

1 Introduction

Cooperative caching [1, 2] creates a scalable aggregate cache for a centralized file server (or servers) by using some memory from each client. Later this idea was extended to work for NUMA multiprocessors [3] and web caching [4, 5]. Similar ideas such as data diffusion are used in data-intensive computing [6]. Cooperative caching is complementary to the location-aware computing done by systems like MapReduce/Hadoop, and it addresses the general problem of caching mostly-read data across a cluster of machines.

Despite the wide applicability of cooperative caching, only a small number of cooperative caching algorithms are known (Section 2), and much design space remains unexplored. In this paper, we present two new, simple, and effective algorithms for file system cooperative caching. They are suitable for an architecture where one or more centralized *directory servers* keep track of the locations of each piece of cached data *chunk* (i.e., a piece of data of fixed size, say, 32KB) [1, 7]. The clients inform the directory server about which chunks they cache locally, ask the directory server which other clients are caching other chunks, and respond to requests for chunks from each other (Section 3).

The first algorithm is called Cooperative-LRU (C-LRU), a simple and intuitive extension of the standard Distributed-LRU (D-LRU) algorithm (Section 4). The idea is that when a client requests a chunk from another client, a new copy of this chunk will be created. Therefore, the importance of the chunk in both clients should be reduced. As the clients run LRU as their local caching algorithm, this idea translates to moving both copies of the chunk closer to the tail (i.e., the least-recently-used end) of their respective LRU lists. This adjustment results in sooner eviction of both copies, and compensates for the space inefficiency caused by the additional copy.

The second algorithm, called RobinHood, is a refinement of the N-Chance algorithm [1], and has a large potential performance gain (Section 5). In N-Chance, when a singlet is evicted from a client's cache, it is forwarded to a random peer and displaces the LRU chunk in that peer. In RobinHood, a singlet is forwarded in a more targeted manner. When a directory server replies to a query from a client as to whether a chunk is a singlet in the affirmative, it also tells the forwarding client which client (called *victim client*) to forward the singlet to and which chunk (called *victim chunk*) in the victim client to displace. The victim chunk is chosen to be one of the "rich" chunks (i.e., those cached at many clients) and the victim client is one that currently caches a copy of the victim chunk. By displacing "rich" chunks with "poor" chunks (i.e., singlets), the overall space efficiency is improved.

We implement these new and old algorithms in a prototype system called TiColi and we use simulation to evaluate them on a variety of workloads, including several real traces that are publicly available [8] (Section 6). We evaluate these algorithms in terms of the loads on the origin server, the directory server, and peers. Our results show that both C-LRU and RobinHood significantly outperform their predecessors on many workloads. We also examine some workload properties that affect how well cooperative caching algorithms can perform.

2 Related Work

The idea of cooperative caching is pioneered by Dahlin et al. [1] and Leff et al. [2]. The architecture used in these papers has a manager keep track of the cached chunks' locations and inform clients where to look for chunks. TiColi also uses this architecture. As the manager might become a bottleneck, other architectures have been proposed. Shark [9] is a distributed file system that uses a DHT to assist cooperative caching among clients. Although this system eliminates the need for central managers, look ups in such a system typically take $\Theta(\log n)$ hops, where n is the number of nodes in the system. Since low latency is one of TiColi's design goals, we expect that a DHT-based approach will not meet our latency goals; a simpler approach of partitioning the directory server would be sufficient to scale it, and would not increase the latency.

Depending on whether an algorithm requires periodic client-to-client exchanges, cooperative caching algorithms can be divided into two classes. Those that do not include Distributed-LRU (D-LRU) and One-Copy-Heuristic (1CH) by Leff et al. [10], and N-Chance by Dahlin et al. [1]. The algorithms proposed in this paper also belongs in this class, and we are only comparing our algorithms with other algorithms in this class. D-LRU is the straightforward extension of LRU in the cooperative environment, and 1CH maintains the invariant that at most one copy of any chunk exists in the architecture. However, if two clients repeatedly request the same chunk, 1CH may incur significant peer reads. The N-Chance algorithm allows a singlet (i.e., a chunk that is cached at one client) to be forwarded to another client for a few times, instead of being immediately discarded, because re-acquiring a singlet is costly.

The other class of algorithms require periodic client-to-client exhanges to allievate the load on the manager. However, if the number of clients is large, full client-to-client exchanges are also expensive operations. For example, if there are 1000 clients, then each exchange incurs about one million client-to-client messages. GMS [7] aims to make a group of small client caches to behave like a big global LRU cache. GMS is similar to N-Chance in that a locally evicted singlet is forwarded to another client, but it differs from N-Chance in that a chunk is forwarded to a peer that probabilistically has more globally old chunks. In order for a client to know which peers have more old chunks, clients periodically exchange the age information of their local chunks. In a sense, RobinHood is similar to GMS in that forwarding in both algorithms are more targeted than random forwarding, but they differ in how they pick the target. Furthermore, RobinHood does not require periodic client-to-client exchanges.

Sarkar and Hartman [11] propose using hints, which are (possibly stale) local information about the global system state, to assist cooperative caching. Clients exchange hints to locate missing chunks among each other, rather than consulting a global manager, which provides facts (as opposed to hints) about the system. The main benefit is the removal of the manager, which makes the system more scalable. The main weakness is the decrease in accuracy, which may result in more work and longer latencies. All algorithms proposed in this paper are manager based.

Jiang et al. [12] present a cooperative caching algorithm that mainly explores the dimension of how to handle eviction of chunks. The idea is to use a metric that can compare the global value of the various chunks and the utilization of the various client caches. Using this metric, a chunk is forwarded to a peer with a lower utilization, and the forwarded chunk replaces a chunk with a lower global value. The suggested metric is called *reuse distance*, the number of other distinct chunks accessed between two consecutive accesses to the chunk. Reuse distance is a local property computed at each client. In order to be able to compare the reuse distance of chunks in different clients, clients need to perform periodic synchronization.

3 TiColi System Overview

We assume an environment where machines are under a single administrative domain and security issues are not our concerns. Consequently, machines can freely cache, request, and receive data from each other. We focus on workloads that largely consist of reads.

3.1 System architecture

TiColi is a large-scale, file-level cooperative-caching system. A large number (hundreds to thousands) of Linux workstations called *clients* are connected by high-speed (Gigabit) Ethernet. Each client has a nontrivial amount of main memory on the order of many MBs that can be used as part of the cooperative cache. The clients access data from one or more file servers called *origin servers*. File servers can be slow either because of excessive load, or because they are remotely accessed. To ameliorate both of these problems, we aim to reduce the frequency that clients have to access data from the origin server. Because our cache operates at the file level and files may be directly updated, we track and validate file meta-data.

TiColi intercepts all file operations at the kernel level. The VFS (Virtual File System) layer in Linux allows a file system module to intercept system calls that are made to files under a particular mount point. FUSE (File system in USEr space) [13] is one such file system module that intercepts and forwards file operations to user space so that the calls can be acted upon by code in user space rather than code directly in the kernel.

To track the locations of cached data, TiColi places near the clients a dedicated *directory server* that keeps a map from chunk ids to a list of clients that *may* currently cache the data. When a

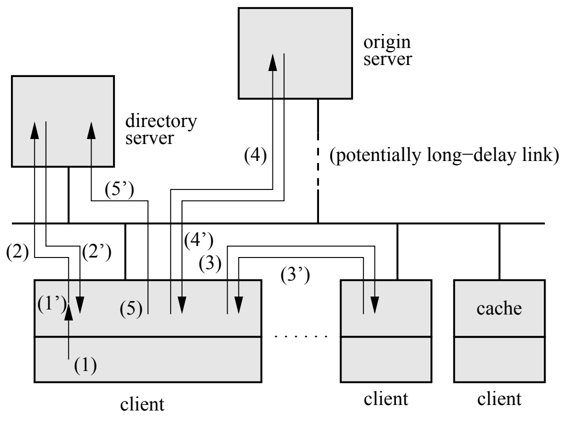

Figure 1: TiColi system architecture.

client changes its cached contents (e.g., obtaining new chunks, evicting existing chunks), it updates the directory server. The directory server also serves to cache attributes and negative entries that are retrieved by clients.

Figure 1 shows a typical path for a client to read a chunk. The client requests (1) the chunk. The data is not found in the local cache (1'), so the client sends a message (2) to the directory server, asking if the chunk is being cached by other clients. The directory server replies (2') with a (possibly empty) list of clients that may be caching the chunk. If the list is not empty, the client sends a request (3) to one of the clients in the list, asking for the data in question. The other client replies (3') with the chunk if it still has it. If the list is empty, the client requests (4) the chunk from the origin server. The origin server sends back (4') the chunk. The client caches the chunk (5) and sends a message (5') to the directory server, informing it of the addition of the chunk to the client's local cache.

3.2 The directory server

Clients keep the directory server (approximately) up-to-date about their cache contents. Whenever a client adds a chunk to or evicts a chunk from its cache, it informs the directory server. The directory server uses a two-level hashing data structure that consists of a file table and multiple chunk tables. The file table maps hashes of file names to client tables. The chunk tables map chunk numbers to lists of clients. The directory server removes those clients that it has not heard from for a long time (which may indicate that the client is either congested or crashed) from all lists. It responds (WhoHasChunkReply) to each request (WhoHasChunkRequest) for the location of a chunk with a random permutation of the list of clients that *may* be caching the chunk. It prunes the list to a pre-specified length (10 in current implementation) if the list is too long.

3.3 Rationale for this architecture

Given the description above, it might appear that the directory server could be a bottleneck of the system. However, we decided on this architecture for two reasons. The first is latency. A main objective of TiColi is to reduce the user-perceived latency for accessing data. When not overloaded, a designated server provides quick and accurate locations for data requests. The second is feasibility.

A well-provisioned modern server can keep track of all locations of all chunks in all clients in the cluster within main memory and maintain tens of thousands of concurrent network connections. Should the directory server indeed becomes a bottleneck, we can configure multiple directory servers by partitioning the file name hash space. This can be done straightforwardly because the different directory servers need not coordinate with each other.

We can illustrate this with a simple calculation: suppose there are 1,000 clients and each client accesses 10,000 files. Hence, the total number of files is ten million. Since the size of a file hash is 20 bytes and each pointer is 4 bytes, the size of the file table is about 24×10,000,000=240MB. Suppose each file is 1MB on average, each chunk is 32KB (i.e., each file has 32 chunks), and each chunk is cached at 10 clients on average. A chunk number takes 4 bytes and each client id takes 4 bytes (i.e., IPv4 address). Therefore, the size of each chunk table is (4+10×4)×32=1.5KB. The total size of all the chunk tables is 1.5KB×10,000,000=15GB. In current days, a directory server can be configured with hundreds of GBs of memory. Therefore, it is feasible for a single directory server to support a large number of clients. We have not investigated further optimizations such as storing two-byte client ids instead of full IP addresses or partitioning the directory server.

3.4 Read and write operations

As TiColi operates on file level, it handles all kinds of file operations such as getattr, open, readlink, remove and so on. For a cooperative caching algorithm, the two important operations are reads and writes. The read path for TiColi has been described in Figure 1. Requesting a chunk from other clients can be done either sequentially (less bandwidth but increased latency) or in parallel (less latency, but increased bandwidth). The default is requesting from at most four peers in parallel.

The write path is roughly as follows. The client takes two steps: it send an indication to the directory server that the cached copies must be re-validated, and then the client writes the data to the origin server. The directory server, upon receipt of this indication, sends an indication to all the clients known to cache the file that their cached copies of the file are stale. Whenever a client receives such a message, it invalidates its cached status information and clears its cache of any chunks associated with the file. As a simplification, any write operation in TiColi invalidates the entire file, rather than the affected chunks. This simple approach is based on the assumption that writes are relatively rare compared to reads and therefore need not be optimized. It also leaves the origin server responsible for ordering concurrent writes. Furthermore, Since we allow for writes outside of TiColi, we have to validate file attributes against origin server. When TiColi writes, the attributes will change unpredictably. Therefore we have to do whole file invalidation since we can't tell if someone else happened to write at the same time.

4 The C-LRU Algorithm

In the next two sections, we present two new algorithms: C-LRU and RobinHood. Both algorithms try to reduce the replicas in the client caches. Although this heuristic is intuitive, it does not guarantee a better performing algorithm under all circumstances. To see why, consider the following example. Suppose algorithm α keeps chunk x at clients A and B, but algorithm β only keeps x at B. Suppose B keeps accessing new data and consequently pushes x out, and then client C requests x. For β, this situation leads to a miss, but not for α (because x is in A). That said, reducing replicas is still an effective heuristic, as will be demonstrated by results on several workloads later

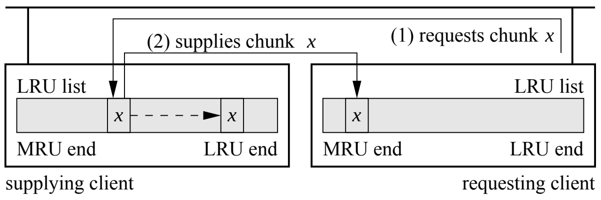

Figure 2: The C-LRU algorithm.

in the paper.

4.1 Algorithm

We begin with considering the simple D-LRU algorithm described in Leff et al. [10]. In this algorithm, a client responds to local accesses exactly the same way as local LRU does. On the other hand, a client responds to remote accesses by providing the chunk but keeping its local LRU list unchanged.

The C-LRU algorithm is based on a simple observation: the more copies that a chunk has in the client caches, the less valuable each copy is: if a new copy of a chunk is to be created, then the values of the existing copies should be decreased. Therefore, it makes sense not to put the new chunk at the head of the local LRU list (i.e., the most-recently-used or MRU end), but rather a place close to the head, so that this new chunk can be evicted sooner. Likewise, upon a remote access, it makes sense to move the supplying copy closer to the end (or the LRU end) of the LRU list. Figure 2 depicts this algorithm.

4.2 Heuristics

One can use several heuristics to make the adjustments for both the requesting client and the supplying client. There are two kinds of heuristics.

The *local heuristics* are for a requesting client to place the new chunk near, but not at, the head of the local LRU list. An obvious choice is to place the new chunk in the middle of the LRU list. We call this variant the C-LRU-L algorithm and will evaluate it in our experiments.

The *remote heuristics* are for a supplying client to move the requested chunk closer to the end of the local LRU list. An obvious choice is to directly move it to the end of the LRU list. We call this variant the C-LRU-R algorithm and will evaluate it in our experiments. An advantage of this heuristic is that it is trivial to implement and incurs no overhead.

We stress that the above are only some of the more obvious heuristics. There are many more. For example, a client can immediately evict a chunk once it is requested by a peer. This heuristic is similar to the one-copy heuristic [10], but allows the chunk to migrate from client to client. However, we feel that this heuristic is too aggressive in terms of evicting chunks and might cause high client-to-client traffic.

4.3 Implementation

The implementation of C-LRU-R is straightforward, but to implement C-LRU-L, we need the ability to quickly “jump” from one location of a linked list to another location some distance away. To do this efficiently, we use the skip list [14] data structure where we maintain the lengths of the

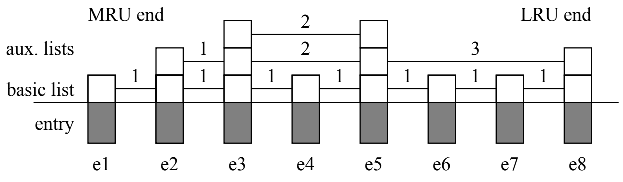

Figure 3: Skip list.

"long jump" links, as shown in Figure 3. Going from one position in the list to another simply follows the longest jump at each hop without over-jumping (by observing the link lengths). With high probability, all operations on a skip list take logarithmic time and space overhead is constant per cache entry (see [14] for a detailed analysis of the skip list's performance properties). (Remark: However, as we will discuss in Section 6, C-LRU-L is inferior to C-LRU-R in terms of reducing miss rates. But without the skip list implementation, C-LRU-L will run so slowly that we wouldn't be able to obtain any simulation results.)

5 The RobinHood Algorithm

5.1 Algorithm

We first give an overview of the N-Chance algorithm, upon which RobinHood is based. In N-Chance, when a chunk is about to be evicted from a client cache, the client queries the directory server to find out if the chunk is a singlet (i.e., there is only one copy of this chunk in all client caches). If not, the chunk is simply discarded. If so, the chunk is forwarded to a random peer, which treats the forwarded chunk as if it has just been locally accessed and adds it to the head of its local LRU list, evicting a local chunk if necessary. To prevent singlets from remaining in the caches forever, each chunk is associated with a *recirculation count*, a small integer (set to two in our current implementation), that is decremented each time a chunk is forwarded. A singlet is evicted when its recirculation count drops to zero.

The RobinHood algorithm also forwards singlets. In addition to identifying singlets, the directory server also selects a *victim chunk*, one cached at many clients, and a *victim client*, one that currently caches a copy of the victim chunk. The directory server then sends back the victim client and the victim chunk id to the forwarding client. The forwarding client sends the singlet and the victim chunk id to the victim client. Upon receiving this message, the victim client replaces the victim chunk with the singlet. If the directory server cannot find a victim chunk, then the RobinHood algorithm falls back to being N-Chance, namely, the forwarding client forwards the singlet to a random peer.

Intuitively, the advantage of RobinHood over N-Chance is that the former uses a singlet to replace a rich chunk, yet N-Chance uses a singlet to replace an LRU chunk at a random peer. The price to pay for RobinHood is keeping track of the copies of the various chunks. RobinHood can also be made to trade-off the overhead of book-keeping and the aggressiveness of reducing replicas. For example, we can keep track of only those chunks that have at least ten copies and fall back to N-Chance when there are no such chunks. In our current implementation, we keep track of all chunks.

On the other hand, an advantage of N-Chance is its ability to utilize inactive clients' caches. By forwarding singlets to random peers, N-Chance makes use of idle caches. In contrast, by targeting

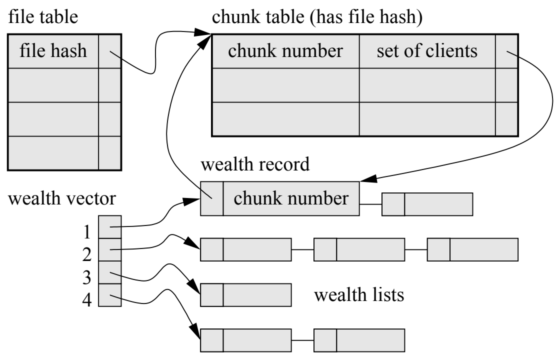

Figure 4: Data structures for the RobinHood algorithm.

rich chunks, RobinHood only forwards chunks to active clients. However, when most of the clients are active, this advantage of N-Chance is not significant.

Several aspects of the RobinHood algorithm merit fine tuning. For example, when the directory server finds multiple clients that are currently caching the victim chunk, how does it pick one out of the multiple candidates? Our current implementation deterministically picks the head of the candidate list. This choice is based on the crude heuristic that the head of the list is the client that needs the victim chunk least because in our implementation, when the directory server receives an IHaveChunk message from a client, it puts the chunk id at the back of the list of clients that are currently caching that chunk. Our experiments found that picking the head tends to do better than picking a random client from the list.

5.2 Implementation

Data structures. The RobinHood algorithm requires the directory sever be able to select a rich chunk quickly. We use a vector of lists to implement the RobinHood algorithm. Besides the file table and chunk tables, we add a vector of lists, called the *wealth vector*, where element i of the vector is a list that contains the wealth records for chunks that are cached at i clients. A *wealth record* contains a chunk number and a pointer to a chunk table, which is augmented with the file hash so that a wealth record need not keep the file hash. These data structures are shown in Figure 4.

Operations. To pick one of the richest chunk and a client that contains that chunk, the RobinHood algorithm first finds the highest numbered non-empty list from the wealth vector. It then picks a chunk from this list and then pick a client from the set of clients that are currently caching the chunk. The most accurate method is to pick a chunk x and a client A such that x is least likely to be needed by A in the near future. However, there is no easy way to tell. We use some heuristics to pick a chunk and a client. To pick a chunk, our current implementation uses the simple and efficient heuristic of picking the first chunk in a wealth list. To pick a client, our current implementation picks the head of a list of clients. When the count of a chunk increases (i.e., the directory server receives an IHaveChunk message), the algorithm can quickly locate the wealth record via the file table and chunk table. The algorithm then removes the record from its current list and adds it to the next list (e.g., from list five to list six). Similarly for decreasing the count of a chunk.

Space and time complexities. For each chunk entry maintained at the directory server, we need to add a few pointers and a chunk number. The search for a rich chunk and a client is amortized constant time, because the wealth vector can become sparse and the search for a non-empty wealth list can take non-constant time. Using a list (instead of a wealth vector) that dynamically removes empty wealth lists, we can achieve per operation constant time, but we have opted for the current implementation for simplicity. In practice, it is unlikely that the vector is sparse. The maintenance of wealth records (i.e., finding them and moving them between lists) is also constant time, assuming hashing (into the file table and the chunk tables) is constant time.

If one considers the above implementation too costly, then a simpler implementation could keep just one wealth list, instead of many, that contains the wealth records for those chunks that have at least some number of replicas (e.g., three). Whenever a chunk grows from two copies to three copies, it is added to the list, and whenever a chunks shrinks from three copies to two copies, it is removed from the list. The implementation is simpler, but less accurate because a forwarded chunk may not be displacing one of the richest chunks. In other words, there are other possible trade-offs between implementation and performance. But this difference may or may not be important. We have not compared the actual performance of these two implementations.

6 Evaluations

6.1 Simulation setup

In our simulations, we use a network topology as shown in Figure 5. The client-rack links are 1 Gbps. Each rack is connected to the core at 10 Gbps. All links at the core level are 2×10 Gbps. All switches (round boxes) have a latency of 1 microsecond. We modeled latency through switches and bandwidth contention on links in order to make the simulation more realistic during bursts of activity that could result in slower responses from the directory server or from peers.

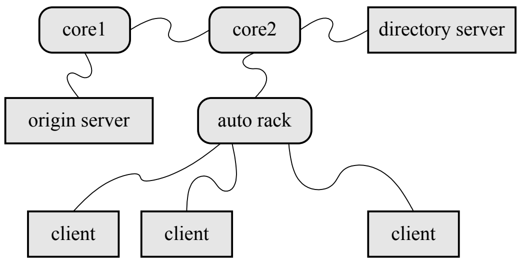

Figure 5: The simulated network topology.

A request for a chunk that is in the local cache or a peer cache is called a *hit*, otherwise it is called a *miss*. There are two kinds of misses. *Compulsory misses* (or *cold misses* for short) refer to those requests that ask for chunks for the first time and consequently, these requests have to miss. *Non-compulsory misses* (or *hot misses* for short) refer to those requests that ask for chunks that once were cached but somehow were evicted or invalidated (due to writes) subsequently. Miss rate, the ratio between the number of misses and the number of chunks requested, is the main criterion for evaluating the effectiveness of a caching algorithm and will be the focus of our experiments.

For each experiment, we calculate miss rates in the following way. We count how many chunks are read by all the clients altogether, let this number be R. We ignore the chunks requested by writes because all writes are misses (i.e., write-through). The directory server keeps a set of chunks

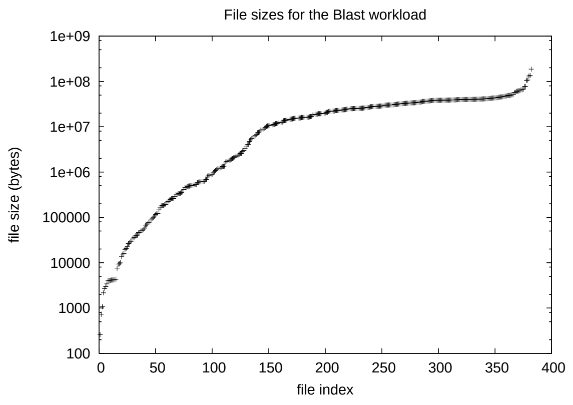

Figure 6: File size distribution for the Blast workload.

that have ever been cached in any client's cache. It collects this information using the IHaveChunk messages from the clients. If a client asks WhoHasChunk(chunk-id) and this chunk-id had been cached before but is not currently cached, then the hot miss count, denoted by H, is incremented by one; otherwise the cold miss count, denoted by C, is incremented by one. We present the combined miss rate $(H + C)/R$, the hot miss rate H/R, and the cold miss rate C/R.

Counting hot misses this way means that a read for an invalidated chunk (due to a previous write) actually incurs a hot miss. An alternative way is not to count such misses (because there is nothing a write-invalidate algorithm can do about such misses), but it involves more refined book-keeping. We have chosen to count in the simpler way as most workloads are read dominant and the same counting method is applied to all the algorithms.

We have evaluated eight algorithms in our experiments, and we will focus on the the following four: D-LRU (the standard Distributed-LRU algorithm), C-LRU-R (the C-LRU algorithm where the remote heuristic is move-to-end, N-Chance (the standard N-Chance algorithm, the recirculation count is set to two), and RobinHood (the RobinHood algorithm, the recirculation count is also set to two).

We have also evaluated the following four algorithms, but they do not perform as well in terms of reducing miss rates: C-LRU-L (the C-LRU algorithm where the local heuristic is move-to-middle; inferior to C-LRU-R). C-LRU-LR (the combination of C-LRU-L and C-LRU-R; not consistently better than C-LRU-R), [C-LRU-R, N-Chance] (the combination of C-LRU-R and N-Chance; inferior to N-Chance), [C-LRU-R, RobinHood] (the combination of C-LRU-R and RobinHood; inferior to RobinHood). An interesting discovery is that "combined algorithms" such as C-LRU-LR do not perform as well as their constituents. We conjecture that combined algorithms are too aggressive in terms of reducing replicas, which may result in pushing out useful chunks too soon. (See example at the beginning of Section 4.)

In what follows, unless otherwise specified, all experimental results are the average of five random runs. All data chunk sizes are 32KB.

6.2 The Blast workload

The Blast workload is motivated by applications in genetics where one wishes to find out if a DNA sequence (a long string) contains a certain short string [15]. The workload consists of two sets

workload	clients	dataset (DS)	read DS	write DS	num. of reads	num. of writes	duration
Blast	100	7.5GB	7.5GB	0	depends on run	depends on run	100 jobs/client
NFS-1/set-5	78	13.2GB	11.2GB	2.2GB	4086134	380265	2 hours
NFS-1/set-12	1394	19.3GB	16.7GB	2.7GB	8562579	592035	2 hours
NFS-2/set-2	233	60.5GB	59.8GB	0.7GB	15355592	300156	1 hour
NFS-2/set-5	134	39.2GB	38.8GB	0.4GB	20672418	360307	1 hour
Harvard	44	3.7GB	3.0GB	2.5GB	2338827	761141	12 hours

Figure 7: Workload characteristics.

of files: the small files (i.e., the short strings), and the big files (i.e., the DNA sequences). The set of small files contains a large number of files, each of which a few KBs. The set of large files contains a few hundred files, each of which tens of MBs. In each Blast job, a random small file (with replacement) is picked from the set of small files and a random large file (with replacement) is picked from the set of large files. Then the blast job determines if the small file is a substring of the large file. This involves reading both files into main memory. A Blast workload consists of many Blast jobs. So essentially a Blast workload is comprised of sequentially reading big random files. When a group of clients with cooperative caching are running the Blast workload, the hope is that they can share their caches in such a way that they do not have to read data from the origin server frequently. Note that in the Blast workload, there are no writes.

In our experiments, we make the following simplification in our simulation. Since the small files are inconsequentially small compared to the large files, we omit simulating the reading of the small files. That is, we only simulate the (sequential) reading of the big files.

In our experiments, we use an example of a real Blast workload. This example has 382 files; the total size of these files is 7714MB; the average size is 20.2MB; the smallest file is 259B; the biggest file is 187MB. The size distribution, in log scale, is shown in Figure 6. Figure 7 summarizes the characteristics of Blast and other workloads.

Figure 8 shows the miss rates (both hot and cold) for the algorithms under different total cache size (i.e., the sum of all the client caches) over dataset ratios. We can see that C-LRU-R performs considerably better than D-LRU. In fact, C-LRU-R performs even better than N-Chance (which will not be true for other workloads). The best of all algorithms is RobinHood, which drops the hot miss count to almost zero when the total cache size is slightly over the dataset size.

6.3 The Animation traces

The second workload we evaluate is the NFS traces from a feature animation company, which is publicly available at [8] and at the SNIA IOTTA Repository [16] (once enough space is available). We evaluate four traces, named NFS-1/set-5, NFS-1/set-12, NFS-2/set-2, and NFS-2/set-5. We choose these traces because they have been analyzed in the literature [17]. We convert the original traces to extract just the read and write operations, and to sort the reads and writes by request time. Our conversions are available at [18], and can be easily re-generated from the original traces. The workload characteristics of these traces are summarized in Figure 7. In the table, dataset (DS) size is the union of all the byte ranges of all read/write operations, and read DS and write DS are similarly defined. Due to the running time and computing resources needed to simulate these traces, our experiments run these traces for their first hours; durations are specified in Figure 7.

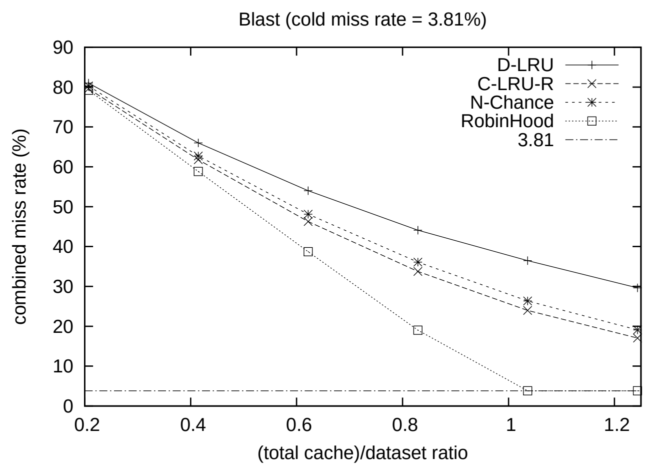

Figure 8: Miss rates for the Blast workload.

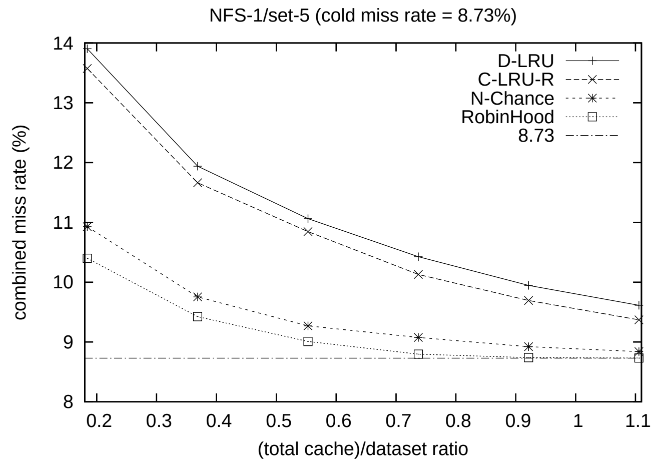

Figure 9: Miss rates for the NFS-1/set-5 trace.

Figure 9 shows the miss rates for NFS-1/set-5. We see that C-LRU-R improves upon D-LRU and RobinHood upon N-Chance both quite noticeably. In particular, RobinHood reduces the hot miss rate to almost zero when the cache size is about 0.75 of the dataset size.

Another angle to evaluate cooperative caching algorithms is to evaluate how many peer reads they incur. We call a read of a chunk from a peer a *peer read*. To be clear, cooperative caching strives to reduce origin reads, at the cost of increased peer reads. But still, we do not want peer reads to be excessive. To see how high peer reads can happen, consider the following example. Suppose there are three clients A, B, and C. Suppose algorithm α caches chunk x at A and algorithm β caches x at A and B. Further suppose that B and C read x a lot but A does not at all. Then the total peer reads in α is higher that in β. The peer reads for A in α is also higher because in α, all reads from B and C for x go to A but in β, reads from B can be satisfied locally. Neither algorithm incurs any miss.

Figure 10 shows the total peer reads for various algorithms for NFS-1/set-5. As expected, C-LRU-R incurs more peer reads than D-LRU, and RobinHood more than N-Chance. But the

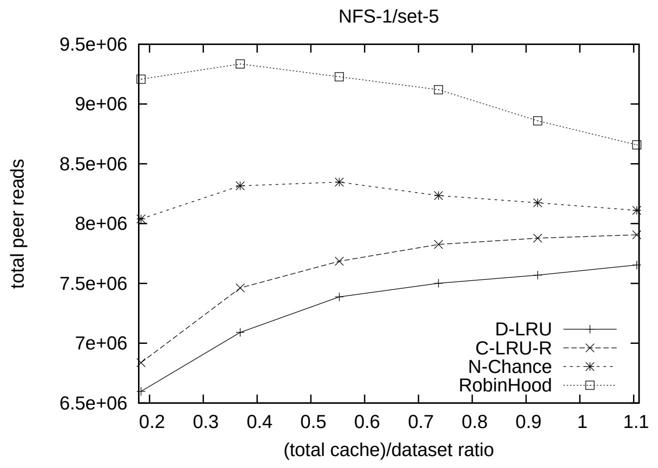

Figure 10: Total peer reads for the NFS-1/set-5 trace.

increase is not excessive. It is interesting to see how, for each algorithm, the peer reads change as the client cache sizes increase. On the one hand, peer reads may increase because more origin reads can be satisfied by peer reads, but on the other hand, peer reads may decrease because more data are cached locally. As we can see, different algorithms show different trends, but these trends are mostly modest, indicating that the above two opposing effects largely cancel out each other.

Figure 11 shows the maximum number of reads provided by any peer. This plot evaluates whether there is an "overwhelmed client" for peer reads. As we can see, as the total cache size increase, all algorithms cause some peer to provide more peer reads, but all algorithms are comparable in this regard.

Figure 12 shows, for an algorithm, this algorithm's peer reads minus D-LRU's peer reads, divided by D-LRU's hot misses minus this algorithm's hot misses. This ratio tells us, for each miss saved, how many peer reads are incurred by this algorithm compared to the base algorithm D-LRU. We note that, depending on the algorithm, this number can be either greater than or less than one. We see that N-Chance incurs the smallest ratio. In contrast, RobinHood incurs a ratio about twice as large. This difference is somewhat expected because RobinHood is more aggressive in terms of reducing duplicates. It is easy to see that having duplicates can help decrease peer reads. However, we believe that trading three to five peer reads for one origin read is a good trade-off. Curiously, C-LRU-R incurs ratios higher than other algorithms.

Figures 13 and 14 show the number of messages received and sent by the directory server, respectively. Compared to D-LRU, C-LRU-R incurs virtually no additional messages, either received or sent. Both N-Chance and RobinHood incur more messages than C-LRU-R or D-LRU, which is expected because the former two algorithms involve the IsThisSinglet requests and replies, which are not needed by the latter two. Compared to N-Chance, RobinHood incurs slightly more sent messages but noticealy more received messages.

Figure 15 shows the miss rates for the NFS-1/set-12 trace. This trace has a lot of clients, 1394 of them (see Figure 7). For this trace, C-LRU-R improves upon D-LRU only modestly. However, RobinHood continues to considerably outperform N-Chance. Figure 16 shows the miss rates for the NFS-2/set-2 trace. For this trace, the improvements of the two new algorithms are more noticeable. We also see that the cold miss rates are slightly different for different algorithms. If one plots only

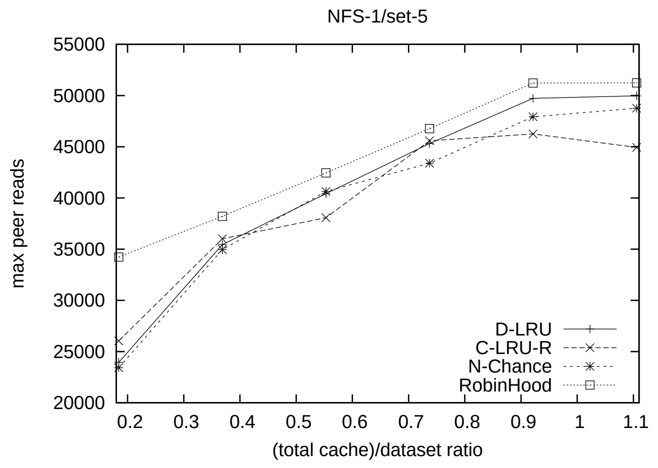

Figure 11: Max peer reads for the NFS-1/set-5 trace.

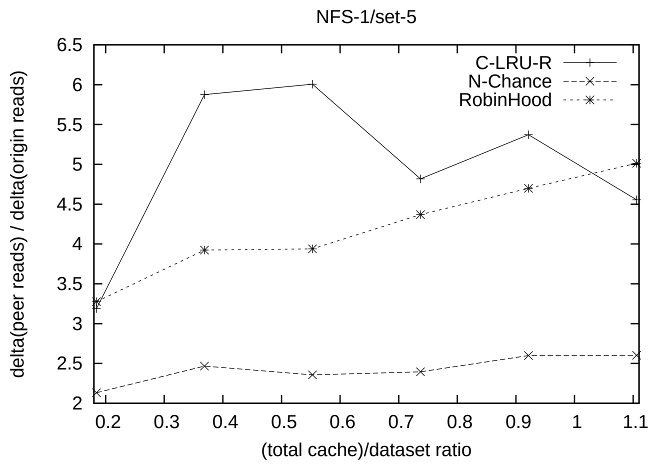

Figure 12: Delta(peer reads)/delta(origin reads) for the NFS-1/set-5 trace.

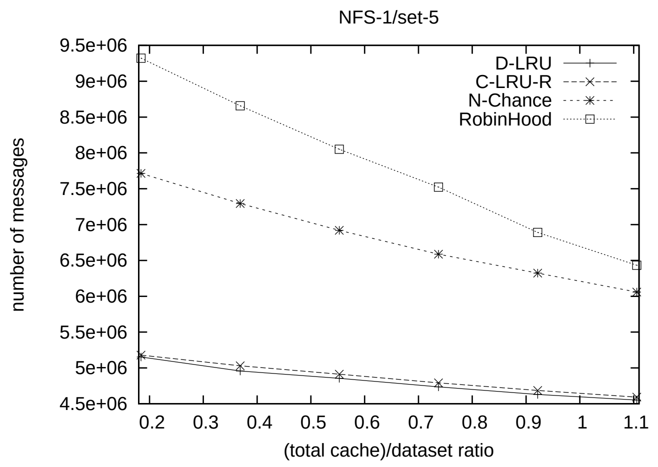

Figure 13: Directory server received messages for the NFS-1/set-5 trace.

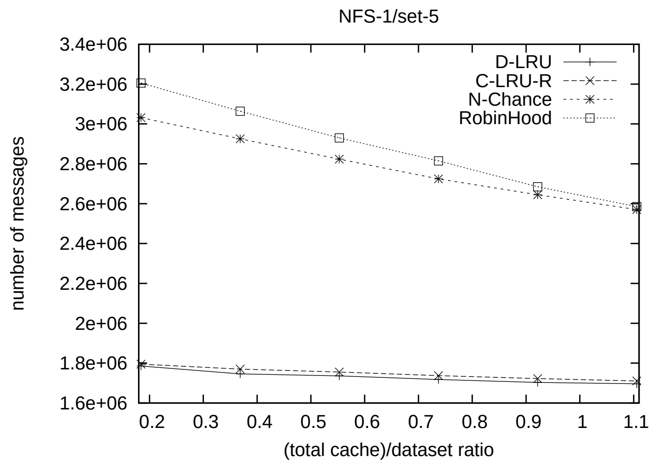

Figure 14: Directory server sent messages for the NFS-1/set-5 trace.

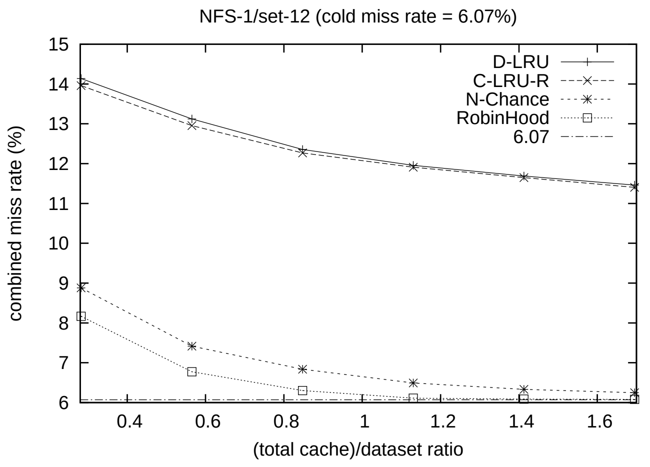

Figure 15: Miss rates for the NFS-1/set-12 trace.

the hot miss rates, the gaps among the algorithms remain similar. Due to space limitations, we omit showing that graph. Figure 17 shows similar differences among the algorithms for the NFS-2/set-5 trace.

We notice that when the total cache size is around the dataset size, RobinHood usually keeps the hot miss count to a fairly low number, indicating that there may not be much room for improvement for RobinHood, because a single cache as big as the dataset size would incur no hot misses.

6.4 The Harvard trace

We have also experimented our algorithms on a Harvard trace, one of the "Ellard traces" (Deasna, 21 October 2002), which is publicly available at [8, 16] and our converted version is at [18]. On this trace, the new algorithms and the old algorithms perform nearly identically; hence, we omit the exact numbers. We believe there are two reasons. First, there are many writes in this trace (see Figure 7). As mentioned earlier in the paper, we have not tried to optimize for the writes in our

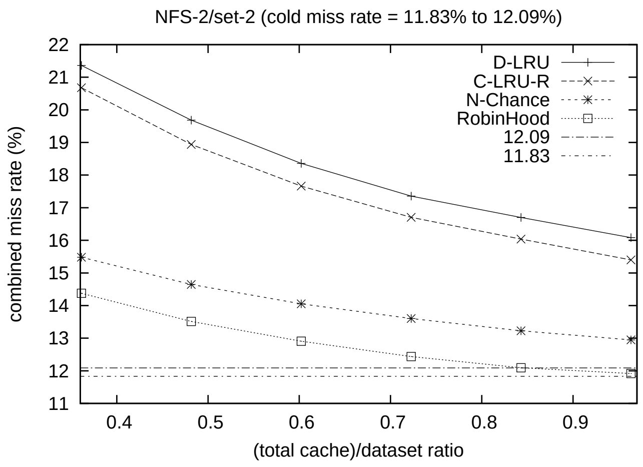

Figure 16: Miss rates for the NFS-2/set-2 trace.

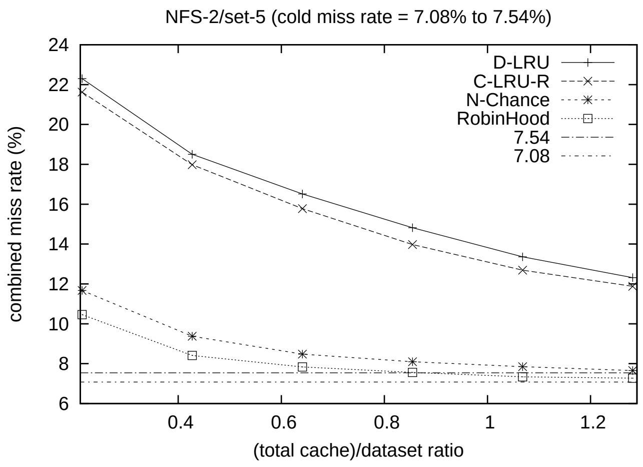

Figure 17: Miss rates for the NFS-2/set-5 trace.

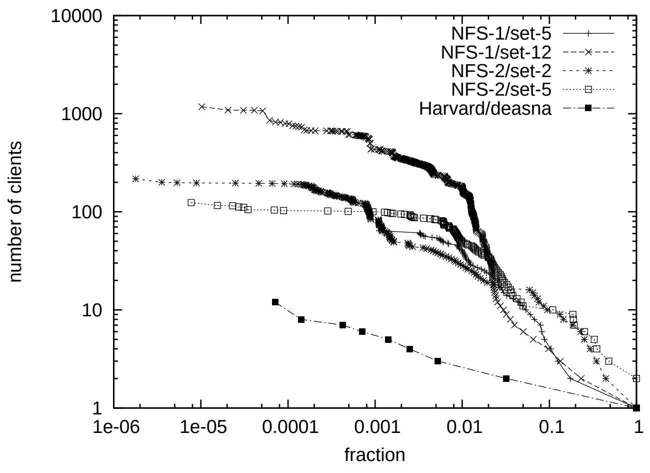

Figure 18: Commonality of files (both axes log scale).

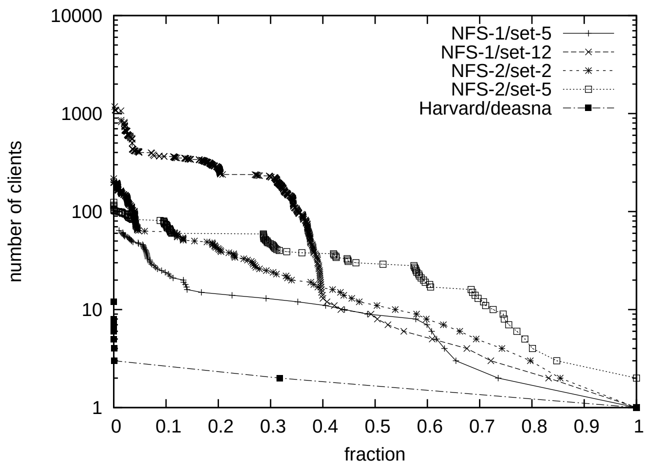

Figure 19: Commonality of accesses (only y-axis log scale).

algorithms. Second, there is very little sharing of data among the clients (there are only 44 clients anyhow).

To confirm these conjectures, we analyze the traces, including the Harvard trace and the animation traces, as follows. We investigate the *commonality* of the files, that is, for each file accessed (read or written) by a trace, how many clients access that file throughout the trace. We call a file that is accessed by exactly k clients during an entire trace a *k-client file* and a file accessed by at least k clients a *k^+-client file.* Note that two k-client files can be accessed by two different sets of k clients. We also investigated, for each access (a read or a write), whether the access is made to a common or uncommon file.

Figure 18 shows the distribution of the commonality of files in various traces. A data point (a, b) in this plot means "a% of the files are b^+-client files." For example, for the Harvard/deasna trace, only about 0.75% of the files are accessed by 3 or more clients, but about 50% of the files are accessed by 3 or more clients in the NFS-1/set-5 traces. In both cases the point is the third tick-mark up, but it is the leftmost and rightmost points respectively. Overall, we can see that, for the Harvard trace, most of the files are accessed by only one client. In contrast, the animation traces have much higher file commonality.

Figure 19 shows the distribution of the commonality of accesses in various traces. A data point (a, b) in this plot means "a% of the accesses are made to b^+-client files." We can see that, for the Harvard trace, about 68% accesses are made to 1-client files. In contrast, the animation traces have much higher access commonality.

We have also taken snapshots of the directory server periodically and notice that very few chunks are cached at multiple clients. In contrast, for the animation traces, it is common for many chunks to be cached at tens of clients. It is also possible that the lack of common data is due to the frequent writes. Since the two new algorithms aim at reducing multiple cached copies of the same chunk, they will not be effective if there are no such chunks. After all, cooperative caching is meaningful only if there is something to cooperate on.

7 Concluding Remarks

In this paper, we have proposed two simple yet effective cooperative caching algorithms, C-LRU and RobinHood. Compared to their predecessors D-LRU and N-Chance, respectively, these two new algorithms perform considerably better on a wide range of workloads. We note that the main novelty in these two new algorithms is the coordination techniques, not the local caching algorithms. As such, the new techniques can in principle be used with local caching algorithms other than LRU (e.g., LFU). In this paper, we have chosen to use LRU as the local caching algorithm because of LRU's simplicity and effectiveness, and LRU has been shown to work well even in non-local caching contexts (e.g., [19]). It would be interesting to investigate whether certain coordination techniques work particularly well with certain local caching algorithms.

Acknowledgements

We are thankful to our colleagues Lucy Cherkasova for comments on the paper and suggestions on workload characteristics, Terence Kelly for several valuable discussions, James Saxe for comments on how to implement the algorithms, Joseph Tucek for comments on the algorithms and how to run the experiments.

References

[1] 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 *Proc. 1st OSDI*, November 1994, pp. 267–280.

[2] A. Leff, P. S. Yu, and J. L. Wolf, "Policies for efficient memory utilization in a remote caching architecture," in *Proceedings of the First International Conference on Parallel and Distributed Information Systems*, December 1991, pp. 198–207.

[3] C. K. Kim, D. Burger, and S. Keckler, "An adaptive, non-uniform cache structure for wire-delay dominated on-chip caches," in *Proc. 10th ASPLOS*, October 2002, pp. 211–222.

[4] L. Fan, P. Cao, J. Almeida, and A. Z. Broder, "Summary cache: A scalable wide-area web cache sharing protocol," *IEEE/ACM Transactions on Networking (TON)*, pp. 281–293, June 2000.

[5] A. Wolman, G. M. Voelker, N. Sharman, N. Cardwell, A. Karlin, and H. M. Levy, "On the scale and performance of cooperative Web proxy caching," in *Proc. 17th SOSP*, December 1999, pp. 16–31.

[6] I. Raicu et al., "The quest for scalable support for data-intensive workloads in distributed systems," in *Proc. 18th HPDC*, June 2009, pp. 207–216.

[7] M. J. Feeley, W. E. Morgan, F. H. Pighin, A. R. Karlin, and H. M. Levy, "Implementing global memory management in a workstation cluster," in *Proc. 15th SOSP*, December 1995, pp. 201–212.

[8] "NFS Traces," http://apotheca.hpl.hp.com/pub/datasets/.

[9] S. Annapureddy, M. J. Freedman, and D. Mazieres, "Shark: Scaling file servers via cooperative caching," in *Proc. 2nd NSDI*, May 2005, pp. 129–142.

[10] A. Leff, J. L. Wolf, and P. S. Yu, "Efficient LRU-based buffering in a LAN remote caching architecture," *IEEE Transactions on Parallel and Distributed Systems (ITPDS)*, vol. 7, no. 2, pp. 191–206, February 1996.

[11] P. Sarkar and J. H. Hartman, "Hint-based cooperative caching," *ACM Transactions on Computer Systems (TOCS)*, vol. 18, no. 4, pp. 387–419, November 2000.

[12] S. Jiang, K. Davis, F. Petrini, X. Ding, and X. Zhang, "A locality-aware cooperative cache management protocol to improve network file system performance," in *Proc. 26th ICDCS*, July 2006, p. 42.

[13] "FUSE: Filesystem in Userspace," http://fuse.sourceforge.net.

[14] W. Pugh, "Skip lists: A probabilistic alternative to balanced trees," *Communications of the ACM (CACM)*, vol. 33, no. 6, pp. 668–676, June 1990.

[15] "BLAST: Basic Local Alignment Search Tool," http://blast.ncbi.nlm.nih.gov.

[16] "SNIA IOTTA NFS Traces," http://iotta.snia.org/traces/list/NFS.

[17] E. Anderson, "Capture, conversion, and analysis of an intense NFS workload," in *Proc. 7th FAST*, February 2009, pp. 139–152.

[18] "Cache Simulation Subset," http://apotheca.hpl.hp.com/pub/datasets/cache-sim/.

[19] B. Reed and D. Long, "Analysis of caching algorithms for distributed file systems," *ACM SIGOPS Operating Systems Review*, vol. 30, no. 3, pp. 12–21, July 1996.