

NFS-cc: Tuning NFS for Concurrent Read Sharing

Ying Xu and Brett D. Fleisch

Abstract—A common file access pattern found in cluster applications is concurrent read sharing: applications running on multiple sites read access the same dataset concurrently. Traditional network file systems are limited by the server’s network bandwidth; therefore cannot satisfy the high-bandwidth concurrent reads that cluster applications typically require. This paper presents NFS-cc: a cooperative caching file system based on NFS and optimized for concurrent read sharing. Cooperative caching used by NFS-cc is a distributed file caching scheme, which allows file system clients to read data from the memory of other clients, instead of the server. If multiple clients cache the same data, they can supply the data to other clients concurrently, which greatly reduces the contention for the server’s network bandwidth. NFS-cc exploits clients’ bandwidth and file caches to improve the aggregate read throughput of the file system. Our preliminary performance measurements show that the aggregate read throughput of NFS-cc was increased with the number of clients and reached as high as 42.6 MB/s with 12 clients connected by a fast Ethernet. NFS-cc also reduced the average block read time by a factor of 50-80% under a variety of concurrent read access patterns compared with NFS.

Index Terms—Cooperative Caching, Network File System, Concurrent Read Sharing, Cluster

I. INTRODUCTION

TRADITIONAL network file systems, such as NFS [7], Andrew [9] and Sprite [8], are designed around a central server model. The advantage of this model is centralized data management. For example, backups can be more easily performed at a centralized server. However, as more clients are added or files are accessed concurrently, the server quickly becomes a performance bottleneck. Many applications that execute in the traditional network file system model support a request-response style interaction rather than parallel execution. Parallel applications running on clusters usually require concurrent access to shared files from multiple sites. Traditional network file systems limited by the server’s network bandwidth cannot satisfy this requirement well;

therefore parallel file systems [6, 10, 11, 12] have been designed to provide concurrent file access for parallel applications. Parallel file systems increases I/O parallelism by striping file data to multiple disks either through shared-disk Storage Area Network (SAN) or across multiple sites on a network, for example, IBM GPFS [6] and PVFS [11].

In this research we investigate a new design space rather than striping data to improve file system performance under concurrent read access workloads. Specifically, we introduce a prototype system—NFS-cc, which exploits redundant copies of file system data blocks that already exist in clients’ cache and utilizes the bandwidth of clients to transfer the data blocks to other clients. NFS-cc, meaning NFS with *cooperative caching*, is based on NFS and is optimized for concurrent read sharing.

Cooperative caching [1] is a caching scheme that differs greatly from that of traditional network file systems. The caching scheme of traditional network file systems can be characterized as client/server caching. File read accesses that don’t hit in a client’s local cache will be sent to the server. The server serves the requested data if the read access hits in its cache; otherwise, it has to issue an expensive disk I/O to bring the data to its memory before serving it. In contrast to client/server caching, cooperative caching enables clients to read data from the memory of other clients, instead of the server. It allows requests not satisfied by a client’s local cache to be satisfied by the cache of another client. Cooperative caching reduces contention for the server’s bandwidth by exploiting the bandwidth of clients: clients are coordinated to serve the requests for others. If multiple clients cache the same data, they can serve the data to other clients simultaneously, which greatly improves the overall file system throughput. Cooperative caching also reduces the number of disk accesses at the server because data no longer cached in the server may still be cached somewhere in a client’s cache. Therefore, the data can be served from a client’s cache somewhere on the network without expensive disk access.

New technology trends are increasing the potential advantage of cooperative caching. First, memory prices continue to decline making it possible for computer systems to continue to increase memory capacities and offer new services that were once memory-limited. The added memory not only allows clients to cache more data in their local

Manuscript received March 23, 2004. This work was supported in part by the Office of Naval Research under grant N00014-01-1-0854.

Ying Xu and Brett D. Fleisch are with the Dept. of Computer Science and Engineering, University of California, Riverside, CA 92521-0304 (e-mail: {yxu,brett}@cs.ucr.edu).

memories, but also increases the possibility for data requests to be satisfied by the coordinated caches of clients. Second, recent technology advances in high bandwidth and low latency networks allow data to be transferred very efficiently between systems. Fetching data from remote memory through a high bandwidth and low latency network, such as Myrinet or Gigabit Ethernet, is 10-20 times faster than reading data from disk. So when a server cache miss occurs, it is much faster to for a client read data from the cache of other clients rather than to wait for the server to make an expensive disk access and bring the data to memory. With the coming of 10 Gigabit Ethernet [14] and Infiniband [15], we could expect fetching data from remote memory to be two orders of magnitude faster than fetching data from disk, which greatly increases the payoff of cooperative caching. Finally, current technologies are creating powerful clients. Even commodity desktop computers are equipped with processors that execute at exceptionally fast clock rates. In a typical cluster, powerful clients with more computing power are able to easily saturate the server. According to our experiments, just two clients can saturate the bandwidth of a NFS server if they are all connected to a fast interconnection network (100Mb/s). Powerful clients add more burden to the server thus limit the scalability of the central file server systems. But in cooperative caching system, powerful clients can also be utilized to serve the requests of others, which offer the possibility of greater system scalability as well as increased performance.

Previous work on cooperative caching, such as Dahlin’s N-chance forwarding [1] and Sarkar’s hint-based cooperative caching [2], focused on remote page replacement algorithms, which utilize the memory of idle clients to avoid expensive disk accesses at the file server. The primary methodology of previous work was trace-driven simulation. Our research differs from previous work because we implemented a prototype cooperative caching file system by extending an in-kernel NFS implementation of Linux and we focus our research on the effectiveness of using cooperative caching to reduce the contention for the server’s bandwidth, which was not analyzed by previous researchers. Our work is complementary to the previous work and the major contribution of our work is to show:

- 1) It is possible to extend an existing network file system to support cooperative caching with reasonable effort. The code added to NFS is about 1200 lines, including the changes of NFS client (300 lines), NFS server (400 lines), and RPC protocol (500 lines).
- 2) Cooperative caching used in NFS-cc and the other techniques we developed for NFS-cc can greatly increase the aggregate read throughput and reduce average block access time under a variety of concurrent read sharing workloads when data are being shared among clients.

The rest of the paper is organized as follows. Section II discusses the design decisions and assumptions that we made

for NFS-cc. Section III describes the detailed cooperative caching scheme in NFS-cc. We present the performance measurements and analysis in Section IV. Section V compares our work to related work, and Section VI summarizes our conclusions and future work.

II. DESIGN DECISIONS AND ASSUMPTIONS

Like previous work on cooperative caching [1, 2, 3, 4], the cooperative caching scheme in NFS-cc is read-only caching. We extend the READ protocol of NFS for our system improvements but we don’t change the WRITE protocol. All modified data blocks are still sent back to the server as they otherwise would be without cooperative caching. Therefore, if a site caching a requested data block fails, the requesting site simply fetches the data from the server.

In order to allow requests not satisfied by a client’s local cache to be satisfied by the cache of another client, we must track the locations of file system data blocks. We use a centralized data structure called *cache directory* to track the data block locations. The cache directory indicates which clients store the specified data block in their local cache. The cache directory is maintained by the file server and the server is a centralized control point as it is in traditional client/server systems. Having a centralized cache directory simplifies our design and enables the server to forward data requests from one client to other clients efficiently. Adding more clients doesn’t add too much load to the server because the added clients can also be leveraged to serve the requests for the server. By coordinating the file caches of all clients, most of the disk read accesses in the server are avoided and instead of serving large data blocks the server only sends small forward messages and receives small notifications and acknowledgements. The extra work added to the server is searching and maintaining the caching directory, which can be easily overlapped with the I/O activities by the operating system. Thus the extra work isn’t expected to affect the performance of the server very much but the overall throughput of the system could be improved.

In the traditional client/server model, clients only trust the server. But in cooperative caching systems, clients are able to read data from the memory of other clients. Therefore, clients must trust each other’s kernel not to send them corrupted data. We assume all clients in the cooperative caching system are equally secure: they are all administered in the same way and protected from outside by firewalls. This is a common assumption in most cluster environments and may also be applied to a group or department’s administrative domain.

III. COOPERATIVE CACHING SCHEME IN NFS-CC

A. Scenarios

The cooperative caching scheme of NFS-cc is illustrated in

the following scenarios.

1) Scenario 1: Server Serves the Request

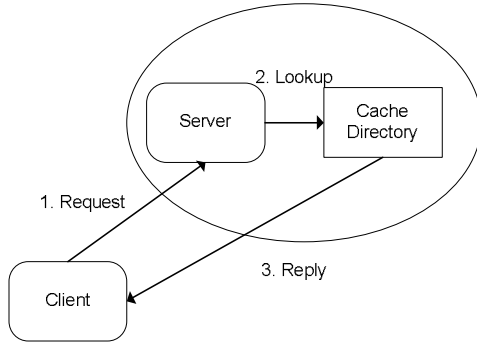


Figure 1. Server serves the request.

In scenario 1 (Figure 1), a client sends a request to the server; the server queries its cache directory. If no client is caching the data, the server checks its local cache. If the requested data block is not in its local cache, the server reads the data from disk, stores it in its local cache and sends it to the client. Otherwise, the server directly sends the data from its local cache. After sending the data block, the server creates a new cache directory entry indicating that the requesting client caches the data block.

2) Scenario 2: Server Forwards the Request

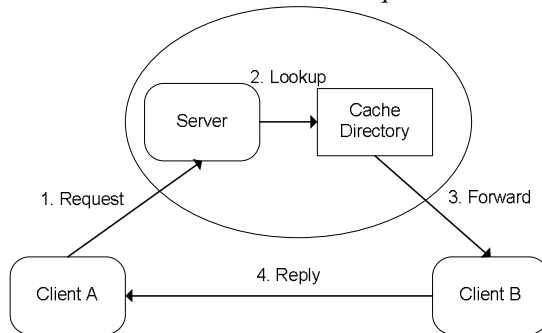


Figure 2. Server forwards the request.

In scenario 2 (Figure 2), Client A sends a request to the server; the server looks up its cache directory and finds there is another client caching the data block. If more than one client caches the data block, the server selects one according to its forwarding policy. If the server chooses Client B, for example, it forwards the request to B.

In this scenario NFS-cc uses an *optimistic strategy* to update the cache directory. When the server forwards the request to Client B, it immediately updates the cache directory indicating Client A caches the requested data block. The assumption is that Client A will receive the data block eventually. The optimistic strategy has two advantages over a *conservative strategy*, which delays the update of cache directory until Client A actually receives the data block. First, in the optimistic strategy clients save a message sent to the server because it doesn't have to send an acknowledgement to the server once it receives the data block. Second, the optimistic strategy allows NFS-cc to better serve concurrent requests. When the server receives multiple concurrent requests for the same data blocks, using a conservative strategy the server has to serve the concurrent requests itself

until it receives acknowledgements that clients have stored the data block. In this case, the server loses the opportunity to forward the requests to clients to reduce the contention for its network bandwidth.

Because the optimistic strategy allows the server to update the cache directory before a client actually receives a data block from another client, it is possible for the server to forward a request to a client that itself is also in the process of obtaining requested data. Therefore, after receiving the forwarded request, Client B not only checks whether the requested data block exists in its local cache but also checks the status of the cached data block. Client B has three possible actions depending on what it finds:

- 1) If B has already cached the requested data block, then B simply sends the data to A;
- 2) If B has reserved the memory space for the requested data block but the data block has not arrived yet then B suspends serving the forwarded request until it completely obtains the data block;
- 3) Otherwise, if Client B has discarded the data block, it sends a negative reply to Client A telling A that it doesn't cache the data block.

Upon receiving the negative reply or after it has not received any reply for a timeout period, Client A will resend the request to the server. This time the server will immediately supply the data block to Client A and also updates the cache directory to indicate Client B doesn't cache the data.

3) Scenario 3: Page Discard Notification

One problem in Scenario 2 is that a request has been forwarded to a client but the client no longer caches the data block because the block's space has been reclaimed. Consequently, the server forwards the request to a client that does not store the block because the cache directory is not up-to-date. The result is that incorrect information in the cache directory increases the latency of the request. If every time a client discards a data block it sends a notification to the server, this action will keep the cache directory much more accurate so the server will hardly ever forward a request to a client that does not cache the requested data block but it also adds considerable load to the server. There is a tradeoff between the accuracy of the information and the overhead of distributing and maintaining this information. We don't send a message for each discarded data block but rather instead batch the updates and use a technique which we call *on-demand* update to solve this problem. The batched update and the on-demand update are described in the following paragraphs:

Every client maintains a queue which records the pages that it once cached but has discarded. When the queue becomes full, the client sends a notification to the server telling it what pages have been discarded from the client's local memory. The server then updates its cache directory.

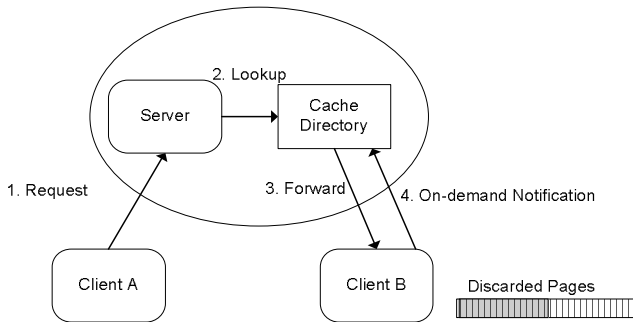


Figure 3. On-Demand Page Discard Notification.

There is another situation when clients send page discard notifications on-demand as shown in Figure 3. Client A sends a request to the server. The server’s cache directory indicates that B caches the requested data, but actually B has already discarded it. Because its queue is not full, B hasn’t sent a notification to the server. So the server forwards a request to a client that does not store the data block. This time Client B sends a page discard notification on-demand including all the records in the queue of discarded pages to the server, which stops the server from making the erroneous forwards again.

B. Cache Consistency

NFS-cc is designed to preserve the consistency model of NFS and it reuses a large portion of code taken from NFS. Any application that works on NFS will continue to work NFS-cc without any modification but NFS-cc doesn’t guarantee more consistency than NFS. Like NFS, NFS-cc adheres to a stateless-server model [7]. The benefit of statelessness is that there is no need to do state recovery after a client or server has crashed and rebooted. However, since the server has no record of which clients are currently using a file, the server cannot make guarantees about cache consistency. The official NFS protocol specification [7] doesn’t provide any means of maintaining strict client-server cache consistency, but the so-called “reference port” implementation does imply a particular cache-consistency scheme, which only allows caches to be inconsistent for a small time window. NFS clients keep their caches consistent by periodically asking the NFS server whether a cached file has been modified. The timeout period is a tunable parameter set by NFS clients. Within the timeout period, the client is not required to ask the server again, but rather just accesses the cached file. In addition to periodical checking, most of the RPC operations on a file object return basic attributes of a file, including the modification time of a file. NFS clients also use the returned file attributes to revalidate the cached file. There are two situations that clients may get stale data accidentally:

- 1) If a client has stale data in its cache, the client may use this data because the timeout period has not expired and it doesn’t know the cached file has been modified.
- 2) A client has new data in its cache but it has not written this data back to the server yet. Therefore, if another

client asks the server for the same data, the server may return stale data because it does not know that one of its clients has a newer version of data.

To maintain the particular cache consistency model of NFS, in the cooperative caching scheme of NFS-cc, whenever the server forwards a request to a client, it also attaches the current file attributes of the requested file with the forward, as illustrated in Figure 4.

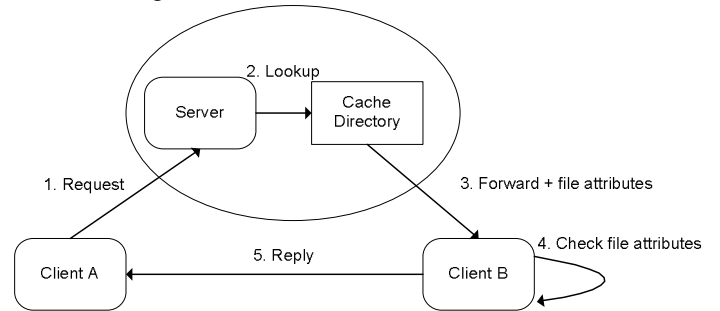


Figure 4. Server attaches file attributes with the forward to enforce cache consistency.

Client A sends a request to the server. After consulting its cache directory, the server forwards the request together with the file attributes of the requested file to Client B. Client B checks the file attributes of the requested file with the file attributes forwarded by the server. If the modification time of the requested file at Client B matches that of the server, Client B then sends the requested data block to Client A as it is explained in Section III-A, Scenario 2. Otherwise, if the modification time of the file at the server is greater than that of Client B, meaning Client B has stale data, Client B refreshes the file attributes of the requested file with the server’s file attributes, invalidates all the cached data blocks associated with the requested file, and sends a negative reply to Client A. In case that the modification time of the file at Client B is greater than that of the server, meaning the server has stale data, Client B will not update the file attributes of the requested file but send a negative reply to Client A only. Client A will not get the most update-to-date data blocks until the server obtains the block, which enforces the fact that Client A should not obtain data blocks from different versions of the file.

To prevent the server from forwarding requests to a client with stale data, whenever the server notices that a file in the exported NFS-cc file system has been changed, it will invalidate all cache directory entries that associated with that file.

C. Request Forwarding Policy

The server always tries to forward a request to the client that caches the requested data block. If multiple clients cache the same data block, the server must select one according to a *forwarding policy*.

NFS-cc uses a simply round robin policy. The server forwards requests to clients caching the requested data block one-by-one in a round robin fashion to balance the load of

clients. The round robin policy is per data block based. So the cache directory entry associated with each data block contains a pointer pointing to the client that serves the latest request for the data block. The server itself is also included in the round robin selections and it may get chances to serve the requested data if cached. To avoid expensive disk accesses at the server, if the server discards a page associated with a data block of the exported NFS-cc file system, it will update the cache directory to exclude itself from the round robin of serving the data block.

The round robin policy is easy to implement and doesn't require client side information. But for more advanced policies, to make a decision requires client-side information. For example, in the *adaptive policy*, the server forwards requests to clients according to their current status, such as CPU usage, unused bandwidth, and idle memory. Giving additional client side information to the server helps the server make a better decision, but on the other hand, the overhead of delivering additional information may limit the benefit of having up-to-date information. Therefore, how to design an advanced policy for NFS-cc requires further study.

D. Failures

NFS-cc preserves the stateless server model of NFS. The cache directory of NFS-cc keeps location information for each data block, which enables cooperative caching. Without a cache directory the NFS-cc server can still function as a normal NFS server but loses all the benefits of cooperative caching. Like NFS servers, there is no need for a NFS-cc server to do state recovery after the server has failed and rebooted. The NFS-cc server will operate as normal NFS server at the beginning and gradually rebuild the cache directory from scratch after a failure.

A client failure affects the performance of NFS-cc because requests forwarded to that client will no longer be served and the client that initiated the request must wait until a timeout occurs to resend the requests. To detect the client failure, NFS-cc reuses a data structure called *request cache* in the NFS implementation of the Linux kernel and maintains a *miss table*. One purpose of the request cache used in NFS is to detect duplicate data requests from the same client within a short period of time. Once a duplicate request is detected, the NFS server will simply drop it while the NFS-cc server will serve it if it has ever been forwarded to a client and the server will also increase the corresponding counter in the miss table, indicating the client that it last forwarded the request to missed a request. The counter will be cleared if the NFS-cc server receives any request or page discard notification from that specific client; otherwise if the counter exceeds a certain threshold the server will send a *ping* message to the corresponding client. If the client still doesn't response, the server will consider the client to be failed.

After having detected a client failure or receiving a mount request from a client, the server will first mark the client as

unavailable, stop forwarding requests to the client, and then start a thread to clean up the cache directory. The server will not allow the client to participate again until the cache directory returns to a consistent state, as indicted by the fact that the client doesn't cache any data blocks of the network file system exported by the server.

E. Implementation and Limitations

We implemented a prototype of NFS-cc by modifying an existing NFS version 3 implementation in Linux kernel 2.4.20. The original RPC protocol supports two messages types: CALL and REPLY. In order to allow the server to forward requests to clients, we added a new message type named FORWARD. The page discard notification is implemented as a new RPC call added to the NFS protocol. In order to support page discard notification, we also added some hooks to the Linux memory management system. Before each page is removed from the page cache, we check if it is associated with a data block in the file system exported by NFS-cc. If it is, then NFS-cc records the page into the queue of discarded pages. In NFS-cc, clients must also be able to serve requests for others. This is no problem because Linux kernels already include the NFS server code. Thus, we simply customize the NFS server code to implement a client-side daemon. The only job for the client side daemon is to serve the requested data block from its local cache efficiently if it caches it.

The cache directory of NFS-cc is organized as a hash table with collision resolved by chaining. The hash key for the cache directory is a pair of inode number and block numbers, which uniquely identifies a data block in a file system. Each cache directory entry corresponds to a file system data block. A bitmap in the cache directory indicates which clients cache the specified data block. To avoid running out of kernel memory, we put a limit on the total number of cache directory entries. This sets the maximum number of file system data blocks of which the NFS-cc server can track. All cache directory entries are linked to a LRU list, so the least recently used entries can be recycled.

One limitation of our prototype is that the server can only export one file system. In order to export multiple file systems, NFS-cc must set up one cache directory per exported file system.

IV. EVALUATION

A. Experimental Setup

We built a small cluster to test our NFS-cc prototype. The cluster has 12 clients, each client has one 400MHz CPU with 64 MB memory. The server is a 2G Hz Dell Dimension 4500 with 640 MB memory. All computers are connected to a fast Ethernet using a 16-port, full duplex Linksys switch. For all the experiments we set the length of NFS-cc's page discard

queue to be 400.

B. Overhead of NFS-cc Server

In order to support cooperative caching, NFS-cc servers must maintain and search their cache directories. The operations on cache directories involve overhead at the server. We used a modified version of the widely-used Andrew benchmark [9] to measure the overhead imposed on NFS-cc servers. It consists of five phases testing different aspects of file systems. The following description is taken from [9]:

<i>MakeDir</i>	Constructs a target subtree that is identical in structure to the source tree.
<i>Copy</i>	Copies every file from the source subtree to the target subtree.
<i>ScanDir</i>	Recursively traverses the target subtree and examines the status of every file in it; does not actually read the contents of any file.
<i>ReadAll</i>	Scans every byte of every file in the target subtree once.
<i>Make</i>	Compiles and links all the files in the target subtree.

We ran the benchmark to compare NFS-cc with the unmodified version of NFS in Linux kernel 2.4.20. During our experiments, no other jobs were run at the client and the server. We ran the benchmark ten times for both NFS and NFS-cc. The elapsed time shown in Table 1 is an average of the ten trials.

TABLE 1
MODIFIED ANDREW BENCHMARK RESULTS

Elapsed Time in milliseconds					
Phase	<i>MakeDir</i>	<i>Copy</i>	<i>ScanDir</i>	<i>ReadAll</i>	<i>Make</i>
NFS	74.6	3143.2	4151.3	4887.0	141030.7
NFS-cc	75.0	3117.6	4151.4	4883.4	139961.8
Overhead	0.48%	-0.81%	0.002%	-0.072%	-0.76%

The overhead of NFS-cc as showed in Table 1 is within the range of $\pm 1\%$, which should be considered within the realm of measurement error. The results confirm our expectations: for workloads that do not have concurrent sharing, like the Andrew benchmark, NFS-cc has a negligible performance penalty.

C. Aggregate Read Throughput

Concurrent read sharing is the file access pattern that NFS-cc is designed to be optimized for. We measure the aggregate read throughput under concurrent read sharing by using a simple benchmark called *concurrent copy*. During the concurrent copy benchmark, multiple clients copy the same file from the server flooding the server with requests for the same file from many clients. Typical NSF servers do not perform well when flooded with this many requests for the same blocks.

Our experiment was set up and carefully established. To eliminate other factors that may affect our benchmark results such as disk I/O performance at the server, we preload the file into the server’s cache. We set the size of the file used in concurrent copy to be 100 MB. We ran the concurrent copy benchmark for both NFS and NFS-cc varying the number of

clients from 1 to 12. As shown in Figure 5, the aggregate read throughput of NFS is limited to around 11.7 MB/s while the aggregate read throughput of NFS-cc increases with the number of clients and reaches 42.6 MB/s with 12 clients. NFS-cc improves the read performance of the file system greatly by a factor of 4 with 12 clients.

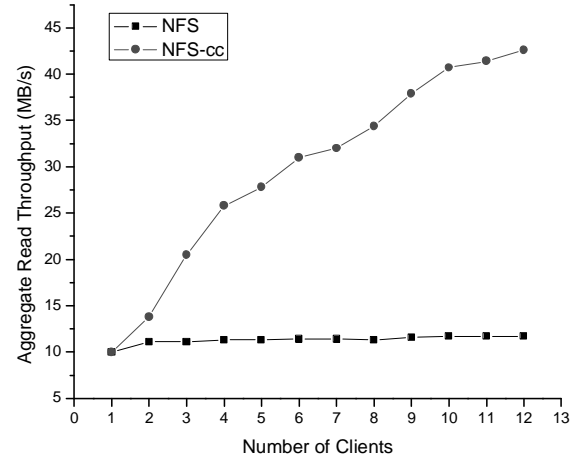


Figure 5. Aggregate Read Throughput

D. Synthetic Benchmarks

In order to analyze NFS-cc in more details we designed four synthetic benchmarks, each of which presents a typical concurrent read sharing pattern. The following are short descriptions for the synthetic benchmarks:

Sequential Makes sequential read request through the entire data set.

Random Makes random requests.

Hotcold-local The dataset is divided into a 20% “hot” region and an 80% “cold” region; 80% of the references go to the “hot” region. Within each region, the requests are random. Each client has its own “hot” region in this benchmark.

Hotcold-global Similar to Hotcold-local but all clients share the same “hot” region.

We used a large file in size of 100 MB as the dataset for all the benchmarks. The benchmarks are set to only make 4 KB read requests. The total number of read requests issued by each benchmark program is 25600. All benchmarks were conducted with 12 clients running the same benchmark program concurrently. Every time we ran each benchmark twice: the first run has empty client caches and the second run has client caches pre-warmed by the first run.

1) Average Block Read Time

Average Block Read Time is the average time required to read a file block. For network file systems, it is determined by the bandwidth of the file server as well as the hit rates in the different layers of the storage hierarchy. Since NFS-cc alleviates the contention for the server’s bandwidth under concurrent read sharing, it also reduces the average time for

applications to read a data block. We ran the four synthetic benchmarks on both NFS and NFS-cc and measured the average time to read a 4 KB data block under different concurrent read sharing patterns. As shown in Figure 6, NFS-cc reduced the average block read time by 50-80% as compared with NFS depending on the read access patterns. The hotcold benchmarks have smaller block read time than other benchmarks because the hot regions of hotcold benchmarks can be fit into the local cache of clients and most of the requests are satisfied by the local caches instead of sent to the server. But the sequential and random benchmarks benefit more from NFS-cc, reducing 60-80% of the average block read time, while the hotcold benchmarks only reduce about 50% of the average block read time. The local cache hit rates of the sequential and random access patterns are much lower than the hotcold patterns, therefore more read requests have been sent to the server. The sequential and random benchmarks cause more serious contention for the server's bandwidth than the hotcold benchmarks, which leaves more room for NFS-cc to reduce the average block read time.

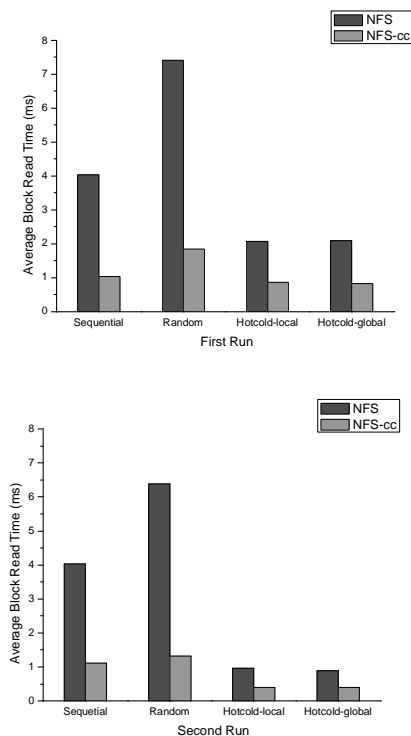


Figure 6. Average Block Read Time.

2) Hits and Misses

In the cooperative caching scheme of NFS-cc, every client is able to serve data requests for others. NFS-cc servers improve the aggregate read throughput by forwarding the requests to clients and exploiting the clients' bandwidth. When a client receives a request forwarded by a NFS-cc server, the client serves the requested data block if it caches the data block. The request is said to be a *hit* in this case. Otherwise, if the client doesn't cache the data block because its space has been reclaimed, the forwarded request is said to

be a *miss*. We measured the number of requests served by the server and the number of hits and misses at each client while running the synthetic benchmarks. Graphs of the number of requests served by the server and the number of hits at each client are shown in Figure 7. We can see in the graphs that the server successfully reduced its load by forwarding the requests to clients. The total number of requests served or forwarded by the server or clients in hotcold-local or hotcold-global benchmark is much smaller than that of sequential and random because the hot region of both hotcold benchmarks can be fit into each client's local memory and 80% references go to that region. The local caches of clients filtered out reads issued by the hotcold benchmark programs, so fewer requests were sent to the server.

Table 2 and Table 3 show the number of misses at each client during the first run and the second run of the synthetic benchmarks. The number of misses is small compared to the number of hits. The total number of misses is less than 1.5 % of that of hits in all the benchmarks. The small number of misses in the synthetic benchmarks confirms our expectation that page discard notifications successfully prevent the server from forwarding the requests to clients that no longer cache the requested data blocks. The number of misses of NFS-cc depends on page replacement activities as well as file access patterns. The hotcold benchmarks have less page replacements than the sequential and random benchmarks; therefore the number of misses is smaller in hotcold benchmarks. In the first run of hotcold-local and hot-cold-benchmark, all client caches are empty at the beginning and the hot regions are always in local caches. There are few page replacement activities at the client side, so it is possible for the hotcold benchmarks to achieve 0 misses as shown in Table 3. Because in the sequential benchmark files are accessed sequentially and the data blocks are only accessed once by each client, the data blocks will hardly be read by other concurrent clients once they have been discarded. Therefore, the number of misses in the sequential benchmark is also much less than that of the random benchmark although they both have active page replacements.

TABLE 3
NUMBER OF MISSES AT EACH CLIENT IN THE SECOND RUN

Benchmark	1	2	3	4	5	6	7	8	9	10	11	12
Sequential	0	215	226	0	289	396	1	164	0	47	787	0
Random	429	331	367	271	323	373	309	313	363	347	445	428
Hotcold-local	30	20	25	23	39	35	15	33	25	35	34	20
Hotcold-global	46	40	38	39	34	52	14	37	41	33	33	27

3) Page Discard Notification Effects

To evaluate how significant page discard notifications are in reducing the cooperative caching miss rates, we conducted an experiment by intentionally turning off the page discard notification in NFS-cc code. Table 4 and 5 show the aggregate miss rates before and after we turned off the page discard notification. Page discard notifications keep the cooperative caching miss rates to be less than 1.5% in all the benchmarks. Without page discard notifications the cooperative caching miss rates of NFS-cc are unacceptably high. High miss rates counteract the benefit of NFS-cc because each miss increases the latency to read a data block.

TABLE 4
COOPERATIVE CACHING MISS RATES WITH PAGE DISCARD NOTIFICATIONS

Benchmark	Sequential	Random	Hotcold-local	Hotcold-global
First Run	0	0.92%	0	0
Second Run	0.88%	1.47%	0.70%	0.97%

TABLE 5
COOPERATIVE CACHING MISS RATES WITHOUT PAGE DISCARD NOTIFICATIONS

Benchmark	Sequential	Random	Hotcold-local	Hotcold-global
First Run	0	24.7%	0	0
Second Run	43.8%	45.7%	10.4%	17.4%

V. RELATED WORK

Previous work on cooperative caching focused on using the remote memory of idle clients to avoid expensive disk accesses on the file server; the basic assumption was that slow disk I/O was the bottleneck. In the seminal paper on cooperative caching [1], Dahlin et al. used trace-driven simulation to evaluate several page replacement algorithms for utilizing remote memory, the best of which is called N-chance forwarding. When the N-chance forwarding algorithm is about to replace a page, it checks whether it is the last copy of a file system data block (a *singlet*) in the cluster. If it is, N-chance forwarding forwards the singlet to a randomly-selected peer, otherwise it discards it. Each singlet has a recirculation count N and the singlet is discarded after it has been forwarded N times.

The N-Chance forwarding has the advantage that it does not need global information, but the performance cost of random forwarding mistakes is also high. In N-chance, a singlet may be forwarded multiple times to sites that do not have any free memory. This causes unnecessary overhead because the sites receive the singlet and re-forward it to

another site. Therefore, the random selection of N-Chance forwarding works well only when most sites have free memory. In the Sprite file system workloads [13] used for N-Chance forwarding trace-driven simulations, 60%-90% of the sites had free memory available during each 10-minute period. So it is not a surprise that N-Chance can achieve nearly optimal performance as pointed out in [18]. But whether the cooperative caching scheme of N-Chance forwarding would perform well on workloads with different characteristics remains an open question. The goal of N-Chance forwarding is to avoid expensive disk access at the server, so the server doesn't forward any requests to clients as long as the server caches the requested data. Under concurrent sharing workloads, where the performance bottleneck becomes the server's bandwidth, N-Chance forwarding doesn't show any performance advantage. However, NFS-cc adopts a different strategy to forward requests as described in Section III-A2; the server forwards a request to a client optimistically as long as the cache directory indicates that the client caches the requested data block. This strategy helps to reduce the contention for the server's bandwidth and thus improves the overall performance under concurrent sharing.

Sarkar and Hartman proposed hint-based cooperative caching [2] to manage cluster-wide file system caches. Hint-based cooperative caching seeks to improve N-chance forwarding by using local information about the state of the system, or *hints*, instead of global state, or *facts* to make page forwarding decisions. Hints don't need to be 100% accurate, and consequently they are less expensive to maintain than facts. Hints have been shown to improve system performance as long as they are mostly accurate [2]. Like N-Chance forwarding, Sarkar and Hartman also used the simulations of the Sprite file system traces to evaluate their hint-base cooperative caching scheme and the hints turned out to be more than 99% accurate. To verify their simulation results, Sarkar and Hartman implemented a prototype and measured its performance for one week. Despite this experimental evaluation, the workload in for their tests was similar to the Sprite workload. Therefore, hint-based cooperative caching, as well as N-Chance forwarding, lacks any analysis of how their algorithm would perform under different workloads, for example concurrent reading or writing. Under concurrent workloads, the hint-based cooperative caching may have some problems because of the way that it tracks the location

of data blocks. Hint-based cooperative caching [3] keeps track of only one copy of each block, the *master copy* (the first copy to be cached by any client). Consider the following example where multiple sites copy the same large file concurrently. The site that starts copying first will always obtain the master copies of the file's data blocks. Since a block location hint contains only the location of the block's master copy and the locations of other copies are not recorded, the subsequent requests by other sites will all be sent to overwhelm the site that first initiated the copying. However, this situation will not happen in NFS-cc because NFS-cc forwards requests in a round robin fashion as described in Section III-C so that all clients caching the requested data block and the server have a chance to serve the requests. The results of the synthetic benchmarks in Section IV-D2 show that NFS-cc distributes the requests to all clients and the server; and there is no single client overwhelmed by requests.

XFS [3], a serverless network file system, is the first network file system that integrated cooperative caching. There is no central server in xFS and any machine in the system can store, cache or control any block of data. The motto of xFS is "anything anywhere". However, the xFS paper [3] didn't examine the use of cooperative caching in the system because all results were studied using simulation as in [1]. The xFS work focused on the issues raised by integrating cooperative caching with the rest of the serverless system. Since xFS also uses N-Chance forwarding for cooperative caching, it may share the same weaknesses of the N-Chance forwarding discussed above.

The consistency model of xFS and NFS-cc are different: xFS keeps a strict consistency model while NFS-cc has a weak one. XFS utilizes a token-based cache consistency scheme like Sprite [8] and Andrew [8]. The metadata managers of xFS keep track of the file system data blocks being cached by each client so that they can not only take appropriate action to guarantee consistency but also use the same state to support cooperative caching. However, for the stateless NFS server, the precise information about what clients are caching is not available. In order to support cooperative caching, our NFS-cc has to set up a cache directory and maintains the cache directory using various techniques, such as page discard notification. Thus the designs of the cooperative caching scheme in xFS and NFS-cc are fundamentally different though they both support cooperative caching.

Lustre [24] is a large scale cluster file system that is currently being tested on three of the largest clusters, each with more than 1,000 nodes. The designers of Lustre found a very common bottleneck in their system is that when all clients are read sharing the same files, the servers will be completely overwhelmed, based purely on the raw bandwidth that they can provide. For example, a typical scenario is that all clients are loading a large application stored at the file servers at the same time. They proposed *collaborative*

caching [16], which is just another term for cooperative caching, to remove the bottleneck. The Google File System [17] is a distributed cluster file system for large distributed data-intensive applications. It was noticed that some servers became hot spots if many clients were accessing the same files. Google fixed this problem by replicating the files to more servers but a long-term solution they recommended is to utilize cooperative caching. Cooperative caching has not been implemented in Lustre nor the Google File System so we cannot compare NFS-cc with them. However, more cluster file system designers have noticed that cooperative caching is a promising solution approach for concurrent read sharing.

There are also several works other than cooperative caching that are related to NFS-cc, including GMS [4] and Oracle's Cache Fusion [5]. GMS is more general than cooperative caching because it is a distributed and coordinated memory management. GMS differs from cooperative caching in that it doesn't provide a consistency mechanism; consistency is responsibility of the high-level software that creates sharing in the first place. In contrast to GMS, cooperative caching is integrated into the network file systems and the network file systems create sharing. Therefore, cooperative caching must preserve the coherence semantics provided by the network file systems. Oracle's Cache Fusion is designed for shared-disk clustered database. It is not a scheme for file system caching, but it does share the same idea with NFS-cc. Like NFS-cc, Cache Fusion allows data blocks to be read directly from caches of other sites, which eliminates the need for extra disk I/Os. However, because Cache Fusion is designed for shared-disk architecture and NFS-cc is based on NFS – a typical network file system, they have completely different design and implementation.

Peer-to-Peer file sharing systems [19, 20, 21, 22] share the same basic idea as NFS-cc, which exploits the bandwidth of clients to alleviate the contention for the server's bandwidth. But P2P file sharing and NFS-cc are developed in different contexts. NFS-cc is developed to support file sharing in a single administrative domain, where clients are forced to be cooperative. However, P2P file sharing is designed for autonomous computers scattered geographically. In this context, it is difficult to make the users cooperate effectively. We make an assumption in NFS-cc that clients must be mutually trusted but this assumption cannot be extended to P2P systems. To ensure data integrity, the clients in P2P systems must verify the hash value of each data block that they receive. In order to provide reasonably consistent download rates for everyone, P2P systems must also penalize uncooperative clients that are downloading without uploading. An algorithm used in BitTorrent file distribution system [19] is a variant of tit-for-tat. Specifically, the idea behind the approach is that the clients will refuse to upload files to peers if they cannot download files from them. P2P file sharing systems and NFS-cc are an example that a similar idea in different contexts may result in completely different

systems.

VI. CONCLUSIONS AND FUTURE WORK

NFS-cc demonstrates the effectiveness of using cooperative caching to improve the aggregate read throughput of an existing network file system under different concurrent sharing workloads. Unlike other network file systems of which the maximum read throughput tends to remain fixed, the maximum read throughput of NFS-cc under concurrent read-sharing increases with the number of clients as shown in Figure 5. Furthermore, the idea of using cooperative caching, as exploited in NFS-cc, is not limited to NFS-cc but can also be applied to parallel file systems or cluster file systems [16, 17] to improve read performance. Future design of parallel file systems and cluster file systems may adapt the techniques used in NFS-cc to support cooperative caching as part of future system improvements.

One of the future improvements for NFS-cc is to use hints, as proposed in [2]. In the current design of NFS-cc, every READ request initiated by any client must go through the server. Hints allow clients to send requests directly to peers bypassing the server. We expect hints will reduce the total number of messages sent to the server and improve the scalability of the file system. Research into the design of a hinting scheme and integration of it into NFS-cc requires further study.

REFERENCES

- [1] M. Dahlin, R. Wang, T. Anderson, and D. Patterson. Cooperative Caching: Using Remote Client Memory to Improve File System Performance. In *Proceedings of the First Symp. on Operating Systems Design and Implementation*, pages 267--280, November 1994.
- [2] Prasenjit Sarkar, John H. Hartman. Hint-based Cooperative Caching. *ACM Transactions on Computer Systems (TOCS)*, Volume 18 Issue 4, November 2000.
- [3] Thomas E. Anderson, Michael D. Dahlin, Jeanna M. Neefe, David A. Patterson, Drew S. Roselli, and Randolph Y. Wang. Serverless network file systems. In *Proceedings of the 15th Symposium on Operating Systems Principles*, pages 109--126, Copper Mountain Resort, Colorado, December 1995.
- [4] M. J. Feeley, W. E. Morgan, and etc. Implementing Global Memory Management in a Workstation Cluster. In *Proceedings of the 15th Symposium on Operating Systems Principles*, pages 201--212, December 1995.
- [5] T. Lahiri, Vinay Srihari, and etc. Cache Fusion: Extending Shared-Disk Clusters with Shared Caches. In *Proceedings of the 27th VLDB Conference*, Roma, Italy 2001.
- [6] F. Schmuck and R. Haskin. GPFS: A Shared-Disk File System for Large Computing Clusters. In *Proceedings of the First Conference on File and Storage Technologies (FAST)*, Jan. 2002.
- [7] Brian Pawlowski, Chet Juszczak, Peter Staubach, Carl Smith, Diane Lebel, and David Hitz. NFS version 3 design and implementation. In *Proceedings of the Summer USENIX Conference*, pages 137--152, June 1994.
- [8] M. Nelson, B. Welch, and J. Ousterhout. Caching in the Sprite Network File System. *ACM Transactions on Computer Systems*, 6(1):134--154, February 1988.
- [9] J. Howard, M. Kazar, S. Menees, D. Nichols, M. Satyanarayanan, R. Sidebotham, and M. West. Scale and Performance in a Distributed File System. *ACM Transactions on Computer Systems*, 6(1):51--81, February 1988.
- [10] Ibrahim F. Haddad. PVFS: A Parallel Virtual File System for Linux Clusters. *Linux Journal*, Volume 2000, Issue 80, November 2000.
- [11] W. B. Ligon III and R. B. Ross. Implementation and Performance of a Parallel File System for High Performance Distributed Applications. In *Proceedings of the Fifth IEEE International Symposium on High Performance Distributed Computing*, August, 1996.
- [12] Len Wisniewski and Brad Smisloff and Nils Nieuwejaar. Sun MPI I/O: Efficient I/O for Parallel Applications. In *Proceedings of SC99: High Performance Networking and Computing*, Portland, OR, 1999.
- [13] Mary G. Baker and John H. Hartman and Michael D. Kupfer and Ken W. Shirriff and John K. Ousterhout. Measurements of a distributed file system. In *Proceedings of the 13th Symposium on Operating Systems Principles*, page 199-212, October 1991.
- [14] 10 Gigabit Ethernet Alliance, <http://www.10gea.org/>.
- [15] InfiniBand® Trade Association, <http://www.infinibandta.org/>.
- [16] Philip Schwan, Luster: Building a File System for 1,000-node Clusters, In *Proceedings of the 2003 Linux Symposium*, July, 2003, Ottawa, Canada.
- [17] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung, The Google File System, In *Proceedings of the nineteenth ACM symposium on Operating systems principles*, pages 29-43, 2003.
- [18] M. J. Feeley, Global Memory Management for Workstation Networks, Ph.D. dissertation, Dept. of Computer Science and Engineering, University of Washington.
- [19] Bram Cohen, Incentives Build Robustness in BitTorrent, <http://bitconjurer.org/BitTorrent/bittorrentecon.pdf>.
- [20] Gnutella website, <http://www.gnutella.com>.
- [21] Napster website, <http://www.napster.com>.
- [22] KaZaA website, <http://www.kazaa.com>.

Ying Xu received his B.E. degree in computer science from Tianjin University, Tianjin, P.R. China, in 1998. He is currently a Ph.D. candidate of computer science at University of California, Riverside. His research interests include distributed system, cluster computing, operating system and mobile code.

Brett D. Fleisch is currently serving as Program Director at the National Science Foundation in Arlington, Virginia. He is also Associate Professor of Computer Science and Engineering at the University of California, Riverside. He received the Ph.D. degree in Computer Science from UCLA in July 1989. He received the B.A. degree in Computer Science at the University of Rochester, the M.S. degree in Computer Science at Columbia University in 1981 and 1983, respectively. He joined the UCLA computer science department in September 1983 where he served as Research Assistant in the Locus group. His dissertation was entitled "*Distributed Shared Memory in a Loosely Coupled Environment*".

In the past, Fleisch has served as a consultant and summer employee at Xerox Corporation's, Webster Research Center, IBM Corporation's Thomas J. Watson Research Center, the Educational Testing Service in Princeton, New Jersey, The College Board (West Coast offices) and the State of California, Department of Motor Vehicles. In addition, he has also worked at Hewlett-Packard Laboratories, Carnegie-Mellon University, Locus Computing Corporation, and has served as a teaching assistant in the UCLA Computer Science Department. In January 2003 he recently spent a six month sabbatical period at the University of Illinois, Chicago working in conjunction with faculty there while cooperating with researchers at Argonne National Labs in Argonne, Illinois. His research interests are in operating systems, mobile code systems, security, DSM, fault-tolerance, reliability and availability. Dr. Fleisch has been funded by the National Science Foundation, Digital Equipment Corporation, International Business Machines Corporation (IBM), Hewlett-Packard Laboratories, Computer Marketplace Incorporated, Sun Microsystems, the Office of Naval Research and the UC Micro program. Dr. Fleisch is a member of the ACM, IEEE Computer Society, and USENIX.