Cooperative Cache Management in S2FS

Jianyong Wang, Mingfa Zhu, and Zhiwei Xu
National Research Center for Intelligent Computing Systems
Institute of Computing Technology
Chinese Academy of Sciences
P.O. Box 2704, Beijing 100080, P.R. China

Abstract *S2FS, a descendant of NCIC's COSMOS file system prototype, is a single-image cluster file system, and good scalability is its main design goal. Reducing client-server communication and the number of disk accesses is critical for I/O scalability. Cooperative caching reduces the number of network hops and disk accesses considerably, however, the network must be used to maintain cache coherence. This paper introduces the design of dual-granularity cooperative cache coherence, which is a new method to further reduce the server-end load and network communication. Experiments show that the dual-granularity approach can get a greater improvement compared to the traditional single-granularity one.*

Keywords: S2FS, dual-granularity cache coherence, enhanced N-Chance algorithm

1 Introduction

S2FS is a descendant of COSMOS—a prototype single-image cluster file system designed for NCIC's dawning superserver. One of its design goals is good scalability. Reducing client-server communication and the number of disk accesses is critical for I/O scalability. Cooperative caching reduces the number of network hops and disk accesses considerably, however, the network must be used to maintain cache coherence.

File systems such as NFS and Sprite validate cached data at file open time, and maintain cache coherence at file-granularity level. Coda[1] file system uses a larger granularity(i. e. volume) callbacks as a way to reduce the client-server communication necessary to maintain file cache coherence in a weakly connected environment. xFS and NCIC's COSMOS file system both provide a 'perfect' cache consistency mechanism, and they maintain cache coherence on a block-by-block basis using a token-based scheme. Coarse-grained cache management used in NFS and Sprite will suffer 'false sharing' and their performance will be drastically decreased in a frequently concurrent write-sharing environment. To solve the 'false sharing' problem, Sprite's approach is cache disabling when write-sharing occurs and NFS just simply doesn't provide strong consistency guarantee. However, maintaining cache coherence at fine-grained level in file systems such as xFS will be a burden on server and will incur some unnecessary overhead by contrast with coarse-grained scheme.

The initial motivation of dual-granularity cache management in S2FS is to overcome shortcomings existing in both coarse-grained and fine-grained schemes. When a client opens a file without conflicting with other clients(i. e. no concurrent write-sharing occurs), the server gives this client a file callback, and promises to notify the client if another client opens this file in a conflicting way. When such a thing occurs, the server sends an invalidation message to the callback-owning client, and recalls the callback. If a client receives a callback break for a file, from then on it will manage its file cache at block-granularity level using token-based scheme, so occurrence of false-sharing incurred by write-sharing will be less frequent.

In the course of implementing the new protocol, we find that the dual-granularity design has several advantages over the single-granularity one. First, it reduces the server load to maintain cache coherence; second, in some cases it can reduce client-server communication, and more important, it provides some helpful hints, S2FS can use these hints to further mitigate the burden on the server and the network overhead between client and server, especially these hints can be used to implement high-performance cooperative caching algorithm. Prior to describing these improvements in detail, we'll first introduce S2FS file system briefly.

2 S2FS File System

S2FS, an acronym for a Scalable Single-image File System, is a global file system. Single-system image, good scalability and high I/O performance are its main design goals. In order to maintain a strict single-system image, it provides location transparency and strong UNIX file-sharing semantics. Being lack of AIX operating system's source code, we can still add S2FS into AIX seamlessly at the Vnode/VFS interface so that S2FS maintains ABI/API compliance with UNIX file system. In addition, S2FS implements cooperative caching, parallel coordinated management for metadata and parallel disk I/O to improve I/O performance and scalability.

S2FS is composed of four pieces: the kernel-related layer and three user level daemons which we call client, manager and storager, respectively. S2FS takes advantage of cooperative cache memory on multiple clients to gain a greater cache capacity available and improve cache hit ratio, and the manager is responsible for storing and maintaining system metadata(including file inodes and superblock), and it also records the data location in the clients' caches so as to preserve cooperative cache coherence. The storage server implements network disk stripping and software RAID 1 storage model, and the implementation of its low-level storage is based on reliable JFS and makes full use of asynchronous I/O functions, which improves I/O bandwidth and storage reliability.

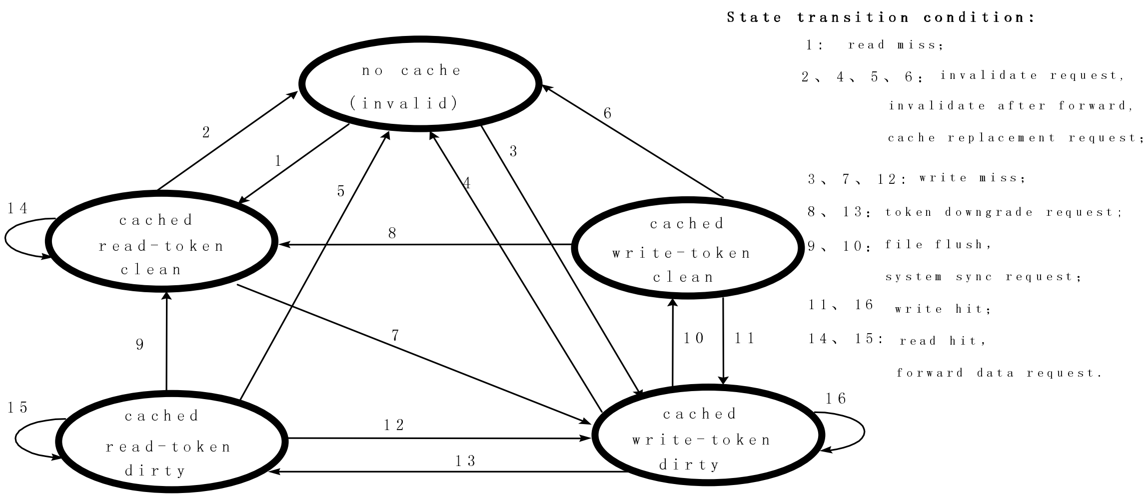


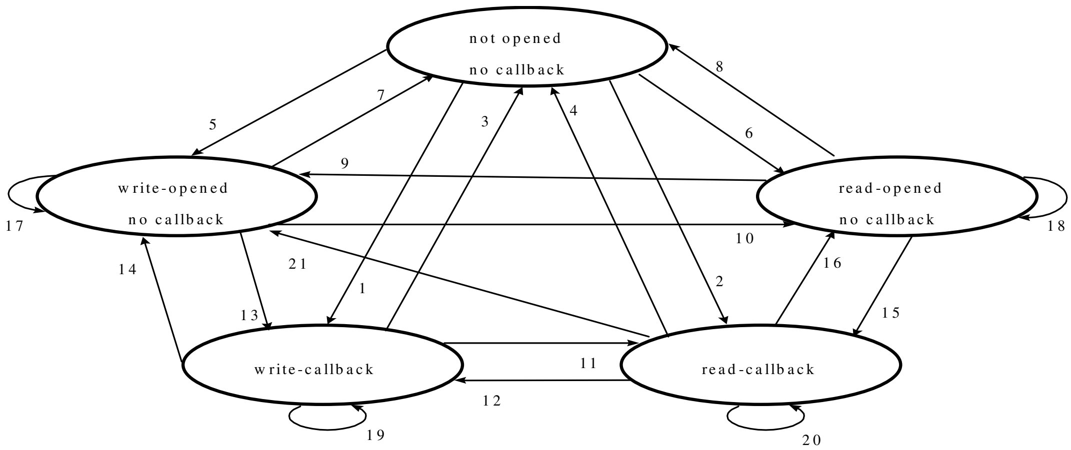
Figure 1: client state transitions at block granularity level

3 Cooperative Cache Coherence in S2FS

3. 1 Dual-granularity Cache Coherence Protocol

Just as section 2 has mentioned, S2FS implements cooperative file cache management. If an I/O request can't be satisfied by local cache, maybe the requested data can be located from a remote client's cache. The previous version of S2FS(i.e. COSMOS) makes use of read/write token to guarantee cache coherence at the block-granularity (4k bytes in S2FS) level. In this approach a file block is always cacheable on at least one client. In order to access a file block, a client must obtain a read-only or read-write token from the manager. Once a client has obtained the appropriate token for a file block, it is free to cache this block and access it. The manager guarantees that at any given time there is either a single write token or any number of read-only tokens outstanding for each file block. Concurrent write-sharing takes the form of conflicting token requests. When this occurs, the manager recalls one or more of the existing tokens in order to grant the new request. When a write token is recalled from a client, the client must forward the corresponding dirty block to the new token owner. If the new token granted is a write token, all the clients that give up tokens for that file block must also invalidate the cache block. Figure 1 shows the client state transition diagram at the block-granularity level.

Here we give above-mentioned 'concurrent write-sharing' a definition: when a file is opened on several nodes and at least one of them has the file open for writing, the file is said to undergo concurrent write-sharing. Based on the observations that clients seldom concurrently write-share data[2] and maintaining cache coherence at fine-grained level such as block-granularity could generate much server load, S2FS implements dual-granularity cache coherence protocol. when a client opens a file without concurrent write-sharing, the manager gives it a file callback, and promises to notify the client if another client opens this file in a conflicting way. When this occurs, the manager sends an invalidation message to break the callback. If a client receives a callback break for a file, it knows that another client is now write-sharing this file with it, and from then onwards it'll maintain cache coherence at block-granularity level using token-based scheme.



State transition condition(here "write" means file access mode is O_WRONLY or O_RDWR, and "read" means file access mode is O_RDONLY):
1: file create, write open;2、6: read open;3、7、10、11:write close;4、8: read close;
5、9、12、21: write open;13、15: another client write close the same file;
14: another client write (or read) open the same file;16: another client write open the same file;
17、19: read open,write open,read close,write close; 18、20:read open,read close;

Figure 2: client state transitions at file granularity level

S2FS makes a distinction between read-only callback and read-write callback. When a client creates a new file or opens a file for update(for example, when file access mode is O_WRONLY or O_RDWR) without concurrent write-sharing, manager will give the client a write callback, otherwise, if a client opens a file with O_RDONLY access mode and no write-sharing for this file occurs, the manager gives it a read-only callback. Figure 2 depicts the five file callback states and their transitions. That a client opens a file but has no callback for that file means concurrent write-sharing is undergoing. If a client owns the write-callback for a file, it can infer that it is the only client that has opened the file.

Along with callback state, a client records some other information in a callback structure, including file index number, number of clients that have opened the file, file length, file's on-disk position, number of read-open times and number of write-open times, etc. Each file's callback structure can be located using a hash function. S2FS uses these information as hints to further mitigate the burden on the manager and the network overhead. We'll illustrate this point in section 3. 2 and 3.3.

In S2FS, the dual-granularity protocol only applies to regular files. Because directory lookup may cause the directory file to be read without first opening it. In addition, when a client closes a file, it doesn't evict the file's cache blocks because maybe it'll be opened soon again. To guarantee cache coherence, the client must flush all dirty cache data of the being closed file to storager before the file close operation has been finished, so that when another client opens the same file, it can read the most up-to-date data from storager, and if a client updates a file block cached on another client who has closed this file, the manager must send an invalidation message to the file-closed client, then the file-closed client releases the updated cache block.

3. 2 I/O Operations Under Dual-granularity Protocol

Intuitively, if no concurrent write-sharing occurs, maintaining cache coherence at a file-by-file basis will cause less overhead than at the block-granularity level. For example, if a client owns the write callback for a file, when it wants to write a block of this file, it just judges whether the requested data block is in local cache and doesn't need to care about the block's token state. If a previous read operation has already fetched this block into its local cache, then it can directly update it and needn't first to contact the manager for obtaining the block's write token. In this case, one network hop is saved. Below we'll describe some I/O operations' re-implementation to illustrate that S2FS can get some useful hints from callback state and further reduce communication overhead between daemons. Here we just consider file read and file flush operations, some other I/O requests such as file write, cache replacement, ftruncate, fclear and fsync can also make use of callback information to reduce system overhead.

If a read miss occurs, the client can judge if it's the only client that opens this file using callback structure, if so, it directly sends a read file request to storager. After it receives the requested data, it then sends an acknowledgement to manager and returns the result to S2FS's kernel layer, until now the whole read operation has been finished. The client can figure out which storager stores the requested data from on-disk position information recorded in callback structure. It can also judge if the read data exceeds the file length, if so, there is no need to send read request to storager and just returns zero. Let's see what needs to do under single-granularity(here it's referred to as 4Kbytes) protocol. When the client finds that the read request can't be satisfied by local cache, it sends the request to manager, manager first looks up cache state table, but finds that the request can't be satisfied by cooperative cache either, then it forwards this request to the corresponding storager, after storager finishes the read operation, it'll send the result to the requesting client. Manager also needs to tell the requesting client from which storager it'll receive the data. After the client receives the result from storager it must acknowledge manager. Upon receiving the acknowledgement from client, manager updates its cache state table to record this event. From the above analysis, we can see that two network hops are saved under dual-granularity protocol.

If a client owns the file's write callback when it receives a file flush request, it knows that all of this file's dirty data can only reside in its local cache, it'll flush the dirty blocks to storager directly, there is no need to contact manager. But under single-granularity protocol, the client must first send file flush request to manager, and the clients caching the flushed file's dirty blocks are figured out by manager, then manager forwards file flush requests to these clients, and the final file flush operations will be finished by these clients.

3.3 Enhanced N-Chance Algorithm

3.3.1 N-Chance Algorithm

The Pioneering Paper on cooperative caching by Dahlin et al.[3] described several schemes for implementing cooperative caching, and settled on one called N-Chance as providing the best performance with the lowest overhead. The xFS file system uses a refined version of N-Chance algorithm.

The N-Chance algorithm dynamically partitions each client's cache into blocks needed by local applications and those stored in the cooperative cache. Managers are responsible for maintaining consistency and block location information. The replacement policy in N-Chance uses a combination of local LRU information and duplicate avoidance to decide the best block to replace. Clients always replace the oldest block on their LRU lists. Whether the client forwards or discards the oldest block depends on the number of copies of the block in the client caches. Blocks with more than one copy are discarded, while blocks with only one copy, or *singlets*, are forwarded to another client chosen at random. If a client does not know whether or not a block is a *singlet*, it asks the manager.

3.3.2 Enhancing N-Chance Algorithm under Dual-granularity Protocol

N-Chance algorithm has the clients cooperate to preferentially cache *singlets*, and this does reduce the number of copies of the same cache block and make the cooperative cache space accommodate more valid data blocks, but it still suffers from several drawbacks. First, when a cache replacement decision is made, and if the displaced block is not a *singlet* forwarded from another client, the manager must be contacted so that the local client can judge if the displaced block is a *singlet*, thus this needs two network hops(i.e. one request and one acknowledgement) and increases network overhead and the replacement response time; Second, when a client knows that the displaced block is a *singlet*, it'll always forward it to another client, in fact, some blocks that are selected to be replaced will not be referenced in the near future or even never again, so it doesn't make sense to forward such *singlets*; Finally, the client forwards the data to a random peer, and does not attempt to make use of the peers' file access patterns, for example, if the block is forwarded to a client that will reference it in the near future, then the local cache hit rate will be improved.

Our dual-granularity protocol design can give some implications on the cooperative caching, we can overcome the above-mentioned N-Chance algorithm's shortcomings to some extent. Let's first look at the possible status of a cache block designated to be replaced. Here we assume client

X decides to replace its *i*th cached block, b_{xi}, and b_{xi} belongs to file F, then b_{xi}'s status may be :

A: *b_{xi} is one of the singlet forwarded from another client;*

B: *There's no client using file F (i.e. no client has opened F);*

C: *F has been opened by at least one client:*

C.1: *Only client X has opened F;*

C.2: *There is at least one remote client that has opened F;*

Under dual-granularity protocol, we could judge b_{xi}'s status using the file-level callback state information. When b_{xi}'s status is A, then there's no distinction between the N-Chance algorithm and our enhanced N-Chance algorithm, namely b_{xi}'s recirculation count is decreased and it is forwarded again to another client at random unless the count is now zero. Status B means that block b_{xi} will not be referenced in the near future or even never again, so in our algorithm, b_{xi} is directly discarded , but N-Chance forwarding will first contact manager, and if manager tells client X that b_{xi} is a *singlet*, then b_{xi} is forwarded to a random peer, so some unnecessary overhead has been generated. The N-Chance algorithm and our enhanced one will do the same thing when b_{xi}'s status is C.1. They both need to contact the manager so as to know whether b_{xi} is a *singlet*, if not , then just discard it, otherwise forward it to a random client. There are some differences between N-Chance algorithm and the enhanced one when b_{xi} is in status C.2. The enhanced N-Chance algorithm will first contact manager also, but the acknowledgement message from the manager contains information about which client (for example, client Y) is using file F (i.e. Y has opened F), on receiving the acknowledgement, client X will forward b_{xi} to client Y instead of a random peer, because Y is using F, so the possibility that client Y will reference b_{xi} in the near future is high, so client Y's local cache hit rate will be improved. All in all, a client could exploit the callback state information and judge the status of the displaced block, then make more desirable decisions to reduce the network overhead and improve the I/O response time.

4 Experimental Results

S2FS is a descendant of COSMOS prototype, it differs from the latter in cache coherence protocol. S2FS implements the dual-granularity protocol based on COSMOS's block-granularity cache coherence protocol, but currently both S2FS and COSMOS haven't implemented N-Chance or enhanced N-Chance algorithm, instead, they use the simple Greedy-Forwarding algorithm[3]. As of our writing this paper, we have done some simple experiments and have got some raw materials. Because S2FS is based on AIX system , and at present there are only 4 PowerPC workstations available for our test in our laboratory, for our test-bed, we use a total of these 4 machines:1 PowerPC running AIX 4.2 operating system and 3 others running AIX 4.1 system. Each of them has 64MB of physical memory and 67Mhz of BUS clock speed. In addition, the AIX 4.2 machine's MPU clock speed is 100Mhz and each of the 3 others is 133Mhz.

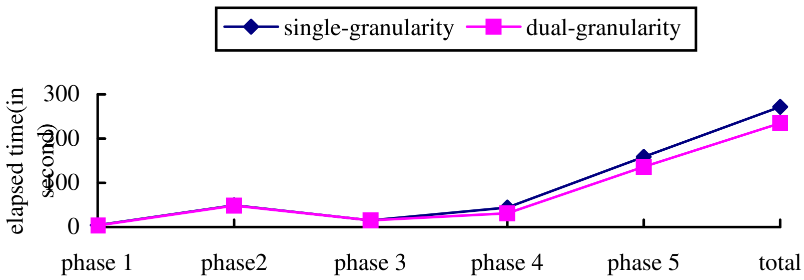


figure 3: performance comparison for the two protocols when running 2 clients

Table 1: number of messages received by each daemon

Number of Messages / Protocol type	Number of messages received by client	Number of messages received by manager	Number of messages received by storager	Total number
COSMOS: single-granularity	5168	3561	10887	19616
S2FS: dual-granularity	2	1931	10685	12618

Our first measurement traced the network communication information. In this experiment we started a client, a manager and a storager daemon on one of the 4 machines. We ran Andrew benchmark on COSMOS prototype and S2FS in turn, and recorded the number of messages that each daemon received during each run respectively. This benchmark was designed as a simple yardstick for comparing application-level performance for common tasks such as creating directories, copying files, recursive directory stats, scanning files and compiling files. Because only 1 client was running in this experiment, so no concurrent write-sharing occurred. Table 1 shows the result about network communication. There is a significant difference between the two protocols. For example, the COSMOS client received 5168 messages, but S2FS client received only 2 messages, almost all the messages COSMOS client received are unnecessary in the view of dual-granularity protocol. The total number of messages under dual-granularity protocol is cut down by 35.7% compared to the single-granularity one.

Figure 3 shows the average time for one client to complete the Andrew benchmark under the two different cache coherence protocols. In this experiment we started 2 client daemons, 4 storager daemons and 3 manager daemons and there’s a little write sharing when we concurrently run two sets of Andrew benchmark. From figure 3 we can see that the dual-granularity approach can get a considerable improvement — by 14% in terms of total elapsed time. So the results of Table 1 and Figure 3 illustrate that the dual-granularity cache coherence protocol is more scalable than the single-granularity protocol and can also improve I/O response time.

5 Summary

In this paper we have introduced dual-granularity cache coherence protocol as a way to reduce the server load and the communication between daemons necessary for cache coherence. The interesting thing is that we can get some hints from the file callback state, and these hints are helpful to cut down many unnecessary network hops and server load. Because S2FS guarantees user-level binary compatibility with unix file system, we don't need users or applications to provide additional information to run dual-granularity protocol, this makes the new protocol more favorable and attractive. We expect to incorporate the enhanced N-Chance algorithm into S2FS and do some experiments with S2FS in a larger scale cluster environment so as to gain more direct experiences with file callbacks in the coming months.

References

[1] Satyanarayanan M., et al. Coda: A Highly Available File System for a Distributed Workstation Environment. IEEE Transactions on Computers, Vol. 39, No. 4, April 1990. pp:447-459.

[2] Baker M. G., et al. Measurements of a Distributed File System. In Proceedings of the ACM 13th Symposium on Operating System Principles(Pacific Grove, Calif.), October 1991. pp:198-212.

[3] Dahlin M.D., et al. Cooperative Caching: Using Remote Client Memory to Improve File System Performance. In Proceedings of the 1st Symposium on Operating Systems Design and Implementation, November 1994. pp:276-280.