Hades: Locality-aware Proxy Caching for Distributed File Systems

Lamprini Konsta Stergios V. Anastasiadis
Department of Computer Science
University of Ioannina, GREECE

Technical Report DCS 2009-01
January 19, 2009

Abstract

In the present paper we explore storage management issues in proxy servers for distributed file systems. We organize the requested data at the disks of the proxy server using locality-aware approaches. Additionally, we introduce improvements in the mapping mechanism from remote to local data and consider cost-aware replacement methods. In a prototype implementation that we developed, we experimentally compare alternative distributed file systems as components of the proxy servers. Through extensive measurements, we demonstrate throughput improvements at the proxy server up to 80% in comparison to the disk-based cache of a commonly used distributed file system.

1. Introduction

Current trends in business and research collaboration encourage secure data sharing over wide-area networks with minimal intervention from the end user. Indeed, the requirement from the users to replicate datasets close to computation resources through explicit file transfers bears significant replica bookkeeping overhead to the user and only makes data usable after an entire file has been fully replicated locally. Therefore, it would be preferable to have a caching proxy to automatically replicate datasets and hide transfer delays during the repetitive use of data. The design of caching proxies for distributed filesystems is mainly attracting research interest in the direction of getting existing filesystems inter-operable with local file systems for persistent caching purposes. Here we point out the need for efficient storage management in caching proxies so that cache accesses from the local disk of the proxy have performance comparable to or better than direct disk accesses from the local file system.

Caching proxies behind web servers have already been broadly used for over a decade in content distribution networks. Originally, copies of web pages requested by users were replicated on proxy servers close to the web browsers over traditional local file systems. However, related experimentation in published literature demonstrated several performance deficiencies related to metadata management of multiple small files, frequent creation and deletion of files, excessive disk head movement from poor clustering of jointly used data or access overheads from multiple small writes. Subsequently, customized file systems emerged that complementarily addressed the above issues through special internal architectures and new access interfaces.

On the other hand, most distributed filesystems were originally designed for serving the storage needs of users within the same organization at a single geographical site. The assumed use of a local-area network limited client-side caching to main memory and made unnecessary the corresponding disk-based caching. One notable exception is the Andrew File System and its descendants that provide the capability to temporarily store data at the local file system of the client machine for improved scalability and availability in distributed environments [1].

However, Andrew assumes that the client machines from individual users are powerful enough to relieve centralized servers from computations. This is not the case when building caching proxies for data sharing among large numbers of clients within an organization. Andrew replicates remote data in chunks of a configurable fixed size. Initially, it creates a large number of individual local files at the client and subsequently uses each of them to store an individual chunk requested from the server. Andrew has been widely successful for over two decades in general file system use. However, in modern scientific and business environments it is common to have numerous small files or enormously large ones. Then, the approach of having a separate local file per chunk might not be the best possible in terms of data access or metadata management efficiency.

The dominant organization in recent published literature is to map each remote file to a local file in the caching proxy [4, 16]. On the contrary, we

claim that apart from offering a consistent view of the data as they appear at the origin server, the caching proxy should be free to manage its local data in whatever way serves its design objectives better. Thus, the main contributions of the present paper include the following:

- Efficient data clustering techniques for disk-based proxy caches.
- Metadata organization methods for faster mapping of remote data to the local cache.
- An approach for data replacement with consideration of the distance of the origin server.
- Experimental comparison of alternative distributed file systems as components of a proxy server.

We organize the present paper across six sections. In Section 2 we present basic design issues that emerge in proxy servers, while in Section 3 we describe the architecture and the implementation of Hades. In Section 4 we evaluate our prototype across a microbenchmark and an actual application. Finally, in Section 5 we compare our work with previous related research and in Section 6 we outline our conclusions and future work.

2. Background

Arguably, caching of file system data on persistent storage can significantly reduce the operation cost of distributed file systems over wide-area networks for the following reasons:

- The cost of purchasing and managing data storage over a time period is orders of magnitude less than the cost of leasing network bandwidth with comparable transfer capacity [19].
- Sharing physical storage resources across multiple clients additionally reduces the network bandwidth requirements.
- Bursty requirements from a single client can more easily be satisfied from multiple storage devices accessible from the local network rather than the remote origin server.

Nevertheless, intermediate layers in the path from the origin server to the client may have adverse effect to the perceived throughput and latency as a result of introduced performance bottlenecks and reduced parallelism in the data transfers (Figure 1). In the present section we examine several resource management issues that may arise when designing proxy servers for file systems and some design choices that we faced in our proposed architecture.

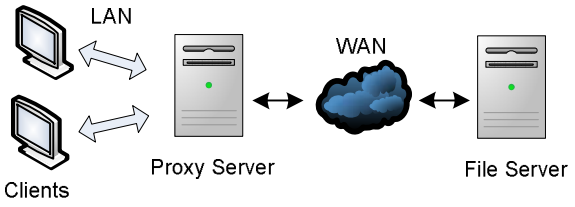

Figure 1. The basic architecture of a proxy server in a distributed file system.

2.1. Storage Allocation

The development of cost-effective systems requires the placement of cached data on hard disks rather than the main memory of the proxy server. Over time, system designers have developed several different techniques to effectively allocate the disk storage space of the cache.

Early systems copied entire files from the file server to the client. This approach was problematic because it incurred high transfer latency and large resource requirements at the client [14]. In later approaches, the designers adopted partial caching approaches. One possible solution manages the remote files in fixed-size chunks that it copies and stores onto corresponding individual local files at the client [14]. In more recent prototypes, the system dynamically replicates the directory and file naming structure from the origin server to the cache. It also transfers the file contents on demand in pages of configurable size [5, 16]. Locally, the system uses a typical file system or a raw disk partition to temporarily store the data of the cache.

A somewhat similar storage allocation problem showed up in web proxy servers. Storage locality concerns can be handled by grouping files and metadata into clusters stored on consecutive blocks of disk. The clustering is based on the temporal locality of the access requests. Additionally, the proxy server can treat large files specially and transfer them directly to the disk bypassing the memory cache [15]. In order to reduce management overheads for small files, the system may group the files by size and store them in a buddy organization. Thus, it eliminates file space fragmentation and reduces considerably the overhead for file creations and deletions. Aggregation of the written data in memory and subsequent appending to disk can pro-

vide additional write throughput improvement [7].

Additionally, it is possible to dynamically reorganize the layout of the data on the disk to match the received access pattern. In typical file systems, previous research has examined to exchange the data across different disk locations in order to minimize the distance of consecutive requests in previously collected statistics [8, 18]. In Hades, we investigate the placement of incoming data into consecutive locations grouped by the identifier of the remote file to which the data corresponds or the requesting user. We leave for future work the study of other grouping criteria such as the identifier of the origin server.

2.2. Consistency

Operation performance considerations of distributed file systems typically lead to weak consistency models that allow local data caching at the client without support for strict distributed cache coherence. This means that the system demonstrates undefined behavior with concurrent write sharing in the absence of locking. For example, in the close-to-open consistency used in NFSv3, the client blocks on the close call until all data is stably stored on the server. On a subsequent file open, the client uses an attribute check to validate the cached data and determine if it will keep it [12].

As an optimization, NFSv4 may delegate open/close and locking operations to the client to eliminate periodic cache consistency checks with the server. The delegation is associated with a lease that expires after a time period. Alternatively the server explicitly revokes the delegation with a preregistered callback [13]. Callbacks were earlier introduced in Andrew to avoid frequent cache validation checks of the client with the server [14]. In a different approach used by CIFS, the server may grant a temporary exclusive lock to a client that opens a file not currently accessed by another application. The server breaks the lock when another client attempts to open the file [2].

2.3. File replacement

Recent page replacement policies in local storage hierarchies are able to simultaneously take into consideration multiple access features such frequency and recency in order to maximize the hit ratio across different workloads. Similarly, in the context of web proxies, the most successful policies seamlessly combine recency with file size, popularity or fetching latency [3,6]. Although early work demonstrated limited data sharing across different clients [10], recent studies related to grid environments show significant benefits from caching when measuring the access latency rather than the hit ratio [11]. Consideration of file duplication across different file systems substantiates further the potential data sharing across different clients [1].

3. The *Hades* Architecture

The majority of the published literature on proxy caching architectures typically refers to web environments with predominantly read-only workloads of limited reliability and consistency demands. Additionally, web proxies support access granularity of entire files and have limited security constraints due to the public nature of the transferred data. Furthermore, the disk-based caching system that is most commonly used for file systems has mainly been developed to run directly on personal workstations and is not optimized to support concurrent requests from large numbers of users arriving from different client machines.

3.1. Goals

In the architecture that we propose, we identify three main directions for the development of efficient file system proxy servers:

1. Organize the requested data on the proxy cache in ways that improve the spatial storage locality.
2. Allow reuse of the data available on the proxy server across different clients using existing standard protocols.
3. Replace files that have not been used recently with consideration of their fetching latency from the origin server.

We transfer data from the server to the proxy in chunks of a configurable size similarly to the AFS architecture. Instead of storing each chunk as a separate file on the disk cache, we organize the chunks as contiguous segments of a large file in the proxy server, called *proxy file*. The proxy file has size that is only limited by the local file system. In our system, we preallocate the space for multiple proxy files according to the anticipated storage requirements of the clients and the available resources.

The chunks of the same remote file that we fetch with a single request from the origin server are stored consecutively at the same local file. We also store on the same local file the data chunks fetched

subsequently either from the same remote file, or from different remote files by the same user. Due to the spatial locality that we enforce at the proxy, we anticipate that data repetitively used by a user can be retrieved from the proxy cache with low access overhead. However, the actual performance seen by the end user also depends on the behavior of the other users concurrently utilizing the same proxy server.

The deployment of file system proxies is mainly motivated by the need to access data across wide-area networks. As a result, different files requested from the proxy incur fetch latencies that vary according to the actual location of the origin server. In our replacement policy, we keep track of the amount of time needed to fetch each chunk to the proxy. We aim to preserve locality and avoid fragmentation during replacement. Thus, we treat as a single unit, called *chunk run*, the group of chunks that are stored consecutively on the proxy and belong to the same remote file.

For each run, we keep track of the average fetch latency across its chunks. We categorize the chunk runs as *local* or *remote* depending on whether their average latency is lower from or exceeds a configurable threshold. At the next replacement operation, we pick as victim the chunk run that is earliest in the LRU list and has the lowest average latency. As a result, we first favor the local runs for replacement. If our search for a local run fails in a pass along the LRU list, then we pick for replacement the remote run that has been least recently used.

3.2. The Structure of OpenAFS

Here, we outline the basic on-disk and memory-based data structures of the OpenAFS system that we modified in our prototype implementation. One critical component of OpenAFS is the *file server* that runs at user level and exports local data to remote clients. The *cache manager* plays fundamental role at the client kernel, because it improves data transfer efficiency through local memory and on-disk caching. The client cache is implemented as a configurable number of disk files that are stored in the ext2/3 file system of the client machine. Each cache file has maximum size equal to the chunk used for the transfers from the server (typically 256KB). The cache files are created at the client in advance during the system initialization. They appear as regular files with names Vi, where the index i takes values between 0 and a maximum configurable value.

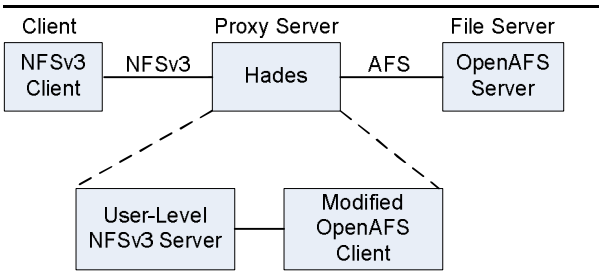

Figure 2. The Hades system combines a modified OpenAFS client with a user-level NFS server

Structures of type *fcache* associate each Vi file with a chunk of a remote file. The fields of fcache store metadata, such as the identifier of the remote file, the offset, the chunk size, and the inode of the local file. An array of fcache structures is stored persistently on disk. For improved indexing efficiency, a subset of the fcache structures is also maintained in memory as a collection of *dcache* structures. Periodic updates keep the fcache contents consistent with their memory counterparts.

Additionally, the client caches the metadata of the remote files in the form of *vcache* structures. Each request to the offset of a remote file can be mapped quickly to the corresponding offset of a local file at the client through the *afs_dchashTbl* hash table. The hash function translates the identifier and the offset of the remote file to a hash table position. A separate auxiliary hash table chains the additional entries required in the case of collisions. If the requested chunk is not available locally, a new index entry is allocated along with a free local file to store the data transferred from the remote server.

3.3. Implementation of Hades

We implemented the Hades proxy server based on a combination of a modified OpenAFS client and a regular NFS server as shown in Figure 2. Hades accesses the files exported by a remote OpenAFS server through an OpenAFS client that we modified appropriately for that purpose. Subsequently, Hades re-exports the accessible remote files through a regular NFS server. Finally, the client machines use a normal NFS client to access the remote data from the proxy server. In order to achieve our design goals we expanded the OpenAFS client along the following three directions:

1. We preallocate multiple large local files and do our own space management for each of them.

2. We expand the mapping structure of each local file to store multiple chunks that belong to different remote files.
3. We keep low the average access cost by replacing locally cached chunks according to their access recency and fetching latency.

Below, we explain in more detail our implementation along each of the above directions.

3.3.1. Cache Files. The size of each local file is only limited by the settings of the local file system at the proxy. We use a separate bitmap to manage the storage space of each file and we call *cacheblock* the respective unit of storage allocation. The default cacheblock size is 4KB. The size of chunks that we transfer between the proxy and the origin server is typically a multiple of the cacheblock size. When an access request arrives, we search for the requested number of consecutive cacheblocks starting from the local file that was used more recently. After the reservation at the bitmap, the fetched chunks will be stored at the corresponding local file.

3.3.2. Mapping. For each remote file that is cached at the proxy, we expanded the vcache structure to maintain pointers to the bitmap and the respective dcache structure of the local cache file that was last used. Other fields that we add to the structure include the starting chunk number and the size of the last request cached at the local file along with the local file offset where the request is stored.

In the dcache structure of each local file we use an array of pointers to fcache structures of remote files (Figure 3). Thus, we associate a local file with all the remote files that store chunks there. This is a departure from the original OpenAFS implementation, where each local file could only store one chunk of a single remote file and only needed one fcache pointer. Finally, in each fcache structure we maintain an array of chunk descriptors. Thus, we keep track of all the chunks of a remote file that have been stored in the respective local file. Each chunk descriptor includes fields about the number and size of the remote file chunk along with the starting and ending offset at the local file.

3.3.3. Allocation. When we receive a request for part of a remote file, we can use the above structures to map the identifier and the chunk of the remote file to the local file and the corresponding offset where it is stored. If the requested part is not locally cached, we reserve the needed number of chunks at the first local file (starting from the last used) that has enough consecutive space available.

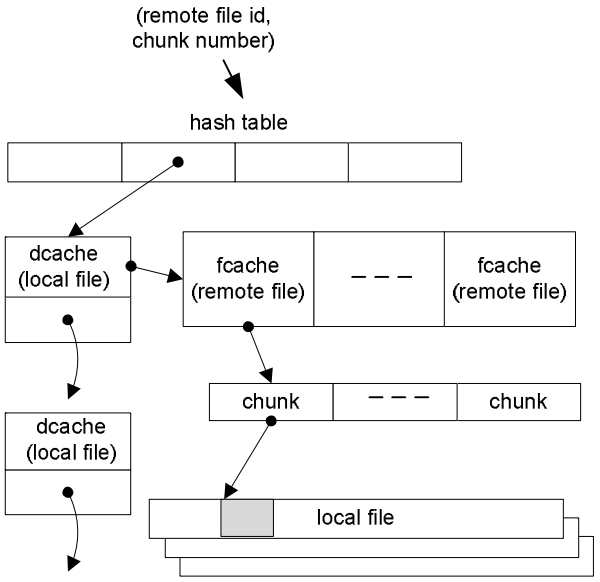

Figure 3. The main structure of the modified OpenAFS client in Hades.

We also update accordingly the dcache and fcache structures about the remote chunks that we cached. We fetch remote data in chunks and store them locally in cacheblocks. Assuming temporal locality in the different requests, we store in nearby disk locations the different chunks of a remote file in order to reduce disk seek overheads during subsequent retrievals.

For performance reasons, we also prefer to keep in the proxy cache clustered the data of different files requested by the same user. Thus, for each active user we maintain a pointer to the bitmap of the local file where the user cached data more recently. During a request, if the user attempts to cache data for first time, the bitmap pointer is null and we only cluster data based on the identifier of the remote file. Otherwise, we search for free space in the local file where the user last cached data. If we don't find sufficiently large space to fit the request there, we continue the search in the subsequent local file.

3.3.4. Hashing In the mechanism that we use to map remote chunks to local file offsets, we use a variation of the hash table used in the original OpenAFS. We should note that in our implementation the space of each local file is partitioned across the different chunks of multiple remote files. This was not the case in the original implementation of OpenAFS, where each local file could only store a unique remote chunk. Therefore, our system hashes differ-

ent chunks to the same entry of the afs_dchashTbl hash table. In order to address the need to map different chunks to the same local file, we remove the auxiliary afs_dcnextTbl table that was previously implementing an open addressing scheme for collisions. Instead, we implement hashing with chaining after attaching a linked list to each entry of the hash table.

When we search for a remote chunk, we hash the identifier of the remote file and the requested chunk number to a hash table entry. Then we search through the attached linked list for a dcache structure that contains pointer to the requested remote file. The fcache structure that corresponds to the remote file should also contain pointer to the requested chunk number. If the search succeeds, we found the chunk locally cached. Otherwise, we add a new node to the linked list of the hash table and make it point to the right dcache structure after the chunk is transferred from the remote server.

3.3.5. Replacement We expand the chunk LRU list maintained by OpenAFS with an extra field, called *fetch latency*, that we add to each list node. There, we store the measured latency to fetch the chunk from the remote server. During replacement, we prefer as victims the LRU chunks with fetch latency less than a preconfigured threshold. Although we fully implemented the above replacement policy, we leave for future work its full evaluation and don't consider it any further here.

3.3.6. Summary The client kernel intercepts each open request from a local application to a remote file. During a read operation, the cache manager uses the hash table to identify the local file that stores the requested chunk. If the chunk is not locally available, the cache manager picks a local file and opens it for access. The requested data is transferred from the remote server to the local kernel buffers. Subsequently it is copied to the user-level address space of the application and the local cache file. Subsequent requests to the same remote file are served faster by taking advantage of locally cached file contents, and also metadata related to volumes, remote files and local chunks.

4. Experimental Evaluation

In the present section, we first describe the experimentation environment that we used to develop and evaluate the Hades prototype. Then, we make an extensive experimental evaluation on the parallel retrieval of remote files and the reuse of cached data across multiple clients.

4.1. Environment

In our experiments, we used rack-mounted x86 servers with one quad-core processor 2.33GHz, 2GB RAM and gigabit ethernet nic. Every server has two SATA disks each of 250GB, 7.5KRPM and 16MB. buffer. The servers run the Debian distribution of Linux kernel version 2.6.18. The Hades implementation is based on version 1.4.5 of OpenAFS, Kerberos version 5 and version 2.2 of user-level NFS server. Unless otherwise specified, we used the default chunk size of 256KB and cacheblock size 4KB, respectively, for transfer and storage of data at the proxy cache.

4.2. Retrieval of Cached Data

Our first set of experiments uses a microbenchmark that we run directly at the proxy server. Our purpose is to evaluate the comparative advantage of Hades with respect to OpenAFS, when we read files stored at the origin server. We measure the latency to read each file block and the corresponding transfer throughput at the proxy server. We consider three file access modes that differ in the concurrency of the transfers and the involvement of the origin server during their service. We refer with Par and Seq to the parallel and sequential transfers, respectively, and we use Cd and Wm for the cold and warm proxy disk cache. Below we describe our three access modes:

Par/Cd. The files are requested with the proxy cache empty. The proxy server first prepares the mapping from the requested files to the local files, then it transfers the files from the origin server to the local page cache in chunk units, and finally it copies the files to the user-level memory of the proxy server in blocks of 4KB.

Par/Wm. The files are requested concurrently after the proxy disk cache has been warmed up. We enforce local disk accesses by flushing the memory page cache before starting the experiment.

Seq/Wm. Depending on the file size, the previous two cases initiated multiple threads across one or two users to request concurrently multiple files. In this mode, we only have one user making a sequence of file accesses from the warm disk cache of the proxy server.

In our experiments we transfer files of four different sizes:

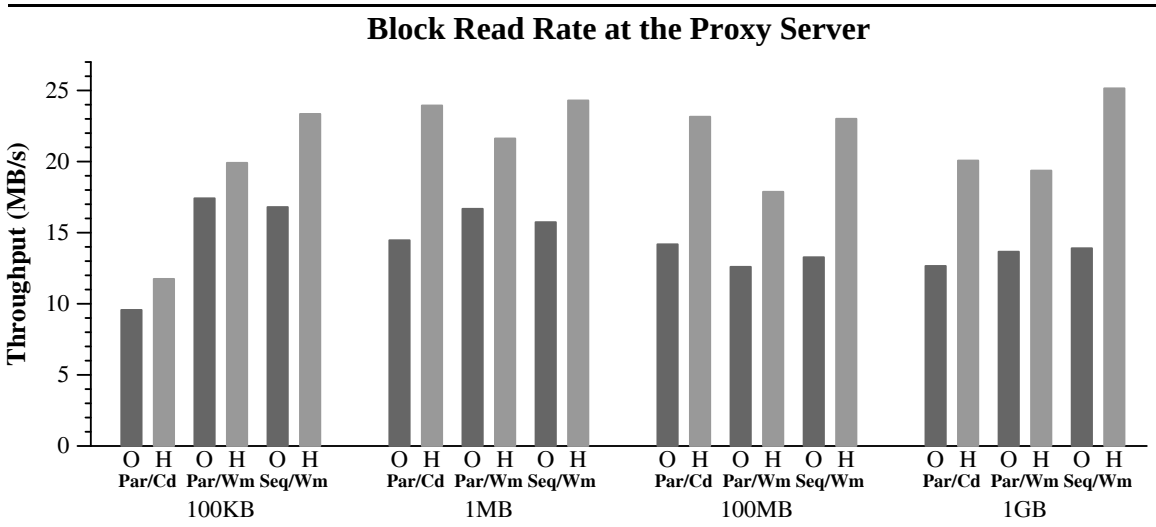

Figure 4. We measure the file access throughput at the proxy server across different sizes of transferred files. Consistently, Hades achieves a substantial throughput improvement with respect to OpenAFS that gets up to 80%. See text for explanation of the Par/Seq and Wm/Cd abbreviations.

100KB. We have either two users reading in parallel two separate sequences of 1000 files each, or one user reading a sequence of 1000 files.

1MB. We have either two users reading in parallel two separate sequences of 500 files each, or one user reading sequentially 500 files.

100MB. In the first two modes we have the transfer of five files in parallel, while in the third mode we only read one file sequentially.

1GB. We transfer in parallel five files for the first two modes, and do a sequential read of one file for the third one.

In Figure 4, we measure the average throughput during the sequential and parallel file transfers across the different file sizes. It is remarkable that Hades improves the measured throughput across all cases. The lowest throughput that we measure is 9.56 MB/s, when the OpenAFS client reads in parallel two sequences of 100KB files from a cold proxy cache. The corresponding Hades throughput is 23% higher at 11.74 MB/s. The highest throughput of OpenAFS is 17.42 MB/s for 1000 files of 100KB read from a warm proxy cache, while the highest throughput of Hades is 25.15 MB/s for a single file of 1GB read from a warm proxy cache.

We attribute the improvement of Hades to different reasons across the cases that we examine. In Figure 5 we can see the breakdown of the block read latency. The read time of each block is spent across (i) mapping the requested block to the offset of the local cache file, (ii) fetching from the origin server and storage to the local cache, (iii) copying from the local cache to the user-level memory. In the category of fetching, we include the rest of unaccountable transfer delays.

We note that the initial read from the cold proxy cache incurs substantial mapping overhead in OpenAFS. This is the cost to insert into the hash structure the information to find the cached remote blocks next time we look for them. Hades avoids this overhead by storing together in an array of chunk descriptors the mapping of all the chunks that correspond to the same remote file. We simplified additionally the hashing structure by attaching a linked list to each entry of the hash table. The length of the lists is short, since we only use a limited number of large local files. Other optimizations that we did include adding a hint for the local file of each remote file, and moving to the front of the list a found local file.

When the files are accessed from a warm proxy cache (Wm), the component fetch+other is negligible. Also, the mapping overhead is insignificant after the mapping structure has been created during the warming up. Thus, the dominant component in accessing the warm cache of the proxy is to get the data from the local disk. The reduction of the block read latency during the parallel transfers (Par/Wm) of Hades can be attributed to the spatial locality in the storage of the cached data. In particular, we store to the same local file the chunks of either the same remote file or different remote files retrieved from the same user. Instead, OpenAFS distributes

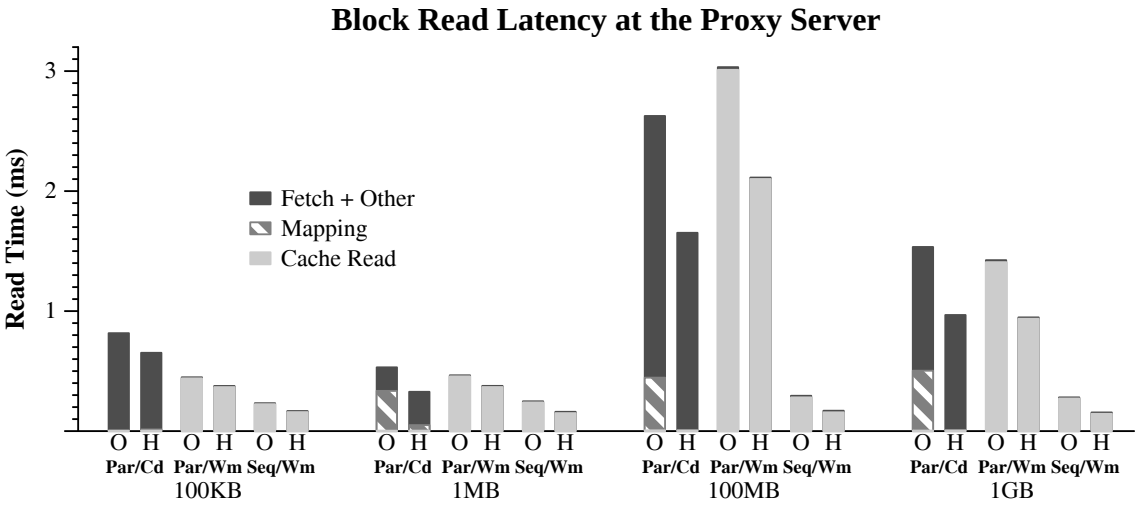

Figure 5. At the proxy server, we measure the time to read multiple files in parallel from the origin server (Remote), and in parallel (Parallel) or sequentially (Single) from the proxy disk cache. The latency to transfer each file block to the proxy server is broken down into fetching from the origin server, mapping to the local file, reading of the local file. In comparison to OpenAFS, Hades reduces substantially the block access latency up to 59%. See text for explanation of the Par/Seq and Wm/Cd abbreviations.

the retrieved chunks across an equal number of separate files in the proxy server. The same reason leads to the reduced block read time of Hades in comparison to OpenAFS when reading one or multiple files sequentially by one user (Seq/Wm). In comparison to the sequential transfer, parallel transfers share the available disk bandwidth and expand correspondingly the page read latency. For example, the bar height of the Seq/Wm measurement is approximately half or one fifth of the Par/Wm measurement depending on whether we have two or five parallel transfers.

In summary, we notice that our decision to cluster at the proxy server the cached data requested from the same remote file or by the same user ends up to a substantially improved read performance from the warm cache. Additionally, we improve the read performance from the cold cache by making more efficient the mechanism of mapping remote chunks to local file offsets.

4.3. Software Compilation

As an application to examine the general benefits of proxy caching, we use the building of linux kernel version 2.6.18. We assume that the source code is made commonly available from an OpenAFS volume (i) directly to OpenAFS clients (OO), (ii) to NFS clients through a proxy server running unmodifed OpenAFS client and user-level NFS server (OON), (iii) to NFS clients through the Hades prototype (OHN). We know in advance that the relative benefits of Hades in comparison to the original OpenAFS client are mostly evident when we retrieve large files from a warm cache. Instead, the present experiment we retrieve small files of a few kilobytes from a cold cache. Nevertheless, the software build is a baseline benchmark typically used in such types of experimentations [4].

In Figure 6(a) we measure the number of received and transmitted bytes at the origin (S) and the proxy (P) server, when we have one client (1), four clients (4) and four clients with the proxy at a distance from the origin of 50ms round-trip time (4D). Obviously, when we increase the number of clients from one to four, there is corresponding increase in the throughput of the origin server at the OO configuration. Instead, the intervention of the proxy server keeps constant the consumed bandwidth at the origin server, as we see in cases OON and OHN.

On the other hand, even with one client talking to the proxy, the NFS system consumes an excess of four time more network bandwidth than what OpenAFS requires for the same connection. Admittedly, the NFSv3 protocol that we use has been previously described as too chatty [4,13]. In fact, the version 4 of NFS makes more efficient the communication between the client and the server for example through

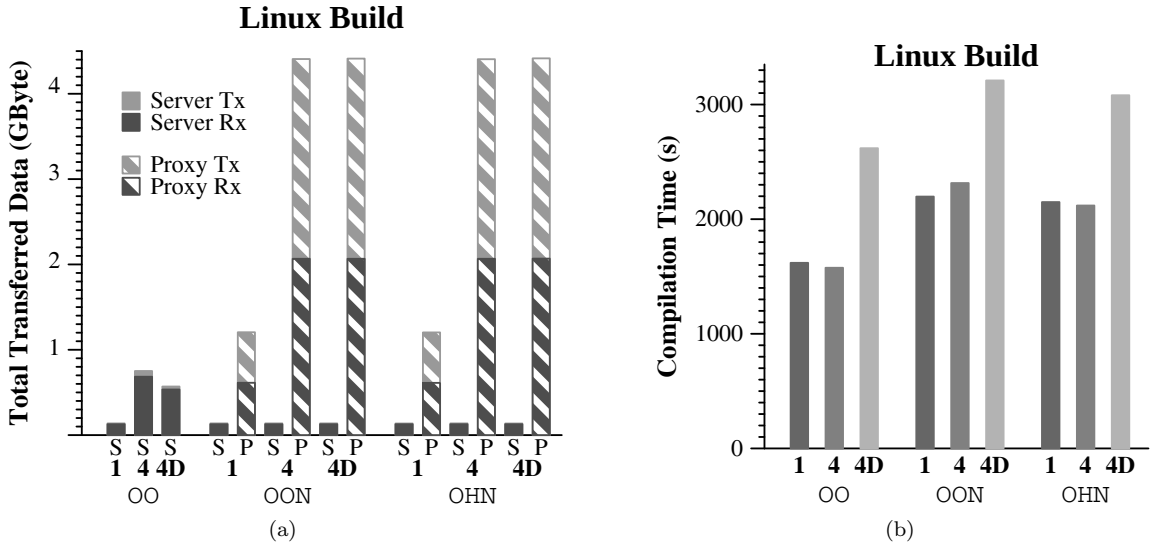

Figure 6. We build the Linux kernel on one (1) client, four (4) clients, and four clients with the origin 50ms away (4D). O refers to OpenAFS, N to NFS and H to Hades. The proxy cache is cold before each experiment that uses it. (a) We measure the total number of received and transmitted bytes in the origin (S) and the proxy (P) server. (b) With cold proxy cache, the intervention of the proxy server increases the compilation time. For retrieved files of only a few kilobytes each, Hades only achieves a modest reduction from 2315 to 2119 s (8.5%) in comparison to the original OpenAFS.

delegations and compound statements. However, for a simple scenario, where multiple clients only read data from a common server without any modifications, the cost of NFS seems excessively high. Therefore, our consideration of OpenAFS as an alternative protocol for building proxy servers demonstrates a lot of potential. In Figure 6(b), we compare the compilation time across the different systems configurations and numbers of clients. As we see, the direct connection between the OpenAFS client and server leads to the shortest compilation time. The benefit of Hades with respect to the unmodified OpenAFS is only limited to 8.5%. This behavior is justified from the small file sizes that typically dominate source codes.

Overall, we conclude that proxy servers can reduce the required network bandwidth from the origin server, but they may introduce access delays during the first access of the requested data from a cold cache. Furthermore, OpenAFS requires significantly less bandwidth when compared to NFS, even though the latter is considered defacto choice for proxy server in the latest related research projects.

5. Related Work

Howells introduced the FS-Cache facility that can be used by a network file system to cache data on local disks [5]. Previous evaluations showed some performance limitations due to double buffering across the local file system and the client of the network file system. Gulati et al. implemented the Nache caching proxy for the NFSv4 [4]. The proxy uses an NFSv4 client to access the remote server, an NFSv4 server to reexport the client to the local users, and CacheFS to cache files in persistent storage. Instead, in the present paper we examine OpenAFS as an alternative basis for building a proxy server. Muntz and Honeyman used simulations to show that the request rate seen by the origin server was not significantly reduced by the proxy server due to the relatively low sharing across the different clients in the traces that they studied [10].

Stolarchuk uses several hints in order to improve the speed of the common case of the AFS Cache Manager [17]. After reducing the overheads of cache consistency checks and file-to-chunk mapping, access of the AFS cache becomes comparable to that of the local file system. Additionally, we consider storage locality as an alternative direction to improve performance. Sivathanu and Zadok proposed

the xCachefs framework that allows to persistently cache the data from any slow file system to a fast file system [16]. They use a directory structure at the cache as exact copy of the source file system, while we organize the cached data at the proxy in ways that improve storage locality.

Matthews et al considered the dynamic reorganization of the stored data in order to improve the read performance of the log-structured file system [8]. In a different work, Vongsathorn and Carson proposed a disk subsystem that adaptively corrects the disparity between expected and actual access pattern by reorganizing the disk data [18]. Instead, we organize the remote data when first cached at the proxy server disks by file id and requesting user. Young proposed the Landlord deterministic online algorithm for replacing files of specific size and retrieval cost in a limited-size cache [20]. The Adaptive Replacement Cache (ARC) automatically keeps a balance between recency and frequency in an online and selftuning manner [9]. We consider data replacement a major issue in proxy servers for distributed file system that deserves further investigation.

6. Conclusions

We introduced several methods that improve the efficiency of storage and metadata management in a proxy server for distributed file systems. Through our prototype implementation we experimentally evaluate the performance and related cost across different file sizes and numbers of clients. Even though a proxy server can reduce substantially the required network bandwidth at the origin server, it may introduce additional latencies when the cache is cold. In the future, we plan to experimentally evaluate additional applications and investigate alternative data replacement methods using our prototype.

7. Acknowledgements

In part supported by project INTERSTORE with contract number I2101005 of the INTERREG IIIA Greece-Italy 2000-2006 program.

References

[1] W. J. Bolosky, J. R. Douceur, D. Ely, and M. Theimer. Feasibility of a serverless distributed file system deployed on an existing set of desktop pcs. In *ACM Sigmetrics*, pages 34–43, June 2000.

[2] A. J. Borr. Secureshare: Safe unix/windows file sharing through multiprotocol locking. In *USENIX Windows NT Symposium*, pages 13–23, Seattle, WA, Aug. 1998.

[3] P. Cao and S. Irani. Cost-aware www proxy caching algorithms. In *USENIX Symposium on Internet Technologies and Systems*, pages 193–206, Monterey, CA, 1997.

[4] A. Gulati, M. Naik, and R. Tewari. Nache: Design and implementation of a caching proxy for nfsv4. In *USENIX Conference on File and Storage Technologies*, pages 199–214, San Jose, CA, 2007.

[5] D. Howells. Fs-cache: A network filesystem caching facility. In *Proceedings of the Linux Symposium*, Ottawa, Canada, 2006.

[6] S. Jin and A. Bestavros. Popularity-aware greedydual-size web proxy caching algorithms. In *IEEE International Conference on Distributed Computing Systems*, pages 254–261, Taipei, Taiwan, 2000.

[7] E. P. Markatos, M. G. H. Katevenis, D. Pnevmatikatos, and M. Flouris. Secondary storage management for web proxies. In *USENIX Symposium on Internet Technologies and Systems*, pages 93–114, Boulder, CO, 1999.

[8] J. N. Matthews, D. Roselli, A. M. Costello, R. Y. Wang, and T. E. Anderson. Improving the performance of log-structured file systems with adaptive methods. In *ACM Symposium on Operating Systems Principles*, pages 238–251, Saint Malo, France, 1997.

[9] N. Megiddo and D. S. Modha. Arc: A self-tuning, low overhead replacement cache. In *USENIX Conference on File and Storage Technologies*, pages 115–130, San Francisco, CA, 2003.

[10] D. Muntz and P. Honeyman. Multi-level caching in distributed file systems. In *USENIX Winter Technical Conference*, pages 305–313, San Francisco, CA, 1992.

[11] E. Otoo and A. Shoshani. Accurate modeling of cache replacement policies in a data grid. In *IEEE/NASA Goddard Conference on Mass Storage Systems and Technologies*, pages 10–19, San Diego, CA, Apr. 2003.

[12] B. Pawlowski, C. Juszczak, P. Staubach, C. Smith, D. Lebel, and D. Hitz. Nfs version 3 design and implementation. In *USENIX Summer Technical Conference*, pages 137–152, Boston, MA, June 1994.

[13] B. Pawlowski, S. Shepler, C. Beame, B. Callagham, M. Eisler, D. Noveck, D. Robinson, and R. Thurlow. The nfs version 4 protocol. In *SANE Conference*, Maastricht, Netherlands, May 2000.

[14] M. Satanarayanan. Scalable, secure, and highly available distributed file access. *Computer*, 23(5):9–21, May 1990.

[15] E. Shriver, E. Gabber, L. Huang, and C. A. Stein. Storage management for web proxies. In *USENIX Annual Technical Conference*, pages 203–216, Berkeley, CA, 2002.

[16] G. Sivanathu and E. Zadok. A versatile persistent caching framework for file system. Technical Report FSL-05-05, Department of Computer Science, SUNY Stony Brook, Stony Brook, NY, 2005.

[17] M. T. Stolarchuk. Faster afs. Technical Report TR 92-3, CITI, University of Michigan, Ann Arbor, MI, 1992.

[18] P. Vongsathorn and S. D. Carson. A system for adaptive disk rearrangement. *Software-Practice and Experience*, 20(3):225–242, 1990.

[19] www.ehternetbuildings.com.

[20] N. E. Young. On-line file caching. In *ACM-SIAM Symposium on Discrete Algorithms*, pages 82–86, San Francisco, CA, 1998.