CITI Technical Report 93–2

Integrating Mass Storage and File Systems

C. J. Antonelli
cja@citi.umich.edu

P. Honeyman
honey@citi.umich.edu

ABSTRACT

This paper describes current and anticipated work at the University of Michigan's Center for Information Technology Integration (CITI) in developing and integrating mass storage with distributed file systems, specifically with the Andrew File System (AFS).

After surveying existing mass storage and associated file systems, this paper presents one approach to integrating AFS with mass storage. We consider the mass store itself to be the file system, not a bag on the side of a disk-based file system. This unifying perspective distinguishes our approach from other large-scale file systems.

Instead of developing a back-end server to manage the movement of data files between traditional disk-based storage systems (employed, in our case, by AFS) and magnetic-tape or optical-based mass storage systems (of which AFS has little or no knowledge), we envision the mass store as a first-class data repository. A traditional disk-based file system serves as a (very large) cache of the mass store system. On top of that is another, large, high-speed memory cache. All storage other than the mass store is used exclusively for caching. In this approach, cache management policies are of fundamental importance.

Two main requirements for this work are that the AFS name space remain unchanged, and performance seen by users must not suffer. For example, users must not have to pre-stage files explicitly in order to achieve acceptable performance.

April 15, 1993

Center for Information Technology Integration
University of Michigan
519 West William Street
Ann Arbor, MI 48103-4943

Integrating Mass Storage and File Systems

C. J. Antonelli
P. Honeyman

April 15, 1993

The University of Michigan has deployed a campus-wide file system based on the Andrew File System, first developed by Carnegie-Mellon University [1]. AFS employs a client-based disk cache to reduce file traffic to remote file servers. In brief, when a process uses an AFS file, all file operations are satisfied from a local cache if possible. When an AFS file is closed, any modified blocks are written to the server. AFS employs a server-mediated cache consistency protocol that flushes a stale cached file from a client cache when the file is modified by others.

At CITI, we have developed the Institutional File System (IFS), a modified version of AFS that facilitates its deployment to campus [2]. Our goal was to scale up AFS to meet the needs of a large institution by increasing both the number and the diversity of the supported client platforms, thereby allowing users to easily move from one client to another while all of their files remain accessible through AFS.

To help meet this goal, we did three things. First, because the most abundant platform on campus is an Apple Macintosh, we developed an intermediate file server that provides cache management functions for the Macintosh machines without requiring changes to the machines themselves. Second, we ported the AFS file servers to the several operating systems running on our IBM mainframes in order to take advantage of the increased throughput and permanent storage offered by these machines. Finally, we developed an intermediate caching server located between an AFS file server and an AFS client; the goal here was to off-load the main server by satisfying client requests from a caching-only server [3].

Of course, we also support several different kinds of workstations fitted with their own AFS cache managers, and we run several workstation-class AFS file servers. This campus topology is shown in Figure 1.

All told, we plan to support 30,000 end-systems via IFS when campus deployment is complete in the mid-'90s.

The Problem

The campus is rapidly heading toward a server disk space crisis. As IFS gains popularity on campus, concern grows over the cost of maintaining all permanent storage on magnetic disk. This problem is particularly severe with our campus mainframes, on which we would like to store the majority of our AFS files, but where disk storage is relatively expensive.

Currently, the IFS Deployment Group makes approximately 100 GB of disk storage available on campus. This gives the average student user 3 MB of IFS disk space, which is regarded as insufficient. While students obtain a larger allocation for class projects, it is

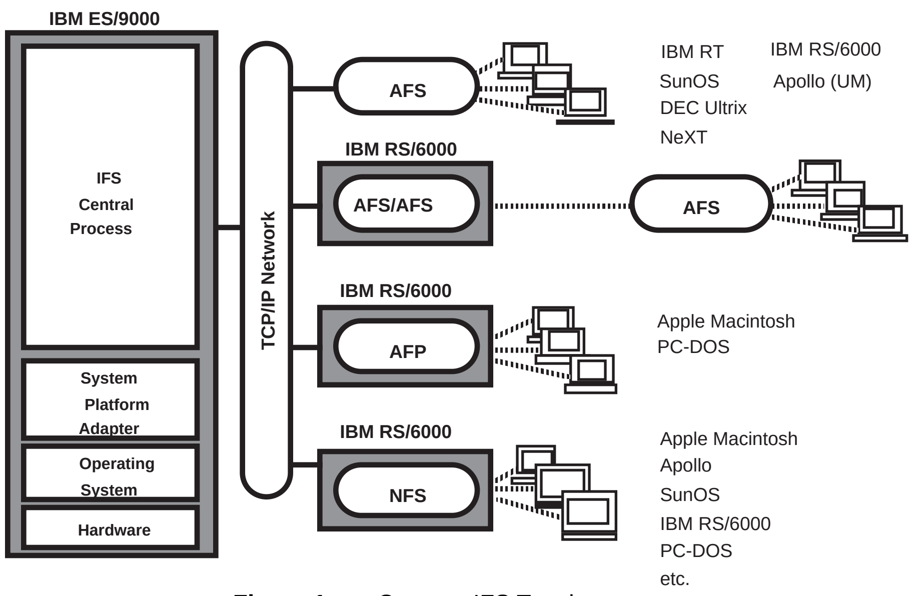

Figure 1. Campus IFS Topology

usually of limited duration. Students therefore cannot store very much permanent data in IFS and thus cannot take advantage of one of its strong points, namely the ability for users to move from machine to machine while retaining access to their files. In fact, at today's densities, a few floppies can hold more data than we give student users in IFS. We are unlikely to be able to double or quadruple this disk storage space while relying exclusively on traditional magnetic disk storage systems.

File sizes are going up. While the average size of a vanilla UNIX file used to be 11 KB [4], we observe a steady increase in file sizes required by windowing, graphics, audio, and video applications. In a preliminary study of one IFS server, we found file sizes that are substantially larger than those observed in earlier studies—over 40 KB for files opened for reading and over 64 KB for files opened for writing, on average—and we expect this trend to continue. In addition, the LADDIS synthetic NFS server benchmark assumes a small file is 136 KB [5,6].

Many applications exist today that could immediately take advantage of vastly increased IFS storage. For example, census data manipulated by the Population Studies Center currently use 4 GB of IFS space; this usage could increase immediately by at least two orders of magnitude if enough IFS space were available.

Requirements

Faced with the need for ever more on-line storage, it is appealing to consider a mass-storage system as the central data repository. In this section, we consider various requirements that must be met by a mass-store-based IFS file server if it is to be a viable alternative to today's disk-based architectures.[1]

It is well known that conventional mass storage systems trade storage capacity for access latency. For example, it may take a minute or more to locate a tape in a repository, move it to an available drive, mount it, and obtain access to a given data block on the tape. The impact of such increased delays is twofold: first, users may be forced to wait for long periods before being able to access the first byte of their data; second, the extreme delays in re-

1. We use the term "mass store" to refer to the automated component of a mass storage system, and the term "mass storage" to refer to the whole system.

sponse to client remote procedure call requests may result in total collapse of the RPC mechanism that underlies most distributed file systems, AFS in particular. The requirement here is to offer IFS services from mass storage systems with essentially no additional observable delay; in exceptional cases, no communications or user protocols should break due to the additional delay.

We must serve the needs of both student and institutional users. This translates into the requirement that the file system support both smaller student files as well as much larger files, such as those needed by Population Studies, with an acceptable degree of efficiency.

Larger files will occupy more space in an IFS client or intermediate cache. It will be necessary to investigate new cache management policies, ones that better conserve cached files. For example, we may wish to prevent a single large file from occupying a significant fraction of any cache, thereby invalidating most other files from the cache and decreasing performance when accessing these files. At the same time, it may not be a good idea to flush the first piece of any file from the cache, since applications are often satisfied with examining the first few bytes of a file. In a similar vein, on multi-user systems, the cache manager might attempt to allocate fairly the limited cache resource among the users.

Users should not be required to learn a new set of administrative commands to use a file server based on a mass store. For example, in our ideal, it must not be necessary for users to pre-stage a file to a disk before running a program that accesses the file.

Finally, while we are deploying IFS to campus, we are currently examining DFS[2], in part to determine its suitability as a replacement for AFS. Whatever caching policies we implement now must not preclude us from making the transition from the current AFS-based campus file system to a DFS-based one.

2. DFS is the file system component of the Open Software Foundation's Distributed Computing Environment (OSF/DCE). DFS is the latest version of Transarc's AFS.

Related Work

In this section, we summarize some existing file systems that utilize mass store.

NAStore

NASA Ames' NAStore extends a typical UNIX file system with a mass store back end [7, 8]. It does this by keeping the file system metadata on magnetic disk and storing additional information in a file's inode that permits the data blocks of the file to reside on the mass store. When a file is accessed, its blocks are retrieved from the mass store and written to disk. A utility program periodically scans the disk and migrates old files back to the mass store, and provision is made for a "panic dump" of preselected files back to the mass store if the file system finds itself short of space.

NAStore attempts to mitigate first-byte latency by blocking a read only until the data it needs has been read from the mass store; reads for data that have already transferred to disk are satisfied immediately. However, programs that do sequential reads from the beginning of a file will block until the mass store delivers the data. Users can avoid this behavior by caching files manually onto a disk.

InfiniteStorage

Epoch Systems' InfiniteStorage Architecture (ISA) also extends a typical UNIX file system with a mass store [9]. It does so by introducing a wrapper layer between the VFS layer and the native file system implementation. The wrapper handles access to file data on the mass store; directory information and metadata remain on magnetic disk. Free space on magnetic disk is managed by staging out unused files when disk usage approaches a threshold; this staging occurs both when needed and at fixed intervals. Cache misses are not severe because ISA uses erasable optical disks, whose first-byte latency is around ten seconds, at the second level of its storage hierarchy. However, files that have been transferred to magnetic tape at the third level will cause extended first-byte latencies.

UniTree

General Atomics' UniTree Central File Manager does not augment a UNIX file system with a mass store [10, 11]. Rather, UniTree has implemented major portions of the Version 4 IEEE Reference Model [12]. Files reside on magnetic disk until they are migrated to a lower level in the storage hierarchy, typically to magnetic tape. Migration is under control of administrative processes. Migrated files retain copies on magnetic disk for faster access unless free disk space falls below a threshold at which time the redundant disk copy is purged. UniTree manages its own namespace via a set of distributed servers and thus avoids limitations associated with storing file system metadata on disk. Files must be staged manually to avoid extended first-byte latencies, which are considerable due to UniTree's policy of caching an entire file when it is first accessed.

Multiple-Residency AFS

The Pittsburgh Supercomputing Center (PSC) has undertaken several projects to extend AFS [13]. One of these provides mass store support and multiple copies of data. The standard AFS vnode is augmented with auxiliary vnodes that point to other copies, or residencies, of a given file. Typically, the other copies reside on a mass store. Random access to data without a disk residency is handled by first creating a disk residency. The PSC migration strategy strives to keep space available for disk residencies by migrating unused files to the mass store. As long as enough disk storage is available, the first-byte latency problem is solved directly via the multiple residency mechanism, which also provides redundancy.

Plan 9

In the Plan 9 [14] file system, 128 MB of high-speed RAM acts as a cache for 100 GB of magnetic disk that acts as a cache for 600 GB of WORM optical disk. Once per day, file system activity is interrupted briefly to allow a checkpoint of changed file system data to be made. The checkpointed data is then written to optical disk while normal file system activity resumes. In this fashion, the relatively slow WORM write time is masked by the disk cache, and all previous versions of the file system are available on the WORM. Plan 9 uses a comprehensive set of administrative procedures to make current and checkpointed files available, but to users it looks like an ordinary file system, much like a UNIX file system.

Proposed Approach

In constructing a mass store-based file system, we propose using an approach different from those used in traditional mass-storage systems, and embrace the Plan 9 approach. Instead of augmenting an existing file system with a mass store—in effect, the mass store becomes a bag hanging from the side of the file system—or developing special strategies for handling a file system that uses a mass store, we propose that the mass store *be* the file system.

In other words, our approach treats a mass store as another instance of a storage device, and we choose to manage it as one. This approach is aligned with the recommendations of Christman, et al. [15], in that we are proposing a transparent byte-level interface to files located on mass store, leading to an operating-system-controlled distributed file system.

Although the Plan 9 file system was designed to be used in a networked environment in which clients obtain file services from any nearby servers, it is optimized to support a few DMA-attached CPU servers. The challenge for us is to scale up the Plan 9 approach to a campus of tens of thousands of end-systems using a few large file servers.

Caching Strategy

Our view of the mass store as a file system allows us to recast the problem of providing reasonable performance from a potentially slow storage system to that of designing a high-performance caching strategy. This problem is not fundamentally different from designing a caching strategy for any file system. Instead of designing special-purpose file migration policies aimed specifically at mitigating mass store deficiencies, a good caching strategy implements these policies as part of its function.

Thus, in our approach, a correct caching strategy is of crucial importance; a failure here is disastrous. Drawing on traditional operating system caching practices, as well as more recent work in mass store-based file systems, we present a partial list of relationships we believe a successful caching strategy must exploit:

- **Temporal Locality—**When accessing a block of a file, the chances of requiring access to the block again in short order are fairly high. Caching the block exploits temporal locality.
- **Spatial Locality—**When accessing a block of a file, the likelihood of requiring access to the next block is high [4]. Prefetching the next block in sequence into the cache exploits temporal locality. Large block sizes also take advantage of spatial locality.
- **File Locality—**The NAStore architects noticed that when a user accesses one archived file, in a significant fraction of instances that user will, in short order, access several more files that were archived to the same tape [16]. Restoring all files on a given tape exploits file locality by obviating repeated access to the mass store.
- **Metadata Cost—**Some blocks in the cache are more important than others. In particular, cache misses on metadata blocks such as directories and other information necessary for traversing the file system name space are more costly than misses on ordinary data blocks, as are losses of metadata due to a failure to migrate to the mass store. On the other hand, unconditionally reserving space in the cache for metadata is wasteful, particularly in view of the fact that parts of any large file system tree are usually devoid of visitors at any given time. We think a strategy that weights blocks according to their relative importance—and that permits this weighting to change dynamically—will be helpful here.
- **Acquisition Cost—**Many caching strategies assign the same weight to all cache blocks of a given type, such as data blocks. This strategy gives the same priority to blocks that were easy to obtain, in terms of expended resources, as it gives to blocks that were not. A strategy that takes into account the cost of reacquiring a block when determining which cache block to discard to make room for an incoming block will help here. We plan to use the cost of acquisition as a first approximation to a block's reaquisition cost.
- **Disposal Cost—**Similarly, it is more expensive to discard some cache blocks than others, since some cache blocks have to be written to the mass store and others do not. Among those that have to be written, some writes will be more resource intensive than others. Choosing appropriately which blocks are cheapest to discard exploits disposal cost.
- **Navigation Cost—**Operators of mass storage have observed that when a user lists the contents of a directory, an access to one or more files of that directory will follow shortly thereafter. In systems that support current user directories, changing to a new current directory may offer the same hints, indicating a higher probability of access to files in the current directory. If also true of IFS users, this

observation translates into a strategy that starts prefetching files when a user changes to a new current directory and aborts this prefetching when the user leaves.

- **Block Cost—**Throwing away some blocks of a file is more costly than discarding others. For example, the first block of a file is much more popular than all the other blocks, because file inspection tools typically need access to the first block only. The last block of a file is also popular, since log files are usually accessed there. This observation translates into an increased weighting for such popular blocks.

Overall, the weight assigned to a block is determined by (a) the cost to the system if the block is not retained in the cache and (b) the cost to the system in obtaining the block from the mass store. Here the cost to the system for (a) can be measured in many ways, e.g., unnecessary delay in accessing the first few bytes of a file in order to determine its type, or the timing out of an NFS client because of delays in traversing a directory tree. For (b) the cost is most likely a fairly static function of the type of mass store; i.e., an optical-disk jukebox should have a much lower cost for obtaining a block than, say, a magnetic-tape repository with a robotic retrieval system.

Multilevel caching problems can be attacked with this approach by increasing the weights of blocks in upstream caches based on recent accesses by downstream caches.

Our goal is to determine the efficacy of exploiting these relationships and to incorporate the effective relationships, and possibly others, into a single, cohesive caching strategy.

Status

Currently, we are simulating the above caching strategy using our corpus of file system trace data [17]. We are experimenting with various cache sizes, initial weights, aging and reverse aging functions, multilevel caches, and so forth.

We have obtained a version of Plan 9 and are in the process of installing it, after which we will start gaining experience with its file system.

If our simulations show that our approach is effective, we will build and evaluate a prototype file system based on mass store.

Conclusion

We have described current and proposed work in developing and integrating mass storage with file systems. Our approach considers the mass store to be the file system, not an appendage of a disk-based file system. In this scheme, caching is critically important. We have identified a set of strategies that we believe are important to effective cache management for mass stores; we are in the process of simulating these strategies using existing file system trace data. Once we have determined an effective overall strategy, we plan to build and evaluate a prototype.

Acknowledgements

Mike Stolarchuk has been talking about cost-of-acquisition-based caching for years. This work is partially supported by IBM.

References

1. Morris, James H., Mahadev Satyanarayanan, Michael H. Conner, John H. Howard, David S. H. Rosenthal, and F. Donelson Smith, “Andrew: A Distributed Personal Computing Environment,” *Communications of the ACM*, Vol. 29, No. 3, pp. 184–201 (March 1986).

2. Hanss, Ted, “University of Michigan Institutional File System,” */AIXTRA: The AIX Technical Review*, pp. 25–32 (January 1992).

3. Howe, James, “Intermediate File Servers in a Distributed File Server Environment,” CITI Technical Report 92–4 (June 1992).

4. Ousterhout, J., H. L. DaCosta, D. Harrison, J. Kunze, M. Kupfer, J. Thompson, “A Trace-Driven Analysis of the UNIX 4.2 BSD File System,” *Proceedings of the 10th ACM Symposium on Operating Systems Principles*, Orcas Island (December 1985).

5. Keith, Bruce and Mark Wittle, “LADDIS: The Next Generation in NFS File Server Benchmarking,” *Summer 1993 USENIX Conference Proceedings*, Cincinati, OH (to appear June 1993).

6. Pawlowski, Brian, personal communication, December 1992.

7. Tweten, David, “Hiding Mass Storage Under UNIX: NASA’s MSS-II Architecture,” *Tenth IEEE Symposium on Mass Storage Systems*, pp. 140-145, Monterey (May 1990).

8. Hahn, Jonathan, Bob Henderson, Ruth Iverson, George Navas, Alan Poston, Tom Proett, Bill Ross, Mark Tangney, and Dave Tweten, “NAStore External Reference Specification,” NAS Systems Division, NASA Ames Research Center, Moffett Field (January 1992).

9. Foster, Antony, and David Habermehl, “Renaissance: Managing the Network Computer and its Storage Requirements,” *Eleventh IEEE Symposium on Mass Storage Systems*, pp. 3-10, Monterey (October 1991).

10. Hogan, Carole, Loellyn Cassell, Joy Foglesong, John Kordas, Michael Nemanic, and George Richmond, “The Livermore Distributed Storage System: Requirements and Overview,” *Tenth IEEE Symposium on Mass Storage System*s, pp. 6-17, Monterey (May 1990).

11. McClain, Fred, “DataTree and UniTree: Software for File and Storage Management,” *Tenth IEEE Symposium on Mass Storage Systems*, pp. 126-128, Monterey (May 1990).

12. “Mass Storage System Reference Model: Version 4”, edited by Sam Coleman and Steve Miller, *IEEE Technical Committee on Mass Storage Systems and Technology* (May 1990).

13. Nydick, Daniel, Kathy Benninger, Brett Bosley, James Ellis, Jonathan Goldick, Christopher Kirby, Michael Levine, Christopher Maher, and Matt Mathis, “An AFS-based Mass Storage System at the Pittsburgh Supercomputer Center,” *Eleventh IEEE Symposium on Mass Storage Systems*, pp. 117-122, Monterey (October 1991).

14. Quinlan, Sean, “A Cached WORM File System,” *Software Practice and Experience*, Vol. 21, No. 12, pp. 1289-1299 (December 1991).

15. Christman, Ronald D., Danny P. Cook, and Christina W. Mercier, “Re-Engineering the Los Alamos Common File System,” *Tenth IEEE Symposium on Mass Storage Systems*, pp. 122-125, Monterey (May 1990).

16. Tweten, David, personal communication, December 1992.

17. Blumson, S., P. Honeyman, T. E. Ragland and M. T. Stolarchuk, “AFS Server Logging,” CITI Technical Report, in preparation.