A Look at Modern File Systems

Mark Abbott, Yunfeng Fei, and Nadya Williams
CSE 221
November 30, 1999

Introduction

In recent years, a great deal of work has been done on distributed and parallel file systems. Although the motivations behind these two types of file systems are quite different, they share many underlying characteristics and techniques. Distributed file systems need to provide clients with efficient, usable, and scalable high-bandwidth access to stored data. Parallel file systems have a somewhat more straightforward goal: to allow multiple processes working together as a single task force to access a shared file or files in parallel. In these specialized systems, speed and scalability are the most important considerations, while issues like portability and ease of use are secondary. This paper will examine eight modern distributed and parallel file systems, looking at the design principles and goals behind each, their actual implementations, the advantages and disadvantages each entails, the innovations each system introduces, and the performance and scalability each offers. The paper concludes with a brief comparison of these systems.

1. xFS

Design principles and goals. The overriding goal behind xFS is scalability. In the authors' view, the way to achieve this is to do away with the central server, distributing all the tasks it would usually do. In xFS, each node of the distributed system can fill one or more of the various roles required in a distributed file system, including storage server, client, or *manager*, a role which is xFS' principal means of distributing file system administration. xFS was designed with secondary goals as well. Reliability was one factor, as evinced by its use of RAIDs. Some other secondary goals, such as graceful crash recovery and dynamic load rebalancing, are addressed by certain unimplemented aspects of their design.

Architecture and Implementation. Each file's metadata is maintained by *metadata managers* that are distributed throughout the system. xFS uses a policy called First Writer to assign files to managers. Each manager maintains an imap; once the manager has been located, it can look up the file's metadata in this imap, including the disk log address. xFS uses a log-based file system (LFS) which is stored on a RAID. The final map used to translate from file index number to physical disk location is the *stripe group map*. The stripe group map performs the mapping of file index numbers onto stripe groups, giving the client access to a list of the disks that comprise that stripe group. The stripe group map is small and static, and is distributed globally.

The task of cleaning the LFS to consolidate data and create empty segments is also distributed in the xFS design, though not yet implemented.

Advantages. The principal advantage of xFS is its scalability. All other design considerations were subordinated to this.

Problems and Disadvantages. Some problems result from the lack of a central server. For example, there is more overhead and complexity in locating files than one would expect in a file system with a central server. Also, xFS requires cumbersome mechanisms for reaching consensus between multiple clients, such as the selection of a stripe group leader.

Another group of problems relates to security. Since there is no central authority, xFS clients must all be trusted and physically secure. In a sense, this imposes inherent limits on scalability. However, it is also possible for a self-contained xFS to be used by remote, untrusted machines in the same manner that remote machines can access an NFS. So as long as a centralized xFS can serve all the needs of the broader, untrusted community, security is not such an important issue.

Innovations. Aside from the notion of doing without the central server, xFS' innovations consist mostly of refinements of existing ideas. For example, log-based file systems were previously used in Sprite, and combined with RAIDs in the Zebra file system, but xFS introduces the concept of a manager map to distribute the administration of the LFS/RAID.

Performance. Each measurement of scalability-related performance was replicated on xFS, NFS, and AFS (an implementation of the Andrew file system), on 32 machines (SPARCStation 10 and 20's). In nearly every case, xFS performed poorly for small numbers of clients, but showed vastly superior performance for higher numbers of clients. The performance of NFS and AFS did not increase substantially with the number of clients, while xFS increased linearly in most tests.

Scalability. The initial set of performance tests involved large writes, large reads, and small writes. In each case, xFS' aggregate bandwidth increased roughly linearly; at 32 clients, xFS performed an order of magnitude better than NFS or AFS. The results from the Andrew benchmark show xFS finished 47% and 79% faster than NFS and AFS, respectively.

2. Calypso

Design Principles and Goals. Calypso is a distributed file system for UNIX clusters. The authors describe the Calypso recovery scheme which uses states distributed among the clients to reconstruct the dynamic server state. The main goal is to achieve a non-disruptive, scalable, modular, and very efficient file recovery system, while providing file server services. During the recovery, data consistency is guaranteed, and congestion control is provided.

Architecture and Implementation. The Calypso file system was implemented using a configuration of model 370 RISC System/6000 processors connected by a non-blocking switch. Multiported disks were attached to at least two nodes, but only one port was active at any given time. Each file system includes several clients and one server. There are three distinct subsystems in the Calypso file system: (1) the token manager that consists of the token clients and token server; (2) the virtual file system, including of data clients and servers; (3) the communication subsystem that provides remote services.

The Calypso recovery system consists of four separate subsystems - the group services, the recovery controller, the congestion controller, and the client state transfer functions and server state reconstruction functions. The group services include the node status service and the global mount. The node status service detects failed nodes and instantiates a recovery controller process. This process uses the global mount service to determine which node has failed and what recovery is needed. The global mount service maintains the required and actual status of file system mounts, client lists, and server and backup nodes. The recovery controller does its job by sequencing through three recovery phases. The separation of phases ensures cache consistency even during recovery.

Advantages. Calypso uses six token types to denote access to various parts of a file, and to authorize certain operations. A client requesting a token has to communicate with other clients who are holding this token to have the token released. This unconventional client-centric approach limits the number of messages at the server, and reduces server complexity.

Calypso offers a mount-time option that forces client cache flushes directly to the server disk. The separation of the subsystems in the recovery system enhances software development and maintenance providing the maximum modularity. Calypso uses inexpensive hardware redundancy for data availability.

Problems and Disadvantages. Writing modified data to the server cache after being flushed from the client cache can cause data loss during a server failure. Calypso offers an option to flush caches directly to the server disk, which increases disk utilization of the server and results in decreased performance.

Innovations. Only Calypso uses multi-ported disks to handle permanent processor failures or scheduled shutdown, resulting in a huge advantage in shortening recovery time. In the event of a server failure, the backup server that attaches to the multi-ported disks is able to take over the disks quickly and start functioning as the file server.

Performance. The authors stated that the goal of their measurements was not to benchmark the recovery time, but to understand the performance characteristics and bottlenecks of the recovery system. The result of the measurements shows that the JFS log-redo and state reconstruction times make up most of the total recovery time. Other results show that using a backup server and the dual-ported external disks dramatically reduces the reboot time.

Scalability. The Calypso clients maintain the majority of the state information themselves. Conceptually, the state reconstruction time will increase linearly with the number of clients. The measurements show no exponential behavior for the tested configuration of up to 32 clients, indicating fairly good scalability.

3. Vesta

Design Principles and Goals. The Vesta Parallel File System is intended for parallel applications. Typically, such file systems achieve parallelism by placing each sequential block on a different disk. Vesta's approach is different: it enables the user to specify how the information in a file will be distributed by dividing a file into *subfiles*. The user can control how the subfiles are laid out on the available disks through certain parameters supplied when the file is created.

Architecture and implementation. Vesta was implemented on an IBM SP1 multicomputer. It consists of two parts. Applications running on compute nodes use a client library, while the I/O nodes run a server. All metadata for a given file is stored on a single master I/O node. The data reside on multiple I/O nodes. The compute nodes sends messages to the appropriate I/O nodes requesting the data, and the I/O nodes pack all requested data into a single reply message, which also serves as an acknowledgment.

Cooperating processes share data by using different subfiles within a file; however, it is also possible to share a subfile by using a shared pointer. Since this is done without verifying that the read was successful, or that EOF was not reached, this method is not as reliable as sharing data by using separate subfiles.

Advantages. Vesta provides control over the layout of subfiles. A compute node is able to request file data in a single step, without directly accessing the file's metadata. The master node contains the file metadata and is responsible for tracking down and replying with the actual data. The policy of not caching data on compute nodes means Vesta does not need to worry about file consistency, or false sharing at the block level.

Problems and Disadvantages. Because of the unusual way in which it interlaces data on disk, it is very difficult to interface Vesta with other file systems. The user must provide a fair amount of Vesta-specific information in order to take advantage of its parallel access. Because of Vesta's minimalist approach to metadata, there is no inode and thus no block list, making it difficult to determine where a file ends. The flat, non-hierarchical name space makes it difficult to remove or rename Xrefs (the equivalent of directories). The network traffic is increased through additional requests to I/O nodes.

Innovations. Vesta's central innovation lies in what the authors refer to as its *two-dimensional file structure*. Vesta presents the user with a grid of blocks, where the columns, called *cells*, correspond to individual disks. A related idea is partitioning each file into *subfiles*. By providing parameters which allow the user to change the layout of subfiles on the grid, Vesta allows the user to optimize performance by taking expected access patterns into account.

Performance. Performance measurements were based on the FastMeshSort. These measurements showed that speedup was proportional to the number of I/O nodes: using 8 I/O nodes, speedup ranged from 4 to 7 times faster than a single node. The authors point out that speedup would certainly be best on I/O-intensive processes, of which FastMeshSort is an example.

Scalability. Vesta's creators took a number of steps to ensure their system would scale well. For example, they allow for a c*ollective attach* operation whereby a single process within a task force of parallel processes would attach to the files to be used and share their metadata with the other processes. The authors found that the bandwidth of the system scaled linearly with the size of the system, as long as the network itself was not overloaded and the ratio of compute nodes to I/O nodes was held constant.

4. Hurricane File System (HFS)

Design Principles and Goals. The Hurricane File System was designed for shared-memory multiprocessors. The main goal was to optimize the performance of parallel applications with diverse requirements. Files in HFS are implemented as building-block compositions. This is the basis of the flexibility of the HFS, which allows the applications to customize file structure and policies to meet its requirements.

Architecture and Implementation. The work was done on a Hector multiprocessor that is constructed from processing modules. Each module has a Motorola 88100 processor, 16Kb data and 16 Kb instruction caches, and a local portion of globally addressable memory. HFS files are implemented by combining a set of building blocks into compositions. Building blocks are implemented as objects that contain a state, and a set of operations that can manipulate the state. The import and export interfaces specify the operations invoked by the building blocks.

Simple building blocks exist to store data on the disk, distribute data to the other building blocks, prefetch data into memory, enforce security, provide compression/decompression, and interact with the memory manager to cache file data. Blocks can be exchanged, and new blocks can be created. Using many fine-grained building blocks rather then a few large ones, and using a larger number of building block types with the identical interfaces give more flexibility in defining numerous compositions. HFS is logically divided into 3 layers:

- *the application layer* is implemented as an Alloc Stream Facility (ASF) I/O library, and provides most of the functionality in order to minimize the servers' communication.

- *the physical layer* implements files, and controls the physical disks on the system. Logically this layer is below the memory manager, allowing the mapped file I/O to exploit its facilities.
- *a logical layer* connects the previous two, and provides the file system authentication services, naming (directories), and locking.

Advantages. Flexibility is the main advantage of HFS. The building block approach gives applications the ability to customize both the file structure, and the file system policies implemented by all three layers of the HFS. The building blocks can be changed at run-time, allowing the application to adapt to different access patterns dynamically during different phases of the execution.

Problems and Disadvantages. Most of the basic operating system, and hardware infrastructure had to be developed from scratch. Test experiments were limited by the small size of the system. Particular characteristics of the Hector multiprocessor, such as slow memory system and lack of cache coherence, decrease HFS' performance.

Innovations. The building-block composition of the HFS structure is unique. The authors took what they called a *holistic* view of the file system, meaning that they considered all the system servers and parts that affect the I/O performance together, instead of one by one. The building blocks are stored in a regular file, and the file is stored on disk. This allows HFS to colocate building blocks and their associated data, and store blocks redundantly for fault tolerance. HFS provides a single consistent technique for customized functionality at all three levels of the system.

Performance. Using building block compositions resulted in greatly reduced code complexity, in easy porting, and in substantial performance improvements. For example, UNIX *diff* runs four times faster using ASF than using the native *stdio* library on an AIX system. It takes only a few seconds to do a crash recovery. The variety of possible building block compositions allows for prefetching, locking, and file cache management policies to match the applications' access patterns. HFS can deliver 100% of the disk bandwidth to the application address space by matching the application's access pattern with the file structure, and policies.

Scalability. The current implementation of HFS supports or can be easily extended to support all the currently existing policies used by other parallel file systems such as CFS, sfs, PIOFS, Vesta, and xFS.

5. Frangipani

Design Principles and Goals. The designers wished to create a scalable system that would give its users a shared access to all files, and provide more storage and increased performance. Frangipani is a two-layer system that runs on a cluster of trusted machines with secure communication under common administration. The lower layer is a distributed storage service that provides incrementally scalable, automatically managed virtual disks. The upper layer is the Frangipani file system, which is run by multiple machines on top of a Petal virtual disk. Coherence is ensured via a distributed lock service.

Architecture and Implementation. Frangipani is a layer on top of Petal that provides a virtual disk to all clients. Petal can replicate data for high availability, and it gives system snapshots for efficient backups. Most of Frangipani's scalability, fault tolerance, and ease of administration comes from the underlying Petal layer, with careful design extending these properties into the upper layer. Frangipani was implemented under DIGITAL Unix 4.0.

The distributed components of Frangipani are the lock servers, the Frangipani servers, and the

Petal servers. They provide different functions that can be assigned to different machines in many ways. Users programs access Frangipani through a standard operating system call interface. All the file servers read and write the same structures on the shared Petal disk but keep separate logs of pending changes in Petal. When the Frangipani server crashes, another server can access the log and run recovery while the rest of the system is running. Frangipani provides coherent access to the same files across multiple machines.

Advantages. The simplicity of the internal structure allows the handling of system recovery, reconfiguration, and load balancing with very little additional hardware. Server deletion and addition provide for load balancing and scaling. Frangipani can create consistent backups while the system is running, and the lock service coordinates virtual disk access, and keeps the kernel buffer caches coherent across multiple servers.

Problems and Disadvantages. Since any Frangipani machine can read and write the Petal virtual disk, Frangipani must run only on machines that have trusted operating systems. Full security has not been implemented yet. The minimal level of security that does exist is achieved by allowing a remote untrusted Frangipani client machine to connect to a Frangipani server machine.

Innovations. Frangipani is a new system in that it manages a collection of disks on multiple machines as a single shared pool of storage. The shared physical disk is replaced by a shared scalable virtual disk provided by Petal. This logical disk layer provides redundancy. Frangipani can dynamically alter its configuration to include new nodes, or remove failed ones. Thus the system remains available when some components fail.

Performance. Frangipani uses write-ahead redo logging of metadata that simplifies failure recovery and improves performance. When the system detects a crashed server, it gives the failed server log to a recovery daemon. Lock service consists of a set of mutually cooperating locks that communicate via asynchronous messages to minimize the amount of memory used and to increase flexibility and performance. The lock service is fully distributed to achieve fault tolerance and scalable performance.

The performance of a single Frangipani server was compared to DIGITAL's Advance File System (AdvFS). AdvFS and Frangipani perform similarly when reading small files, and directories. For large files, Frangipani has a good read and write throughput

Scalability. Frangipani has excellent single-server performance and can be scaled well with addition of new servers. Its scalability was tested with a configuration of seven Petal servers and six Frangipani servers.

6. Ufo

Design Principles and Goals. In this article, the authors present a new method for extending an existing operating system at the user level. Their primary goal was to create a mechanism for accessing files on multiple networked machines. The authors wished to implement their file system at the user level, so it could be used only by users who needed it, and installed and used without need for root access.

Architecture and Implementation. Ufo is implemented on a SUN Ultra 1 workstation with 64Mb of RAM, under Solaris 2.5.1. It works using a *Catcher* tool. The Catcher extends a Unix operating system at the user level, by catching and modifying selected system calls. The Catcher works by monitoring the */proc* virtual file system. The Catcher can be attached on a process-by-process basis, or it can be attached dynamically, to an already-running process.

Ufo captures all *open* system calls via the Catcher, and parses them to see if they are local or remote. Local opens go through untouched. If a remote file is being opened, Ufo checks to see whether there is a local cached copy, using a cache consistency protocol based on timeouts. If no local copy exists, Ufo retrieves the entire file from the remote site. In either case, Ufo then modifies the system call to open the local copy. Ufo uses write-back caching to minimize the number of times a file being written must be transferred back to its source.

Advantages. The primary benefits are ease of use and flexibility. Because Ufo runs as a user process, it can be installed, and used without root access. This is an important issue, since the idea is to make user's remote files transparent while using machines which may be spread over several institutions, machines over which the user has no authority. Furthermore, Ufo can be attached only to those processes which need it, when they need it.

Problems and Disadvantages. The Catcher is not able to control *setuid* programs. This is a minor problem, since the few programs that are installed as setuid generally would not use Ufo's remote file access. The Catcher is vulnerable to a SIGKILL signal: because it runs as a user process, it can be killed. The timeout mechanism for cache consistency relies on the user to define an appropriate tradeoff between performance (long timeouts) and consistency (short timeouts).

Innovations. The major innovations involve ease of installation and use: Ufo runs as a user-level process, and it requires no changes to the underlying operating system, nor does it require changes or additions to existing libraries.

Performance. When using the Catcher and Ufo, individual *open*, *close*, *stat* and *getpid* system calls are very expensive. For IO-intensive applications, this is a crippling slowdown. However, for most applications, the overhead becomes reasonable when amortized over the life of the application, within a range of 5-29%. In addition, because Ufo can be applied only to those processes which need it, the overhead is not applied globally to all processes. Furthermore, the performance hit affects only the user who runs an application under Ufo, and is not incurred by other users of the system.

Scalability. It is unclear to what extent the notion of scalability applies to this system, since the model is *a user at a single machine accessing any number of machines worldwide via a variety of protocols*. If the only criteria for scalability is that the number of machines accessed should be able to grow without affecting performance, Ufo would seem to have achieved it.

7. Hierarchy and Content (HAC)

Design Principles and Goals. HAC is a new file system intended to provide convenient access to vast amounts of information. The benefit of this file system is that it allows the user both to find the right information, and to transfer the needed data quickly. The main goal is convenient and intuitive integration of information. The HAC file system combines the traditional name-based hierarchical file system and a content-based semantic file system. While the full power of hierarchical systems is preserved, additional control-based access (CBA) can be maintained and controlled by the user.

Architecture and Implementation. The authors implemented HAC on top of a UNIX file system (SunOS) using Glimpse as the default CBA mechanism. HAC interacts with UNIX and Glimpse using APIs. Both syntactic and semantic directories coexist in the same file system. This allows users to define their own personal name spaces, and intercept all file system calls that access the directory or its contents, providing a transparent interface to all applications. The commands provided by HAC to manipulate queries and semantic directories are very intuitive.

In HAC, every query and its corresponding semantic directory have a scope, which is the set of files over which the query is evaluated. HAC creates symbolic links to all the files that satisfy the query. HAC allows users to edit the results of queries, and tune these results as needed using semantic directories. A data-inconsistency problem arises when the set of transient symbolic links does not match the current result of evaluating the associated query. HAC handles this problem by invoking the CBA mechanism to re-index the file system periodically. HAC also allows users to initiate re-indexing at any time, and for any part of the file system.

Advantages. HAC does not require a kernel modification. It was implemented as a dynamically linked library, which makes it easier to use and port. The handling of the data-inconsistency problem described above is a major contribution of this file system.

Problems and Disadvantages. There is a performance penalty due to the large overhead of the HAC file system's library calls.

Innovations. HAC file systems accomplish both name-based and content-based access by using a radically different approach: it starts with a hierarchical naming system, adding the features of a content-based access (CBA) later. This gives users a lot of flexibility and power, and it makes the system easy and intuitive to use.

Performance. Two experiments were run to compare the performance of HAC to that of regular file systems. In the first experiment, the HAC was used the same way as UNIX. The result shows that HAC incurs a great deal of overhead when creating new directories and coping files, and somewhat less overhead when scanning and reading. Overall, HAC is about 46% slower than UNIX. The second experiment measured the speed of indexing and searching a database. The result of indexing shows HAC has a 27% time overhead, and 15% space overhead. The search result shows the overhead of creating a semantic directory in HAC decreases when a greater number of files match the search query. The authors believe that the overhead is reasonable, considering that HAC creates and maintains additional data structures that provide content-based access to the files. The extra disk space required is N/8 bytes per semantic directory, where N is the number of indexed files.

Scalability. The implementation described in this article is suited for personal file systems, and is not scalable to very large file systems.

8. JetFile System

Design Principles and Goals. JetFile targets the demands of personal computing. JetFile is a multicast-based distributed file system designed to efficiently handle the daily tasks of a regular user and to become an alternative to the local file system. The goal is realized by decentralizing most of the traditional file server responsibilities, turning every client into a file access server.

Architecture and Implementation. JetFile contains four distinct components - File Manager, Versioning Server, Storage Server, and Key Server. The prototype was implemented for HP 735/99 model running HP-UX 9.05, connected to 10Mb/s Ethernet, and includes only the file manager and versioning server. The communication among these components relies on the Scalable Reliable Multicast (SRM) paradigm. The SRM is logically layered above IP multicast, and guarantees the eventual delivery of all data to all receivers. This minimal reliability is achieved by having each receiver responsible for detecting lost data and initiating repairs. The lost data is detected through the comparison of version numbers.

The file manager consists of a kernel module, and a user module. The kernel module implements a new file system called YFS (Yet another File System). The YFS redirects reads and writes to this local file with almost no overhead. The user module is responsible for keeping track of the current version and state of locally created and cached files. The file name space in JetFile is divided into volumes. The versioning server keeps track of the current version of all files in a volume, and replies to requests for new version numbers.

The storage server is responsible for the long-term storage of files and backup functions. The key server stores and distributes cryptographic keys used for signing and encrypting file contents. Neither of these two components is implemented in the prototype.

Advantages. Using multicast communication instead of unicast communication has the advantage of decentralizing server responsibilities. Since clients also serve as file servers, there is no need to write the file data through the file cache, and over the network to the server. It offloads both servers and network when a file is not actively shared.

The optimistic approach to file updates hides the effects of transmission delays and errors. It is a key factor in JetFile's ability to work well over both long high-speed networks and high delay/ high loss wireless networks.

Problems and Disadvantages. Because JetFile takes a best-effort approach to maintain cache coherency, there is no guarantee that a call-back reaches all destinations. Clients may not be aware of the cache invalidity for up to 30 seconds. JetFile requires file managers to join a multicast group for each file they actively use or server. This implies that routers will be forced to manage a large multicast routing state.

Innovations. JetFile brings networking concepts such as IP multicast routing and Scalable Reliable Multicast (SRM) into the realm of the distributed file system. This technique keeps communication to a minimum and contributes to a scalable distributed file system across the Internet or large intranets.

Performance. Performance tests were conducted using the Andrew Benchmark. The comparison between the local HP-UX Unix File System (UFS) and the JetFile, running with hot caches, shows their performances are very similar in most phases. JetFile performs slightly better in certain phases because of the benefit of its asynchronous inode allocation. In summary, the tests prove that the performance of JetFile with a warm cache is comparable to a performance of a local file system.

Scalability. The use of scalable reliable multicast minimizes communication and decentralizes server operations; by decreasing reliance on a single server, this dramatically increases the scalability of the JetFile system.

Summary

The papers considered here show a broad range of approaches and adopt a variety of tradeoffs. One such tradeoff is between specialization and optimization on the one hand, and portability and flexibility on the other. For example, Ufo implements transparent access to remote files using an easily-portable user-level layer, at the cost of a major decrease in performance; at the other end of the extreme, Vesta is optimized to perform high-speed parallel computing, but requires specialized programs and writes to disk in a manner completely incompatible with other file systems, while HFS optimizes the file system for I/O-intensive parallel applications with little processing overhead.

Since the presented file systems have different intended uses, and their design goals do not match exactly, the file systems exhibit different properties. The overlap in the properties is summarized in Table 1. Most of the file systems employ layering as a valuable technique for making a design modular, and thus easier to implement. Most systems scale well with a number of clients or servers, making scalability their primary focus.

The papers present a variety of methods to ensure ease of recovery, configuration and administration. Ease of administration comes mainly from systems that use user-level extensions to the existing OS. In terms of hardware, distributed server facilities provide data replication on many file systems, and on some guarantee that the whole system can run even after individual components fail.

Overall, less attention was paid to the issues of portability and security. For parallel file systems, security is not a major concern due to the specialized nature of these machines. The systems that use a preexisting underlying kernel or file system generally just employ whatever security is already in place; this is possible because these designs did not opt for modifications to the kernel or system libraries.

Scalability and performance are the main driving forces behind all these designs. In fact, only Ufo and HAC are not scalable in the usual sense of the term. In many systems, scalability is the most important aspect of performance. Unfortunately, there is no single performance measurement that can be used for comparisons across all of these systems. In some, the relevant criteria is how fast the system can recover from failure while providing access to the servers; in others it is disk access bandwidth; and in yet others it is an ability of a client to take over for a failed server while the server recovers from crash.

All systems either use a completely new approach to a particular part of the design or, when built on top of existing systems, use them in a novel way, not intended by the original designers. We believe that these papers present an interesting view of the current research, and a good basis for further file system development.

Paper	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**
Target	PD,E	C	PA	PA	PD,E	PD,E	PC	PC
Layering	√		√	√	√	√		
Security	√			√		√		√
Scalability	√	√	√	√	√			√
Data replication	√	√		√	√			√
Easy recovery	√	√		√	√			√
Easy reconfiguration	√	√		√	√	√	√	√
Easy administration	some	√		√	√	√	√	√
Portability				some		√	√	some

Table 1: File systems properties. Notations: PA - parallel applications, PD - program development, E - engineering, C - commercial use, PC - personal computing.

References

1. Thomas E. Anderson, Michael D. Dahlin, Jeanna M. Neefe, David A. Patterson, Drew S. Roselli, and Randolph Y. Wang. Serverless network file systems. *ACM Transactions on Computer Systems*, 14(1):41-79, February 1996.

2. Murthy Devarakonda, Bill Kish, and Ajay Mohindra. Recovery in the Calypso file system. *ACM Transactions on Computer Systems*, 14(3):287-310, August 1996.

3. Peter F. Corbett, and Dror G. Feitelson. The Vesta parallel file system. *ACM Transactions on Computer Systems*, 14(3):225-264, August 1996.

4. Orran Krieger, and Michael Stumm. HFS: a performance-oriented flexible file system based on building-block compositions. *ACM Transactions on Computer Systems*, 15(3):286-321, August 1997.

5. Chandramohan A. Thekkath, Timothy Mann, and Edward K. Lee. Frangipani: a scalable distributed file system. In *Proceedings of the Sixteenth Symposium on Operating System Principles*, October 1997, 224-237.

6. Albert D. Alexandrov, Maximilian Ibel, Klaus E. Schauser, and Chris J. Scheiman. Ufo: a personal global file system based on user-level extensions to the operating system. *ACM Transactions on Computer System*s, 16(3):207-233, August 1998.

7. Burra Gopal, and Udi Manber. Integrating content-based access mechanisms with hierarchical file systems. In *Proceedings of the Third symposium on Operating Systems Design and Implementation*, February 1999, New Orleans, LA USA, 265-278.

8. Björn Grönvall, Assar Westerlund, and Stephen Pink. The design of a multicast-based distributed file system. In *Proceedings of the Third Symposium on Operating Systems Design and Implementation*, February 1999, New Orleans, LA USA, 251-264.