

An Overview of File System Architectures

Florin Isaila

Department of Computer Science
University of Karlsruhe
florin@ipd.uni-karlsruhe.de

1 INTRODUCTION

The ever increasing gap between processor and memory speeds on one side and disk systems on the other side has exposed the I/O subsystems as a bottleneck for the applications with intensive I/O requirements. Consequently, file systems, as low-level managers of storage resources, have to offer flexible and efficient services in order to allow a high utilization of disks.

A storage device is typically seen by users as a contiguous linear space that can be accessed in blocks of fixed length. It is obvious that this simple uniform interface can not address the requirements of complex multi-user, multi-process or distributed environments. Therefore, *file systems* have been created as a layer of software that implements a more sophisticated disk management. The most important tasks of file systems are:

- Organize the disk in linear, non-overlapping *files*.
- Manage the pool of free disk blocks.
- Allow users to construct a logical name space.
- Move data efficiently between disk and memory.
- Coordinate access of multiple processes to the same file.
- Give the users mechanisms to protect their files from other users, as for instance access rights or capabilities.
- Offer recovery mechanism for the case the file system becomes inconsistent.
- Cache frequently used data.
- Prefetch data predicted to be used in the near future.

In this chapter we investigate how different types of file systems address these issues. The next section describes the access patterns of sequential and parallel applications, as reported by several research groups. The access patterns are interesting, because they are often used to motivate file system design choices. Section 3 details some tasks of file systems, as enumerated above. We continue by explaining general issues of distributed file systems in the first part of Section 4. In the second part of Section 4, we address parallel (4.5), shared (4.6) and grid file systems (4.7) and show how file systems may handle operations for mobile computers(4.8). We summarize in Section 5.

2 FILE ACCESS PATTERNS

Applications have various needs for accessing files. Therefore, in order to design a efficient file system it is very important to understand the file *access patterns*. A large number of studies have been devoted to this goal. In this section we will summarize some of their conclusions.

2.1 File access patterns of sequential applications

Several studies [41,3,34] analyzed the file access patterns of applications running on uniprocessor machines and accessing files belonging to either a local or a distributed file system. Their results were used either for implementing an efficient local or distributed file systems.

- Most files are small, under 10K [41,3]. Short files are used for directories, symbolic links, command files, temporary files.
- Files are open for a short period of time. 75% of all file accesses are open less than 0.5 seconds and 90% less than 10 seconds [41,3,34].
- Life time of files is short. A distributed file system study [3] measured that between 65% and 80% of all files lived less than 30 seconds. This has an important impact on the caching policy. For instance, short-lived files may eventually not be sent to disk at all, avoiding unnecessary disk accesses.
- Most files are accessed sequentially [41,34,3]. This suggests that a sequential pre-fetching policy may be beneficial for the file system performance.
- Reading is much more common than writing. Therefore, caching can bring a substantial performance boost.
- File sharing is unusual. Especially the write sharing occurs infrequently [3]. This justifies the choice for a relaxed consistency protocols.
- File access is bursty [3]. Periods of intense file system utilization alternate with periods of inactivity.

2.2 Parallel I/O access characterization

Several studies of parallel I/O access patterns are available [31,45,44,11]. Some of them focus on parallel scientific applications that are typically multiprocess applications, which typically access a huge amount of data.

Some of their results are summarized below, illustrated by the parallel access example from Figure 1. The figure shows two different way of physical partitioning of a two-dimensional 4x4 matrix, over two disks attached two different I/O nodes: (a) by striping the columns and (b) by striping the rows. The matrix is logically partitioned between four compute nodes, each process accessing a matrix row. For instance , this kind of access can be used by a matrix multiplication algorithm.

- Unlike in the uniprocessor applications, file sharing between several compute nodes is frequent. However, concurrent sharing between parallel applications

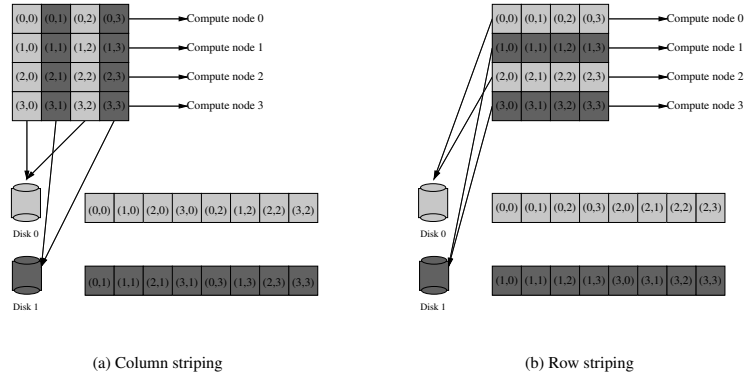


Fig. 1. Parallel file access example

of data within the file is rare [31,45]. In the example, it is obvious that the file is shared among the four compute nodes. The accesses of the individual processes do not overlap, and are therefore, non-concurrent.

- The files are striped over several disks in order to increase the access throughput. In Figure 1(a) the matrix columns 0 and 2 reside on disk 0 and columns 1 and 3 on disk 1.
- Individual compute nodes often access the file non-contiguously. For instance, writing the first matrix line by compute node 0 from Figure 1(a) results in two non-contiguous disk accesses: (0,0) and (0,2) at disk 0, and (0,1) and (0,3) at disk 1. Non-contiguous accesses cause costly disk head seek movements, and therefore, have a negative influence on performance. However, with the different physical partitioning from Figure 1(b), the contiguous access of each process translates into contiguous disk access.
- The compute nodes frequently access a file in interleaved access patterns. This may result in high global inter-process spacial locality of data at I/O nodes [31], but also in a poor intra-process spatial locality [44]. In our example from 1(a), the accesses of the four compute nodes translate into interleaved accesses at disk 0 and 1. If they occur approximatively at the same time, the global disk access pattern is sequential, and therefore optimal. Otherwise, unnecessary seek times may drastically affect the application performance. This problem was addressed by different collective I/O operations designs and implementations [15,23]. For the disk layout from Figure 1(b), the disk accesses of individual processes are not interleaved.
- In MIMD systems there is a high number of small I/O requests [31]. To a large extent this is the result of the non-contiguous and interleaved accesses, as described earlier. In Figure 1(a), the process 0 access results in sending four small requests, two to each disk. However, the layout from 1(b) permits the row access with a single request.
- Parallel I/O is bursty, periods of intensive I/O activity alternate with computation [45]. It has been shown that the intensive I/O periods benefit from

a collective approach rather than from one that treats the parallel requests independently [23,15].

- Parallel applications accesses cause sometimes an unbalanced use of disks. For example, suppose that the four compute nodes in the Figure 1 show a bursty access by repeatedly executing the following two steps: reading one element from the file in the first step and performing some computation in the second step. In case (a), the two disks are used alternatively, and therefore, the parallel access of the compute nodes is serialized at the disks, with a direct impact on the speedup, according to Amdahl's law. Nevertheless, in the second case, the maximum disk parallelism degree may be achieved for each access, both disks being equally loaded.
- Some applications cause large contention of compute nodes' requests at the disks. In the extreme case, each compute node has to send requests to all disks, due to an unfortunate file data distribution. This may have a negative impact on both computing and I/O scalability: incrementing the number of compute nodes, may increase the number of requests of a collective access by the number of disks; similarly, incrementing the number of disks, may increase the number of requests by the number of compute nodes. In our example from Figure 1(a), in order to read the matrix, each of the four compute nodes has to access both disks. However, for the physical partitioning from Figure 1(b), each compute node has to send requests to a single disk.
- Parallel applications frequently use simple and nested strided access patterns. This denotes the use of multidimensional arrays, partitioned across the compute nodes [31].

3 FILE SYSTEM DUTIES

The storage space has to be managed in a way that responds to the needs of different applications. In an operating system this is usually the role of the file system. This section outlines the most important duties of a file system.

File abstraction. A disk is used to store data belonging to different users or processes and having various contents. These justify the separation of the linear physical space of the disk into several logical units, called files. A file is a logical linear-addressable sequence of bytes that is mapped onto physical disk blocks.

A separate part of the disk is typically reserved for system information, called *metadata*. Each file is associated with a metadata structure. This structure contains the file attributes like file creation, modification and access times, owner, rights, mapping information, etc. In traditional UNIX system this structure is called *i-node*.

Mapping a file on the disk. There are various data structures used for mapping logical file offsets to physical disk addresses. The MS-DOS file system uses a linked list containing a direct logical to physical mapping. The structure is not scalable for random accesses because a linear search is used. In the traditional UNIX file systems, the logical blocks are multiples of disk blocks. The mapping structure consists of a tree whose root has thirteen children: the first

ten contain direct pointers to disk blocks and the next three simple-, double- and triple-indirect pointers [2]. The disadvantage of this scheme is that large file access may result in several disk accesses only for mapping a file block.

Recently developed file systems like JFS [20], ReiserFS [29] and XFS [43] use more efficient schemes. First, instead of mapping blocks they map contiguous sequences of blocks called *extents*. Second, the extents are indexed by their file offset and organized in B trees. The mapping consists of a tree descent having the goal of locating the disk address corresponding to a given file offset. For efficiency reasons, the first extents of a file are not kept in the B tree, but are directly mapped onto disks. Therefore, small files can quickly find the disk addresses without a tree lookup. There are two more possible small file optimizations, which reduce the number of disk accesses. First, the metadata and data of small files may be put on the same disk block. Second, several small files may be placed on the same disk block.

Name space. Another important role of a file system is to provide a *naming* mechanism that allows users to differentiate among the set of files. Each file may be assigned a string of characters as a name. The names are organized into a tree structure by means of special files called *directories*. Directories represent the inner nodes of the tree and act as repositories for other directories or files. The files containing actual user data are the leaf nodes. The path from the root to the file's node uniquely identifies the file in the name space.

The UNIX file system stores the directory content into a linked list, linearly stored on the disk. Therefore, a linear scan has to be used for file name lookup. This affects the scalability for directories containing a large number of files. Recent file systems use B trees for providing efficient lookup. XFS and JFS use a B tree for each directory, whereas ReiserFS keeps a B tree for the whole name space.

Disk space management. Files are normally stored on disks, therefore *disk space management* is an important part of a file system. Usually the disk space is managed by structures maintaining information about the free blocks. Traditional UNIX file systems keep a bitmap structure stored on the disk. A bit has the value "0" if the corresponding block is free and "1" otherwise. The main disadvantage of these scheme is that the bitmap size increases linearly with the size of the disk. These may cause a significant performance penalty for large disks, due to large times devoted to linearly scanning the bitmap for a free block.

Newer file systems use mainly two techniques for making disk space management more efficient: extents and B trees structures. The main advantage of extents is that they allow finding several free blocks at a time. The space overhead for storing extents is lower than for bitmaps, as long as the file system is not very fragmented. B trees allow a quicker lookup for a free block than lists, which have to be sequentially scanned. Extents and B trees can be used in conjunction. Indexing by extent size makes possible to quickly allocate chunks of contiguous blocks of a certain size. Indexing by extent position allows to quickly locate blocks by their addresses. XFS is an example of a file system that employs all these techniques.

Caching. When performing a read from a file, the data is brought block-wise from the disk into the memory. A later file access may find parts of the data in memory. This technique is called *caching*. We will refer to the memory region used for this purpose in a file system as *file cache*. Caching improves the performance of the applications which exhibit temporal locality of access, i.e. in a program, once a block has been accessed, it is highly probable that it will be accessed again in the near future. Performance measurements show that this is the case with most applications. In the local file systems, caching is used to improve local disk access times, providing copies of the low-speed disks in the faster memory.

Prefetching. *Prefetching* is the technique of reading ahead from disk into cache data blocks probable to be accessed in the near future. The prefetching is motivated by several factors:

- the predictability of file access patterns of some applications
- the availability of file access hints
- the poor utilization of disk parallelism
- the large latencies of small non-sequential disk requests

On the other hand, prefetching may hurt performance by causing the eviction of valuable disk blocks from the cache. In this respect, it is very important to take prefetching decisions at the right time.

Prefetching can be done manually or automatically. The programmers can manually insert prefetching hints into their programs. This approach supposes that the source code is available. Additionally, this involves a high cost of program understanding and modification.

There are several automatic prefetching approaches.

- **Sequential read-ahead.** This is the simplest type of prefetching. It is based on the file access studies that have found out that the most frequent type of access is sequential. This type of automatic prefetching is implemented by the most file systems.
- **History-based prefetching.** The file system keeps information about past file accesses and tries to predict the future access pattern [24,12,17,25,26].
- **Statical analysis.** In this approach a compiler analyzes the program and inserts prefetching hints into the code [28,9,46,35].
- **Hint-based.** The applications should generate informing hints that disclose their future accesses allowing the file system to make optimal global prefetch decisions [37,5].
- **Speculative Execution.** The idle processor times due to blocking I/O are used for speculative pre-execution that generates prefetching hints [6].

Failure recovery.failure recovery An important duty of file systems is to recover from failure. Traditional file systems used a recovery mechanism that searches the whole disk in order to correct corrupted blocks. As the storage devices are increasing in capacity this approach may let a file system unavailable for a significant time.

The *log-structured file systems* (LFS) [38] uses a combination between checkpointing and roll-forward in order to allow quick recovery. *Checkpointing* is the technique of saving the state of a system on a permanent storage device. The checkpointing may be done on a regular basis, when certain conditions are met, by request. The saved state can be subsequently used for a full recovery of the system at the last checkpoint. For more recovery accuracy a roll-forward technique may be used. *Roll-forward* consists of replaying all the changes recorded on the disk after the last checkpoint, such that the consistency of the system is preserved. LFS uses both recovery techniques. The main idea of LFS is to gather all the data and metadata updates into a memory segment called log and to store it to disk in a single operation when it becomes full or by need.

The disk in LFS is seen as a circular buffer of logs. When a file block is modified, its previous disk position is invalidated and the block is written anew on the log. This approach causes holes within the logs residing on the disks. Therefore, a garbage collector is used for rearranging the blocks on the disk in order to create new free segments for the log. The garbage collector becomes very complex, because it has to deal with both metadata and data scattered on the disk that has to be rearranged. In the *journalled file systems* [20,29,43], the design is simplified. The log is renamed journal and it is used only for metadata updates. Data updates have to be done manually or by another external checkpointing module. The advantage of these scheme resides in a much cleaner design by eliminating the need for garbage collection.

4 DISTRIBUTED FILE SYSTEMS

The main goal of *distributed file systems* is to make a collection of independent storage resources appear as a single system. Distributed file systems have different characteristics depending on the resources they are built on or on their utilization goals. The following main factors influence the architecture and terminology of distributed file systems:

Storage attachment. The storage may be computer-attached or network-attached. In the computer-attached case, a single computer has exclusive access to the storage and acts as a server that serves disk requests, either directly from the disk or through its local file system. A distributed file system using network-attached devices is called *shared file system* [32], as described in Subsection 4.6.

Network connectivity. The connectivity plays an important role for the type of services a distributed system may offer. There are file systems for tightly-connected networks as those for supercomputers like IBM SP's or cluster of computers using high-performance networking technologies. In this case the file systems can be used by scientific applications that process a huge amount of data and are typically parallel. They need a high throughput of disk systems and a tight interconnection of their parallel running processes.

Distributed systems may also span several different administrative domains. In this case the network connectivity varies due to different reasons as resource heterogeneity or traffic variations. Security is also an important issue that has

to be dealt with. These tasks are implemented by *grid file systems*, as we show in Subsection 4.7.

The ever increasing usage of mobile computers has brought the need to address changing network characteristics ranging from high connectivity to no connectivity. Subsection 4.8 explains specific issues related to file systems for mobile computers.

Parallel access. A distributed file system does not necessarily offer parallel file access. One such example is NFS. NFS servers serialize all the accesses from clients, even though they refer to different files or different parts of the same file. However, as seen in the access pattern section, parallel application performance may suffer drastically from such a limitation. This is one of the main reasons why *parallel file systems* decluster a file over different disks managed by independent servers. Some systems also distribute metadata in order to allow parallel access to different files or different parts of the directory tree. More details can be found in Subsection 4.5.

4.1 Location transparency and location independence

A distributed file system typically presents the user a tree-like name space that contains directories and files distributed over several machines that can be accessed by a path. A file's location is *transparent* when the user cannot tell, just by looking at its path, if a file is stored locally or remotely. NFS [40] implements location transparency. A file location is *independent* if the files are accessed with the same path from all the machines using the distributed file system. It can be noticed that location independence entails location transparency, but not vice-versa. AFS [18] files are location-independent.

4.2 Distributed file system architectures

The traditional distributed file system architecture was based on the *client-server* paradigm [40,18,4]. A file server manages a pool of storage resources and offers a file service to remote or local clients. A typical file server has the following tasks: stores file data on its local disks, manages metadata, caches metadata and data in order to quickly satisfy client requests, and eventually manages data consistency by keeping track of clients that cache blocks and updating or invalidating stale data.

Examples of file systems using a client-server architecture are NFS [40] and AFS [18]. A NFS server exports a set of directories of a local file system to remote authorized client machines. Each client can mount each directory at a specific point in its name tree. Thereafter, the remote file system can be accessed as if it is local. The mount point is automatically detected at file name parsing. AFS's Vice is a collection of file servers each of which stores a part of the system tree. The client machines run processes called Venus that cooperate with Vice for providing a single location-independent name space and shared file access.

A centralized file server is a performance and reliability bottleneck. As an alternative, a *server-less* architecture has been proposed [1]. The file system

consists of several cooperating components. There is no central bottleneck in the system. The data and metadata are distributed over these components, can be accessed from everywhere in the system and can be dynamically migrated during operation. This gives also the opportunity of providing available services by transferring failed component tasks to remaining machines.

4.3 Scalability

Even though a distributed file system manages several resources in a network, it does not necessary exploit the potential for parallelism. We have seen that in a typical client-server architecture [40,18] the data and metadata of a particular file are stored at a single server. Therefore, if a file system is accessed in parallel, even if there are several disks in the network, only the server disks are utilized. Additionally, when the number of clients increases, the file server becomes a bottleneck for both data and metadata requests.

Data *scalability* can be achieved by file replication or distribution. *File replication* allows the distribution of requests for the whole file over distinct disks. For instance, it is suitable for read-only access in a Web file system, in order to avoid hot spots [39]. The main drawback of replication is the cost of preserving consistency among copies. A file may also be *declustered* or striped over several disks by using for instance a software RAID technique [36]. The primary advantage of this approach is the high throughput due to disk performance aggregation.

4.4 Caching

Distributed file system caches have two main roles: the traditional role of caching blocks of local disks and providing local copies of remote resources (remote disks or remote file caches). In our discussion, we will use the attribute *local* for a resource (i.e. cache) or entity (i.e. page cache) that is accessible only on a single machine and *global* for a resource that is accessible by all machines in a network.

Cooperative caches. In order to make a comparison between caching in client-server and server-less architectures we consider a hybrid file caching model in a distributed file system. In this model "server" represents a file server for the client-server architecture and a storage manager, in the sense of disk block repository, for the server-less architecture. "Client" is also used in a broader sense meaning the counterpart of a server in the client-server architecture, and a user of the file system in the server-less architecture.

In this hybrid model we identify six broad file caching levels of the disk blocks, from the perspective of a client:

1. client local memory
2. server memory
3. other clients' memory
4. server disk

5. client disk
6. other clients' disks

In the client-server design only four levels are used: 1,2,4,5. For instance, in order to access file data, NFS looks up the levels 1, 2 and 4 in this order and AFS 1, 5, 2 and 4. The client has no way of detecting if the data is located in the cache of some other client. This could represent a significant performance penalty if the machines are inter-connected by a high-performance network. Under the circumstances of the actual technologies, remote memory access can be two to three orders of magnitude quicker than disk access. Therefore, cached blocks in other clients' memory could be fetched more efficiently.

Coordinating the file caches of many machines distributed on a LAN in order to provide a more effective global cache is called *cooperative caching*. This mechanism is very suitable to the cooperative nature of the server-less architecture. Dahlin and al. [13] describe several cooperative caching algorithms and results of their simulations.

There are three main aspects one has to take into consideration when designing a cooperative caching algorithm. First, cooperative caching implies that a node is able to look up for a block not only in the local cache but also in remote caches of other nodes. Therefore, a cooperative caching algorithm has to contain both *a local and a global lookup policy*. Second, when a block has to be fetched into a full file cache, the node has to choose another block for eviction. The eviction may be directed either to the local disk or to the remote cache/disk of another node. Therefore, an algorithm has to specify both *a local and a global block replacement policy*. Finally, when several nodes cache copies of the same block, an algorithm has to describe *a consistency protocol*. The following algorithms are designed to improve cache performance for file system reads. Therefore, they do not address consistency problems.

Direct client cooperation. At file cache overflow, a client uses the file caches of remote clients as an extension of its own cache. The disadvantage is that a remote client is not aware of the blocks cached on behalf of some other client. Therefore, he can request anew a block he already caches resulting in double caching. The lookup is simple, each client keeps information about the location of its blocks. No replacement policy is specified.

Greedy forwarding. Greedy forwarding considers all the caches in the system as a global resource, but it does not attempt to coordinate the contents of these caches. Each client looks up the levels 1,2,3,4 in order to find a file block. If the client does not cache the block it contacts the server. If the server caches the block, it returns it. Otherwise, the server consults a structure listing the clients that are caching the block. If it finds a client caching the block it instructs him to send the block to the original requester. The server sends a disk requests only in the case the block is not cached at all. The algorithm is greedy, because there is no global policy, each client managing its own local file cache, for instance by using a local "least recently used" (LRU) policy, for block replacement. This can result in unnecessary data duplication on different clients. Additionally, it

can be noticed that the server is always contacted in case of a miss and this can cause a substantial overhead for a high system load.

Centrally coordinated caching. Centrally coordinated caching adds coordination to the greedy forwarding algorithm. Besides the local file caches, there is a global cache distributed over clients and coordinated by the server. The fraction of memory each client dedicates to local and global cache is statically established. The client looks up the levels 1,2,3,4 in order to find a file block, in the same way as greedy forwarding does. Unlike greedy forwarding, centrally coordinated caching has a global replacement policy. The server keeps lists with the clients caching each block and evicts always the least recently used blocks from the global cache. The main advantage of centrally coordinated caching is the high global hit rate it can achieve due to the central coordinated replacement policy. On the other side it decreases the data locality if the fraction each client manages greedily is small.

N-Chance forwarding. This algorithm is different from greedy forwarding in two respects. First, each client adjusts dynamically the cache fraction it manages greedily based on activity. For instance, it makes sense that an idle client dedicates its whole cache to the global coordinated cache. Second, the algorithm considers a disk block that is cached at only one client as very important and it tries to postpone its replacement. Such a block is called a *singlet*. Before replacing a singlet, the algorithm gives it n chances to survive. A recirculation count is associated with each block and is assigned to n at the time the replacer finds out, by asking the server, that the block is a singlet. Whenever a singlet is chosen for replacement, the recirculation count is decremented, and, if it is not zero, the block is sent randomly to another client and the server is informed about the new block location. The client receiving the block places the block at the end of its local LRU queue, as if it has been recently referenced. If the recirculation count becomes zero, the block is evicted. N-Chance forwarding degenerates into greedy forwarding when $n = 0$. There are two main advantages of N-Chance forwarding. First, it provides a simple trade-off between global and local caches. Second, favoring singlets provides a better performance, because evicting a single is much expensive as evicting a duplicate, because a duplicate can be later found in other client's cache. The N-Chance forwarding algorithm is employed by xFS distributed file system.

Hash-distributed caching. Hash-distributed caching differs from centrally coordinated caching in that each block is assigned to a client cache by hashing its address. Therefore, a client that does not find a block in its cache is able to contact directly the potential holder, identified by hashing the block address, and only in miss case the server (lookup order: 1,3,2,4). The replacement policy is the same as in the case of centrally coordinated caching. This algorithm reduces significantly the server load, because each client is able to bypass the server in the first lookup phase.

Weighted LRU. The algorithm computes a global weight for each page and it replaces the page with the lowest value/cost ratio. For instance, a singlet is more valuable than a block cached in multiple caches. The opportunity cost of

keeping an object in memory is the cache space it consumes until the next time the block is referenced.

Semantics of file sharing. Using caching comes at the cost of providing consistency for replicated file data. Data replication in several caches is normally the direct consequence of file sharing among several processes. A consistency protocol is needed when *at least* one of the sharing processes writes the file. The distributed file systems typically guarantee a *semantics of file sharing*.

The most popular model is UNIX semantics. If a process writes to a file a subsequent read of any process must see that modification. It is easy to implement in the one-machine systems, because they usually have a centralized file system cache which is shared between processes. In a distributed file system, caches located on different machines can contain the copy of the same file block. According to UNIX semantics, if one machine writes to its copy, a subsequent read of any other machine must see the modification, even if it occurred a very short time ago. Possible solutions are invalidation or update protocols. For instance, xFS uses a token-based invalidation protocol. Before writing to its locally cached block copy, a process has to contact the block manager, that invalidates all other cached copies before sending back the token. Update or invalidation protocols may incur a considerable overhead. Alternatively the need for a consistency protocol can be eliminated by considering all caches in the distributed system as a single large cache and not allowing replication [10]. However, the drawback of this approach is that it would reduce access locality.

In order to reduce the overhead of a UNIX semantics implementation, relaxed semantics have been proposed. In the *session semantics*, guaranteed by AFS, all the modifications made by a process to a file after opening it, will be made visible to the other processes only after the process closes the file.

Transaction semantics guarantees that a transaction is executed atomically and all transactions are sequentialized in an arbitrary manner. The operations not belonging to a transaction may execute in any order.

NFS semantics guarantees that all the modification of a client will become visible for other clients in 3 seconds for data and 30 seconds for metadata. This semantics is based on the observation of access patterns studies that file sharing for writing is rare.

4.5 Parallel file systems

Files are rarely shared by uni-process processor applications as we have shown in subsection 2.1. Distributed file systems were often designed starting from these premise. For instance, file data and metadata in NFS are stored at a single server. Therefore, all parallel accesses to a certain file are serialized. However, in subsection 2.2, we have seen that the parallel applications typically access files in parallel. The distributed file systems have to be specialized in order to allow efficient parallel access to both data and metadata. In this case they are called parallel file systems.

File physical distribution In order to allow true parallel access the files have to be physically stored on several disks. The nodes of supercomputers are typically split into *compute nodes* running the application and *I/O nodes* equipped with disks that store the files. It is also possible to use a node for I/O and computation (part-time I/O). The parallel file systems for clusters have also adopted this design [21,22,42].

The files are typically declustered over several I/O nodes or disks by simple striping [42,21] or by more advanced declustering techniques [14,19,7,30,22]. As the characterization studies have shown, declustering has an important impact on performance. For instance, an unfortunate file declustering may turn parallel logical file access into sequential disk access as the example from subsection 2.2 has shown.

For instance, files in GPFS [42] and PVFS [21] are split into equally-sized blocks and the blocks are striped in a round-robin manner over the I/O nodes. This simplifies the data structure used for file physical distribution, but it can affect the performance of a parallel application due to a poor match between access patterns and data placement, as we have shown in subsection 2.2.

Other parallel file systems subdivide files into several subfiles. The file is still a linearly addressable sequence of bytes. The subfiles can be accessed in parallel as long as they are stored on independent devices. The user can either rely on a default file placement or control the file declustering. The fact that the user is aware of the potential physical parallelism helps him in choosing a right file placement for a given access pattern.

In the Vesta Parallel File System [7,8], a data set can be partitioned into two-dimensional rectangular arrays. The nCube parallel I/O system [14] builds mapping functions between a file and disks using address bit permutations. The major deficiency of this approach is that all sizes must be powers of two. A file in Clusterfile parallel file system [22] can be arbitrarily partitioned into subfiles.

Logical views Some parallel file systems allow applications to logically partition a file among several processors by setting views on it. A *view* is a portion of a file that appears to have linear addresses. It can thus be stored or loaded in one logical transfer. A view is similar to a subfile, except that the file is not physically stored in that way. When an application opens a file it has by default a view on the whole file. Subsequently, it might change the view according to its own needs.

Views are used by parallel file systems (Vesta [7,8], PVFS [21], Clusterfile [22]) and by libraries like MPI-IO [27]. MPI-IO allows setting arbitrary views by using MPI datatypes. With Vesta, the applications may set views only in two dimensional rectangular patterns, which represents obviously a limitation. Multidimensional views on a file may be defined in PVFS. Like MPI-IO, Clusterfile allows for setting arbitrary views. An important advantage of using views is that it relieves the programmer from complex index computation. Once the view is set the application has a logical sequential view of the set of data it needs and can access it in the same manner it accesses an ordinary file.

Setting a view gives the opportunity of early computation of mappings between the logical and physical partitioning of the file. The mappings are then used at read/write operations for gathering/scattering the data into/from messages. The advantage of this approach is that the overhead of computing access indices is paid just once at view setting.

Views can also be seen as hints to the operating system. They actually disclose potential future access patterns and can be used by I/O scheduling, caching and pre-fetching policies. For example, these hints can help in ordering disk requests, laying out of file blocks on the disks, finding an optimal size of network messages, choosing replacement policies of the buffer caches, etc.

File pointers Another typical task of file systems is file pointer management. Unix file system assigns a file pointer to each open file. The pointers of the same file are always manipulated independently. As the parallel I/O characterization studies found out, the parallel applications often share files. If the file access is independent, Unix-style independent pointers can be used.

However, the parallel processes of an applications need sometime to cooperate in accessing a file. A shared file pointer may help implementing coordinated access. For instance, shared pointers can be used by a self-scheduling policy. Each process of a parallel application needs to access a portion of a file and then let the others know about this access through the shared pointer. A shared pointer is not proper for non-coordinated access, because it represents a centralized entity and therefore hinders parallelism when independent access is needed.

Collective I/O operations *Collective I/O* is a technique that merges several small I/O requests of several compute nodes. It is based on two well known facts. First, the overhead of sending small messages over the network is high. Second, non-contiguous disk access is inefficient. The collective I/O targets the minimization of the number of requests that are sent over the network and to the disk. There are two well known collective I/O methods: *two-phase I/O* [15] and *disk-directed I/O* [23].

The two phases of the first one are the shuffle and the I/O operation. In the shuffle phase the data is permuted by the compute nodes into a distribution that matches the physical distribution of data over the I/O nodes. In the second phase the data is redistributed according to the user defined distribution. The advantage of this scheme is that it decouples the logical distribution selected by the user from the physical distribution. Therefore, the data access phase performs always good, regardless of the logical distribution. The disadvantage is that the data is sent typically twice over the network, once in each phase and that additional memory is needed for permutation phase.

In the disk directed I/O the compute nodes send the requests directly to the I/O nodes. I/O nodes merge the requests and then arranges them in an order that minimizes disk head movement. The disk-directed I/O has several advantages over two-phase I/O: the disk performance is improved by merging and sorting requests, data is sent only once over the interconnect, communication is spread

throughout disk transfer, not concentrated in the shuffle phase, no additional memory is needed at compute node for permuting the data.

4.6 Shared file systems

Originally the storage devices were typically attached to computers. Direct disk access from remote hosts was possible only indirectly by contacting the machine they were attached to. Therefore, if the machine becomes unavailable due to crash, overloading or any other reason, the disks also become inaccessible. This may happen even though the unavailability reason had nothing to do with the disk to be accessed. The advent of network-attached storage (NAS) has allowed the separation of storage from computers. The disks can be attached to the network and accessed by every entitled host directly.

There are several advantages of this approach. First, the NAS allows the separation of file data and metadata [16]. A server running on one machine takes care of metadata, whereas the data is kept on NAS. The clients contact the server when opening a file and receive an authorization token, which can be subsequently used for accessing the disks bypassing the server. This makes the servers more scalable, because they are relieved from data transfer duties. Second, this may eliminate the need for expensive dedicated servers, because a lighter load allows a machine to be used also for other purposes. Third, the performance is boosted, because a client request doesn't have to go through expensive software layers at the server (e.g. operating system), but it can contact the disks directly. Forth, in a server-attached disk, the data has to go through two interconnects: the server internal I/O bus and the external network. The server-attached disks reduce the number of network transits from two to one. Fifth, if a host in the network fails, the disks continue to be available for the remaining machines.

The challenge is how to manage concurrent access by several clients to the disks. If the disks are smart (have a dedicated processor), the concurrent access may be implemented at disks. Otherwise, an external lock manager may be needed. Each client has to acquire a lock for a file or for a file portion before effectively accessing the disks.

4.7 Grid file systems

The Internet is a heterogeneous collection of resources, geographically distributed over a large area. It spawns several administrative domains, each of which running specific software. A single organization may have its resources distributed over the Internet and need a unified management, as for instance the meteorological stations. Or distinct organizations may need to cooperate for a common goal, for instance academic institutions that work in a common project.

Grid file systems [33] offer a common view of storage resources distributed over several administrative domains. The storage resources may be not only disks, but also higher-level abstractions as files, or even file systems or data

bases. The grid file system gathers them into a unified name space and allows their shared usage.

Grid file systems are usually composed of independent administrative domains. Therefore, they must allow the smooth integration or removal of resources, without affecting the integrity of neither the individual independent domains nor the system as a whole.

4.8 Mobility

The increasing development of mobile computing and the frequent poor connectivity have motivated the need for weakly connected services. The file system users should be able to continue working in case of disconnection or weak connectivity and update themselves and the system after reintegration.

A mobile computer typically pre-fetches (hoards) data from a file system server in anticipation of disconnection [4]. The data is cached on the local disk of the mobile computer. If disconnection occurs the mobile user may continue to use the locally cached files. The modification of the files can be sent to the server only after reconnection. If several users modify the same file concurrently, consistency problems may occur. The conflicts may be resolved automatically at data reintegration by trying to merge the updates from different sources. If the automatic process fails, the data consistency has to be solved manually. However, as we have shown in the access pattern section, the sequential applications rarely share a file for writing. Therefore, it is likely that the conflicts occur rarely and the overhead of solving them is small.

5 SUMMARY

This chapter provided an overview of different file system architectures. We showed the influence of I/O access pattern studies results on file system design. We presented techniques, algorithms and data structures used in file system implementations. The paper overviewed issues related to both local and distributed file systems. We described distributed file system architectures for different kinds of network connectivity: tightly-connected networks (clusters and supercomputers), loosely-connected networks (computational grids) or disconnected computers (mobile computing). File systems architectures for both network-attached and computer-attached storage were reviewed. We showed how the parallel file systems address the requirements of I/O bound parallel applications. Different file sharing semantics in distributed and parallel file systems were explored. We also presented how efficient metadata management can be realized in journaled file systems.

References

1. Thomas Anderson, Michael Dahlin, Jeanna Neefe, David Paterson, Drew Roselli, and Randolph Wang. Serverless network file systems. In *Proceedings of the 15th*

- Symposium on Operating Systems Principles*, pages 109–126, Copper Mountain Resort, Colorado, December 1995.
2. M. Bach. *The Design of the UNIX Operating System*. Prentice Hall, 1990.
 3. M. Baker, J. Hartman, M. Kupfer, K. Shirriff, and J. Ousterhout. Measurements of a distributed file system. In *Proceedings of the 13th ACM Symposium on Operating Systems Principles*, pages 198–212, 1991.
 4. P.J. Braam. The Coda distributed file system. *Linux Journal*, June 1998.
 5. P. Cao, E.W. Felten, and K. Li. Implementation and performance of application controlled file caching. In *Proceedings of 1st USENIX Operating Systems Design and Implementation*, pages 165–177, 1994.
 6. F.W. Chang and G.A. Gibson. Automatic I/O hint generation through speculative execution. In *Proceedings of 1st USENIX Operating Systems Design and Implementation*, pages 1–14, 1999.
 7. P.F. Corbett and D.G. Feitelson. The Vesta parallel file system. *ACM Transactions on Computer Systems*, 14(3):225–264, August 1996.
 8. P.F. Corbett, D.G. Feitelson, J.-P. Prost, G.S. Almasi, S.J. Baylor, A.S. Bolmaričich, Y. Hsu, J. Satran, M. Snir, R. Colao, B. Herr, J. Kavaky, T.R. Morgen, and A. Zlotek. Parallel file systems for IBM SP computers. *IBM Systems Journal*, 34(2):222–248, 1995.
 9. T. H. Cormen and A. Colvin. ViC*: A preprocessor for virtual-memory C*. Technical report, Departement of Computer Science, Darmouth College, 1994.
 10. T. Cortes, S. Girona, and L. Labarta. PACA: A distributed file system cache for parallel machines. performance under UNIX-like workload. Technical Report UPC-DAC-RR-95/20, Departament d'Arquitectura de Computadors, Universitat Politecnica de Catalunya, 1995.
 11. P.E. Crandall, R.A. Aydt, A.A. Chien, and D.A. Reed. Input/Output characteristics of scalable parallel applications. In *Proceedings of Supercomputing '95*, 1995.
 12. K.M. Curewitz, P. Krishnan, and J.S. Vitter. Practical prefetching via data compression. In *Proceedings of ACM International Conference on Management of Data(SIGMOD)*, pages 257–266, 1993.
 13. M. Dahlin, R. Yang, T. Anderson, and D. Patterson. Cooperative caching: Using remote client memory to improve file system performance. In *Proceedings of the 1st Symposium on Operating Systems Design and Implementation*, pages 267–280, 1994.
 14. E. DeBenedictis and J.M. De Rosario. nCUBE parallel I/O software. In *Proceedings of 11th International Phoenix Conference on Computers and Communication*, 1992.
 15. J. del Rosario, R. Bordawekar, and A. Choudhary. Improved parallel I/O via a two-phase run-time access strategy. In *Proceedings of 7th International Parallel Processing Symposium Workshop on Input/Output in Parallel Computer Systems*, 1993.
 16. G.A. Gibson, D.F. Nagle, K. Amiri, F.W. Chang, H. Gobioff, E. Riedel, D. Rochberg, and J. Zelenka. Filesystems for network-attached secure disks. Technical Report CMU-CS-97-118, School of Computer Science, Carnegie Mellon University, 1997.
 17. J. Griffioen and R. Appleton. Reducing file system latency using a predictive approach. In *Proceedings of USENIX Summer 1994 Technical Conference*, pages 197–207, 1994.
 18. J. H. Howard. An overview of the Andrew file system. In *Proceedings of the USENIX Winter Technical Conference*, pages 23–26, 1988.

19. J.V. Huber, C.L. Elford, D.A. Reed, A.A. Chien, and D.S. Blumenthal. PPFS: A high performance portable file system. In *Proceedings of the 9th ACM International Conference on Supercomputing*, pages 385–394, 1995.
20. IBM Corporation, <http://www-124.ibm.com/developerworks/oss/jfs/>. *JFS website*.
21. W.B. Ligon III and R.B. Ross. An overview of the Parallel Virtual File System. In *Proceedings of the Extreme Linux Workshop*, 1999.
22. F. Isaila and W. Tichy. Clusterfile: A flexible physical layout parallel file system. In *Third IEEE International Conference on Cluster Computing*, pages 37–44, October 2001.
23. D. Kotz. Disk-directed I/O for MIMD multiprocessors. In *Proceedings of the 1st USENIX Symposium on Operating Systems Design and Implementation*, pages 61–74, 1994.
24. D. Kotz and C.S. Ellis. Practical prefetching techniques for multiprocessor file systems. *Journal of Distributed and Parallel Databases*, 1(1):33–51, January 1993.
25. T.M. Kroeger and D.D. E. Long. Predicting future file-system actions from prior events. In *Proceedings of USENIX Annual Technical Conference*, pages 319–328, 1996.
26. H. Lei and D. Duchamp. An analytical approach to file prefetching. In *Proceedings of USENIX Annual Technical Conference*, 1997.
27. Message Passing Interface Forum. *MPI2: Extensions to the Message Passing Interface*, 1997.
28. T.C. Mowry, A.K. Demke, and O. Krieger. Automatic compiler-inserted I/O prefetching for out-of-core applications. In *Proceedings of 2nd USENIX Operating Systems Design and Implementation*, pages 3–17, 1996.
29. Namesys, <http://www.reiserfs.org/>. *ReiserFS Whitepaper*.
30. N. Nieuwejaar and D. Kotz. The Galley parallel file system. *Parallel Computing*, 23(4-5):447–476, June 1997.
31. N. Nieuwejaar, D. Kotz, A. Purakayastha, C.S. Ellis, and M.L. Best. File access characteristics of parallel scientific workloads. *IEEE Transactions on Parallel and Distributed Systems*, 7(10):1075–1089, October 1996.
32. M.T. O’Keefe. Shared file systems and Fibre Channel. In *Proceedings of the 6th NASA Goddard Space Flight Center Conference on Mass Storage Systems and Technologies*, 1998.
33. R. Oldfield and D. Kotz. A parallel file system for computational grids. In *Proceedings of the 1st IEEE/ACM International Symposium on Cluster Computing and the Grid*, pages 194–201, 2001.
34. J. Ousterhout, H. Da Costa, D. Harrison, J. Kunze, M. Kupfer, and J. Thompson. A trace-driven analysis of the UNIX 4.2 BSD file system. In *Proceedings of the 10th Symposium on Operating Systems Principles*, pages 15–24, 1985.
35. M. Paleczny, K. Kennedy, and C. Koelbel. Compiler support for out-of-core arrays on data parallel machines. In *Proceedings of Symposium on Frontiers of Massively Parallel Computation*, 1995.
36. D.A. Patterson, G.A. Gibson, and R.H. Katz. A case for redundant arrays of inexpensive disks (RAID). In *Proceedings of Proceedings of ACM International Conference on Management of Data(SIGMOD)*, pages 109–116, 1988.
37. R.H. Patterson and G.A. Gibson. Exposing I/O concurrency with informed prefetching. In *Proceedings of 3rd International Conference on Parallel and Distributed Information Systems*, 1994.

38. M. Rosenblum and J.K. Ousterhout. The design and implementation of a log-structured file system. *ACM Transactions on Computer Systems*, 10(1):26–52, February 1992.
39. A. Rowstron and P. Druschel. Storage management and caching in PAST, a large-scale, persistent peer-to-peer storage utility. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles*, pages 188–201, 2001.
40. R. Sandberg, D. Goldberg, S. Kleinman, D. Walsh, and B. Lyon. Design and implementation of the Sun network filesystem. In *Proceedings of the Summer USENIX Conference*, 1985.
41. M. Satyanarayanan. A study of file sizes and functional lifetimes. In *Proceedings of 8th ACM Symposium on Operating System Principles*, pages 96–108, 1981.
42. F. Schmuck and R. Haskin. GPFS: A shared-disk file system for large computing clusters. In *Proceedings of 1st USENIX Conference on File and Storage Technologies*, pages 231–244, 2002.
43. SGI Corporation, <http://oss.sgi.com/projects/xfs/>. *XFS: A high-performance journaling file system*.
44. H. Simitici and D.A. Reed. A comparison of logical and physical parallel I/O patterns. *International Journal of High Performance Computing Applications, special issue (I/O in Parallel Applications)*, 12(3):364–380, 1998.
45. E. Smirni and D.A. Reed. Workload characterization of I/O intensive parallel applications. In *Proceedings of the Conference on Modelling Techniques and Tools for Computer Performance Evaluation*, pages 169–180, 1997.
46. R. Thakur, R. Bordawekar, and A. Choudhary. Compilation of out-of-core data parallel programs for distributed memory machines. In *Proceedings of 8th International Parallel Processing Symposium Workshop on Input/Output in Parallel Computer Systems*, 1994.