Evaluation of Design Alternatives for a Cluster File System

Murthy Devarakonda, Ajay Mohindra, Jill Simoneaux,* and William H. Tetzlaff

IBM Thomas J. Watson Research Center, Hawthorne, NY.

Abstract – Based on implementation experience and measurements, this paper presents an evaluation of design alternatives to a *cluster file system.* The file system is targeted for IBM cluster systems, Scalable POWERparallel and AIX HACMP/6000. We considered a *shared disk approach* where serialized, multiple instances of a single-system file system directly access file data as disk blocks, and a *shared file system approach* which is the conventional method of distributing file system function between a client and a server. We conclude that the shared disk approach suffers from the difficulties of metadata serialization, poor write-sharing performance, and read throughput.

1 Introduction

Based on implementation experience and measurements, this paper presents an evaluation of design alternatives to a cluster file system. The file system is targeted for IBM clusters, Scalable POWERparallel [1] and AIX HACMP/6000 [2]. VAXClusters [3] are other well known cluster systems. In general, a cluster is a loosely coupled system of processing nodes, each with its own CPU, memory, disks, and network adapters. The cluster systems considered here also have characteristics such as single administrative domain, peer relationship among nodes, homogeneity, and special purpose hardware (i.e. high-speed switch or multiported disks). A cluster file system, unlike a conventional network file system, can take advantage of these characteristics for high performance and fault tolerance. Functionally, a cluster file system is expected to provide strong cache consistency, POSIX [4] support, and high availability.

*J. Simoneaux was an employee of IBM while working on this project; She is now with the Intel Corp.

We considered two design approaches to the cluster file system: (1) a *shared disk approach,* where multiple, serialized instances of a single-system file system run on the cluster nodes, directly accessing the file data as disk blocks; (2) a *shared file system approach,* which is the conventional client/server scheme using the vnode interface. Note that compatibility with the existing disk format is a requirement so that users can easily migrate to a cluster from a single system.

This evaluation is based on implementations in AIX/6000 using the same cache consistency mechanism and the cache manager. Cache consistency has been implemented using a distributed token manager [5], and AIX virtual memory has been used as the data cache. In the shared disk approach, the lesser known of the two approaches, the AIX Journaling File System (JFS) [6] has been extended by adding cluster-wide serialization with the help of the token manager. Remote disk access has been provided over the network. Comparison of this parallel JFS implementation with the shared file system approach is one part of the evaluation. The conclusion is that efficient metadata serialization is intrinsically difficult in the shared disk approach.

The second part of the evaluation is based on performance measurements of the two implementations. (Although adequate for an insightful evaluation, our implementation of the shared disk approach can simultaneously support only one read-write mount and many read-only mounts.) We have used an Nhfsstone-derived micro-benchmark, Andrew benchmark based scalability measurements, read and write throughput measures, and a concurrent benchmark. Results show that the shared file system approach performs better in concurrent read-write access and in reading a large file sequentially, whereas the shared disk approach has a small advantage when almost all accesses can be serviced by the local cache. Since a file system for a homogeneous cluster can be designed using either one of the approaches, an experimental evaluation is valuable; we are unaware of such previous work.

Although fault tolerance is important in a cluster file system, we did not attempt an evaluation based on fault tolerance. However, a non-disruptive recovery scheme exploiting the multi-ported disk hardware has been implemented for the shared file system approach. Details can be found in a technical report [7].

The rest of the paper is organized as follows: The next section summarizes the workload expected for the cluster file system. Section 3 describes how the two design approaches relate to earlier file systems. Section 4 describes the two design approaches. Section 5 discusses serialization challenges in the shared disk approach. Section 6 presents performance measurements. Section 7 concludes the paper.

2 Workload Characterization

The application domain for VAXClusters [8] provides a working example for the types of applications suitable for clusters and cluster file systems. Based on this and our understanding of commercial applications, we anticipate two types of workloads: one, the traditional UNIX usage patterns [9, 10], and the other, transactional accesses to large files. The first type of workload is characterized by many small files, whole-file sequential accesses, mostly read accesses, little write-sharing, and high locality of reference. In a cluster, these patterns occur when the cluster file system is used to access commands, configuration files, user files, and so on. We also expect interactive computing uses of the cluster for editing, compiling, and testing of software. This usage is predominant when the users connect through Xterminals and use the cluster as a fault-tolerant, scalable computing system. For this workload, in addition to such well known techniques as caching, cache consistency, read-ahead, and write-behind, a private protocol[1] on the internal high-speed switch can provide additional performance improvement.

The second type of workload is characterized by large files, random accesses to file pages, and read-write sharing. Typically this type of workload is present when the cluster is used as a highly available transaction server. The transaction system may be a client-server application, where the server part runs on each of the cluster nodes and accepts requests from the clients running on PCs or workstations. The transaction system may be a vendor provided database or custom built application. The transaction system can be designed using a partitioned or shared data model. The partitioned model does not require a single-image file system. The application takes on complete responsibility for directing client requests to the appropriate server node and recovery from server node failures. The shared data model can exploit a single-image file system for load balanced, seamless use of the cluster. It can also make use of the fault-tolerant characteristics for high data availability.

[1] Implementations described here do not use a private protocol at this time.

3 Related Work

The shared file system approach has been used in many existing distributed file systems such as NFS [11], AFS [12], DFS [13], Locus file system [14], and Sprite file system [15]. Unlike these file systems, our cluster file system is intended to run in a homogeneous environment, therefore a shared disk approach is a potential alternative.

Before discussing the related work in the shared disk approach, we should note that the caching and degree of cache consistency vary significantly among above mentioned file systems. Both approaches discussed here use a distributed token manager to provide strong cache consistency which is functionally similar to DFS and Sprite.

One existing file system that employs a shared disk approach is the file system for the VAXCluster [16]. The file system is not a UNIX file system, but it supports operations similar to open, close, read, and write. The authors started with a single-process (i.e. non-reentrant) version of the file system, and developed a version that supported not only multiple processes but also the cluster. We followed a similar methodology in developing the shared disk approach from JFS, except that JFS was already capable of supporting multiple tasks. One key difference is that the VAXCluster file metadata has simpler structure and layout as compared to JFS. Another important difference is that the operation of the VAX-Cluster file system is not as closely tied to the virtual memory manager as JFS. However, JFS derives significant performance benefits due to its metadata design and close ties to the virtual memory manager.

Welch has compared three shared file system architectures in [17], but we are not aware of any previous work that compares the shared disk approach with the shared file system approach. Since a cluster system allows both approaches to the design of a single-image file system, an experimental evaluation of the approaches is valuable.

4 Two Design Approaches

In this Section, we briefly discuss the token manager, as it is common to both approaches. Next, we provide an overview of the two approaches. Metadata serialization in the shared disk approach is discussed separately (in Section 5.1), as it requires an in-depth treatment.

4.1 The Distributed Token Manager

Tokens convey authorization to perform certain actions on file data. For example, after acquiring a read token on the data pages of a file, a node can cache the pages and allow read operations on the data until the token is revoked. In both designs, the relevant tokens are marked busy during the execution of a vnode operation. Thus, cluster-wide locking is defined as marking a token busy, and cluster-wide unlocking is defined as marking it *not* busy. A token, once acquired, remains with the cluster node until it is revoked by some other node. A token can be revoked only when it is marked not busy.

A node initiates token arbitration to acquire, upgrade, or voluntarily give up a token. When a node is attempting to acquire a token, the *token client* contacts the *token server* first. The token server is the node with a physical connection to the disks containing the file system. The server always maintains the current status of a token. Therefore, when a request is received, the server grants the token if there are no conflicts. If conflicts do exist, server replies with the *copyset*, i.e. the list of nodes holding the tokens. In the latter case, the requesting node revokes or downgrades the tokens (from write to read) at the copyset nodes, and then acquires the token.

What tokens are used and what objects they represent are important questions. However, the answers depend on the specifics of the design. Note that the token manager operation does not depend on what objects the tokens represent but only on the token semantics (such as the single-writer/multiple-readers semantics).

4.2 The Shared Disk Approach

In the shared disk approach, multiple instances of the standard AIX file system (JFS), each running on a different cluster node, directly access on-disk file data as disk blocks. Each JFS instance caches disk blocks as in the stand-alone system. However, because all instances are simultaneously accessing the disk blocks, it is necessary to serialize accesses and provide data consistency. This is similar to "parallelization" for a symmetric, shared-memory multiprocessor. Therefore, we call this approach *Parallel JFS (PJFS)*. Here, however, locking is done at a larger granularity to account for the communication overhead in a cluster.

First consider the control flow resulting from file related system calls in the stand-alone AIX. The *logical file system* implements system calls by invoking vnode operations of a virtual file system type. In the stand-alone AIX, the virtual file system type would be JFS and hence its vnode or vfs operations are invoked. Since JFS uses the virtual memory for caching, the virtual memory manager handles file cache misses by reading from the *logical volume* containing the file system. Similarly, file cache flushes are written to the logical volume. A logical volume is an abstraction of a Unix disk partition, and presents a block device interface to the virtual memory manager and JFS. This block device interface is supported through a pseudo-device driver called *logical volume manager* (LVM).

PJFS extends this architecture to a cluster, by implementing two additional mechanisms: A *remote logical volume (RLV)* pseudo-device driver to provide access to a disk from multiple cluster nodes, and a limited number of modifications to JFS to invoke the token manager functions for cluster-wide serialization. Figure 1(a) is a schematic view of this design. If disks are sufficiently multi-ported, the remote logical volume device driver is not needed. However, the present hardware supports only four ports.

The JFS implementation has a convenient 'hook' for using the cluster-wide locking scheme. At the beginning of every vnode operation, JFS locks relevant, in-core inodes using single-system semaphores, and unlocks them at the end. Each

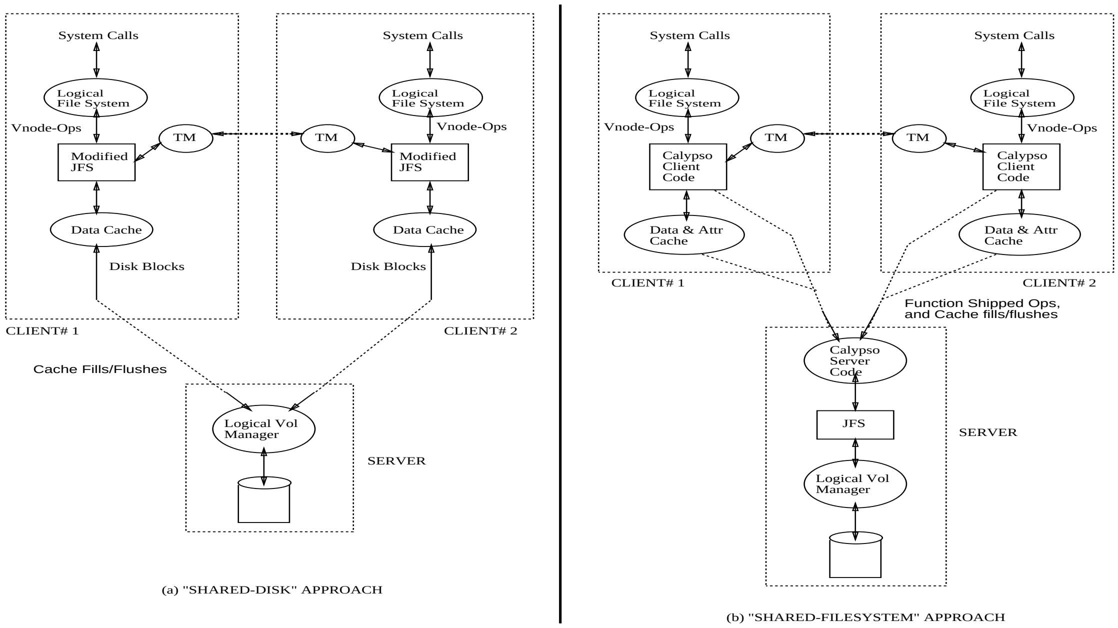

Figure 1: The Two Design Approaches

inode has a semaphore field for this purpose. We modified the inode locking function to invoke our cluster-wide locking function first and then to execute the local locking.

JFS cache data is flushed or purged as per the token protocol. When a write token is downgraded or revoked, any modifications of the file are flushed to the disks. When a token is revoked, the data and attributes of the file are purged from the cache. Existing interfaces are used to implement these flush and purge operations. After acquiring a token, caches are filled in the normal course of JFS activity.

Token Definitions In PJFS, one token represents the whole file. However, for some meta-files, which are discussed in Section 5.1, a token represents a page. For a normal file, therefore, a token represents the in-core inode, in-core indirect blocks (if any), and all the data pages of the file (Figure 2). Because JFS caches file data and indirect blocks only once, this token also represents on-disk file data and indirect blocks. However, the token does not represent the on-disk inode. JFS caches disk inodes separately from in-core inodes,[2] and carefully manages the transfer of contents between disk inodes and in-core inodes. Therefore, we use separate tokens to represent disk inodes as discussed in Section 5.1.

Implementation Status of PJFS We have implemented this PJFS approach to the extent that a file system could be mounted for read-write on one node (not necessarily the node with the disk) and read-only on any number of other nodes at the same time. This was enough to carry out meaningful performance measurements as well as to develop an in-depth understanding of the implementation issues.

4.3 The Shared File System Approach

The shared file system approach is the conventional method of distributing the file system function between a client and the server using the vnode interface [18]. We call this approach Calypso. Figure 1(b) provides a schematic view of this approach. One of the cluster nodes connected

[2] JFS inodes have different structures on-disk and in-core, however, the in-core inode is a superset of the on-disk inode.

to the disk, functions as the server for file systems on the disk. Other cluster nodes function as the clients. Similar to NFS, a node may be the server for some file systems, and a client for some others. On clients, a new virtual file system is implemented, which is shown as the Calypso client in Figure 1(b). On the server, an interface layer, shown as the Calypso server in Figure 1(b), provides interface to unmodified JFS. The clients obtain remote services from the server using Remote Procedure Calls (RPCs). In addition to the distributed token manager, this design requires the client and server parts.

Client Part The client part implements *vnode* and *VFS* operations as defined in AIX/6000. Every vnode operation first acquires cluster-wide locks on necessary files, proceeds to perform the function as required, and then releases the locks. The vnode operations make use of memory caching of file pages and attributes. File pages are cached using the virtual memory manager. Attributes (e.g., access rights, ownership, and size) are cached in inode-like structures, and name to vnode mappings are cached as in NFS. As needed, these caches are filled and flushed using remote service requests to the server. Directory changing vnode operations such as the file `create` are implemented entirely at the server.

Server Part The server part consists of processes and code to service remote requests from the clients. All operations use JFS vnode interface to access actual file data.

Token Definitions Although, we later evolved a more complex token definition [5], for this evaluation, we used one token to represent all accesses to the file.

5 PJFS Implementation Experience

In this section, we discuss metadata serialization and journal management for the shared disk approach. Then we present a summary of the implementation difficulties. Finally, we discuss a generalization of the implementation experience.

5.1 JFS Metadata Serialization

Similar to most UNIX file systems, JFS metadata consists of a superblock, inodes, inode extensions containing optional access control lists, indirect blocks, a disk allocation map, and an inode allocation map. However, JFS accesses these data structures themselves as files using well known inode numbers that are inaccessible to normal users. In the next three subsections, we discuss serialization and consistency schemes for the inodes, indirect blocks, and the disk allocation map. We use these three data structures as examples to illustrate the implementation difficulties of the shared disk approach. For details on JFS, the reader is referred to Chang *et al* [6].

Inodes

Page-level tokens provide cluster-wide serialization of the inodes file. A read token is needed for reading an inodes-file page, and a write token is needed for writing a page. The decision to use page-level tokens is based on the observation that the inodes, as a whole, experience more sharing than a regular file, so a single lock would be a serialization bottleneck. However, locking at less than the physical page level is difficult to implement. Page-level tokens are a compromise between two extremes. As dictated by the token protocol, the inodes-file is flushed or invalidated on a page-by-page basis.

Indirect Blocks

Unlike the inodes, the indirect blocks in JFS don't have a fixed position in the indirect-blocks file. Instead, indirect blocks are mapped on demand during the use of the file system. An indirect block is a just another disk block until it is referenced. Because of this design, invalidation of indirect blocks requires unmapping and remapping of the indirect blocks when large files are write-shared.

Disk Map

The disk allocation map consists of several pages, each with control information and allocation status bits. Because of the need for journaling, JFS creates a temporary allocation map. It differs from the permanent map in allocations that have not yet been journaled. Because the temporary map can be paged out, JFS uses the same virtual memory segment as for the permanent map. As a result both are backed by the file system on the disk.

When extended to an N-way cluster system, this requires N different temporary maps (because the contents may differ), each with its own backing store. A scalable solution is to use a 'working storage' segment for each temporary map. This

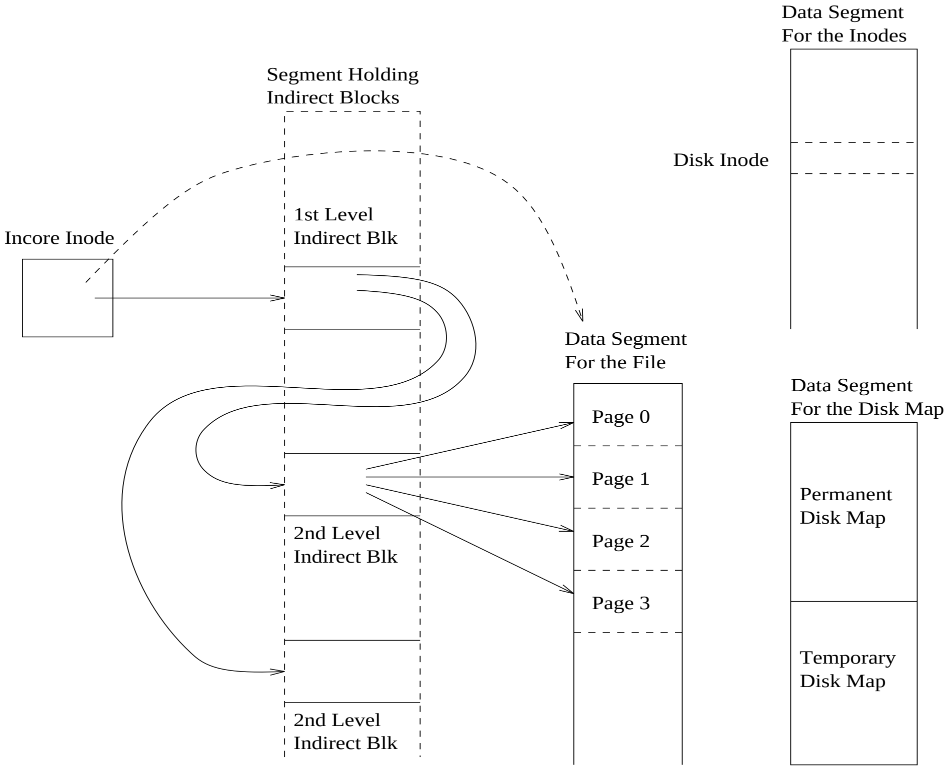

Figure 2: Components and Organization of JFS Meta-Data

type of segment can be created as needed, and backed by paging space. Note that a single, file-system backed segment (as in JFS) is not scalable. The number of temporary maps that can fit into a segment is limited.

However, using working-storage segments is not practical because it requires extensive changes to existing JFS code, which undermines our goal of keeping cluster-specific changes to a small number of lines and modules. Even if we had resolved this problem, disk map organization in units of 4K-byte pages would have caused a significant bottleneck. It would be difficult to share efficiently such a structure across the cluster nodes. Furthermore, each page contains 256-bytes of control information relating to the entire disk map.

5.2 Journal Management

In PJFS, write-ahead logging can be implemented in several ways. For example, for each file system, either a single log can be used for all cluster nodes, or a log per cluster node can be used. An intermediate solution is also possible. The single log solution has the obvious problem of poor concurrency, whereas the multi-log solution has the following disadvantages:

- It requires a two stage recovery;
- The log records must contain a timestamp using a global clock or counter so that, at the time of merging the sublogs, the log records can be properly ordered;
- It introduces the possibility of a sublog being unavailable at the time of recovery (partial failure scenario versus total failure scenario).

Neither of the solutions seem appropriate for the purpose. So, the PJFS design is to use a single log for each file system, but instead of caching the last page of the log, the log writes are function-shipped to the server node where they are written to the log device. Usually, several log records are combined in each function shipping operation.

5.3 Implementation-Based Assessment

Meta-data serialization with high concurrency is the most difficult and is clearly the limiting factor. Specifically, the journal and inode management can be supported with a reasonable level of concurrency, but the disk map and indirect blocks would be most difficult to support and likely to perform poorly. To achieve efficient serialization, the metadata should be designed with clusters in mind. For example, JFS uses a 4K-byte page to store the allocation information for 30,720 disk pages, which is 120 Mbytes of disk storage. PJFS uses page-level tokens and caching for the allocation map because it is the most natural extension to JFS. However, a serialization granularity of 120 Mbytes of disk storage is too large. Therefore, the designer of a cluster file system, if starting from the scratch, would make different trade-offs regarding the metadata structures.

Another troublesome characteristic of a single-system file system is that the references to the metadata are scattered throughout the code. In a cluster system, such references must be limited to a small critical section under cluster-wide serialization.

5.4 Generalization Beyond JFS

Any existing single-system optimized file system would have the same problems as JFS. Such file systems typically cache metadata extensively using virtual memory and may even employ journaling for quick restart. Note that journaling is employed in other commercial file systems (e.g., Veritas [19] and Episode [20]) besides JFS. In clusters, distributed log management and efficient serialization of metadata are difficult. Furthermore, optimization for a single-system use is not usually the same as for a cluster.

What about non-journaling, buffer-pool based file system such as the BSD fast file system? Even in such a file system, achieving high concurrency for the metadata (such as the disk map) remains a problem because page-level granularity would still be too large. In addition, it is not clear that parallelizing the BSD fast file system would be useful, since quick restart and reliability have already been demonstrated in journaling file systems.

6 Performance Comparison

To quantify design implications of PJFS and Calypso, we used micro and macro benchmarks to measure caching efficiency, scalability, throughput, and write sharing overhead. The key findings are:

- Calypso performs significantly better than PJFS when data is write shared among the nodes;
- For cached data, PJFS has a slight edge over Calypso;
- Calypso's read throughput over the network is better than that of PJFS.

Measurements described here are taken on RISC System/6000 Model 530's on a 16 Mbit/s token ring. Each system has 80 Mbytes of memory and SCSI-2 disks. NFS (Version 2) measurements are also shown where appropriate so that the differences in PJFS and Calypso can be seen in relationship to a well known network file system.

6.1 Micro Benchmark Performance

A micro benchmark called *Fsstone* has been derived from Nhfsstone by removing NFS specific code. It measures performance of individual file operations. The benchmark opens several files in a directory, and executes a pre-defined mix of system calls such as `read`, `write`, `stat`, `create`, and `unlink`. The total number of system calls executed is typically large, such as 30,000, therefore the results show the positive effects of caching in the file system.

The measurements are taken with one (client) node running the benchmark on a file system residing on another (server) node. For PJFS, both the file system and the log are on the server. Results are shown in Table 1.

Table 1 shows that there is no clear winner between PJFS and Calypso. PJFS caches all types of data whereas Calypso caches most but not all. This works in favor of PJFS most of the time. However, in some complex operations, such as `mkdir`, `rmdir`, and `create`, caching and local manipulation of low-level metadata is expensive. Calypso's strategy of function shipping these operations is more efficient. Overall, for a representative mix of operations, PJFS has a small advantage over Calypso because such a mix is dominated by `lookup`, `read`, `readdir`, and `fsstat` operations, where PJFS has the advantage. It can be seen that both PJFS and Calypso out perform NFS due to their strong cache consistency scheme.

6.2 Scalability

Scalability measurements are another way of looking at the positive effects of caching. The

Table 1: Results from a micro benchmark showing average times for various file operations. The benchmark makes a mix of file-related system calls on a set of files.

Operation	Milliseconds/Call		
	PJFS	Calypso	NFS
get attributes	0.55	0.58	10.33
set attributes	1.59	5.47	6.81
lookup	0.69	1.03	6.12
read	0.53	0.58	4.45
write+fsync	22.75	22.25	25.94
create	24.29	21.38	45.54
remove	37.55	28.55	41.62
rename	38.95	43.18	39.22
readlink	1.09	11.75	14.06
link	39.94	34.38	44.85
symlink	51.10	35.87	39.39
mkdir	57.15	41.78	43.41
rmdir	51.89	40.69	49.05
readdir	0.93	9.53	10.41
fsstat	0.74	5.60	9.90

Andrew benchmark, which has been used in AFS scalability measurements [12], is used for the purpose. The benchmark consists of five phases, in which subdirectories are created, source files are copied, file attributes and contents are examined, and the source is compiled. Thus, this benchmark also represents the workload of interactive computing typical in research and software development.

Figure 3 shows results of the measurements. PJFS and Calypso have essentially an identical scalability characteristics as they are both based on an identical cache consistency scheme. PJFS exhibits a small edge over Calypso at each measurement point, mainly because of the performance difference in the `readdir` operation (see Table 1), which is prominent in the third phase of the benchmark. It can be seen that NFS performs poorly relative to both these file systems.

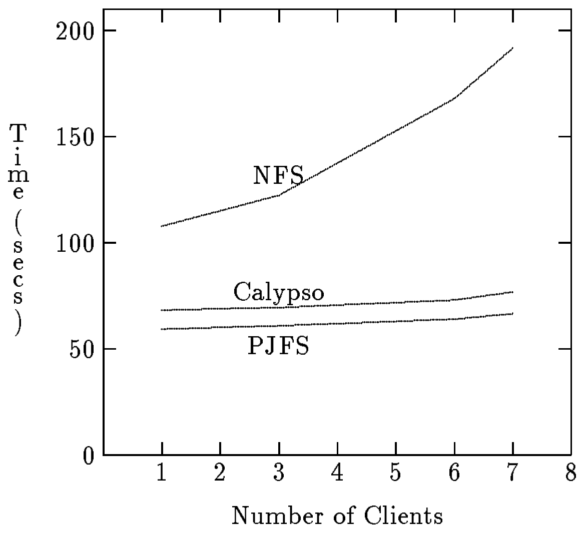

Figure 3: Scalability measurements using the Andrew benchmark. Elapsed times for the benchmark are shown for increasing number of nodes. Each client ran the benchmark in a separate directory.

6.3 Throughput Measurements

In the next two experiments, a 2 Mbyte file is written and a 2 Mbyte file is read. In the file-write experiment, the source is in the memory thus isolating the file system write throughput. Similarly, in the file-read experiment, caches are invalidated to obtain cold cache read throughput. Results of are shown in Figure 4.

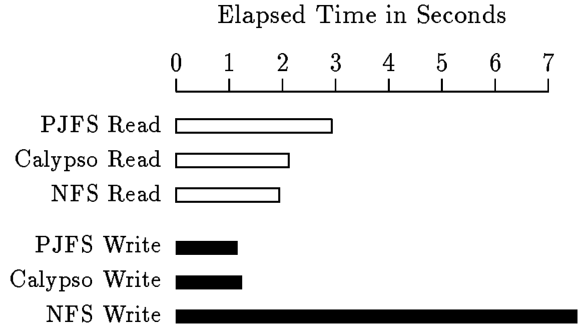

Figure 4: Throughput measurements. The first set shows read times for a 2M file under cold cache conditions. The second set shows elapsed times for copying a 2M file from memory.

Calypso and NFS show superior read throughput, as much as 30% better performance than PJFS. The reason is that Calypso and NFS use read-aheads at the client as well as at the server, whereas PJFS can perform read-aheads at the client only. On the client side, all three file systems use AIX virtual memory as the cache manager, which initiates read-aheads (by default, up to 8 pages) upon recognizing sequential access. However, on the server side, only NFS and Calypso have the advantage of substantial read-aheads as they both use local JFS to read file data from the disk. PJFS is limited to using a device driver that can perform at most one block read-ahead. (On the server, the RLV mechanism of the PJFS uses UNIX buffer management operations, *bread, breada* and *bwrite*, for disk access [21].)

Write throughput is equally good in PJFS and Calypso as they both write data to the client cache pending a flush operation at periodic or forced *sync*. NFS performance is considerably worse as it writes to server disk synchronously (particularly, when user writes as much data as in this benchmark).

6.4 Concurrent Access Performance

To understand performance trade-offs for transactional workloads where read-write sharing can be expected, we used a concurrent benchmark. The benchmark consists of reader and writer programs: The writer program repetitively writes a pre-defined number of bytes starting at the zero offset, and at the same time, the reader program repetitively reads file contents starting also at the zero offset. The elapsed times for several thousands of these simultaneous accesses are measured. To quantify sensitivity to the number of bytes accessed, we conducted two experiments, one using 8 bytes, and the other using 4K bytes. Note that AIX page size is 4K bytes. The results are shown in Figure 5.

Calypso performs significantly better than PJFS, in fact, two to five times better for this benchmark because: (1) PJFS needs to manage changes to the file and its inode separately, whereas Calypso needs to manage only the file; (2) Flushing small changes to the file system data is more expensive in PJFS than in the client code of Calypso; (3) Sharing data through a disk is more expensive than sharing through memory, even though the disk has a front-end buffer pool acting as a write-through cache.

In PJFS, to read a file, the reader first requests a token on the file. This forces the writer to flush changes using internal *fsync* and then yield the token. The fsync has a side effect of journaling inode changes. Next, the reader attempts to refresh the inode by reading the relevant page of the inodes file, which requires obtaining the corresponding token. The second token request forces the writer to flush the inodes page and yield the token. Now the reader reads two disk blocks – the inodes page and the file data block, thus completing the read access. In summary, it requires two token downgrades at the writer, involving disk writes to flush an inodes page, a journal page, and a data block. Our measurements indicate that these downgrades take a total of about 94 milliseconds, which is the major contributor to 118 milliseconds of elapsed time for the read access. Note that transfer of 4K byte page takes about 6 to 8 milliseconds, and about 12 milliseconds for disk I/O. The rest of time is in system overhead, CPU contention from the writer-initiated token activity, and disk seek time.

In PJFS, the write access requires one revoke on the reader node, which invalidates the cached inode and data. This revoke plus the rest of token upgrade takes about 24 milliseconds. Since the inode and data are still cached in the writer, there are no further disk I/O's involved. The write access elapsed time is about 58 milliseconds because of the additional CPU contention from token downgrade activity initiated by the reader.

In Calypso, the reader needs to downgrade only the file token on the writer. This forces the writer to flush the file contents to the server, where they are written to JFS cache. In the 8 bytes experiment, this takes about 11 milliseconds. On the reader, 4 additional milliseconds are used for acquiring the token and for reading the data. The actual elapsed time is about 21 milliseconds, because of the CPU contention from writer-initiated revoke activity.

Finally, in Calypso, the writer takes about 14 milliseconds to revoke the token on the reader; because of the additional CPU contention from the reader-initiated downgrade, the total elapsed time for the write is about 22 milliseconds. When 4K bytes are written and read, the times increase to 26 and 27 milliseconds respectively because of the additional time needed for moving the extra bytes. Overall, Calypso benefits from the fact that only one token is being revoked or downgraded, and data exchange is going through the JFS cache.

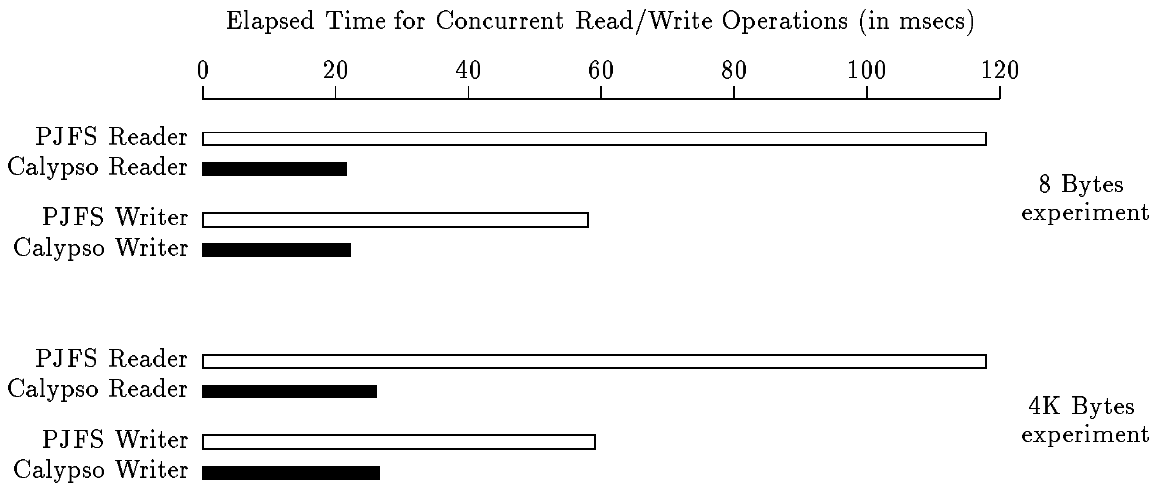

Figure 5: Results from the concurrent access measurements. Two sets of experiments are conducted, and in each set a reader and a writer simultaneously read and write the contents of the same file. The file contains 8 bytes in first experiment, and 4K bytes in the second.

6.5 Effects of Multi-Ported Disks

In the experiments above, we used the RLV mechanism to access the remote disk. However, since a multi-ported disk provides a direct connection to the disk, it can obviously improve access time. This is particularly true in the case of the micro benchmark and 2 Mbyte file read measurements discussed in Sections 6.1 and 6.3. As the hardware fan out is limited to four ports at this time, scalability measurements are not relevant.

However, a multi-ported disk can actually result in poor performance for the concurrent benchmark unless the disk uses non-volatile RAM cache. Without the NVRAM cache, the reader in the concurrent test experiences a latency of 24 milliseconds to read a 4K byte block (12 milliseconds for the writer to flush it to the disk, and another 12 milliseconds for the reader to read it). When using the RLV, the latency is only 16 milliseconds because of the buffering in the RLV server. Not all multi-ported disks use NVRAM cache. Given the fan out limitations and poor write-sharing performance, a general use of multi-ported disks may be in tolerating node failures.

6.6 Generalization Beyond JFS

How dependent are these results on the fact that PJFS uses JFS as the basis? Clearly, the dependency is significant. However, performance similar to PJFS can be expected from any physical file that treats metadata (e.g., inodes) separate from actual data, and most UNIX file systems do so. JFS's journaling also contributes to PJFS's poor performance in the concurrent test. But, without the journaling, JFS (and hence PJFS) performs poorly for typical interactive workloads because of the need to write small amounts of data to the disk. The important insight from these measurements is that file sharing as opposed to data sharing is the most efficient method. The shared file system approach provides caching and hiding of the internal representation to help this.

7 Conclusion

This paper evaluated two design alternatives to a file system targeted for clusters such as the IBM Scalable POWERparallel and HACMP. Since a cluster consists of homogeneous processors, two possible design alternatives were considered: a shared disk approach where serialized, multiple instances of a single-system file system directly access file data as disk blocks; and a shared file system approach, which is the conventional method of distributing file system function between a client and a server using the vnode interface. While most network file systems such as NFS and AFS follow the shared file system approach, the VAXCluster file system employs the shared disk approach. The work reported here is the first to compare the two approaches. For an even comparison, both approaches have been implemented using the same cache consistency mechanism and cache manager. The implementation experience and performance measurements are the basis for the evaluation reported here.

Our implementation experience suggests that the serialization of metadata accesses is the most difficult in the shared disk approach, unless the single-system file system is designed with cluster usage in mind. As we extended AIX single-system file system to the cluster, single-system optimized metadata design and journaling of metadata changes turned out to be most difficult to handle.

Performance measurements of caching efficiency, scalability, throughput, and write-sharing show that the shared file system approach has two advantages: about 2 to 4 times better performance in write-sharing, and about 30% better read throughput. The shared disk approach, however, has a small advantage when almost all accesses can be serviced by the local file cache. Measurements also indicate that, in the shared-disk approach, multi-ported disks do not improve write sharing performance unless the disks use NVRAM caching. The overall conclusion is that the shared file system approach is the right direction for a cluster file system. It should exploit cluster hardware for performance and high availability, and provide efficient support for transactional as well as interactive workloads.

Acknowledgments

We thank the anonymous referees, in particular, our shepherd Peter Honeyman, for helping to improve the paper significantly. We gratefully acknowledge the contributions of the rest of the Calypso team: Bill Kish, Ananda Rao Ladi, Andy Zlotek, Edie Gunter, and Arup Mukherjee. Comments and input from Bob Curran, Wayne Zakaras, Donavon Johnson, and Damon Permezel during the Calypso project have been invaluable.

References

[1] IBM Corporation. *9076 Scalable POWERparallel 2: Technical Brochure; Order Number: GH23-2485-00*, May 1994. Information available from `http://ibm.tc.cornell.edu` Web page.

[2] IBM Corporation. *AIX HACMP/6000 System Overview Ver. 2.1.0; Order Number: SC23-2595-00*, December 1993. Available from IBM Branch Offices.

[3] N. P. Kronenberg, H. M. Levy, and W. D. Strecker. VAXClusters: A Closely-Coupled Distributed System. *ACM Transactions on Computer Systems*, 4(2), May 1986.

[4] POSIX (IEEE Std 1003.1-1990 ISO/IEC Std. 9945-1: 1990). *Portable Operating System Interface (POSIX) Part 1: System Application Program Interface.* IEEE, 1990.

[5] A. Mohindra and M. Devarakonda. Distributed Token Management in Calypso File System. In *Proc. of IEEE Symp. on Parallel and Distributed Processing*, Oct 1994.

[6] A. Chang, M. F. Mergen, R. K. Rader, J. A. Roberts, and S. L. Porter. Evolution of Storage Facilities in AIX Version 3 for RISC System/6000 Processors. *IBM Journal Research and Development*, 34(1), January 1990.

[7] M. Devarakonda, B. Kish, and A. Mohindra. Non-Disruptive Recovery in Calypso File System. Technical Report RC 19794, IBM Thomas J. Watson Research Center, Yorktown Heights, NY 10591, Oct 1994.

[8] W. E. Snaman, Jr. Application Design in a VAXCluster System. *Digital Technical Journal*, 3(3), Summer 1991.

[9] 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 *Proc. of the 10th Symposium on Operating System Principles*, 1985.

[10] M. Baker, J. Hartman, M. Kupfer, K. Shirriff, and J. Ousterhout. Measurements of a Distributed File System. In *Proc. of the 13th Symposium on Operating System Principles*, 1991.

[11] R. Sandberg, D. Goldberg, S. Kleiman, D. Walsh, and B. Lyon. Design and Implementation of the Sun Network File System. In *Proc. of Summer USENIX Conference*, 1985.

[12] J. H. Howard, M. L. Kazar, S. G. Menees, D. A. Nichols, M. Satyanarayanan, R. N. Sidebotham, and M. J. West. Scale and Performance in a Distributed File System. *ACM Transactions on Computer Systems*, 6(1), February 1988.

[13] M. L. Kazar and others. DEcorum File System Architectural Overview. In *Proc. of Summer USENIX Conference*, June 1990.

[14] G. J. Popek and B. J. Walker. *The LOCUS Distributed System Architecture*. The MIT Press, 1985.

[15] M. Nelson, B. Welch, and J. Ousterhout. Caching in the Sprite Network File System. *ACM Transactions on Computer Systems*, 6(1), February 1988.

[16] A. Goldstein. The Design and Implementation of a Distributed File System. *Digital Technical Journal*, (5), September 1987.

[17] B. Welch. A Comparison of Three Distributed File System Architectures. *Computing Systems*, 7(2), Spring 1994.

[18] S. R. Kleiman. Vnodes: An Architecture for Multiple File System Types in Sun UNIX. In *Proc. of Summer USENIX Conference*, 1986.

[19] VERITAS Software, Santa Clara, CA 95054. *VERITAS VxFS File System*, Jan 1994.

[20] S. Chutani, O. T. Anderson, M. L. Kazar, B. W. Leverett, W. A. Mason, and R. N. Sidebotham. The Episode File System. In *Proc. of Winter USENIX Conference*, Jan 1992.

[21] Maurice J. Bach. *The Design of the UNIX Operating System*. Prentice-Hall, 1986.

Murthy Devarakonda received his Ph.D. in Computer Science from the University of Illinois at Urbana-Champaign in January 1988. Since then he has been with the IBM Thomas J. Watson Research Center, Yorktown Heights. At the present, he manages the Object-Oriented Systems Technology group focusing on distributed data management, object-orientation in systems software, and object-support systems. His electronic mail address is `mdev@watson.ibm.com`.

Ajay Mohindra received his Ph.D. in Computer Science from Georgia Institute of Technology in 1993. Currently, he is a Research Staff Member in the Object-Oriented Systems Technology group at the IBM Thomas J. Watson Research Center, Yorktown Heights. His research interests include computer architecture, operating systems, and distributed systems. He can be reached at `ajay@watson.ibm.com`.

Jill Simoneaux was a member of the Calypso project at the IBM Thomas J. Watson Research Center, Yorktown Heights. She is now employed by the Intel Corporation. Her electronic mail address is `jill_simoneaux@ccm.hf.intel.com`.

William H. Tetzlaff is currently Senior Manager of Operating System Technology at the IBM Thomas J. Watson Research Center, Yorktown Heights. He received his MS in Computer Science from Polytechnic University and his BS from Northwestern University. He is a Senior Technical Staff Member at IBM and a member of the IBM Academy of Technology. His electronic mail address is `tetzlaf@watson.ibm.com`.