A Technique for Lock-less Mirroring in Parallel File Systems

Bradley W. Settlemyer and Walter B. Ligon III
Dept. of Electrical and Computer Engineering
Clemson University
105 Riggs Hall
Clemson, SC 29634-0915
Email: {bradles,walt}@clemson.edu

***Abstract*—As parallel file systems span larger and larger numbers of nodes in order to provide the performance and scalability necessary for modern cluster applications, the need for fault-tolerance and high data availability file systems has arisen. Modern parallel file systems spanning tens, hundreds, or even thousands of servers will require fault tolerance to avoid job failure and catastrophic data loss due to a single disk failure or server loss. Effective fault tolerance in parallel file systems must provide a high degree of data resiliency, consistency, and scalable performance. In this paper, we describe a data replication technique that meets the resiliency and consistency requirements of parallel file systems and provides scalable performance. We measure the performance of our proposed mechanism by implementing it in a popular parallel file system, PVFS.**

I. Introduction

In order to support the increasing I/O throughput demands of modern scientific applications, cluster storage subsystems attempt to leverage large numbers of network connections and disks simultaneously. One popular approach to achieving high bandwidth access to large datasets is parallel file systems. Parallel file systems stripe file data across independent storage nodes to provide high bandwidth read and write operations to client applications running on many compute nodes (Figure 1). Cluster file system performance and storage capacity are then governed by the number of storage nodes employed in the cluster.

In addition to increasing the aggregate bandwidth of traditional file systems, parallel file systems also increase the number of independent system components that may fail during a job execution. Traditional data resilience techniques such as RAID can mitigate some degree of the data resilience problem; however, parallel file systems introduce new reliability issues with respect to data availability. Loss of data availability occurs when either the file system cannot be accessed by a job (perhaps due to a network failure), or the file system performance degrades so severely that I/O bound jobs cannot complete in a reasonable amount of time.

In this paper we experiment with a technique for increasing data resilience by mirroring data across storage nodes. The resulting data is both striped and mirrored across the storage nodes, similar to the way RAID 10 (striped mirroring) distributes data across a disk array. When updating a mirrored data set, the file system must ensure that both copies of the file

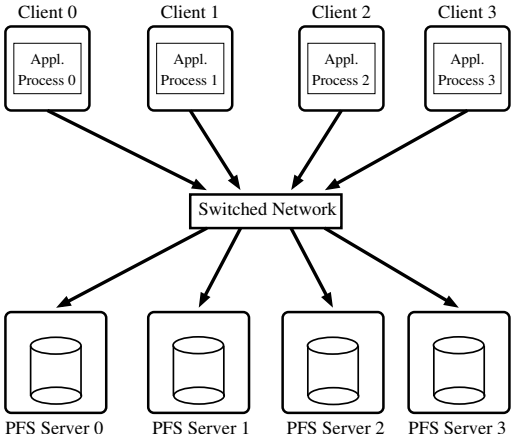

Fig. 1. Parallel File System Network Configuration

data are identical. Other systems have used a lock to guarantee exclusive access to both copies; the client acquires the lock from a distributed lock manager, updates both copies of the data, and finally releases the lock [7], [9]. The use of locks is not ideal for two reasons: First, lock management tends to hurt scalability as it involves additional network messages and potential bottlenecks both at clients and at servers. Second, lock management is notoriously complex [5]; particularly, since it relies on clients which are prone to failure.

Thus, in this paper we explore the use of a lock-less mirroring scheme based on the primary-copy technique [1]. We also describe how our technique stands up to several different types of faults that commonly occur in cluster environments, including byzantine faults. Our experimental results measure the performance overhead and verify the scalability of our lock-less mirroring technique. We implemented our proposed replication mechanism as a prototype using the Parallel Virtual File System (PVFS), a popular parallel file system that is designed to run on Beowulf clusters using commodity networks or high performance cluster networks. Finally, we discuss the limitations of our approach and some ideas for potentially improving data resilience and availability in parallel file systems.

II. Related Work

The Google File System [3] is one of the best documented fault tolerant parallel file systems. A modified primary-copy replication technique is used with a single primary server called a master and two layers of backup servers called primary replicas and secondary replicas. The existence of multiple primary and secondary replica servers is critical to achieving performance in the application environment. The Google File System also does not guarantee that all replicas are kept in a consistent, or identical state. Instead the client application is responsible for handling inconsistent file regions.

The River environment [2] uses a modified form of chained declustering [4] to achieve data mirroring. Graduated declustering implements the basic primary-copy technique, however the primary and backup are no longer single servers. A collection of nodes, each with an entire copy of the data, acts as the primary and another collection of nodes acts as the backup. The existence of multiple servers acting as the primary allows file region accesses to be load balanced across all the nodes, however file writes will still experience the delay associated with the primary-copy technique further multiplied by the number of disks participating in the disk clusters.

GPFS [7], the general parallel file system, provides support for software-based data replication and online fail over. All write requests first obtain a lock token, and then the client sends the data to two separate storage nodes. Because GPFS uses client-based data replication (i.e. the data is sent directly from the client to 2 separate storage nodes), it must use a distributed locking subsystem with support for two-phase locks and lock timeouts to achieve sequential consistency. In conjunction with the locking subsystem, GPFS uses a heartbeat system to detect client and storage node failures.

The CEPH File System [10] provides replication between object storage devices via the primary-copy technique, serial chaining, or splay replication. Splay replication is similar to chained replication although multiple replica servers are written in parallel with the receipt of a successful heartbeat from the primary and replicas required to complete a file read or write.

III. Methods

A. Data Distribution

Parallel file systems stripe file data across several I/O nodes to achieve greater aggregate bandwidth. File system clients can access all the I/O nodes in parallel, reducing the amount of time required to perform file reads and writes. The *data distribution* describes how the logical file is mapped onto physical storage. For example, a simple striping distribution is shown in Figure 2. The individual components of the file data stored on each individual disk are called *data objects*. A file in a parallel file system is composed of all of the data objects for the file and the file metadata, which is typically stored in a structure and location separate from the file data.

Figure 3 shows the data distribution for our replicated file system. The replicated data objects, or secondary copies, are shown with a double outline. Each data object is stored on two separate and independent I/O nodes. For example, strip B is stored on nodes 1 and node 2. The distribution scheme doubles the total number of data objects and ensures that data is duplicated to data objects residing on different servers. An alternative distribution with nodes acting as perfect mirrors was also implemented and provides similar performance results; however, the pure mirror distribution is less flexible since it requires an even number of I/O servers. In the discussion section we propose a possible distribution scheme to improve system performance with a failed I/O node.

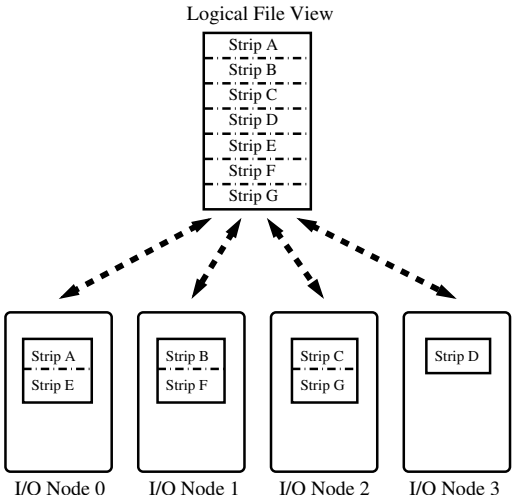

Fig. 2. Data Distribution for a Parallel File System

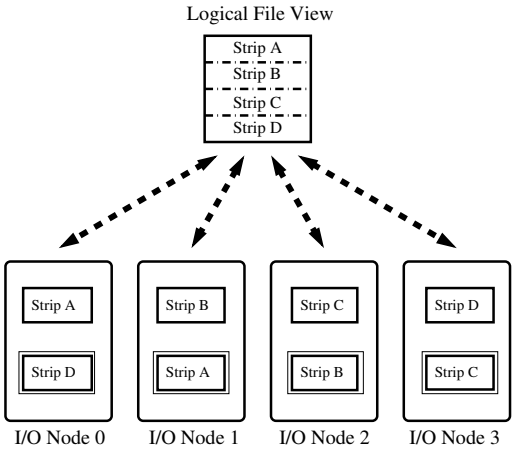

Fig. 3. A Simple Data Distribution for a Replicated Parallel File System

B. Communication Mechanism

The simplest way to achieve file replication is to modify the file system client so that instead of sending one copy of the data to an I/O server, the clients send identical data to two I/O nodes. One problem with this approach is that the client processes are now responsible for ensuring that both file requests are identically applied to the file system. If one of the write requests completes immediately, there is a danger that an

error may occur before the write is committed to the second I/O server. Early client termination also makes it difficult to guarantee data consistency in many failure scenarios.

The primary-copy technique simplifies resiliency and consistency for parallel file systems because the client does not have to deal with inconsistency issues between the primary and replicated data stores. Instead, the primary server and secondary are responsible for determining their own consistency status using server to server communication. In the traditional primary-copy technique, all client update requests are directed to the primary server, which forwards the request to the backup. If the backup successfully completes the update, it sends an acknowledgment to the primary, which then also performs the update and signals success to the client. We have modified the protocol to allow the primary server to simultaneously commit data to the disk while forwarding it to the secondary server. The coherence protocol described below ensures that identical copies of the data are available during node failures.

C. Replicated File System Data Consistency

PVFS provides read and write semantics suitable for the access and performance requirements of scientific applications. Due to the performance demands placed upon parallel file systems, PVFS provides data coherence semantics rather than sequential consistency. In both sequential consistency and coherence schemes, no concurrent read serialization is performed. Coherence provides more parallelism because all write requests can proceed simultaneously; however, it poses several difficulties for a replicated file system. Our protocol is implemented in the file system rather than at the device level and we cannot guarantee which data will be written to a given byte for two concurrent and overlapping writes. Because of this, the write data may be committed to disk in one order for the primary data object, and in a different order for the secondary data object.

Out-of-sync primary and secondary servers are likely to lead to errors in applications and difficulties for application developers. We have chosen what we believe is the simplest scheme to achieve sequential consistency for simultaneous overlapping writes: we serialize overlapping writes in each server's request processing queue. The serialization only occurs on the primary server where the overlapping writes occur, on all other storage nodes the write requests proceed in parallel as expected.

D. Ensuring Consistency During Failures

In a cluster interconnection network where temporary outages may occur, the primary or secondary server may be unavailable during a write, but become available again during a later read from the written region. The client then must determine whether the primary or secondary data is the most up-to-date. This issue is a special case of the classic Byzantine generals problem in computer science [6].

In order for primary and secondary servers to determine their relative consistency status, we add a variable indicating the consistency state to each data object on an I/O node. When a file is created, a data object, or file storage area, is setup on each I/O node. At creation time, we initialize the consistency status to the *consistent* state, indicating that the data object is consistent with its mirror data object (i.e. both objects are empty and therefore consistent). Before any change (i.e. a file write) is performed to the data object, the consistency state must be checked against the secondary to determine if normal operation can proceed or if failover must occur. Similarly, before any read completes, the consistency states must be checked against the mirror copy to ensure that both objects are mutually consistent.

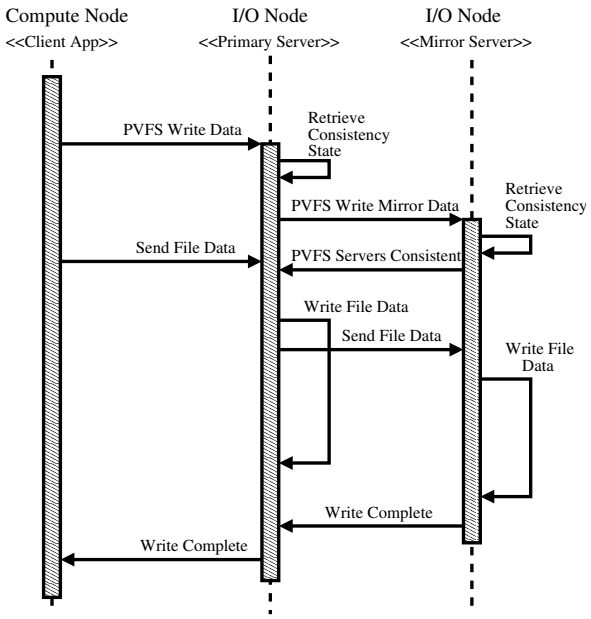

Fig. 4. Sequence Diagram for Writes

1) Consistency During File Writes: Figure 4 shows a sequence diagram documenting the interaction between the primary server and secondary server when their respective data objects are mutually consistent during a file write. The primary server must first ensure that both it and the secondary server are in the *consistent* state before the write can be committed to the repository. If the state values indicate that either the primary data object or the secondary data object are not consistent, then a fail-over protocol will be activated.

2) Consistency During File Reads: Figure 5 shows a sequence diagram documenting the interaction between the primary and secondary servers for a successfully read. In this diagram, the objects are consistent and the read is successful. Though the consistency status must be checked on all reads, there is no requirement that clients read from only the primary server. Instead, clients may read from both the primary and secondary servers simultaneously, with each server checking its relative consistency status on every read. The servers must ensure that the repositories are consistent before completing the read; however, it is possible to begin returning read data before the consistency check has completed.

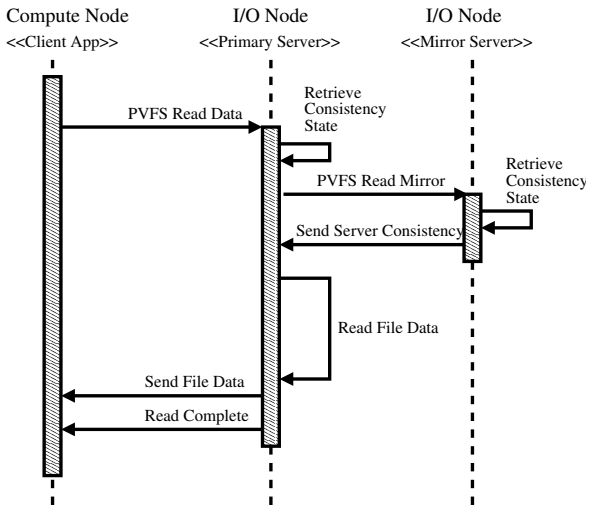

Fig. 5. Sequence Diagram for Reads

E. Failure Scenarios

In the case of a failure for a primary or secondary server, the remaining server will enter the *standalone* state for the data object. The standalone state ensures that once a server fails, the failed server cannot service any future requests for that data object until recovery is performed. No checking with the mirror server is performed by the standalone server, instead the standalone copy of the data object is guaranteed consistent.

In order to enter standalone state, an I/O server must request a vote with a majority of the active servers agreeing that the mirror has not already entered standalone mode. Once one server has entered the standalone state, the quorum that participated in that vote will not allow the other server to enter the standalone state until a recovery to normal operation has occurred. A complete description of the failure protocol for all possible failure scenarios is provided in [8].

1) Voting and Quorums: Establishing a quorum and voting on which repository holds the most up-to-date data is a technique commonly employed in distributed systems. Due to the additional delay associated with voting, we only employ our voting protocol whenever a failure is first detected. A valid quorum is achieved when more than half of the servers (a simple majority) send a previously agreed upon response to the requesting server. In the case of our replication mechanism, the possible quorum responses are that no server has entered standalone mode for the failed data object, or that a server has already entered standalone mode for the specified data object.

In the case where a quorum cannot be achieved (i.e. half of the servers cannot be contacted by the primary I/O node or its backup) the client is notified of the file I/O failure and the file is lost until a system administrator can interact with the system to attempt recovery.

IV. Results

To determine the real world effects of our proposed replication scheme we performed a series of experiments using an aggregate I/O bandwidth benchmark. PVFS includes in its distribution a program called mpi_io_test that we use to measure aggregate bandwidth as seen by the client. The test program uses the standard MPI collective, MPI_File_write so that each of the computation nodes simultaneously writes 16MB of local non-contiguous and non-overlapping data to construct a single contiguous file that spans all the I/O servers in the PVFS file system. We measure the write bandwidth as the time it takes for the clients to finish writing all of the data, irrespective of the amount of data committed to disk being doubled due to data replication. Similarly, the program performs a collective MPI_File_read to read the contiguous data file to construct local non-contiguous data arrays on each of the computation nodes. The read bandwidth is measured as the time it takes for the clients to finish receiving a single copy of all of the file data without respect to whether or not the data has been replicated. For these tests all I/O nodes acted as both a primary and secondary server. All reported bandwidths are the average result over 10 identical executions.

Write Bandwidth (MB/Sec)			
Num Clients	Stock PVFS	PVFS w/Replication	Percent Performance
1	10.9	9.8	89.9%
4	33.3	28.7	86.2%
8	54.8	39.8	72.6%
16	83.3	40.3	48.4%
32	77.5	38.2	49.3%

TABLE I
Relative replication performance on Adenine with 16 I/O servers

In order to measure the effects of replication in realistic scenarios we performed our tests on two different clusters: one with performance primarily limited by the network and another cluster with performance primarily limited by the file system speed.

For the network constrained cluster we used Adenine, the Clemson University PARL lab's Beowulf cluster. Adenine is composed of 48 nodes, each running a 2.6 Linux Kernel on dual 1GHz Pentium III processors with 1GB RAM and interconnected with 100Mbit Fast Ethernet. In all of our test configurations we assigned each node to be either a computation node or an I/O node. No more than one client process or PVFS server ran on any of the nodes.

For the file system constrained cluster we used Argonne National Laboratory's Jazz cluster. Jazz is composed of approximately 250 nodes, each running a 2.4 Linux Kernel on a single 2.4GHz Pentium IV with 1GB RAM and connected with a Myricomm 2000 interconnection network. The Myrinet interconnect uses cut through switching for low latency and can achieve a sustained bandwidth of approximately 1.2Gbit.

A. Write Performance

The performance impact of replication on write bandwidth is primarily governed by two constraints. On the one hand,

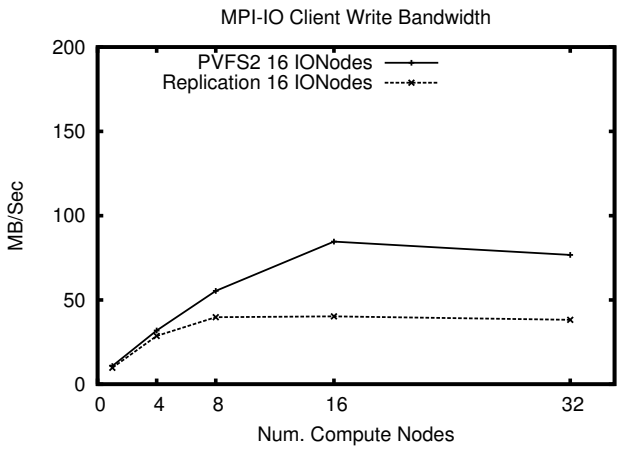

Fig. 6. Performance on Adenine w/ 16 IO Nodes

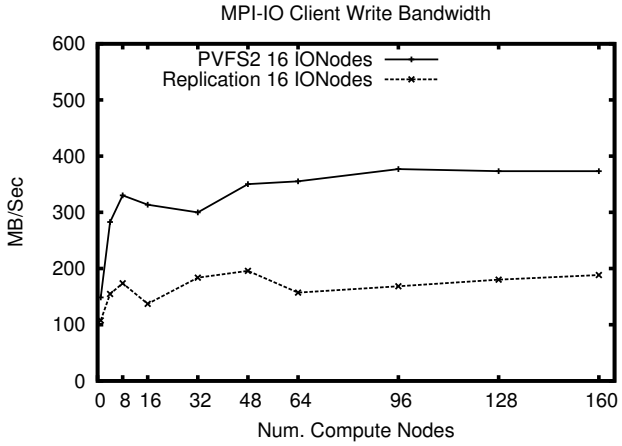

Fig. 7. Performance on Jazz w/ 16 IO Nodes

we expect that the cost of writing twice as much data over the network and to disk will double the time to complete a given write. On the other hand, the use of pipelining may allow us to more effectively utilize the network and disk, thus mitigating some of the additional bandwidth requirements when the system is lightly loaded. Figure 6 and table I show that as the number of clients is increased (i.e. the system load is increased) the write bandwidth of the replicated file system is reduced to 50% of the normal write bandwidth.

Evaluating the performance of replication on the Adenine cluster can be difficult because the network bandwidth quickly becomes the bottleneck even while using a relatively small number of clients. The Jazz computational cluster uses a high performance interconnection network based on cut through switching that alleviates the network bottleneck during writes. Instead, the bottleneck becomes how fast the operating system can commit data to the file system, which is much faster than data can be written to disk. Table II and Figure 7 show a 16 I/O node file system configuration on Jazz with up to 128 clients. In general, as the load increases the replicated

Write Bandwidth (MB/Sec)			
Num Clients	Stock PVFS	PVFS w/Replication	Percent Performance
1	145.7	107.6	73.9%
4	269.8	154.6	57.3%
8	305.2	173.7	56.9%
16	287.7	137.1	47.7%
32	312.2	183.7	58.8%
64	363.9	157.1	43.2%
128	367.4	180.1	49.0%

TABLE II
RELATIVE REPLICATION PERFORMANCE ON JAZZ WITH 16 I/O SERVERS

Write Bandwidth (MB/Sec)			
Num Clients	PVFS 8 I/O Nodes	PVFS w/Rep. 16 I/O nodes	Percent Performance
1	10.3	9.8	95.1%
4	28.2	28.7	101.8%
8	43.5	39.8	91.5%
16	43.4	40.3	92.9%
32	50.1	38.2	76.2%

TABLE III
PERFORMANCE OF A REPLICATED AND NON-REPLICATED FILE SYSTEM ON ADENINE

file system's available client bandwidth is half the available non-replicated bandwidth. Unfortunately, there is significant noise in the data making it difficult to draw exact conclusions. The low latency interconnection network used on Jazz is hierarchical, and communication between any two nodes may experience congestion and additional routing time between some destinations. It is difficult to control for this behavior, particularly on large jobs that are guaranteed to traverse the network switch hierarchy.

Even with the data variance we can see that the knee of the curve has been reached on client sizes larger than 32 nodes. The bandwidth curves are effectively parallel at this point with a replicated file system providing 50% of the available bandwidth in a non-fault tolerant file system.

Table III compares the performance of an 8 I/O node non-replicated file system with the performance of a 16 I/O node fault tolerant file system. At all configurations less than 32 clients, the file systems provide very similar write bandwidths. At 32 clients the performance of the replicated file system falls to only 72.6% of the non-fault tolerant file system.

B. Read Performance

Figure 8 indicates that on a network bound cluster the read performance with our replication protocol is effectively identical to a parallel file system with no fault tolerance. We see some degree of noise in the read bandwidth on Jazz, Figure 9, due to the shared cut-through switching network; however, read performance is mostly identical for a fault

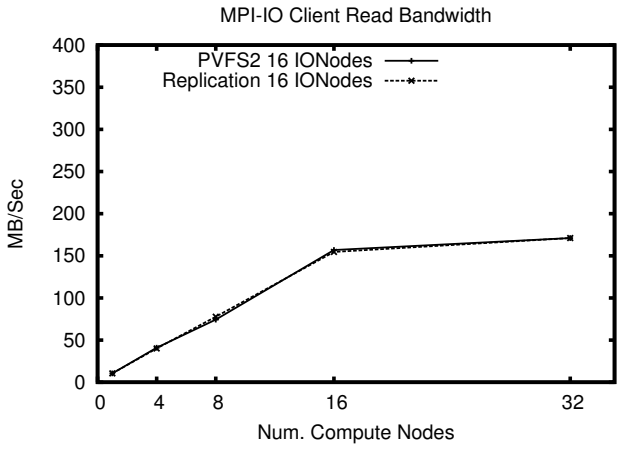

Fig. 8. Performance on Adenine w/ 16 IO Nodes

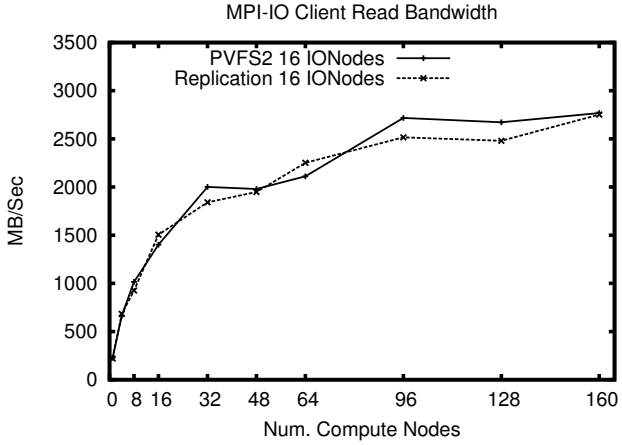

Fig. 9. Performance on Jazz w/ 16 IO Nodes

tolerant and non-fault tolerant parallel file system on a disk bound cluster as well. The additional overhead of consistency checks is inconsequential and the file system scalability is virtually the same.

V. Discussion

We have described a technique for improving data resilience in parallel file systems by replicating data across storage nodes. Replication is able to tolerate up to $1/2$ the I/O nodes failing with an expectation that $1/4$ of the servers can fail before the file system can no longer reconstruct a file. We also described how consistency is maintained using additional state attributes stored with each data object.

Our benchmarking shows that on a lightly loaded parallel file system the performance impacts on write bandwidth may be much less than a 50% degradation, and on a heavily loaded I/O system the available client write bandwidth is very close to 50% of the non-replicated file systems bandwidth with an overhead generally less than 3%. In the case where performance is critical, our results indicate that doubling the number of storage nodes in the cluster will achieve similar performance to the non-redundant file system.

One expected problem with our approach is the performance during a storage node failure. We expect that the read and write bandwidths of the file system would fall by 50% with only a single failure (though further failures would not further reduce performance) as the node in standalone mode would become the file access bottleneck. One possible solution is to alter the replicated data distribution so that the secondary server for each strip, rather than each data object, would be mirrored to servers in a round robin fashion. Such a data distribution could lead to scalable performance from a file system experiencing storage node failures without altering the performance in a don-degraded file system. Alternatively, on large file systems it may instead be advantageous to simply select subsets of the available storage nodes for data object placement, thus reducing the probability of a file being affected by the failure of a single storage node. Finally, we may wish to combine both approaches to enhance both performance and reliability.

Acknowledgment

We would like to thank Dr. Philip Carns and the PVFS development team for their technical assistance in developing our prototype. We also gratefully acknowledge use of the "Jazz" cluster operated by the Mathematics and Computer Science Division at Argonne National Laboratory as part of its Laboratory Computing Resource Center.

References

[1] Peter A. Alsberg and John D. Day, *A principle for resilient sharing of distributed resources*, Proceedings of the 2nd international conference on Software engineering, IEEE Computer Society Press, 1976, pp. 562–570.

[2] Remzi H. Arpaci-Dusseau, Eric Anderson, Noah Treuhaft, David E. Culler, Joseph M. Hellerstein, David Patterson, and Kathy Yelick, *Cluster i/o with river: making the fast case common*, Proceedings of the sixth workshop on I/O in parallel and distributed systems, ACM Press, 1999, pp. 10–22.

[3] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung, *The Google file system*, Proceedings of the nineteenth ACM symposium on Operating systems principles, ACM Press, 2003, pp. 29–43.

[4] Hui-I Hsiao and David DeWitt, *Chained Declustering: A new availability strategy for multiprocessor database machines*, Proceedings of 6th International Data Engineering Conference, 1990, pp. 456–465.

[5] Paris C. Kanellakis and Christos H. Papadimitriou, *Is distributed locking harder?*, PODS '82: Proceedings of the 1st ACM SIGACT-SIGMOD symposium on Principles of database systems (New York, NY, USA), ACM Press, 1982, pp. 98–107.

[6] Lamport, Shostak, and Pease, *The byzantine generals problem*, Advances in Ultra-Dependable Distributed Systems, N. Suri, C. J. Walter, and M. M. Hugue (Eds.), IEEE Computer Society Press, 1995.

[7] Frank Schmuck and Roger Haskin, *GPFS: A shared-disk file system for large computing clusters*, Proc. of the First Conference on File and Storage Technologies (FAST), January 2002, pp. 231–244.

[8] Bradley W. Settlemyer, *A mechanism for scalable redundancy in parallel file systems*, Master's thesis, Clemson University, Clemson, SC, May 2006.

[9] Chandramohan A. Thekkath, Timothy Mann, and Edward K. Lee, *Frangipani: A scalable distributed file system*, Symposium on Operating Systems Principles, 1997, pp. 224–237.

[10] Sage A. Weil, Andrew W. Leung, Scott A. Brandt, and Carlos Maltzahn, *Rados: A fast, scalable, and reliable storage service for petabyte-scale storage clusters*, Petascale Data Storage Workshop SC07, 2007.