A High Performance Redundancy Scheme for Cluster File Systems

Manoj Pillai, Mario Lauria*

Department of Computer and Information Science

The Ohio State University

2015 Neil Ave #395

Columbus, OH 43210, USA

{pillai, lauria}@cis.ohio-state.edu

Abstract

Striped file systems such as the Parallel Virtual File System (PVFS) deliver high-bandwidth I/O to applications running on clusters. An open problem of existing striped file systems is their vulnerability to disk failures. In this paper we describe a novel redundancy scheme designed to maximize performance delivered to the applications. By dynamically switching between RAID1 and RAID5 redundancy based on write size, our hybrid scheme consistently achieves the best of two worlds - RAID1 performance on small writes, and RAID5 efficiency on large writes. We demonstrate the hybrid redundancy within CSAR, a proof-of-concept implementation based on PVFS, and test its performance using representative scientific applications. CSAR achieves substantially similar read bandwidth and 73% of the write bandwidth of PVFS over 7 I/O nodes. We describe the design issues encountered in implementing our redundancy scheme in a striped file system like PVFS and our solutions to them.

Keywords: High Performance I/O, Cluster File Systems, PVFS, Distributed RAID, High Availability I/O

*Corresponding Author

1 Introduction

Input/Output has often been identified as the weakest link for parallel computing, particularly on commodity clusters [20]. A number of cluster file systems have been developed in recent years to provide scalable storage in a cluster. The goal of the Parallel Virtual File System (PVFS) project [11] is to provide high-performance I/O in a cluster, and a platform for further research in this area. An implementation of the MPI-IO library over PVFS is available, and has contributed to the popularity of PVFS as a file system for parallel computing on Linux clusters. In PVFS, as in most other cluster file systems, clients directly access storage servers on data transfer operations, providing scalable performance and capacity.

A major limitation of PVFS, however, is that it does not store any redundancy. As a result, a disk crash on any of the many I/O servers will result in data loss. Because of this limitation, PVFS is mostly used as high-bandwidth scratch space; important files have to be stored in a low-bandwidth, general-purpose file system.

The goal of the Cluster Storage with Adaptive Redundancy (CSAR) project is to study issues in redundant data storage in high-bandwidth cluster environments. We believe efficient redundancy is crucial in data-intensive computing environments dealing with very large data sets. Traditional solutions like backup are impractical in these environments because of the bandwidth they consume. A number of previous projects have studied the issue of redundancy in disk-array controllers. However, the problem is significantly different in a distributed cluster file system like PVFS where there is no single point through which all data passes.

We have augmented PVFS with redundancy with the long term objective of making it tolerant of single disk failures. Adding redundancy inevitably reduces the performance seen by clients, because of the overhead of maintaining the extra data; other effects further contribute to the reduction of overall performance. Our primary design objective was to design a redundancy scheme designed from scratch to maximize the overall performance of disk accesses seen by the applications.

We have implemented and studied three redundancy schemes in CSAR. The first scheme is a striped, block-mirroring scheme which is a variation of the RAID1 and RAID10 schemes used in disk controllers. In this scheme the total number of bytes stored is always twice the amount stored by PVFS. The second scheme is a RAID5-like scheme, where parity is used to reduce the number of bytes needed for redundancy. In addition to adapting these well-known

schemes to the PVFS architecture, we have designed a hybrid scheme that selects the appropriate reliabilty level on the fly. In our scheme RAID5 style (parity-based) is selected for large write accesses, and mirroring for small writes. The goal of the Hybrid scheme is to provide the best of the other two schemes by adapting dynamically to the istantaneous workload presented by the application.

A number of issues needed to be solved in implementing these redundancy schemes in PVFS. One of these issues is the change in consistency semantics when a RAID5-like redundancy is added, where the parity introduces a data dependency between I/O servers in a system like PVFS that was designed to avoid such dependencies. We also report on some previously poorly documented performance issues arising at the interface between PVFS and the underlying file system. These issues are due to unwanted side effects of partial block writes to disk, and are observed in conjunction with i) non-blocking network receives preceding disk writes, and ii) unaligned writes to disk of uncached pre-existing files.

The paper is structured as follows. Section 2 describes the advantages and disadvantages of the RAID1 and RAID5 schemes, and provides the motivation for the Hybrid scheme. Section 3 describes related work. Section 4 gives an overview of the PVFS implementation, and the changes we made in order to implement each of our redundancy schemes. Section 5 describes experimental results. Section 6 provides our conclusions.

2 Motivation

As clusters are becoming an increasingly popular platforms for high end parallel computing, the need for efficient, scalable and inexpensive I/O becomes more and more pressing. Striped file systems satisfy these requirements, and a number of research projects have explored different aspects of this type of design. For example the main focus of the PVFS project has been how to deliver scalable performance to the applications, either through a standard file system interface or a popular I/O library such as MPI-IO [3].

One common issue of striped file systems is their vulnerability to faults; having a file striped over several disks allows high read/write bandwidth, but it also increases the likelihood of failures. Some striped file system projects such as Zebra [7], xFS [1] and Swarm [8] have incorporated data redundancy in their designs from the start.

Distributed redundancy schemes have been extensively studied in connection with striped file systems; table 2

compares the main features of some recent projects in the area. The application of RAID-like redundancy to striped file systems can be seen as a natural extensions of the original, disk-array based RAID design. However the distributed nature of these file systems and the lack of a single controller introduces new design issues not present on disk arrays, such as the consistency of data and parity in a stripe.

Our work differs from previous projects in several ways:

1. Our emphasis is on absolute performance (as bandwidth seen by the applications) rather than storage overhead; we have allowed our design to be a little more wasteful in terms of storage whenever that could be translated in higher performance.

2. We strive to improve bandwidth for the whole range of access sizes, from small sizes (problematic in RAID5) to large disk accesses (bandwidth wasteful in RAID1).

3. We don't make any assumption on the underlying file system functionality in terms of locks, aggregation of small writes (like in log-based file systems), or other. To emphasize this concept we demonstrate our scheme on a striped file system that was not designed to include redundancy.

4. We are interested in working with commodity systems, and therefore we have studied the interactions of our scheme with an unmodified, stock operating system such as Linux.

Most of the distributed redundancy solutions introduced so far are based on variants of the basic RAID schemes [4]. Each one of these schemes represent different tradeoffs between storage overhead, bandwidth requirements, and fault tolerance. In the RAID1 configuration, two copies of each block are stored on different disks. This configuration has fixed high storage and bandwidth overheads, independent of the number of disks in the array. In the RAID5 configuration, storage is organized into stripes. Each stripe consists of n blocks, one block per disk. One of the blocks in the stripe stores the parity of all other blocks. Thus, for a disk-array with n disks, the storage (and bandwidth) overhead in RAID5 is just $1/n$.

The major problem with the RAID5 configuration is its performance for a workload consisting of small writes, i.e. disk accesses that modify only a portion of a stripe. For such a write, RAID5 needs to perform the following sequence of operations: 1) read the old version of the data being updated and the old parity for it, 2) compute the new parity, 3)

Table 1: Comparison between some research projects on parallel and distributed RAIDs

System Attributes	**USC RAID-x**	**Digital Petal**	**Berkeley Tertiary Disk**	**OSU CSAR**
RAID architecture	Orthogonal Striping and mirroring over a Linux cluster	Chained declustering in a Unix cluster	A RAID5 built with a Solaris PC cluster	Striping and dynamic redundancy over a Linux cluster
Data consistency maintenance	Lock at device driver level	Frangipani file system	Lock in the xFS file system	Sequencing of user requests
Reliability implementation	Orthogonal striping and mirroring	Striped mirroring	SCSI disks with parity checks	Dynamic redundancy scheme

write out the new data and new parity. The latency of a small write is quite high in RAID5 and disk utilization is poor because of the extra reads.

A number of variations have been proposed to the basic RAID5 scheme that attempt to solve the performance problem of RAID5 for small writes. Previous work has focused on retaining the low storage overhead of RAID5 compared to RAID1. Our starting point is a high-performance, cluster file system intended primarily for parallel application. In combining the small write performance of RAID1 with the bandwidth parsimony of RAID5, our approach represents a new point in the design space of distributed redundancy. In rethinking traditional design priorities, we optimized performance seen by the applications at all access sizes at the expense of storage efficiency.

Current technological trends support an approach that privileges bandwidth over storage efficiency. Over the last decade, processor performance and disk capacity have been growing at a rate of approximately 1.6x/year; RAM capacity has grown approximately at 1.4x/year. However, the various components of the physical data path (I/O bus, disk bus, disk bandwidth) have grown at a smaller rate. The bandwidth of the PCI bus has gone from the 132 MB/s of its initial implementations to the current 528 MB/s, for a overall 1.2x/year growth rate. The SCSI bus has grown from 5MB/s to 160MB/s over the course of fifteen years, resulting in a growth rate of 1.25x/year; a comparable rate has been observed for the internal disk bandwidth. The practical consequence of this disparity can be seen in Figure 1, drawn using historical data from [5]: the ratio of disk size over disk bandwidth (shown as the time required to fill a disk to capacity) has grown tenfold over the last fifteen years.

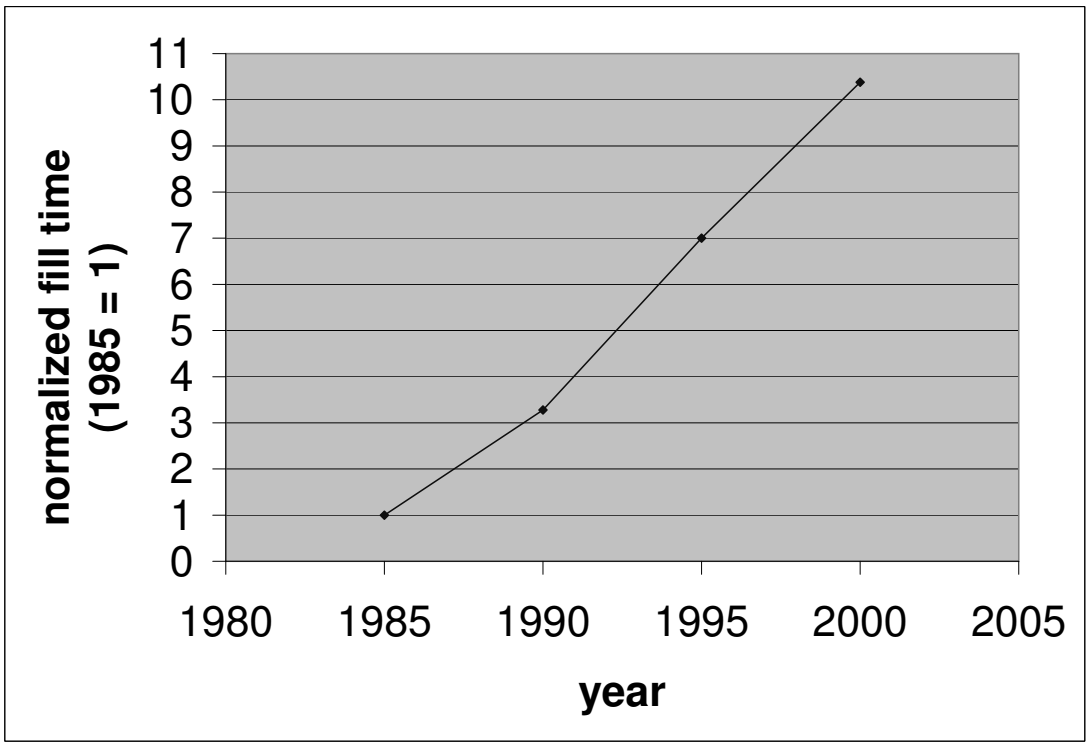


Figure 1: The time to fill a disk to capacity over the years

3 Related Work

The distributed RAID concept was explored by Stonebraker and Schloss [18]. Prototyping of distributed RAIDs started with the Petal project [10] at Digital Lab and the Tertiary Disk project [19] at UC Berkeley. A comparison with our approach with these works is shown in Table 2.

Swift/RAID [12] was an early distributed RAID implementation. It implemented RAID levels 0, 4 and 5. In the Swift/RAID implementation, RAID5 obtained only about 50% of the RAID0 performance on writes; RAID4 was worse. Our implementation performs much better relative to PVFS. We experienced many of the same design issues reported in the Swift paper. For example, computing parity one word at a time instead of one byte at a time significantly improved the performance of the RAID5 and Hybrid schemes. The Swift/RAID project did not implement any variation of the basic RAID levels similar to our Hybrid scheme.

Zebra [7], xFS [1] and Swarm [8] are distributed storage systems with multiple storage servers that store data using RAID5 redundancy. All of these systems combine striping with log-structured writes to solve the small-write problem of RAID5. As a result, they suffer from the garbage collection overhead inherent in a log-structured systems [16]. We use a different storage scheme in order to perform well for a larger range of workloads. Note that our overflow regions

are conceptually similar to logs used in log-structured systems since data is not overwritten. However our scheme does not require a log-structured file system, nor it relies on a large number of small writes being batched into a larger one.

The RAID-x architecture [9] is a distributed RAID scheme that uses a mirroring technique. To improve performance, RAID-x delays the write of redundancy, and by employs a storage layout that allows mirrored writes to be batched into large disk accesess. Delaying the write improves the latency of the write operation, but it need not improve the throughput of the system. For applications that need high, sustained bandwidth, RAID-x suffers from the limitation of mirroring. Also, since their scheme delays the writing of redundancy, it does not provide the same level of fault-tolerance as the schemes discussed here.

Our work is similar to HP AutoRAID [21], parity logging [17] and data logging [6] in that it addresses the small write problem of RAID5 by using extra storage space. However, ours is a distributed RAID solution whereas these are meant for centralized storage controllers. There are also differences in the assumptions about locality. For example, AutoRAID uses RAID1 for hot data (write-active data) and RAID5 for cold data, and maintains metadata in the controller's non-volatile memory to keep track of the current location of blocks. It promotes blocks that become write-active to the RAID1 region on a large block granularity to preserve the spatial locality of reference. In our scheme only partial stripe writes are written using RAID1, and a later full stripe write automatically moves this data back to RAID5.

4 Implementation

In this section we will give an overview of the PVFS design and a brief description of the Hybrid scheme. Further details on the Hybrid scheme can be found in [15].

PVFS Overview PVFS is designed as a client-server system with multiple I/O servers to handle storage of file data. There is also a manager process that maintains metadata for PVFS files and handles operations such as file creation. Each PVFS file is striped across the I/O servers. Applications can access PVFS files either using the PVFS library or by mounting the PVFS file system. When an application on a client opens a PVFS file, the client contacts the manager and obtains a description of the layout of the file on the I/O servers. To access file data, the client sends requests directly to the I/O servers storing the relevant portions of the file. Each I/O server stores its portion of a PVFS file as a

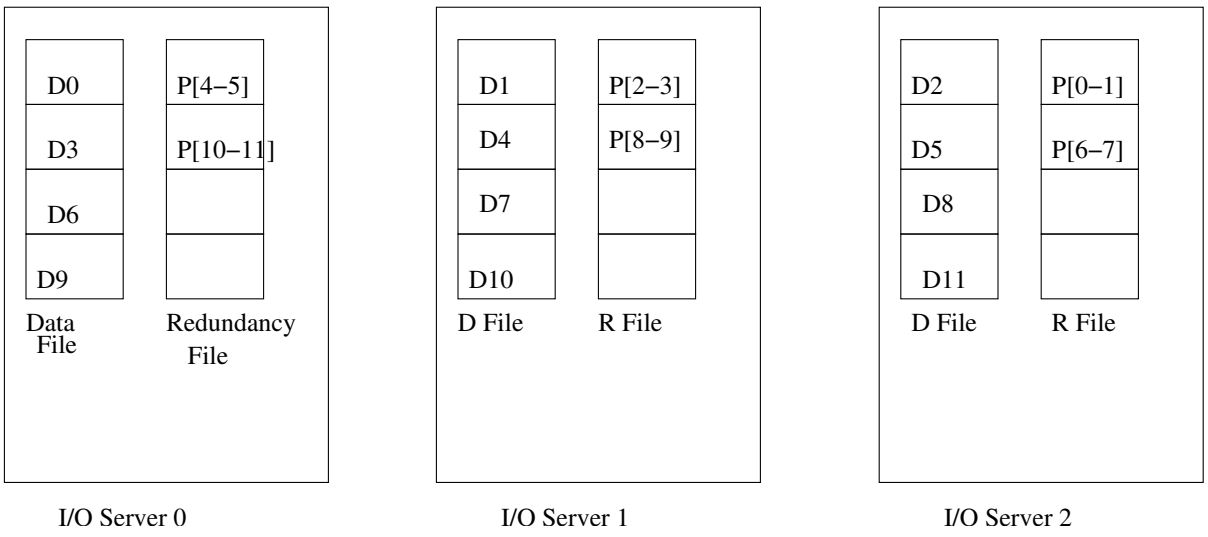


Figure 2: The RAID5 scheme

file on its local file system.

RAID1 Implementation In the RAID1 implementation in CSAR, each I/O daemon maintains two files per client file. One file is used to store the data, just like in PVFS. The other file is used to store redundancy.

RAID5 Implementation Like the RAID1 scheme, our RAID5 scheme also has a redundancy file on each I/O server in addition to the data file. However, the contents of these files contain parity for specific portions of the data files. The layout for our RAID5 scheme is shown in Figure 2. The first block of the redundancy file on I/O server 2 (P[0-1]) stores the parity of the first data block on I/O Server 0 and the first data block on I/O Server 1 (D0 and D1, respectively.) On a write operation, the client checks the offset and size to see if any stripes are about to be updated partially. There can be at most two partially updated stripes in a given write operation. The client reads the data in the partial stripes and also the corresponding parity region. It then computes the parity for the partial and full stripes, and then writes out the new data and new parity.

In both the RAID1 and the RAID5 scheme, the layout of the *data* blocks is identical to the PVFS layout. By designing our redundancy schemes in this manner, we were able to leave much of the original code in PVFS intact and implement redundancy by adding new routines. In both schemes, the expected performance of reads is the same as in PVFS because redundancy is not read during normal operation.

The Hybrid Scheme In the Hybrid scheme, the level of redundancy is selected on the fly for every write access. Different sections of a single access can be written to disk using different forms of redundancy according to the following rule. Every write is broken down into three portions: (1) a partial stripe write at the start, (2) a portion that updates an integral number of full stripes, and (3) a trailing partial write. Depending on data alignment and size, portions (1) and/or (3) can be empty. For the portion of the write that updates full stripes, we compute and write the parity, just like in the RAID5 case. For the portions involving partial stripe writes, we write the data and redundancy like in the RAID1 case, except that the updated blocks are written to an overflow region on the I/O servers. The blocks cannot be updated in place because the old blocks are needed to reconstruct the data in the stripe in the event of a crash. When a file is read, the I/O servers return the latest copy of the data which could be in the overflow region.

In addition to the files maintained for the RAID5 scheme, each I/O server in the Hybrid scheme maintains additional files for storing overflow regions, and a table listing the overflow regions for each PVFS file. When a client issues a full-stripe write any data in the overflow region for that stripe is invalidated. The actual storage required by the Hybrid scheme will depend on the access pattern. For workloads with large accesses, the storage and bandwidth requirements will be close to that of the RAID5 scheme. If the workload consists mostly of partial stripe writes, a significant portion of the data will be mirrored in the overflow regions, with storage and bandwidth requirements close to that of RAID1.

5 Implementation issues

In this section we describe two important implementation issues related to distributed redundancy in a commodity environment. The implementation of RAID5 redundancy presents an additional challenge in a distributed environment. In parallel applications, it is common for multiple clients to write disjoint portions of the same file. With the RAID5 scheme, care must be taken to ensure that two clients writing to disjoint portions of the same stripe do not leave the parity for the stripe in an inconsistent state. Hence, an implementation of RAID5 in a cluster file system would need additional synchronization for clients writing partial stripes. RAID1 does not have a similar issue because there is no dependency between the redundancy data of two separate blocks. Another implementation issues we had to solve deal with the interaction of PVFS with the underlying system software. An interesting lesson we learned is that

the aggressive caching performed by contemporary file systems, while effective most of the time, can occasionally produce undesired performance degradation effects.

5.1 Enforcing Consistency for RAID5 Parity

In contrast to a RAID5 disk-array controller, our implementation of the RAID5 scheme in a distributed file system needs a mechanism to ensure the consistency of parity updates when multiple clients are writing to the same stripe. We implemented a simple distributed locking mechanism for this purpose.

When an I/O server receives a read request for a parity block, it knows that a partial stripe update is taking place. If there are no outstanding writes to the stripe, the server sets a lock on the parity block and then returns the data requested by the read. Subsequent read requests for the same parity block are put on a queue associated with the lock. When the I/O server receives a write request for a parity block, it writes the data to the parity file, and then checks if there are any blocked read requests waiting on the block. If there are no blocked requests, it releases the lock; otherwise it wakes up the first blocked request on the queue.

The client checks the offset and size of a write to determine the number of partial stripe writes to be performed (there can be at most 2 in a contiguous write.) If there are two partial stripes involved, the client serializes the reads for the parity blocks, waiting for the read for the lower numbered block to complete before issuing the read for the second block. This ordering of parity block reads avoids deadlocks in the locking protocol.

Note that this locking scheme guarantees the consistency of the parity block when concurrent writes are to non-overlapping regions of a file. If clients write overlapping regions of a file concurrently, the parity block can become inconsistent with the data in the stripe, because the lock does not extend to the data block. While concurrent writes to non-overlapping regions of a file are a very important access pattern in PVFS, the current PVFS approach does not give any meaningful guarantees to overlapping concurrent writes. Indeed, the RAID1 scheme can also leave the mirror in an inconsistent state with respect to the data file if there are overlapping concurrent writes. Our locking scheme lends itself to simple extensions, should PVFS evolve in the direction of stronger consistency guarantees.

We measured the overhead added by the locking scheme using a microbenchmark. Figure 3 shows measured bandwidth with 5 clients writing different blocks of the same stripe (in this case, there are 5 data blocks in one RAID5 stripe.) In this graph and in the following RAID0 refers to the original PVFS performance (i.e. striping, no

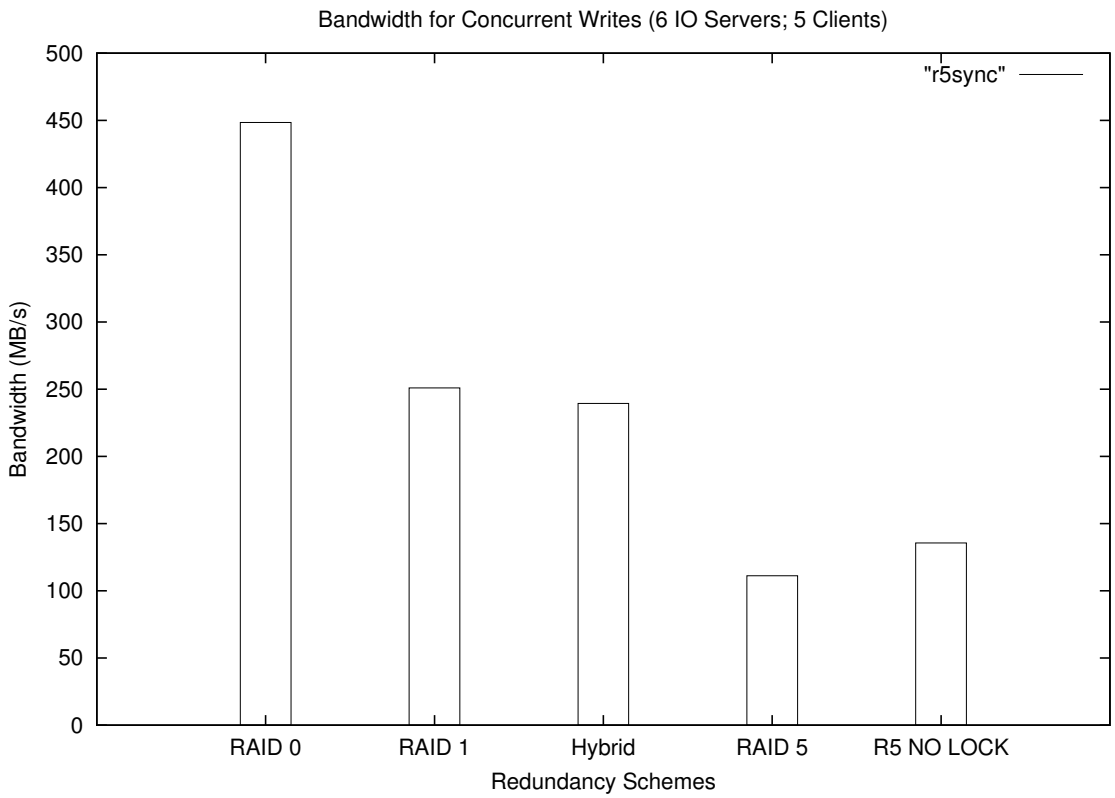


Figure 3: The lock overhead

redundancy). The *R5 NO LOCK* scheme is a RAID5 implementation without the locking code. In this experiment, the *R5 NO LOCK* scheme transfers the same amount of data between the clients and the servers as the RAID5 scheme, but leaves the parity block inconsistent. It can be seen that in this experiment, locking adds about 20% overhead; this includes the cost of the additional lines of code and the and the drop in performance due to serialization.

5.2 Partial Writes to Preexisting Files

In the course of our experiments, we discovered a performance problem in PVFS due to unwanted side effects of partial block writes to disk. The problem manifested itself in the form of degraded write performance when accessing a preexisting file compared to the writing of a new file. This problem is observed in conjunction with i) non-blocking network receives preceding disk writes, and ii) unaligned writes to disk of uncached pre-existing files.

In PVFS, the I/O servers use a non-blocking receive to get available data from a socket when a file write is in progress. The data received is then written immediately to the server's local file. This mode of writing can result in small fractions of file blocks being written. If the block is not in the cache when the write is being performed, this

causes the block to be read from disk into the cache before the write is applied.

To fix this problem, we implemented a write buffering scheme. In this scheme, each write connection on the server is given a small write buffer whose size is a multiple of the local file system block size. Data received from the network is accumulated in this write buffer until the buffer is full or the write is complete. This allows data to be written to file in full blocks if the client access is large enough. At the same time, it allows us to continue to use non-blocking receives, thereby permitting concurrency at the network level. All the experiments described below were conducted with the write buffering scheme.

6 Performance Results

6.1 Experimental Setup

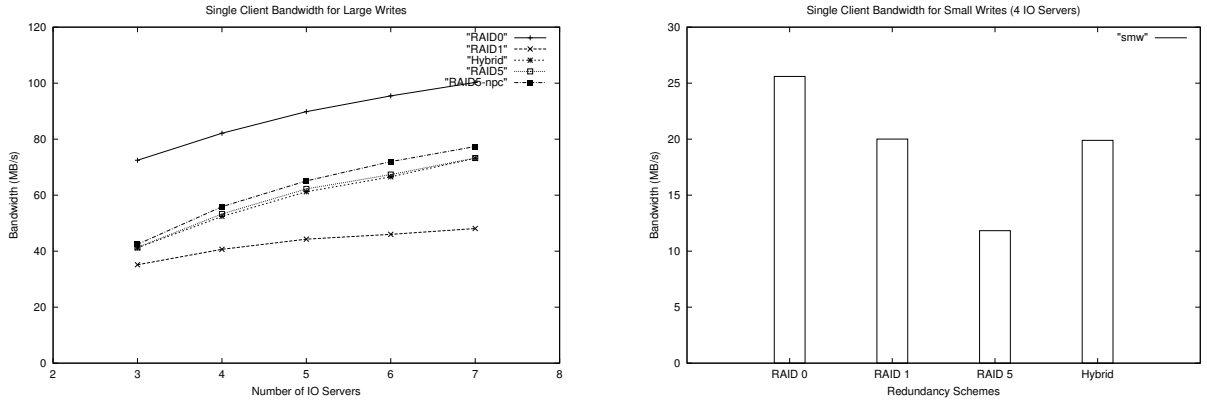
We used two testbeds for our experiments, one consisting of a small experimental cluster with high bandwidth disks, the other of a large production cluster at the Ohio Supercomputer Center.

The first cluster consists of 8 nodes with dual 1GHz Pentium III processors and 1GB of RAM. The nodes are interconnected using a 1.3 Gb/s Myrinet network and using Fast Ethernet. In our experiments, the traffic between the clients and the PVFS I/O servers used Myrinet. Each node in the cluster has two 60GB IBM Deskstar 75GXP disks connected using a 3Ware controller in RAID0 configuration.

The second cluster consists of 74 nodes with dual 900MHz Itanium II processors, 4GB of RAM and a single 80GB SCSI disk, interconnected with a Myrinet network. We used the OSC cluster for all experiments requiring more than eight nodes to run.

6.2 Performance for Full Stripe Writes

We measured the performance of the redundancy schemes with a microbenchmark in which a single client writes large chunks to a number of I/O servers. The write sizes were chosen to be an integral number of the stripe size. This workload represents the best case for a RAID5 scheme. For this workload, the Hybrid scheme has the same behavior as the RAID5 scheme. Figure 4(a) shows that for this workload, RAID1 has the worst performance of all the schemes, with no significant increase in bandwidth beyond 4 I/O servers. This is because RAID1 writes out a larger number of



(a) (b)

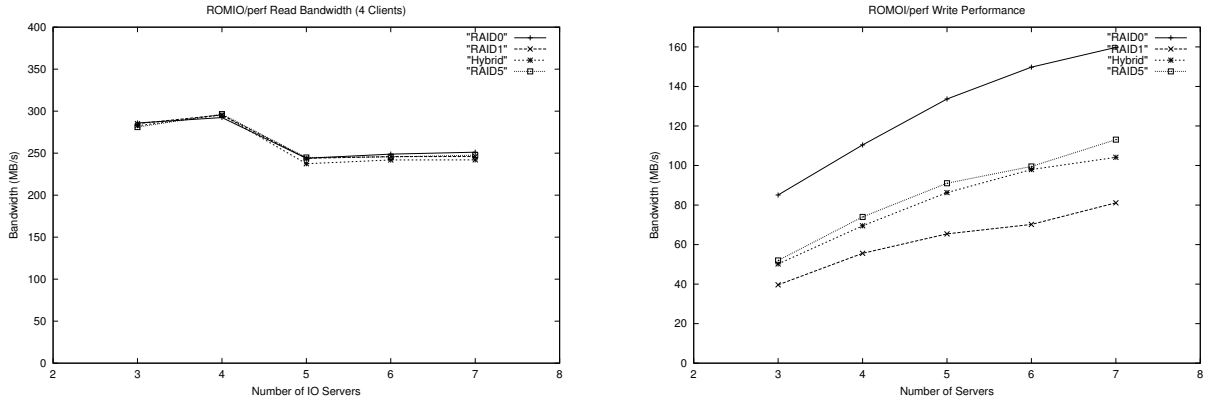
Figure 4: Performance of large (a) and small (b) writes

bytes, and the client network link soon becomes a bottleneck. With only 8 nodes in our cluster, we were not able to go beyond 7 I/O servers. But based on the RAID1 performance, we expect the bandwidth for RAID0 to hit a peak with about 8 I/O servers. We expect the bandwidth for RAID5 and the Hybrid case to continue to rise marginally with increasing number of I/O servers, because of the reduction in the parity overhead.

The *RAID5-npc* graph in Figure 4(a) shows the performance of RAID5 when we commented out the parity computation code. As can be seen, the overhead of parity computation on our system is a modest 8%.

6.3 Performance for Partial Stripe Writes

To measure the performance for small writes, we wrote a microbenchmark where a single client creates a large file and then writes to it in one-block chunks. For this workload, RAID5 has to read the old data and parity for each block, before it can compute the new parity. Both the RAID1 and the Hybrid schemes simply write out two copies of the block. Figure 4(b) shows that the bandwidth observed for the RAID1 and the Hybrid schemes are identical, while the RAID5 bandwidth is lower. In this test, the old data and parity needed by RAID5 are found in the file system cache of the servers. As a result, the performance of RAID5 is much better than it would be if the reads had to go to disk. For larger data sets that do not fit into the server caches, the RAID1 and Hybrid schemes will have a greater advantage over RAID5.



(a) (b)

Figure 5: ROMIO/perf Read (a) and Write (b) Performance

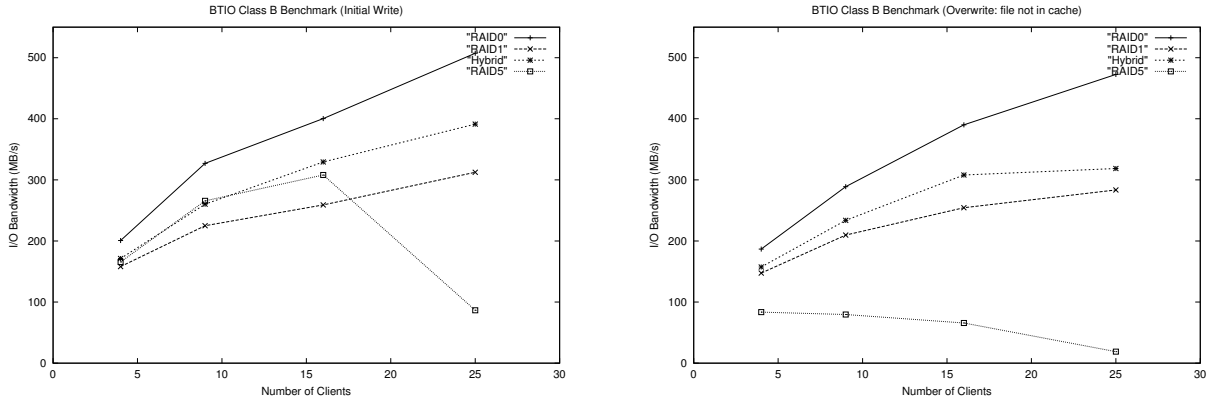
6.4 ROMIO/perf Benchmark Performance

In this section, we compare the performance of the redundancy schemes using the *perf* benchmark included in the ROMIO distribution. *perf* is an MPI program in which clients write concurrently to a single file. Each client writes a large buffer, to an offset in the file which is equal to the rank of the client times the size of the buffer. The write size is 4 MB by default. The benchmark reports the read and write bandwidths, before and after the file is flushed to disk. Here we report only the times after the flush.

Figure 5(a) shows the read performance for the different schemes. All the schemes had similar performance for read. The write performance of the RAID5 and the Hybrid schemes, shown in Figure 5(b) are better than RAID1 in this case because the benchmark consists of large writes.

6.5 BTIO Benchmark

The BTIO benchmark is derived from the BT benchmark of the NAS parallel benchmark suite, developed at NASA Ames Research Center. The BTIO benchmark performs periodic solution checkpointing in parallel for the BT benchmark. In our experiments we used *BTIO-full-mpiio* – the implementation of the benchmark that takes advantage of the collective I/O operations in the MPI-IO standard. We report results for Class B and Class C versions of the benchmark. The Class B version of BTIO outputs a total of about 1600 MB to a single file; Class C outputs about 6600 MB. The BTIO benchmark accesses PVFS through the ROMIO implementation of MPI-IO. ROMIO optimizes small,



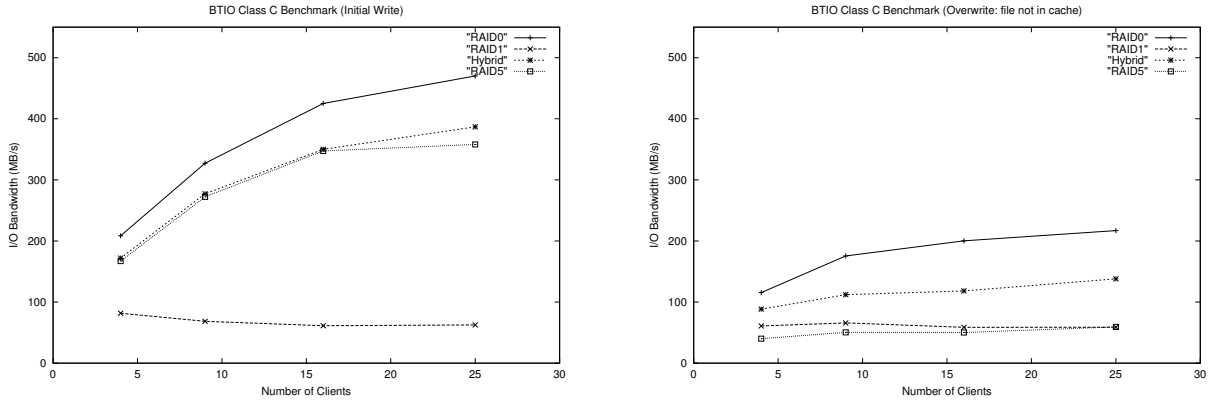
(a) (b)

Figure 6: BTIO - Class B: Initial Write (a) and Overwrite (b) Performance

non-contiguous accesses by merging them into large requests when possible. As a result, for the BTIO benchmark, the PVFS layer sees large writes, most of which are about 4 MB in size. The starting offsets of the writes are not usually aligned with the start of a stripe and each write from the benchmark usually results in one or two partial stripe writes.

The benchmark outputs the write bandwidth for each run. We recorded the write bandwidths for two cases: (1) when the file is created initially. (2) when the file is being overwritten after its contents have been removed from the cache. Figure 6(a) shows the write performance for the Class B benchmark for the initial write; Figure 6(b) shows the write performance for Class B when the file already exists and is being overwritten.

The performance of the Hybrid scheme and RAID-5 in Figure 6(a) are comparable for 4 and 9 processes, both being better than RAID-1. For 16 processes, the performance of RAID-5 drops slightly, and then for 25 processes it drops dramatically. By comparing the reported bandwidth to a version of RAID-5 with no locking (meaning that the parity block could be inconsistent in the presence of concurrent writes), we were able to determine that most of the drop in RAID-5 performance is due to the synchronization overhead of RAID-5. When the output file exists and is not cached in memory at the I/O servers, the write bandwidth for RAID-5 drops much below the bandwidths for the other schemes, as seen in Figure 6(b). In this case, partial stripe writes result in the old data and parity being read from disk, causing the write bandwidth to drop. There is a slight drop in the write bandwidth for the other schemes. This drop results because the alignment of the writes in the benchmark results in some partial block updates at the servers. To verify this we artificially padded all partial block writes at the I/O servers so that only full blocks were written. For the RAID-0, RAID-1 and Hybrid case, this change resulted in about the same bandwidth for the initial write and the



(a) (b)

Figure 7: BTIO - Class C: Write (a) and Overwrite (b) Performance

overwrite cases; for RAID-5, padding the partial block writes did not have any effect on the bandwidth (results not shown). The reason is that the pre-read of the data and parity for partial stripe writes brings these portions of the file into the cache. As a result when the partial block writes arrive, the affected portions are already in memory.

The performance of the schemes for the BTIO Class C benchmark is shown in Figure 7(a) for the initial write case and in Figure 7(b) for the overwrite case. The effect of the locking overhead in RAID-5 is less significant for this benchmark. The performance of RAID-1 is seen to be much lower than the other two redundancy schemes. The reason is that the caches on the I/O servers start to overflow in the RAID-1 scheme because of the large amount of data written – the Class C benchmark writes about 6600 MB of data, and the amount of data written to the I/O servers is twice that in the RAID-1 scheme. For the case when the file is overwritten, the big drop in bandwidth for the RAID-5 scheme is seen in this benchmark also. For the overwrite case, the bandwidth for Hybrid is about 230% of the other two redundancy schemes.

6.6 Application Performance

In this section we present the performance of the various scheme using representative scientific applications and applications kernels.

The FLASH I/O benchmark contains the I/O portion of the ASCI FLASH benchmark [13]. It recreates the primary data structures in FLASH and writes a checkpoint file, a plotfile with centered data and a plotfile with corner data. The benchmark uses the HDF5 parallel library to write out the data; HDF5 is implemented on top of MPI-IO, that in our

experiments was set to use the PVFS device interface. At the PVFS level, we see mostly small and medium size write requests ranging from a few kilobytes to a few hundred kilobytes.

Cactus [2] is an open source modular application designed for scientists and engineers. The name Cactus comes from the design of a central core (or "flesh") which connects to application modules (or "thorns") through an extensible interface. Thorns can implement custom developed scientific or engineering applications, such as computational fluid dynamics. For our experiments we used a thorn called BenchIO, a benchmark application that measures the speed at which large amounts of data (e.g. for checkpointing) can be written using different IO methods. We ran the application on eight nodes and we configured it so that each node was writing approximately 400MB of data to a checkpoint file in chunks of 4MB. Cactus/BenchIO uses the HDF5 library to perform I/O.

Hartree-Fock [14] is a code for the simulation of the non-relativistic interactions between atomic nuclei and electrons, allowing the computation of properties such as bond strengths and reaction energies. Three distinct executables comprise the calculation: *setup* initializes data files from input parameters, *argos* computes and writes integral corresponding to the atomic configuration, and *scf* iteratively solves the self-consistent field equations. The number reported for this experiments correspond to the execution time of *argos*, the most I/O intensive of the three; it writes about 150MB of data, with most write requests of size 16K. In this experiment Hartree-Fock was configured to run as a sequential application, accessing the PVFS file system through the PVFS kernel module. For BTIO, we have used the Class B version and we ran it on eight nodes, like the other applications. The BTIO benchmarks generate fairly large write accesses (a few megabytes) at the PVFS layer.

Figure 8 shows the total output time for the four applications for different redundancy schemes. Execution times are normalized with respect to the original PVFS (RAID0) execution time. Overall, the Hybrid scheme performs comparably or better than the best among RAID1 and RAID5, confirming the results of the experiments with benchmarks. The only exception is Hartree-Fock, for which however the four execution times are not significantly different (they are within 5result to the leveling effect of the significant overhead of small disk accesses through the kernel module.

6.7 Storage Requirement

We measured the storage requirement of the different redundancy schemes using the same applications described in the previous section. Table 2 shows the sum of the file sizes at the I/O servers for each redundancy scheme. As can be

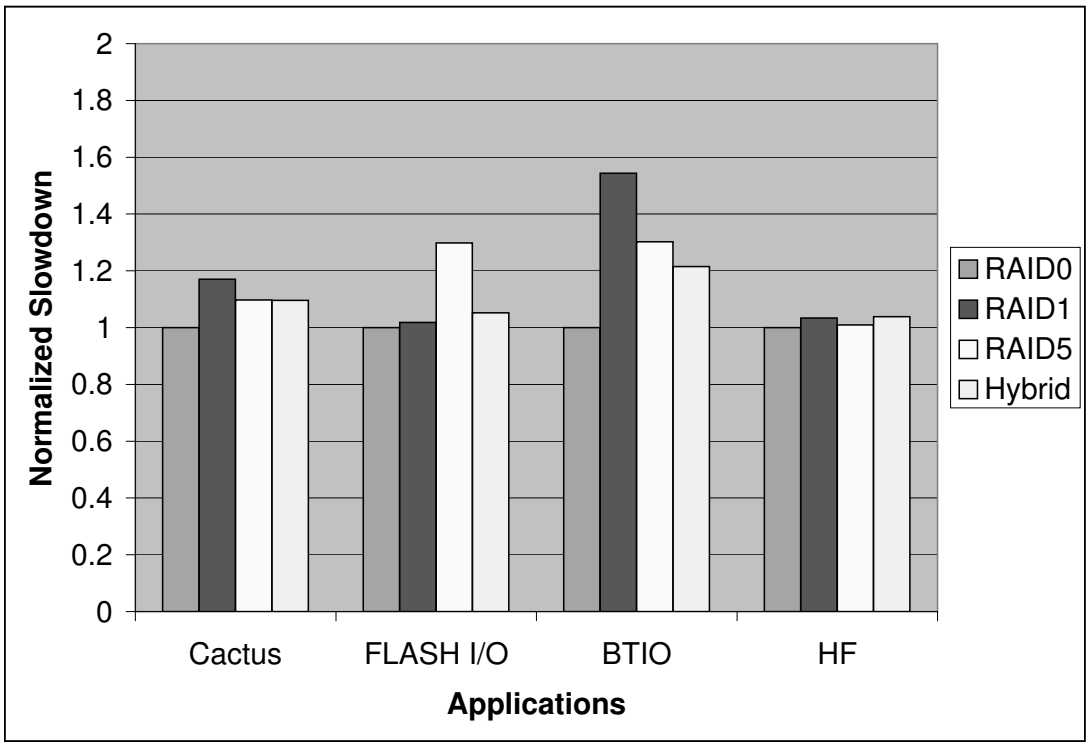


Figure 8: Application Performance

seen, that the amount of extra storage for the Hybrid scheme is highly application dependent.

For these benchmarks, the storage used by the Hybrid scheme is generally close to RAID5, and much less than RAID1. The exceptions are those applications where most of the write accesses are small compared to the stripe size. The FLASH I/O benchmark is such an application, in that it generates a large number of small requests.

For FLASH I/O with 4 processes 46% of the requests were less than 2KB is size. For FLASH I/O with 24 processes, 37% of the requests were less than 2KB in size. The rest of the requests were in the 100KB-300KB range. We show results using two different stripe unit sizes, 16KB and 64KB. With a 64KB stripe unit, there are only a few full stripe writes in the benchmark. For the 64KB stripe unit results, the Hybrid scheme had a larger storage requirement than RAID1. For the 16KB cases, the Hybrid scheme needed less storage. One reason for this is that a smaller stripe unit results in a larger number of full stripes, for which only partial redundancy (parity) is stored. In addition, a smaller stripe unit results in less fragmentation in the overflow regions.

While we are not concerned with storage efficiency at this point of the project, it is conceivable that our scheme could be modified to reduce storage requirement. For example the storage used for overflow regions could be recovered by implementing a simple process that reads files in their entirety and writes them in a large chunk. Taking advantage of

Table 2: Storage Requirement for Redundancy Schemes

Benchmark	**RAID0**	**RAID1**	**RAID5**	**Hybrid**
BTIO Class A	419 MB	838 MB	503 MB	503 MB
BTIO Class B	1698 MB	3396 MB	2037 MB	2353 MB
BTIO Class C	6802 MB	13605 MB	8163 MB	9311 MB
FLASH (4 proc, 16K stripe unit)	45 MB	90 MB	54 MB	74 MB
FLASH (4 proc, 64K stripe unit)	45 MB	90 MB	54 MB	107 MB
FLASH (24 proc, 16K stripe unit)	235 MB	470 MB	282 MB	403 MB
FLASH (24 proc, 64K stripe unit)	235 MB	470 MB	282 MB	646 MB
Hartree-Fock	149 MB	298 MB	179 MB	299 MB
CACTUS/BenchIO	2949 MB	5899 MB	3540 MB	4011 MB

the frequently reported burstiness of I/O activity in scientific applications, this process could be run in the background and activated when the system is under a low load. With such a mechanism, the long-term storage of the Hybrid scheme would be the same as the RAID5 scheme.

7 Conclusions

In this paper we have described a new scheme for distributed redundancy conceived to maximize the performance seen by the applications. In reassessing traditional design priorities to accommodate current technology trends, we optimized bandwidth across all access sizes in exchange for a reduction in storage efficiency. In our experiments, the Hybrid redundancy scheme performed as well as RAID5 for workloads comprising full stripe writes and as well as RAID1 for small writes. For an important parallel benchmark, the Hybrid scheme substantially outperformed both RAID5 and RAID1.

We demonstrated our approach by building CSAR, a striped file system obtained extending PVFS with Hybrid redundancy. Since our scheme does not assume any supporting functionality from the underlying file system such

as locks or aggregation of small writes, it can be easily applied to other striped file systems. As part of the CSAR implementation, we have proposed a simple locking mechanism that addresses the consistency problem of distributed implementations of RAID5 redundancy.

Our project was carried out using commodity system software. In the course of the CSAR implementation, we solved a previously unreported performance problem due to the interactions of PVFS with a stock operating system such as Linux.

Experiments using real applications were facilitated by the presence of standard interfaces. Having implemented the redundancy scheme as an extension to the PVFS library we were able to reuse programs that use either the PVFS library or the standard MPI-IO interface. PVFS also provides a kernel module that allows the PVFS file system to be mounted as a normal Unix file system. We have presented the results of an application that uses CSAR transparently through this kernel module. The popularity of PVFS as a file system for high performance computing on Linux clusters makes our results immediately available to the research community.

8 Acknowledgements

This work was partially supported by the Ohio Supercomputer Center grants PAS0036-1 and PAS0121-1. We are grateful to Dr. Pete Wyckoff and Troy Baer of OSC for their help in setting up the experiments with the OSC clusters. We would like to thank Dr. Rob Ross of Argonne National Labs, for clarifying many intricate details of the PVFS protocol and for making available the PVFS source to the research community.

References

[1] T. Anderson, M. Dahlin, J. Neefe, D. Patterson, D. Roselli, and R. Young. Serverless network file systems. *ACM Transactions on Computer Systems*, February 1996.

[2] Cactus Team. BenchIO Benchmark. http://www.cactuscode.org/Benchmark/BenchIO.html.

[3] P. H. Carns, W. B. Ligon III, R. B. Ross, and R. Thakur. PVFS: a Parallel File System for Linux Clusters. *Proceedings of the 4th Annual Linux Showcase and Conference*, October 2000.

[4] P. Chen, E. Lee, G. Gibson, R. Katz, and D.Patterson. Raid: Highperformance, reliable secondary storage. *ACM Computing Surveys, Vol.26, No.2, June 1994, pp.145-185*, 1994.

[5] Mike Dahlin. Mike Dahlin's Page on Technology Trends. http://www.cs.utexas.edu/users/dahlin/techTrends/.

[6] Eran Gabber and Henry F. Korth. Data logging: A method for efficient data updates in constantly active raids. *Proc. Fourteenth ICDE*, February 1998.

[7] J. Hartman and J. Ousterhout. The zebra striped network file system. *ACM Transactions on Computer Systems*, August 1995.

[8] John H. Hartman, Ian Murdock, and Tammo Spalink. The swarm scalable storage system. *Proceedings of the 19th International Conference on Distributed Computing Systems*, May 1999.

[9] Kai Hwang, Hai Jin, and Roy Ho. RAID-x: A new distributed disk array for I/O-centric cluster computing. In *Proceedings of the Ninth IEEE International Symposium on High Performance Distributed Computing*, pages 279–287, Pittsburgh, PA, 2000. IEEE Computer Society Press.

[10] Edward K. Lee and Chandramohan A. Thekkath. Petal: Distributed virtual disks. In *Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 84–92, Cambridge, MA, 1996.

[11] Walter B. Ligon and Robert B. Ross. An overview of the parallel virtual file system. *Proceedings of the 1999 Extreme Linux Workshop*, June 1999.

[12] Darrell D. E. Long, Bruce Montague, and Luis-Felipe Cabrera. Swift/RAID: A distributed RAID system. *Computing Systems*, 7(3), Summer 1994.

[13] Michael Zingale. FLASH I/O Benchmark Routine. http://flash.uchicago.edu/~zingale/flash_benchmark_io/.

[14] Pablo Research Group. Hartree-Fock Source Code Distribution. http://www-pablo.cs.uiuc.edu/Project/IO/Applications/HF/hf-src.htm.

[15] Manoj Pillai and Mario Lauria. CSAR: Cluster Storage with Adaptive Concurrency. In *CIS Technical Report OSU-CISRC-3/03-TR15*, 2003. Available at ftp://ftp.cis.ohio-state.edu/pub/tech-report/2003/TR15.pdf.

[16] M. Rosenblum and J. Ousterhout. The design and implementation of a log-structured file system. *ACM Transactions on Computer Systems*, 10(1), February 1992.

[17] D. Stodolsky, M. Holland, W. Courtright, and G.Gibson. Parity logging disk arrays. *ACM Transaction on Computer System, Vol.12 No.3, Aug.1994*, 1994.

[18] M Stonebraker and G. Schloss. Distributed RAID - A new multiple copy algorithm. In *6th Intl. IEEE Conf. on Data Eng. IEEE Press*, pages 430–437, 1990.

[19] N. Talagala, S. Asami, D. Patterson, and K. Lutz. Tertiary disk: Large scale distributed storage. Technical Report No. UCB//CSD-98-989, Univeristy of California at Berkeley, 1998.

[20] Rajeev Thakur, Ewing Lusk, and William Gropp. I/O in parallel applications: The weakest link. *The International Journal of High Performance Computing Applications*, 12(4):389–395, Winter 1998. In a Special Issue on I/O in Parallel Applications.

[21] John Wilkes, Richard Golding, Carl Staelin, and Tim Sullivan. The HP AutoRAID hierarchical storage system. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, pages 96–108, Copper Mountain, CO, 1995. ACM Press.