CSAR: Cluster Storage with Adaptive Redundancy

Manoj Pillai, Mario Lauria
Department of Computer and Information Science
The Ohio State University
Columbus, OH, 43210 Email: 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 how to provide efficient data redundancy to decrease their vulnerability to disk failures. In this paper we describe CSAR, a version of PVFS augmented with a novel redundancy scheme that addresses the efficiency issue while using unmodified stock file systems. By dynamically switching between RAID1 and RAID5 redundancy based on write size, CSAR achieves RAID1 performance on small writes, and RAID5 efficiency on large writes. On a microbenchmark, our scheme achieves identical read bandwidth and 73% of the write bandwidth of PVFS over 7 I/O nodes. We describe the issues in implementing our new scheme in a popular striped file system such as PVFS on a Linux cluster with a high performance I/O subsystem.

1 Introduction

Input/Output has been identified as the weakest link for parallel applications, especially those running on clusters [10]. 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 [6] 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 made PVFS very popular 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 CSAR project is to study issues in redundant data storage in high-bandwidth cluster environments. We have extended PVFS so as to make it tolerant of single disk failures by adding support for redundant data storage. Adding redundancy inevitably reduces the performance seen by clients, because of the overhead of maintaining redundancy. Our primary concern is to achieve reliability with minimal degradation in performance. A number of previous projects have studied the issue of redundancy in disk-array controllers. However, the problem is significantly different in a cluster file system like PVFS where there is no single point through which all data passes.

We have implemented 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 uses RAID5 style (parity-based) writes for large write accesses, and mirroring for small writes. The goal of the Hybrid schemes is to provide the best of the other two schemes by adapting dynamically to the presented workload.

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 and outlines future work.

2 Motivation

Redundancy schemes have been studied extensively in the context of disk-array controllers. The most popular configurations used in disk-array controllers are RAID5 and RAID1. In the RAID5 configuration, storage is organized into stripes. Each stripe consists of one block on each 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 overhead in RAID5 is just 1/n.

In addition to having low storage overhead, RAID5 also provides good performance for writes that span an integral number of stripes. The performance overhead in RAID5 has two components: (1) the overhead for computing the parity. (2) the overhead for writing out the parity blocks. The overhead for computing parity depends on the number of data bytes being written. The overhead for writing out the parity depends on the number of parity bytes to write – if the number of data bytes is kept constant, the number of parity bytes to write decreases as the number of disks in the array increases. Thus for very large disk arrays, and large writes, RAID5 has both low storage overhead, and low performance overhead.

The major problem with the RAID5 configuration is its performance for a workload consisting of small writes that modify only a portion of a stripe. For such a write, RAID5 needs to do the following: (1) Read the old version of the data being updated and the old parity for it (2) Compute the new parity (3) 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.

In the RAID1 configuration, two copies of each block are stored. This configuration has fixed storage overhead, independent of the number of disks in the array. The performance overhead is also fixed, since one byte of redundancy is written for each byte of data.

A number of variations have been proposed to the basic RAID5 scheme that attempt to solve the performance problem of RAID5 for small writes. The Hybrid scheme that we present in this paper is one such scheme. Our work differs from previous work in two ways:

1. 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. Our emphasis is on performance rather than storage overhead.

2. Many of the previous solutions are intended for disk-array controllers. We are interested in a cluster environment where multiple clients access the same set of storage servers.

In a cluster environment, the RAID5 scheme presents an additional problem. In parallel applications, it is common for multiple clients to write disjoint portions of the same file. With the RAID5 scheme, two clients writing to disjoint portions of the same stripe could leave the parity for the stripe in an inconsistent state. To avoid this problem, an implementation of RAID5 in a cluster file system would need additional synchronization for clients writing partial stripes. RAID1 does not suffer from this problem.

Our Hybrid scheme uses a combination of RAID5 and RAID1 writes to store data. Most of the data are stored using RAID5 style redundancy. RAID1 is used to temporarily store data from partial stripe updates, since this access pattern results in poor performance in RAID5. As a result, the storage system has reliability with low storage overhead, while also providing good bandwidth for a range of access patterns. Since partial stripe writes are written using the RAID1 scheme, we avoid the synchronization necessary in the RAID5 scheme for this access pattern.

3 Related Work

There are a number of projects that address the performance problems with a centralized file server. Zebra [3], xFS [1] and Swarm [4] all use multiple storage servers similar to PVFS, but store data using RAID5 redundancy.

They use 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 [8]. This overhead is low for some workloads, like software development workloads, but it can be quite significant for others, like transaction processing workloads. Our work uses a different storage scheme in order to perform well for a larger range of workloads.

Swift/RAID [7] implements a distributed RAID system with RAID levels 0, 4 and 5. In their 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 some of the characteristics reported in the Swift paper. For example, doing parity computation 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 does not implement any variation similar to our Hybrid scheme.

The RAID-x architecture [5] is a distributed RAID scheme that uses a mirroring technique. RAID-x achieves good performance by delaying the write of redundancy. Delaying the write improves the latency of the write operation, but it need not improve the throughput of the system. Also, a scheme that delays the writing of redundancy does not provide the same level of fault-tolerance as the schemes discussed here.

HP AutoRAID [11], parity logging [9] and data logging [2] address the small write problem of RAID5, but they provide solutions meant to be used in centralized storage controllers. These solutions cannot be used directly in a cluster storage system. For example, AutoRAID uses both RAID1 for hot data (write-active data) and RAID5 for cold data. It maintains metadata in the controller's non-volatile memory to keep track of the current location of blocks as they migrate between RAID1 and RAID5. In moving to a cluster, we have to re-assign the functions implemented in the controller to different components in the cluster in such a manner that there is minimal impact on the performance.

4 Implementation

4.1 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 file on its local file system. The name of this local file is based on the inode number assigned to the PVFS file on the manager.

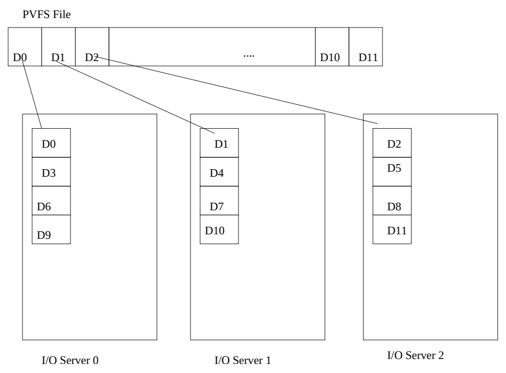

Figure 1: File Striping in PVFS

Figure 1 shows the striping for a PVFS file using 3 I/O servers. Each I/O server has one data file corresponding to the PVFS file in which it stores its portion of the PVFS file. The blocks of the PVFS file are labeled D0, D1 etc. and the figure shows how these blocks are striped across the I/O servers. PVFS achieves good bandwidth on reads and writes because multiple servers can read and transmit portions of a file in parallel.

4.2 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. Figure 2 shows how the data and redundancy blocks are distributed in the RAID1 scheme to prevent data loss in the case of single disk crashes. The data blocks are labeled D0, D1 etc. and the corresponding redundancy blocks are labeled R0, R1 etc. The contents of a redundancy block are identical to the contents of the corresponding data block. As can be seen from the figure, the data file on an I/O server has the same contents as the redundancy file on the succeeding I/O server.

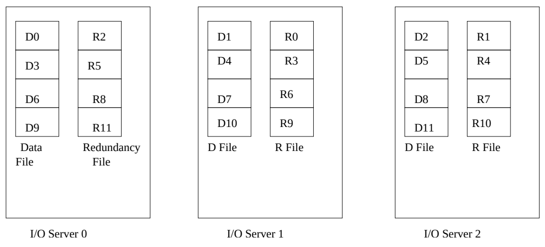

Figure 2: The RAID1 scheme

As is the case in PVFS, the RAID1 scheme in CSAR is able to take advantage of all the available I/O servers on a read operation. On a write, all the I/O servers may be used but the RAID1 scheme writes out twice the number of bytes as PVFS.

4.3 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 RAID5 scheme is shown in Figure 3. 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 to see if any stripes are about to be updated partially. There can be at most two such partially updated stripes in a given write operation. The client reads the data in the partial stripes and also the corresponding parity region, computes the parity, 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

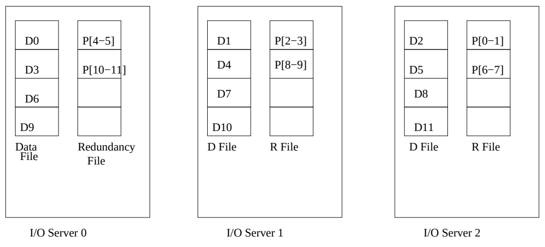

Figure 3: The RAID5 scheme

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.

4.4 The Hybrid Scheme

In the Hybrid scheme, every client write in broken down into three portions: (1) a partial stripe write at the start (2) a portion that updates an integral number of full stripes (3) a trailing partial write. 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 requirement 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 in the overflow regions.

PVFS allows applications to specify striping characteristics like stripe block size and number of I/O servers to be used. The same is true of the redundancy schemes that we have implemented. Even though redundancy is transparent to clients, the design of our redundancy schemes allows the same I/O servers used for storing data to be used for redundancy.

5 Performance Results

5.1 Cluster Description

Our testbed consists of 8 nodes each with two 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 60 GB disks connected using a 3Ware controller in RAID0 configuration.

5.2 Performance for Full Stripe Writes

We measured the performance of the redundancy schemes with a single client writing 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 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 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 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 quite low.

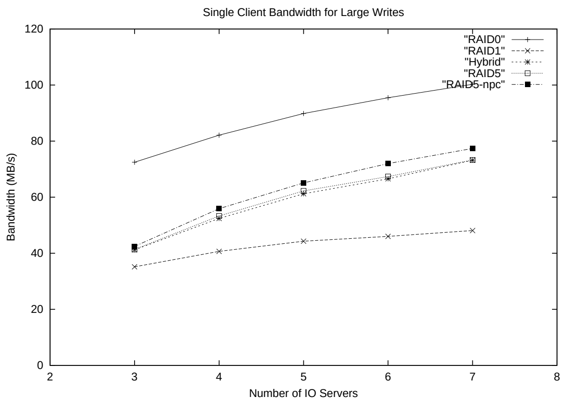

Figure 4: Large Write Performance

5.3 Performance for Partial Stripe Writes

To measure the performance for small writes, we wrote a benchmark 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 5 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 memory 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.

5.4 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 6 shows the read performance for the different

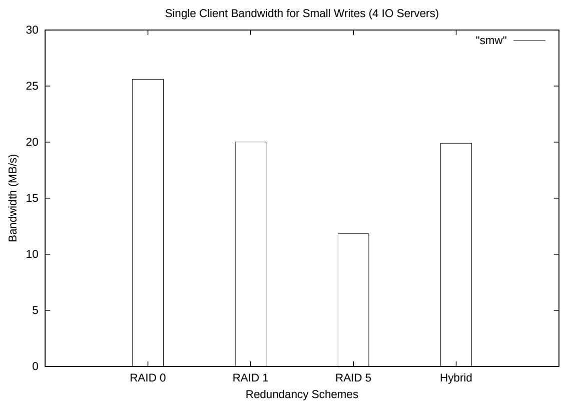

Figure 5: Small Write Performance

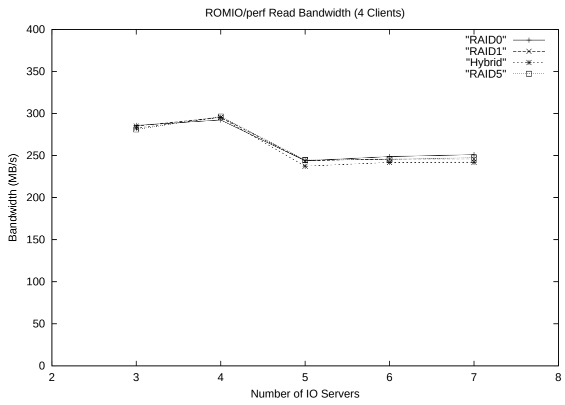

Figure 6: ROMIO/perf Read Performance

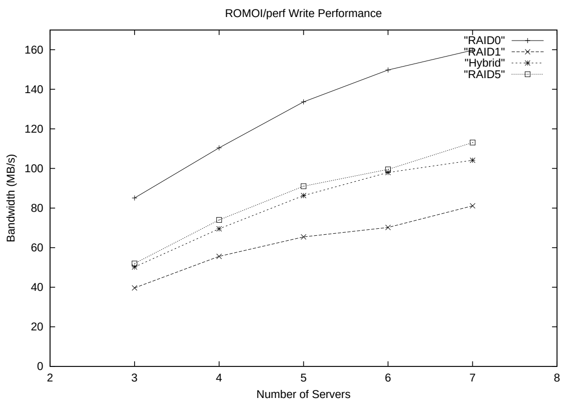

Figure 7: ROMIO/perf Write Performance

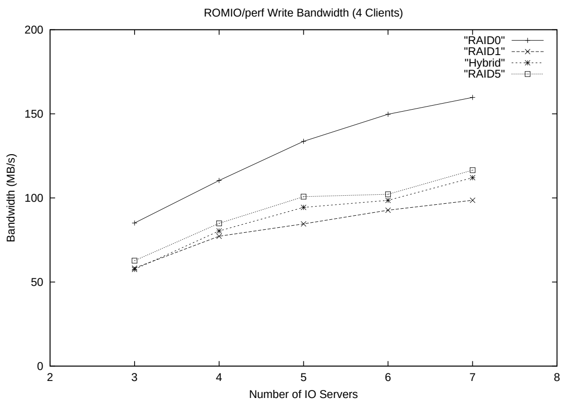

Figure 8: ROMIO/perf Performance with Delayed Redundancy Write

schemes. All the schemes had similar performance for read. The write performance of the RAID5 and the Hybrid schemes, shown in Figure 7 are better than RAID1 in this case because the benchmark consists of large writes.

5.5 Effect of Delaying Redundancy Write

The degradation in write performace can be reduced by delaying the write of redundancy to disk. In the experiments reported above, the effect of an *fdatasync* call by the application is to flush both the data and redundancy to disk. Other projects have reported better performance for mirroring schemes compared to RAID5 by delaying the writing of redundancy [5].

We modified the I/O server code so that on an *fdatasync* operation, the server flushes only the data files to disk. In this scheme, the files storing redundancy are eventually written back to disk when the local file system on the I/O server flushes its cache. In the Hybrid case, the table holding the list of overflow regions was also flushed to disk on the *fdatasync*. The results are shown in Figure 8. In our implementation, RAID-5 and the Hybrid schemes perform better than RAID1 even when the flush of redundancy is delayed for all schemes.

It is to be noted that delaying the flush of redundancy improves the latency seen by the client, but it does not improve the throughput of the storage system. In that respect, the parity-based schemes which write fewer number of bytes have an advantage over RAID1.

6 Conclusions and Future Work

Our experiments demonstrate that the Hybrid redundancy scheme performs as well as RAID5 for workloads comprising large writes and as well as RAID1 for small writes. We expect that for applications consisting of a mix of small and large writes, the Hybrid scheme will show significant improvement over both RAID1 and RAID5. However, more experiments are needed before we can reach that conclusion. Our experiments so far have not given a good idea of the overhead of RAID5 for applications with large data sets, or the synchronization overhead in RAID5. Also, the storage overhead of the Hybrid scheme for real applications needs to be studied. We are currently investigating these issues.

Currently, we specify the redundancy scheme to be used at the time CSAR is compiled. We are implementing the changes necessary to allow the redundancy scheme to be chosen on a file granularity at the time of creation of a file. The main motivation for this feature is to allow temporary files to be created without the overhead of redundancy.

We have implemented the redundancy schemes as changes to the PVFS library. Our implementation allows us to write programs that use the PVFS library and also run programs written using the MPI-IO interface. PVFS also provides a kernel module that allows the PVFS filesystem to be mounted like a normal Unix filesystem. So far, we have implemented a kernel module only for the RAID1 scheme. Implementing the kernel module will allow us to test our system with other interesting benchmarks. PVFS provides an interface, called the *list I/O* interface, that allows access to non-contiguous portions of a file. Non-contiguous accesses can be used to improve the performance of small accesses by batching together a number of accesses. The redundancy schemes have not been implemented for the *list I/O* interface yet.

One of the goals of the PVFS project was to provide a platform for further research in the area of cluster storage. In our case, it has done that. We found it useful to have an actual file system where we could test our ideas. PVFS has become very popular for high-performance computing on Linux clusters, and implementing our schemes in PVFS will give us access to interesting applications to test our implementations.

7 Acknowledgements

This work was partially supported by the Ohio Supercomputer Center grant # PAS0036-1. 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] Eran Gabber and Henry F. Korth. Data logging: A method for efficient data updates in constantly active raids. *Proc. Fourteenth ICDE*, February 1998.

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

[4] 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.

[5] 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.

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

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

[8] M. Rosenblum and J. Ousterhout. The design and implementation of a log-structured file system. *ACM*

Transactions on Computer Systems, 10(1), February 1992.

[9] 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.

[10] 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.

[11] 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.