RAAC: An Architecture for Scalable, Reliable Storage in Clusters

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

Striping data across multiple nodes has been recognized as an effective technique for delivering high-bandwidth I/O to applications running on clusters. However the technique is vulnerable to disk failure, and a number of research efforts have focused on ways to improve reliablity of striped storage systems at a minimal cost in terms of performance and scalability. In this paper we present a novel I/O architecture for clusters called Reliable Array of Autonomous Controllers (RAAC) that builds on the technique of RAID style data redundancy. The RAAC architecture uses a two-tier layout that enables the system to scale in terms of storage capacity and transfer bandwidth while avoiding the synchronization overhead incurred in a distributed RAID system. We describe our implementation of RAAC in PVFS, a popular parallel file system for Linux clusters. We compare the performance of parity-based redundancy in RAAC and in a conventional distributed RAID architecture using microbenchmarks as well as recognized parallel benchmarks and application kernels.

Keywords: Clusters, Fault Tolerance, I/O and File Systems, Distributed RAID

1 Introduction

Input/Output has long been identified as the weakest link for parallel applications, including those running on clusters [3, 16]. The current shift of high-performance computing toward data-intensive applications is further increasing the urgency of developing scalable, high-bandwidth storage systems that can meet the storage requirements of scientific computing environments. The need for scalable bandwidth has resulted in a shift from traditional single-server file systems like NFS to multi-server cluster file systems like the Parallel Virtual File System (PVFS) [9]. A similar trend exists in enterprise data centers, where Storage Area Networks (SANs) and SAN file systems have been deployed to provide shared access to large collections of storage devices. We define a storage cluster as a storage system that is characterized by a large number of storage devices and controller nodes interconnected by a high-bandwidth, low-latency network.

One of the important issues in any storage system is dealing with the failure of disks and other components that can result in loss of critical data. The large number of components in a storage cluster means that there is a high probablity that at any given time the cluster is in a state of partial failure. Traditional techinques like backup are impractical in these environments because of the bandwidth they consume and because of the complexity of administration. We believe efficient redundancy is crucial in data-intensive computing environments dealing with very large data sets. Redundancy allows storage clusters to deal with partial failures without administrator intervention and with minimal degradation in performance.

*Corresponding Author

High-bandwidth storage systems for clusters provide scalable capacity and performance by striping data across multiple nodes, leveraging the high speed communication available on clusters. File striping can naturally be extended to include RAID-like redundancy (Redundant Array of Independent Disks), and previous research results are available on distributed RAID implementations. One major issue introduced by this kind of data redundancy is that, in order to ensure consistency, clients have to synchronize their accesses to the storage system. In large applications with many clients, this synchronization overhead is a major concern. Earlier work on distributed RAID [7, 12, 11] has quantified the significant performance penalty paid for maintaining consistency.

In this paper, we present a novel architecture for storage clusters that addresses the performance problems of distributed redundancy schemes. The Reliable Array of Autonomous Controllers (RAAC) architecture provides scalable storage by employing multiple controller nodes acting as the clients of a distributed RAID scheme, but avoids the synchronization performance penalty by allowing autonomous operation of these controllers. The key is a novel mapping scheme that grants controllers exclusive ownership of their portions of the storage. The RAAC architecture is suitable for cluster file systems and for network-attached multi-controller storage boxes. We implement the RAAC architecture in PVFS and study the tradeoffs of adding an intermediate buffering layer. We compare the RAAC architecture to our earlier implementation of distributed RAID in PVFS.

The paper is structured as follows. Section 2 describes the RAAC architecture. Section 3 discusses related work. Section 4 gives an overview of PVFS, and describes our implementation of RAAC. Section 5 describes experimental results. Section 6 provides our conclusions.

2 The RAAC Architecture

In the RAAC architecture, the storage system is organized into two tiers. At the upper tier, clients stripe data across an array of *storage controllers* using a RAID0 scheme (striping without redundancy) as in a non-redundant striped file system. At the second tier, the storage controllers write to a set of *storage nodes* over a network using a redundancy scheme. The storage nodes form a Redundant Array of Network Disks (RAND) that is shared by all controllers; in contrast traditional RAID storage is accessed by a single controller over a private bus. The storage in the RAND is organized into stripes, just as in a RAID; however, blocks of the RAND are mapped to the storage controllers in a way that each stripe is owned completely by a single controller.

Figure 1 shows the RAAC architecture. C0, C1 and C2 are storage controllers. S0, S1, S2 and S3 are the storage nodes that form the RAND. In this case, the RAND is organized in a RAID5-style, rotating parity layout. The data blocks D0, D1, D2 and the parity block P0 form stripe 0, and are all mapped to C0 i.e. all accesses to these blocks are made by controller C0. Similarly, C1 and C2 have ownership of stripe 1 and stripe 2, respectively. In our implementation of RAAC, a storage controller is a server process running on a machine in the cluster; a storage node is a server process that stores data on the local disk of the machine on which it is running. A single node in the cluster may host both a storage controller and a storage node.

In a traditional RAID system, a single controller serves accesses to the storage. In a distributed RAID, each client acts as a controller that shares the storage with other controllers. The RAAC architecture has a number of advantages compared to a distributed RAID:

- Redundancy is removed from clients, and hence they do not need to perform any additional synchronization as required by distributed RAID.
- Since each storage controller owns a portion of the shared storage, it does not need to synchronize with other controllers when accessing the shared storage nodes.

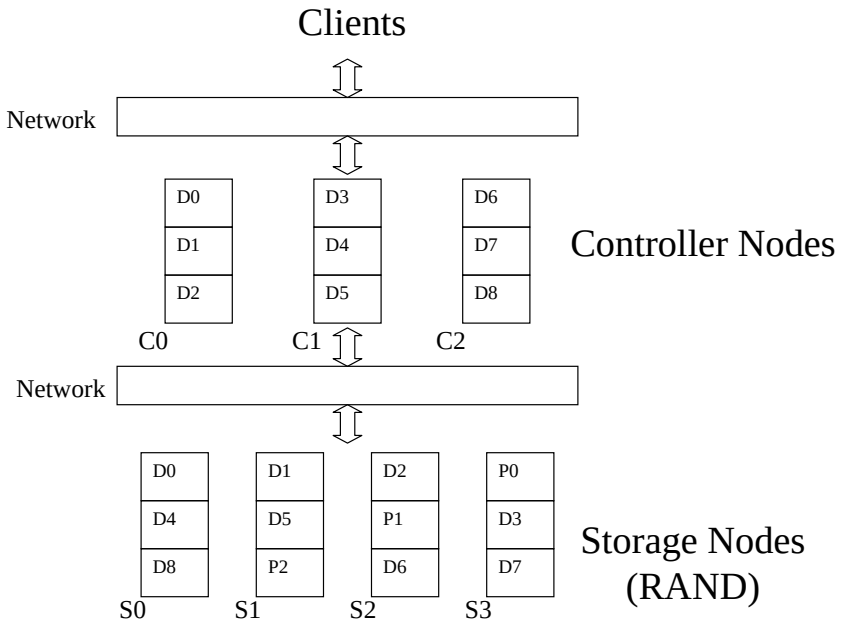

Figure 1: The RAAC Architecture

- In the event of failure of a RAAC controller, the shared storage can be remapped and accessed using the other controllers. As a result, the architecture can be used in high-availability environments.
- All the data for a particular stripe on the RAND are transferred through a single controller. This enables the controller to aggregate writes from different clients into larger writes. In the case of RAID5 the aggregation can result in fewer partial-stripe writes. A distributed RAID does not permit this optimization.
- Perhaps the most interesting aspect of RAAC is that it makes it is easy to employ optimizations of the basic RAID5 scheme. Many such schemes have been developed for traditional RAID controllers; examples include parity logging and hot mirroring. Usually these variations require the controller to maintain additional metadata; they are difficult to implement in a distributed RAID architecture because this metadata would have to be shared by all the clients. However, in the RAAC architecture, the controllers do not need to share their metadata with one another during normal operation; they only need to ensure that the metadata can be recovered by other controllers in the event of a controller crash. This key property of the architecture allows easy and efficient implementation of sophisticated partial-redundancy schemes.
- The RAAC layer provides a level of indirection that can be useful in hiding the heterogeneity of storage devices from clients, and in allowing reorganization of storage transparently.
- The RAAC nodes can act as agents for performing routine data maintenance operations like automated backup and compression of cold data.

The main disadvantage of the RAAC architecture is the extra network hop in the data access path. We believe the RAAC architecture makes up for this disadvantage by allowing efficient implementation of partial-redundancy schemes like RAID5. The bottleneck in today's storage systems is not the network but the disk bandwidth. The RAAC architecture attempts to address this bottleneck by optimizing disk bandwidth at the expense of network bandwidth.

The bandwidth disadvantage of RAAC can be minimized by pipelining the transfers at the two tiers. Preliminary results show an implementation of RAAC in PVFS obtaining write bandwidth comparable to unmodified PVFS. The caching provided by the RAAC controllers can help reduce the latency disadvantage of RAAC. Many high-end cluster file systems already use caching nodes to improve latency and bandwidth; the RAAC architecture assigns an additional function to these nodes.

3 Related Work

The distributed RAID concept was explored by Stonebraker and Schloss [15] and Swift/RAID [10]. In these systems, network bandwidth was the main bottleneck. Recent advances in high-speed networking have resulted in disk bandwidth becoming the performance bottleneck in cluster environments, and necessitate a re-evaluation of distributed RAID in these environments. Petal [8] provides a distributed RAID implementation based on mirroring but does not support partial-redundancy schemes.

Zebra [5], xFS [1] and Swarm [6] 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 [13]. The RAID-x architecture [7] 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 accesses. For applications that need high, sustained bandwidth, RAID-x suffers from the limitation of mirroring.

Aspects of RAID have been studied extensively in the context of disk-array controllers [2]. Various optimizations have been suggested to address the small-write problem of parity-based redundancy schemes. Examples include HP AutoRAID [17], parity logging [14] and data logging [4]. These solutions are difficult to adapt to a distributed RAID architecture. However, they suggest optimizations that might be used in the RAAC architecture to address the performance issues with RAID5.

4 Implementation

In this section we give an overview of the PVFS design and our implementation of RAAC and distributed RAID.

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.

4.2 Distributed RAID5 Implementation

The Distributed RAID5 implementation is described in [11]. In the RAID5 implementation, each I/O server maintains a local parity file in addition to the local data file maintained by PVFS. The client PVFS library has been modified to compute and write parity to the I/O servers. Clients also synchronize their accesses to maintain consistency of the parity blocks during partial stripe updates. The I/O servers provide a simple locking scheme to serialize conflicting accesses to the same stripe.

4.3 RAAC Implementation

In our RAAC implementation, clients view the storage system as a normal PVFS filesystem and perform accesses using an unmodified PVFS library. However, each I/O server seen by the client is a RAAC storage controller that, instead of writing to the local file system, writes to a second PVFS filesystem. Each storage controller is a modified PVFS I/O server that is dual-threaded and maintains a user-level cache. A master thread receives data from the clients and writes to the cache. A slave thread flushes data in the cache to the second PVFS filesystem that forms the RAND storage. The RAAC controller maintains the buffers allocated to a file in a linked list, and also maintains the buffers in an LRU list. When there are no free buffers available, buffers are reclaimed from the LRU list. The LRU list is also used to determine the order in which the controller performs write-back to the RAND storage.

On read accesses, the master thread checks whether the data being requested is in the controller cache. If it is, the data is sent to the client requesting the read; if it is not, the slave thread is instructed to read the requested portion of the file from the RAND storage. The master thread also initiates readahead for sequential read accesses. The controller cache is accessed by both the master and slave threads; they synchronize their accesses using mutex locks and condition variables.

The RAND storage uses a version of PVFS augmented with parity-based redundancy. The implementation is derived from our distribued RAID5 system described in [11]. However, since controllers have exclusive ownership of portions of files, this RAID5 implementation does not have the synchronization code present in the earlier version. In order to avoid copying, reads and writes by the controllers to the RAND storage use scatter/gather versions of the PVFS read/write operations (*pvfs_readv* and *pvfs_writev*).

5 Performance Results

5.1 Experimental Setup

We used two production clusters at the Ohio Supercomputer Center to run our experiments. The first cluster consists of 124 nodes with dual 1400MHz AMD Athlon processors, 2GB of RAM and a single 80GB SCSI disk, interconnected with a Myrinet network. The second cluster consists of 128 nodes with dual 900MHz Itanium II processors, 4GB of RAM and a single 80GB SCSI disk, interconnected with a Myrinet network.

5.2 Microbenchmark Performance

Figure 2 shows the performance of the storage schemes with a microbenchmark in which a single client writes and reads 64MB of data to a PVFS filesystem with 6 I/O servers on the AMD cluster. The data written easily fits in the cache at the I/O servers. The storage schemes have the same read performance for this benchmark. The write bandwidth of RAID5 is only about 62% of PVFS because of the overhead of computing and writing parity. The write bandwidth of RAAC is about 4% better than PVFS. We saw this behavior in our experiments with applications that write small amounts of data that fit into the cache at the RAAC controllers. The reason is that RAAC receives the data directly into user-level cache buffers, whereas PVFS has a copy to the kernel file system cache on the critical path of the write. We expect that adding user-level caching to PVFS would negate the performance advantage of RAAC for this access pattern.

Figure 3 shows the results for a microbenchmark in which a single client writes a large amount of data in fixed-size chunks to a PVFS filesystem. We configured the benchmark to write 20GB of data and used 6 I/O servers on the AMD cluster. The total amount of data written is much larger than the combined cache sizes at the I/O servers. The RAAC controllers were configured with 512MB of data. We report results for two write sizes: a small chunk size of 64KB and a large chunk size of 64MB.

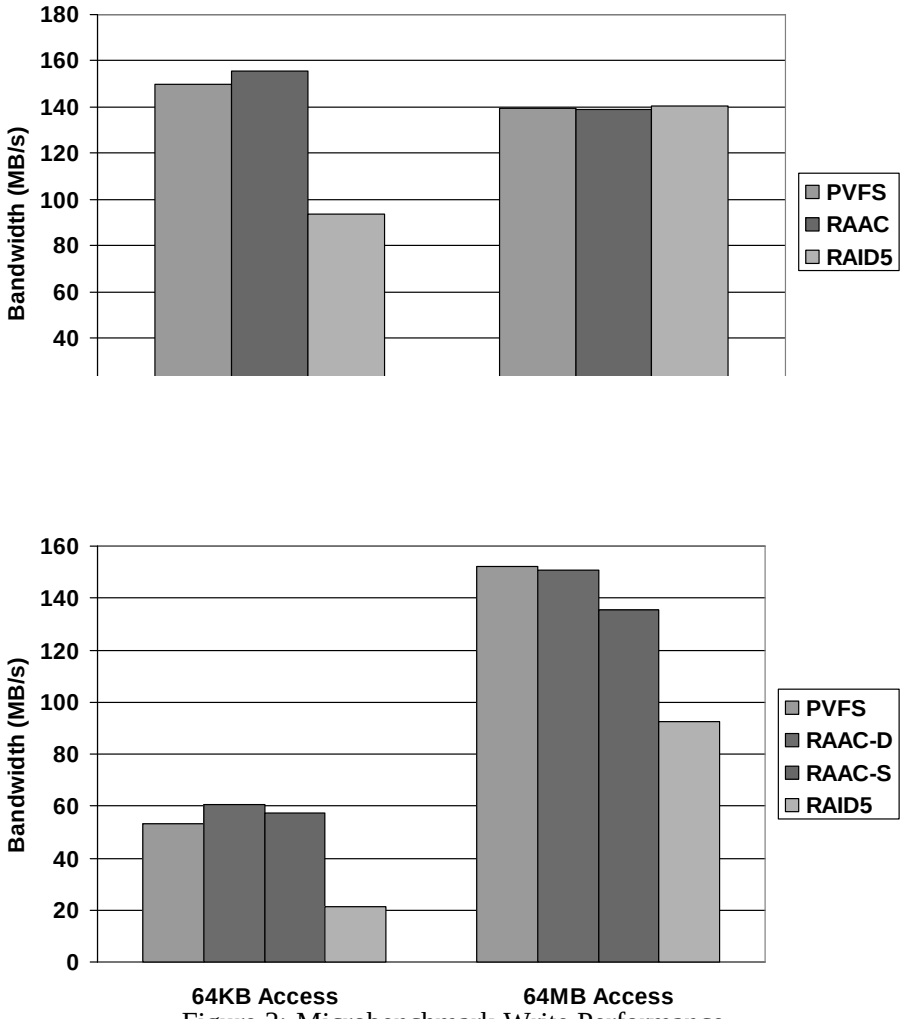

Figure 3: Microbenchmark Write Performance

We used two different configurations for RAAC: in one case, each physical server node is shared by one controller and one storage server; in the other, the controllers and storage servers of RAAC ran on dedicated nodes. The *RAAC-S* results in Figure 3 correspond to the case where the physical node is shared by a controller and a storage server; the *RAAC-D* numbers correspond to the configuration where each physical node performs a dedicated function. The *RAAC-S* configuration uses the same hardware resources as PVFS and RAID5; the *RAAC-D* configuration uses additional hardware resources.

For the 64KB write size, the performance of RAAC is better than PVFS by about 10%. For the 64MB write size, *RAAC-D* has about the same performance as PVFS, whereas *RAAC-S* is slower by about 10%. The large amount of data written by the benchmark means that the RAAC controllers have to flush data to the storage nodes. With small write requests, the controllers have more time in-between requests to flush the data. This fact combined with the efficient user-level caching on the controllers results in better performance for RAAC compared to PVFS for small requests. The results from Figure 3 show that our dual-threaded implementation of the RAAC controller is able to effectively overlap the receiving data from the clients with flushing data to the storage servers. Note that the RAAC controllers add parity to the data they write to the storage nodes. However, we noticed that the results were not affected when we commented out the code for computing and writing parity.

For large write requests the performance of RAID5 is 60% of the performance of PVFS. The slowdown

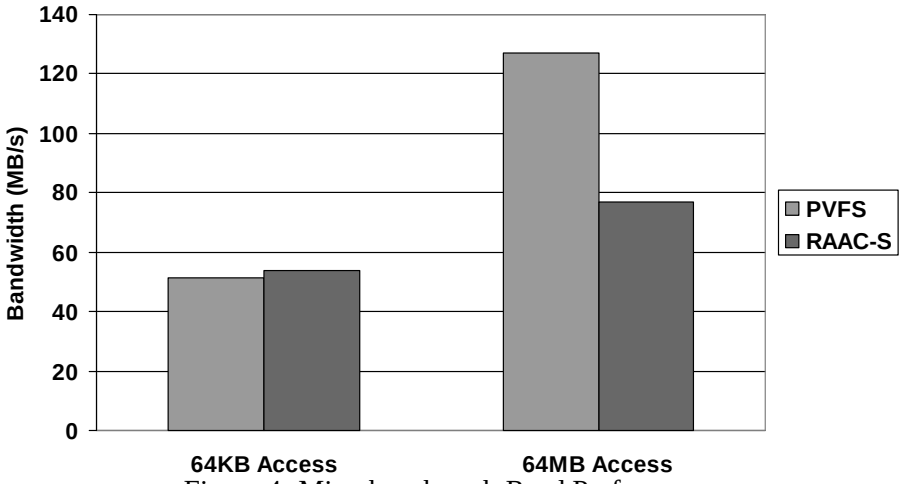

Figure 4: Microbenchmark Read Performance

results from the overhead of computing and writing parity. For small write requests, the performance of RAID5 is only 40% of the performance of PVFS because of the additional overhead of reading the old data and parity. Since there is only one client, there is no synchronization overhead for RAID5 in this benchmark.

Figure 4 shows the results for the same benchmark configured for reading from a PVFS filesystem. The benchmark writes 20GB of data to 6 I/O servers and then measures the bandwidth when reading back the data sequentially. Only the results for *RAAC-S* and PVFS are shown. RAID5 has the same layout for the data files as PVFS, and is expected to have the same performance as PVFS. Also, the *RAAC-D* numbers were similar to *RAAC-S*. For the 64KB access size, RAAC has about the same performance as PVFS. For the 64MB access size, the read bandwidth of RAAC is only about 61% of PVFS. Large, sequential reads for data not present in the controller cache is a problematic access pattern for RAAC that needs to be addressed.

5.3 BTIO Benchmark

The BTIO benchmark is derived from the BT benchmark of the NAS parallel benchmark suite, developed at the 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 the Class A and Class B versions of the benchmark. The Class A version of BTIO outputs a total of about 400 MB to a single file; Class B outputs about 1600 MB. The BTIO benchmark accesses PVFS through the ROMIO implementation of MPI-IO. ROMIO optimizes small, non-contiguous accesses by merging them into large requests when possible. As a result, for the BTIO benchmark, the PVFS layer sees large writes. 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 results in the following sections were obtained using the same handware resources for each storage scheme. That is, the number of I/O servers was kept the same; for RAAC, the controller and storage node were co-located on the same machine. Figure 5 shows the write performance for the Class A benchmark on the AMD cluster. The performance of RAAC is consistently better than PVFS because of the efficient user-level caching described above. The figure also shows the synchronization overhead incurred by distributed RAID5. The *RAID5.nl* graph shows the write performance of RAID5 with the locking code commented out. The figure shows that synchronization between the clients causes significant slowdown with 9 and 25 processes.

The performance of RAID5 relative to PVFS and RAAC is worse on the Itanium cluster. The results are

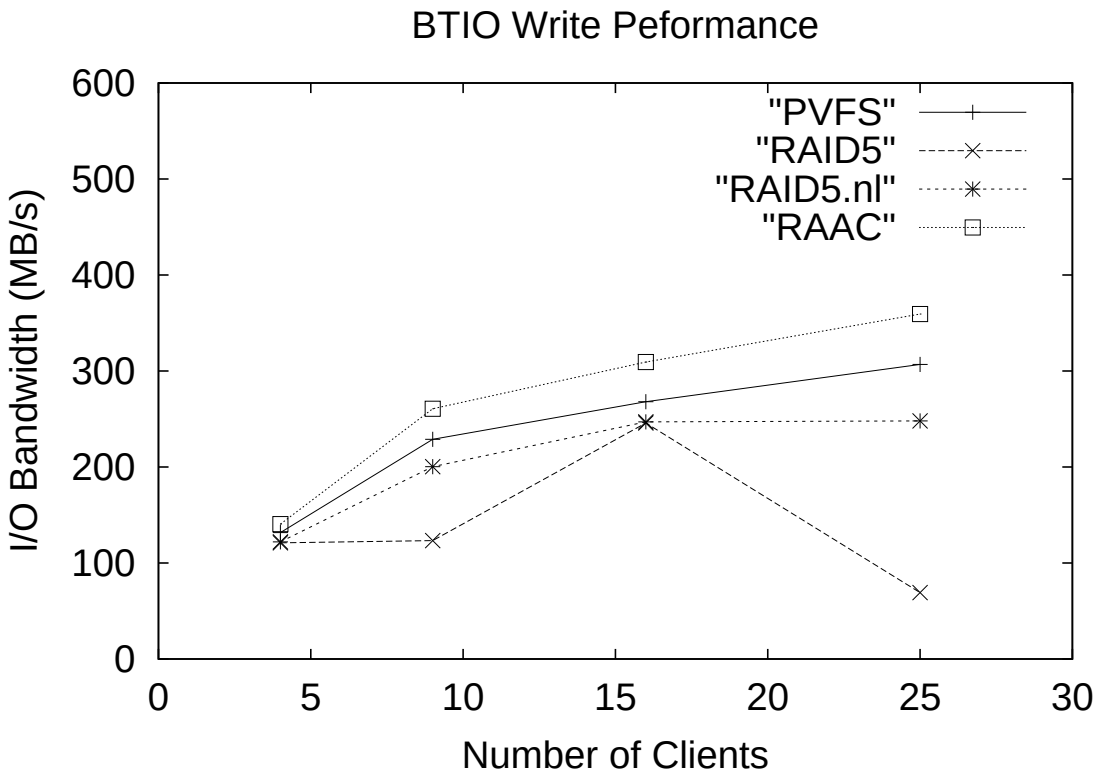

Figure 5: BTIO Class A Write Performance

Table 1: Access Sizes for BTIO Class A Benchmark

Number of Clients	Access Size
4	2621440
9	1165085
16	655360
25	419431

shown in Figure 6. For 25 processes, RAID5 obtains only 2% of the bandwidth of PVFS. Commenting out the locking code only results in a modest improvement in performance for RAID5, shown by the *RAID5.nl* graph. The *RAID5.nr* shows the performance of RAID5 when the code for reading old data and parity are also commented out.

The poor performance of RAID5 can be explained by looking at the access sizes generated by the BTIO benchmark. The access sizes are shown in Table 1. These experiments were run with 6 I/O servers, and with the default PVFS stripe unit size of 64K. With these parameters, accesses for 4 and 16 processes result in full stripe writes for RAID5. For 9 and 25 processes, each access results in two partial stripe writes. These accesses incur the locking overhead as well as the penalty for reading old data and parity.

5.4 Application Performance

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

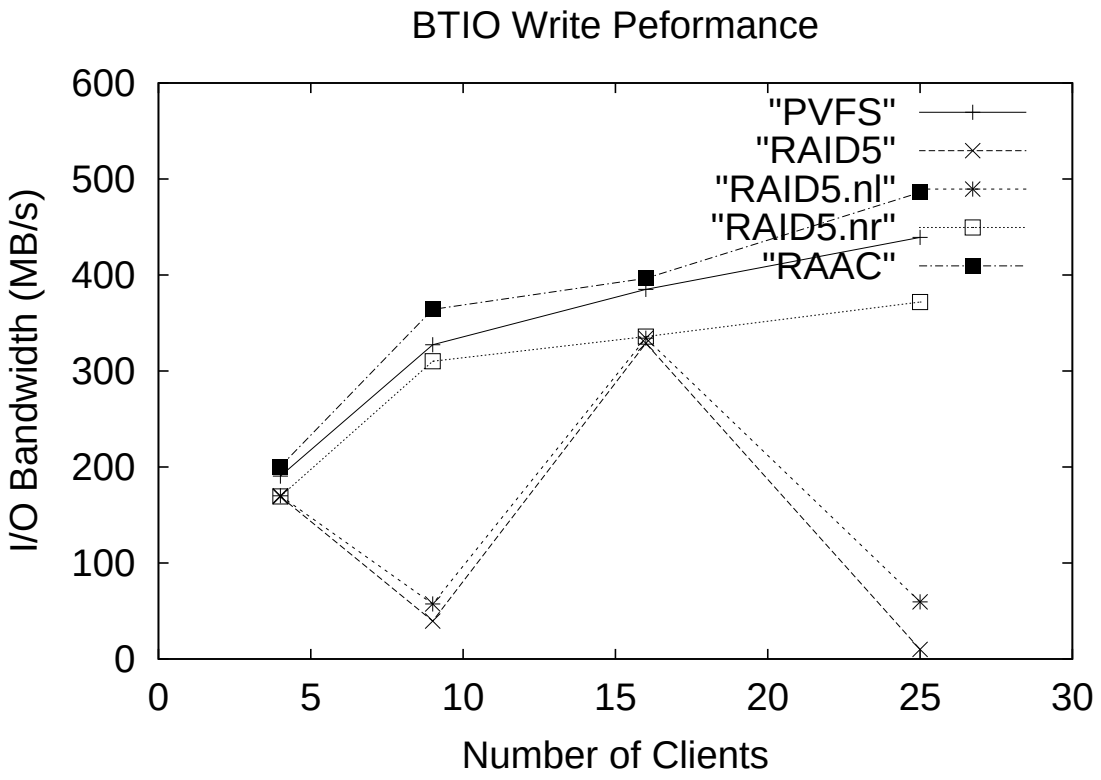

Figure 6: BTIO Class A Write Performance

The FLASH I/O benchmark contains the I/O portion of the ASCI FLASH benchmark. 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 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. Cactus/BenchIO uses the HDF5 library to perform I/O.

The results for Cactus, FLASH I/O and BTIO Class B on the AMD cluster are shown in Figure 7. The performance of RAID5 and RAAC are normalized with respect to PVFS. We ran the FLASH I/O benchmark with 4 clients and recorded the time for outputting the large checkpoint file. For this configuration, the benchmark writes about 37 MB of data. We ran the Cactus 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. The BTIO Class B numbers are for 25 clients. For all applications, RAAC performs better than RAID5. The slowdown for RAAC compared to PVFS results from the delay of applying operations to the second tier.

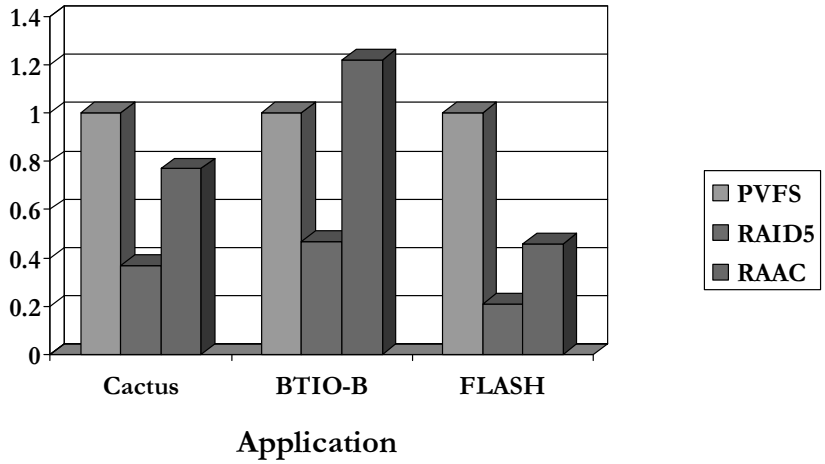

Figure 7: Normalized Application Performance

6 Conclusions

In this paper we have described RAAC, a new storage architecture for large data centers. RAAC adapts RAID to a multi-controller architecture that can scale in bandwidth and capacity, and can serve the storage needs of large clusters running data-intensive applications. RAAC uses a clever mapping of storage to controllers to allow autonomous operation of the controllers and avoids the synchronization overhead seen in distributed RAID systems.

RAAC can be implemented in a striped file system or in a distributed block-level storage system, and can support different redundancy schemes. We have described an implementation of RAAC in PVFS, a popular striped file system with a large user base. For a number of benchmarks and well-known application kernels, RAAC consistently outperforms a distributed RAID5 implementation. For the BTIO Class A benchmark, RAAC performs upto 17% better than PVFS despite the redundancy overhead. These results point to the added benefit of the user-level buffering at the controllers which is the elimination of copies to the kernel cache.

This paper has studied the performance of RAAC relative to a distributed RAID5 implementation in the absence of failures. Recovery from failures and performance in degraded mode have to be studied.

7 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 TOCS*, February 1996.

[2] P. Chen, E. Lee, G. Gibson, R. Katz, and D.Patterson. RAID: High-Performance, Reliable Secondary Storage. *ACM Computing Surveys, Vol.26, No.2, June 1994, pp.145-185*, 1994.

[3] Phyllis E. Crandall, Ruth A. Aydt, Andrew A. Chien, and Daniel A. Reed. Input/Output Characteristics of Scalable Parallel Applications. In *Proceedings of Supercomputing '95*, San Diego, CA, December 1995. IEEE Computer Society Press.

[4] Eran Gabber and Henry F. Korth. Data Logging: A Method for Efficient Data Updates in Constantly Active RAIDs. *Proc. Fourteenth ICDE*, February 1998.

[5] J. Hartman and J. Ousterhout. The Zebra Striped Network File System. *ACM TOCS*, August 1995.

[6] John H. Hartman, Ian Murdock, and Tammo Spalink. The Swarm Scalable Storage System. *Proceedings of the 19th ICDCS*, May 1999.

[7] Kai Hwang, Hai Jin, and Roy Ho. RAID-x: A New Distributed Disk Array for I/O-Centric Cluster Computing. In *Proceedings of HPDC-9*, Pittsburgh, PA, 2000.

[8] Edward K. Lee and Chandramohan A. Thekkath. Petal: Distributed Virtual Disks. In *Proceedings of the Seventh ASPLOS*, Cambridge, MA, 1996.

[9] Walter B. Ligon and Robert B. Ross. An Overview of the Parallel Virtual File System. *Proceedings of the 1999 Extreme Linux Workshop*, June 1999.

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

[11] Manoj Pillai and Mario Lauria. A High Performance Redundancy Scheme for Cluster File Systems. In *IEEE International Conference on Cluster Computing (Cluster 2003)*, Hong Kong, Dec 2003.

[12] Manoj Pillai and Mario Lauria. CSAR: Cluster Storage with Adaptive Redundancy. In *ICPP 2003*, Kaohsiung, Taiwan, Oct 2003.

[13] M. Rosenblum and J. Ousterhout. The Design and Implementation of a Log-Structured File System. *ACM TOCS*, 10(1), February 1992.

[14] D. Stodolsky, M. Holland, W. Courtright, and G.Gibson. Parity Logging Disk Arrays. *ACM TOCS, Vol.12 No.3, Aug.1994*, 1994.

[15] 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.

[16] 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.

[17] John Wilkes, Richard Golding, Carl Staelin, and Tim Sullivan. The HP AutoRAID Hierarchical Storage System. In *Proceedings of the Fifteenth ACM SOSP*, pages 96–108, Copper Mountain, CO, 1995. ACM Press.