Uncovering Errors: The Cost of Detecting Silent Data Corruption

Sumit Narayan John A. Chandy

Dept. of Electrical & Computer Engineering
University of Connecticut
Storrs, CT 06269
{sumit.narayan,john.chandy}@uconn.edu

Samuel Lang Philip Carns Robert Ross

Mathematics & Computer Science Division
Argonne National Laboratory
Argonne, IL 60439
{slang,carns,rross}@mcs.anl.gov

ABSTRACT

Data integrity is pivotal to the usefulness of any storage system. It ensures that the data stored is free from any modification throughout its existence on the storage medium. Hash functions such as cyclic redundancy checks or checksums are frequently used to detect data corruption during its transmission to permanent storage or its stay there. Without these checks, such data errors usually go undetected and unreported to the system and hence are not communicated to the application. They are referred as "silent data corruption." When an application reads corrupted or malformed data, it leads to incorrect results or a failed system. Storage arrays in leadership computing facilities comprise several thousands of drives, thus increasing the likelihood of such failures. These environments mandate a file system capable of detecting data corruption. Parallel file systems have traditionally ignored providing integrity checks because of the high computational cost, particularly in dealing with unaligned data request from scientific applications. In this paper, we assess the cost of providing data integrity on a parallel file system. We present an approach that provides this capability with as low as 5% overhead for writes and 22% overhead for reads for aligned requests and some additional cost for unaligned requests.

Keywords

cyclic redundancy checks · data integrity · parallel file systems

1. INTRODUCTION

File systems use disk drives to provide a persistent medium for applications to store data. These disk drives are complex electro-mechanical systems with several moving parts, making them unreliable in terms of availability and integrity of the data. They could suffer from potential data loss or complete drive failure. They are controlled by sophisticated firmware and software to communicate with other components of the system. These codes could carry bugs and cause data corruption. Given the uncertainties present with storing data on the disk, they are thus referred as "untrusted disk" [1]. But despite such limitations, disk drives are the most common form of storage medium, primarily because of their ability to provide good I/O performance at a very low price. Their price advantage motivated the design of redundant arrays of inexpensive disks (RAID) [2], where several cheap drives are put together to provide data redundancy and high bandwidth. However, increasing the number of drives also increases the chances of failures. Most of the reliability measures taken by RAID focus on protecting data from complete disk failures. Mechanisms such as parity or mirroring can help with restoring data in cases of such failures but don't, by themselves, guarantee the integrity of the data. The reason is that RAID does not have a notion of what is "correct" data from the user's point of view. Disk drives suffer from another type of failure—data corruption. Data corruption refers to unintended modification of data during its transmission, storage, or retrieval. These changes go undetected or unreported to the storage system. They result in incorrect data being flushed to the drive or sent to the application, depending on the flow of data operation. These kind of errors could occur for several reasons. Misdirected writes, torn writes, data path corruption, and bit-rot are some of the ways in which data corruption can occur, resulting in incorrect data being stored on the block. Even if the data was stored correctly, a bug in the firmware code could cause incorrect data to be read from the disk resulting in an error. Since these kinds of data corruption are silent and are not communicated to the system, they are referred as "silent data corruption." Detecting such alteration of data requires integrity checks that provide users guarantees that the same analysis of the same data leads to the same results.

Hash functions such as cyclic redundancy checks (CRCs) or checksums are frequently used for testing the integrity of the data. In high-performance computing (HPC) environments, providing an integrity check becomes a particularly hard problem. File systems connected to supercomputers such as IBM's Blue Gene/P comprise large arrays of storage units typically consisting of thousands of drives, thus increasing the probability of such failures. HPC workloads are also a complex mix of concurrent, unaligned accesses of varying sizes, making the design requirements unpredictable. Further, any changes on these machines must be done with minimal impact on performance because the scientific applications running on them are very sensitive to system's throughput.

A recent study of over 1.5 million disk drives in the NetApp database at different sites over 32 months showed that more than 8.5% of all nearline disks and around 1.9% of all enterprise class disks suffered from silent data corruption [3].

Supercomputing PDSW '09, Nov. 15, 2009. Portland, OR, USA

The same study also found that 13% of these nearline disk errors and nearly 38% of enterprise disk errors went undetected by background verification processes. Further analysis of the same dataset showed that over 400,000 checksum mismatches had occurred during the 41 months of observation [4]. Another large-scale study of disk drives showed that faulty codes in storage system's protocol stack accounted for 5–10% of all failures [5]. Apart from these, an in-house study at European Organization for Nuclear Research (CERN) also concluded that low-level data corruption exists [6]. They ran a simple program to write a 2 GB file on 3,000 nodes every 2 hours for 5 weeks. During this period, they hit 500 errors on 100 nodes, 10% of which were sector-sized or page-sized corrupt data and 80% were 64K regions of corrupted data. Such a high rate of faults clearly indicates that data corruption is not limited to just consumer-based disks but is also present in enterprise-grade disk drives. In HPC environments, these errors could result in significant loss of both data and computation time. Any faults in these petascale storage systems serving critical scientific applications is intolerable. Hence, we need a better understanding of how to perform data integrity checking in parallel file systems.

In this paper, we study the cost of providing integrity checks in a parallel file system and present results by implementing our ideas on the open source Parallel Virtual File System (PVFS). This paper is organized as follows. We discuss data integrity in more detail in Section 2. In Section 3, we provide a brief overview of PVFS. Section 4 lays out our design and implementation in PVFS, and Section 5 provides some experimental results. We end with a discussion of related work and our conclusions.

2. DATA INTEGRITY

Several methods are available for detecting data modification or silent corruption. Data integrity guarantees the users that once their data is stored, it is persistent and perpetually available in the same condition as it was when submitted. A variety of checksum algorithms are available for detecting data corruption. These algorithms convert strings of different lengths of data to short, fixed-size results. Since they are designed to avoid collisions, finding two strings with the same hash result is almost impossible. CRCs, Adler32, and Fletcher's checksum are examples of noncryptographic hash algorithms for detecting unintentional modification of data. Other examples of hash algorithms capable of detecting intentional modification include secure hash algorithms (SHA0, SHA1, SHA2) and message digest algorithm (MD5). They use cryptographic hash functions that are usually complex and computationally expensive. Such algorithms are typically used to encrypt data or to provide digital signatures for files that are transferred over hostile networks. This study concentrates on identifying unnoticed unintentional data corruption which occurs primarily because of hardware faults present in the storage medium or bugs in the software controlling them. In our work, we use the CRC algorithm for evaluating data integrity because of its easier implementation.

Several corruption detection mechanisms exist for handling disk-related errors within the hardware itself. For example, most nearline disks provide a 64-byte data integrity segment after every block of 4 KB for storing checksums [4]. Hard drives also set aside extra space for storing error correction code (ECC) for each sector that can be used to detect and correct misreads or corrupted data. However, studies have shown that not all of these errors are always detected by the system [3–5]. CRCs and checksums provide check codes that help uncover such errors. These data verification codes are calculated inline with a write request and written to disk for validation at a later time. For every change to the data, its verification code is also updated accordingly. During a read operation, the hash code is regenerated for the data read from the disk and verified with the code saved during write. If they match then the data is complete and verified; otherwise, a read error is sent to the application.

3. THE PARALLEL VIRTUAL FILE SYSTEM

The Parallel Virtual File System (PVFS) is an open source implementation of a parallel file system for high-performance computing applications [7]. It consists of a set of servers providing metadata and data services. It is based on the idea of separating metadata from the data, thus keeping storage management functionalities away from the real data access. PVFS achieves high throughput by striping data across many storage nodes. It provides separate API abstractions for network and storage modules that simplify infrastructure-specific implementations. The buffered method interface (BMI) provides a network-independent interface, while Trove provides a persistence layer for manipulating metadata and data streams. PVFS saves a file's data in a directory tree and metadata in a Berkeley DB database, both saved on a local file system. Each node that is part of a file's stripe has a portion of the file saved on a local file system as a *bstream*. Typically, a file is striped across all available I/O servers, and the only available mechanism for fault tolerance other than providing RAID on local file system is by attaching storage to multiple servers with fail-over enabled. This approach, however, provides tolerance only to the failure of local file system or a complete node; it has no provision for detecting and correcting data errors. Berkeley DB allows checksum verification of pages read from the underlying file system enabled through a flag, thus protecting PVFS's metadata from corruption. However, the data *bstream*, which is written to the disk in a local file, depends on the underlying storage system to provide data integrity checks. Lack of such support from the base file system will cause the applications using PVFS to be unaware of any corruption that might occur as a result of a faulty read or write operation at the device level.

In this paper, we study the cost of providing integrity checks that might occur on faulty storage hardware in context of a parallel file system. Similar bit-flip errors could occur on faulty network cards or interconnects. Detection of such faults during network transmission would be better handled at PVFS's BMI abstraction layer, a study of which is beyond the scope of this paper.

4. DESIGN

Adding checksum capability to a parallel file system brings its own challenges. With data spread across several servers, constantly updating CRCs and verifying reads with the latest CRC values becomes a challenge. Further, all of these tasks must be done with minimum impact on performance of the parallel file system. File systems such as Btrfs [8] provide both data and metadata integrity checks and could be used as the underlying file system for PVFS. However, Btrfs is currently under heavy development and not yet ready to be deployed with PVFS. Such a setup also ties PVFS with other file systems and limits its portability. On the other

hand, providing CRC checks within PVFS allows taking advantage of user space caching on servers. It also gives PVFS the flexibility of setting its own CRC's buffer size. This design can also be extended to handle verification only for critical files, without affecting performance for others. Using MPI hints or POSIX extended attributes, applications could provide PVFS information as to which files require validation.

The hash code for each file can be stored either on an MDS along with the file's metadata or on a storage node along with the file's data. Storing the CRC on the MDS allows us to use the client's usually higher processing power compared to storage nodes for calculation of the hash code. Storing the CRC on the MDS also gives us a single point of contact for all data integrity checks. However, this also means that the CRCs will have to be updated atomically with every change of data. In large environments with multiple clients, doing frequent updates creates a single node bottleneck. Although the checksums are small in size even for a very large file and can be easily cached in the client's memory, in a multiple-client environment with multiple writers of the same file it is necessary to always have the latest copy of the CRC code, making the metadata server a major bottleneck.

Alternatively, placing the hash code on storage nodes along with the data allows us to take advantage of distributing the CRC computation across several nodes. It also allows us to take advantage of the node's local cache by placing the CRCs in memory after writes and avoiding CRC reads for subsequent read requests. The I/O overhead of providing integrity checks will thus be visible to the user only during writes. Any following reads will require only obtaining the CRC from the local memory and validating the data. For these reasons, in our work, we chose to place the check codes on storage nodes. We modified the DBPF Trove implementation in PVFS (version 2.8.1) to create a *cksum* file for every *bstream* file that is saved on the storage server to store the CRC codes. CRCs are calculated for every fixed chunk size of a *bstream* file, also referred to as CRC buffer size. The size of this chunk can be changed during the configuration of PVFS. All CRC codes for the data file are saved sequentially in the same order in the *cksum* file as shown in Figure 1.

In our implementation, we used multiple threads to achieve maximum performance for calculating the CRC code. During reads, we use the threads to read-ahead the *cksum* block if it is not already available in memory. This process is done in 4 KB block sizes even if only one CRC code is required. The remaining codes are stored in PVFS's CRC cache instead of being discarded. They remain in the cache until a CRC watermark, another configurable parameter, is reached, after which it is discarded based on a simple LRU algorithm. Once the read is complete, the thread simply checks for correctness of the data using the hash code. For writes, we use the threads to calculate the CRC code in parallel to the write operation. The CRC is updated on disk after the write is complete but before the handle is returned to the client. Any failure between results in a failed write and an error is reported to the application.

Variations in request size and offset create a huge challenge in handling I/O on servers. Nodes can receive requests that may or may not be aligned to the CRC chunk offsets. The offset of the request plays an important role in how expensive the CRC computation will be. The reason for the variation in computation of the hash code for each each of them is explained in detail below.

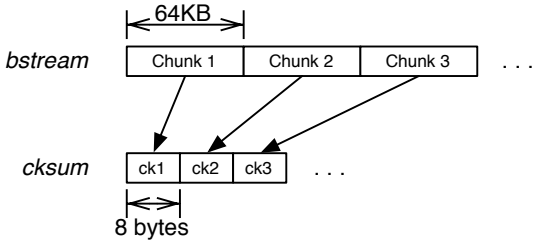


Figure 1: CRCs are stored in a *cksum* file in the same order of data chunks appearing in the *bstream* file.

4.1 Aligned requests

Since the CRCs are always calculated for the complete CRC buffer length, having request offsets aligned to the buffer offsets translates into doing a single I/O of full CRC buffer length on *bstream* and a single I/O on *cksum*. Requests on a *cksum* file will not be necessary if the hash code is already available in the node's cache. Data read from the disk can be verified with the hash code read from *cksum*, while data written will have its corresponding CRC updated on disk and also preserved in cache.

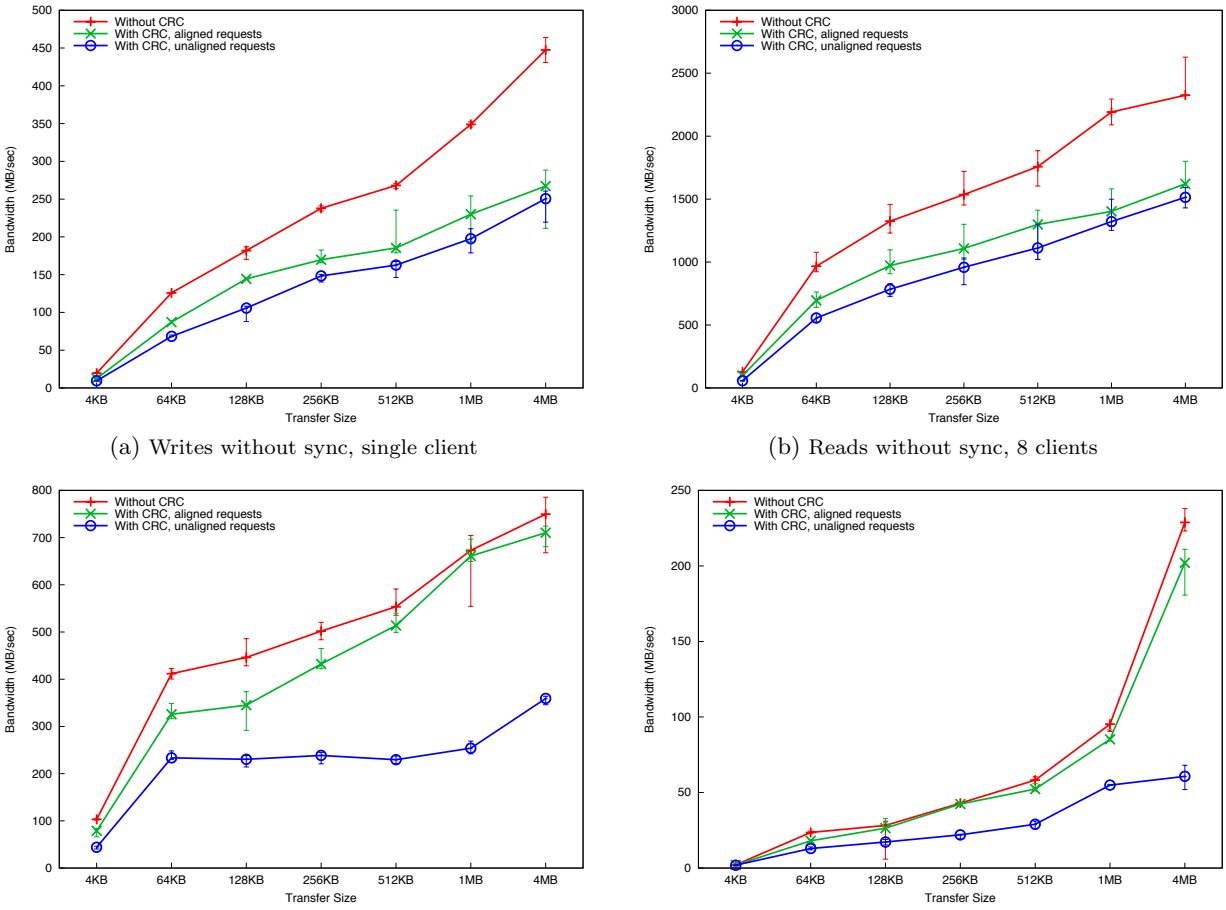
4.2 Unaligned requests

Because of an offset to the CRC data buffer, unaligned requests need to be handled in a special way. In order to check the integrity of the data during reads, the complete CRC buffer has to be read from disk and verified, although only a part of it is returned to the client. Unaligned writes require a more complex operation. The partial blocks of the request will have to be read and verified before being updated with new data. The CRC will then have to be calculated for the updated block and flushed to the disk. If we do not verify the blocks with partial offsets and simply overwrite them, we may fail to detect any corruption that might already exist in the region with old data. This could place an extra overhead on calculating CRCs. The idea of appending the CRC code for updated data to the end of the CRC file is used by the Google file system [9], but it requires keeping track of each updated region and cleaning the CRC file for inactive chunks during the server's idle time.

5. RESULTS

We ran our experiments on a Linux cluster with 8 PVFS I/O nodes serving both as storage servers and as metadata servers. Each node has two dual-core AMD Opteron 2.8 GHz processors with 4 GB RAM. Each PVFS server was configured to use an XFS file system on RAID5 with four 80 GB disks for local storage. We set the CRC buffer size to 64 KB. To test the performance of our system, we used the IOR benchmark developed at LLNL to measure the aggregate I/O throughput [10]. IOR is a benchmarking tool for parallel file systems that allows I/O through a variety of interfaces including POSIX and MPI-IO. The tests were run for single client and multiple client scenarios doing I/O on a 8 GB file using the MPI-IO library. The results shown here are an average of 10 runs with the errorbars indicating the maximum and minimum measurements for each setup. The CRC cache was available in all cases.

We tested the performance of our implementation in two different scenarios and compared it with results from the production release of PVFS. Figure 2(a) shows the write throughput for a single client writing to an 8 GB file without data sync for different transfer sizes. We observed that for



(a) Writes without sync, single client

(b) Reads without sync, 8 clients

(c) Writes without sync, 8 clients

(d) Writes with sync, 8 clients

Figure 2: IOR aggregate I/O bandwidth with 8 GB file using MPI-IO.

aligned requests, with an overhead of 36% for 4 KB transfer sizes and an overhead of 40% for 4 MB transfer sizes, we were able to provide integrity checks on the data stored. Contrary to our assumption of higher overhead for unaligned requests, we observed only a little extra overhead of about 6% over aligned requests for 4 MB transfer sizes.

Figures 2(b) and 2(c) show the read and write bandwidth for 8 clients with the data synchronization turned off and I/O being done asynchronously. We observed a read overhead of 22% for 4 KB transfer sizes and a 30% overhead for 4 MB transfer sizes. For writes, the overhead varied between 24% for 4 KB transfer sizes to 5% for 4 MB transfer sizes. Most of this overhead was due to the computation time of the CRCs. We observed that for reads, the threads would almost always complete the read-ahead of CRCs before the data was read from the *bstream* and for writes, the threads would wait for CRC calculation. Having a separate file for storing CRC codes thus had little to almost no impact on performance. In our experiments, reads took a larger hit compared to writes because for reads we compute the CRC after the data is read, while for writes we do the computation in parallel. For unaligned write requests, we observed a larger overhead of 57% for 4 KB transfer sizes and 52% for 4 MB transfer sizes. This overhead is because of reading the old data with partial requests from the disk and calculating the CRC to verify the existing data before updating it with the new data. This overhead was not visible in the case of a single client because fewer requests were being handled by the I/O nodes.

Figure 2(d) shows the performance when I/O on *bstream* was done synchronously by 8 clients. Writes followed almost the same pattern as in the case of requests without sync. Aligned write overhead varied from 5% for 4 KB transfer sizes to 12% for 4 MB transfer sizes. Although writes for aligned requests gave throughput almost as good as the default PVFS, unaligned requests took a large hit. Again, this difference is caused by the read-verify-write steps required for unaligned write requests. For unaligned writes, the overhead changed from 9% for 4 KB to 73% for 4 MB transfer sizes. For reads (results not shown here), we observed very little variation in the throughput between aligned and unaligned requests. In our experiments, we observed an overhead of 43% for aligned requests of 4 MB transfer sizes and 45% for unaligned requests of 4 MB transfer sizes.

6. RELATED WORK

Significant work has been done on identifying data corruption in large-scale storage environments [3–5], each of them concluding that silent data corruption is a real problem. Gopalan et al. in their paper [11] provided a detailed survey of different integrity assurance mechanisms that are in use today to tackle the problem. However, little work has been done on studying the real cost of providing checks to verify integrity of the data, particularly in the context of a parallel file system. Single-node journaling file systems such as Ext4, JFS and ReiserFS provide user-level tools such as

fsck to check file system consistencies. These tools are provided to ensure proper system startup after unclean shutdown, by keeping the file system's metadata consistent, but they do not perform data integrity checks. ZFS [12] and Btrfs [8] are examples of file systems that provide both data and metadata verification by using checksums. The IRON file system is a prototype file system that provides in-disk checksumming [13] to verify the integrity of the data. This work demonstrates that checksums can be applied to local file system without a large overhead; however, the work is limited to direct attached storage.

Most distributed file systems provide integrity checks only for transmitted data. Lustre [14], for example, has provisions for checksum verification for all blocks transmitted over the network. Any corrupt data detected by that mechanism can be requested again from the sender. However, it fails to provide integrity checks once the data is stored on the disk. Ceph [15] provides integrity checks by using Btrfs as its backend file system. GPFS [16], PFS [17], PanFS [18] and GFS [9] are other popular distributed file systems that provide integrity checks, but only as an option during mount for persistent checks. None of these file systems provide any insight into how expensive checksum computations are, mostly for commercial reasons. Moreoever, most of these file systems provide integrity checks only as a mount option although all studies stress the need for having such a feature available by default.

7. CONCLUSIONS

In this paper we gave an overview of the cost of providing data integrity in a parallel file system. We showed that integrity errors exist and can lead to unexpected results with data loss if not identified immediately. We provided a prototype implementation for integrity checks in PVFS and demonstrated that, with an overhead of around 5% for writes and 22% for reads, we can ensure that the data is free from any errors.

New emerging technologies such as flash translation layers (FTLs) that allow arrays of NAND flashes to be addressed in logical sectors are finding their way into the HPC domain because of their ability to do random data access at several order of magnitude faster than traditional drives. Despite having wear-leveling algorithms and efficient bad-blocks management, however, they are not free from data errors. Flash-based storage has a limited number of write cycles before it wears out and stops storing correct data. This situation emphasizes the need for having a mechanism for data integrity checks for them in place. Our work also shows that the biggest issue of providing such verification checks lies with the cost of computing check codes and not in storing them on disks. We plan to modify the way we calculate the CRCs by establishing a read/write request pipeline, thus using each individual component of the system more efficiently. We are also studying ways in which we could make use of highly parallel general-purpose Graphical Processing Units (GPGPUs) to compute integrity checks. These devices suffer from memory transfer overheads. However, recent studies have shown that using a pipelined architecture to keep the GPU buffers busy, several magnitudes of performance improvement could be achieved [19].

Acknowledgments

This work was supported by the Office of Advanced Scientific Computing Research, Office of Science, U.S. Dept. of Energy, under Contract DE-AC02-06CH11357.

References

[1] G. Sivathanu, C. P. Wright, and E. Zadok, "Enhancing file system integrity through checksums," Stony Brook University, Tech. Rep. FSL-04-04, 2004.

[2] D. A. Patterson, G. A. Gibson, and R. H. Katz, "A case for redundant arrays of inexpensive disks (RAID)," in *Proceedings of the ACM SIGMOD International Conference on Management of Data*, June 1988, pp. 109–116.

[3] L. N. Bairavasundaram, G. R. Goodson, S. Pasupathy, and J. Schindler, "An analysis of latent sector errors in disk drives," in *ACM SIGMETRICS Performance Evaluation Review*, vol. 35, no. 1, June 2007, pp. 289–300.

[4] L. N. Bairavasundaram, G. Goodson, B. Schroeder, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "An analysis of data corruption in the storage stack," in *Proceedings of 6th USENIX Conference on File and Storage Technologies*, February 2008, pp. 223–238.

[5] W. Jiang, C. Hu, Y. Zhou, and A. Kanevsky, "Are disks the dominant contributor for storage failures?" in *Proceedings of the 6th USENIX Conference on File and Storage Technologies*, February 2008.

[6] B. Panzer-Steindel, "Data integrity." [Online]. Available: http://indico.cern.ch/getFile.py/access?contribId=3&sessionId=0&resId=1%&materialId=paper&confId=13797

[7] "Parallel Virtual File System (PVFS)." [Online]. Available: http://www.pvfs.org

[8] "Btrfs: Copy-on-write file system for linux." [Online]. Available: http://btrfs.wiki.kernel.org/index.php/Main_Page

[9] S. Ghemawat, H. Gobioff, and S.-T. Leung, "The Google file system," in *ACM Symposium on Operation System Principles*, October 2003, pp. 29–43.

[10] "IOR HPC benchmark." [Online]. Available: http://sourceforge.net/projects/ior-sio/

[11] G. Sivathanu, C. P. Wright, and E. Zadok, "Ensuring data integrity in storage: Techniques and applications," in *Proceedings of International Workshop on Storage Security and Survivability (StorageSS)*, November 2005.

[12] Sun Microsystems, "ZFS: The last word in file systems." [Online]. Available: http://www.sun.com/2004-0914/feature

[13] V. Prabhakaran, L. N. Bairavasundaram, N. Agrawal, H. S. Gunawi, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "IRON file systems," in *Proceedings of the 20th ACM Symposium on Operating Systems Principles*, October 2005, pp. 206–220.

[14] P. J. Braam and R. Zahir, "Lustre - a scalable high performance file system," Cluster File Systems, Inc., Mountain View, CA, Tech. Rep., July 2001.

[15] S. A. Weil, S. A. Brandt, E. Miller, and D. D. E. Long, "Ceph: A scalable, high-performance distributed file system," in *Proceedings of Symposium on Operating System Design and Implementation*, November 2006, p. 22.

[16] F. Schmuck and R. Haskin, "GPFS: A shared-disk file system for large computing clusters," in *Proceedings of the Conference on File and Storage Technologies*, January 2002, pp. 231–244.

[17] C. A. Stein, J. H. Howard, and M. Seltzer, "Unifying file system protection," in *Proceedings of the 2001 USENIX Annual Technical Conference*, June 2001.

[18] H. Tang, A. Gulbeden, J. Zhou, W. Strathearn, T. Yang, and L. Chu, "The Panasas ActiveScale Storage Cluster - Delivering Scalable High Bandwidth Storage," in *Proceedings of SuperComputing*, November 2004, p. 53.

[19] M. L. Curry, A. Skjellum, H. L. Ward, and R. Brightwell, "Accelerating Reed-Solomon coding in RAID systems with GPUs," in *Proceedings of the 22nd IEEE International Symposium on Parallel and Distributed Processing (IPDPS)*, April 2008, pp. 1–6.