Parity File System

Himani Apte and Meenali Rungta

Computer Sciences Department
University of Wisconsin, Madison
{himani, meenali} @cs.wisc.edu

May 11, 2005

Abstract

Modern disks no longer operate in a simple "fail-stop" manner, yet commodity operating systems assume they do. We design and implement a parity based approach to improve the robustness of journaling file systems. We modify the existing ext3 file system for data and ordered journaling modes to incorporate parity and call it the "Parity File System". Using PFS, we are able to recover from a single latent sector error or silent block corruption within a given file. We show that the performance overhead for PFS compared to ext3 is minimal while the robustness is significantly improved.

1 Introduction

"Very little is needed to make a happy life" -Marcus Aurelius Antoninus

Disks fail. Unfortunately, they don't just fail like they used to traditionally, in a simple *fail-stop* manner (working or not). The fault model presented by modern disk drives is much more complex [2], owing to increasing complexity of today's disks and the pressures of time-to-market and cost increase[1]. For example, disks can exhibit latent sector faults, where a block or a set of blocks are inaccessible. Also, there may be silent data corruption in one or more disk blocks. These failures could be either transient or permanent.

However, commodity file systems are not able to handle these kinds of failures gracefully. Empirical study shows that both ext3 and IBM JFS are not designed to handle sector failures [1]. Under such failures, both of these file systems are shown to commit failed transactions to disk; this can lead to serious problems, including an unmountable file system.

In this paper, we investigate the use of parity as a technique to improve file system robustness. We enhance the design layout of ext3 file system to include parity over the data blocks of a single file. Our aim is to show that using such a design, a single failed data block of a file can be easily recovered. We focus only on recovery as detection techiques have been looked into previously [2,3]. We therefore assume that the initial fault is detected by a failure to read/write a block (in case of latent sector error) or by some other mechanism like checksumming (in case of block corruption).

We have modified the ext3 file system to incorporate a parity based design, which has been named as the Parity File System (PFS). We have analyzed the effects of these modifications on the performance of PFS under various benchmarks and under different journaling modes(data and ordered). We have also developed a tool that can be used to recover a given block's content offline (by specifying the block to be recovered). Our design allocates the parity block in the beginning of data blocks of the file; this ensures that access to the parity block is cheap. We show that PFS incurs minimal overhead in terms of both space

and time while significantly increasing the robustness of ext3 file system to localized disk faults.

However, there is definitely scope for further improvement in the design. Including parity over both data and metadata blocks would further improve the robustness of the system. Making the number of parity blocks per file a configurable parameter would give more control to the user. Also the current implementation is limited to support writes to a single file at a time. We are working towards improving our first prototype implementation of PFS to provide greater robustness.

The rest of the paper is organized as follows. In Section 2, we discuss the related work and in Section 3, we present a description of the design and structure of Parity File System. We evaluate the prototype implementation of PFS and present the results in Section 4 and conclude in Section 5.

2 Related Work

The parity based file system concept and the PFS borrow ideas from many different storage management systems and their analysis.

The extent of dependability of journaling file system was measured using fault-injection methodology [1]. This study uses a model-based approach to analyzing file system behavior in different journaling modes under various disk failures. This brings forth a number of design flaws and weaknesses of many popular journaling file systems including ext3, IBM JFS and Reiserfs under latent sector failures. Many of these file systems simply assume that sector failures do not occur and so are not designed to handle them, resulting in catastrophic effect on the on-disk data. The file system reaction may range from commiting failed transactions to disk leading to an unmountable file system to a more drastic measure like a system crash. This work provides a basis for the fact that there is a need for building more robust file systems that are not just aware of the possibility of sector failures but are also capable of handling and gracefully recovering from them. Our work extends in this direction to bring in robustness and recovery to an existing journaling file system namely ext3.

The Rusty File system [2] further motivates the problem using a *fractured failure model* for disks and classifying the file system failure policies using the Rusty taxonomy for failure detection and recovery techniques. Analysis of various commodity file systems illustrates lack of consistent failure handling and recovery policies. The authors evaluate the use of in-disk checksumming and replication as a means of improving robustness. However, checksumming can be used only for error detection and not error correction. Also, replication definitely adds a considerable space overhead. In our design of the PFS, we take a different approach to failure recovery based on parity. We find that this approach incurs minimal time and space overheads.

Various earlier work have addressed the issue of improving file system robustness against silent data corruption using data integrity checks. The Solaris Dynamic File system [3] developed by Sun Microsystems, Inc. uses copy-on-write transactional model with checksums stored with indirect blocks to detect phantom writes, misdirections and common administrative errors. Other systems like NetApp Filers and Tandem NonStop OS rely on end-to-end checksums. In all these approaches, the checksums can only be used to detect a block corruption. However, failure recovery is not enabled using this approach. On the other hand, PFS gives a major thrust on recovery from failures that includes single sector error where one block becomes inaccessible in a given file.

The parity based file system definitely draws on ideas from the classic RAID [4] approach to improving the performance, availability and reliability of multiple disk systems. Fourth and fifth level RAID use parity computed over the data blocks across muliple disks to enable recovery of failed blocks. The success of this approach to improving multi-disk reliability has motivated the extension of this concept to single disk systems. PFS is a prototype implementation of this concept where the parity is computed over multiple data blocks of the file system and used for reconstruction of failed blocks.

3 PFS Design

Linux ext3 is a journaling file system, built as an extension to the ext2 file system. In this organization, the disk is split into a number of block groups; within each block group are inode blocks, data blocks and bitmaps. The ext3 journal is commonly stored as a file within the file system. Linux ext3 includes three different journaling modes namely writeback mode, ordered mode and data journaling mode. The three modes are distinguished based on their consistency semantics for the metadata and data. The inode structure of ext3 includes 12 direct pointers, 1 indirect, 1 double indirect and 1 triple indirect pointer.

Parity File system uses the same code base as the Linux ext3 file system with minimal modifications to include the parity block. It maintains file based parity wherein each file has one parity block. The parity is calculated over all the data blocks of the file. To this end, the ext3 inode structure was modified to have 11 direct pointers instead of 12. Hence the last pointer available in the inode structure is used to track the parity block of the file.

The first implementation of PFS supported only data journaling mode of ext3 as this logs both metadata and data to journal. It thereby ensures an atomic update to file's data as well as the parity blocks. The second implementation of PFS extends its operation to also support the ordered journaling mode. In this mode, the parity block is treated as a metadata and hence is journaled along with the other metadata blocks. The inherent ordering constraint of this mode for the data writes to their fixed locations on disk and the journal writes of the metadata guarantees that both the data and metadata (including parity blocks) will be consistent after recovery.

3.1 Logical control flow

When the first block of a file is written, a parity block is associated with it and zeroed out. After the write completes, the contents of the parity block are the same as the contents of the first data block (equivalent to replication). On subsequent writes to different blocks of the file, the previous contents of the parity block are XORed with the contents of the new data blocks to reflect the updated parity as shown in Figure 1. If one of these blocks is later overwritten, then the new parity is calculated using the general formula:

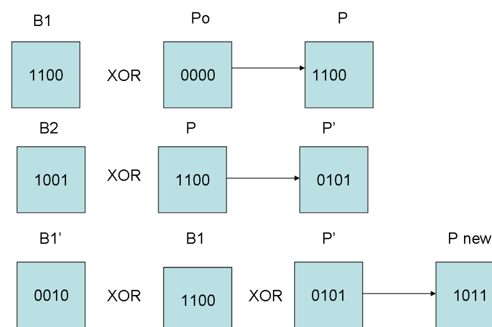

Figure 1: *Parity Computation: B1 - first data block, B2 - second data block, P - parity block, B1' - Modified B1*

$$new_parity = new_data \ XOR \ old_data \ XOR \ old_parity \quad (1)$$

3.2 Implementation details

The prototype implementation of PFS involved modifications to only two files in ext3, namely *ext3_fs.h* and *inode.c*. Overall the changes have been minimal.

The header file *ext3_fs.h* was modified to change the inode structure as follows:

```
/* Constants relative to the data blocks */
/* 11 instead of 12 direct pointers */
#define EXT3_NDIR_BLOCKS 11
#define EXT3_IND_BLOCK   EXT3_NDIR_BLOCKS
#define EXT3_DIND_BLOCK  (EXT3_IND_BLOCK + 1)
#define EXT3_TIND_BLOCK  (EXT3_DIND_BLOCK + 1)
#define EXT3_N_BLOCKS    (EXT3_TIND_BLOCK + 2)
/* in order to maintain 15 pointers in inode */
/* instead of EXT3_TIND_BLOCK + 1 */
```

The following modifications were made to *inode.c* to include parity:

Data Journaling mode

- ext3_prepare_write():
  - makes parity block using an existing function- 31 lines added
  - calls block_prepare_write: this tracks whether a new block is allocated for the write- 16 lines added
  - calls walk_page_buffers_prepare: this computes parity over old data blocks if the block was not newly allocated- 88 lines added
- ext3_journalled_commit_write(): calls walk_page_buffers_commit which computes parity over new data blocks and journals the parity block- 75 lines added
- ext3_block_to_path(): associates 15th pointer of the inode with the parity block- 2 lines changed

Ordered Journaling mode

The changes are essentially the same as the previous case. The differences are as mentioned below:

- ext3_prepare_write(): this calls walk_page_buffers_ordered which is a new function that computes parity over old data blocks, if the block was not newly allocated- 88 lines added
- ext3_ordered_commit_write(): This calls walk_page_buffers_commit which computes parity over the new data blocks and journals the parity block- 75 lines added

3.3 Recovery using PFS

The prototype implementation of PFS supports recovery of single failed block per file. A simple parity calculation over all the other data blocks and the corresponding parity block of the file is used to reconstruct the failed data block as shown in Figure 2.

We have also developed an offline *recovery tool* that can be used to reconstruct a failed data block of a given file. The user has to specify the file name and the logical block number that has to be reconstructed.

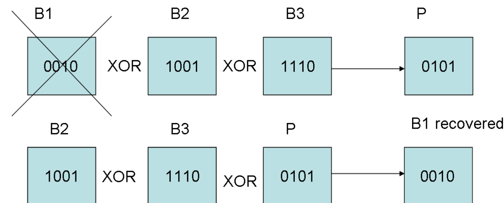

Figure 2: *Parity based Recovery: B1, B2 and B3 are the data blocks of a file, P - parity block of the file in PFS, Block B1 has failed and is reconstructed using B2, B3 and P blocks.*

The tool reads the inode of the file from the disk in order to obtain the disk block numbers of the data blocks and the parity block of the file. It then reads these blocks from the disk and computes parity over their contents in order to reconstruct the failed data block. This tool was used as a means for verification of the correctness of our PFS implementation and also to demonstrate the effectiveness of PFS in recovery from a single latent sector error per file.

4 Evaluating PFS

We now present the correctness and performance evaluation of PFS. We wish to answer the following questions:

- Does PFS work correctly?
- What time overheads are introduced?
- How many extra disk accesses occur in PFS?

4.1 Platform

PFS is implemented as a loadable kernel module (LKM) on Linux 2.6.11. All experiments were conducted on the Crash and Burn lab machine at the Computer Sciences Department at University of Wisconsin, Madison. The machine configuration is 1 GHz Pentium III processor with 512 MB RAM and one 20

GB IDE disk. DMA was enabled during all experiments and write-caching was turned off in order to ensure that the performance results measure the actual disk writes.

Does PFS work correctly? The prototype implementation of Parity File system was tested for correctness under various workloads. The first set of tests was done using the brute force method. This involved writing controlled data to a file's data blocks, keeping track of the disk block number of the parity block and then reading the contents of the parity block from disk to verify that the contents match the parity computation over the file's data blocks. The parity computations of PFS were verified over a range of workloads such as file create, append, overwrite, partial write etc. and was found to be correct in all cases.

Another approach used for verification was based on the recovery tool. Data block of a file was reconstructed using its other data blocks and the parity block and was then checked to ensure that the reconstructed block exactly matches the original data block. This also shows that in case of single-block failure per file, our prototype implementation of PFS can recover the failed block using the parity block.

What time overheads are introduced? A set of three different microbencmarks were conducted to test the PFS performance. The experiments were conducted for both data journaling mode and ordered mode. The results obtained were compared against the performance observed in ext3 file system. The time stamp counter available on the x86 architecture was used to measure the time required for each operation in the two file systems. The details of the experiments and the results obtained are discussed in the following section.

How many extra disk accesses occur in PFS? In order to be able to track the number of reads and writes that are sent to the disk for each operation, a pseudo device driver was used. This pseudo device driver is based on *semantic block-level analysis* (SBA) [5] which combines knowledge of on-disk data structures with the disk traffic generated by the benchmarks to record the quantity of disk traffic and how it is divided between reads and writes.

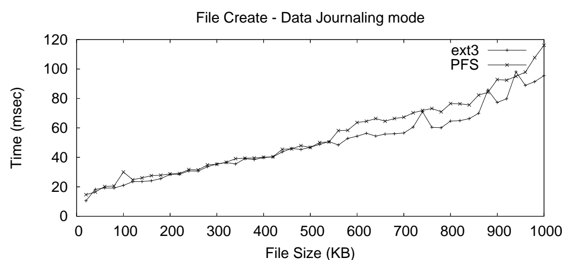

Figure 3: *File Create: The above graph plots the time taken (in msec) to create files of different sizes in data journaling mode for PFS and ext3. The performance of PFS is slightly worse than ext3 due to the extra disk write for the parity block.*

4.2 Benchmarks

This section gives a detailed description of the workloads used to compare the performance of PFS with that of ext3.

4.2.1 File Create

The microbenchmark for measuring performance for file creation consists of the following. Each run of the experiment first removed all the files and cleared the cache by unmounting and then mounting the file system. For a given file size, 21 files were created and time for file creation was measured using *rdtsc()*. The first few values were found to be skewed due to the overhead of reading the metadata blocks. Hence a few files of the given file size were first created, but not measured in order to warm up the cache with the metadata. The runs were conducted for file sizes ranging from 20 KB to 1 MB for both PFS and ext3. Also the number of disk reads and writes were simultaneously tracked using the SBA device driver. The measured values were averaged for each file size.

Data Journaling mode The performance results for file creation in data journaling mode are presented in Figure 3. The time taken for file creation in PFS is found to be slightly higher than that in ext3 as the extra overhead involved in PFS is the one extra disk write for the parity block. Figure 4 shows the number of disk writes incurred for the two file systems. The

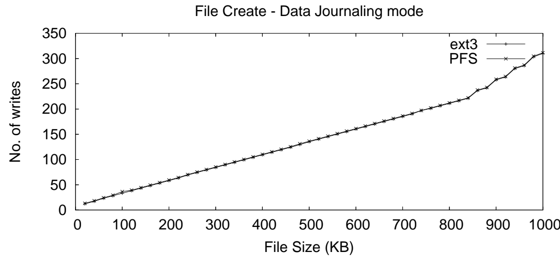

Figure 4: *File Create: The number of disk writes for PFS and ext3 for file create workload is plotted for different file sizes, which are observed to be almost the same.*

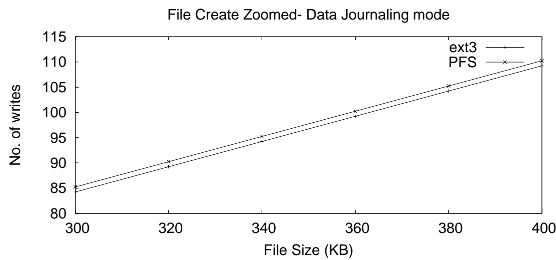

Figure 5: *File Create: This is an enlarged graph of Figure 4. PFS has exactly one extra disk write for the parity block compared to ext3*

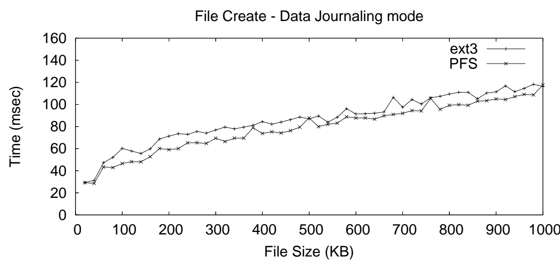

Figure 6: *File Create: The above graph plots the time taken (in msec) to create files of different sizes in ordered journaling mode for PFS and ext3. The performance of PFS is slightly worse than ext3 due to the extra disk write for the parity block.*

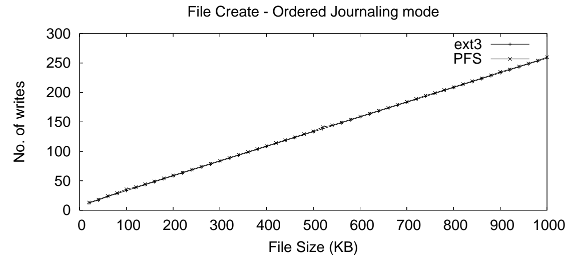

Figure 7: *File Create: The number of disk writes for PFS and ext3 for file create workload is plotted for different file sizes. PFS has exactly one extra disk write for the parity block compared to ext3*

number of writes observed include data and metadata blocks. For every data point, the number of disk writes for PFS is one more than that in ext3 as observed in Figure 5 which correponds to the write of the parity block.

Ordered Journaling mode Figure 6 shows the performance for the file create microbenchmark for PFS and ext3 in ordered journaling mode. The time taken in PFS is higher than that in ext3. The number of disk writes is shown in Figure 7. Similar to the results obtained for data journaling mode, the number of disk writes in PFS are found to be exactly one more than that in ext3 due to the parity block write.

4.2.2 File Overwrite

The file overwrite benchmark first cleared the file cache by unmounting the file system and then mounting it. Each run consisted of completely overwriting 20 different files of a given size with new data. The file sizes used in this experiment range from 20 KB to 1 MB. The file cache was cleared between each run of the workload. The time taken for each overwrite was measured separately and the average over the 20 iterations of each file size were computed.

Data Journaling mode The results obtained for the file overwrite microbenchmark in data journaling mode is presented in Figure 8. A 30% - 50% decrease in bandwidth (MB/s) is observed in PFS compared to that in ext3. This can be explained using the number of disk reads and writes for the two file systems.

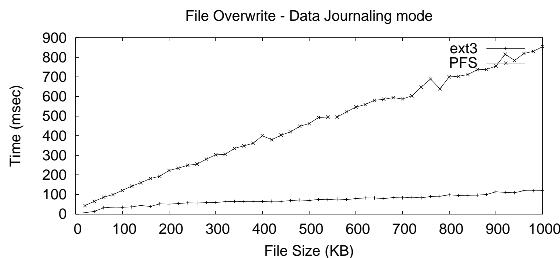

Figure 8: *File Overwrite: This figure shows the performance of PFS vs ext3 for file overwrite microbenchmark in data journaling mode. There is 30%-50% decrease in bandwidth in PFS compared to that in ext3 as PFS has to read the old data block for parity computation.*

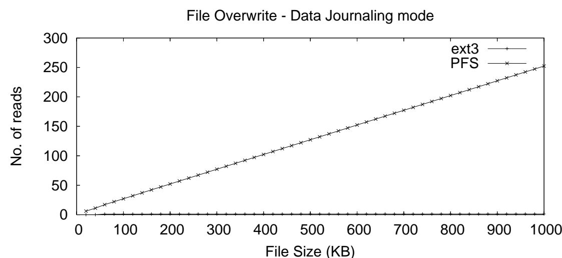

Figure 9: *File Overwrite: The number of disk reads for file overwrite microbenchmark is plotted for different file sizes in data journaling mode. PFS has to read the old data blocks from the disk for parity computation and hence has a very large number of disk reads compared to ext3.*

As observed in Figure 10, the number of disk writes in PFS is just one more than that in ext3 which accounts for the parity block write. The considerable overhead is observed because PFS has to read an old data block from disk per block overwrite in order to compute the parity. This is clearly observed in Figure 9 which shows the number of disk reads in PFS versus those in ext3. However, this is not a huge overhead as far as realistic workloads are concerned, as the percentage of bytes that sequentially overwrite existing files is extremely small as compared to the percentage of bytes that are appended to files [6].

Ordered Journaling mode The results obtained for the file overwrite microbenchmark in ordered journaling mode are the same as that in data journaling mode. The considerably higher overhead in case of PFS as observed in Figure 11 can be accounted from the fact that the number of disk reads in PFS are much higher than in ext3 as shown in Figure 12. There is only one extra disk write (Figure 13) in PFS compared to that in ext3 for each file size.

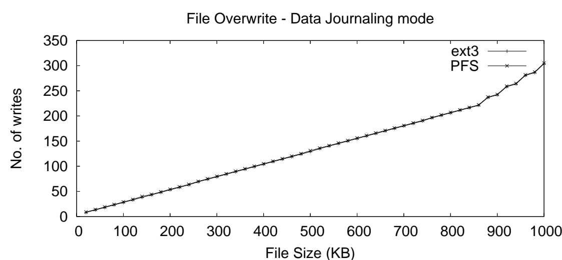

Figure 10: *File Overwrite: The number of disk writes for file overwrite microbenchmark is shown for different file sizes in data journaling mode. PFS has only one extra disk write for the parity block compared to ext3.*

4.2.3 File Partial write

This microbenchmark was used to compare the performance of PFS and ext3 for partial writes. The file cache is cleared before each run of the experiment. Each run consists of 2 KB writes (half of the block size) to the data blocks of 20 different existing files

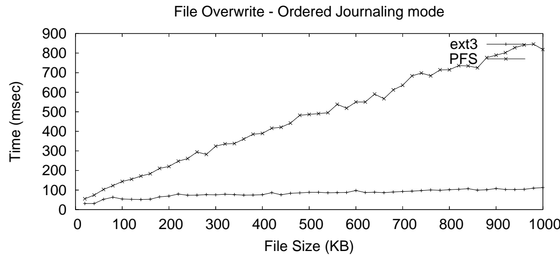

Figure 11: *File Overwrite: This figure shows the performance of PFS vs ext3 for file overwrite microbenchmark in ordered journaling mode. There is 30%-50% decrease in bandwidth in PFS compared to that in ext3 as PFS has to read the old data block for parity computation.*

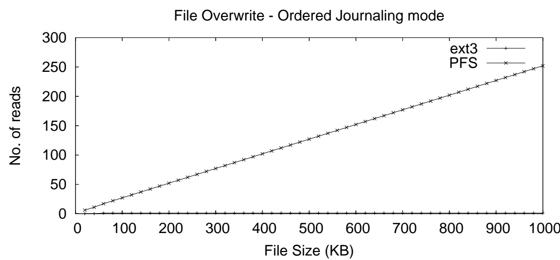

Figure 12: *File Overwrite: The number of disk reads for file overwrite microbenchmark is plotted for different file sizes in ordered journaling mode. PFS has to read the old data blocks from the disk for parity computation and hence has a very large number of disk reads compared to ext3.*

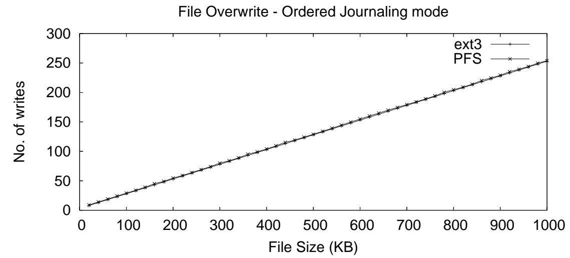

Figure 13: *File Overwrite: The number of disk writes for file overwrite microbenchmark is shown for different file sizes in ordered journaling mode. PFS has only one extra disk write for the parity block compared to ext3.*

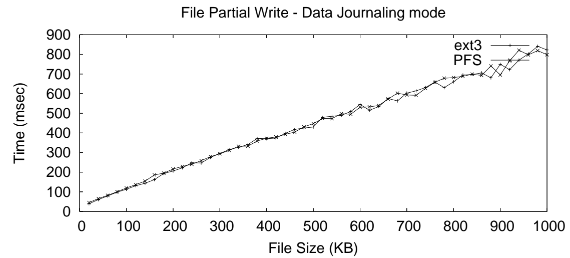

Figure 14: *File Partial write: The above graph plots the time taken (in msec) for partial writes to files of different sizes in data journaling mode for PFS and ext3. The performance of PFS is almost the same as that in ext3 as partial write does not incur extra overhead in PFS.*

of given size. This benchmark gives insight into the performance of PFS when the file is partially overwritten.

Data Journaling mode The time taken for partial writes for different file sizes as plotted in Figure 14 demonstrates that PFS and ext3 have very similar performance as there is no extra overhead involved in PFS. This is further validated using the observation from Figures 15 and 16 that the number of disk accesses for the two file systems are almost the same.

Ordered Journaling mode The performance results (Figure 17) obtained for partial write workload in ordered journaling mode for the two file systems and the number of disk accesses observed as shown

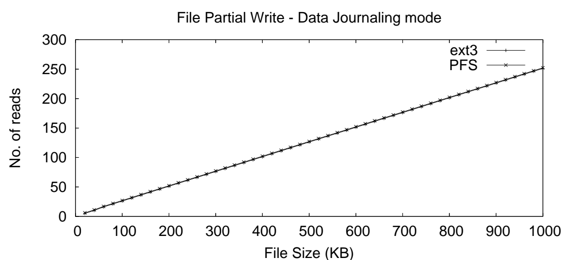

Figure 15: *File Partial write: The number of disk reads for partial file write for different file sizes is shown for PFS and ext3. Both file systems have exactly the same number of disk reads.*

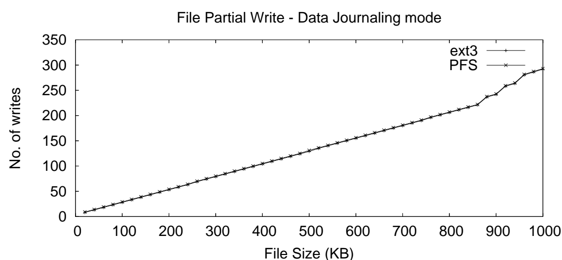

Figure 16: *File Partial write: The number of disk writes for partial file write for different file sizes is shown for PFS and ext3. PFS has exactly one extra disk write compared to that in ext3.*

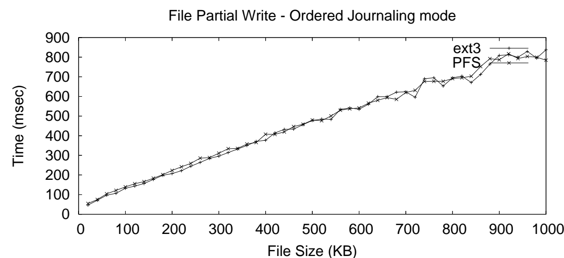

Figure 17: *File Partial write: The above graph plots the time taken (in msec) for partial writes to files of different sizes in ordered journaling mode for PFS and ext3. The performance of PFS is almost the same as that in ext3 as partial write does not incur extra overhead in PFS.*

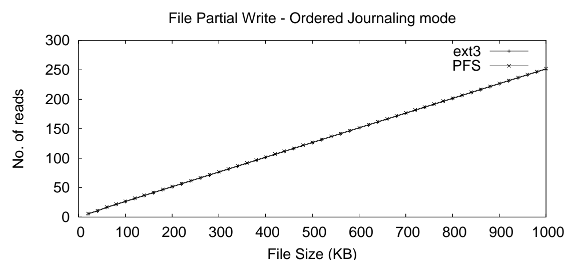

Figure 18: *File Partial write: The number of disk reads for partial file write for different file sizes is shown for PFS and ext3. Both file systems have exactly the same number of disk reads.*

in Figures 18 and 19 are simlar to those observed in the data journaling mode. PFS performs almost the same as ext3 in this case.

5 Conclusion

Commodity operating systems have not been designed to handle localized disk faults such as latent sector errors and block corruption. Thus they are not robust against the kinds of failures prevalent in modern disk drives.

In this paper, we design a parity-based approach to enable journaling file systems to recover from a fault localized to a given file. Using the design, we build

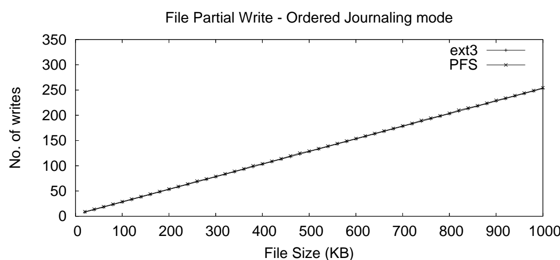

Figure 19: *File Partial write: The number of disk writes for partial file write for different file sizes is shown for PFS and ext3. PFS has exactly one extra disk write compared to that in ext3.*

a new file system that is robust to such failures. We evaluate the performance and number of disk accesses of the new file system as compared to ext3 for file create, overwrite and partial write cases. From our anaylsis, we find that PFS perform similar to ext3 in file create and partial write cases and shows a 30%-50% degradation in bandwidth in case of overwrite. We prove that parity is a simple and efficient way to improve the robustness of modern file systems.

Many challenges remain: The design needs to be reworked to handle multiple faults in a given file. This can be done, for example, by having one parity block over every 200 blocks of the file instead of over the entire file. The design also needs to be revamped to handle failures of consecutive blocks, which are common. This can be done by having row and column parity instead of having just a row-based parity. By working further in this area, it may be possible to begin an entirely new generation of "fault-aware" robust file systems.

6 Acknowledgements

We would like to thank Professor Remzi Arpaci-Dusseau for his excellent and encouraging guidance throughout this project. We would also like to thank Vijayan Prabhakaran for his thoughtful suggestions throughout the design and implementation process and for providing us with the pseudo device driver for semantic block level analysis. Finally, we thank the Computer Systems Lab for providing the infrastructure for conducting this project.

References

[1] Vijayan Prabhakaran, Andrea C. Arpaci-Dusseau and Remzi H. Arpaci-Dusseau, *Model-Based Failure Analysis of Journaling File Systems*, DSN 2005.

[2] *Rusty File Systems*, under submission.

[3] *Solaris Dynamic File System*, Sun Microsystems Inc., http://members.visi.net/ thedave/sun/DynFS.pdf

[4] David A. Patterson, Garth Gibson and Randy H. Katz, *A Case for Redundant Arrays of Inexpensive Disks (RAIS)*, Proceedings of the 1988 ACM SIGMOD Conference on Management of Data, Chicago IL, June 1988.

[5] Vijayan Prabhakaran, Andrea C. Arpaci-Dusseau and Remzi H. Arpaci-Dusseau, *Analysis and Evolution of Journaling File Systems*,Usenix 2005.

[6] Baker, M., Hartman, J., Kupfer, M., Shirriff, K., and Ousterhout, J., *Measurements of a Distributed File System*, Proceedings of the Thirteenth Symposium on Operating System Principles, October 1991.