Improving Ext2 Integrity with Checksums

Chloe Schulze, Brian Pellin
{cschulze,bpellin}@cs.wisc.edu

Computer Sciences Department
University of Wisconsin, Madison

Abstract

We have modified the ext2 file system to include checksumming on a per block basis. Our goal is to follow a realistic failure model, which accepts that disks can partially fail and do so silently. The addition of checksumming to ext2 detects silent data corruption and notifies the user if data has been changed on disk without his/her knowledge. This solution may also solve some other errors, such as malicious changes to data. We make no attempt to recover lost data, only to notify the user of the occurrence. Our implementation is comparable to the performance of ext2, with the addition of overhead incurred from reading, writing and computing checksums. We broke down the components of the overhead and found that for warm cache reads the computation of the checksum dominates CPU costs. Additionally, for cold cache operations, we found that the time is severely dominated by the disk positioning to read the checksums.

1 Introduction

Reliability is a major source of concern in file system design, and the main issue behind it is data loss or data corruption. The weak links in this chain are disks. Disk errors, though possibly few and far between, pose a problem for data integrity. Though precautionary actions, such as regular backups, are essential for data recovery and can greatly reduce the problems associated with disk failure, those efforts do not help if it is not apparent that the disk has failed. Disks do fail, however, they do not always do so completely, making it hard to recognize exactly when (or if) the failure occurred. Obviously, for a system to actually meet the claim of reliability, this is an unacceptable mode of operation.

There are two issues that need to be addressed in order to solve this problem. The first is a need for a new disk failure model. No longer can we assume a fail-stop model, and so we develop a new paradigm that takes into account the gray area of silent failures. The second issue is one of detection and notification. The data corruption must first be identified and then the user and/or file system must be notified of the occurrence.

In this paper, we describe a new file system, which deals with several of the aforementioned problems. This system, named Second Extended File System with Checksumming or ext2c, is a modified version of ext2 that uses checksums to verify the correctness of file data.

The paper will proceed as follows: in Section 2 we describe our failure model, then in Section 3 we outline ext2c and our checksumming mechanism. Performance is addressed in Section 4. Section 5 considers further work to improve ext2c. Section 6 discusses related work and in Section 7 we conclude.

2 Fault Model

For a disk failure model to be accurate, it must take into account all the ways in which a disk can fail. Examples are:

- Full disk failure: entire disk stops working, can be permanent or temporary.

- Partial disk failure: latent sector errors, sectors of a disk go bad, can also be permanent or temporary.

- Data corruption caused by improper disk operation. This includes misdirection (data written to the wrong block) and phantom writes (the disk returns that the write completed, but it did not actually complete)[6]

These errors can all cause data to be lost or corrupted without the user's or file system's knowledge.

Keeping the above types of errors in mind, a more accurate failure model is one in which a disk can fail silently or experience data corruption at anytime with no indication that the data may be incorrect. Using this fault model, we designed a system to handle several of these types of failures.

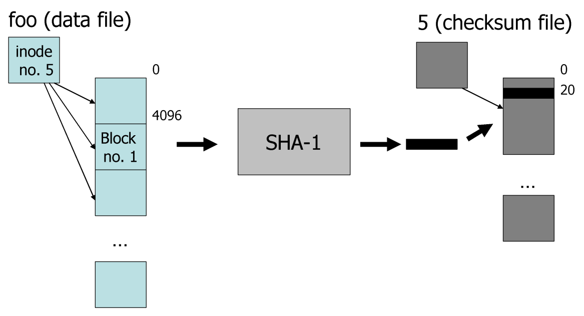

Figure 1: **Checksum Creation.** *This figure shows the series of steps that happen when a block is written to a file.*

3 EXT2C

The goal of ext2C is to detect silent data corruption and notify the user when it occurs. To facilitate this goal, we have modified ext2 to compute and store checksums. The file system operates by computing a checksum whenever data is written and storing that checksum in a file. When data is read, a new checksum is calculated and then compared to the stored one. If the checksums do not match, an error is returned to the user.

3.1 Checksum Granularity

Various design choices of where and how to store checksums are discussed in the paper by Sivanthanu, et. al. [8]. We chose to compute checksums on a block by block basis, per data file, and then store those checksums in a separate file. The checksum files are named according to their respective data files' inode numbers, and these in turn are stored in a checksum directory. Each checksum is 20 bytes in length, and can be indexed by the block number of the block in the file that it represents. Figure 1 demonstrates the creation of a checksum.

3.2 EXT2C Write

When compared to other implementations, there are several advantages to computing checksums in this manner. To begin with, the granularity of the checksum is such that certain blocks of a file can be pinpointed without invalidating the entire file. Additionally, a block of a file can be updated without having to read the entire file (to compute a checksum over it).

There are also advantages to storing the checksums in a separate file as opposed to storing them within the file data. This implementation is simpler than attempting to store the checksums inline (within each block) or in some part of the inode or its indirect blocks, because there is no need to modify the any of the structures of the file system. Also, because of the size of each checksum, about two hundred blocks of a file can be read using only one block of the checksum file. This is obviously advantageous for long, sequential reads. Though other solutions have definite advantages, such as ease of access, we decided to forgo the gains (and added complexity) of their implementation and leave them as further work and optimization to the system.

3.3 Computing the Checksum

To compute the checksum over a block of data, we use the SHA-1 secure hash function[2].[1] Our main reason for using such a hash function is to minimize the possibil-

[1]Although it has recently been confirmed that some collision attacks exist on SHA-1, these attacks do not make it easy to find a collision for an arbitrary hash value. So, it is still adequate for our purposes.

ity of a collision of checksum results, and additionally, to have a fixed size output for a variable size input. SHA-1 produces a digest of 160 bits, which is highly unlikely to collide with another hash of a different input.

3.4 Changes to EXT2

We decided to use ext2 as a basis for our implementation because of its simplicity. Further work could be done in extending this concept to other file systems or even to VFS, which would be another compelling project.

To add checksumming to ext2, we wrote our own wrapper functions for the normal ext2 read and write calls. We then modified the mapping from the system call read and write to point to our new functions. Both of the new functions work generally in the same manner.

3.5 EXT2C Write

The write function first calls the regular ext2 write call. It then determines the number of blocks changed in the file and the index of the first changed block. The newly written data is then read in, and a checksum is computed over each block. At this juncture we do not check the validity of the old checksum because it is being written over and so it should not matter if the data has been corrupted. We then open up the checksum file and write the new checksum to the appropriate place (which is the index of the file block * 20 bytes). We do this for each changed block in the file, and then we close the checksum file and return the result of the initial ext2 write.

The additional overhead in this function is the opening and closing of the checksum file, the computation and writing of the checksum to the checksum file, and the extra read that pulls in the newly written data.

3.6 EXT2C Read

Analogous to ext2c write, the read function first performs a regular read call. Then the function computes the number of blocks that are being read and the index of the first block and then reads each of those blocks in one at a time. As each block is being read in, a checksum is computed on that block and then it is compared to the corresponding checksum from the checksum file. If the checksums do not match, an error message is thrown[2] and the read does not complete. If the two match, however, the operation proceeds and at the end of the function the result of the initial ext2 read call is returned. Figure 2 outlines the steps taken by the ext2c read function.

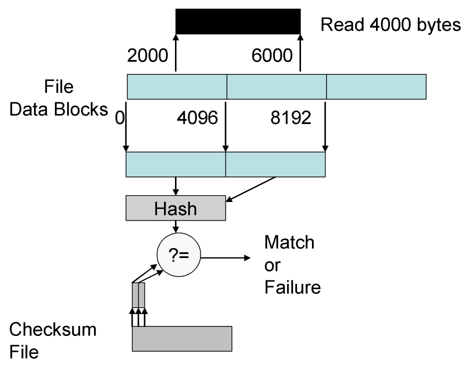

Figure 2: **Read Operation.** *This figure shows the series of steps that occur during the ext2c read.*

The added overhead incurred by the read function is similar to that of the write function. There is an extra data read, the read of the checksum, the computation of a checksum and the open and close of the checksum file.

3.7 Reading Unwritten Data

In a few cases the Linux file system semantics allow for reading data from a file that has not be written. If the file pointer is advanced beyond the end of a file and the file is written to at that point, then the data in-between the old end of the file, and the start of the new write can be read. For example, one can create a new file, seek to byte 4096 and write data. The semantics state that a read from the first 4096 bytes should return all 0's.

Originally, this created a problem for us. Our initial implementation only updated checksums for blocks of the

[2]If the match fails, the read call will return -1 and errno will be set to ECHECKSUM.

file that had changed. However, in one of these special writes, the automatically extended part of the file is changing as well. Consider the above example where a file is created and data is written to the second block of the file. The checksum gets written for the second block of the file, but not the first. Now, if we try and read from the first block of the file, we will get a checksum mismatch.

The solution, is to detect during writes when this situation occurs. We do this by monitoring the effect of a write on a file's length. If this creates any new blocks, then the code considers the new blocks to overlap with the write as well.

3.8 Targeted Problems

Ext2c can completely or partially detect the following problems: bit rot, phantom writes, misdirection and malicious writes. Bit rot, or the spontaneous flipping of bits, is completely detectable because we have the guarantee that the checksum has been written to disk. Therefore, when the data is read from disk, the checksum that is computed over the data will not match the stored checksum.

With the latter three problems, we no longer have the guarantee that the checksum is written to disk. With phantom writes, if the checksum is written but the data is not, then the error is detectable. However, if both are not written, or if the data is written but the checksum is not, then the data corruption will not correctly be detected. Misdirection is similar; if the checksum is written correctly then ext2c will detect the corruption, otherwise the data will be silently corrupted. The detection of malicious writes depends on the nature of the malicious entity. If the malicious entity directly modifies the disk, then the write is similar to bit rot, and ext2c can detect the corruption. However, if the malicious entity is either smart enough to write checksums, or somehow uses our modified read and write functions than the error will be undetectable.

In summary, ext2c can detect most normal (i.e unmalicious) error occurrences. However, because the checksums are also stored on disk, it is clear that the same types of problems plaguing data are extendible to the checksums themselves. A solution to this problem would be to store the checksums somewhere else, such as nonvolatile memory. We leave this problem as further work.

4 Results

In this section, we first look at how we evaluated the correctness of our implementation. That is followed by an analysis of the performance of ext2 versus ext2c.

In our performance section, we closely examine the costs of individual small operations. Then, we see the results of some benchmarks that better capture the behavior of real workloads.

Our experiments were conducted on a Pentium III 700 MHz machine with 128MB of memory. The disk running our file system is a Western Digital Protègè[3] that runs at 5400 RPM. Both file systems use a block size of 4 KB. Ext2c is based on the implementation of the Second Extended File System (ext2) in the 2.6.11.6 version of the Linux kernel.

4.1 Implementation Correctness

In order to verify that our implementation is correct, we subjected the file system to a variety of workloads. In order to cover all bases, there are several read/write situations to consider:

- I/O's aligned with the block boundary
- I/O's smaller than a block
- I/O's larger than a block
- I/O's that cross block boundaries
- I/O's that extend files
- All combinations of the above

We first tested all of the above for writes. We verified that for the writes issued, the correct blocks were identified as needing updates, and that the appropriate places in the checksum file were updated. Then, we did the same for reads. This time also verifying that the checksums matched the hash of the data stored.

As an additional assurance of correctness, we were able to run all of the benchmarks mentioned later in the paper without any checksum errors.

We also needed to ensure that ext2c accurately detects when checksums do not match data.

[3]Model Number: WD100EB-00BHF0

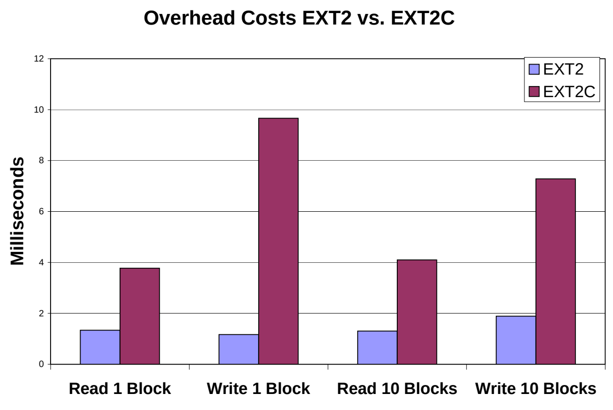

Figure 3: **Cold Cache Small Read/Write.** *This graph shows the comparative costs of small reads and writes in ext2 and ext2c.*

Since we did not change any of the on disk formats from ext2, ext2c file systems can be mounted as ext2 file systems. When mounted this way, data can be altered without the checksums being updated. Additionally, a checksum file can be edited just like any other file.

We used these combined abilities to inject errors into the file system. Ext2c successfully detected errors both when the checksums were corrupted as well as when the data was corrupted.

4.2 Small Scale Performance

An important aspect of ext2c to study is the breakdown of the cost of individual operations as compared to ext2. Since our file system is an extension of ext2, it makes the perfect baseline. It is identical in every respect to ext2c, with the exception of the additions mentioned in Section 3.

4.2.1 Cold Cache Behavior

First, we consider the cold cache behavior of reads and writes. In these benchmarks, every read has to retrieve data from the disk and every write must to commit its data to the disk.

Figure 3 shows the results of an experiment in which we perform 1 and 10 block reads and writes. The operations are performed on 80KB files.

Reading and writing single blocks takes about 3 times longer in ext2c. Some of this extra time is expected, because we are accessing extra data on the disk, namely the checksum file. The reason the cost is 3 times as expensive is mainly due to the disk's positioning time when moving from the data block to the checksum block.

The fact that the time is dominated by positioning also explains why 10 block operations are very similar to the single block operations. The time it takes to access the 9 additional data blocks is not significant when compared to the repositioning of the disk in order to read the checksum block. Also, it should be noted that in this case the checksum block on disk only needs to be accessed once. The first checksum block is cached on its initial use, and it is then reused from memory for the subsequent data blocks.

This suggests that performance is highly sensitive to the position of the checksum file relative to the regular data. Due to this fact, a method for placing checksum files near data is highly desireable.

4.2.2 Warm Cache Behavior

As a consequence of our implementation, checksum consistency is maintained not only on the disk, but in the memory cache of disk blocks, as well.

This leads to the question, how expensive is maintaining checksums in memory? In Figure 4, the 'Normal Read' bar represents the time it takes to complete ext2 reads. The other sections of the bar represent components of additional cost needed to perform an ext2c read. Thus, the full bar is the total time required for an ext2c read.

The first added cost, is the time it takes to perform the open and close on the checksum file. Though this is not a large amount of the total cost, we could probably eliminate it, by only open/closing the checksum file on the data file's open/close.

Reading from the checksum file and the extra read from the data file make up the next component of cost. Not much can be done to reduce this cost. The data needs to be read from the file to feed as input to the hash function. The hash also needs to be read to verify the file data.

The last component is by far the most significant. Checksumming the data takes about ten times as long as servicing a normal read from cache. This suggests that the

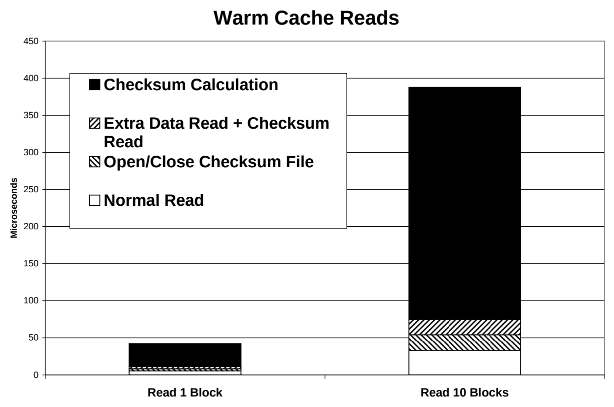

Figure 4: **Warm Cache Small Reads.** *This graph shows the break down of costs in a cached read. The bottom bar represents the time it takes to do a normal ext2 read. The next bar is the extra time required to open and close the checksum file. The next shows the costs for performing additional memory copies of the data and the checksum. The final shows the cost for calculating the checksum.*

hash function is a significant bottleneck for in memory file operations.

The hash function's time is not as significant when compared to how long it takes to service a disk request, so its efficiency was not something we initially considered. However, these results suggest that an efficient hash function is important to cache performance.

4.3 Large Scale Performance

While small scale micro-benchmarks are useful for determining how the mechanics of a file system perform, they do not give a good picture of the impact of those results.

To better understand the impact of performance losses, we take a look at benchmarks that represent more realistic workloads.

4.3.1 PostMark

PostMark[4] is a benchmark designed to simulate realistic small file workloads. It performs random pairs of create/delete and read/append operations in order to limit the influence of file system caching and read-ahead techniques.

PostMark Component	**EXT2**	**EXT2C**
Total Transactions	5000	2500
Create	500	500
Read	2499	1249
Append	2483	1241
Delete	628	628

Figure 5: **PostMark Results (transactions / second)** *The results of running PostMark for 10,000 transactions, even read/append bias. The numbers are a measure of operation throughput.*

The results from our test are presented in Figure 5. Create and delete operations perform at the same rate for both file systems, because ext2c does not perform any additional work for these operations.

For both the read and append operations, ext2c accomplishes about half of throughput as ext2. This is fairly expected. Ignoring the effects of caching, checksumming approximately doubles the number of I/Os required to complete reads and writes. The reason why the performances losses as not as bad as seen in Section 4.2.1, is mainly because PostMark creates small files which tended to have their checksum files closer to file data. This made the disk operations closer to sequential and hence, more efficient.

This has several implications. Small file workloads are rather common for file systems. So, a loss of one half the throughput could be a heavy detraction from the desire to use ext2c. However, many people are willing to tolerate extra latencies to gain the benefits of network file systems[7], so if the benefits of checksumming are equally desired, this is a reasonable trade off to make.

4.3.2 Large Sequential Reads

Another important file system workload is large sequential operations. This is typically when disks reach their best performance, because positioning time is amortized over long data accesses. Current disk technology typically performs much worse with respect to random accesses.

It has already been seen that extra postioning time is introduced in ext2c when the file system moves from accessing data to acessing checksums. Since a single check-

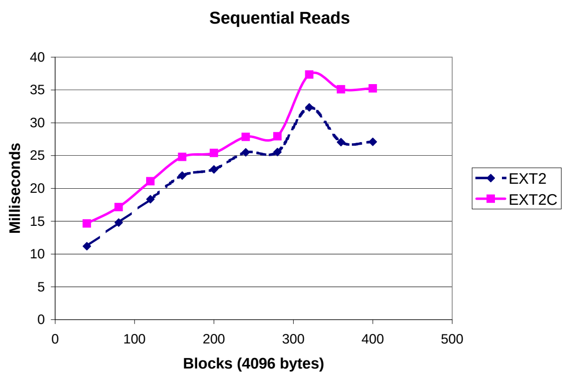

Figure 6: **Sequential Reads.** *Demonstrates the ext2 vs. ext2c times for large reads. Ranges from size 40 blocks to 400 blocks.*

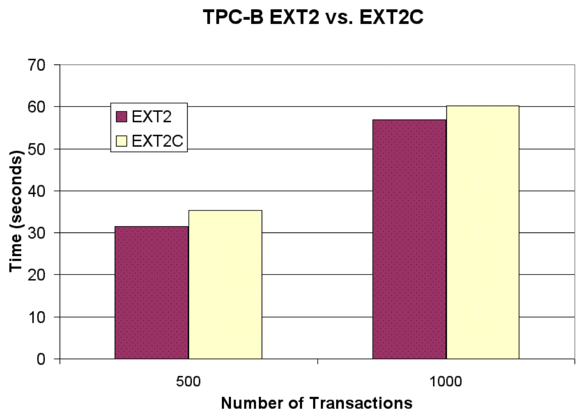

Figure 7: **TPC-B.** *The results of the TPC-B benchmark run both on the ext2 and ext2c file system. TPC-B is a transactional processing benchmark.*

sum block holds checksums for over 200 data blocks, caching the checksum block will allow ext2c to amortize its extra positioning time, as well. Since checksums are treated as any other file, accesses to them benefit from from all of the caching technicues employed by ext2.

This is demonstrated in Figure 6. There is a fixed additional cost for using ext2c for large reads. This fixed cost mostly represents postioning time to retrieve the checksum block for the first time. Then, the checksum block stayes in memory cache for the next 200 data block reads.

Checksumming during large sequential operations is relatively cheap. For workloads that involve these operations (archival, large media storage), it is much easier to justify the price of checksumming.

4.3.3 TPC-B

TPC-B[1] is a benchmark designed to represent Database Management System workloads. As such, it simulates the workload of several clients connecting to a backend and making transactional requests, like those of a bank or ATM. The results can be interpreted to determine how much load a server can handle.

Figure 7 displays the results of running the TPC-B benchmarks with 500 and 1000 transactions. In this case, ext2c performs only slightly worse than ext2. We can attribute the good performance to effective caching of checksum blocks. Thus, seeks to the checksum files are amortized over large data reads.

This suggests that ext2c can handle transactional workloads as well as it handles large sequential reads. This is promising, because transactional workloads occur in places such as banks, where data integrity is critical.

5 Further Work

Throughout the paper we have briefly mentioned further optimizations and improvements that could be made to ext2c, we discuss them in more detail here.

Our performance results have shown that the overhead experienced by ext2c over ext2 is primarily related to rotational positioning time of the disk to access the checksum files. In light of this finding, it is clear that further study of the effects of varying the locations of the checksum files needs to be made to develop an optimal positioning scheme. A method of positioning the checksums on disk could greatly reduce the disk access overhead and so reduce the overall time taken by reads and writes.

Another source of CPU bound overhead is the hashing function used to compute the checksums. Further work could be directed at finding a more efficient hash function and so reduce that cost.

An additional optimization mentioned in the performance section would be to open and close the checksum files along with the data files, instead of within the read and write functions. This would cause a small reduction in the overhead of reading and writing (as shown in the performance section). However, opening and closing the checksum file along with the data file might make a big difference if the file is left open for awhile and many read/write operations are performed on it.

6 Related Work

A lot of related work has been done to solve or recover from disk failures. Some implementations, however, have a more complex goal then just the detection of data corruption.

File systems such as I3FS[3] and PFS[10], seek to protect against malicious data change. File systems and data integrity can be seriously compromised through attacks on the system and these two file systems use checksumming and other techniques to identify such breaches of security and verify correctness of data. Though we do mention that we can partially detect malicious writes, this is obviously not our main concern with respect to the implementation of ext2c.

There are also systems that are more comparable to our goals. The Solaris Dynamic File System[11] and RUSTY[6] are both file systems, whose primary aim is to verify data integrity in the face of disk related errors. Both use checksumming and transactional updates or data replication in addition to other policies to provide a sense of security of data to the user.

There are other systems that use different techniques to ensure data persistence. Systems such as RAID[5] and D-GRAID[9] attempt to solve disk problems by relying on strength of numbers. These systems utilize many disks, as well as schemes of data replication or parity to prevent data loss. Typically, however, these systems are designed with the fail-stop model of fault detection in mind an so are only applicable when a full disk fails. Therefore, arrays of disks are unlikely to notice silent data corruption and so cannot give a complete guarantee on the correctness of data. It is important to emphasize, however, that the goals of these types of systems are more along the lines of reliability of disks and performance, and so it is natural that do not guarantee the correctness of data.

7 Conclusions

We have added additional functionality to the ext2 file system to enable us to give certain guarantees to the user about the correctness of his/her data. Our aim is to ensure that the data read by the file system is not corrupted. To accomplish this goal, we first developed a more realistic fault model for disks that accounts for the fact that disks can corrupt data silently. We then followed this model by adding checksumming to ext2 to create a new file system ext2c. Ext2c works by checksumming data that is written to disk and then when reading data back, it computes a new checksum and compares it against the stored checksum. If the two checksums do not match, then the data has become corrupted on disk and an error message is returned to the user. In this manner we provide certain guarantees to users about their data.

Through testing with PostMark we have discovered that ext2c has generally half the throughput of ext2 on small files. This is due mainly to the cost of positioning the disk to read from the checksum files. We find that on large sequential I/Os, however, that the positioning cost is amortized, and so the additional cost over ext2 in sequential I/O is fixed. The same is reflected in the TPC-B benchmark. Additionally, all of our benchmarks served to test the correctness of ext2 as well as the performance. Overall, it is clear that there are some performance losses in using ext2c, but under workloads where caching is effective, the loss of performance is modest.

As further work, we could optimize our implementation in several ways in order to improve its performance. The most important improvement would be to find an optimal placement for the checksum files on disk to reduce the positioning time of the disk. A second valuable optimization would be to find a more efficient hashing function so as to lower CPU costs when updating checksums in memory.

In general, we have found that checksums help to validate data at the cost of some throughput. As with everything, the usefulness of our file system is determined by trade offs based on the user's priorities. Depending on the workload required, the cost of added integrity may be high, but for the right workloads, integrity can be gained at a cheap cost.

8 Acknowledgments

We would like to acknowledge Remzi for his excellent suggestions and guidance. We would also like to thank Vijayan Prabhakaran for providing help and access to benchmarks.

References

[1] Transaction processing performance council. tpc-b. `http://www.tpc.org`.

[2] P. A. DesAutels. Sha1: Secure hash algorithm., 1997. `www.w3.org/PICS/DSig/SHA1_1_0.html`.

[3] A. Kashyap, S. Patil, G. Sivathanu, and E. Zadok. I3FS: An In-Kernel Integrity Checker and Intrusion Detection File System. In *Proceedings of the 18th USENIX Large Installation System Administration Conference (LISA 2004)*, pages 69–79, Atlanta, GA, November 2004.

[4] J. Katcher. Postmark: A new file system benchmark. Technical Report TR3022, Network Appliance Inc., October 1997.

[5] D. A. Patterson, G. A. Gibson, and R. H. Katz. A case for redundant arrays of inexpensive disks (raid). In H. Boral and P.-Å. Larson, editors, *Proceedings of the 1988 ACM SIGMOD International Conference on Management of Data, Chicago, Illinois, June 1-3, 1988*, pages 109–116. ACM Press, 1988.

[6] V. Prabhakaran, N. Agrawal, L. Bairavasundaram, H. Gunawi, A. Arpaci-Dusseau, and R. Arpaci-Dusseau. RUSTY file systems. Draft, April 2005.

[7] R. Sandberg, D. Goldberg, S. Kleiman, D. Walsh, and B. Lyon. Design and implementation of the Sun Network Filesystem. In *Proc. Summer 1985 USENIX Conf.*, pages 119–130, Portland OR (USA), 1985.

[8] G. Sivathanu, C. P. Wright, and E. Zadok. Enhancing File System Integrity Through Checksums. Technical Report FSL-04-04, Computer Science Department, Stony Brook University, May 2004. `www.fsl.cs.sunysb.edu/docs/nc-checksum-tr/nc-checksum.pdf`.

[9] M. Sivathanu, V. Prabhakaran, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Improving storage system availability with D-GRAID. In *Proceedings of the USENIX FAST '04 Conference on File and Storage Technologies*, pages 15–30, San Francisco, CA, March 2004. University of Wisconsin, Madison, USENIX Association.

[10] C. A. Stein, J. H. Howard, and M. I. Seltzer. Unifying file system protection. pages 79–90.

[11] G. Weinberg. Solaris Dynamic File System. `http://members.visi.net/˜thedave/sun/DynFS.pdf`