Checksumming RAID

Brian Kroth
bpkroth@cs.wisc.edu

Suli Yang
suli@cs.wisc.edu

Abstract

Storage systems exhibit silent data corruptions that go unnoticed until too late, potenially resulting in whole trees of lost data. To deal with this, we've integrated a checksumming mechanism into Linux's Multi-Device Software RAID layer so that we are able to detect and correct these silent data corruptions. The analysis of our naive implementation shows that this can be done with a reasonable performance overhead.

1 Introduction

Storage systems, perhaps more than other computer system components, exhibit silent partial failures. These failures can occur anywhere along the storage stack: from the operating system arranging and scheduling requests, to the device drivers, through the communication channel, to the backing device medium itself and all of its internal firmware and mechanical underpinnings. The failures themselves can be partial or complete, temprorary or permanent, and, despite some hardware support for it, detected or not. They occur in the form of bit flips along the path, bit rot on the medium over time, misdirected writes to the medium, and phantom writes that never actually make it to the medium.

At the same time, though perhaps unwise, most software in a system presumes a stop-failure environment in which all the underlying components either completely work or fail in their entirety. For example, if a region in memory is damaged it is likely that either the program will crash, or in the case of virtual memory, perhaps the machine will crash. These particular problems are probably not the end of the world though, since the program or the machine can simply be restarted to restore a clean state.

However, when this happens in a storage component, the resulting state can remain with the system across reboots. The effect of this could be as simple as a corrupted block within a file. In some cases, the application format can deal with this, however in others it renders the file, including the rest of its intact data, useless. It is also possible the corrupted block occurs in some metadata for the filesystem living on top of the storage component, such as a directory. In this case whole subtrees of data may become inaccessible. The results of this can be anywhere from lost data to an inoperable machine.

Given that the storing and processing of data is the primary purpose of most computers, and that the data people store represents an investment in time and energy, these failures and their ensuing loses can be very costly. Thus, we propse a storage system that is not only capable of detecting these silent corruptions, but also correcting them before it's too late. To that end, we've extended the Software RAID layer in Linux's Multi-Device driver, which already includes some redundancy information to be able to recover from some stop-failures scenarios, to include an integrity checking component in the form of checksums. Using this we show that our solution is capable of detecting and correcting single block corruptions within a RAID stripe at a reasonable performance overhead.

The remainder of this paper is structured as follows. In Section 2 we'll discuss some background and related work. Section 3 will discuss our implementation details, followed by our evaluation and results in Section 4, and finally our conclusions in Section 5.

2 Background

In this section we review the problem of silent corruptions and discuss a number of other approaches to dealing with it.

2.1 The Problem

The issue of silent data corruption within storage systems and beyond has been well understood and studied. The IRON Filesystems paper [9] presents a good overview of the possible sources of corruption along the storage stack. Studies at CERN in 2007 [5] and statistics quoted by [4] showed that error rates occurred in only 1 in 10^{12} to 10^{14} bits (8 to 12TB). However, other studies [9] note that this rate will only increase as disks become more complex and new technologies such as flash take hold even as market pressures force the cost and quality

of these components downwards. Indeed, these statistics already increase when we take into account the other components in the system such as memory and I/O controllers [**?**]. Given that this problem is well understood, it is natural to ask what existing solutions there are, which we now discuss.

2.2 RAID

One common technique used to deal with failures in storage infrastructures is RAID [6]. RAID typically provides data redundancy to protect against drive failure by replicating whole or calculated parity data to a separate disk in the array. However, this mechanism only protects against whole disk failures since the parity block isn't usually read, or more importantly *compared*, when a data block is read. Indeed, in a mostly read oriented array, a silent data corruption could occur in a seldomly read parity block resulting in the eventual restoration of bad data. The integrity of a RAID system can be verified periodically by a "scrubbing" process. However, this process can take a very long time, so is seldom done. In contrast, our solution incorporates an active check of data and parity block integrity at the time of each read or write operation to the array. This allows us to detect and repair individual active stripes much sooner. For idle data that is not being actively read, it may still be wise to run periodic scrubbing processes in order to detect corruptions before we can't repair them. However, this process could possibly be optimized to only examine the inactive data.

2.3 End to End Detection

Modern drives already provide some ECC capabilites to detect individual sector failures. However, studies [**?**] have shown that these don't detect all errors. Other solutions, such as Data Integrity Extension (DIX) [8] attempt to integrate integrity checks throughout the entire storage stack – from the application through the OS down to the actual device medium. These particular extensions operate by adding an extra 8 bytes to a typical on disk sector to store integrity information that can be passed along and verified by each component along the storage path so that many write errors can be detected almost immediately. Though OS support for these is improving [7], many consumer devices still lack it. Moreover, they are an incomplete solution. By storing integrity data with the data itself the system is not capable of detecting misdirected or phantom writes. Section 3.1.1 explains how by spreading our checksum storage across multiple disks we are able to handle these problems as well. However, since we rely on a read event to detect corruptions we believe that DIX is a complimentary addition to our solution.

2.4 Filesystem Detection

Other, software based solutions such ZFS [10] also aim to provide more complete OS awareness of integrity checking. In ZFS, the device, volume, and filesystems layers have been integrated to provide a well integrated checksumming storage infrastructure. However, accomplishing this at such levels often requires significant upheaval and is not universally applicable to other filesystems. For example, we originally looked at adding checksum support to Linux's ext4 filesystem. [1] Because ext4 is an extent based journalling filesystem [3] and ZFS is copy-on-write, providing complete and consistent checksumming protection to both the file and meta data would have required creating a new journalling mode and significantly altering the on disk layout, allocation policies, page cache system, and more. In contrast our system adds data integrity checks at a layer that can hopefully do something useful when a mismatch is detected (*e.g.:* repair it) as opposed to simply kicking the error back to the caller. Moreover, it does it in such a way that no changes to the rest of the system are *required*, though for performance and OS and application awareness some may still be desirable.

3 Implementation

As stated previously, the goal of checksumming RAID is to detect silent data corruption and recover from it using the parity data present in typical RAID whenever possible. To achieve this, we have modified the Software RAID layer of Linux's Multi-Device (MD) driver to add two new RAID levels: RAID4C and RAID5C (C for checksumming). This new checksuming RAID system operates by computing a checksum whenever a block is written to the device and storing that checksum in its corresponding checksum block. When a block is read, its integrity is checked by calculating a checksum for the data received and then comparing it to the stored checksum. If the checksums do not match, we start a recovery process to restore the corrupted data by invoking the typical RAID4/RAID5 failed stripe recovery process. That is, we read in the remaining data and parity blocks, verifying their checksums as well, and XOR them to recover the corrupted data block.

[1] ext4 has support for checksumming journal transactions, but that protection does not extend to the final on disk locations.

Figure 1: Our Checksumming RAID Layout. We have added another block to a typical RAID stripe to hold checksum data. The checksum block has a header which includes the block number, followed by a number of checksum entries indexed by the disk number, including one for itself.

3.1 Checksumming RAID Layout

MD's RAID implementation organizes stripes and their respective operations in `PAGE_SIZE` (4K in our case) units from each disk. Among various choices of where and how to store checksums, we chose to store checksums for each stripe in a dedicated checksum block. So, in addition to the typical RAID layout (several data blocks plus a parity block), we added a checksum block for each stripe.

A typical checksumming RAID stripe layout is shown in Figure 1. Within the checksum block, all the checksums for the data blocks and parity block are stored contiguously, indexed by their corresponding disk's position in the stripe. We also store some header information, such as the block number and the checksum for checksum block, to help detect corruption within the checksum block itself.

3.1.1 Layout Analysis

With this layout, bit rot and phantom writes to data blocks are caught and repaired through checksum verifications during read. Misdirected writes can be caught and repaired through a combination of the checksum block number and disk index. Phantom writes of the checksum block are caught indirectly. Such a situation will manifest itself in the form of a checksum mismatch of a data block, upon which a recovery will be initiated. However, the recovery will find at least two checksum mismatches (one for the data and the parity, though potentially of the blocks don't match their checksums) and can at this point choose to either rebuild the checksum block from the available data, or return an error to the caller. In effect, this treatment is no different from parity recovery processes for typical RAID systems.

3.2 Computing the Checksum

To compute checksums, we use the kernel's built-in CRC32 libraries. It's very fast to compute, and recent processors even have dedicated instructions for it. It also provides very reliable protection over our 4K block size [1]. Since MD limits arrays to a maximum of 27 members, we have extra space reserved for each checksum entry. With this we could potentially use more complex checksum, such as SHA-512 at the expense of extra computational resources since we have plenty of space in our checksum block, but we felt that CRC32 was good enough.

3.3 Typical Operation Processes

To better illustrate how this checksumming mechanism works, we trace the typical path of reading, writing, and recovery operations in some detail in the following sections.

3.3.1 Typical Write Process

As discussed in Section 3.4, write operations in MD are buffered in memory until they must be force to disk. Once that happens, when writing to a data block, as in typical RAID, we must calculate the new parity for that stripe. However, in checksumming RAID we also calculate its checksum, the checksum of the new parity, and the checksum of the new checksum block which will hold all the preceding checksums. This may involve some reads to get the checksum block and other block(s) for the parity calculation. [2] Once we have the appropriate data and have made the necessary calculations we

[2] In the case of minimal RAID array members, MD avoids read-write-modify operations (RWM), and instead reconstruction writes are performed by reading the other data from the remaining data disk block in the stripe.

issue write requests through the generic block layer for the affected data, parity, and checksum blocks back to their corresponding array members.

3.3.2 Typical Read Process

When issuing a read to a data block, we want to verify its integrity, so we hook the I/O completion functions to make sure we've fetched the corresponding checksum block as well. Upon completion of checksum block read, we compare the block number, calculate its checksum and compare with its stored value. Similarly, upon completion of the data block read we calculate its checks compare it with the checksum stored in the valid checksum block. For each, if the two checksum agree, we finish read successfully; if they don't agree, we start a recovery process.

3.3.3 Data Block Corruption Recovery

As in typical RAID, when recovering a block from parity we need to read in the remaining blocks in the stripe. However, in checksumming RAID we also verify the integrity of each of these blocks by computing and comparing their checksums. Once we have successfully restored the corrupted block content, we rewrite it to disk and then retry the failed read operation. If the reread fails, we assume there has been a more pervasive storage system failure and return an error to the caller.

3.3.4 Checksum Block Corruption Recovery

The checksum block itself can also be corrupted in the same ways that any other block can. To detect this, we store a checksum over the checksum block itself, which is calculated by zeroing out the corresponding checksum entry and block number. Unfortunately, since the parity block only provides redundancy information for the data blocks in a stripe, we have no recourse other than to rebuild the checksum block from the rest of the data in the stripe. To do this, we read in the remaining blocks in the stripe, recalculate their checksums, recalculate the checksum of the checksum block, write it back to disk and reread it. As in the data block corruption recovery case, if the reread fails, we return an error to the caller.

3.4 Cache Policy

MD uses a fixed size pool of stripe buffers to act as a cache for future reads. We made use of this to alleviate the cost of doing extra I/O to the checksum block when multiple blocks from the same stripe are being accessed. Similarly, the stripe pool is used to buffer partial writes for a while [3] in the hope that future write requests result in the ability to perform full stripe writes, thus saving the cost of extra parity and checksum block reads.

A possible improvement on this simple mechanism would be to add some way of preferentially keeping checksum blocks cached for a longer period in order to try and avoid the extra I/O cost to read them later on. However, this would have added significant complexity to our initial implementation so we postpone it for future work.

3.5 Changes to Linux's Software RAID Driver

Most of the functionalities we are seeking are already provided in the Linux's Software RAID driver code. We changed the driver and its corresponding userspace utilities so that it could recognize RAID4C and RAID5C levels, and issue read/write to checksum blocks when appropriate while operating on these levels. To do this we hooked into the generic block layer's I/O completion functions to issue the appropriate actions (*e.g.*: calculate, store, verify, recover). In general, it is a modest modification to the original driver; about 2000 lines of code are added in total.

It's worth noting that our implementation is a relative naive one and there is plenty of space for optimization. For example, the checksum calculation could be done asynchronously, like most XOR and memory copy operations are already done in the current RAID code. Also, for now we have an unnecessary read of checksum block when doing full stripe write, which should be removed to improve performance.

3.6 Crash Recovery

One issue we have not discussed is what to do during crash recovery. If the system crashes during a write, upon reboot we may have a partial write somewhere within the array. In standard RAID5 this could mean a stripe whose parity block doesn't match its data blocks. In our checksumming RAID5C it can also mean a checksum block whose contents don't represent the data blocks it protects. If we were to encounter such a situation in steady state we would potentially see this as a data block checksum mismatch and issue a recovery operation from non matching parity blocks, thus potentially leading to semantically corrupted data. Thus our only recourse is to rebuild the checksum block contents for those partial write stripes at recovery time.

[3] The exact amount of time depends upon memory pressure, timers, etc.

However, without more information we don't know what stripes need to be repaired and a full RAID resync operation can take days. Since standard RAID5 has to deal with this situation as well MD supports a write intent log which can be used to keep track of which portions of the array were recently being modified. At restore time, the log can used to inform the resync layer which stripes should be examined. The intent log however can add a significant overhead.

Previous work [2] has been done to make use of the journalled filesystem that typically sits on top of the RAID device to inform it which stripe regions should be examined. By reusing the journals write ahead logging, the overhead of typical operations is kept to a modest and reasonable amount.

Today logical volume managers are also commonly placed on top of RAID devices as well. Thus, we propose extending the Journal Guided techniques so that they can be passed through the necessary layers to the underlying storage device to let them deal with if and how to recover, however our current implementation does not address this.

4 Evaluation

In this section we present an evaluation of our implementation. In particular, in Section 4.3 we present results on its correctness by verifying that it in fact catches and corrects errors we introduce. Section 4.1 presents an analysis of its overheads which we use to help develop a model to explain results in Sections 4.5 and 4.6 which discuss the effects of disk layout within the array.

4.1 Overheads

Since our layout stores checksums in a separate block, our implementation must issue and wait on extra I/O operations and perform several checksum calculation operations. For example, a cold cache random read operation needs to perform one extra read operation to the checksum block, when compared to typical RAID5. However, for this same random read it requires, two checksum calculations: one for the data block, and one for the checksum block itself. On the other hand a random read-modify-write requires two extra I/Os, a read and write to the checksum block, and six checksum calculations: verifying the checksum block, verifying the old parity, verifying the old data, computing the new parity, computing the new data, and computing the new checksum block checksums. For each operation (random/sequential read/write), we summarized the per-stripe checksum calculations and extra I/O operations incurred by our checksumming implementation in Table 1 and Table 2 respectively. We expect that the extra I/O operations will dominate the cost of computing checksums in most operations.

[4] In our implementation there was a bug which caused an extra read and checksum calculation call, beyond those presented in the tables, to the checksum block even though full stripe writes can just overwrite it.

4.2 Test Setup

We begin by first explaining the details of our test setup. Due to a lack of physical resources, a number of tests were developed and conducted in a virtual environment. Our VM was given 2GB RAM, 2 CPUs, and installed with Debian and a 2.6.32.25 kernel To act as our MD array members the VM was alloted 10 8GB virtual virtual disks backed by a relatively powerful, but mostly idle, 14 15K RPM disk RAID50 SAN. Since there are a handful of layers in between the VM and the SAN (*e.g.*: iSCSI, caching, buffer, etc.), the numbers we present should not be treated as absolute timings to be comparable with real arrays. For this reason we also focus our attention on sequential access patterns. However, since the changes we made largely effect the kernel interactions, and we're comparing apples to apples, so to speak, we feel that the relative comparisons made in each test are still valid. For completeness, we also conducted a test on a physical machine with an 3 GHz E8400 Core2Duo, 2GB RAM, and a single extra 80GB disk.

4.3 Correctness

In order to verify the correctness of our implementation, we assembled a minimal four disk array for both RAID4C and RAID5C and used the `dd` command to write random data to the first 750 data block in a disk in our array. For RAID4C this setup corrupts 750 data blocks. For RAID5C this setup corrupts 494 data blocks, 128 parity blocks and 128 checksum blocks. We then read the first part of the array to induce corruption detection and correction. Finally we examine the `dmesg` output to make sure that each block that had a `mismatch detected` message had a corresponding `mismatch corrected` message and counted the results.

For RAID4C we successfully detected and recovered all the 750 corruptions. For RAID5C we detected and recovered the 494 corrupted data blocks and 128 corrupted checksum blocks. The remaining 128 parity block corruptions are not detected because the data blocks they protect were not corrupted, so they are not read in normal operations. In both cases we also verify that the data

Checksum Calculations	Full Stripe (Sequential)	Single Block (Random)
Read	$N-1$	2
Write	N [4]	—
Read-Modify-Write	—	6
Reconstruction Write	—	5
Data Block Recovery	—	$N-1$
Checksum Block Recovery	—	N

Table 1: A table depicting the number of checksum calculations required to perform various operations. N is meant to represent the number of disks in the array.

Extra IOPS	Full Stripe (Sequential)	Single Block (Random)
Read	1	1
Write	1 [4]	–
Read-Modify-Write	—	2
Reconstruction Write	—	1 [4]
Data Block Recovery	—	1

Table 2: A table depicting the number of extra IO operations required for checksumming RAID to perform various operations as compared to standard RAID. N is meant to represent the number of disks in the array.

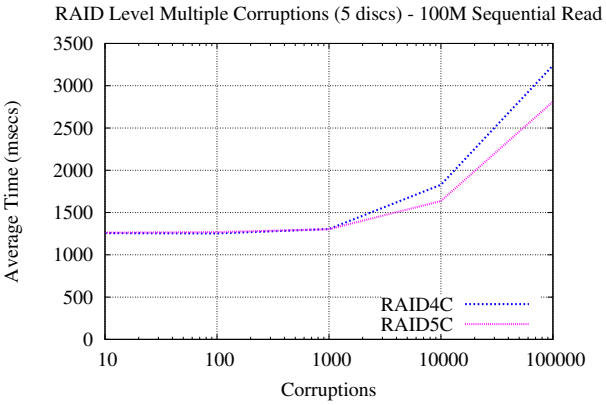

Figure 2: We varied the number of corruptions in the array to show its impact on sequential read performance due to the extra recovery processes.

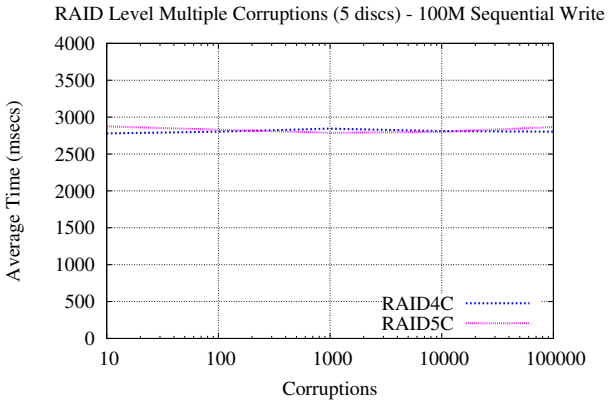

Figure 3: The number of corruptions in an array does not effect full stripe sequential writes since the checksum block never has to be consulted so no corruptions are detected and no recovery processes are necessary.

we were returned matches our expectations, which indicates that our implementation is correct.

4.4 Effects of Increasing Corruptions

As the number of corruptions in an array increases we expect that the performance of the array decreases as it spends more time performing recovery operations. To confirm this we injected various numbers of corruptions to random locations of one disk and observe the time taken to perform 100M sequential read and writes to random locations within the array. The results are shown in Figure 2 and Figure 3. As expected we can see that sequential read performance degrades exponentially as the number of corruptions exponentially increases since we are performing increasing numbers of recovery operations. Sequential write performance however is unaffected by the frequency of corruptions it encounters. This makes sense since for full stripe writes, we can simply overwrite the checksum block with the new checksum information rather than perform any read operations which would have induced a recovery.

4.5 Effects of Disk Counts

As noted in Table 2 our implementation incurs extra I/O operations for some stripe operations. Therefore, we

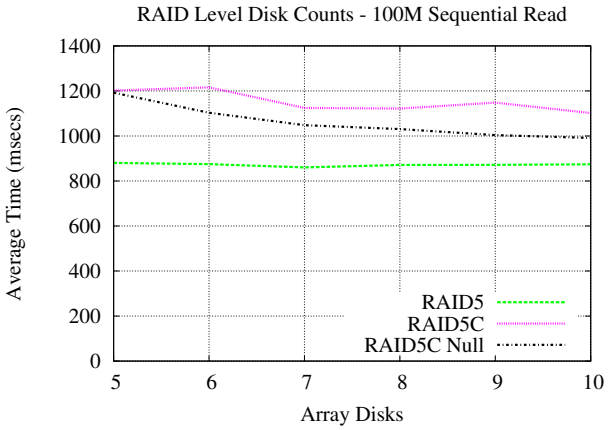

Figure 4: We varied the number of virtual disks in our test array to show that the cost of extra IO and checksum calculations in amortized with larger stripes.

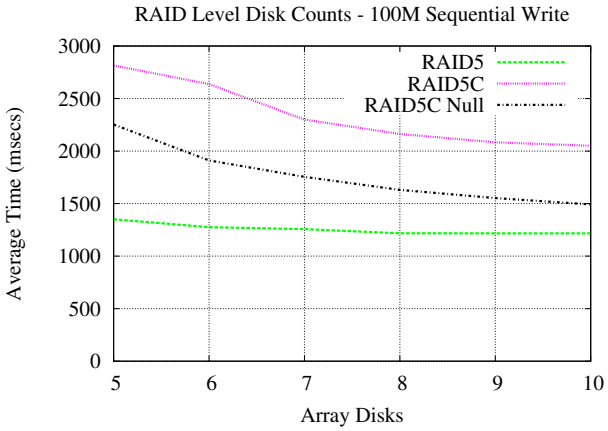

Figure 5: We varied the number of virtual disks in our test array to show that the cost of extra IO and checksum calculations in amortized with larger stripes.

wanted to evaluate how increasing the stripe size in the array affects the performance of checksumming RAID. We expected that the cost of these extra operations for sequential access patterns is amortized over larger stripe sizes due to more checksum operations being serviced from cache rather than disk. To do this we assembled arrays of increasing disk numbers. For each array size we evaluated sequential read and write performance of standard RAID levels 4 and 5, compared with our checksumming RAID levels 4C and 5C by reading and writing 100M of data 100 times. To distinguish the overhead incurred by checksum calculations from the extra I/O operations to the checksum blocks, we also conducted a series of experiments using a null checksum operation which simply returned a constant value when calculating, storing, and comparing checksums. For fair comparisons the results, shown in Figure 4 and Figure 5, of standard RAID levels are shifted over one so that we are comparing a similar number of data disks in the arrays. For legibility we have also excluded the RAID4 and RAID4C comparisons from these graphs.

In general we can note that our basic expectations that increased stripe size improves performance holds true. For sequential reads we see a decrease of 1300 ms to 1100 ms and writes decrease from 3000 ms to 2000 ms. We could also see that the overhead of our checksum mechanism is within 50However further analysis of these graphs holds some confusing results.

For instance, the performance of RAID5 does not decrease much as the number of disks in the array increases, though it should since there should be more bandwidth available from the array in total. This seems to indicate that we are hitting some caching or read ahead level that is causing the results to bottleneck on something else in the system besides IO. However, our checksumming RAID levels should be hitting the same plateau, but we see more variation in their results. We therefore conclude that the difference we see is the result of our extra read operations.

We also note that while the overall overhead of checksumming RAID in both cases appears to be split between 50% waiting for the extra read operation to compute and 50% for the actual checksum calculation, the computation portion of this overhead appears to remain relatively constant regardless of the number of disks in the array. This doesn't seem to make sense since for a 5 disk array we have 32000 checksum operations to perform while a 10 disk array has some 42000. This would also seem to indicate that our implementation is bottlenecked by something else such as the number of memory operations it performs.

Finally, the overall timings and overhead difference between reads and writes is substantial – over 100% in the 5 disk case. This would seem to indicate significant caching happening within the SAN.

4.6 Single Disk Tests

Since data integrity is also of value to home users who cannot always afford a complex multi disk RAID system, we also evaluated a checksumming RAID setup assembled on a single physical disk split into 4 equal partitions. Using the predictions in Table 2 we can see from Figure 6 closely follows this behavior. The standard RAID5 setup has 3 data disks, thus a write results in a read-modify-write operation: 4 I/Os to read in the old data, old parity, and write out the new data and parity. Accordingly, we see that the time taken to issue a random write to the RAID5 is about 4 times that of the raw device. The RAID5C on the other hand only has two data disks, so

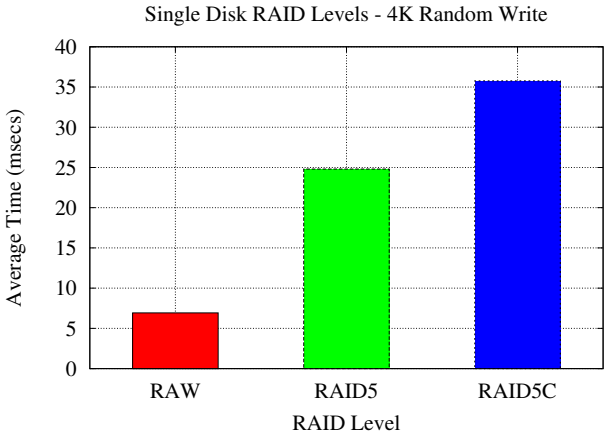

Figure 6: We constructed various RAID levels over a single disk split into 4 equal sized partitions to show how our results match the overhead models predicted in Tables 1 and 2.

MD performs a reconstruction write instead: 5 I/Os to read the old checksums, old data, and write the new data, new parity, and new checksums. The results for this test show that is about 5 times that of the raw device – 1 more than the RAID5 layer, just as Table 2 predicted.

Though these results better match our predicted overheads, clearly this naive layout has some severe drawbacks on a single disk. It causes the spindle to make long seeks across the whole disk to read and write data in the same stripe. A better approach to do this on a single disk would be define a new RAIDC level that arranges stripes contiguously on disk. We postpone further discussion of this technique for future work.

5 Conclusions

We have added checksumming RAID levels to Linux's Software RAID driver to provide the user the capability to detect and repair silent data corruptions. RAID4/5C works by checksumming the data written to disk and verifying the checksum when reading data from disk.

Our experiments has shown that the checksums help to validate data, find silent data corruptions and repair them, though with some performance cost. The performance overhead of adding checksum mechanism is within 50%-100% compared to the stock RAID level in our naive implementation, which we feel is acceptable under lots of scenarios. Even so, we have identified a number of areas for improvement in the previous sections of our paper including asynchronous checksum calculations, contiguous single disk layouts, and journal guided restoration.

Acknowledgments

Our thanks to Professor Remzi Arpaci-Dusseau for his input and guidance in our efforts and to the many others that put up with us during the process.

References

[1] J. Becker. Block-based error detection and correction for ocfs2. `http://oss.oracle.com/osswiki/OCFS2/DesignDocs/BlockErrorDetection`.

[2] T. Denehy, A. Arpaci-Dusseau, and R. Arpaci-Dusseau. Journal-guided resynchronization for software RAID. In *Proceedings of the 4th conference on USENIX Conference on File and Storage Technologies-Volume 4*, page 7. USENIX Association, 2005.

[3] A. Mathur, M. Cao, S. Bhattacharya, A. Dilger, A. Tomas, and L. Vivier. The new ext4 filesystem: current status and future plans. In *Ottawa Linux Symposium*, 2007.

[4] B. Moore. Zfs - The Last Word In File systems, 2007.

[5] B. Panzer-Steindel. Data Integrity, CERN IT Group, 2007.

[6] D. Patterson, G. Gibson, and R. Katz. A case for redundant arrays of inexpensive disks (RAID). In *Proceedings of the 1988 ACM SIGMOD international conference on Management of data*, pages 109–116. ACM, 1988.

[7] M. K. Peterson. Block/scsi data integrity support. `http://lwn.net/Articles/280023`.

[8] M. K. Peterson. I/O Controller Data Integrity Extensions, 2009.

[9] V. Prabhakaran, L. Bairavasundaram, N. Agrawal, H. Gunawi, A. Arpaci-Dusseau, and R. Arpaci-Dusseau. IRON file systems. In *Proceedings of the twentieth ACM symposium on Operating systems principles*, pages 206–220. ACM, 2005.

[10] Yupu Zhang, Abhishek Rajimwale, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau. End-to-end Data Integrity for File Systems: A ZFS Case Study. In *To Appear in the Proceedings of the 8th Conference on File and Storage Technologies (FAST '10)*, San Jose, California, February 2010.