Exploiting Type-Awareness in a Self-Recovering Disk*

Kiron Vijayasankar, Gopalan Sivathanu, Swaminathan Sundararaman, and Erez Zadok
Stony Brook University Computer Science Department
Stony Brook, NY 11794-4400
{kvijayas,gopalan,swam,ezk}@cs.sunysb.edu
Appears in the proceedings of the Third ACM Workshop on Storage Security and Survivability (StorageSS 2007)

ABSTRACT

Data recoverability in the face of partial disk errors is an important prerequisite in modern storage. We have designed and implemented a prototype disk system that automatically ensures the integrity of stored data, and transparently recovers vital data in the event of integrity violations. We show that by using pointer knowledge, effective integrity assurance can be performed inside a block-based disk with negligible performance overheads. We also show how semantics-aware replication of blocks can help improve the recoverability of data in the event of partial disk errors with small space overheads. Our evaluation results show that for normal user workloads, our disk system has a performance overhead of only 1–5% compared to traditional disks.

Categories and Subject Descriptors

D.4.2 [**Operating Systems**]: Storage Management—*Secondary Storage*

General Terms

Performance, Reliability

Keywords

Type-awareness

1. INTRODUCTION

Modern commodity disks do not follow the *fail stop* failure model where the disk stops operation when there is a hardware error [17]. Partial failures in disks today can be attributed to latent sector faults [2] or even silent block corruption [1], which can be hard to detect. While expensive high-end disk systems (e.g., RAID [13]) implement recovery methods to deal with partial faults, cheaper desktop hard drives (e.g., SATA disks) are much less reliable in handling faults. Partial faults that are not detected on time can lead to serious software malfunction and data loss. For example, a bad bitmap block in a file system can result in valid blocks being overwritten.

Although current disks have mechanisms such as error-correcting codes and dynamic remapping of bad blocks to protect against block corruption and failed writes, they do not handle several fault scenarios. Applications and file systems employ integrity checks and recovery mechanisms to handle disk errors [4, 6, 9, 14, 15, 22]. Disk-level integrity assurance and data recovery is useful for two key reasons: (1) mechanisms can be completely transparent to higher-level storage software such as file systems or databases, thus reducing their complexity; (2) disks can make use of their internal hardware knowledge and load characteristics to perform efficient I/O.

Today's disks treat data as completely opaque entities and they lack information about higher-level data semantics. This makes it hard to implement efficient integrity checking techniques at the disk-level. For example, if block checksumming is used to verify integrity, information about access patterns can enable intelligent prefetching of checksum blocks. Similarly, efficient recovery mechanisms can be implemented if the relative importance of blocks are known at the disk-level.

Knowledge about higher-level pointers at the disk level (e.g., an inode block pointing to several data blocks) can be used to infer three key details useful for integrity checking and recovery:

1. The paths used to access blocks, i.e., a sequence of blocks that need to be accessed before a block is accessed (e.g., an inode block has to be read before reading a data block pointed to by it).

2. The relative importance of blocks; pointers help in communicating the reachability of blocks. Blocks that have outgoing pointers are more important as they impact reachability of other blocks.

3. Block-liveness information; all allocated blocks must be reachable from at least one block.

We therefore leverage Type-Safe Disks (TSDs) [18] to build a self-recovering disk system that uses higher level data semantics to perform efficient integrity checking and recovery. A TSD is a disk system that uses pointer information (i.e., type information) to enforce active constraints on data access. For example, a TSD can prevent applications from accessing unallocated blocks (an unallocated block is one that is not pointed to by any other block). The disk interface has been modified to allow file systems to communicate pointers to the disk. The file systems uses the disk APIs such as `alloc_block`, `create_pointer`, and `delete_pointer` to

*This work was partially made possible by NSF CAREER EIA-0133589 and CCR-0310493 awards and HP/Intel gifts numbers 87128 and 88415.1.

notify the disk about the relationships among blocks that are stored in it.

We have extended Type-Safe Disks to be more robust to disk errors. We call our modified disk system *Self-Recovering Disks* (SRDs). SRDs perform integrity checks by storing the checksums of all blocks; during a block read, the SRD computes and compares the block's checksum with the one stored on disk. To reduce the overhead of storing and comparing checksums, SRDs store the checksum blocks close to the original block's parent block.

In order to provide recovery for reference blocks, SRDs perform two-way replication of all reference blocks. Upon detecting an integrity violation, the replica is transparently used in the place of the corrupt reference block. It has been shown that some form of block failures (e.g., a scratched surface) exhibit spacial locality, thereby making a group of blocks inaccessible [8]. Hence, SRD places the reference block and its replica bit far away from each other. This decreases the probability that both the block and its replica will be affected by latent sector faults. In addition to the integrity checks and recovery mechanisms, SRDs are *self-correcting*: when an SRD detects block corruption, it overwrites the corrupt block with the data from its replica.

We benchmarked SRDs against regular disks for two different workloads: Postmark, which represents a busy mail server, and a kernel compile, which represents a developer's machine. While providing integrity checks for all blocks and recovery of reference blocks, SRDs had a small overhead of 4.9% for Postmark and 0.7% for kernel compile benchmark.

SRDs provide better reliability at the disk by checking the integrity of all blocks during read operations and replicating reference blocks (blocks that impact reachability of other blocks) bit far away from each other. SRDs intelligently store the checksums of blocks near their parent blocks and pre-fetch them when their parent blocks are read. Finally, SRDs try to improve the disk performance by redirecting read requests to the nearest copy during reference block reads.

The rest of the paper is organized as follows. In Section 2, we describe integrity assurance mechanisms, data recovery techniques and Type-Safe Disks as a means of disk-level error detection and recovery. Section 3 discusses the design of SRDs. Section 4 describes our prototype implementation of SRDs. We evaluate all our prototype implementation in Section 5. We discuss related work in Section 6, and conclude in Section 7.

2. BACKGROUND

In this section, we describe common integrity assurance mechanisms, and data recoverability mechanisms. We also describe why Type-Safe Disks (TSDs) [18] are a good design choice for disk-level error detection and recovery.

Integrity Assurance.

Sivathanu et al. [19] broadly classify integrity assurance mechanisms as physical redundancy techniques and logical redundancy techniques. Physical redundancy techniques explicitly store redundant information for integrity checking. Logical redundancy techniques exploit structural redundancies that exist in the data for integrity checking. Checksumming and parity are two physical redundancy techniques used commonly. SRDs employ block-level checksumming as a means of integrity assurance.

Data Recoverability.

Existing techniques like RAID [13] use redundant information to the recover original information when a disk fails. Although different levels of redundancy can be used for different performance and reliability requirements, a fundamental drawback of existing techniques is the inability to selectively replicate data at the disk-level based on the data's importance. This limitation arises out of the information gap between the storage systems and the higher layers [3, 5].

TSD for Disk-level Data Recoverability.

Type-Safe Disks (TSDs) [18] aim to bridge the information gap between the storage systems and the higher layers [3, 5] through pointers. Pointers serve as a simple yet powerful mechanism to bridge this information gap. TSDs can distinguish reference blocks from data blocks through pointers. Reference blocks are those that have at least one incoming and outgoing pointer, whereas data blocks have no outgoing pointers. Example of reference blocks would be inode or indirect blocks that have outgoing pointers to data or indirect blocks, in the case of an FFS-like file system [12]. A block has to be allocated first using the TSD API before it can be accessed; this is because free-space management is moved to the disk from file systems, and blocks that are not pointed by any other block are automatically garbage collected. TSDs have type information (i.e., the ability to differentiate between reference and data blocks) and block liveness information inside them. This makes TSDs a good design choice for disk-level data recoverability techniques. SRDs try to leverage TSDs' type and liveness information to provide efficient disk-level error detection and recovery while keeping the same TSD interface.

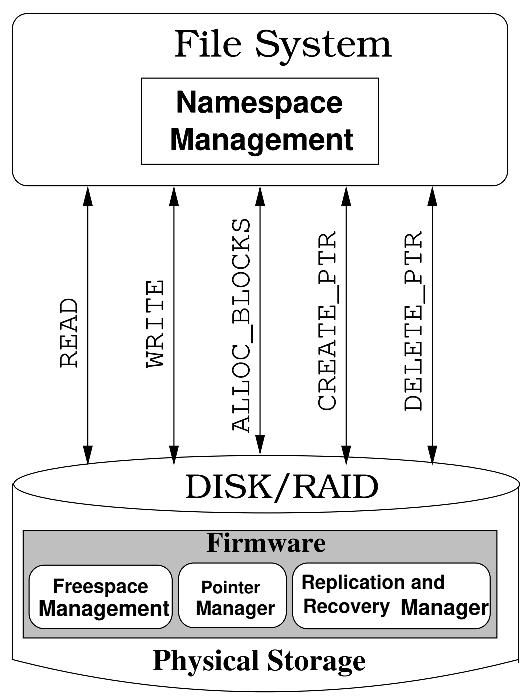

Figure 1: Self-Recovering Disk

3. DESIGN

SRDs aim to provide error detection and recovery at disk-level for a single disk. SRDs are designed to leverage the available pointer information to selectively replicate important blocks. This enables recovery of key data while consuming marginally more space.

3.1 Detecting Block Errors

Checksums using collision-resistant hash functions has been a popular way of ensuring integrity. To provide data integrity, SRDs compute collision-resistant checksum of all the blocks except those that store the checksums. Our current implementation of SRDs use the MD5 [16] algorithm to compute the checksum of blocks. However, the design does not restrict the hash algorithm used and MD5 can be replaced by any other hash algorithm. Checksums are updated when the blocks are written to the disk. SRDs compute the checksum of data read from the block and compares it with the stored checksum of block to check for block corruption. Figure 1 shows the interface and components of SRDs.

3.1.1 Data Structures

SRDs use TSD on-disk structures [18] and additional data structures to keep track of the redundant information needed for error detection and recovery. SRDs also maintain in-memory structures that act as caches to improve performance.

On-Disk Structures.

PCTABLE. SRDs maintain a reference-block tracking table called PCTABLE that is indexed by the reference block number. Each table entry contains the reference block number, the list of block numbers that store checksums for the blocks pointed to by the reference block, and the bitmaps associated with each of these checksum blocks. A new PCTABLE entry is added when the first outgoing pointer is created from a block.

PTABLE. SRDs also maintain a pointer tracking table called PTABLE that stores the set of all pointers. The PTABLE is indexed by the destination block of the pointer. Each PTABLE entry contains a reference to the PCTABLE entry corresponding to the pointer's source block, and the offset of the destination block's checksum in the list of checksum blocks associated with the source block. A new PTABLE entry is added when a new pointer is created.

In-Memory Structures.

LRU-CTABLE. This is an in-memory table that caches block checksums for fast updates and verification. It is a hash table indexed by block number and each node contains block number, checksum and the on-disk address of the checksum that is cached. All checksum fetches and updates first go to LRU-CTABLE. An LRU-CTABLE cache miss results in the corresponding checksum block being read and LRU-CTABLE entries being created. We use an LRU algorithm to purge entries from LRU-CTABLE and limit its size within the maximum limit.

LRU CHECKSUM BLOCK CACHE. It is a list of checksum blocks cached by SRD in order to speed up writing checksums back to the disk. When a checksum block is read, it is added to this list. The size of this list is also limited to a fixed maximum value. Old checksum blocks are written back to the disk (or discarded if they are not dirty) to make space for newly read checksum blocks in the list. We use an LRU algorithm to reclaim entries from this list.

3.1.2 SRD Operations for Block Reads and Writes

When a meta-data block is read, the checksum blocks of the data that it points to are also read. The data read from these prefetched checksum blocks are populated in LRU-CTABLE to reduce the time required to read and verify the checksums of the data blocks that this meta-data block points to. Sometimes the checksum entry may be reclaimed from LRU-CTABLE by the time the data block is read. In such cases, the checksum block is read again and used to populate the entries in LRU-CTABLE. The read request for the block has to wait during this period of time. From our benchmark results we show that this situation seldom occurs. In the majority of the time, the data-blocks are read immediately after their parent block. The operations that are performed during block writes are quite similar to the operations performed during block reads. During block writes, checksums are updated in LRU-CTABLE and marked dirty. If the entry is not present in LRU-CTABLE, they are repopulated by reading the checksum blocks as in the case of block reads. When dirty entries are reclaimed from LRU-CTABLE, the corresponding checksum blocks are updated on disk. Updating checksum blocks is optimized by keeping a cache of recently read checksum blocks and updating all dirty checksums from LRU-CTABLE that belong to a particular checksum block before it is being written.

3.2 Selective Block Replication

Previous works such as D-GRAID [20] does selective meta-data replication. D-GRAID understands file system data structures and hence D-GRAID can replicate naming and system meta-data structures of the file system to a high degree while using standard redundancy techniques for data. However, D-GRAID targets high-end RAID systems whereas SRDs try to solve the same problem for a single disk. SRDs understand the importance of blocks based on pointer information. Hence SRDs are able to replicate important blocks selectively while not replicating less important blocks, thereby saving space without compromising much on error-recovery capabilities. The reachability of data blocks is determined by their reference blocks [18]. This means that reference blocks are more important than data blocks. Therefore, SRDs replicate reference blocks and not data blocks.

Since SRDs replicate only reference blocks, we use PCTABLE to hold the block number of the replica block along with each reference block number. Since higher level software is not aware of reference block replication, all I/O requests are identified by the block number of the primary copy. As the PCTABLE is indexed by the block number of the primary copy, it is easy to retrieve the replica block number in case of an I/O error. All I/O operations on replica blocks keep track of the block number of the primary copy since it is needed to update PCTABLE and to pass the result to the higher-level software layers.

Recovery mechanism during failed block writes are already present in disks [15]. SRD focuses on unreadable blocks or block corruption. During a block read, the stored checksum of the block may not match the one computed from its contents. If the block is a reference block SRD tries to locates its replica. The replica could be scheduled to be written to disk, waiting in the disk queue, or needs to be read back from the disk and is returned back to the user once its integrity is verified.

The prototype implementation of SRD does not handle crash consistency. If power is lost abruptly, the disk may go to an inconsistent state: copies of the meta-data block that are scheduled to be written could be lost. We can avoid this situation by using write ahead logging for the meta-data updates. This would ensure that during crash recovery, the disk replays the operations reading from the logs.

3.3 Data Recovery

We check the replica block when the checksum does not match with the original block. If the checksum matches with the replica block, the data is recovered. If the checksum of the replica does not match with the stored checksum, then we assume that the stored checksum block is corrupted. SRDs compute checksums for the original block and its replica and compare them to see if they are the same. If the checksums match, SRDs replace the corrupt checksum entry. Hence SRDs are able to recover data when at least two of the three redundant data items are available (primary copy, replica, and checksum).

3.4 Limitations

Error detection and recovery at disk-level alone cannot handle certain types of errors, like bus errors or firmware bugs. Performing error detection and recovery at the file-system level could be a better option in these cases.

4. IMPLEMENTATION

We implemented a prototype SRD based on our TSD implementation as a pseudo-device driver in Linux kernel 2.6.15 that stacks on top of an existing disk block driver. Our implementations added approximately 1,500 lines of kernel code to the prototype TSD block driver implementation while changing about 50 lines of existing TSD code. No change was needed at file system level or user level. SRDs work with file systems modified for TSDs without any further changes. This shows that SRDs can be used with any software that runs on TSDs.

The SRD layer accepts all but only those primitives accepted by TSDs. The SRD layer intercepts all read/write requests, performs the operations needed for error detection/recovery and redirects the requests to the lower-level device driver. The SRD layer also intercepts replies from the lower level device driver and performs the necessary SRD operations on the I/O. The operations performed at SRD layer include checksum update and verification, as well as replication.

We implemented the PTABLE and the PCTABLE as in-memory hash tables which get written flushed to disk at regular intervals of time through an asynchronous commit thread. We did not modify other TSD data structures such as the RTABLE and thus they remain unchanged from the TSD implementation we used.

We implemented the SRD in-memory structures (LRU-CTABLE and CHECKSUM BLOCK CACHE) as data structures with fixed maximum size limit. This was to ensure that they can fit into the memory available on a real disk. We used the LRU strategy to reclaim entries from the LRU-CTABLE and the CHECKSUM BLOCK CACHE.

The replication of a block has to be initiated when the first pointer is created from that block. This is when the block changes from a normal block to a reference block. We achieve this by allocating the replica block and explicitly initiating the first replication when a block is newly added to the PCTABLE. To handle subsequent writes to a reference block, each write I/O is intercepted by the SRD layer and we perform a lookup to check to see if the block is present in the PCTABLE. If it is present, then the replica block number is fetched from the PCTABLE and an asynchronous write is issued to the replica block with the same data.

Our current prototype implementation of SRDs does not address data-block recovery. In the future, parity methods can be used to recover data-blocks. SRDs cannot overcome multiple failures of the same meta-data block (i.e., if the block and its replica are corrupted).

5. EVALUATION

We evaluated the performance of our prototype SRD framework in the context of Ext2TSD [18]. Ext2TSD is the Linux Ext2 file system modified to support TSDs. Ext2TSD is similar to Ext2 except for the fact that allocations and de-allocations are done by using the disk API. We ran general-purpose workloads on our prototypes and compared them with an unmodified Ext2 file system on a regular disk. This section is organized as follows: first we describe our test platform and configurations. We then analyze the performance of the SRD framework using the Ext2TSD file system with the Postmark and kernel compile benchmarks.

We conducted all tests on a 2.8GHz Xeon with 1GB RAM, and a 74GB 10Krpm Ultra-320 SCSI disk. We used Fedora Core 4, running a vanilla Linux 2.6.15 kernel. To ensure a cold cache, we unmounted all involved file systems between each test. We ran all tests at least five times and computed the 95% confidence intervals for the mean elapsed, system, user, and wait times using the Student-t distribution. In each case, the half-widths of the intervals were less than 5% of the mean. We define wait time as the elapsed time less CPU time used and it consists mostly of I/O, but process scheduling can also affect it.

5.1 Space Overheads

We measured the space overhead of the SRD over a traditional disk for a Linux-2.6.15 kernel tree. This is a 253MB dataset with 1,160 directories and 18,798 files. The extra space taken by the SRD was 2.01%, out of which 1.90% was for PTABLE and 0.11% was for PCTABLE. For large files, the overhead will be less since the number of pointer blocks will be less. 1.90% of this space overhead is for TSD meta-data and the SRD meta-data adds only 0.11% since PCTABLE is the only persistent meta-data maintained by the SRD other than the TSD meta-data.

5.2 Postmark

We used Postmark v1.5 to generate an I/O-intensive workload. Postmark stresses the file system by performing a series of operations such as directory lookups, creations, reads, appends, and deletions on small files [17]. For all runs, we ran Postmark with 50,000 files and 500,000 transactions.

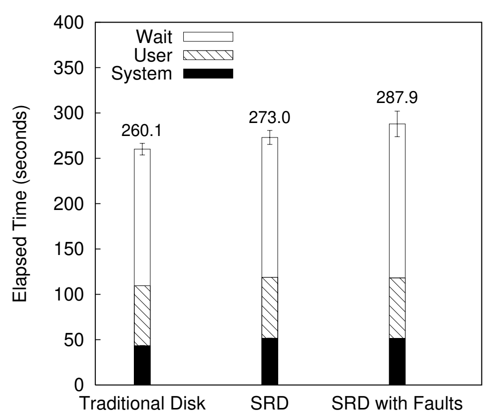

Figure 2: Postmark results for SRD

Figure 2 shows the comparison of Ext2TSD over SRD with regular Ext2 over a traditional disk. SRD has a system time overhead of about 19% compared to a traditional disk. The increase in system time is due to checksum computation and hash table lookups required for checksum updates and verification. The wait time of SRD was only about 2% higher than that of a traditional disk. SRD impose additional overhead when reading and writing checksum blocks. However, this overhead is offset to some extent by the better spatial locality in SRD for the Postmark workload. Ext2's allocation policy takes into account future file growth and hence leaves free blocks between newly created files. Ext2TSD does not implement this policy and hence we have better locality for small files. Overall, the elapsed time for SRD is 5% more than that for the regular disk.

We also tested the performance of SRD with artificial fault injection. For this, we injected 20% faults during reads. However, since the prototype implementation of SRD cannot recover corrupted data blocks, we ignored data block corruption and passed the read block to the higher layers. The implementation cannot recover if both copies of a pointer block are corrupted. So we injected faults periodically, once for every five reads. Faults injected in pointer block reads were caught by SRD and the replica blocks were read. Periodic injection of faults ensured that both the original and replica reads were not fault injected. The total overhead was about 5% over SRD without fault injection. The system and wait times were comparable to normal SRD, but wait time was about 9% higher due to the extra reads needed for replica blocks.

5.3 Kernel Compile

To simulate a relatively CPU-intensive user workload, we compiled the Linux kernel source code. We used a vanilla Linux 2.6.15 kernel, and analyzed the overheads of Ext2TSD, for the `untar`, `make oldconfig`, and `make` operations combined.

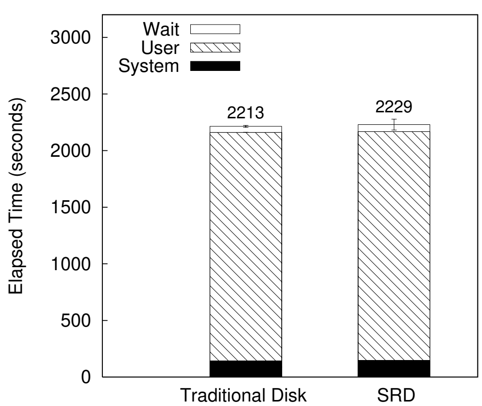

Figure 3: Kernel compile results for SRD

Figure 3 shows the comparison of Ext2TSD over SRD with regular Ext2 over a traditional disk. The system time overhead of SRD over a traditional disk for kernel compile workload is about 4%. The elapsed-time overhead of Ext2TSD over SRD compared to Ext2 over a traditional disk under this benchmark is less than 1%. The wait time overhead is about 16%. This increase in wait time is not only due to the increase in I/O. The increase occurs also because the SRD checksum cache update thread preempts the CPU to update the checksum cache and write dirty checksums to the disk. This thread takes more system time, which manifests as wait time in the context of the kernel compile benchmark.

6. RELATED WORK

SRDs combine integrity check, physical redundancy, and disk-level performance optimization to provide efficient error detection and recovery. In this section, we discuss previous work related to data integrity, physical redundancy and performance optimization at disk-level.

Data integrity.

Data integrity has become more important in recent times due to unreliable hardware devices. Many file systems [4, 6, 9, 21, 23] use checksums to verify the integrity of the data stored on disk.

Our work is closely related to IRON file systems [15], a technology which makes file systems more robust to disk errors by computing checksums for all blocks, replicating meta-data blocks for redundancy, and recovering corrupted or inaccessible blocks. As seen from their benchmark results, the overheads of integrity verification, replication, and recovery are higher when performed at the file-system level. Conversely, SRD fulfills these responsibilities inside the disk, avoiding duplication of functionality for multiple file systems. With the help of type information and internal disk state, SRDs can perform these operations more efficiently.

ZFS [22] is another file system that is close to our work. ZFS provides block-level integrity verification, replication, and recovery. ZFS also stores the checksums of each data block in the parent block that points to it. ZFS's recovery mechanism makes use of multiple disks, if available. In contrast, SRD's recovery mechanism works within a single disk. SRD provides functionality similar to the Storage Pool Allocator layer of ZFS, but at disk level. Since SRD checks integrity, replicates, and recovers blocks in a manner that is transparent to other layers in the storage stack, it works with any file system designed for TSDs.

Replicating blocks.

Replicating blocks at the file system and the disk level has been a popular solution for providing redundancy. The Fast File System [12] replicates the super block across all platters of the disk. RAID systems [13] replicate blocks for redundancy. These systems do not have type information inside the disk, hence they cannot replicate blocks selectively (e.g., only metadata blocks) in the disk.

FS2 [7] uses the free space in the file system to replicate blocks in the disk according to their access pattern. This is not a complete solution as the disk cannot capture the access patterns correctly due to three reasons: (1) caching of blocks by OS, (2) changing access patterns, and (3) the fact that replicating all blocks would consume at least half of total disk space. In contrast, type-awareness enables SRDs to know the relationship between blocks stored on the disk and selectively replicate meta-data blocks. We believe that SRDs can also use the information about access patterns of blocks to improve their performance.

Performance optimization in the disk.

The idea of utilizing the available disk bandwidth by interleaving low-priority requests between high priority requests has been explored in freeblock scheduling [10, 11]. SRDs take a similar approach by interleaving low-priority operations such as writing back modified checksum blocks and replicating meta-data blocks between regular block requests, incurring a very small overhead.

7. CONCLUSIONS

In this paper, we have shown that with pointer information at the disk-level, effective integrity assurance and data recoverability mechanisms can be built inside the disk. Our pointer-guided prefetching mechanism for checksum blocks achieves negligible I/O overheads for reading redundant data used for integrity checking. With selective block replication, we have handled one of the most important forms of data recoverability, by just replicating a small fraction of blocks on disk. Intelligent placement and prefetching of checksum blocks ensure that the overall overheads are negligible for SRDs as shown by our evaluation results. We believe that our design represents an effective choice for building more reliable disks while remaining compatible with a wide-range of storage applications such as file systems and databases.

8. ACKNOWLEDGMENTS

We thank the anonymous reviewers for their comments, and Avishay Traeger and Sean Callanan for their insightful comments on earlier drafts of the paper.

9. REFERENCES

[1] W. Barlett and L. Spainbower. Commercial fault tolerance: A tale of two systems. In *Proceedings of the IEEE Transactions on Dependable and Secure Computing*, pages 87–96, January 2004.

[2] P. Corbett, B. English, A. Goel, T. Grcanac, S. Kleiman, J. Leong, and S. Sankar. Row-Diagonal Parity for Double Disk Failure Correction. In *Proceedings of the Third USENIX Conference on File and Storage Technologies (FAST 2004)*, pages 1–14, San Francisco, CA, March/April 2004. USENIX Association.

[3] T. E. Denehy, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Bridging the information gap in storage protocol stacks. In *Proceedings of the Annual USENIX Technical Conference*, pages 177–190, Monterey, CA, June 2002. USENIX Association.

[4] K. Fu, M. F. Kaashoek, and D. Mazières. Fast and Secure Distributed Read-Only File System. In *Proceedings of the 4th Usenix Symposium on Operating System Design and Implementation (OSDI '00)*, pages 181–196, San Diego, CA, October 2000. USENIX Association.

[5] G. R. Ganger. Blurring the Line Between OSes and Storage Devices. Technical Report CMU-CS-01-166, CMU, December 2001.

[6] S. Ghemawat, H. Gobioff, and S. T. Leung. The Google File System. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP '03)*, pages 29–43, Bolton Landing, NY, October 2003. ACM SIGOPS.

[7] H. Huang, W. Hung, and K. Shin. FS2: Dynamic Data Replication in Free Disk Space for Improving Disk Performance and Energy Consumption. In *Proceedings of the 20th ACM Symposium on Operating Systems Principles (SOSP '05)*, pages 263–276, Brighton, UK, October 2005. ACM Press.

[8] H. Kari, H. Saikkonen, and F. Lombardi. Detection of defective media in disks. In *Proceedings of the IEEE International Workshop on Defect and Fault Tolerance in VLSI Systems*, Washington, DC, 1993. IEEE Computer Society.

[9] 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. USENIX Association.

[10] C. Lumb, J. Schindler, G. R. Ganger, and D. F. Nagle. Towards higher disk head utilization: Extracting free bandwidth from busy disk drives. In *Proceedings of the 4th Usenix Symposium on Operating System Design and Implementation (OSDI '00)*, pages 87–102, San Diego, CA, October 2000. USENIX Association.

[11] C. R. Lumb, J. Schindler, and G. R. Ganger. Freeblock Scheduling Outside of Disk Firmware. In *Proceedings of the First USENIX Conference on File and Storage Technologies (FAST 2002)*, pages 275–288, Monterey, CA, January 2002. USENIX Association.

[12] M. K. McKusick, W. N. Joy, S. J. Leffler, and R. S. Fabry. A fast file system for UNIX. *ACM Transactions on Computer Systems*, 2(3):181–197, August 1984.

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

[14] H. Patterson, S. Manley, M. Federwisch, D. Hitz, S. Kleinman, and S. Owara. SnapMirror: File System Based Asynchronous Mirroring for Disaster Recovery. In *Proceedings of the First USENIX Conference on File and Storage Technologies (FAST 2002)*, pages 117–129, Monterey, CA, January 2002. USENIX Association.

[15] V. Prabhakaran, N. Agrawal, L. N. Bairavasundaram, H. S. Gunawi, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. IRON File Systems. In *Proceedings of the 20th ACM Symposium on Operating Systems Principles (SOSP '05)*, pages 206–220, Brighton, UK, October 2005. ACM Press.

[16] R. L. Rivest. RFC 1321: The MD5 Message-Digest Algorithm. In *Internet Activities Board*. Internet Activities Board, April 1992.

[17] Fred B. Schneider. Implementing fault-tolerant services using the state machine approach: a tutorial. *ACM Computer Survey*, 22(4):219–319, 1990.

[18] G. Sivathanu, S. Sundararaman, and E. Zadok. Type-Safe Disks. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation (OSDI 2006)*, pages 15–28, Seattle, WA, November 2006. ACM SIGOPS.

[19] G. Sivathanu, C. P. Wright, and E. Zadok. Ensuring data integrity in storage: Techniques and applications. In *Proceedings of the First ACM Workshop on Storage Security and Survivability (StorageSS 2005)*, pages 26–36, FairFax, VA, November 2005. ACM.

[20] M. Sivathanu, V. Prabhakaran, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Improving Storage System Availability with D-GRAID. In *Proceedings of the Third USENIX Conference on File and Storage Technologies (FAST 2004)*, pages 15–30, San Francisco, CA, March/April 2004. USENIX Association.

[21] C. A. Stein, J. H. Howard, and M. I. Seltzer. Unifying file system protection. In *Proceedings of the Annual USENIX Technical Conference*, pages 79–90, Boston, MA, June 2001. USENIX Association.

[22] Sun Microsystems, Inc. Solaris ZFS file storage solution. *Solaris 10 Data Sheets*, 2004. `www.sun.com/software/solaris/ds/zfs.jsp`.

[23] C. P. Wright, M. Martino, and E. Zadok. NCryptfs: A secure and convenient cryptographic file system. In *Proceedings of the Annual USENIX Technical Conference*, pages 197–210, San Antonio, TX, June 2003. USENIX Association.