Loris - A Dependable, Modular File-Based Storage Stack

Raja Appuswamy, David C. van Moolenbroek, Andrew S. Tanenbaum
Dept of Computer Science, Vrije Universiteit, Amsterdam, Netherlands
{*raja, dcvmoole, ast*}@cs.vu.nl

Abstract—The arrangement of file systems and volume management/RAID systems, together commonly referred to as the storage stack, has remained the same for several decades, despite significant changes in hardware, software and usage scenarios. In this paper, we evaluate the traditional storage stack along three dimensions: reliability, heterogeneity and flexibility. We highlight several major problems with the traditional stack. We then present Loris, our redesign of the storage stack, and we evaluate several aspects of Loris.

I. Introduction

Over the past several years, the storage hardware landscape has changed dramatically. A significant drop in the cost per gigabyte of disk drives has made techniques that require a full disk scan, like *fsck* or whole disk backup, prohibitively expensive. Large scale installations handling petabytes of data are very common today, and devising techniques to simplify management has become a key priority in both academia and industry. Research has revealed great insights into the reliability characteristics of modern disk drives. "Fail-partial" failure modes [15] have been studied extensively and end-to-end integrity assurance is more important now than ever before. The introduction of SSDs and other flash devices is sure to bring about a sea change in the storage subsystem. Radically different price/performance/reliability trade-offs have necessitated re-thinking several aspects of file and storage management [11], [6]. In short, the storage world is rapidly changing and our approach to making storage dependable must change, too.

Traditional file systems were written and optimized for block-oriented hard disk drives. With the advent of RAID techniques [14], storage vendors started developing high-performance, high-capacity RAID systems in both hardware and software. The block-level interface between the file system and disk drives provided a convenient, backward-compatible abstraction for integrating RAID algorithms.

As installations grew in size, administrators needed more flexibility in managing file systems and disk drives. Volume managers [21] were designed as block-level drivers to break the "one file system per disk" bond. By providing logical volumes, they abstracted out details of physical storage and thereby made it possible to resize file systems/volumes on the fly. Logical volumes also served as units of policy assignment and quota enforcement. Together, we refer to the RAID and volume management solutions as the *RAID layer* in this paper.

This arrangement of file system and RAID layers, as shown in Figure 1(a), has been referred to as the storage stack [4]. Despite several changes in hardware landscape, the traditional storage stack has remained the same for several decades. In this paper, we examine the block-level integration of RAID and volume management along three dimensions: reliability, flexibility, and heterogeneity. We highlight several major problems with the traditional stack along all three dimensions. We then present Loris, our new storage stack. Loris improves modularity by decomposing the traditional file system layer into several self-contained layers, as shown in Figure 1(b). It improves reliability by integrating RAID algorithms at a different layer compared to the traditional stack. It supports heterogeneity by providing a file-based stack in which the interface between layers is a standardized file interface. It improves flexibility by automating device administration and enabling per-file policy selection.

This paper is structured as follows. In Sec. 2, we explain in detail the problems associated with the traditional stack. In Sec. 3, we briefly outline some attempts taken by others in redesigning the storage stack and also explain why other approaches fail to solve all the problems. In Sec. 4, we introduce Loris and explain the responsibilities and abstraction boundaries of each layer in the new stack. We also sketch the realization of these layers in our prototype implementation. In Sec. 5, we present the advantages of Loris. We then evaluate the performance of Loris in Sec. 6 and conclude in Sec. 7.

II. Problems with the Traditional Storage Stack

In this section, we present some of the most important problems associated with the traditional storage stack. We present the problems along three dimensions: reliability, flexibility and heterogeneity.

A. Reliability

The first dimension is reliability. This includes the aspects of data corruption, system failure, and device failure.

1) Data Corruption: Modern disk drives are not "fail-stop." Recent research has analyzed several "fail-partial" failure modes of disk drives. For example, a *lost write* happens when a disk does not write a data block. A *misdirected* write by the disk results in a data block being
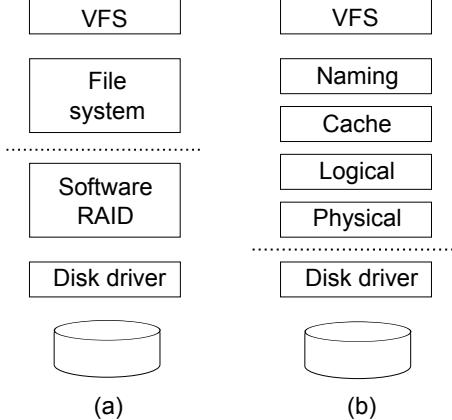
Figure 1: The figure depicts (a) the arrangement of layers in the traditional stack, and (b) the new layering in Loris. The layers above the dotted line are file-aware, the layers below are not.

written at a different position than its intended location. A *torn write* happens when a write operation fails to write all the sectors of a multisector data block. In all these cases, if the disk controller (incorrectly) reports back success, data is silently corrupted. This is a serious threat to data integrity and significantly affects the reliability of the storage stack.

File systems and block-level storage systems detect data corruption by employing various checksumming techniques [16]. The level of reliability offered by a checksumming scheme depends heavily on what is checksummed and where the checksum is stored. Checksums can be computed on a per-sector or per-block (file system block) basis, where a block is typically 2, 4, 8, or more sectors. Using per-sector checksums does not protect one against any of the failure cases mentioned above if checksums are stored with the data itself. Block checksums, on the other hand, protect against torn writes but not against misdirected or lost writes.

In yet another type of checksumming, called *parental checksumming*, the checksum of a data block is stored with its parent. For instance, a parental checksumming implementation could store block checksums in the inode, right next to the block pointers. These checksums would then be read in with the inode and used for verification. Formal analysis has verified that parental checksumming detects all of the aforementioned sources of corruption [13].

However, using parental checksumming in the current storage stack increases the chance of data loss significantly [13]. Since the parental relationship between blocks is known only to the file system, parental checksums can be used only by the file system layer. Thus, while file system initiated reads can be verified, any reads initiated by the RAID layer (for partial writes, scrubbing, or recovery) escape verification. As a result, a corrupt data block will not only go undetected by the RAID layer, but could also be used for parity recomputation, causing parity pollution [13], and hence data loss.

2) System Failure: Crashes and power failures pose a *metadata consistency* problem for file systems. Several techniques like soft updates and journaling have been used to reliably update metadata in the face of such events. RAID algorithms also suffer from a *consistent update* problem. Since RAID algorithms write data to multiple disk drives, they must ensure that the data on different devices are updated in a consistent manner.

Most hardware RAID implementations use NVRAM to buffer writes until they are made durable, cleanly side-stepping this problem. Several software RAID solutions, on the other hand, resort to whole disk *resynchronization* after an unclean shutdown. During resynchronization, all data blocks are read in, parity is computed, and the computed parity is verified with the on-disk parity. If a mismatch is detected, the newly computed parity is written out replacing the on-disk parity.

This approach—in addition to becoming increasingly impractical due to the rapid increase in disk capacity—has two major problems: (1) it increases the vulnerability window within which a second failure can result in data loss. This problem is also known as the "RAID write hole", and (2) it adversely affects availability, as the whole disk array has to be offline during the resynchronization period [3].

The other approach adopted by some software RAID implementations is journaling a block bitmap to identify regions of activity during the failure. While this approach reduces resynchronization time, it has a negative impact on performance [5], and results in functionality being duplicated across multiple layers.

3) Device Failure: Storage array implementations protect against a fixed number of disk failures. For instance, a RAID 5 implementation protects against a single disk failure. When an unexpected number of failures occur, the storage array comes to a grinding halt. An ideal storage array however, should degrade gracefully. The amount of data inaccessible should be proportional to the number of failures in the system. Research has shown that to achieve such a property, a RAID implementation must provide: (1) selective replication of metadata to make sure that the directory hierarchy remains navigable at all times, and (2) fault-isolated positioning of files so that a failure of any single disk results in only files on that disk being inaccessible [17].

By recovering *files* rather than *blocks*, file-level RAID algorithms reduce the amount of data that must be recovered, thus shrinking the vulnerability window before a second failure. Even a second failure during recovery results in the loss of only some file(s), which can be restored from backup sources, compared to block-level RAID where the entire array must be restored. None of these functionalities can be provided by the traditional storage stack as the traditional RAID implementation operates strictly below a block interface.

B. Flexibility

We discuss two points pertaining to flexibility: management and policy assignment.

1) Management Flexibility: While traditional volume managers make device management and space allocation more flexible, they introduce a series of complex, error prone administrative operations, most of which should be automated. For instance, a simple task such as adding a new disk to the system involves several steps like creating a physical volume, adding it to a volume group, expanding logical volumes and finally resizing file systems. While new models of administering devices, like the storage pool model [1], are a huge improvement, they still suffer from other problems, which we will describe in the next section.

In addition to device management complexity, software-based RAID solutions expose a set of tunable parameters for configuring a storage array based on the expected workload. It has been shown that an improperly configured array can render layout optimizations employed by a file system futile [19]. This is an example of the more general "information gap" problem [4]— different layers performing different optimizations unaware of the effect they might have on the overall performance.

2) Policy Assignment Flexibility: Different files have different levels of importance and need different levels of protection. However, policies like the RAID level to use, encryption, and compression, are only available on a per-volume basis rather than on a per-file basis. Several RAID implementations even lock-in the RAID levels and make it impossible to migrate data between RAID levels. In cases where migration is supported, it usually comes at the expense of having to perform a full-array dump and restore. In our view, an ideal storage system should be flexible enough to support policy assignment on a per-file, per-directory, or a per-volume basis. It should support migration of data between RAID levels on-the-fly without affecting data availability.

C. Heterogeneity Issues

New devices are emerging with different data access granularities and storage interfaces. Integrating these devices into the storage stack has been done using two approaches that involve either extending either the file system layer, or the block-level RAID layer with new functionality.

The first approach involves building file systems that are aware of device-specific abstractions [11]. However, as the traditional block-based RAID layer exposes a block interface, it is incompatible with these file systems. As a result, RAID and volume management functionalities must be implemented from scratch for each device family.

The second approach is to be backward compatible with traditional file systems. This is typically done by adding a new layer that translates block requests from the file system to device-specific abstractions [6]. This layer cannot be integrated between the file system and RAID layers, as it is incompatible with the RAID layer. Hence, such a layer has to either reimplement RAID algorithms for the new device family, or be integrated below the RAID layer. This integration retains compatibility with both traditional file system and RAID implementations. However, this has the serious effect of widening the information gap by duplicating functionality. For instance, both the file system and translation layers now try to employ layout optimizations – something that is completely unwarranted. The only way to avoid this duplication is by completely redesigning the storage stack from scratch.

III. SOLUTIONS PROPOSED IN THE LITERATURE

Several approaches have been taken to solving some of the problems mentioned in the previous section. However, none of these approaches solve all these problems by design. In this section, we highlight only the most important techniques. We classify the approaches taken into four types, namely: (1) inferring information, (2) sharing information, (3) refactoring the storage stack, and (4) extending the stack with stackable filing.

One could compensate for the lack of information in the RAID layer by having it infer information about the file system layer. For instance, semantically smart disks [17] infer file system specific information (block typing and structure information), and use the semantic knowledge to improve RAID flexibility, availability, and performance. However, by their very nature, they are file-system-specific, making them nonportable.

Instead of inferring information, one could redesign the interface between the file system and RAID layers to share information. For example, ExRAID [4], a software RAID implementation, provides array related information (such as disk boundaries and transient performance characteristics of each device) to an informed file system (I-LFS), which uses it to make informed data placement decisions.

While both inferring and sharing information can be used to add new functionality, they do not change the fundamental division of labor between layers. Hence, most of the problems we mentioned remain unresolved.

A few projects have refactored the traditional storage stack. For instance, ZFS [1]'s storage stack consists of three layers, the ZFS Posix Layer (ZPL), the Data Management Unit (DMU), and the Storage Pool Allocator (SPA). ZFS solves the reliability and flexiblity problems we mentioned earlier by merging block allocation with RAID algorithms in its SPA layer. SPA exposes a virtual block abstraction to DMU and acts as a multidisk block allocator. However, because SPA exposes a block interface, it suffers from the same heterogeneity problems as the RAID layer. In addition, we believe that layout management and RAID are two distinct functionalities that should be modularized in separate layers.

RAIF [12] provides RAID algorithms as a stackable file system. RAID algorithms in RAIF work on a per-file basis. As it is stackable, it is very modular and can be layered on any file system, making it device independent. While this does solve the flexibility and heterogeneity problems, it does not solve the reliability problems.

IV. THE DESIGN OF LORIS

We now present our new storage stack, Loris. The Loris stack consists of four layers in between the VFS layer and the disk driver layer, as shown in Figure 1(b). Within the stack, the primary abstraction is the *file*. Each layer offers an interface for creating and manipulating files to the layer above it, exposing per-file operations such as *create*, *read*, *write*, *truncate*, and *delete*. A generic *attribute* abstraction is used to both maintain per-file metadata, and exchange information within the stack. The *getattr* and *setattr* operations retrieve and set attributes. Files and attributes are stored by the lowest layer.

We implemented a Loris prototype on the MINIX 3 multiserver operating system [9]. The modular and fault-tolerant structure of MINIX 3 allows us to quickly and easily experiment with invasive changes. Moreover, we plan to apply ongoing research in the areas of live updates [7] and many-core support to our work. MINIX 3 already provides VFS and the disk drivers, each running as a separate process, which improves dependability by allowing failed OS components to be replaced on the fly.

The four layer implementations of the prototype can be combined into one process, or separated out into individual processes. The single-process setup allows for higher performance due to fewer context switches and less memory copying overhead. The multiprocess setup by nature imposes a very strict separation between layers, provides better process fault isolation in line with MINIX 3's dependability objectives [9], and is more live-update-friendly [7]. The difference between these configurations is abstracted away by a common library that provides primitives for communication and data exchange.

We will now discuss each of the layers in turn, starting from the lowest layer.

A. The Physical Layer

The lowest layer in the Loris stack is the *physical layer*. The physical layer algorithms provide device-specific layout schemes to store files and attributes. They offer a *fail-stop physical file* abstraction to the layers above it. By working on physical files, the rest of the layers are isolated from device-specific characteristics of the underlying devices (such as access granularity).

As the physical files are fail-stop, every call that requests file data or attributes, returns either a result that has been verified to be free of corruption or an error. To this end, every physical layer algorithm is required to implement parental checksumming. To repeat, above the physical layer, there is no silent corruption of data. A torn, lost, or misdirected write is converted into a hard failure that is passed upward in the stack.

In our prototype, each physical device is managed by a separate, independent instance of one of the physical layer algorithms. We call such an instance a *module*. Each physical module has a global *module identifier*, which it uses to register to the logical layer at startup.

1) On-Disk Layout: The on-disk layout of our prototype is based directly on the MINIX 3 File System (MFS) [20], a traditional UNIX-like file system. We have extended it to support parental checksums. Figure 2 shows a high-level view of the layout. We deliberately chose to stay close to the original MFS implementation so that we could measure the overhead incurred by parental checksumming.

Each physical file is represented on disk by an *inode*. Each inode has an *inode number* that identifies the physical file. The inode contains space to store persistent attributes, as well as 7 direct, one single indirect and one double indirect *safe block pointers*. Each safe block pointer contains a block number and a CRC32 checksum of the block it points to. The single and double indirect blocks store such pointers as well, to data blocks and single indirect blocks, respectively. All file data are therefore protected directly or indirectly by checksums in the file inode.

The inodes and other metadata are protected by means of three special on-disk files. These are the inode bitmap file, the block bitmap file, and the "root file." The bitmap file inodes and their indirect blocks contain safe block pointers to the bitmap blocks. The root file forms the hierarchical parent of all the inodes—its data blocks contain checksums over all inodes, including the bitmap file inodes. The checksums in the root file are stored and computed on a per-inode basis, rather than a per-block basis. The checksum of the root file's inode is stored in the superblock.

Figure 3 shows the resulting parental checksumming hierarchy. The superblock and root data blocks contain only checksums; the inodes and indirect blocks contain safe block pointers. Please note that this hierarchy is used only for checksumming—unlike copy-on-write layout schemes [10], blocks are updated in-place.

2) Delayed Checksum Verification: One of the main drawbacks of parental checksumming is cascading writes. Due to the inherent nature of parental checksumming, a single update to a block could result in several blocks being updated all the way up the parental tree to the root. Updating all checksums in the mainstream write path would slow down performance significantly.

The physical layer prevents this by delaying the checksum computation, using a small metadata block cache that is type-aware. By knowing the type of each block, this cache makes it possible to perform checksum computation only when a block is written to or read from the underlying
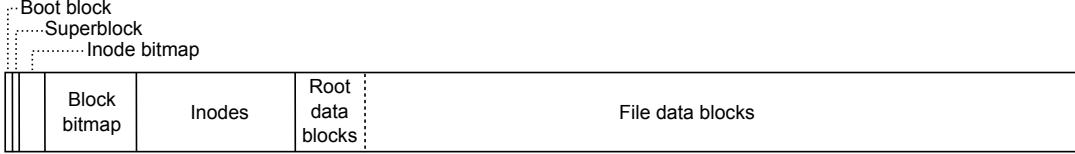
Figure 2: High-level overview of the on-disk layout used by the physical layer prototype.

device. For instance, by knowing whether a block is a bitmap or inode block, it can compute and update checksums in the bitmap and root file inodes lazily, that is, only right before flushing these metadata blocks from the cache.

3) Error Handling: Parental checksumming allows the physical layer to detect corrupt data. When a physical module detects a checksum error in the data or indirect block of a file, it marks that portion of the file as corrupted. If the file's inode checksum, as stored in the root file, does not match the checksum computed over the actual inode, then the entire file is marked as corrupted. In both cases, reads from the corrupt portion will result in a checksum error being returned to the logical layer. The physical module uses two attributes in the file's inode, begin range and end range, to remember this *sick range*.

For instance, consider a read request to a physical module for the first data block of a file. If the physical module detects a checksum mismatch on reading in the data block, it sets the begin and end range attributes to 0 and 4095 respectively, and responds back to the logical layer with a checksum error. We will detail the recovery process initiated by the logical layer when we describe its error handling mechanism.

In contrast to inodes and data blocks, if a part of the other on-disk metadata structures is found corrupted, the physical module will shut down for offline repair. While we could have supported metadata replication to improve reliability, we chose not to do so for the first prototype to stay as close as possible to the original MFS implementation in order to get honest measurements.

If the underlying device driver returns an error or times out, the physical module will retry the operation a number of times. Upon repeated failure, it returns back an I/O error to the logical layer. An I/O error from the physical module is an indication to the logical layer that a fatal failure has occurred and that the erroneous device should not be used anymore.

B. The Logical Layer

The logical layer implements RAID algorithms on a per-file basis to provide various levels of redundancy. The logical layer offers a *reliable logical file* abstraction to the layers above. It masks errors from the physical layer whenever there is enough data redundancy to recover from them. One logical file may be made up of several independent physical files, typically each on a different physical module, and possibly at different locations. As such, the logical layer acts as a centralized multiplexer over the individual modules in the physical layer.

Our prototype implements the equivalents of RAID levels 0, 1, and 4—all of these operate on a per-file basis. There is only one module instance of the logical layer, which operates across all physical modules.

1) File Mapping: The central data structure in our logical layer prototype is the *mapping*. The mapping contains an entry for every logical file, translating *logical file identifiers* to *logical configurations*. The logical configuration of a file consists of (1) the file's RAID level, (2) the stripe size used for the file (if applicable), and (3) a set of one or more physical files that make up this logical file, each specified as a physical module and inode number pair. The RAID level implementations decide how the physical files are used to make up the logical file.
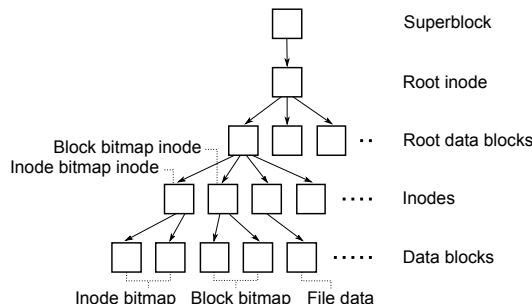

Figure 3: Parental checksumming hierarchy used by the physical layer prototype. With respect to parental checksumming, the two special bitmap files are treated as any other files. Indirect blocks have been omitted in this figure.
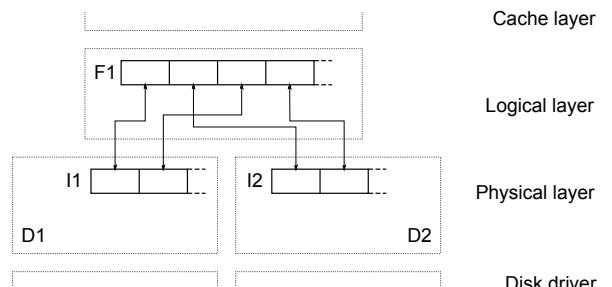

Figure 4: An example of the file abstractions provided by the logical and physical layers. The logical layer exposes a logical file, F1, which is constructed out of two physical files, namely I1 on physical module D1 and I2 on physical module D2, by means of striping.

The *create* operation creates a mapping entry for a given logical file identifier, with a given configuration (more about this later). For all other file operations coming in from above, the logical layer first looks up the file's logical configuration in the mapping. The corresponding RAID algorithm is then responsible for calling appropriate operations on physical modules.

Figure 4 shows how a logical file that is striped across two devices, is constructed out of two independent physical files. The mapping entry for this file F1 could look like this: F1=<raidlevel=0, stripesize=4096, physicalfiles=<D1:I1, D2:I2>>. The entry specifies a RAID level of 0, a stripe size of 4096 bytes, and two physical files: file I1 on physical module D1 and file I2 on physical module D2. Now consider a read request for the first 16384 bytes of this file coming down to the logical layer. Upon receiving the read request, the logical layer looks up the entry for F1 in its mapping, and passes on the call to the RAID 0 algorithm. The RAID 0 code uses the entry to determine that logical bytes 0-4095 are stored as bytes 0-4095 in physical file D1:I1, logical bytes 4096-8191 are stored as bytes 0-4095 in file D2:I2, logical bytes 8192-12287 are stored as bytes 4096-8191 in file D1:I1, and logical bytes 12288-16383 are stored as bytes 4096-8191 in file D2:I2. The logical layer issues two read requests, one to D1 for the first 8192 bytes of I1, and the other to D2 for the first 8192 bytes of I2. The results are combined to form a "flat" result for the layer above.

The mapping itself is a logical file. The logical configuration of this file is hardcoded. The mapping file is crucial for accessing any other files, and is therefore mirrored across all physical modules for increased dependability. It uses the same static inode number on all physical modules.

2) Error Handling: When the logical layer gets a checksum error in response to an operation on a physical module, it will restore the correct data for the file involved in the operation if enough data redundancy is present. If on-the-fly restoration is not possible, the logical layer fails the operation with an I/O error.

For instance, let us consider a read request for the first block of a file mirrored on two physical modules P1 and P2. If P1 responds back with a checksum error, the logical layer first retrieves the begin and end sick range attributes from P1. Assuming that only the first block was corrupt, these values would be 0 and 4095. The logical layer then issues a read request to module P2, for data in the range 0–4095. If this read request succeeds, the logical layer issues a write request to P1 for this data range, thereby performing on-the-fly recovery. It finally clears the sick range by resetting begin and end ranges to their defaults.

When the logical layer gets an I/O error from a physical module, it considers this to be a permanent error, and disables the physical module. The logical layer will continue to serve requests for affected files from redundant copies

where possible, and return I/O errors otherwise.

C. The Cache Layer

The cache layer caches file data in main memory. This layer may be omitted in operating systems that provide a unified page cache. As MINIX 3 does not have a unified page cache, our prototype implements this layer. The cache layer is also needed on systems that do not have any local physical storage such as PDAs and other small mobile devices.

1) File-Based Caching: The prototype cache is file-aware and performs readahead and eviction of file data on a per-file basis. Files are read from the lower layers in large readahead chunks at a time, and only entire files are evicted from the cache. The cache maintains file data at the granularity of memory pages.

Early experiments showed a large performance penalty incurred by small file writes. Small file writes were absorbed completely by the cache until a sync request was received or the cache needed to free up pages for new requests. During eviction, each file would be written out by means of an individual *write* operation, forcing the physical layer to perform a large number of small random writes. To counter this problem, we introduced a *vwrite* call to supply a vector of write operations for several files. The cache uses this call to pass down as many dirty pages as possible at once, eventually allowing the physical modules to reorder and combine the small writes.

2) Problems with Delayed Allocation: The cache delays writes, so that *write* calls from above can be satisfied very quickly. This results in allocation of data blocks for these writes to be delayed until the moment that these blocks are flushed out. This delayed allocation poses a problem in the stack. Because of the abstraction provided by the logical layer, the cache has no knowledge about the devices used to store a file, nor about the free space available on those devices. Therefore, when a write operation comes in, the cache cannot determine whether the write will eventually succeed.

Although we have not yet implemented a solution for this in our prototype, the problem can be solved by means of a free-space reservation system exposed by the physical modules through the logical layer to the cache.

D. The Naming Layer

The naming layer is responsible for naming and organizing files. Different naming layer algorithms can implement different naming schemes: for example, a traditional POSIX style naming scheme, or a search-oriented naming scheme based on attributes.

1) POSIX Support: Our prototype implements a traditional POSIX naming scheme. It processes calls coming from the VFS layer above, converting POSIX operations into file operations.

Only the naming layer knows about the concept of directories. Below the naming layer, directories are stored as files. The naming layer itself treats directories as flat arrays of statically sized entries, one per file. Each entry is made up of a file name and a logical file identifier. Again, this layout was chosen for simplicity and to stay close to the original MFS implementation for comparison purposes. A new naming module could implement more advanced directory indexing.

The naming layer is also responsible for maintaining the POSIX attributes of files (file size, file mode, link count, and so on). It uses Loris attributes for this: it uses the *setattribute* call to send down POSIX attribute changes, which are ultimately processed and stored by the physical layer in the file's inode.

2) Policy Assignment: When creating a new file, the naming layer is responsible for picking a new logical file identifier, and an initial logical configuration for the file. The logical configuration may be picked based on any information available to the naming layer: the new file's name, its containing directory, its file type, and possibly any flags passed in by the application creating the file. The chosen logical configuration is passed to lower layers in the *create* call in the form of attributes.

By default, directories are mirrored across all devices in order to provide graceful degradation. Upon getting a *create directory* request from VFS, the naming layer picks a new file identifier for the directory, and sends down a *create* call to the cache, with RAID level 1 specified as the file's logical policy. The cache forwards the call to the logical layer. The logical layer creates a new entry in the mapping for this file, and forwards the create call to all of the physical modules. Upon return, the logical layer stores the resulting inode numbers in the mapping entry as well.

V. The Advantages of Loris

In this section, we highlight how Loris solves all the problems mentioned in Sec. 2 by design. This section has been structured to mirror the structure of Sec. 2 so that readers can match the problems with their corresponding solutions one-to-one.

A. Reliability

We now explain how Loris protects against the three threats to data integrity.

1) Data Corruption: As RAID algorithms are positioned in the logical layer, all requests, both user application initiated reads and RAID initiated reads, are serviced by the physical layer. Thus, any data verification scheme needs to be implemented only once, in the physical layer, for all types of requests. In addition, since the physical layer is file-aware, parental checksumming can be used for detecting all possible sources of corruption. Thus, by requiring every physical layer algorithm to implement parental checksumming, fail-partial failures are converted into fail-stop failures. RAID algorithms can safely provide protection against fail-stop failures without propagating corruption.

2) System Failure: Journaling has been used by several file systems to provide atomic update of system metadata [8]. Modified journaling implementations have also been used to maintain the consistency between data blocks and checksums in the face of crashes [18]. While any such traditional crash recovery techniques can be used with Loris to maintain metadata consistency, we are working on a new technique called *metadata replay*. It protects the system from both hard crashes (power failures, kernel panic, etc.) and soft crashes (modules failing due to bugs). We will provide a brief description of this technique now, but it should be noted that independent of the technique used, Loris does not require expensive whole disk synchronization due to its file-aware nature.

Metadata replay is based on the counterintuitive idea that user data (file data and POSIX attributes), if updated atomically, can be used to regenerate system metadata. To implement this, we have two requirements: (1) a lightweight mechanism to log user data, and (2) some way to restore back the latest consistent snapshot. With highly flexible policy selection in place, the user could log only important files, reducing the overhead of data logging. We plan to add support for selective logging and metadata snapshotting. When a crash occurs, the logical layer coordinates the rollback of all physical layers to a globally consistent state. Then, the logged user data are replayed, regenerating metadata in both logical and physical layers, and bringing the system to new consistent state.

3) Device Failure: Graceful degradation is a natural extension of our design. Since RAID policies can be selected on a per-file basis, directories can be replicated on all devices while file data need not be, thereby providing selective metadata replication. Since the logical layer is file-aware, fault-isolated placement of files can also be provided on a per-file basis. Furthermore, recovery to a hot spare on a disk failure is faster than a traditional RAID array since the logical layer recovers files. As mentioned earlier, file-granular recovery restores only "live data" by nature, i.e., unused data blocks in all physical layers do not have to be restored. Because traditional RAID operates at the block level, it is unaware of which data blocks hold file data, and has to restore all data blocks in the array.

B. Flexibility

The new file-oriented storage stack is more flexible than the traditional stack in several ways. Loris supports automating several administrative tasks, simplifies device management, and supports policy assignment at the granularity of files.

1) Management Flexiblity: Loris simplifies administration by providing a simple model for both device and quota management. It supports automating most of the traditional

Benchmark	MFS	Loris (single-process)			Loris (multiprocess)		
		No checksum	*Layout only*	*Checksum*	*No checksum*	*Layout only*	*Checksum*
PostMark	794.00 (1.00)	729.00 (0.92)	723.00 (0.91)	797.00 (1.00)	760.00 (0.96)	763.00 (0.96)	822.00 (1.04)
Applevel (copy)	79.86 (1.00)	86.41 (1.08)	86.00 (1.08)	101.36 (1.27)	94.66 (1.19)	94.38 (1.18)	109.63 (1.37)
Applevel (build)	58.06 (1.00)	58.50 (1.01)	58.48 (1.01)	60.21 (1.04)	73.68 (1.27)	73.78 (1.27)	75.61 (1.30)
Applevel (find and grep)	16.55 (1.00)	16.90 (1.02)	17.01 (1.03)	19.73 (1.19)	22.61 (1.37)	22.50 (1.36)	25.53 (1.54)
Applevel (delete)	19.76 (1.00)	10.01 (0.51)	10.01 (0.51)	13.06 (0.66)	18.68 (0.95)	18.96 (0.96)	22.00 (1.11)

Table I: Transaction time in seconds for PostMark and wall clock time for applevel benchmarks. Table shows both absolute and relative performance numbers, contrasting MFS with Loris in several configurations.

administrative chores. For instance, when a new device is added to an existing installation, Loris automatically assigns the device a new identifier. A new physical module corresponding to this device type is started automatically and this module registers itself with the logical layer as a potential source of physical files. From here on, the logical layer is free to direct new file creation requests to the new module. It can also change the RAID policy of existing files on-the-fly or in the background. Thus, Loris supports a pooled storage model similar to ZFS.

File systems in ZFS [1] serve as units of quota enforcement. By decoupling volume management from device management, these systems make it possible for multiple volumes to share the same storage space. We are working on a file volume management implementation for Loris. We plan to modify the logical layer to add support for such a volume manager. File volumes in Loris will be able to provide functionalities similar to ZFS since files belonging to any volume can be allocated from any physical module.

2) Policy Assignment Flexiblity: Loris provides a clean split between policy and mechanism. For instance, while RAID algorithms are implemented in the logical layer, the policy that assigns RAID levels to files can be present in any layer. Thus, while the naming layer can assign RAID levels on a per-file, per-directory or even per-type basis [12], the logical layer could assign policies on a per-volume basis or even globally across all files.

C. Heterogeneity

All of the aforementioned functionalities are device-independent. By having the physical layer provide a physical file abstraction, algorithms above the physical layer are isolated from device-specific details. Thus, Loris can be used to set up an installation where disk drives coexist with byte-granular flash devices and Object-based Storage Devices (OSDs), and the administrator would use the same management primitives across these device types. In addition, as the physical layer works directly on the device without being virtualized, device-specific layout optimizations can be employed without creating an information gap.

VI. EVALUATION

In this section, we will evaluate several performance aspects of our prototype. We also evaluated the ability of Loris to perform on-the-fly data recovery, and its availability

under unexpected failures. However, due to lack of space, we were unable to include these results in this paper. These evaluations can be found in [2].

A. Test Setup

All tests were conducted on an Intel Core 2 Duo E8600 PC, with 4 GB RAM, and four 500 GB 7200RPM Western Digital Caviar Blue SATA hard disks (WD5000AAKS), three of which were connected to separate ICIDU SATA PCI EXPRESS cards. We ran all tests on 8 GB test partitions at the beginning of the disks. In experiments where Loris is compared with MFS, both were set up to work with a 32 MB buffer cache.

B. Performance Evaluation

In this section we present the performance evaluation of Loris. We first present an evaluation of the overhead of two important aspects of our new architecture: the parental checksumming scheme as implemented in the physical layer, and the process isolation provided by splitting up our stack into separate processes. We then present an evaluation of our file-level RAID implementation.

1) Checksumming and Process Isolation: We now evaluate parental checksumming and process isolation using two macrobenchmarks: (1) PostMark, configured to perform 20,000 transactions on 5,000 files, spread over 10 subdirectories, with file sizes ranging from 4 KB to 1 MB, and read/write granularities of 4 KB, and (2) an application-level macrobenchmark, which we will refer to henceforth as *applevel*, consists a set of very common file system operations including copying, compiling, running find and grep, and deleting.

We tested three checksumming schemes: no checksumming, checksumming layout only, and full checksumming. In the layout-only scheme, the CRC32 routine has been substituted by a function that always returns zero. This allowed us to measure only the I/O overhead imposed by parental checksumming. We ran these three schemes in both the single-process and the multiprocess Loris versions, yielding six configurations in total.

Table I shows the performance of all six new configurations, compared to the MFS baseline. Loris outperforms MFS in most configurations with PostMark. This is primarily due to the fact the delete algorithm used by Loris performs

Benchmark	Loris (without checksumming)				Loris (with checksumming)			
	RAID 0-4	*RAID 1-1*	*RAID 1-2*	*RAID 4-4*	*RAID 0-4*	*RAID 1-1*	*RAID 1-2*	*RAID 4-4*
PostMark	195.00	197.00	205.00	268.00	228.00	204.00	214.00	319.00

Table II: Transaction time in seconds for different RAID levels. Each column RAID X-Y shows the performance of RAID level X in a Y-disk configuration.

better than MFS, as seen in the applevel delete benchmark in Table I.

Looking at the applevel results, it can be seen that the single-process case suffers from an overhead of about 8 % compared to MFS during the copying phase. This overhead is due to the difference in caching logic between Loris and MFS.

The block-level buffer cache in MFS makes it possible to merge and write out many physically contiguous files during sync periods. Since Loris has a file-level cache, it is unaware of the physical file layout and hence might make less-than-optimal file evictions. In addition, our prototype also limits the number of files that can be written out during a vectored write call, to simplify implementation. These two factors result in a slight overhead, which is particularly noticeable for workloads with a very large number of small files.

Since the copy phase involves copying over 75,000 files, of which a significant percentage is small, there is an 8 % overhead. Even though the overhead is small, we plan on introducing the notion of *file group identifiers*, to enable the passing file relationship hints between layers. The cache layer could then use this information to evict physically contigous files during a vectored write operation. This and several other future optimizations should remove this overhead completely.

Another important observation is the fact that in both single-process and multiprocess configurations, the checksum layout incurs virtually no overhead. This means that the entire parental checksumming infastructure is essentially free. The actual checksum computation, however, is not, as illustrated by a 7 % overhead (over no checksum case) for PostMark, and 3-19 % overhead in applevel tests.

It should be noted that with checksumming enabled, *every* file undergoes checksum verification. We would like to point out that with per-file policy selection in place, we could reduce the overhead easily, by either checksumming only important file data, or by adopting other lightweight verification approaches as opposed to CRC32 (such as XOR-based parity). For example, we could omit checksumming compiler temporaries and other easily regeneratable files.

We also see that the multiprocess configuration of Loris suffers consistently, with an overhead ranging between 11-45 %. This contradicts the results from PostMark, where the multiprocess case has an overhead of only about 3 % compared to the single-process case. This is again due to the fact that the applevel benchmark has a large number of small files compared to PostMark. Data copying and context switching overheads constitute a considerable portion of the elapsed time when small files dominate the workload. With large files, these overheads are amortized over the data transfer time. We confirmed this with separate microbenchmarks, not shown here, involving copying over a large number of small files.

2) File-Level RAID: In this section, we evaluate our RAID implementation. We test two RAID 1 configurations: (1) RAID 1 on a single disk, and (2) RAID 1 with mirroring on 2 disks. The RAID 0 and RAID 4 implementations use all four disks, with RAID 0 configured to use 60 KB stripe units, and RAID 4 80 KB stripe units for all files. These stripe sizes align full stripe writes with the maximum number of blocks in a vectored write request (240 KB).

We ran PostMark in a different configuration compared to the earlier benchmark—20,000 transactions on 60,000 files, distributed across 600 directories, with file sizes ranging from 4 KB to 10 KB. Small-file workloads are challenging for any RAID implementation since they create lots of partial writes. We chose this benchmark to evaluate how our file-level RAID implementation handles small files.

The most important observation from the PostMark results is that RAID 4 is much slower than RAID 1 with mirroring. The three main reasons for this slowdown are as follows. (1) RAID 4 suffers from the partial-write problem we mentioned earlier. It is important to note that such partial writes would translate into partial stripe requests in a block-level RAID implementation. (2) Parity computation in RAID 4 is expensive compared to mirroring in RAID 1. Both block and file RAID implementations share these two problems. (3) Our implementation of RAID 4 negates the advantages of vectored writes for small files.

To illustrate this problem, consider a vectored write request at the logical layer. The algorithms used to process this request in our implementation are very similar to a block-level RAID one. Each request is broken down into individual file requests, which are further divided into constituent stripe requests, and each stripe request is processed separately. In our current prototype, we implemented sequential processing of stripe requests. Thus, if the workload consists of many small files, a vectored write gets translated into single writes for each file, negating the benefit of vectoring the write request.

The solution to all the aforementioned problems is simple in our case. Parity amortizes the cost of redundancy only when write requests span multiple stripe units. Thus, we are better off using RAID 1 for small files. As our RAID implementation is file-aware, we can monitor and collect file read/write statistics, and use it to (re)assign appropriate

RAID levels to files. Matching file access patterns to storage configurations is future work.

VII. Conclusion

Despite dramatic changes in the storage landscape, the interfaces between the layers and the division of labor among layers in the traditional stack have remained the same. We evaluated the traditional stack along several different dimensions, and highlighted several major problems that plague the compatibility-driven integration of RAID algorithms. We proposed Loris, a file-level storage stack, and evaluated several performance aspects of our prototype.

References

[1] Sun microsystems, solaris zfs file storage solution. solaris 10 data sheets, 2004.

[2] APPUSWAMY, R., VAN MOOLENBROEK, D. C., AND TANENBAUM, A. S. Loris - a dependable, modular file-based storage stack. Technical Report IR-CS-59, Department of Computer Science, Vrije Universiteit, Amsterdam, September 2010.

[3] BROWN, A., AND PATTERSON, D. A. Towards availability benchmarks: a case study of software raid systems. In *In Proc. of the 2000 USENIX Ann. Tech. Conf.* (2000), pp. 263–276.

[4] DENEHY, T. E., ARPACI-DUSSEAU, A. C., AND ARPACI-DUSSEAU, R. H. Bridging the Information Gap in Storage Protocol Stacks. In *The Proc. of the USENIX Ann. Tech. Conf. (USENIX '02)* (June 2002), pp. 177–190.

[5] DENEHY, T. E., ARPACI-DUSSEAU, A. C., AND ARPACI-DUSSEAU, R. H. Journal-guided resynchronization for software raid. In *FAST'05: Proc. of the Fourth USENIX Conf. on File and Storage Technologies* (2005), USENIX Association, pp. 7–7.

[6] GAL, E., AND TOLEDO, S. Algorithms and data structures for flash memories. *ACM Comput. Surv. 37*, 2 (2005), 138–163.

[7] GIUFFRIDA, C., AND TANENBAUM, A. S. Cooperative update: a new model for dependable live update. In *HotSWUp '09: Proc. of the Second Intl. Workshop on Hot Topics in Software Upgrades* (2009), ACM, pp. 1–6.

[8] HAGMANN, R. Reimplementing the cedar file system using logging and group commit. In *SOSP '87: Proc. of the Eleventh ACM Symp. on Operating Systems Principles* (1987), ACM, pp. 155–162.

[9] HERDER, J. N., BOS, H., GRAS, B., HOMBURG, P., AND TANENBAUM, A. S. Construction of a highly dependable operating system. In *EDCC '06: Proc. of the Sixth European Dependable Computing Conf.* (2006), IEEE Computer Society, pp. 3–12.

[10] HITZ, D., LAU, J., AND MALCOLM, M. File system design for an nfs file server appliance. In *WTEC'94: Proc. of the USENIX Winter 1994 Tech. Conf.* (1994), USENIX Association, pp. 19–19.

[11] JOSEPHSON, W. K., BONGO, L. A., FLYNN, D., AND LI, K. Dfs: A file system for virtualized flash storage. In *FAST'10: Proc. of the Eighth USENIX Conf. on File and Storage Technologies* (2010), USENIX Association.

[12] JOUKOV, N., KRISHNAKUMAR, A. M., PATTI, C., RAI, A., SATNUR, S., TRAEGER, A., AND ZADOK, E. RAIF: Redundant Array of Independent Filesystems. In *Proc. of Twenty-Fourth IEEE Conf. on Mass Storage Systems and Technologies (MSST 2007)* (September 2007), IEEE, pp. 199–212.

[13] KRIOUKOV, A., BAIRAVASUNDARAM, L. N., GOODSON, G. R., SRINIVASAN, K., THELEN, R., ARPACI-DUSSEAU, A. C., AND ARPACI-DUSSEA, R. H. Parity lost and parity regained. In *FAST'08: Proc. of the Sixth USENIX Conf. on File and Storage Technologies* (2008), USENIX Association, pp. 1–15.

[14] PATTERSON, D. A., GIBSON, G., AND KATZ, R. H. A case for redundant arrays of inexpensive disks (raid). In *SIGMOD '88: Proc. of the 1988 ACM SIGMOD Intl. Conf. on Management of data* (1988), ACM, pp. 109–116.

[15] PRABHAKARAN, V., BAIRAVASUNDARAM, L. N., AGRAWAL, N., GUNAWI, H. S., ARPACI-DUSSEAU, A. C., AND ARPACI-DUSSEAU, R. H. Iron file systems. In *SOSP '05: Proc. of the twentieth ACM Symp. on Operating Systems Principles* (2005), ACM, pp. 206–220.

[16] SIVATHANU, G., WRIGHT, C. P., AND ZADOK, E. Ensuring data integrity in storage: techniques and applications. In *StorageSS '05: Proc. of the 2005 ACM workshop on Storage security and survivability* (2005), ACM, pp. 26–36.

[17] SIVATHANU, M., PRABHAKARAN, V., ARPACI-DUSSEAU, A. C., AND ARPACI-DUSSEAU, R. H. Improving storage system availability with d-graid. *Trans. Storage 1*, 2 (2005), 133–170.

[18] STEIN, C. A., HOWARD, J. H., AND SELTZER, M. I. Unifying file system protection. In *Proc. of the General Track: 2002 USENIX Ann. Tech. Conf.* (2001), USENIX Association, pp. 79–90.

[19] STEIN, L. Stupid file systems are better. In *HOTOS'05: Proc. of the Tenth Conf. on Hot Topics in Operating Systems* (2005), USENIX Association, pp. 5–5.

[20] TANENBAUM, A. S., AND WOODHULL, A. S. *Operating Systems Design and Implementation (Third Edition)*. Prentice Hall, 2006.

[21] TEIGLAND, D., AND MAUELSHAGEN, H. Volume managers in linux. In *USENIX Ann. Tech. Conf., FREENIX Track* (2001), pp. 185–197.