# Block-level RAID is dead

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

## Abstract

*The common storage stack as found in most operating systems has remained unchanged for several decades. In this stack, the RAID layer operates under the file system layer, at the block abstraction level. We argue that this arrangement of layers has fatal flaws. In this paper, we highlight its main problems, and present a new storage stack arrangement that solves these problems.*

## 1 Introduction

The concept of RAID [13] is a landmark in the history of storage systems. Taking advantage of the traditional block interface used by file systems, RAID algorithms were integrated at the block level, thus, retaining perfect backward compatibility with existing installations. As installations became larger, administrative tools like volume managers [20] followed suit. These tools broke the "one file system per disk" bond and made it possible to resize file systems on the fly. Volumes also served as a convenient point for policy assignment (choosing RAID levels for instance) and quota enforcement. Together, we refer to RAID and volume management solutions as the *RAID layer*.

The compatibility-driven integration of the RAID layer at the block-level has remained unchanged despite significant changes in the storage hardware landscape. We believe that it is time to retire block-level RAID. In this paper, we highlight several significant problems associated with the traditional block-level RAID implementation (Section 2). We briefly discuss proposed solutions and explain why they do not solve all the problems (Section 3). We then present Loris, a clean-slate design of the storage stack (Section 4), and highlight how it solves all the problems by design (Section 5).

## 2 Problems with block-level RAID

In this section, we will provide an in-depth look at some of the problems that plague block-level RAID implementations.

### 2.1 Silent data corruption

Modern disk drives exhibit a range of partial failures [14, 6], like lost, misdirected, and torn writes. In all these cases, the drive reports back a success, resulting in data being silently corrupted.

Various checksumming techniques have been developed to detect data corruption [16] and they offer varying levels of reliability. One technique that is capable of detecting all the aforementioned sources of corruption, involves storing the checksum of a block with its parent (the inode for instance). This has been referred to as *parental checksumming*. Since such a technique involves knowledge of block relationships, it can only be employed by file systems.

However, parental checksumming loses its benefit when used in combination with block level RAID. This is due to the fact that while file system-initiated reads undergo verification, RAID-initiated reads (a subtractive read to recompute parity for instance) are completely unverified. As a result, RAID can propagate data corruption, leading to data loss [12]. For instance, if a corrupt unverified data block is used for parity computation, the parity block becomes corrupted. Parental checksumming detects this problem on the next read request, but the data cannot be recovered.

### 2.2 System failure

Crashes and power failures in RAID systems result in the *consistent update* problem where two or more disks must be updated in a consistent manner. A window of vulnerability exists between a crash and complete recovery, during which a disk failure can result in data loss. This has also been referred to as the "write hole" [4]. While hardware RAID relies on NVRAM to solve this problem, crash recovery in software RAID systems is provided by either doing an expensive whole disk resynchronization or using journaling. Resynchronization adversely affects availability [7] and is impractical with disk sizes doubling every few years. Using journaling in block-level RAID on the other hand has a significant impact on performance, and results in functionality being replicated in both the file system and RAID layers [9].

### 2.3 Device failure

Research has revealed the benefits of a storage array that degrades gracefully [17] when unexpected failures occur. To achieve such a property, a storage array must

(1) replicate metadata selectively to make sure that the whole directory hierarchy remains navigable, (2) provide fault-isolated positioning of files such that a failure of any single disk does not render *all* files unavailable. An array that is block-liveliness aware can avoid restoring unused data blocks during recovery and thus, reduce the data loss window before another failure occurs. It is impossible to provide any of these functionalities in a traditional block-level RAID implementation, because it is unaware of both relationship between blocks and liveliness of blocks [5].

## 2.4 Administration nightmare

Block-level volume management is a tedious process involving a series of error-prone steps. A simple operation such as adding a new disk drive requires several steps like adding space to a volume group, expanding a logical volume, and then expanding a file system [20].

Software-based RAID solutions expose an array 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 [18]. This is an example of the more general "information gap" problem [8] – different layers in the storage stack performing their own optimizations oblivious to the effect that these optimizations might have when combined.

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. We argue that an ideal storage system should be flexible enough to support policy assignment on a per-file, per-directory, or a per-volume basis.

## 2.5 Heterogeneity issues

New devices are emerging with different data access granularities and new storage interfaces. Integrating these devices into the storage stack has been done in two ways. The first approach involves building file systems that are aware of device-specific abstractions [10]. However, the traditional block-based RAID layer is incompatible with devices that offer an abstraction different from the block abstraction (like byte-granular flash devices). As a result, device-specific file systems cannot be positioned on top of the traditional RAID layer.

The other approach involves adding a layer that translates block requests to match device-specific abstractions [10]. Such a translation layer could then be positioned below the RAID layer. Not only does this cause duplication of functionality, but it also widens the information gap. The only way to avoid this duplication is by redesigning the storage stack from scratch [10, 2].

# 3 Proposed solutions

There are several approaches to solving the problems mentioned in the previous section. We outline the most important ones here, namely, (1) inferring semantic information, (2) sharing semantic information, (3) merging file system and RAID layers, and (4) stackable filing. However, as we will show, none of these approaches tackle all the problems.

Internal information of one layer can be *inferred* by another layer. Semantically smart disks [5, 17] infer file system specific information to improve reliability and performance at the RAID level. This approach requires the RAID layer to know or guess the file system layout, making it nonportable and error-prone.

Instead of inferring information, one can share layer-internal information between these layers. For example, ExRAID [8] exposes size and performance characteristics about individual RAID disks to the file system. While sharing information can be used to make informed decisions [8], it does not provide a new division of labor among cooperating layers. As a result, the reliability and heterogeneity problems cannot be solved.

A few projects have refactored the traditional file system/RAID layering. For example, the ZFS [4] proposes a new stack made up of three layers. The lowest layer, called the Storage Pool Allocator (SPA), provides both block allocation and RAID support. However, because the SPA exposes a block interface, its algorithms suffer from the same heterogeneity problems as the traditional RAID layer.

RAIF [11] is a stackable file system that introduces RAID policies on a per-file basis, without changing the underlying file systems. While this provides more flexibility, it does not solve the reliability or heterogeneity problems.

# 4 The Loris storage stack

In this section, we discuss the design of Loris, our new storage stack. Loris is made up of four layers as shown in Figure 1. The interface of each layer to the layer above (its *client*) is a standardized file interface that supports operations such as *create*, *delete*, *read*, *write*, and *truncate*. In addition to file manipulation operations, the interface also has operations for setting and retrieving *attributes*. The use of attributes in Loris is to (1) share information between layers, and (2) store out-of-band data on a per-file basis, as will become clear when we describe the internals of each layer.

We now detail the division of labor between layers in a bottom-up fashion. Just like the network protocol stack, each layer in Loris has well-defined functionalities. Similar to networking protocols, different protocols can be used within each of these layers to provide dif-

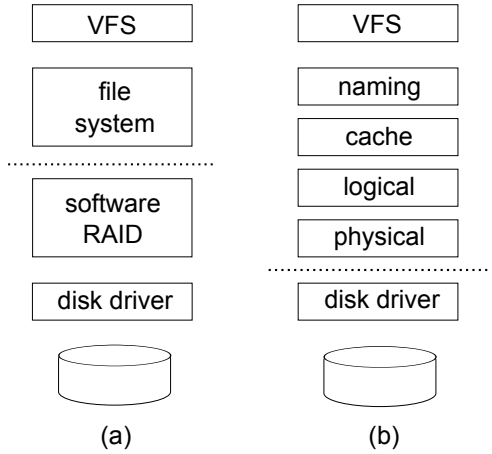


Figure 1: The figure depicts (a) the arrangement of layers in the traditional stack, and (b) the new layering in Loris. Only the layers above the dotted line are file-aware.

ferent functionalities. For each layer, we first provide an abstract description of its responsibilities. We then describe the protocol we implemented for that layer, in our prototype, to make things more concrete. Our prototype has been implemented on MINIX 3 [19]. Our protocols attempt to mirror the MINIX 3 file system's algorithms, so that we can make a fair evaluation of any overheads introduced by Loris.

## 4.1 Physical layer

The physical layer provides device specific layout schemes to store files and attributes, exposing a *physical file* abstraction to its clients. By working with physical files, the clients are abstracted away from device specific protocols employed by the physical layer. All physical layer protocols are also required to implement parental checksumming.

Our prototype physical layer protocol provides a modified MINIX 3 file system layout scheme for block devices. Each file is represented on disk by an *inode*. Like the traditional UNIX inode, our inode stores direct and indirect block pointers. Each block pointer is accompanied in the inode by a CRC32 checksum for the block it points to. We refer to this block pointer - checksum pair as a "safe block pointer." The indirect blocks similarly store sets of safe block pointers. In addition to these safe pointers, the inode has a fixed amount of space available for storing attribute data.

In order to make it possible to checksum metadata blocks, there are three special files on the disk: the inode bitmap file, the block bitmap file, and the "root file." The bitmap files contain checksums for the bitmap blocks. The root file forms the root of the parental hierarchy. Its data blocks contain pairs of inode numbers and corresponding checksums. Each inode on the disk is checksummed individually, in contrast to the bitmaps which are checksummed on a per-block basis. The root file's checksum is stored in the superblock. In order to prevent each individual block write from triggering cascading checksum updates all the way to the root, we use a small metadata cache to delay the checksum computation of metadata blocks until they are written out.

We run multiple instances of this layout protocol, one per disk drive, as separate processes, in our prototype stack. Each disk is assigned a global *device identifier* when it is added to Loris. Each layout instance registers itself to the logical layer using this identifier during system startup.

## 4.2 Logical layer

The logical layer combines multiple physical files to provide a virtualized, *logical file* abstraction. Clients of the logical layer perceive the logical file to be a single, flat file. The logical layer abstracts away the details of how many physical files there are, and where they are stored, which may differ on a per-file basis. Protocols in the logical layer provide the traditional RAID and volume management functionalities.

Our prototype logical layer protocol provides support for RAID levels 0, 1, 4, and 5. The central data structure in our protocol is the *mapping file*. The mapping file is array of entries, one per logical file. Each entry contains: (1) the *file identifier* used by clients to identify the file, (2) the RAID level for this file, (3) stripe size used for this file and (4) device identifiers - inode number pairs, to identify the physical files that make up this logical file. Thus, a crucial difference between our implementation and any traditional block-level RAID implementation is the fact that RAID levels are assigned on a per-file basis. We will describe how these entries are created when we describe the naming layer. Since the mapping file is a crucial piece of metadata, it is mirrored on all devices.

Once an entry has been made in the mapping, processing a request involves looking up the entry corresponding to the file identifier received, and forwarding the call to the right physical instances. Let us consider a read request for a logical file. Let us assume that the mapping entry for this file looks like <F, 0, 4096, PF1=<D1, I1>, PF2=<D2, I2>>. It maps file identifier F to RAID level 0, stripe size 4096, and two physical files, PF1, the file on device D1 with inode number I1, and PF2, the file on device D2 with inode number I2. When the logical layer receives a request to read, say, 40960 bytes, from offset 0, it determines that the logical byte ranges 0-4096, 8192-12288 and so on, map onto the byte range 0-20480 in physical file PF1 and the logical byte ranges 4096-8192, 12288-16384 map onto the byte range 0-20480 in physical file PF2. The logical layer now forwards the

read request to the physical instances that handle PF1 and PF2, for the aforementioned byte ranges, to satisfy the read request. Once the read results come back, they are combined into a properly-ordered flat result.

The logical layer also provides on-the-fly failure recovery on a per-file basis. For instance, if one of the physical instances responds back with a checksum error for a read request, the logical layer tries to recover the corrupt data, if possible, by recomputing valid data from redundant information, and restoring it onto the physical instance that failed.

### 4.3 Cache layer

The *cache layer* provide the in-core file abstraction to the naming layer. Cache layer protocols are responsible for providing file data caching in collaboration with the virtual memory subsystem. In systems that provide a unified page cache at the VFS level, the naming layer could communicate directly with the logical layer.

Our prototype uses a static file cache to buffer user data, as MINIX 3 does not have a unified page cache.

### 4.4 Naming layer

The naming layer provides support for naming, organizing and searching files. For example, a naming layer protocol could support POSIX style naming while another could support a more search-friendly, attribute-based naming.

Our prototype naming protocol provides a POSIX front-end. On file creation, the naming layer picks a file identifier and stores the name/identifier mapping as a directory entry. It is important to note here that directories are also stored as regular files–below the naming layer, there are no directories, only files. All POSIX attributes, like file mode and access time, are stored as Loris attributes. The naming layer uses the *setattribute* call to pass down the attribute data. The physical layer processes the *setattribute* call by storing the attribute data in the inode.

The naming layer also uses attributes to mirror directories on all disks for providing graceful degradation. When a directory is created, the naming layer sends a create call to the cache layer. In addition to passing the file identifier for this directory, the naming layer also passes the RAID level (RAID 1) as an attribute. The cache forwards the create call to the logical layer, which then uses the attribute to construct a new entry for this file in the mapping, and forwards the create call to all of the physical instances.

### 4.5 Legacy support in Loris

The new division of labor among layers in Loris makes it incompatible with the traditional file system design. The naming, cache and physical layers in Loris together perform the role of the traditional file system layer. The logical layer in Loris corresponds to the RAID layer in the traditional stack. This essentially means that one cannot integrate a traditional file system with the new stack. However, because Loris runs under VFS, one could run legacy file systems unmodified next to it.

## 5 Solving problems the Loris way

In this section, we describe how Loris solves the problems we mentioned in Section 2.

### 5.1 Data corruption

As a result of repositioning the RAID layer, the physical layer serves requests originating from the logical layer just like application-issued requests. Thus, a physical layer that implements parental checksumming acts as a single point of data verification. Therefore, by requiring all layout schemes in the physical layer to implement parental checksumming, we convert fail-partial failures [14] into fail-stop failures. The logical layer can then work on fail-stop devices to protect against data loss without worrying about spreading data corruption.

### 5.2 Crash recovery

While traditional crash recovery techniques guarantee that the storage array or the file system is brought back to a consistent state, none of them guarantee atomic updates of user data. Data journaling is one way to provide such atomicity but is generally considered too expensive since it involves writing all data twice. With a highly flexible policy selection in place, we have an infrastructure where users are free to specify policies like RAID level on a per-file basis. An approach to crash recovery that we are investigating makes use of this fact to provide "metadata replay."

Metadata replay is based on the counterintuitive idea that it is user data that must be protected and atomically updated, not the system metadata. To make this possible, we intend to (1) provide support for write-ahead logging for selective user data, and (2) add support for maintaining metadata consistency using a technique similar to doublefs [1]. With this infrastructure, when a crash occurs, the logical layer coordinates the recovery of all physical layers to the last globally consistent state. The write-ahead log can now be replayed to atomically update user data and, as a side effect, regenerate system metadata.

### 5.3 Graceful failure and file-aware rebuild

Our new design is a natural fit for implementing graceful degradation: (1) the availability of per-file policy selection makes it possible to provide selective metadata

replication, and (2) since the physical layers expose files rather than blocks, the logical layer can choose to store whole files in a single device, thereby providing fault-isolated positioning. In addition, since the RAID algorithms are file-aware, no unused data blocks need to be recovered thereby reducing the reconstruction time.

### 5.4 Simplified administration

By positioning RAID and associated functionalities in the logical layer, the new stack provides direct support for (1) Storage pool [4] style device management, and (2) AFS-style volume management [15].

When a new device is added to Loris, it will be assigned a new device identifier. A new physical instance will be started for this device and it registers itself with the logical layer as a source of physical files. From here on, the logical layer can use this physical instance to create new files or migrate existing files.

By using AFS-style volume management, volumes can be created on a per-user or per-project basis, for instance. These would serve as units of administration and quota enforcement. The logical layer would support operations for creating and deleting these volumes. All volumes would share the storage space, since files belonging to any volume can be located on any physical instance. We are working on implementing such a volume manager for Loris.

In addition to these advantages, the new stack provides a policy/mechanism split. While the mechanism to provide RAID algorithms is in the logical layer, the policy that assigns RAID levels to files can be enforced by any layer. For instance, the naming layer can specify the policy on a per-file basis similar to RAIF [11]. For instance, cherished photographs could be replicated on multiple devices while compiler-temporary files are not. Thus, policies can be applied at several granularities – per-file, per-directory, per-volume or even globally, across all files.

### 5.5 Embracing heterogeneity

By providing the traditional volume management techniques at the file level, Loris makes these functionalities device-independent. Thus, Loris can support heterogeneous installations where block-based disks coexist with byte-granular flash devices and object-granular OSDs [3]. Functionalities like RAID and snapshotting would apply across these device types without requiring any modification. In addition, as physical layer protocols work on devices directly, they can exploit device-specific properties to improve reliability and performance.

## 6 Conclusion

The integration of RAID algorithms at the block level has remained the same over several decades. We investigated the compatibility-driven block-level integration of RAID and presented several problems inherent to this integration. We proposed Loris, a clean-slate design of the storage stack that solves all these problems by design. The new layering in Loris opens up several interesting questions for future research.

## References


[1] Double the metadata, double the fun: A cow-like approach to file system consistency. `http://valerieaurora.org/review/doublefs.pdf`.

[2] Memory technology devices. `http://www.linux-mtd.infradead.org/`.

[3] Object-based storage device commands, ansi standard incits 400-2004.

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

[5] ARPACI-DUSSEAU, A. C., ARPACI-DUSSEAU, R. H., BAIRAVASUNDARAM, L. N., DENEHY, T. E., POPOVICI, F. I., PRABHAKARAN, V., AND SIVATHANU, M. Semantically-smart disk systems: past, present, and future. *SIGMETRICS Perform. Eval. Rev. 33*, 4 (2006), 29–35.

[6] BAIRAVASUNDARAM, L. N., GOODSON, G. R., SCHROEDER, B., ARPACI-DUSSEAU, A. C., AND ARPACI-DUSSEAU, R. H. An Analysis of Data Corruption in the Storage Stack. In *Proc. of the Sixth USENIX Conf. on File and Storage Technologies (FAST '08)* (February 2008).

[7] 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. Conference* (2000), pp. 263–276.

[8] 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.

[9] 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.

[10] 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.

[11] 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.

[12] 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.

[13] 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.

[14] 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.

[15] SIDEBOTHAM, B. Volumes: the andrew file system data structuring primitive. In *EUUG'86* (1986).

[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, 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.

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

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