Integrated System and Process Crash Recovery in the Loris Storage Stack

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

Abstract— **In this paper, we look at two important failure classes in the storage stack: system crashes, where the whole system shuts down unexpectedly, and process crashes, where a part of the storage stack software fails due to an implementation bug. We investigate these two problems in the context of the Loris storage stack. We show how restoring metadata consistency can provide a common first step for recovery from both types of crashes. In addition, we present fine-grained and corruption-resistant data resynchronization as the second step for system crash recovery, and an in-memory roll-forward log that can provide strong guarantees as the second step for process crash recovery in a microkernel setting. We implement our findings in our Loris prototype, and implement a new crash-resistant on-device layout as part of our proof of concept. The evaluation shows that our approach provides increased reliability at a reasonable performance cost.**

I. Introduction

In virtually any computer system, there is a component responsible for storing users' data: the storage stack. A large part of the storage stack is software (usually) in the operating system. While it is important that the storage stack has proper performance, we argue that reliability is at least as important. After all, not dealing with storage stack failures can translate directly into data loss. In this paper, we look at two threats: system crashes, and storage stack software (process) crashes.

A system crash is a whole-system failure of a machine. The causes of such failures include power outages, hardware failures, and operating system kernel crashes. In the event of a system crash, the storage stack does not get the opportunity to write out dirty data in memory, or even complete the current operation. This may result in inconsistent on-device data structures, which could lead to failure to reload these data structures from disk after the system has restarted. That in turn could lead to data loss.

Another major reliability threat comes from software bugs. Previous research has suggested that the number of bugs in software is roughly linear in its number of lines of code [12]. A full-fledged operating system storage stack can easily consist of hundreds of thousands of lines of code, and this makes the presence of many bugs highly probable. Any such bug has the potential to subvert the proper operation of the storage stack and, again, cause data loss.

In previous work, we have designed a new storage stack called Loris [2]. This storage stack has advantages in the areas of reliability, heterogeneity, and flexibility. In this paper, we investigate how to add support in Loris for system crash recovery, and for process crash recovery from transient failures in the lower layers of the stack, without compromising Loris' other reliability guarantees.

We show that the recovery procedure for both types of crashes require a single shared first step, namely restoring consistency of all metadata maintained internally by the storage stack. We argue that the storage stack should incorporate first-class support for this, and to this end we add the concept of global consistent *checkpoints* to Loris.

After this shared first step, each of the crash recovery procedures requires a different second step. For system crashes, recovery involves restoring proper redundancy of (user) data, using a resynchronization procedure similar to that of traditional software RAID. We present a data resynchronization approach that is both fine-grained and corruption-resistant. For process crashes, recovery involves an in-memory log to roll forward from the last checkpoint to the current state. We employ this approach in a microkernel environment to provide better recovery guarantees than any previous work.

We implement these ideas in our Loris prototype. As part of this, we present a new crash-resistant device layout and a corresponding software implementation called "TwinFS." We evaluate our work using performance benchmarks and reliability tests, aiming to prove that our design can be adopted in environments where a moderate performance overhead is acceptable, but high reliability is a requirement.

The rest of the paper is organized as follows. Sec. II describes the Loris storage stack that we developed previously. In Sec. III, we describe the two main problems to address, and we sketch an architecture that integrates a solution for both. In Sec. IV–VI, we present the design and implementation of the three parts that make up the solution. In Sec. VII, we evaluate the prototype. Sec. VIII covers related work. Sec. IX concludes and lists future work.

II. Background: the Loris storage stack

The traditional storage stack as found in most operating systems is shown in Fig. 1a. A Virtual File System (VFS) layer multiplexes application calls across file systems. File systems are generally designed to operate on one device, although a software RAID layer may transparently add storage redundancy using multiple devices below it. The actual devices are controlled by disk driver software.

The Loris stack was formed by first splitting the traditional file system into three layers (naming, cache, and layout), and

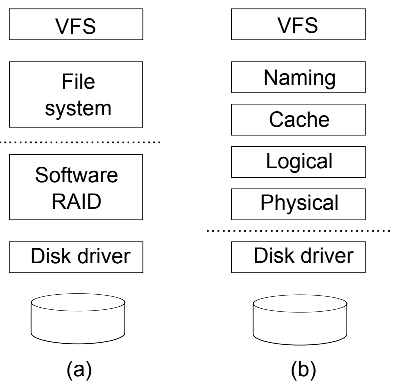

Fig. 1: The figure shows (a) the layers of the traditional stack, and (b) the new arrangement in Loris. The layers above the dotted line, and only those, are file-aware.

then swapping the layout layer (also called the *physical* layer) and the traditional software RAID layer (forming the *logical* layer). The VFS and disk driver layers are left unchanged. The result is depicted in Fig. 1b.

The Loris stack is completely file-oriented: the four layers communicate in terms of *files* only. Each file has a unique file identifier, and a small set of attributes associated with it. The layers use and implement the following operations: create, delete, read, write, truncate, getattr, setattr, and sync.

Compared to the traditional stack, the Loris stack has reliability, heterogeneity, and flexibility advantages [2]. We have built a Loris prototype on the MINIX 3 microkernel system [8], where all layers and file stores are separate user space processes. We will now describe the four layers.

A. Layers of the stack

At the bottom, the **physical layer** consists of one or more **file stores**. Each file store manages one underlying device, and maintains the layout on that device. It exposes an independent set of *physical files*, each with a physical file ID chosen by the file store. Each file store has a small local cache for the metadata specific to that file store, which we call "layout metadata." All file stores are required to implement *parental checksumming* in their layout [2]. As a result, they can reliably detect all whole-device failures as well as any form of (overt and silent) corruption.

Our prototype implements one file store called "PhysFS," based on the traditional UNIX file system. In PhysFS, the layout metadata structures form a virtual tree. All parents in this hierarchy point to their children by means of *safe block pointers*. A safe block pointer consists of the block number of the child block, and a checksum of its contents. File metadata, including attributes, are stored in *inodes*. Inodes use safe block pointers to point directly to data blocks, and to indirect blocks that contain (safe) pointers to either data blocks or other indirect blocks. Free inodes and blocks are tracked using inode and block *bitmaps*. The parental checksumming hierarchy is completed with three special inodes that point to the blocks of the inode area and bitmap areas. These inodes are stored in a *root block*, which is self-checksummed and forms the root of the metadata tree.

The inode, bitmap, and root block metadata areas are preallocated and statically sized. Out of all the layout metadata, only indirect blocks are allocated dynamically, and stored together with data blocks in the data area.

The **logical layer** implements a file-based version of RAID, providing the abstraction of *logical files*. Each logical file is made up of one or more physical files on different file stores. The logical layer multiplexes operations across the file stores in a RAID-like fashion. For example, a two-way mirrored logical file is stored as two identical physical files on different file stores (and thus devices). The logical layer keeps a *mapping*, which for each logical file specifies: the RAID level, the corresponding file stores and physical file IDs, and other RAID parameters such as the stripe size. The logical layer stores the mapping in a special file that is mirrored across all file stores. This file is part of the global metadata of the Loris stack, which we call "stack metadata."

The logical layer also implements RAID-like recovery mechanisms. If any of the file stores report a checksum error, recovery is attempted. In case of permanent failure, operations will continue to be served as long as enough redundancy is available. Redundancy guarantees follow the standard RAID failure model [14], although on a per-file basis.

The **cache layer** implements in-memory caching of logical file data. It uses a large amount of system memory for caching the contents of files.

The **naming layer** provides a POSIX abstraction by translating VFS operations to Loris operations. This layer implements directories, which are stored using Loris files. Lower layers are only aware that these directory files are part of the stack metadata. The naming layer uses Loris' file attributes to store POSIX attributes. It is responsible for picking logical file IDs for new files, and for tracking open deleted files.

III. THE CASE FOR INTEGRATED RECOVERY

In this section, we present the two challenges that we would like to address in the Loris storage stack: recovery from system crashes (Sec. III-A) and process crashes (Sec. III-B). We then show that we can exploit significant overlap between the solutions to both problems (Sec. III-C).

A. Recovering from system crashes

1) Metadata consistency: In the traditional storage stack, the file system typically implements a consistency scheme. Such a scheme returns the on-device structures to a consistent state after a system crash. Some limit themselves to metadata for performance reasons; others also cover the user data. Well-known schemes include journaling [6], logging [13], copy-on-write (CoW) [9], and soft updates [5].

All such schemes can be roughly described as periodically establishing consistent restore points, *checkpoints*, that can be reloaded such that any potentially inconsistent changes made after it are discarded upon system crash recovery. For example: copy-on-write schemes do this by writing a new root block; logging and journaling schemes do this by writing a commit record, and soft-update schemes effectively create a new checkpoint upon every metadata write.

In the Loris stack, each file store is free to implement a layout tailored to its underlying device. Layout metadata structures (such as inodes) are thus managed on a per-device basis. Consistency of these structures must therefore be managed on a per-device basis as well. Thus, each file store is necessarily responsible for managing its own *local checkpoints*.

However, it is not enough for each file store to restore its local layout metadata to just any consistent state after a system crash. The higher layers of the stack rely on the file stores in the physical layer to properly store stack metadata. For example, it is crucial for the stack that all mirrored copies of directories and the mapping are in sync and their contents are consistent with the file stores. Thus, the first step towards system crash recovery is restoring *global metadata consistency* across all the file stores, which includes both layout metadata and stack metadata.

There are two different approaches that we can adopt for this step. The first is to allow file stores to take local checkpoints whenever they choose, and then bring them back in sync at restore time. However, this approach imposes several requirements. For example, bringing the file stores back in sync can only be done if the file stores keep on-device logs that allow them to roll each other back or forward as appropriate. This rules out consistency schemes that do not keep such a log (e.g., copy-on-write). In addition, the file stores would have to become aware of consistency requirements for stack metadata updates, for example between file creates and directory writes, so as to create consistent local checkpoints. This effectively imposes stack-wide support for atomic transactions.

A better alternative is to globally coordinate the creation and reloading of checkpoints throughout the whole stack, and across all file stores. That means that all file stores were in sync at the time that the local checkpoints were taken, and they are thus again in sync if the same checkpoints are reloaded after a system restart. In Loris, we can extend the **sync** call to establish such *global checkpoints*. The whole stack is involved, so all layers get the chance to flush any pending stack metadata changes. The only downside is that the sync call must be a stack-wide barrier operation: while the checkpoint is being taken, any new state changes could subvert its consistency. Overall, this approach is preferable because it is simple to implement, and gives each file store a large freedom in choosing a local consistency scheme that is optimal for the underlying device.

2) Data resynchronization: The checkpointing system protects metadata, but for performance reasons it may not fully cover user data. A system crash may thus cause inconsistency between redundantly stored copies of the same data, due to a partially completed multidevice write operation. For example, a data write call to a mirrored file may make it to one mirror before a crash, but not to the other.

Traditional software RAID faces the same issue. For this reason, it typically implements *resynchronization*, whereby all blocks from the devices are read in and checked to find and fix any cross-device inconsistencies after a system crash. A similar resynchronization process is also necessary for user data in Loris. Thus, the second step towards system crash recovery is resynchronizing data.

However, traditional software RAID resynchronization suffers from two major problems. First, the RAID layer has no knowledge about the file system or even about liveness of blocks, and thus has to resort to scanning all devices in their entirety, including all metadata, data, and unused blocks. This may take a prohibitively long time–in the order of magnitude of hours to days. Second, the RAID layer cannot tell whether an inconsistency is the result of the system crash or data corruption. It may thus restore RAID consistency by overwriting a valid copy of data with a corrupted one, causing data loss. This is known as the RAID "write hole." As we show later, we can significantly improve the approaches to solving both problems in the Loris storage stack.

B. *Recovering from process crashes*

In a microkernel environment, most operating system components are implemented as user space processes. Each such system process has its own address space and restrictions on inter-process communication (IPC), and as such, is an individual failure domain. We use the term *process crash* to describe an observable failure in a system process. A process is said to "crash" when it performs an illegal CPU or memory operation, does not respond in time to periodic ping signals, performs disallowed IPC, or exits prematurely, for example due to a failing `assert`. If the cause of the crash is transient, repeating the operations leading up to the crash may not result in a crash the next time. We only consider transient crashes.

MINIX 3 provides detection of process crashes and supports basic recovery. When a process crashes, a fresh instance of the process is started. The internal state of the crashed process is lost. This approach works well for MINIX 3's device driver processes (including the disk drivers in the storage stack), which have little to no internal state [8]. However, other system components need to have their internal state fully restored before they can continue normal operation.

In this work, we focus on process crash recovery of the lower two layers of the Loris stack: the logical and physical layers. These layers form the largest part of the entire stack, and typically contain large amounts of in-memory state–mainly metadata structures that have been updated in memory as a result of application calls, but have not yet made it to disk. We discuss the other layers in Sec. IX.

Our goal here is twofold. First, we want to provide recovery that is fully transparent to applications. This means that system calls must not be aborted with an error due to a process crash, and that the effects of earlier system calls must never be lost. Second, we want to make no assumptions about what has happened *inside* the crashed process prior to the crash. Thus, recovering the state from the crashed process memory image is not an option, as this state may have been corrupted.

It is too costly to make a second in-memory copy of all process state and changes to it; it is even more costly to constantly flush the latest state to disk. However, it is possible to perform recovery using on-disk and in-memory state in combination. A large part of the required state is on the device at any time, and the necessary remaining part can be kept in memory until it is stored on disk. The recovery procedure then consists of first reloading a previous state from disk, and afterwards replaying from memory any state changes that have been made since.

C. Integrated recovery

We now show how both the recovery approaches can share the same first step. The checkpointing system that forms the basis for system crash recovery can be used to provide the first step toward process crash recovery as well. Thus, after a process crash, the recovery starts by rolling back the logical and physical layers to the latest checkpoint, discarding any state modified since then. The second step then consists of rolling forward these layers from the latest checkpoint to the current state.

In the next three sections, we present the design and implementation of this overall approach in Loris. We explain each of the three necessary parts: restoring global metadata consistency with checkpointing (Sec. IV), data resynchronization for system crash recovery (Sec. V), and in-memory roll-forward logging for process crash recovery (Sec. VI).

IV. Checkpointing

We will now describe the design and implementation of the checkpointing system in Loris. As outlined, this system establishes *global* checkpoints by coordinating the file stores' creation of *local* checkpoints. For this work, we have developed a new file store called "TwinFS," which implements a new on-device layout with support for local checkpoints. We start by describing this file store (Sec. IV-A). Then, we define the requirements for file store consistency schemes in general (Sec. IV-B). Finally, we describe the procedures for establishing and reloading global checkpoints (Sec. IV-C).

A. The TwinFS file store

TwinFS is based directly on our original PhysFS file store implementation as described in Sec. II-A. It adds the concept of checkpoints, by employing a copy-on-write-like scheme for the blocks that are part of those checkpoints. We call such blocks *protected*. At the very least, protected blocks are used to store all the stack and layout metadata.

Each protected block has two preallocated on-device locations ("twins"). At any time, one of these locations is used to store the "stable" version of the block: the block as it was at the time of the last checkpoint. The other is used to store the "unstable" version of the block: the most current version, which may be updated several times before the next checkpoint is taken. When a checkpoint is taken, the roles of all modified blocks are swapped: the unstable twin is marked stable, and the previously stable twin will be used to store subsequent block updates. Unmodified blocks are left untouched, so not all blocks switch twin sides between each subsequent checkpoints. An example is shown in Fig. 2.

In theory, each of the two twins could be located anywhere on the device. We simplify our implementation by hardcoding the block distance between each of the two twins to a static *twin offset*. This way, we can represent the *twin state* of each block using just two bits: one bit that identifies the last-modified twin (left or right), and one bit that identifies whether the block has been modified since the last checkpoint. The latter keeps the file store from having to track in memory which blocks have been modified since the last checkpoint (and thus have already switched sides).

The two-bit twin state of each block is stored in the *safe block pointer* in its parent. Marking a block as unstable thus implies recursively marking all parent blocks as unstable, all the way up to the root of the hierarchy. Exactly the same already happens in PhysFS due to updating the parental checksums. Hence, updating the blocks' twin state introduces no extra overhead. Note that unlike with true copy-on-write schemes, the preallocation of the twins guarantees that no new blocks need to be allocated in this process.

The overall TwinFS on-device layout is shown in Fig. 3. All layout metadata blocks are protected using the "twinning" scheme. In the statically allocated metadata areas, this results in a repeated pattern of N left twins being followed by N right twins, were N is the twin offset. These metadata areas are doubled in size as a result. Indirect blocks in the data area are twinned as well. TwinFS can also protect the data blocks of selected files. First and foremost, this data block protection is applied to the stack metadata files (logical mapping; directory files), since these must be included in the checkpoints as per Sec. IV-B. As stated before, file data of important user files can also be protected with this approach; however, it would be costly (in performance and space usage) to protect *all* file data this way. Thus, in the data area, the twin pairs of indirect blocks and protected data blocks are interspersed with unprotected data blocks. When a protected block is created in the data area, two physical disk block locations at a fixed

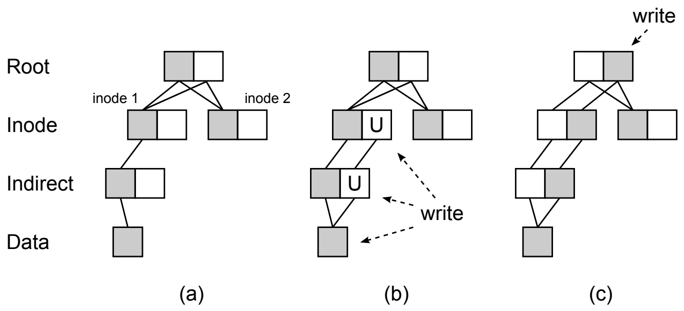

Fig. 2: TwinFS hierarchy example. Each block's left and right twins are shown; each latest stable twin is grayed. The lines represent safe block pointers. As simplification, each inode block contains only one inode. In (a), inode 1 has one indirect block pointing to a data block. In (b), this data block is overwritten in-place, and its ancestors are updated with new checksums by writing to the unstable (U) twins–up to but excluding the root block. In (c), a new checkpoint is established by writing a new root block.

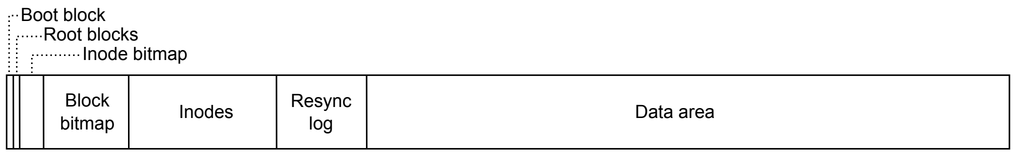

Fig. 3: High-level overview of the TwinFS on-device layout. The root, inode, and bitmap blocks are metadata and thus protected. The data area contains both protected and unprotected data blocks, as well as indirect blocks, which are metadata and thus protected. The resynchronization log is described in Sec. V-C.

separation must be allocated.

The root block of the layout scheme is "twinned" as well. Writing out a new root block equals taking a new checkpoint, and this is done only as part of a **sync** call. The root block is self-checksummed and contains a timestamp. A write cache flush is performed on the underlying device both before and after writing out the root block. Since the twin state of the root block cannot be stored anywhere, TwinFS has to read in and verify the checksum of both root block twins at startup.

B. General consistency scheme requirements

During recovery, all file stores must agree on the checkpoint to reload. For some file stores, this may be the penultimate checkpoint they have taken: it is possible that a system or process failure occurs during the checkpointing operation, whereby some file stores have finished establishing the checkpoint, and others have not. This results in the following requirements for the consistency schemes employed by the file stores: 1) it must always be possible to recover all (layout and stack) metadata to the state in the last checkpoint; and, 2) right after taking one checkpoint locally, but before this checkpoint has been finished across all file stores, it must remain possible to restore the *previous* checkpoint as well.

TwinFS meets these requirements. At any time, one of the root block twins identifies a stable checkpoint, since all of the metadata locations it refers to are stable and thus are left intact at least until after taking the next checkpoint. Immediately after taking a new checkpoint, both root block twins identify usable checkpoints, and either can be reloaded. Once new "unstable" data has been written out, only the latest checkpoint can be reloaded.

Even though we use only TwinFS in our prototype, each file store has the freedom to employ *any* consistency scheme that satisfies the stated requirements. This includes several of the more well-known consistency schemes:

- Copy-on-write consistency schemes effectively implement checkpoints by persisting new root nodes of the metadata tree. A minimum of two root nodes is enough.
- For journaling schemes, each *transaction* has to span between two checkpoints, and taking a checkpoint amounts to committing the current transaction. Copying from the journal to the original location may commence only once taking the checkpoint has finished globally.
- Logging schemes could be used as is, as long as the recovery procedure restores the latest checkpoint without performing any roll-forward on metadata.
- However, soft update schemes rely on frequently updating metadata in-place. This makes rollback impossible. Any such scheme is not suitable.

For consistency schemes that can overwrite data blocks in-place, such as some forms of journaling, there is an additional requirement: a data block must never be overwritten with contents of one file, if according to the last checkpoint this block was assigned to another file. If this were allowed, the new data could be read from the old file after a checkpoint restore, which could constitute a security violation. While parental checksumming helps to protect against this case, it does not provide a secure solution. In TwinFS, blocks that are freed are not reused until a new checkpoint is taken. This equally applies to protected and unprotected blocks.

C. Taking and reloading checkpoints

New global checkpoints are established using the Loris **sync** operation, which travels from the naming layer down the stack. This call causes all layers to flush pending data and stack metadata updates, and tells the file stores to create a new checkpoint. Sync calls are initiated upon application request (with POSIX' *sync* or *fsync*), periodically, and at system shutdown. During the sync operation, other state-changing Loris operations are deferred.

Upon startup, the logical layer queries all file stores for valid checkpoints. In response to this, all file stores return the timestamps of their valid checkpoints. If the system shut down cleanly, or crashed while no sync call was ongoing, all file stores will share the same latest checkpoint timestamp. If the system crashed during a sync call, not all file stores may have the latest checkpoint, but they will all have the penultimate one. After all file stores have reported their available checkpoint timestamps, the logical layer instructs them to load the most recent common checkpoint.

V. Data resynchronization

Some consistency schemes include *all* data in the checkpoints. Examples would be a pure copy-on-write or log-structured layout. Journaling and twinning layouts may or may not. For those that do not, data resynchronization may be needed to restore full consistency after a system crash.

In Sec. III-A.2, we listed two problems in traditional RAID resynchronization. We now show that the Loris stack offers the opportunity to improve on both problems: the large area to scan (Sec. V-A) and the possibility of corruption (Sec. V-B). We then describe how TwinFS implements support for data resynchronization (Sec. V-C), and we outline the full resynchronization procedure (Sec. V-D).

A. Limiting the areas to scan

In the Loris stack, we can very narrowly define the areas to which data resynchronization should be applied. First of all, since the file stores are completely file-based, unused parts of the disk are inherently excluded from resynchronization. Furthermore, resynchronization only involves data files. The layout metadata and the stack metadata files are already covered by the checkpointing system.

Second, Loris' per-file policies allow certain files to be stored with more redundancy than others. Files that are stored on one device only, need not be resynchronized. Moreover, the per-file policy system allows the user to assign a level of *importance* to a file. Files that are deemed especially important by the user, can be included in the checkpoints. This excludes them from resynchronization.

Finally, resynchronization is needed only for data blocks that have been overwritten in-place since the last checkpoint. After all, not-in-place updates are simply discarded when restoring a checkpoint. Since the security requirement from Sec. IV-B imposes that no file data can be overwritten with another file's data, resynchronization can be limited to in-place overwrites within the *same* file.

B. Verifying data

If a data block is overwritten in-place, and the last checkpoint is restored afterwards, the parental checksum of the block will no longer match. The file store can then no longer discern whether the block was merely overwritten, or has been subject of corruption. Thus, it has to either discard the data block, resulting in data loss, or ignore the checksum, risking to pass corrupted data to the application. For a reliable stack, neither is acceptable.

The file store must therefore make sure that when a data block is overwritten in-place, its new parental checksum has already been written to disk. The file store thus has to persist this information outside the checkpoints. This can be done by means of a log of "checksum records." Depending on the consistency scheme, this may be a dedicated resynchronization log, or be integrated in the main journal or main log. Before a data block is overwritten, the file store persists a record that contains the new checksum for the block. After restoring the last checkpoint, the file store can scan the log for these records.

The checksum records have two purposes. First, it allows the file store to tell whether the contents of an overwritten data block are valid or corrupted. The block contents are considered valid if the block checksum matches any one of the recorded checksums. After all, a system failure may occur between writing the record and writing the data block, in which case an earlier checksum is still valid. This checksum is possibly in an earlier checksum record and otherwise in the last checkpointed copy of the inode. Note that more than two checksums may have to be tested: checksum records may be generated for intermediate in-memory block updates that are never actually written to the device.[1]

[1]This could be avoided with a counter in the checksum record that obsoletes that many previous checksums for the block. We did not implement this.

Second, it allows the file store to tell exactly which blocks have been overwritten and thus should be subject to resynchronization. This limits resynchronization exactly to the areas defined in the previous section. However, even though files that are stored without redundancy also require no resynchronization, checksum records also have to be generated for those files, because the corruption concern applies equally to them.

If a system crash resulted in a torn write of a data block, none of the checksums will match. This case cannot be distinguished from other forms of corruption, and the data block will be lost. Given sufficient redundancy, it can be restored from other file stores. In the worst case however, every device suffers from a torn write. By including their file data in the checkpoints, stack metadata files and important user data files are protected from this problem.

C. The TwinFS resynchronization log

Since TwinFS does not include all data blocks in its checkpoints, it may end up overwriting unprotected data blocks in-place. Thus, TwinFS must perform data resynchronization. To this end, we add a dedicated resynchronization log to it, which uses a reserved device area to store checksum records.

Each checksum record contains a physical file ID, a file block offset, and the new block checksum. Pending checksum records are aggregated into "log blocks," which are self-checksummed and contain the corresponding checkpoint timestamp. In order to let file stores generate and aggregate checksum records ahead of the actual write operations, we add a **prewrite** Loris call. This call is sent down from the cache to the physical layer when an application performs a write call, providing an early copy of the data to the file stores. Whenever TwinFS receives a write operation that overwrites unprotected blocks, it ensures that the corresponding log blocks have been flushed to the device first.

After a checkpoint has been reloaded, TwinFS goes through the log area, and processes log blocks that have both a valid checksum and a matching checkpoint timestamp. It performs two actions on the records in each valid log block. First, it compares the checksum in the record to the computed checksum of the corresponding data block. If those match, it updates the data block's safe block pointer (in the file inode or an indirect block) with this checksum, effectively marking the data block as not corrupted. Second, it reports the file byte range from the record to the logical layer, for the purpose of resynchronization across file stores.

D. Resynchronization procedure

After the file stores have checked local data checksums, the next step is resynchronization *across* file stores. After all, the redundantly stored copies may still be out of sync.

The resynchronization procedure is performed by the logical layer as part of system crash recovery, right after restoring the appropriate checkpoint. The logical layer queries all file stores for a list of physical file IDs and byte ranges that are to be resynchronized. In the worst case, resynchronization involves all data from all unprotected, redundantly stored files; in the common case, only a few files and blocks will be involved. File stores that include all data in their checkpoints always report an empty set.

The logical layer then maps each reported (physical) file ID to a logical file ID. We do this by storing the logical file ID as an attribute of each physical file. The logical layer reads in the reported byte ranges from all file stores involved in storing this logical file, and optionally writes back resynchronized data to some of them. In the case of (RAID1-like) mirroring, all mirrors are synchronized to the contents from the first file store that does not report a checksum error. In the case of (RAID4/5-like) parity-striping, the parity is recomputed, unless one of the file stores reports a checksum error. In that case, that file store's contents are recomputed. If more checksum errors are reported than supported by the RAID failure model, the affected byte ranges are marked as bad, and will result in an error being returned to the application when being read later.

By disregarding data copies with checksum errors before performing resynchronization across file stores, we solve the write hole problem. We do not offer guarantees about which valid data copy is restored, however. Content-level data consistency is thus left to applications, as with standard RAID.

VI. In-memory roll-forward logging

The process crash recovery procedure consists of two steps: first restoring the last checkpoint, and then rolling forward the lower layers by replaying operations. The cache layer, which we currently assume to be a stable point in our stack, performs the second step, by keeping an in-memory log of operations. We describe how this log interacts with the checkpointing system (Sec. VI-A), the operation of logging and replay (Sec. VI-B), and the resulting assumptions and guarantees for process crash recovery (Sec. VI-C).

A. Interaction with checkpointing

When any of the processes in the lower two layers crashes, we choose to restart *all* of them, including all file stores. This has two advantages. First, this saves us from adding extra custom recovery code in those processes. Upon their restart, they will simply cooperate in restoring the latest checkpoint as part of their normal startup procedure. Second, there are no corner cases to handle when multiple processes crash at once.

The logical layer can detect when any file store or itself has restarted. First, when a Loris process is started, it will report its presence to the next layer up the stack. An unexpected presence notification from a file store thus indicates that a file store has restarted. Second, when a Minix 3 system process crashes, it is restarted with a flag indicating that it crashed. If the logical layer itself restarts, this flag will be set.

Upon detecting a file store crash, the logical layer commits suicide (gracefully), forcing itself to restart. When it comes back up after a restart, the restart flag will be set; this may be the a result of either a local crash or committing suicide. The logical layer then kills and thus restarts all file stores. The result is that regardless of where a crash (or multiple concurrent crashes) happened, both the logical and physical layers will end up being restarted with a fresh state.

After that, the logical layer will perform the standard checkpoint reloading procedure as part of its startup procedure. However, when the logical layer's restart flag is set, it skips the unnecessary data resynchronization phase: any in-file overwrites that took place after taking the latest checkpoint but before the crash, will be performed again.

B. Logging and replay

At all times, the cache keeps an in-memory log of operations performed after taking the last checkpoint. This log is cleared upon each successful sync call. All Loris operations that modify state in the lower layers are stored in the log: create, delete, write, truncate, and setattr. Since the cache uses pages as the smallest unit of storage, the new version of an entire page is stored as part of the log entry for a Loris write operation. Multiple writes to the same page are merged, and pages are removed from the log as appropriate upon Loris truncate and delete calls. As long as a page is in the main cache, the log only keeps a pointer to it; a copy is made for the log when the page is evicted from the main cache. If the total size of the log exceeds a configurable threshold, the cache makes an upcall to the naming layer to trigger an early sync. Note that the naming layer's subsequent flush will first cause the log to expand further (but see Sec. IX on future work).

After startup, the logical layer always announces its presence to the cache layer. The cache layer can tell from an unexpected presence announcement that the logical layer has restarted. The cache layer then cancels all ongoing downcalls, replays the in-memory log by issuing all operations in the log in sequence, and, upon success, restarts the previously ongoing calls. None of this is exposed to applications in any way. If the replay procedure fails, it is retried for a predefined number of times. Upon consistent failure, application-transparent recovery becomes impossible, and the entire stack is shut down to prevent further data loss.

There is one exception. It may happen that the system crashes before a sync call completes, but after all file stores have established a new checkpoint. The newer checkpoint would then be reloaded, causing unexpected failures during replay. For this reason, the logical layer informs the cache layer about checkpoint timestamps, and upon a mismatch after a crash, the cache clears its log instead of replaying it.

C. Assumptions and guarantees

Due to the process isolation offered by the microkernel environment, the only way in which failures can propagate, is through inter-process communication between processes. Moreover, since we completely restart the logical and physical layers, we throw out all of their internal state, including any state that has been corrupted as part of the failure.

As a result, we make only two assumptions about the behavior of any failing process: 1) no "bad" (corrupted) results are passed up to the cache; 2) no bad (meta)data may be written to the devices, unless they are discarded again once a

Benchmark	PhysFS	PhysFS+df	TwinFS-0	TwinFS-4	TwinFS-8	TwinFS-16	TwinFS-32
PostMark (transaction time, sec)	1097	1086	1101	1192	1158	1144	1150
FileBench File Server (ops/sec)	349.53	350.25	372.18	343.78	344.16	344.16	343.47
FileBench Web Server (Zipf) (ops/sec)	549.06	548.01	571.78	535.94	536.04	540.82	543.57
OpenSSH build (sec)	618.78	620.20	629.23	630.96	630.48	630.41	631.88

TABLE I: Transaction time in seconds for PostMark (lower is better), operations per second for File Server and Web Server (higher is better), and wall clock time for OpenSSH build (lower is better). Performance is shown for PhysFS, PhysFS with delayed freeing, and TwinFS with various twin block offsets.

checkpoint is restored. The second point implies that writing out corrupted unstable blocks is allowed in the failure model, as long as a crash happens before these blocks are made stable. This facilitates performing internal integrity checks before taking a new checkpoint.

As long as the two assumptions are not violated, this approach guarantees proper recovery from *any* bad behavior in the lower layers. This includes: arbitrary memory overwrites (wild writes), including heap and stack corruption; arbitrary function calls; infinite loops; and, arbitrary allocation of resources available to the processes, including memory.

VII. Evaluation

We now present a performance and reliability evaluation of our prototype. All of the following experiments were conducted on an Intel Core2Duo E8600 PC, with 4GB of RAM, and two 500GB 7200RPM Western Digital Caviar Blue (WD5000AKS) SATA hard drives for testing purposes. The tests were run on the first 8GB of the disks. All benchmarks were run on Minix 3. In order to stress the lower layers, the cache layer was given a small (64MB) buffer cache. A sync call is made once every five seconds.

A. Performance evaluation

For performance evaluation, we used these macrobenchmarks: PostMark, altered to perform a sync call before the transactions phases, configured to perform 80,000 transactions on 40,000 files in 10 directories, with file sizes between 4KB and 28KB, using 4KB I/O operations; FileBench File Server, altered to use the same randomly chosen random seed for each round of experiments, configured with 10,000 files at an average of 20 files per directory; FileBench Web Server, altered to pick files using a Zipf distribution pattern in order to introduce locality, configured with 25,000 files with an average of 20 per directory; and finally, an OpenSSH build test which unpacks, configures and builds OpenSSH.

1) TwinFS: We started by evaluating the performance of TwinFS, on a single disk, initially without a resynchronization log. Application data was unprotected (not twinned). We compared various TwinFS configurations to the original PhysFS file store implementation. We did not succeed in time to get a journaling file store to perform well enough for a head-on comparison to another crash-consistent layout.

In early experiments, TwinFS kept outperforming PhysFS. This turned out to be due to TwinFS' delayed block freeing, which resulted in more favorable block allocation patterns. We modified PhysFS to perform the same delayed freeing; we refer to this version as "PhysFS+df."

For TwinFS, we varied the hardcoded offset between the left and right twin of all pairs. A twin offset of 1 block means the twins are adjacent on the device. We show the results for offsets of 4, 8, 16, and 32 blocks; other twin offsets did not result in overall more favorable results. For comparison purposes, we also tested a twin offset of 0, causing all blocks to be updated in-place. This effectively reduces TwinFS to PhysFS+df, with one major difference: TwinFS issues device write cache flushes when writing the root block.

Table I shows the median performance result out of five runs for each configuration and benchmark. For all tests, the cache size was too small to contain the working set. This resulted in long run times and a significant amount of I/O. Surprisingly, in some tests, TwinFS-0 performed better than PhysFS+df. We confirmed that this is due entirely to the added write cache flush call–an oddity of the disk used.

Compared to the best of the crash-unsafe alternatives, TwinFS with a twin offset of 16 blocks yielded the overall best performance in our tests (with an overhead of 2–8%), although only by small margins compared to the other twin offsets. Microbenchmarks showed that performance degrades when otherwise contiguous metadata blocks use a mix of left and right twins, destroying contiguity. When this is not the case, performance goes up with larger twin offsets, because in that case the stretches of contiguous blocks are larger as well.

2) Data resynchronization: We used the same single-disk configuration and the same set of benchmarks to evaluate the overhead of the TwinFS resynchronization log. For each benchmark, we also measured the maximum amount of data that would have to be resynchronized after a system crash, if all the files were stored with redundancy.

The resulting performance numbers are shown in Table. II. The extra run-time overhead was negligible. Also, the worst-case amount of data to resynchronize in case of a crash is very small in all tests–we confirmed that the resulting resync times are negligible (sub-second) even if all files were mirrored on two disks. Finally (not shown), none of the benchmarks ended up writing more than a handful of resync log blocks between any two checkpoints, suggesting that TwinFS requires only a small log area. Note that in our benchmarks, most in-place overwrites were the result of appending data to log files. Web Server only appends block-aligned chunks to its log, resulting in almost no data being overwritten.

Overall, the **prewrite** solution allowed for proper aggregation of checksum records. However, it proved not to be ideal, since it complicates parity precomputation for parity-striped files. We now believe that a better solution is to let the file stores cache small numbers of data blocks for short times.

Benchmark	T-4	T-8	T-16	T-32	Peak
PostMark	1197	1161	1146	1153	1663 KB
File Server	342.61	345.74	345.83	342.04	1327 KB
Web Server	530.25	536.86	546.52	539.03	4 KB
OpenSSH	632.23	632.53	631.90	632.33	1266 KB

TABLE II: The same benchmarks, now for TwinFS with the resync log enabled. Also shown is the worst-case size of the user data to resynchronize if all data were stored with redundancy.

Benchmark	No cache log	With cache log	Memory usage
PostMark	1132	1150	170 MB
File Server	337.79	337.70	192 MB
Web Server	531.85	540.60	112 MB
OpenSSH	635.05	634.28	230 MB

TABLE III: The same benchmarks, now with two TwinFS-16 instances (with resync log) and all files mirrored across these two instances. Also shown is the peak memory usage for the cache log.

3) The cache log: Next, we tested the overhead of the cache log, both in performance overhead and in resource usage. We configured Loris to mirror all files on two disks, using TwinFS with a twin offset of 16 blocks (and, although unused, a resynchronization log) on both mirrors. We did not bound the cache log size; rather, we measured the maximum amount of memory needed for the log between any two checkpoints. To keep the comparison fair, we do not let the cache save on I/O by reusing (meta)data stored in the in-memory log if it already evicted the primary copy; otherwise, this would speed up the performance due to a bigger effective total cache size. Table III shows the results.

As shown, the mirroring case added little overhead to the single-disk case. The cache log added almost no visible overhead on top of that. The memory usage for the log was significant, but this is expected to get relatively smaller with larger cache sizes, since more data pages will be shared between the main cache and the log in that case.

B. Reliability evaluation

For the reliability evaluation, we used the same configuration as for the cache log test.

1) Kill tests: We tested both the TwinFS checkpointing system and the process crash recovery procedure at once, by killing the processes in the lower two Loris layers. Each kill was performed by injecting a trap instruction at the process program counter. We repeatedly ran the OpenSSH benchmark, killing one of the processes at random intervals–once every twenty seconds on average.

We performed 10,701 kills this way. In all cases, the crash was hidden completely from the application layer, and the benchmark completed successfully. The median cache log replay time was 0.22 seconds; the maximum was 6.5 seconds. This depended only on the size of the log. However, the logical and physical layers have to refill their local caches from disk after each restart; this caused an overall benchmark performance degradation of up to 44%. Of course, we do not expect crashes to occur this often in practice. In 85 cases, the cache refrained from replaying its log due to a crash at the very end of a sync. In 88 cases, a TwinFS instance had to reload its penultimate checkpoint.

2) Targeted fault injection: In addition, we manually injected a number of less-trivial process faults, based on bugs we experienced in practice while developing Loris.

Stack overrun: In early Loris versions, thread stacks had no guard pages. If the stack for one thread's execution path was too small, part of the next thread's stack would be overwritten. That would cause a crash in the next thread, but only if that thread was active. This bug occasionally triggered in TwinFS with deep recursive parent block updates.

Heap corruption: In one case, a static array in the logical layer was too small for the maximum amount of data stored in it, and this would sometimes result in other variables on the heap to be overwritten. An assert would then go off when one of the corrupted fields was used afterwards.

Deadlock: A typical source of transient failures is threads contending for shared resources. Any programming errors in mutual exclusion code can cause race conditions. During TwinFS development, we ran into a case where multiple threads would concurrently try to acquire a large number of data buffers from a shared pool for a write operation, resulting in buffer exhaustion and a deadlock between the threads. This would trigger call timeouts after a while.

We simulated these and 12 other comparable bugs in the latest Loris version. Each bug eventually triggered, and caused a process crash. In all cases, the recovery mechanism performed successful recovery, and the system kept running.

VIII. Related work

We list the most directly related work on system and process crash recovery.

A. System crash recovery

TwinFS can be described as a hierarchical version of doublefs [7], or a selective copy-on-write file store where each protected block has two preallocated copies. As such, this layout shares several advantages with copy-on-write file systems: no need to write out metadata updates more than once, and no metadata recovery code. At the same time, TwinFS does not share some of their disadvantages: high fragmentation, cascaded metadata allocation, and difficulty to track free space. Finally, the implementation is fairly simple. It does however require more on-device space. Compared to doublefs, TwinFS uses the hierarchy to determine which block copy to use, eliminating the necessity to read in both copies on every read. Intra-device redundancy, like Stable Storage [11], could be added as an orthogonal feature.

We are not aware of work that involves global consistency across arbitrary local heterogeneous file/object stores. A similar problem can be found in fan-out stackable file systems that require their own metadata storage. An example is RAIF [10], which does not fully address this problem.

Several solutions have been proposed to limit the areas of resynchronization in traditional software RAID. Most comparable to our work is journal-guided resynchronization [4], which proposes extending existing file system journals with records for data resynchronization. Our checksum records add checksumming to this, allowing the resynchronization procedure to determine not only what to resynchronize, but also which of the redundantly stored copies (not) to use. Our checksum log is similar to the hash log used for general data integrity, but not resynchronization, in [16].

The write hole can be eliminated by never overwriting data, thus avoiding the need for resynchronization altogether. This approach is employed by for example ZFS [1].

B. Process crash recovery

Little research has focused on storage stack reliability in microkernel environments. Studies that do (e.g., [3], [15]), depart much more radically from the traditional storage stack model, leaving open the question whether a reasonably efficient POSIX-compliant system could be built on top.

Membrane [17] uses checkpoints and in-memory logging to recover from crashes in existing file systems on Linux. We use the same basic approach, but can offer stronger guarantees because of the microkernel-provided process isolation, allowing recovery from a broader class of failures. This does come at an extra performance cost; ongoing research aims to reduce that cost significantly by exploiting multicore architectures. Compared to Membrane, we protect the rather complex equivalent of the RAID layer. On the other hand, Membrane protects the equivalent of our naming layer. By leaving out the naming layer in this work, we can make fewer assumptions about the implementation of the layers we do protect. For example, the file stores are not required to generate the same physical file ID upon retried create operations, and open deleted files are not a special case.

Toward the other end of the spectrum, Re-FUSE [18] can recover from crashes in a more diverse set of file systems, at the cost of making more assumptions about their behavior.

IX. Conclusion and future work

In this paper, we have made a case for integrated support for recovery from system and process crashes in the Loris storage stack, using a single shared base: restoring global metadata consistency by means of checkpointing. On top of this, we have presented a fast and reliable approach to data resynchronization for recovery from system crashes, and recovery from transient process crashes in the lower two layers of the stack with relatively few assumptions. Our proof-of-concept implementation shows that these additions increase the overall reliability of our prototype, at a reasonable performance cost.

Dealing with process crashes in other layers in the stack is part of future work, as sketched in [19]. For the naming layer, this involves making sure that every VFS call immediately flushes its state changes down to the cache. As a result, the cache will always be consistent with respect to the naming layer's stack metadata. That in turn means the cache can perform a sync operation at any time, allowing the cache log's memory size threshold to be strictly enforced (see Sec. VI-B). The cache layer itself is deemed a stable point in the stack; still, we are working on a new crash recovery technique for the cache, even though we have to make stronger assumptions about the failures that can occur.

One unsolved problem in this work is efficient support for *fsync* and transactions. We intend to investigate the implications of adding support for those. This will likely result in exploring the other option laid out in Sec. III-A.1.

Acknowledgment

This research was supported in part by European Research Council Advanced Grant 227874.

References

[1] Sun Microsystems, Solaris ZFS file storage solution. Solaris 10 Data Sheets, 2004.

[2] R. Appuswamy, D. C. van Moolenbroek, and A. S. Tanenbaum. Loris - A Dependable, Modular File-Based Storage Stack. In *Pacific Rim International Symposium on Dependable Computing*, PRDC'10, pages 165–174, 2010.

[3] F. M. David, E. M. Chan, J. C. Carlyle, and R. H. Campbell. CuriOS: Improving Reliability through Operating System Structure. In *USENIX Symposium on Operating Systems Design and Implementation*, OSDI'08, pages 59–72, 2008.

[4] T. E. Denehy, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Journal-guided resynchronization for software RAID. In *Proc. of the Fourth USENIX Conf. on File and Storage Technologies*, FAST'05, pages 7–7, 2005.

[5] G. R. Ganger and Y. N. Patt. Metadata update performance in file systems. In *Proc. of the First USENIX Conf. on Operating Systems Design and Implementation*, OSDI'94, page 5, 1994.

[6] R. Hagmann. Reimplementing the Cedar file system using logging and group commit. In *Proc. of the Eleventh ACM Symp. on Operating Systems Principles*, SOSP'87, 1987.

[7] V. Henson and T. Ts'o. Double the Metadata, Double the Fun: A COW-like Approach to File System Consistency. `http://valerieaurora.org/review/doublefs.pdf`.

[8] J. N. Herder, H. Bos, B. Gras, P. Homburg, and A. S. Tanenbaum. Construction of a Highly Dependable Operating System. In *Proc. of the Sixth European Dependable Computing Conference*, EDCC'06, pages 3–12, 2006.

[9] D. Hitz, J. Lau, and M. Malcolm. File system design for an NFS file server appliance. In *Proc. of the USENIX Winter 1994 Tech. Conf.*, 1994.

[10] N. Joukov, A. M. Krishnakumar, C. Patti, A. Rai, S. Satnur, A. Traeger, and E. Zadok. RAIF: Redundant Array of Independent Filesystems. In *Proc. of Twenty-Fourth IEEE Conf. on Mass Storage Systems and Technologies (MSST 2007)*, pages 199–212, September 2007.

[11] B. W. Lampson. Atomic Transactions. In *Distributed Systems - Architecture and Implementation, An Advanced Course*, pages 246–265, 1980.

[12] T. J. Ostrand and E. J. Weyuker. The distribution of faults in a large industrial software system. In *Proc. of the 2002 ACM SIGSOFT int. symp. on Software testing and analysis*, ISSTA '02, pages 55–64, 2002.

[13] J. Ousterhout and F. Douglis. Beating the I/O bottleneck: a case for log-structured file systems. *SIGOPS Oper. Syst. Rev.*, 23:11–28, 1989.

[14] D. A. Patterson, G. Gibson, and R. H. Katz. A case for redundant arrays of inexpensive disks (RAID). In *Proc. of the 1988 ACM SIGMOD Intl. Conf. on Management of data*, pages 109–116, 1988.

[15] J. S. Shapiro and J. Adams. Design Evolution of the EROS Single-Level Store. In *Proc. of the USENIX Annual Technical Conf.*, 2002.

[16] C. A. Stein, J. H. Howard, and M. I. Seltzer. Unifying File System Protection. In *Proc. of the General Track: 2002 USENIX Ann. Tech. Conference*, pages 79–90. USENIX Association, 2001.

[17] S. Sundararaman, S. Subramanian, A. Rajimwale, A. C. Arpaci-Dusseau, R. H. Arpaci-Dusseau, and M. M. Swift. Membrane: operating system support for restartable file systems. In *Proc. of the Eighth USENIX Conf. on File and Storage Technologies*, FAST'10, pages 21–21, 2010.

[18] S. Sundararaman, L. Visampalli, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Refuse to Crash with Re-FUSE. In *Proc. of the 6th European Conf. on Computer Systems*, EuroSys'11, 2011.

[19] D. C. van Moolenbroek, R. Appuswamy, and A. S. Tanenbaum. Integrated End-to-End Dependability in the Loris Storage Stack. In *Proc. of the Seventh int. conf. on Hot topics in system dependability*, HotDep'11, 2011.