Improving File System Reliability with I/O Shepherding

Haryadi S. Gunawi*, Vijayan Prabhakaran†, Swetha Krishnan*,
Andrea C. Arpaci-Dusseau*, Remzi H. Arpaci-Dusseau*

*Department of Computer Sciences, University of Wisconsin, Madison
†Microsoft Research, Silicon Valley

{haryadi, swetha, dusseau, remzi}@cs.wisc.edu, vijayanp@microsoft.com

ABSTRACT

We introduce a new reliability infrastructure for file systems called *I/O shepherding*. I/O shepherding allows a file system developer to craft nuanced *reliability policies* to detect and recover from a wide range of storage system failures. We incorporate shepherding into the Linux ext3 file system through a set of changes to the consistency management subsystem, layout engine, disk scheduler, and buffer cache. The resulting file system, CrookFS, enables a broad class of policies to be easily and correctly specified. We implement numerous policies, incorporating data protection techniques such as retry, parity, mirrors, checksums, sanity checks, and data structure repairs; even complex policies can be implemented in less than 100 lines of code, confirming the power and simplicity of the shepherding framework. We also demonstrate that shepherding is properly integrated, adding less than 5% overhead to the I/O path.

Categories and Subject Descriptors:
D.4.3 [**Operating Systems**]: File Systems Management
D.4.5 [**Operating Systems**]: Reliability
General Terms: Design, Experimentation, Reliability
Keywords: I/O shepherding, storage, fault tolerance, reliability

1. INTRODUCTION

We present the design, implementation, and evaluation of a new reliability infrastructure for file systems called *I/O shepherding*. I/O shepherding provides a simple yet powerful way to build robust reliability policies within a file system, and does so by adhering to a single underlying design principle: *reliability should be a first-class file system concern.* Current approaches bury reliability features deep within the code, making both the intent and the realization of the approach to reliability difficult to understand or evolve. In contrast, with I/O shepherding, the reliability policies of a file system are well-defined, easy to understand, powerful, and simple to tailor to environment and workload.

The I/O shepherd achieves these ends by interposing on each I/O that the file system issues. The shepherd then takes responsibility for the "care and feeding" of the request, specifically by executing a *reliability policy* for the given block. Simple policies will do simple things, such as issue the request to the storage system and return the resulting data and error code (success or failure) to the file system above. However, the true power of shepherding lies in the rich set of policies that one can construct, including sophisticated retry mechanisms, strong sanity checking, the addition of checksums to detect data corruption, and mirrors or parity protection to recover from lost blocks or disks. I/O shepherding makes the creation of such policies simple, by providing a library of primitives that can be readily assembled into a fully-formed reliability policy.

I/O shepherding focuses on reliability in the face of storage system faults, as they are the primary cause of failure in modern storage systems [26]. Modern disks, due to their complex and intricate nature [2], have a wide range of "interesting" failure modes, including latent sector faults [21], block corruption [13, 16], transient faults [36], and whole-disk failure [31]. Thus, many of the primitives provided in the shepherd programming environment center around the detection of and recovery from storage system faults.

A major challenge in implementing I/O shepherding is proper systems integration. We show how to take an existing journaling file system, Linux ext3, and transform it into a shepherding-aware file system, which we call CrookFS. Doing so requires changes to the file system consistency management routines, layout engine, disk scheduler, and buffer cache, as well as the addition of thread support. Many of these alterations are necessary to pass information throughout the system (*e.g.*, informing the disk scheduler where replicas are located so it can read the closer copy); some are required to provide improved control to reliability policies (*e.g.*, enabling a policy to control placement of on-disk replicas).

Of those changes, the most important interaction between the shepherd and the rest of the file system is in the consistency management subsystem. Most modern file systems use *write-ahead logging* to a journal to update on-disk structures in a consistent manner [17]. Policies developed in the shepherd often add new on-disk state (*e.g.*, checksums, or replicas) and thus must also update these structures atomically. In most cases, doing so is straightforward. However, we have found that journaling file systems suffer from a general *problem of failed intentions*, which arises when the intent as written to the journal cannot be realized due to disk failure during checkpointing. Thus, the shepherd incorporates *chained transactions*, a novel and more powerful transactional model that allows policies to handle unexpected faults during checkpointing and still consistently update on-disk structures. The shepherd provides this support *transparently* to all reliability policies, as the required actions are encapsulated in various systems primitives.

We demonstrate the benefits of I/O shepherding through experimentation divided into two parts. First, we show that I/O shepherding enables simple, powerful, and correctly-implemented reliability policies by implementing an increasingly complex set of policies and demonstrating that they behave as desired under disk faults. For example, we show CrookFS is *flexible* by building policies that mimic the failure handling of different file systems such as ReiserFS and NTFS, with only a few lines of policy code. We show that CrookFS is *powerful* by implementing a broad range of poli-

SOSP'07, October 14–17, 2007, Stevenson, Washington, USA.

cies, including single-disk parity protection to guard against data loss due to latent sector errors. We demonstrate that CrookFS enables *fine-grained policies* to be developed; for example, we show that CrookFS can implement D-GRAID style protection for metadata [32], thus ensuring that files remain available despite unexpected disk loss. Throughout the development of these reliability policies, we show how one must consider environmental and workload characteristics to develop a well-reasoned policy.

Second, we show that I/O shepherding is effectively integrated into the rest of the system. Specifically, we show how shepherding adds little overhead across a set of typical workloads and can implement a broad range of policies efficiently. We also establish that chained transactions operate as desired, by forcing the system to crash at key points and observing that the final state of on-disk structures is consistent. Finally, we show how a reliability policy can interact with the file system layout engine to control replica placement on disk, and subsequently that a shepherd-aware disk scheduler can use knowledge of replicas to improve read performance by fetching data from the closer copy. We thus conclude that I/O shepherding is a powerful framework for building robust and efficient reliability features within file systems.

The rest of this paper is as follows. We first present extended motivation (§2) and then discuss the design of I/O shepherding (§3). We next integrate the shepherd into CrookFS (§4) and explore how to craft reliability policies and evaluate their overheads (§5). We close by discussing related work (§6) and concluding (§7).

2. EXTENDED MOTIVATION

We now motivate two underlying assumptions of our work. First, we argue that file systems should be built to handle faults that arise from the storage system. Second, we posit that reliability policies should be flexible, enabling the deployment of different policies depending on the workload and environment.

2.1 The Need For Storage Fault Handling

Storage system failures arise for a large number of reasons. In single-disk systems, despite a vast amount of internal machinery to safeguard their operation [2, 20], disks are failing in an increasingly diverse manner. A recent study of millions of disks found that latent sector faults are prevalent in the field [4], occurring in 8.5% of SATA drives studied. Silent block corruption is an issue [13, 16] and transient faults also occur [36]. Thus, besides whole-disk failure, one should expect both temporary and permanent single-block faults and corruptions [30]. For a desktop PC with a single disk, this has strong implications: a system should likely be prepared to detect and recover from localized faults.

One way to increase reliability of the storage system is to add more disks and a redundancy scheme [29]; although RAID techniques can improve reliability, they do not solve all reliability problems, for three primary reasons [30]. First, RAID does not protect against faults that occur above the disk system (*e.g.*, while the data is in transit). Second, many RAID solutions are geared only towards increasing reliability when entire disks fail; single block faults are not often considered. Finally, most storage arrays do not enable fine-grained policies, due to a lack of information [32].

Of course, file system failure can arise for a variety of other reasons. For example, systems software [8, 12] and operator error [7, 14, 28] are well-known sources of unreliability. However, recent work demonstrates the largest source of faults in a well-built storage system is disk failure [26]; therefore, we focus on adding reliability features to cope with disk faults. Other techniques to improve software resilience (*e.g.*, Nooks [35]) are thus complimentary.

2.2 The Need For Flexible Policies

File systems have classically been deployed in diverse settings. For example, a file system that runs on a desktop machine with a single SATA drive is often the same file system that runs atop a hardware RAID consisting of high-end SCSI drives connected to the system via a storage-area network.

Further, file systems typically run underneath a wide variety of application workloads with differing needs. Again consider the desktop, where the workload might consist of personal productivity applications such as email, word processing, and web browsing, versus a back-end server that runs a web server and supporting database. In the former scenario, the user may wish for high data reliability with modest storage overhead and reasonable performance; in the latter, an administrator may desire the highest performance possible combined with modest reliability. Despite the clear difference in workload requirements, however, the same file system is usually deployed.

From the perspective of reliability, the task of building a file system would certainly be much easier if a single "best" approach to reliability could be decided upon. Unfortunately, recent work demonstrates that today's commodity file systems take different approaches [30]. Linux ext3, for example, is highly sensitive to read faults, as it remounts the file system read-only if a single such fault occurs. In contrast, many other file systems simply propagate read failures to the calling application. Another point in the spectrum is NTFS, which recognizes the sporadic nature of faults, retrying many times before declaring an operation as failed.

Higher-end systems also employ varying approaches to reliability. For example, systems from Tandem, Network Appliance, and others use checksums in some form to protect against corruption [6, 34]; however, only recently has Network Appliance added protection against "lost writes" (*i.e.*, writes the disk claims to have written but has not [34]). Innovation in data protection strategies continues.

Thus, we believe that the best file system reliability strategy is a function of the environment (*e.g.*, how reliable the storage system is and what types of faults are likely to occur) and the workload (*e.g.*, how much performance overhead can be tolerated). What is needed is a flexible environment for developing and deploying differing reliability strategies, enabling developers or administrators to tailor the behavior of the file system to the demands of the site. This need for flexibility drives the I/O shepherding approach.

3. I/O SHEPHERDING

We now describe the goals and design of a file system containing a general framework for providing reliability features. After presenting the goals of I/O shepherding, we present the system architecture and describe how developers specify reliability policies.

3.1 Goals

The single underlying design principle of this work is that *reliability should be a first-class file system concern.* We believe a reliability framework should adhere to the following three goals: simple specification, powerful policies, and low overhead.

3.1.1 Simple specification

We believe that developers should be able to specify reliability policies simply and succinctly. Writing code for reliability is usually complex, given that one must explicitly deal with both misbehaving hardware and rare events; it is especially difficult to ensure that recovery actions remain consistent in the presence of system crashes. We envision that file system developers will take on the role of fault *policy writers*; the I/O shepherd should ease their task.

To simplify the job of a policy writer, the I/O shepherd provides a diverse set of detection and recovery primitives that hide much of the complexity. For example, the I/O shepherd takes care of both the asynchrony of initiating and waiting for I/O and keeps multiple updates and new metadata consistent in the presence of crashes. Policy writers are thus able to stitch together the desired reliability policy with relatively few lines of code; each of the complex policies we craft (§5) requires fewer than 80 lines of code to implement. The end result: less code and (presumably) fewer bugs.

3.1.2 Powerful policies

We believe the reliability framework should enable not only correct policies, but more powerful policies than currently exist in commodity file systems today. Specifically, the framework should enable composable, flexible, and fine-grained policies.

A *composable* policy allows the file system to use different sequences of recovery mechanisms. For example, if a disk read fails, the file system can first retry the read; if the retries continue to fail, the file system can try to read from a replica. With shepherding, policy writers can compose basic detection and recovery primitives in the manner they see fit.

A *flexible* policy allows the file system to perform the detection and recovery mechanisms that are most appropriate for the expected workload and underlying storage system. For example, one may want different levels of redundancy for temporary files in one volume and home directories in another. Further, if the underlying disk is known to suffer from transient faults, one may want extra retries in response. With I/O shepherding, administrators can configure these policy variations for each mounted volume.

A *fine-grained* policy is one that takes different recovery actions depending on the block that has failed. Different disk blocks have different levels of importance to the file system; thus, some disk faults are more costly than others and more care should be taken to prevent their loss. For example, the loss of disk blocks containing directory contents is catastrophic [32]; therefore, a policy writer can specify that all directory blocks be replicated. With I/O shepherding, policies are specified as a function of block type.

3.1.3 Low overhead

Users are unlikely to pay a large performance cost for improved reliability. We have found that it is critical to properly integrate reliability mechanisms with the consistency management, layout, caching, and disk scheduling subsystems. Of course, reliability mechanisms do not always add overhead; for example, a smart scheduler can utilize replicas to improve read performance [19, 39].

3.2 Architecture

To manage the storage system in a reliable way, the I/O shepherd must be able to interpose on every I/O request and response, naturally leading to an architecture in which the shepherd is positioned between the file system and disks (Figure 1). As shown therein, I/O requests issued from different components of a file system (*e.g.*, the file system, journaling layer, and cache) are all passed to the I/O shepherd. The shepherd unifies all reliability code in a single location, making it easier to manage faults in a correct and consistent manner. The shepherd may modify the I/O request (*e.g.*, by remapping it to a different disk block) or perform additional requests (*e.g.*, by reading a checksum block) before sending the request to disk. When a request completes, the response is again routed through the shepherd, which performs the desired fault detection and recovery actions, again potentially performing more disk operations. After the shepherd has executed the reliability policy, it returns the response (success or failure) to the file system.

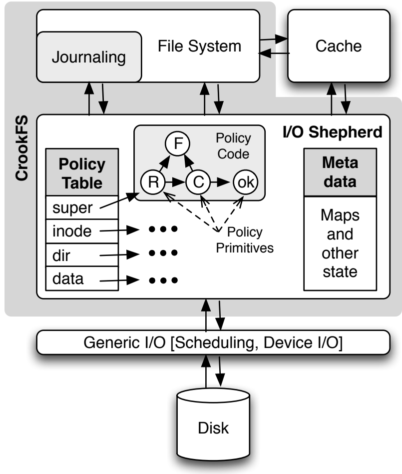

Figure 1: **System Architecture.** *The architecture of a file system containing an I/O shepherd is shown. The file system proper (including journaling) and caching subsystems sit above the I/O shepherd, but have been modified in key locations to interact with the shepherd as necessary. The shepherd itself consists of a policy table, which points to policy code that dictates the detection and recovery strategy for that particular block type. Beneath the shepherd is the generic I/O layer (including disk scheduling, which is slightly modified as well) and one (or more) disks.*

3.2.1 Policy Table and Code

With I/O shepherding, the reliability policy of the file system is specified by a *policy table*; this structure specifies which code to execute when the file system reads or writes each type of on-disk data structure (*e.g.*, superblock, inode, or directory). Each entry in the table points to *policy code*, which defines the sequence of actions taken for a block of a particular type. For example, given an ext3-based file system, a policy writer can specify a different policy for each of its 13 block types; the table could thus mandate triple replication of the superblock, checksum and parity protection for other metadata, and an aggressive retry scheme for user data.

Although this design does not directly support different policies for individual files, the I/O shepherd allows a different policy table per mounted file system. Thus, administrators can tailor the policy of each volume, a basic entity they are accustomed to managing. For example, a `/tmp` volume could employ little protection to obtain high performance while an archive could add checksums and parity to improve reliability at some performance cost.

3.2.2 Policy Metadata

To implement useful policies, an I/O shepherd often requires additional on-disk state to track the location of various blocks it is using (*e.g.*, the location of checksums, replicas, or parity blocks). Thus, to aid in the management of persistent metadata, the I/O shepherd framework provides *maps*. Some commonly used maps are a `CMap` to track checksum blocks, an `RMap` to record bad block remappings, and an `MMap` to track multiple replicas.

A policy can choose to use either a *static* or *dynamic* map for a particular type of metadata. With static mapping, the association between a given on-disk block and its checksum or replica location is fixed when the file system is created. With a dynamic map, new associations between blocks can be created over time.

There are obvious trade-offs to consider when deciding between static and dynamic maps. Static maps are simple to maintain but inflexible; for example, if a static map is used to track a block and its copy, and one copy becomes faulty due to a latent sector error, the map cannot be updated with a new location of the copy.

Dynamic maps are more flexible, as they can be updated as the file system is running and thus can react to faults as they occur. However, dynamic maps must be reflected to disk for reliability. Thus, updating dynamic maps consistently and efficiently is a major challenge; we describe the problem and our approach to solving it in more detail below (§4.1).

3.2.3 Policy Primitives

To ease the construction of policy code, the shepherd provides a set of *policy primitives*. The primitives hide the complexity inherent to reliable I/O code; specifically, the primitives ensure that policy code updates on-disk structures in a single transaction. Clearly, a fundamental tension exists here: as more functionality is encapsulated in each primitive, the simpler the policy code becomes, but the less control one has over the reliability policy. Our choice has generally been to expose more control to the policy writers.

The I/O shepherd provides five classes of reliability primitives. All primitives return failure when the storage system itself returns an error code or when blocks do not have the expected contents.

Read and Write: The I/O shepherd contains basic primitives for reading and writing either a single block or a group of blocks concurrently from disk. A specialized primitive reads from mirrored copies on disk: given a list of blocks, it reads only the block that the disk scheduler predicts has the shortest access time.

Integrity: On blocks that reside in memory, primitives are provided to compute and compare checksums, compare multiple blocks, and perform strong sanity checks (*e.g.*, checking the validity of directory blocks or inodes).

Higher-level Recovery: The I/O shepherd contains primitives to stop the file system with a panic, remount the file system read-only, or even reboot the system. Primitives are also provided that perform semantic repair depending upon the type of the block (*e.g.*, an inode or a directory block) or that run a full `fsck` across the disk.

Persistent Maps: The I/O shepherd provides primitives for looking up blocks in an indirection map and for allocating (or reallocating and freeing) new entries in such a map (if it is dynamic).

Layout: To allow policy code to manage blocks for its own use (*e.g.*, for checksums, remapped blocks, and replicas), the I/O shepherd can allocate blocks from the file system. One primitive exposes information about the current layout in the file system while a second primitive allocates new blocks, with hooks to specify preferences for block placement. With control over block placement, policy code can provide trade-offs between performance and reliability (*e.g.*, by placing a replica near or far from its copy).

3.3 Example Policy Code

The I/O shepherd enables one to specify reliability policies that are traditionally implemented across different levels of the storage stack. For example, one can specify policies that operate on a single block and are often performed within disks (*e.g.*, retrying, remapping, and checksums), policies that operate across multiple blocks or multiple disks (*e.g.*, mirrors and parity), and finally, one can specify policies requiring semantic information about the failed block and are usually performed by the file system (*e.g.*, stopping the file system, data structure repair, and fsck). A shepherd enables policies that compose all of these strategies.

We now show the simplicity and power of the shepherd through a number of examples. The names of all policy primitives begin with `IOS` for clarity. We simplify the pseudo-code by ignoring some of the error codes that are returned by the policy primitives, such as `IOS_MapLookup` and `IOS_MapAllocate` (with irony noted).

The first example policy is based loosely on NTFS [30]. The NTFS policy tries to keep the system running when a fault arises by first retrying the failed read or write operation a fixed number of times; if it is unable to complete the operation, the fault is simply propagated to the application. We show the read version of the code here (the write is similar).

```
NTFSRead(DiskAddr D, MemAddr A)
  for (int i = 0; i < RETRY_MAX; i++)
      if (IOS_Read(D, A) == OK)
          return OK;
  return FAIL;
```

The second example policy loosely emulates the behavior of ReiserFS [30]. This policy chooses reliability over availability; whenever a write fault occurs, the policy simply halts the file system. By avoiding updates after a fault, this conservative approach minimizes the chance of further damage.

```
ReiserFSWrite(DiskAddr D, MemAddr A)
  if (IOS_Write(D, A) == OK)
      return OK;
  else
      IOS_Stop(IOS_HALT);
```

The next two examples show the ease with which one can specify policies that detect block corruption. `SanityRead` performs type-specific sanity checking on the read block using a shepherd primitive; note in this example how the block type can be passed to and used by policy code. `ChecksumRead` uses checksums to detect block corruption; the policy code first finds the location of the checksum block, then concurrently reads both the stored checksum and the data block (the checksum may be cached), and then compares the stored and newly computed checksums.

```
SanityRead(DiskAddr D, MemAddr A, BlockT T)
   if (IOS_Read(D, A) == FAIL)
      return FAIL;
   return IOS_SanityCheck(A, T);
ChecksumRead(DiskAddr D, MemAddr A)
   DiskAddr cAddr;
   ByteOffset off;
   CheckSum onDisk;
   IOS_MapLookupOffset(ChMap, D, &cAddr, &off);
   // read from checksum and D concurrently
   if (IOS_Read(cAddr, &onDisk, D, A)==FAIL)
      return FAIL;
   CheckSum calc = IOS_Checksum(A);
   return IOS_Compare(onDisk, off, calc);
```

The next two examples compare how static and dynamic maps can be used for tracking replicas. `StaticMirrorWrite` assumes that the mirror map, `MMap`, was configured for each block when the file system was created. `DynMirrorWrite` checks to see if a copy already exists for the block being written to; if the copy does not exist, the code picks a location for the mirror and allocates (and persistently stores) an entry in `MMap` for this mapping.

```
StaticMirrorWrite(DiskAddr D, MemAddr A)
   DiskAddr copyAddr;
   IOS_MapLookup(MMap, D, &copyAddr);
   // write to both copies concurrently
   return (IOS_Write(D, A, copyAddr, A));
DynMirrorWrite(DiskAddr D, MemAddr A)
   DiskAddr copyAddr;
   // copyAddr is set to mirrored block
   //    or NULL if no copy of D exists
   IOS_MapLookup(MMap, D, &copyAddr);
   if (copyAddr == NULL)
      PickMirrorLoc(MMap, D, &copyAddr);
      IOS_MapAllocate(MMap, D, copyAddr);
   return (IOS_Write(D, A, copyAddr, A));
```

The final two policy examples show how blocks can be remapped; the map of remapped blocks is most naturally a dynamic map, since the shepherd does not know *a priori* which writes will fail. RemapWrite is responsible for the remapping; if a write operation fails, the policy code picks a new location for that block, allocates a new mapping for that block in RMap, and tries the write again. RemapRead checks RMap to see if this block has been previously remapped; the read to the disk is then redirected to the possibly new location. Of course, all of these policies can be extended, for example, by retrying if the disk accesses fail or stopping the file system on failure.

```
RemapWrite(DiskAddr D, MemAddr A)
   DiskAddr remap;
   // remap is set to remapped block
   //    or to D if not remapped
   IOS_MapLookup(RMap, D, &remap);
   if (IOS_Write(remap, A) == FAIL)
      PickRemapLoc(RMap, D, &remap);
      IOS_MapAllocate(RMap, D, remap);
   return IOS_Write(remap, A);
RemapRead(DiskAddr D, MemAddr A)
   DiskAddr remap;
   IOS_MapLookup(RMap, D, &remap);
   return IOS_Read(remap, A);
```

4. BUILDING CROOKFS

We now describe how to integrate I/O shepherding into an existing file system, Linux ext3. For our prototype system, we believe that it is important to work with an existing file system in order to leverage the optimizations of modern systems and to increase the likelihood of deployment. We refer to the ext3 variant with I/O shepherding as CrookFS, named for the hooked staff of a shepherd.

Integrating shepherding with ext3 instead of designing a system from scratch does introduce challenges in that the shepherd must explicitly interact with numerous components of the file system, including the journal, buffer cache, layout algorithm, and disk scheduler. We devote most of our discussion to how we integrate CrookFS with ext3 journaling to ensure consistency, and then describe integration with other key subsystems.

4.1 Consistency Management

In order to implement some reliability policies, an I/O shepherd requires additional data (*e.g.*, checksum and replica blocks) and metadata (*e.g.*, persistent maps). Keeping this additional information consistent can be challenging. As an example, consider policy code that dynamically creates a replica of a block; doing so requires picking available space on disk for the replica, updating the mirror map to record the location of the replica, and writing the original and replica blocks to disk. One would like these actions to be performed atomically despite the presence of crashes.

Given that ext3 employs journaling for file system metadata and user data, a natural question is whether the ext3 journal can also be used for consistently updating CrookFS data and metadata. Therefore, we briefly review how journaling is performed in ext3.

4.1.1 Journaling Basics

Ext3 and most other current commodity file systems (including Reiser3, IBM JFS, and NTFS) use journaling [17]. To describe the basics of journaling, we borrow the terminology of ext3 [38]. Journaling exists in three different modes (data, ordered, and writeback), each of which provides different levels of consistency. In data journaling, all traffic to disk is first committed to a log (*i.e.*, the journal) and then checkpointed to its final on-disk location. The other journaling modes journal only metadata, enabling consistent update for file systems structures but not for user data. Given that the goal of I/O shepherding is to enable highly robust file systems, we build upon the most robust form of journaling, data journaling.

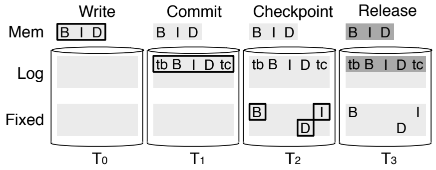

Figure 2: **Journaling Data Flow.** *The figure shows the series of actions that take place in data journaling mode. Both in-memory (top) and on-disk (bottom) states are shown.* **D** *is a data block,* **I** *an inode,* **B** *a bitmap,* **tb** *the beginning of a transaction, and* **tc** *the commit block. Darker gray shading indicates that blocks have been released after checkpointing.*

The sequence of operations when an application appends a data block **D** to a file is shown in Figure 2. At time T_0, the data block **D** and bitmap **B** are updated in memory and a pointer to **D** is added to the inode **I**; all three blocks (**D, I, B**) now sit in memory and are dirty. At time T_1, the three blocks **D, I,** and **B** are wrapped into a *transaction* and the journaling layer *commits* the transaction (which includes markers for the start **tb** and end **tc** of the transaction) to the journal; the blocks are now marked clean but are still pinned in memory. After the commit step is complete, at T_2, the blocks are *checkpointed* to their final fixed locations. Finally, at T_3, the transaction is *released* and all three blocks are unpinned and can be flushed from memory and the log space reused. Note that multiple transactions may be in the checkpointing step concurrently (*i.e.*, committed but not yet released). If the system crashes, the file system will recover by replaying transactions in the journal that are committed but not yet released.

4.1.2 Strawman Shepherds

To understand how CrookFS uses the ext3 journal to maintain consistency, we begin with two strawman approaches. Neither work; rather, we use them to illustrate some of the subtleties of the problem. In the *early strawman* approach, the shepherd interposes on the preceding journal writes to insert its own metadata for this transaction. This requires splitting policy code for a given block type into two portions: one for the operations to be performed on the journal write for that block and one for operations on a checkpoint. In the *late strawman*, the shepherd appends a later transaction to the journal containing the needed information. This approach assumes that the policy code for a given block is invoked only at checkpoint time. We now describe how both strawmen fail.

First, consider the DynMirrorWrite policy (presented in §3.3). On the first write to a block **D**, the policy code picks, allocates, and writes to a mirror block **C** (denoted copyAddr in the policy code); at this time, the data bitmap **B'** and the mirror map **M** are also updated to account for **C**. All of these actions must be performed atomically relative to the writing of **D** on disk.

The early strawman can handle the DynMirrorWrite policy, as shown in Figure 3. When the early strawman sees the entry for **D** written to the journal (T_1), it invokes policy code to allocate an entry for **C** in **M** and **B'** and to insert **M** and **B'** in the current transaction. When **D** is later checkpointed (T_2), similar policy code is again invoked so that the copy **C** is updated according to the mirror map **M**. With the early strawman, untimely crashes do not cause problems because all metadata is in the same transaction.

Now, consider the RemapWrite policy (presented in §3.3). This policy responds to the checkpoint failure of a block **D** by remapping **D** to a new location, **R** (denoted remap in the policy code). However, the early strawman cannot implement this policy. As shown in Figure 4, after the write to a data block **D** fails (T_2) the policy wants to remap block **D** to **R** (T_3), which implies that the bitmap and RMap are modified (**B'** and **M**). However, it is too late to mod-

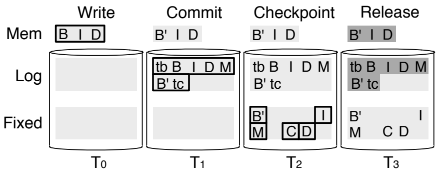

Figure 3: **Early Strawman for** `DynMirrorWrite`**.** *The figure shows how the early strawman writes to a replica of a data block* **D**.

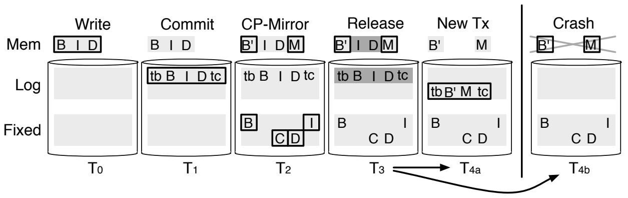

Figure 5: **Late Strawman for** `DynMirrorWrite`**.** *The figure shows the incorrect timing of the new transaction commit.*

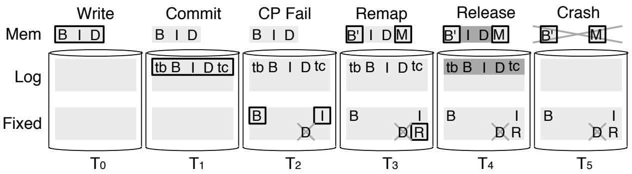

Figure 4: **Early Strawman for** `RemapWrite`**.** *The figure illustrates how the early strawman cannot deal with the problem of failed intentions.*

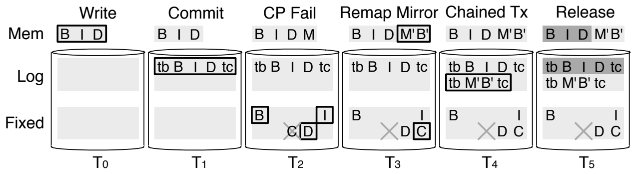

Figure 6: **Chained Transactions for** `RemapMirrorWrite`**.** *The figure shows how chained transactions handle failed intentions.*

ify the transaction that has been committed. Thus, if a crash occurs (T_5) after the transaction is released (T_4), all metadata changes will be discarded and the disk will be in an inconsistent state. Specifically, the data block **D** is lost since the modified `RMap` that has the reference to **R** has been discarded.

More generally, the early strawman cannot handle any checkpoint failures that result in metadata changes, because it must calculate *a priori* to the actual checkpoint what will happen at checkpoint time. We refer to this as the problem of *failed intentions*. Failed intentions occur when a commit to the journal succeeds but the corresponding checkpoint does not; the intent of the update (as logged in the journal) cannot be realized due to checkpoint failure.

We now examine the late strawman which only invokes policy code at checkpoint time. Given that it is "too late" to modify a transaction at checkpoint time, the late strawman adds another transaction with the new metadata. Unfortunately, the late strawman cannot correctly handle the `DynMirrorWrite` policy, as shown in Figure 5. During the checkpoint of block **D** (T_2), the late strawman invokes the policy code, creates and updates the copy **C** as desired. After this transaction has been released (T_3), a new transaction containing **B'** and **M** is added to the journal (T_{4a}). The problem with the late strawman is that it cannot handle a system crash that occurs between the two transactions (*i.e.*, T_{4b}, which can occur between T_3 and T_{4a}): **D** will not be properly mirrored with a reachable copy. When the file system recovers from this crash, it will not replay the transaction writing **D** (and **C**) because it has already been released and it will not replay the transaction containing **B'** and **M** because it has not been committed; as a result, copy **C** will be unreachable. Thus, the timing of the new transaction is critical and must be carefully managed, as we will see below.

4.1.3 Chained Transactions

We solve the problem of failed intentions with the development of *chained transactions*. With this approach, like the late strawman, all metadata changes initiated by policy code are made at checkpoint time and are placed in a new transaction; however, unlike the late strawman, this new chained transaction is committed to the journal *before* the old transaction is released. As a result, a chained transaction makes all metadata changes associated with the checkpoint appear to have occurred at the same time.

To illustrate chained transactions we consider a reliability policy that combines mirroring and remapping. We consider the case where this is not the first write to the block **D** (*i.e.*, an entry in the mirror map should already exist) and it is the write to the copy **C** that fails. Code paths *not* taken are gray and slanted.

```
RemapMirrorWrite(DiskAddr D, MemAddr A)
   DiskAddr copy, remap;
   Status status1 = OK, status2 = OK;
   IOS_MapLookup(MMap, D, &copy);
    // remap is set to D if not remapped
   IOS_MapLookup(RMap, D, &remap);
   if (copy == NULL)
      PickMirrorLoc(MMap, D, &copy);
      IOS_MapAllocate(MMap, D, copy);
   if (IOS_Write(remap, A, copy, A) == FAIL)
      if (IOS_Failed(remap))
         PickRemapLoc(RMap, D, &remap);
         IOS_MapAllocate(RMap, D, remap);
         status1 = IOS_Write(remap, A);
      if (IOS_Failed(copy))
         PickMirrorLoc(MMap, D, &copy);
         IOS_MapAllocate(MMap, D, copy);
         status2 = IOS_Write(copy, A);
   return ((status1==FAIL)||(status2==FAIL));
```

Figure 6 presents a timeline of the activity in the system with chained transactions. With chained transactions, committing the original transaction is unchanged as seen at times T_0 and T_1 (policy code will be invoked when each of the blocks in the journal is written, but its purpose is to implement the reliability policy of the journal itself). When the data block **D** is checkpointed, the `RemapMirrorWrite` policy code is invoked for **D**. The policy code finds the copy location of **C** (denoted `copy`) and the potentially remapped location of **R** (denoted `remap`). In our example, we assume that writing to the copy **C** fails (T_2); in this case, the policy code allocates a new location for **C** (hence dirtying the bitmap, **B'**), writes the copy to a new location, and updates the mirror map **M'** (T_3). Our integration of the shepherd primitive, `IOS_MapAllocate`, with the ext3 journaling layer ensures that the chained transaction containing **B'** and **M'** is committed to the journal (T_4) before releasing the original transaction (T_5). At time T_6 (not shown), when the chained transaction is checkpointed, the blocks **B'** and **M'** are finally written to their fixed locations on disk; given that these are normal checkpoint writes, the relevant policy code will be applied to these updates.

With a chained transaction, a crash cannot occur "between" the two related transactions, because the second transaction is always committed before the first is released. If the system crashes before the first transaction is released, all operations will be replayed.

Chained transactions ensure that shepherd data and metadata are kept consistent in the presence of crashes. However, if one is not careful, chained transactions introduce the possibility of deadlock. Specifically, because CrookFS now holds the previous checkpoint while waiting to commit the chained transaction, we must avoid the two cases that can lead to circular dependencies. First, CrookFS must ensure that sufficient space exists in the journal for all chained transactions; this constrains the number of remappings (and subsequent chained transactions) that can occur in any policy code. Second, CrookFS must use shadow copies when updating a shepherd metadata block that exists in a previous transaction (just as it does for its own metadata), instead of acquiring locks.

To test the chained transaction infrastructure in the presence of system crashes, we have constructed a testing framework that inserts crashes at interesting points in the journaling sequence. We have injected 16 crash points during journaling updates and 8 crash points during recovery. We have verified that in all cases the recovery process brings the file system to the expected on-disk state.

4.1.4 Non-Idempotent Policies

To maintain consistency, all failure scenarios must be considered, including repeated crashes during recovery. Repeated crashes will cause CrookFS to replay the same transactions and their corresponding checkpoints. In such scenario, only *idempotent* policy code will work correctly.

For example, consider a policy that protects data blocks with parity. Although parity can be computed using an idempotent equation ($P = D1 \oplus D2 \oplus ... \oplus Dn$), this approach performs poorly because $N - 1$ blocks must be read every time a block is modified. However, the more efficient way of computing parity ($P_{new} = P_{old} \oplus D_{old} \oplus D_{new}$) is non-idempotent since P_{old} and D_{old} will be overwritten with P_{new} and D_{new}, respectively. Thus, repeated journal replays will incorrectly calculate the parity block.

To handle non-idempotent policies such as parity, CrookFS provides an old-value logging mechanism [15]. The old-value log annotates versions to distinguish old and new values, and writes the old data and its corresponding version into the log atomically. Thus, non-idempotent policy code must take care to read the old values and log them into the old-value log, using support within CrookFS. Simplified policy code for `ParityWrite` is as follows.

```
ParityWrite(DiskAddr D, MemAddr aNew)
  DiskAddr P;
  MemAddr  aOld, apOld, apNew;
  IOS_MapLookup(PMap, D, &P);
  if (IOS_ReadStable(D,aOld,P,apOld) == FAIL)
      return FAIL;
  if (IOS_WriteAndLog(D,aOld,P,apOld) == FAIL)
      return FAIL;
  apNew = ComputeParity(apOld, aOld, aNew);
  return (IOS_Write(D, aNew, P, apNew));
```

4.1.5 Reliability of the Journal and Shepherd Maps

A final complication arises when the reliability policy wishes to increase the protection of the journal or of shepherd metadata itself. Although there are a large number of reasonable policies that do not add protection features to the journal (since it is only read in the event of an untimely crash), some policies might wish to add features (*e.g.*, replication or checksumming). The approaches we describe above do not work for the journal, since they use the journal itself to update other structures properly. Thus, we treat journal replication, checksumming, and other similar actions as a special case, mandating a restricted set of possible policies. Similar care must be taken when applying policy to shepherd metadata such as is found in the various map structures (*e.g.*, the mirror map).

4.2 System Integration

To build effective reliability policies, the shepherd must interact with other components of the existing file system. Below, we discuss these remaining technical hurdles.

4.2.1 Semantic Information

To implement fine-grained policies, the shepherding layer must have information about the current disk request; in our implementation, shepherding must know the the type of each block (*e.g.*, whether it is an inode or a directory) and whether the request is a read or a write in order to call the correct policy code as specified in the policy table.

In the case where requests are issued directly from the file system, acquiring this information is straightforward: the file system is modified to pass the relevant information with each I/O call. When I/O calls are issued from common file system layers (*e.g.*, the generic buffer cache manager), extra care must be taken. First, the buffer cache must track block type for its file blocks and pass this information to the shepherd when calling into it. Second, the buffer cache must only pass this information to shepherd-aware file systems. A similar extension was made to the generic journaling and recovery layers to track the type of each journaled block.

4.2.2 Threads

I/O shepherding utilizes threads to handle each I/O request and any related fault management activity. A thread pool is created at mount time, and each thread serves as an execution context for policy code. Thus, instead of the calling context issuing a request directly and waiting for it to complete, it enqueues the request and lets a thread from the pool handle the request. This thread then executes the corresponding policy code, returning success or failure as dictated by the policy. When the policy code is complete, the caller is woken and passed this return code.

We have found that a threaded approach greatly simplifies the task of writing policy code, where correctness is of paramount importance; without threads, policy code was split into a series of event-based handlers that executed before and after each I/O, often executing in interrupt context and thus quite difficult to program. A primary concern of our threaded approach is overhead, which we explore in Section 5.2.

4.2.3 Legacy Fault Management

Because the shepherd now manages file system reliability, we removed the existing reliability code from ext3. Thus, the upper layers of CrookFS simply propagate faults to the application. Note that some sanity checks from ext3 are kept in CrookFS, since they are still useful in detecting memory corruption.

One issue we found particularly vexing was correct error propagation; a closer look revealed that ext3 often accidentally changed error codes or ignored them altogether. Thus, we developed a simple static analysis tool to find these bugs so we could fix them. To date we have found and fixed roughly 90 places within the code where EIO was not properly propagated; we are currently building analysis tools to more thoroughly address this problem.

4.2.4 Layout Policy

Fault management policies that dynamically require disk space (*e.g.*, for checksum or replica blocks) must interact with the file system layout and allocation policies. Since reliability policies are likely to care about the location of the allocated blocks (*e.g.*, to place blocks away from each other for improved reliability), we have added two interfaces to CrookFS. The first exposes information about the current layout in the file system. The second allows a reliability policy to allocate blocks with options to steer block placement. Policy code can use these two interfaces to query the file system and request appropriate blocks.

Changes in Core OS	
Chained transactions	26
Semantic information	600
Layout and allocation	176
Recovery	108
Total	*910*
Shepherd infrastructure	
Thread model	900
Data structures	743
Read/Write + Chained Transactions	460
Layout and allocation	66
Scheduling	220
Sanity + Checksums + fsck + Mirrors	429
Support for multiple disks	645
Total	*3463*

Table 1: **CrookFS Code Complexity.** *The table presents the amount of code added to implement I/O shepherding as well as a breakdown of where that code lives. The number of lines of code is counted by tallying the number of semi-colons in code that we have added or changed.*

4.2.5 Disk Scheduling

For improved performance, the disk scheduler should be integrated properly with reliability policies. For example, the scheduler should know when a block is replicated, and access the nearer block for better performance [19, 39].

We have modified the disk scheduler to utilize replicas as follows. Our implementation inserts a request for each copy of a block into the Linux disk scheduling queue; once the existing scheduling algorithm selects one of these requests to be serviced by disk, we remove the other requests. When the request completes, the scheduler informs the calling policy which replica was serviced, so that faults can be handled appropriately (*e.g.*, by trying to read the other replica). Care must be taken to ensure that replicated requests are not grouped and sent together to the disk.

4.2.6 Caching

The major issue in properly integrating the shepherd with the existing buffer cache is ensuring that replicas of the same data do not simultaneously reside in the cache, wasting memory resources. By placing the shepherd beneath the file system, we circumvent this issue entirely by design. When a read is issued to a block that is replicated, the scheduler decides to read one copy or the other; while this block is cached, the other copy will never be read, and thus only a single copy can reside in cache.

4.2.7 Multiple Disks

One final issue arose from our desire to run CrookFS on multiple disks to implement more interesting reliability policies. To achieve this, we mount multiple disks using a concatenating RAID driver [11]. The set of disks appears to the file system as one large disk, with the first portion of the address space representing the first disk, the second portion of the address space representing the second disk, and so forth. By informing CrookFS of the boundary addresses between disks, CrookFS allocation policies can place data as desired across disks (*e.g.*, data on one disk, a replica on another).

4.2.8 Code Complexity

Table 1 summarizes the code complexity of CrookFS. The table shows that the changes to the core OS were not overly intrusive (*i.e.*, 910 C statements were added or changed); the majority of the changes were required to propagate the semantic information about the type of each block through the file system. Many more lines of code (*i.e.*, 3463) were needed to implement the shepherd infrastructure itself. We are hopeful that incorporating I/O shepherding into file systems other than ext3 will require even smaller amounts of code, given that much of the infrastructure can be reused.

	PostMark	TPC-B	SSH-Build
Linux ext3	1.00	1.00	1.00
Propagate	1.00	1.05	1.01
Retry and Reboot	1.00	1.05	1.01
Parity	1.14	1.27	1.02
Mirror_{Near}	1.59	1.41	1.04
Mirror_{Far}	1.65	1.87	1.06
Sanity Check	1.01	1.05	1.01
Multiple Lines of Defense	1.10	1.28	1.01

Table 2: **Performance Overheads.** *The table shows the performance overhead of different policies in CrookFS relative to unmodified ext3. Three workloads are run: PostMark, TPC-B, and ssh. Each workload is run five times; averages are reported (there was little deviation). Running times for standard ext3 on PostMark, TPC-B, and SSH-Build are 51, 29.33, 68.19 seconds respectively. The Multiple Lines of Defense policy incorporates checksums, sanity checks, and mirrors.*

Propagate	8	Mirror	18
Reboot	15	Sanity Check	10
Retry	15	Multiple Lines of Defense	39
Parity	28	D-GRAID	79

Table 3: **Complexity of Policy Code.** *The table presents the number of semicolons in the policy code evaluated in Section 5.*

5. CRAFTING A POLICY

We now explore how I/O shepherding simplifies the construction of reliability policies. We make two major points in this section. First, the I/O shepherding framework does not add a significant performance penalty. Second, a wide range of useful policies can be easily built in CrookFS, such as policies that propagate errors, perform retries and reboots, policies that utilize parity, mirroring, sanity checks, and checksums, and policies that operate over multiple disks. Overall, we find that our framework adds less than 5% performance overhead on even I/O-intensive workloads and that no policy requires more than 80 lines of policy code to implement.

We also make two relatively minor points. First, effective yet simple reliability policies (*e.g.*, retrying requests and performing sanity checks) are not consistently deployed in commodity file systems, but they should be to improve availability and reliability. Second, CrookFS is integrated well with the other components of the file system, such as layout and scheduling.

5.1 Experimental Setup

The experiments in this section were performed on an Intel Pentium 4 machine with 1 GB of memory and up to 4 120 GB 7200 RPM Western Digital EIDE disks (WD1200BB). We used the Linux 2.6.12 operating system and built CrookFS from ext3 therein.

To evaluate the performance of different reliability policies under fault-free conditions, we use a set of well-known workloads: PostMark [22], which emulates an email server, a TPC-B variant [37] to stress synchronous updates, and SSH-Build, which unpacks and builds the ssh source tree. Table 2 shows the performance on PostMark, TPC-B, and SSH-Build of eight reliability policies explored in more detail in this section, relative to unmodified ext3.

To evaluate the reliability policies when faults occur, we stress the file system using type-aware fault injection [30] with a pseudo-device driver. To emulate a block failure, the pseudo-device simply returns the appropriate error code and does not issue the operation to the underlying disk. To emulate corruption, the pseudo-device changes bits within the block before returning the data. The fault injection is type aware in that it can be selectively applied to each of the 13 different block types in ext3.

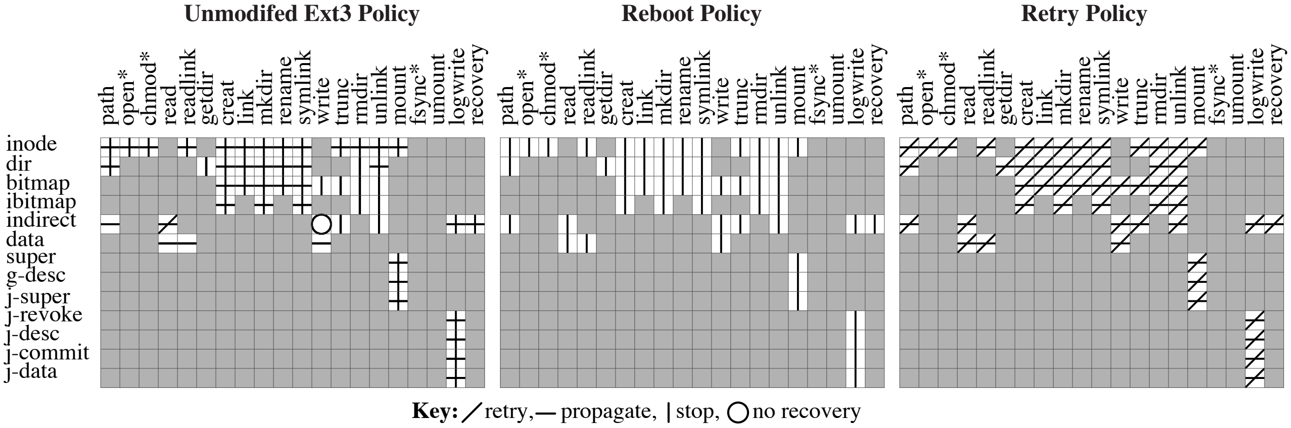

Figure 7: **Comparison of Ext3, Reboot, and Retry Policies.** *The table shows how unmodified ext3 (left) and CrookFS with a reboot (middle) and a retry (right) policy react to read faults. Along the x-axis, different workloads are shown; each workload stresses either a posix API call or common file system functionality (e.g., path lookup). Along the y-axis, the different data structures of the file system are presented. Each (x,y) location presents the results of a read fault injection of a particular data structure (y) under the given workload (x). The four symbols (/, –, |, and O) represent the detection and recovery techniques used by the file systems. If multiple techniques are used, the symbols are superimposed. A gray box indicates the workload does not access the given block type. Some columns represent multiple workloads: open* → (open, stat, access); chmod* → (chmod, chown); fsync* → (fsync, sync).*

5.2 Propagate

The first and most basic question we answer is: how costly is it to utilize the shepherding infrastructure within CrookFS? To measure the basic overhead of I/O shepherding, we consider the simplest reliability policy: a null policy that simply propagates errors through the file system. Table 3 reports the number of lines of code to implement each reliability policy; the basic propagate policy is extremely simple, requiring only 8 statements.

The second line of Table 2 reports the performance of the propagate policy, normalized with respect to unmodified Linux ext3. For the propagate policy, the measured slowdowns are 5% or less for all three workloads. Thus, we conclude that the basic infrastructure and its threaded programming model do not add noticeable overhead to the system.

5.3 Reboot vs. Retry

We next show the simplicity of building robust policies given our I/O shepherding framework. We use CrookFS to implement two straightforward policies: the first halts the file system upon a fault (with optional reboot); the second retries the failed operation a fixed number of times and then propagates the error code to the application. The pseudo code for these two policies was presented earlier (§3.3). As shown in Table 3 the actual number of lines of code needed to implement these policies is very small: 15 for each.

To demonstrate that CrookFS implements the desired reliability policy, we inject type-aware faults on read operations. To stress many paths within the file system, we utilize a synthetic workload that exercises the POSIX file system API. The three graphs in Figure 7 show how the default ext3 file system, the Reboot, and Retry policies respond to read faults for each workload and for each block type. The leftmost graph (taken from [30]) shows that the default ext3 file system does not have a consistent policy for dealing with read faults; for example, when reading an indirect block fails as part of the file writing workload, the error is not even propagated to the application.

The middle and rightmost graphs of Figure 7 show CrookFS is able to correctly implement the Reboot and Retry policies; for every workload and for every type of block, CrookFS either stops the file system or retries the request and propagates the error, as desired. Further, during fault-free operation, the CrookFS implementation of these two policies has negligible overhead; Table 2 shows that the performance of these two policies is equivalent to the simple Propagate policy on the three standard workloads.

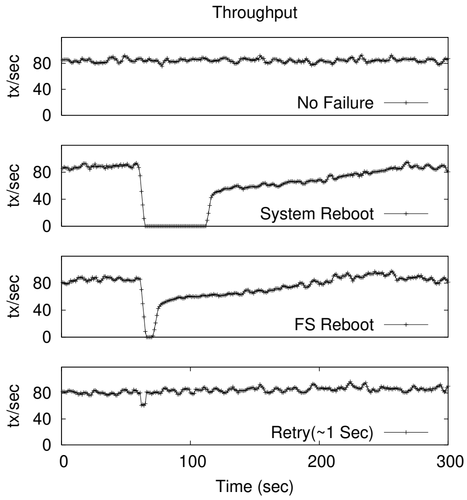

Figure 8: **Reboot vs. Retry (Throughput).** *The throughput of PostgreSQL 8.2.4 running pgbench is depicted. The database is initialized with 1.5 GB of data, and the workload performs a series of simple SELECTs. Four graphs are presented: the first with no fault injected (top), and the next three with a transient fault. The bottom three graphs show the different responses from three different policies: full system reboot, file system reboot, and retry.*

Figure 8 compares the availability implications of system reboot, a file system microreboot (in which the file system is unmounted and remounted), and retrying in the presence of a transient fault. For these experiments, we measure the throughput of PostgreSQL 8.2.4 running a simple database benchmark (pgbench) over time; we inject a single transient fault in which the storage system is unavailable for one second. Not surprisingly, the full reboot can be quite costly; the system takes nearly a minute to reboot, and then delivers lower throughput for roughly another minute as the cache warms. The microreboot fairs better, but still suffers from the same cold-cache effects. Finally, the simple retry is quite effective in the face of transient faults.

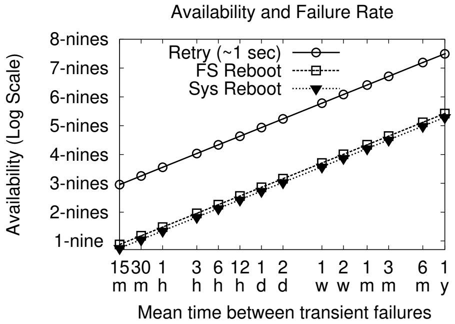

Figure 9: **Reboot vs. Retry (Availability).** *The graph shows the computed availability (in terms of "nines") plotted versus the mean time between transient failures for the three policies: full system reboot, file system reboot, and retry. The system is considered "available" when its delivered performance is within 10% of average steady-state performance.*

Given these measurements, one can calculate the impact of these three reliability policies on system availability. Figure 9 plots system availability as a function of the frequency of transient faults, assuming that unavailability is due only to transient faults and that the system is available when its delivered throughput is within 10% of its average steady-state throughput. To calibrate the expected frequency of transient faults, we note that although most disks encounter transient faults only a few times a year, a poorly-behaving disk may exhibit a transient fault once per week [3]. Given a weekly transient fault, the reboot strategy has availability of only "three 9s", while the retry strategy has "six 9s".

In summary, it is well known that rebooting a system when a fault occurs has a large negative impact on availability; however, many commodity file systems deployed today still stop the system instead of retrying an operation when they encounter a transient error (*e.g.*, ext3 and ReiserFS [30]). With CrookFS, one can easily specify a consistent retry policy that adds negligible slowdown and can improve availability markedly in certain environments.

5.4 Parity Protection

With the increasing prevalence of latent sector errors [4], file systems should contain reliability policies that protect against data loss. Such protection is usually available in high-end RAIDs [10], but not in desktop PCs [30]. For our next reliability policy, we demonstrate the ease with which one can add parity protection to a single drive so that user data can survive latent sector errors.

The parity policy is slightly more complex than the retry and reboot policies, but is still quite reasonable to implement in CrookFS; as shown in Table 3, our simple parity policy requires 28 lines of code. As described in Section 4.1.4, calculating parity efficiently is a non-idempotent operation, and thus the policy code must perform old-value logging. We employ a static parity scheme, which adds one parity block for k file system blocks (k is configured at boot time). A static map is used to locate parity blocks.

To help configure the value of k, we examine the trade-off between the probability of data loss and space overhead. Figure 10 shows both the probability of data loss (bottom) and the space overhead (top) as a function of the size of the parity set. To calculate the probability of data loss, we utilize recent work reporting the frequency of latent sector errors [4], as described in the figure caption. The bottom graph shows that using too large of a parity set leads to a high probability of data loss; for example, one parity block for the entire disk (the rightmost point) has over a 20% chance of data loss. However, the top graph shows that using too small of a parity set leads to high space overheads; for example, one parity block per

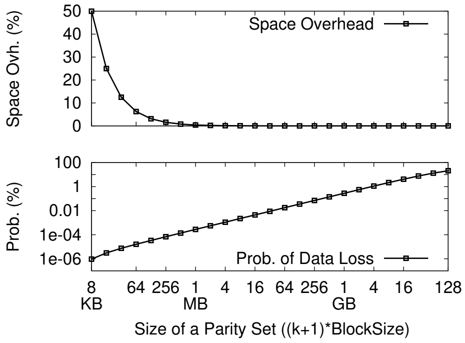

Figure 10: **Overhead and Reliability with Parity.** *The bottom graph shows the probabilty of data loss and the top graph the space overhead, as the size of the parity set is increased from 2 4-KB blocks (equivalent to mirroring) to one parity block for the entire disk. To compute probability of data loss, we focused on the roughly 1 in 20 ATA disks that exhibited latent sector errors; for those disks, the data in [4] reports that they exhibit roughly 0.01 errors per GB per 18 months, or a block failure rate* F_B *of* 2.54×10^{-8} *errors per block per year. Thus, if one has such a disk, the odds of at least one failure occurring is* $1 - P(NoFailure)$ *where* $P(NoFailure) = (1 - F_B)^N$ *on a disk of size N. For a 100 GB disk, this implies a 63% chance of data loss. A similar analysis is applied to arrive at the bottom graph above, but assuming one must have 2 (or more) failures within a parity set to achieve data loss. Note that our analysis assumes that latent sector errors are independent.*

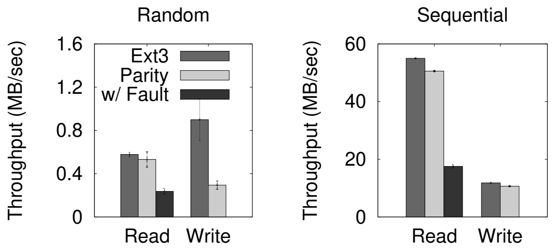

Figure 11: **Parity Throughput.** *The figure plots the throughput of the parity policy under some simple microbenchmarks. For sequential writes, we simply write 24 MB to disk. For random reads and writes, we either read or update random 4-KB blocks in a large (2 GB) file. For reads, both the normal and failure cases are reported; failures are injected by causing each initial read to fail which triggers reconstruction. Each experiment is repeated 60 times; averages and standard deviations are reported.*

file system block (the leftmost point) is equivalent to mirroring and wastes half the disk. A reasonable trade-off is found with parity sets between about 44 KB and 1 MB ($k = 10$ and $k = 255$); in this range, the space overhead is reasonable (*i.e.*, less than 10%) while the probability of loss is small (*i.e.*, less than 0.001%). In the rest of our parity policies, we use parity sets of $k = 10$ blocks.

Adding parity protection to a file system can have a large impact on performance. Figure 11 shows the performance of the parity policy for sequential and random access workloads that are either read or write intensive. The first graph shows, given no faults, that random reads perform well; however, random updates are quite slow. This result is not surprising, since each random update requires reading the old data and parity and writing the new data and parity; on a single disk, there is no overlap of these I/Os and hence the poor performance. The second graph shows that when there are no faults, the performance impact of parity blocks on sequential I/O is minimal, whether performing reads or writes. The parity policy

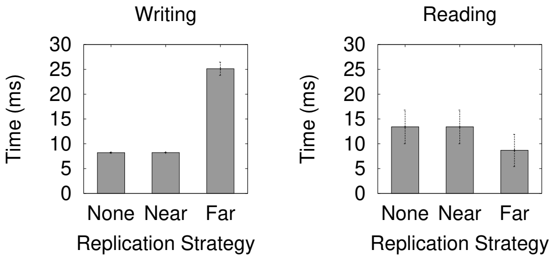

Figure 12: **Mirroring: Layout and Scheduling.** *The leftmost graph shows the average cost of writing a small file (4 KB) synchronously to disk, under three different replication strategies. The rightmost graph shows the average cost of reading a random 4 KB block alternately from two files. Different replication strategies are used; "None" indicates no replication, "Near" that replicas are placed as close to the original as possible, and "Far" that replicas are placed approximately 20 GB away).*

code optimizes sequential write performance by buffering multiple updates to a parity block and then flushing the parity block in a chained transaction. Finally, given a latent sector error on each initial read, read performance is significantly slower because the data must be reconstructed; however, this is (hopefully) a rare case.

In summary, CrookFS can be used to add parity protection to file systems. Although parity protection can incur a high performance cost for random update-intensive workloads (*e.g.*, TPC-B in Table 2), it still adds little overhead in many cases. We believe that parity protection should be considered for desktop file systems, since it enables important data to be stored reliably even in the presence of problematic disks.

5.5 Mirroring

For parity protection, we assumed that the parity location was determined when the file system was created. However, improve performance or reliability, more sophisticated policies may wish to control the location of redundant information on disk. We explore this issue in the context of a policy that mirrors user data blocks. The code for this policy has been presented (§3.3); implementing it requires 18 statements, as shown in Table 3.

We first examine the cost of mirroring during writes. The leftmost graph of Figure 12 presents the results of a simple experiment that repeatedly writes a small 4 KB block to disk synchronously. Three approaches are compared. The first approach does not mirror the block (None); the second does so but places the copy as near to the original as possible (Near); the third places the copy as far away as possible (Far). As one can see, placing the copy nearby is nearly free, whereas placing the blocks far away exacts a high performance cost (a seek and a rotation).

However, when reading back data from disk, spreading mirrors across the disk surface can improve performance [19, 39]. The rightmost graph of the figure shows an experiment in which a process reads a random block alternately from each of two files placed on opposite ends of the disk. Without replication (None), performance is poor, incurring high seek costs. With the file replica near its original (Near), there is also no benefit, as expected. Finally, with replicas far away, read performance improves dramatically: the scheduler is free to pick the copy to read from, reducing access time by nearly a factor of two.

In summary, the best choice for mirror locations is highly nuanced and depends on the workload. As expected, when the workload contains a significant percentage of metadata operations, performance suffers with mirroring, regardless of the mirror location (*e.g.*, the PostMark and TPC-B workloads shown in Table 2). However, in other cases, the location does matter. If spatially localized faults are likely, or read operations dominate (*e.g.*, in a transactional workload), the Far replication strategy is most appropriate; however, if data write performance is more critical (*e.g.*, in an archival scenario), the Near strategy may be the best choice. In any case, CrookFS can be used to dynamically choose different block layouts within a reliability policy.

5.6 Sanity Checks

Our next policy demonstrates that CrookFS allows different reliability mechanisms to be applied to different block types. For example, different sanity checks can be applied to different block types; we have currently implemented sanity checking of inodes.

Sanity checking detects whether a data structure has been corrupted by comparing each field of the data structure to its possible values. For example, to sanity check an inode, the mode of an inode is compared to all possible modes and pointers to data blocks (*i.e.*, block numbers) are forced to point within the valid range. The drawback of sanity checks are that they cannot detect bit corruption that does not lead to invalid values (*e.g.*, a data block pointer that is shifted by one is considered valid as long as it points within the valid range).

Table 3 shows that sanity checks require only 10 statements of policy code, since the I/O shepherd contains the corresponding primitive. To evaluate the performance of inode sanity checking, we constructed two inode-intensive workloads: the first reads one million inodes sequentially while the seconds reads 5000 inodes in a random order. Our measurements (not shown) reveal that sanity checking incurs no measurable overhead relative to the baseline Propagate policy, since the sanity checks are performed at the speed of the CPU and require no additional disk accesses. As expected, sanity checks also add no overhead to the three workloads presented in Table 2.

In conclusion, given that sanity checking has no performance penalty, we believe all file systems should sanity check data structures; we note that sanity checking can be performed in addition to other mechanisms for detecting corruption, such as checksumming. Although file systems such as ext3 do contain some sanity checks, it is currently done in an *ad hoc* manner and is diffused throughout the code base. Due to the centralized architecture of I/O shepherding, CrookFS can guarantee that each block is properly sanity checked before being accessed.

5.7 Multiple Levels of Defense

We next demonstrate the use of multiple data protection mechanisms within a single policy. Specifically, the multiple levels of defense policy uses checksums and replication to protect against data corruption. Further, for certain block types, the policy employs repair routines when a structure does not pass a checksum match but looks mostly "OK" (*e.g.*, all fields in an inode are valid except time fields). Finally, if all of these attempts fail to repair metadata inconsistencies, the system unlocks the block, queues any pending requests, runs `fsck`, and then remounts and begins running again. As indicated in Table 3, the multiple levels of defense policy is one of the more complex policies, requiring 39 lines of code.

Figure 13 shows the activity over time in a system employing this policy for four different fault injection scenarios; in each case, the workload consists of reading a single inode. The topmost part of the timeline shows what happens when there are no disk faults: the inode and its checksum are read from disk and the checksums match, as desired. In the next experiment, we inject a single disk fault, corrupting one inode; in this case, when the policy sees that the checksums do not match, it reads the alternate inode which

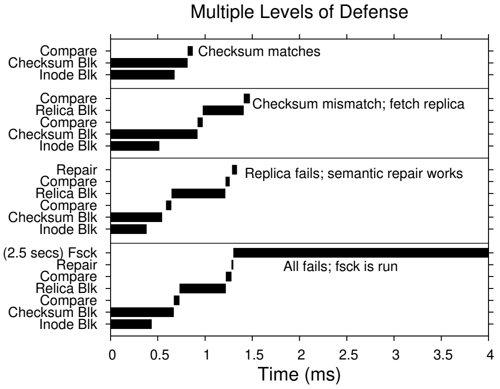

Figure 13: **A Multi-Level Policy.** *The figure shows four different runs of the multiple lines of defense policy. From top to bottom, each experiment induces a new fault and the y-axis highlights which action the system takes. These experiments use UML, which impacts absolute timing.*

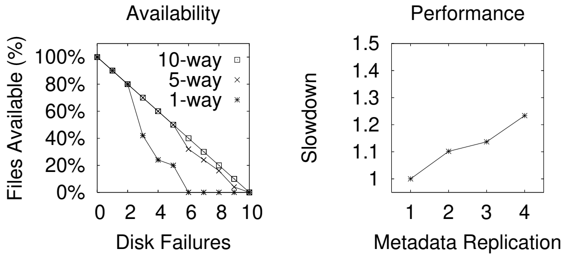

Figure 14: **D-GRAID Availability and Performance.** *The graphs show the availability and performance of D-GRAID on a workload creating 1000 4-KB files. On the left, each line varies the number of metadata replicas, while increasing the number of injected disk failures along the x-axis up to the full size of an emulated 10-disk array. The y-axis plots the percentage of files available. On the right, performance on four disks is shown as the number of metadata replicas increases; the y-axis shows slowdown compared to a single copy.*

matches, as desired. In the third, we perform a small corruption of both copies of the inode; here, the policy finds that neither inode's calculated checksum matches the stored checksum, but finds that the inode looks mostly intact and can be repaired simply (*e.g.*, clears a non-zero dtime because the inode is in use). In our final experiment, we corrupt both copies of the inode more drastically. In this case, all of these steps fail to fix the problem, and the policy runs the full `fsck`; when this action completes, the file system remounts and continues serving requests (not shown).

The performance overhead of adding multiple levels of defense for inode blocks is summarized in Table 2. Given no faults, the basic overheads of this policy are to verify and update the inode checksums and to update the inode replicas. Although updating on-disk checksums and replicas is costly, performing multiple levels of defense has a smaller performance penalty than some other policies since the checks are applied only to inode blocks.

5.8 D-GRAID

To demonstrate the full flexibility of CrookFS, we consider a fine-grained reliability policy that enacts different policies for different on-disk data types. We explore this policy given multiple disks. In this final policy, we implement D-GRAID style replication within the file system [32]. In D-GRAID, directories are widely replicated across many disks and a copy of each file (*i.e.*, its inode, all data blocks, and any indirect blocks) is mirrored and isolated within a disk boundary. This strategy ensures graceful degradation in that the failure of a disk does not render all of the data on the array unavailable.

With shepherding, the D-GRAID policy is straightforward to implement. First, the policy code for important metadata (such as directories and the superblock) specifies a high degree of replication. Second, the policy for inode, data, and indirect blocks specifies that a mirror copy of each block should be dynamically allocated to a particular disk. As indicated in Table 3, the D-GRAID policy requires 79 statements; although this is more than any of the other policies, it is significantly less than was required for the original D-GRAID implementation [32].

Figure 14 presents the availability and performance of CrookFS D-GRAID. The leftmost graph shows the availability benefit: with a high degree of metadata replication, most files remain available even when multiple disks fail. The rightmost graph shows performance overhead; as the amount of metadata replication is increased from one to four, the time for a synchronous write (and thus metadata-intensive) benchmark increases by 25%.

In conclusion, CrookFS is particularly interesting given multiple disks since it enables the file system to add reliability features across the disks without a separate volume manager (much like ZFS [33]). Due to the ability of CrookFS to enact different policies for different block types, we are able to implement even relatively complex reliability policies, such as D-GRAID.

6. RELATED WORK

The philosophy behind I/O Shepherding is similar to work on aspect-oriented programming, stackable file systems, the congestion manager and Click. We discuss each in turn.

Aspect-oriented programming [9, 23] addresses the general issue that code to implement certain high-level properties (*e.g.*, "performance") is *scattered* throughout systems, much as we observed that fault-handling is often diffused through a file system. Aspect-oriented programming provides language-level assistance in implementing these "crosscutting concerns," by allowing the relevant code to be described in a single place and then "weaved" into the code with an aspect compiler. Thus, one could consider building I/O Shepherding with aspects; however, the degree of integration required with OS subsystems could make this quite challenging.

Stackable file systems provide a more flexible way of configuring file systems from a set of composable layers [18, 40]. An I/O Shepherd could be viewed as a stackable layer that sits underneath the file system proper; however, we modified many system components to build the shepherd, breaking the encapsulation that stacking implies.

We drew inspiration from both the Congestion Manager (CM) [1, 5] and Click [27]. CM centralizes information about network congestion within the OS, thus enabling multiple network flows to utilize this knowledge and improve their behavior; in a similar manner, the I/O Shepherd centralizes both information and control and thus improves file system reliability. Click is a modular system for assembling customized routers [27]. We liked the clarity of Click router configurations from basic elements, and as a result there are parallels in our policies and primitives.

We also note that chained transactions are similar to *compensating transactions* in the database literature [25]; both deal with the case where committed transactions are treated as irreversible, and yet there is a need to change them. In databases, this situation commonly arises when an event external to the transactional setting occurs (*e.g.*, a customer returns a purchase); in our case, we use chained transactions in a much more limited manner, specifically to handle unexpected disk failures during checkpointing.

7. CONCLUSION

In this paper, we have described a flexible approach to reliability in file systems. I/O Shepherding provides a way to tailor reliability features to fit the needs of applications and the demands of the environment. Through its basic design, shepherding makes sophisticated policies simple to describe; through careful integration with the rest of the system, shepherding implements policies efficiently and correctly.

7.1 Porting the Shepherd

Our intention is to eventually specify a general shepherding layer of which all file systems can take advantage. Similar to other interfaces internal to the OS [24], our goal is to enable multiple file systems to leverage the same functionality.

Currently, we are porting shepherding to Linux ext2 and ReiserFS. The ext2 port has been straightforward, as it is simply a non-journaling version of ext3. Thus, we removed all consistency management code and embrace the ext2 philosophy of writing blocks to disk in any order. An additional `fsck`-like pass before mounting (not yet implemented) could rectify inconsistencies if so desired.

ReiserFS has been more challenging as it utilizes entirely different on-disk structures than the ext family. Thus far, we have successfully built simple policies and are working on integration with ReiserFS consistency management (we are using Reiser3 data journaling mode). Through this work, we are slowly gaining confidence about the general applicability of the shepherding approach.

7.2 Lessons

Adding reliability through the I/O shepherd was simple in some ways and challenging in others. In the process of building the environment, we have learned a number of lessons.

- *Interposition simplifies fault management.* One of the most powerful aspects of I/O Shepherding is its basic design: the shepherd interposes on all I/O and thus can implement a reliability policy consistently and correctly. Expecting reliability from code that is scattered throughout is unrealistic.
- *Block-level interposition can make things difficult.* The I/O shepherd interposes on block reads and writes that the file system issues. While natural for many policies (*e.g.*, replicating blocks of a particular type), block-level interposition makes some kinds of policies more difficult to implement. For example, implementing stronger sanity checks on directory contents (which span many blocks) is awkward at best. Perhaps a higher-level storage system interface would provide a better interposition target.
- *Shepherding need not be costly.* The shepherd is responsible for the execution of all I/O requests in the system. Careful integration with other subsystems is essential in achieving low overheads, with particular attention paid to the concurrency management infrastructure.
- *Good policies stem from good information.* Although not the main focus of this paper, shaping an appropriate reliability policy clearly requires accurate data on how the disks the system is using actually fail as well as the nature of the workloads that run on the system. Fortunately, more data is becoming available on the true nature of disk faults [4, 31]; systems that deploy I/O shepherding may also need to incorporate a fault and workload monitoring infrastructure to gather the requisite information.
- *Fault propagation is important (and yet often buggy).* An I/O shepherd can mask a large number of faults, depending on the exact policy specified; however, if a fault is returned, the file system above the shepherd is responsible for passing the error to the calling application. We have found many bugs in error propagation, and are currently working on a more general analysis of file systems to better understand (and fix) their error propagation behavior.
- *Fault handling in journaling file systems is challenging.* By its nature, write-ahead logging places intentions on disks; reactive fault handling by its nature must behave reasonably when these intentions cannot be met. Chained transactions help to overcome this inherent difficulty, but at the cost of complexity (certainly it is the most complex part of our code). Alternate simpler approaches would be welcome.

This last lesson will no doubt inform our future work. We are particularly interested in the question of whether future storage systems should employ journaling or shadow paging [15]; there are many reliability trade-offs in these approaches that are not yet well understood. Further work will be required to provide a deeper level of insight on this problem.

Acknowledgments

We thank the anonymous reviewers and Frans Kaashoek (the I/O shepherd's shepherd) for their tremendous feedback, which caused us to rethink and rework many aspects of both our presentation and technical content. We would also like to thank Nitin Agrawal for his earlier work and feedback on this project, and the CSL at Wisconsin for their tireless assistance.

This material is based upon work supported by the National Science Foundation under the following grants: CCF-0621487, CNS-0509474, CCR-0133456, as well as by generous donations from Network Appliance and Sun Microsystems.

Haryadi S. Gunawi was awarded an SOSP student travel scholarship, supported by Infosys, to present this paper at the conference. Swetha Krishnan was awarded an SOSP student travel scholarship, supported by Hewlett-Packard.

Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of NSF or other institutions.

8. REFERENCES

[1] David G. Andersen, Deepak Bansal, Dorothy Curtis, Srinivasan Seshan, and Hari Balakrishnan. System Support for Bandwidth Management and Content Adaptation in Internet Applications. In *OSDI '00*, pages 213–226, San Diego, CA, October 2000.

[2] Dave Anderson, Jim Dykes, and Erik Riedel. More Than an Interface: SCSI vs. ATA. In *FAST '03*, San Francisco, CA, April 2003.

[3] Lakshmi Bairavasundaram. On the frequency of transient faults in modern disk drives. Personal Communication, 2007.

[4] Lakshmi Bairavasundaram, Garth Goodson, Shankar Pasupathy, and Jiri Schindler. An Analysis of Latent Sector Errors in Disk Drives. In *SIGMETRICS '07*, pages 289–300, San Diego, CA, June 2007.

[5] Hari Balakrishnan, Hariharan S. Rahul, and Srinivasan Seshan. An Integrated Congestion Management Architecture for Internet Hosts. In *SIGCOMM '99*, pages 175–187, Cambridge, MA, August 1999.

[6] Wendy Bartlett and Lisa Spainhower. Commercial Fault Tolerance: A Tale of Two Systems. *IEEE Transactions on Dependable and Secure Computing*, 1(1):87–96, January 2004.

[7] Aaron B. Brown and David A. Patterson. Undo for Operators: Building an Undoable E-mail Store. In *USENIX '03*, San Antonio, TX, June 2003.

[8] Andy Chou, Jun-Feng Yang, Benjamin Chelf, Seth Hallem, and Dawson Engler. An Empirical Study of Operating System Errors. In *SOSP '01*, pages 73–88, Banff, Canada, October 2001.

[9] Yvonne Coady, Gregor Kiczales, Mike Feeley, and Greg Smolyn. Using AspectC to Improve the Modularity of Path-Specific Customization in Operating System Code. In *ESEC/FSE-9*, September 2001.

[10] Peter Corbett, Bob English, Atul Goel, Tomislav Grcanac, Steven Kleiman, James Leong, and Sunitha Sankar. Row-Diagonal Parity for Double Disk Failure Correction. In *FAST '04*, pages 1–14, San Francisco, CA, April 2004.

[11] Timothy E. Denehy, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Bridging the Information Gap in Storage Protocol Stacks. In *USENIX '02*, pages 177–190, Monterey, CA, June 2002.

[12] Dawson Engler, David Yu Chen, Seth Hallem, Andy Chou, and Benjamin Chelf. Bugs as Deviant Behavior: A General Approach to Inferring Errors in Systems Code. In *SOSP '01*, pages 57–72, Banff, Canada, October 2001.

[13] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The Google File System. In *SOSP '03*, pages 29–43, Bolton Landing, NY, October 2003.

[14] Jim Gray. A Census of Tandem System Availability Between 1985 and 1990. Technical Report 90.1, Tandem Computers, 1990.

[15] Jim Gray and Andreas Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann, 1993.

[16] Roedy Green. EIDE Controller Flaws Version 24. http://mindprod.com/jgloss/eideflaw.html, February 2005.

[17] Robert Hagmann. Reimplementing the Cedar File System Using Logging and Group Commit. In *SOSP '87*, Austin, TX, November 1987.

[18] John S. Heidemann and Gerald J. Popek. File-system development with stackable layers. *ACM Transactions on Computer Systems*, 12(1):58–89, 1994.

[19] Hai Huang, Wanda Hung, and Kang G. Shin. FS2: dynamic data replication in free disk space for improving disk performance and energy consumption. In *SOSP '05*, pages 263–276, Brighton, UK, October 2005.

[20] Gordon F. Hughes and Joseph F. Murray. Reliability and Security of RAID Storage Systems and D2D Archives Using SATA Disk Drives. *ACM Transactions on Storage*, 1(1):95–107, February 2005.

[21] Hannu H. Kari, H. Saikkonen, and F. Lombardi. Detection of Defective Media in Disks. In *The IEEE International Workshop on Defect and Fault Tolerance in VLSI Systems*, pages 49–55, Venice, Italy, October 1993.

[22] Jeffrey Katcher. PostMark: A New File System Benchmark. Technical Report TR-3022, Network Appliance Inc., October 1997.

[23] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-Oriented Programming. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, pages 220–242, 1997.

[24] Steve R. Kleiman. Vnodes: An Architecture for Multiple File System Types in Sun UNIX. In *USENIX Summer '86*, pages 238–247, Atlanta, GA, June 1986.

[25] Henry F. Korth, Eliezer Levy, and Abraham Silberschatz. A Formal Approach to Recovery by Compensating Transactions. In *VLDB 16*, pages 95–106, Brisbane, Australia, August 1990.

[26] Larry Lancaster and Alan Rowe. Measuring Real World Data Availability. In *Proceedings of the LISA 2001 15th Systems Administration Conference*, pages 93–100, San Diego, California, December 2001.

[27] Robert Morris, Eddie Kohler, John Jannotti, and M. Frans Kaashoek. The Click Modular Router. In *SOSP '99*, pages 217–231, Kiawah Island Resort, SC, December 1999.

[28] Kiran Nagaraja, Fabio Olivera, Ricardo Bianchini, Richard P. Martin, and Thu D. Nguyen. Understanding and Dealing with Operator Mistakes in Internet Services. In *OSDI '04*, San Francisco, CA, December 2004.

[29] David Patterson, Garth Gibson, and Randy Katz. A Case for Redundant Arrays of Inexpensive Disks (RAID). In *SIGMOD '88*, pages 109–116, Chicago, IL, June 1988.

[30] Vijayan Prabhakaran, Lakshmi N. Bairavasundaram, Nitin Agrawal, Haryadi S. Gunawi, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. IRON File Systems. In *SOSP '05*, pages 206–220, Brighton, UK, October 2005.

[31] Bianca Schroeder and Garth Gibson. Disk failures in the real world: What does an MTTF of 1,000,000 hours mean to you? In *FAST '07*, pages 1–16, San Jose, CA, February 2007.

[32] Muthian Sivathanu, Vijayan Prabhakaran, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Improving Storage System Availability with D-GRAID. In *FAST '04*, pages 15–30, San Francisco, CA, April 2004.

[33] Sun Microsystems. ZFS: The last word in file systems. www.sun.com/2004-0914/feature/, 2006.

[34] Rajesh Sundaram. The Private Lives of Disk Drives. http://www.netapp.com/go/techontap/matl/sample/0206tot_resiliency.html, February 2006.

[35] Michael M. Swift, Brian N. Bershad, and Henry M. Levy. Improving the Reliability of Commodity Operating Systems. In *SOSP '03*, Bolton Landing, NY, October 2003.

[36] Nisha Talagala and David Patterson. An Analysis of Error Behaviour in a Large Storage System. In *The IEEE Workshop on Fault Tolerance in Parallel and Distributed Systems*, San Juan, Puerto Rico, April 1999.

[37] Transaction Processing Council. TPC Benchmark B Standard Specification, Revision 3.2. Technical Report, 1990.

[38] Stephen C. Tweedie. Journaling the Linux ext2fs File System. In *The Fourth Annual Linux Expo*, Durham, North Carolina, May 1998.

[39] X. Yu, B. Gum, Y. Chen, R. Y. Wang, K. Li, A. Krishnamurthy, and T. E. Anderson. Trading Capacity for Performance in a Disk Array. In *OSDI '00*, San Diego, CA, October 2000.

[40] Erez Zadok and Jason Nieh. FiST: A Language for Stackable File Systems. In *USENIX '00*, pages 55–70, San Diego, CA, June 2000.