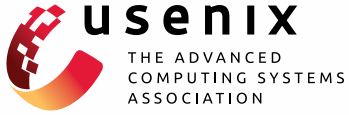

Physical Disentanglement in a Container-Based File System

Lanyue Lu, Yupu Zhang, Thanh Do, Samer Al-Kiswany, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau, *University of Wisconsin—Madison*

https://www.usenix.org/conference/osdi14/technical-sessions/presentation/lu

This paper is included in the Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation.

October 6–8, 2014 • Broomfield, CO

978-1-931971-16-4

Open access to the Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation is sponsored by USENIX.

Physical Disentanglement in a Container-Based File System

Lanyue Lu, Yupu Zhang, Thanh Do, Samer Al-Kiswany
Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau

Department of Computer Sciences
University of Wisconsin, Madison
{ll, yupu, thanhdo, samera, dusseau, remzi}@cs.wisc.edu

Abstract

We introduce IceFS, a novel file system that separates physical structures of the file system. A new abstraction, the *cube*, is provided to enable the grouping of files and directories inside a physically isolated container. We show three major benefits of cubes within IceFS: localized reaction to faults, fast recovery, and concurrent file-system updates. We demonstrate these benefits within a VMware-based virtualized environment and within the Hadoop distributed file system. Results show that our prototype can significantly improve availability and performance, sometimes by an order of magnitude.

1 Introduction

Isolation is central to increased reliability and improved performance of modern computer systems. For example, isolation via virtual address space ensures that one process cannot easily change the memory state of another, thus causing it to crash or produce incorrect results [10].

As a result, researchers and practitioners alike have developed a host of techniques to provide isolation in various computer subsystems: Verghese et al. show how to isolate performance of CPU, memory, and disk bandwidth in SGI's IRIX operating system [58]; Gupta et al. show how to isolate the CPU across different virtual machines [26]; Wachs et al. invent techniques to share storage cache and I/O bandwidth [60]. These are but three examples; others have designed isolation schemes for device drivers [15, 54, 61], CPU and memory resources [2, 7, 13, 41], and security [25, 30, 31].

One aspect of current system design has remained devoid of isolation: the physical on-disk structures of file systems. As a simple example, consider a bitmap, used in historical systems such as FFS [37] as well as many modern file systems [19, 35, 56] to track whether inodes or data blocks are in use or free. When blocks from different files are allocated from the same bitmap, aspects of their reliability are now *entangled*, i.e., a failure in that bitmap block can affect otherwise unrelated files. Similar entanglements exist at all levels of current file systems; for example, Linux Ext3 includes all current update activity into a single global transaction [44], leading to painful and well-documented performance problems [4, 5, 8].

The surprising entanglement found in these systems arises from a central truth: logically-independent file system entities are not physically independent. The result is poor reliability, poor performance, or both.

In this paper, we first demonstrate the root problems caused by physical entanglement in current file systems. For example, we show how a single disk-block failure can lead to global reliability problems, including system-wide crashes and file system unavailability. We also measure how a lack of physical disentanglement slows file system recovery times, which scale poorly with the size of a disk volume. Finally, we analyze the performance of unrelated activities and show they are linked via crash-consistency mechanisms such as journaling.

Our remedy to this problem is realized in a new file system we call *IceFS*. IceFS provides users with a new basic abstraction in which to co-locate logically similar information; we call these containers *cubes*. IceFS then works to ensure that files and directories within cubes are physically distinct from files and directories in other cubes; thus data and I/O within each cube is *disentangled* from data and I/O outside of it.

To realize disentanglement, IceFS is built upon three core principles. First, there should be no shared physical resources across cubes. Structures used within one cube should be distinct from structures used within another. Second, there should be no access dependencies. IceFS separates key file system data structures to ensure that the data of a cube remains accessible regardless of the status of other cubes; one key to doing so is a novel *directory indirection* technique that ensures cube availability in the file system hierarchy despite loss or corruption of parent directories. Third, there should be no bundled transactions. IceFS includes novel *transaction splitting* machinery to enable concurrent updates to file system state, thus disentangling write traffic in different cubes.

One of the primary benefits of cube disentanglement is *localization*: negative behaviors that normally affect all file system clients can be localized within a cube. We demonstrate three key benefits that arise directly from

such localization. First, we show how cubes enable localized *micro-failures*; panics, crashes, and read-only remounts that normally affect the entire system are now constrained to the faulted cube. Second, we show how cubes permit localized *micro-recovery*; instead of an expensive file-system wide check and repair, the disentanglement found at the core of cubes enables IceFS to fully (and quickly) repair a subset of the file system (and even do so online), thus minimizing downtime and increasing availability. Third, we illustrate how transaction splitting allows the file system to commit transactions from different cubes in parallel, greatly increasing performance (by a factor of 2x–5x) for some workloads.

Interestingly, the localization that is innate to cubes also enables a new benefit: *specialization* [17]. Because cubes are independent, it is natural for the file system to tailor the behavior of each. We realize the benefits of specialization by allowing users to choose different journaling modes per cube; doing so creates a performance/consistency knob that can be set as appropriate for a particular workload, enabling higher performance.

Finally, we further show the utility of IceFS in two important modern storage scenarios. In the first, we use IceFS as a host file system in a virtualized VMware [59] environment, and show how it enables fine-grained fault isolation and fast recovery as compared to the state of the art. In the second, we use IceFS beneath HDFS [49], and demonstrate that IceFS provides failure isolation between clients. Overall, these two case studies demonstrate the effectiveness of IceFS as a building block for modern virtualized and distributed storage systems.

The rest of this paper is organized as follows. We first show in Section 2 that the aforementioned problems exist through experiments. Then we introduce the three principles for building a disentangled file system in Section 3, describe our prototype IceFS and its benefits in Section 4, and evaluate IceFS in Section 5. Finally, we discuss related work in Section 6 and conclude in Section 7.

2 Motivation

Logical entities, such as directories, provided by the file system are an illusion; the underlying physical entanglement in file system data structures and transactional mechanisms does not provide true isolation. We describe three problems that this entanglement causes: global failure, slow recovery, and bundled performance. After discussing how current approaches fail to address them, we describe the negative impact on modern systems.

2.1 Entanglement Problems

2.1.1 Global Failure

Ideally, in a robust system, a fault involving one file or directory should not affect other files or directories, the remainder of the OS, or other users. However, in current file systems, a single fault often leads to a *global failure*.

Global Failures	Ext3	Ext4	Btrfs
Crash	129	341	703
Read-only	64	161	89

Table 1: **Global Failures in File Systems.** *This table shows the average number of crash and read-only failures in Ext3, Ext4, and Btrfs source code across 14 versions of Linux (3.0 to 3.13).*

Fault Type	Ext3	Ext4
Metadata read failure	70 (66)	95 (90)
Metadata write failure	57 (55)	71 (69)
Metadata corruption	25 (11)	62 (28)
Pointer fault	76 (76)	123 (85)
Interface fault	8 (1)	63 (8)
Memory allocation	56 (56)	69 (68)
Synchronization fault	17 (14)	32 (27)
Logic fault	6 (0)	17 (0)
Unexpected states	42 (40)	127 (54)

Table 2: **Failure Causes in File Systems.** *This table shows the number of different failure causes for Ext3 and Ext4 in Linux 3.5, including those caused by entangled data structures (in parentheses). Note that a single failure instance may have multiple causes.*

A common approach for handling faults in current file systems is to either *crash* the entire system (e.g., by calling `BUG_ON`, `panic`, or `assert`) or to mark the whole file system *read-only*. Crashes and read-only behavior are not constrained to only the faulty part of the file system; instead, a global reaction is enforced for the whole system. For example, Btrfs crashes the entire OS when it finds an invariant is violated in its extent tree; Ext3 marks the whole file system as read-only when it detects a corruption in a single inode bitmap. To illustrate the prevalence of these coarse reactions, we analyzed the source code and counted the average number of such global failure instances in Ext3 with JBD, Ext4 with JBD2, and Btrfs from Linux 3.0 to 3.13. As shown in Table 1, each file system has hundreds of invocations to these poor global reactions.

Current file systems trigger global failures to react to a wide range of system faults. Table 2 shows there are many root causes: metadata failures and corruptions, pointer faults, memory allocation faults, and invariant faults. These types of faults exist in real systems [11, 12, 22, 33, 42, 51, 52], and they are used for fault injection experiments in many research projects [20, 45, 46, 53, 54, 61]. Responding to these various faults in a non-global manner is non-trivial; the table shows that a high percentage (89% in Ext3, 65% in Ext4) of these faults are caused by entangled data structures (e.g., bitmaps and transactions).

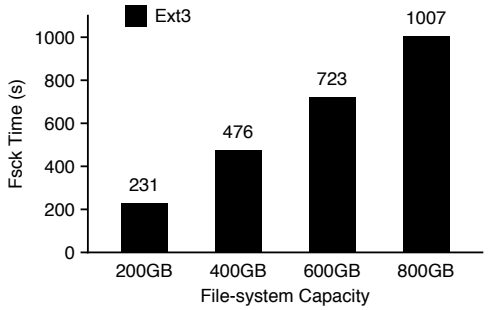

Figure 1: **Scalability of E2fsck on Ext3.** *This figure shows the fsck time on Ext3 with different file-system capacity. We create the initial file-system image on partitions of different capacity (x-axis). We make 20 directories in the root directory and write the same set of files to every directory. As the capacity changes, we keep the file system at 50% utilization by varying the amount of data in the file set.*

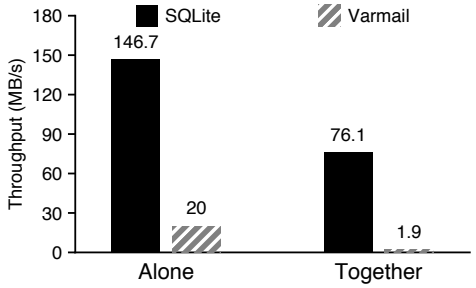

Figure 2: **Bundled Performance on Ext3.** *This figure shows the performance of running SQLite and Varmail on Ext3 in ordered mode. The SQLite workload, configured with write-ahead logging, asynchronously writes 40KB values in sequential key order. The Varmail workload involves 16 threads, each of which performs a series of create-append-sync and read-append-sync operations.*

2.1.2 Slow Recovery

After a failure occurs, file systems often rely on an offline file-system checker to recover [39]. The checker scans the whole file system to verify the consistency of metadata and repair any observed problems. Unfortunately, current file system checkers are *not scalable*: with increasing disk capacities and file system sizes, the time to run the checker is unacceptably long, decreasing availability. For example, Figure 1 shows that the time to run a checker [55] on an Ext3 file system grows linearly with the size of the file system, requiring about 1000 seconds to check an 800GB file system with 50% utilization. Ext4 has better checking performance due to its layout optimization [36], but the checking performance is similar to Ext3 after aging and fragmentation [34].

Despite efforts to make checking faster [14, 34, 43], check time is still constrained by file system size and disk bandwidth. The root problem is that current checkers are *pessimistic*: even though there is only a small piece of corrupt metadata, the entire file system is checked. The main reason is that due to entangled data structures, it is hard or even impossible to determine which part of the file system needs checking.

2.1.3 Bundled Performance and Transactions

The previous two problems occur because file systems fail to isolate metadata structures; additional problems occur because the file system journal is a shared, global data structure. For example, Ext3 uses a generic journaling module, JBD, to manage updates to the file system. To achieve better throughput, instead of creating a separate transaction for every file system update, JBD groups all updates within a short time interval (e.g., 5s) into a single global transaction; this transaction is then committed periodically or when an application calls `fsync()`.

Unfortunately, these bundled transactions cause the performance of independent processes to be bundled. Ideally, calling `fsync()` on a file should flush only the dirty data belonging to that particular file to disk; unfortunately, in the current implementation, calling `fsync()` causes unrelated data to be flushed as well. Therefore, the performance of write workloads may suffer when multiple applications are writing at the same time.

Figure 2 illustrates this problem by running a database application SQLite [9] and an email server workload Varmail [3] on Ext3. SQLite sequentially writes large key/value pairs asynchronously, while Varmail frequently calls `fsync()` after small random writes. As we can see, when these two applications run together, both applications' performance degrades significantly compared with running alone, especially for Varmail. The main reason is that both applications share the same journaling layer and each workload affects the other. The `fsync()` calls issued by Varmail must wait for a large amount of data written by SQLite to be flushed together in the same transaction. Thus, the single shared journal causes performance entanglement for independent applications in the same file system. Note that we use an SSD to back the file system, so device performance is not a bottleneck in this experiment.

2.2 Limitations of Current Solutions

One popular approach for providing isolation in file systems is through the namespace. A namespace defines a subset of files and directories that are made visible to an application. Namespace isolation is widely used for better security in a shared environment to constrain different applications and users. Examples include virtual machines [16, 24], Linux containers [2, 7], `chroot`, BSD `jail` [31], and Solaris Zones [41].

However, these abstractions fail to address the problems mentioned above. Even though a namespace can restrict application access to a subset of the file system, files from different namespaces still share metadata, sys-

tem states, and even transactional machinery. As a result, a fault in any shared structure can lead to a global failure; a file-system checker still must scan the whole file system; updates from different namespaces are bundled together in a single transaction.

Another widely-used method for providing isolation is through static disk partitions. Users can create multiple file systems on separate partitions. Partitions are effective at isolating corrupted data or metadata such that read-only failure can be limited to one partition, but a single `panic()` or `BUG_ON()` within one file system may crash the whole OS, affecting all partitions. In addition, partitions are not flexible in many ways and the number of partitions is usually limited. Furthermore, storage space may not be effectively utilized and disk performance may decrease due to the lack of a global block allocation. Finally, it can be challenging to use and manage a large number of partitions across different file systems and applications.

2.3 Usage Scenarios

Entanglement in the local file system can cause significant problems to higher-level services like virtual machines and distributed file systems. We now demonstrate these problems via two important cases: a virtualized storage environment and a distributed file system.

2.3.1 Virtual Machines

Fault isolation within the local file system is of paramount importance to server virtualization environments. In production deployments, to increase machine utilization, reduce costs, centralize management, and make migration efficient [23, 48, 57], tens of virtual machines (VMs) are often consolidated on a single host machine. The virtual disk image for each VM is usually stored as a single or a few files within the host file system. If a single fault triggered by one of the virtual disks causes the host file system to become read-only (e.g., metadata corruption) or to crash (e.g., assertion failures), then all the VMs suffer. Furthermore, recovering the file system using fsck and redeploying all VMs require considerable downtime.

Figure 3 shows how VMware Workstation 9 [59] running with an Ext3 host file system reacts to a read-only failure caused by one virtual disk image. When a read-only fault is triggered in Ext3, all three VMs receive an error from the host file system and are immediately shut down. There are 10 VMs in the shared file system; each VM has a preallocated 20GB virtual disk image. Although only one VM image has a fault, the entire host file system is scanned by e2fsck, which takes more than eight minutes. This experiment demonstrates that a single fault can affect multiple unrelated VMs; isolation across different VMs is not preserved.

2.3.2 Distributed File Systems

Physical entanglement within the local file system also negatively impacts distributed file systems, especially in multi-tenant settings. Global failures in local file systems manifest themselves as machine failures, which are handled by crash recovery mechanisms. Although data is not lost, fault isolation is still hard to achieve due to long timeouts for crash detection and the layered architecture. We demonstrate this challenge in HDFS [49], a popular distributed file system used by many applications.

Although HDFS provides fault-tolerant machinery such as replication and failover, it does not provide fault isolation for applications. Thus, applications (e.g., HBase [1, 27]) can only rely on HDFS to prevent data loss and must provide fault isolation themselves. For instance, in HBase multi-tenant deployments, HBase servers can manage tables owned by various clients. To isolate different clients, each HBase server serves a certain number of tables [6]. However, this approach does not provide complete isolation: although HBase servers are grouped based on tables, their tables are stored in HDFS nodes, which are not aware of the data they store. Thus, an HDFS server failure will affect multiple HBase servers and clients. Although indirection (e.g., HBase on HDFS) simplifies system management, it makes isolation in distributed systems challenging.

Figure 4 illustrates such a situation: four clients concurrently read different files stored in HDFS when a machine crashes; the crashed machine stores data blocks for all four clients. In this experiment, only the first client is fortunate enough to not reference this crashed node and thus finishes early. The other three lose throughput for 60 seconds before failing over to other nodes. Although data loss does not occur as data is replicated on multiple nodes in HDFS, this behavior may not be acceptable for latency-sensitive applications.

3 File System Disentanglement

To avoid the problems described in the previous section, file systems need to be redesigned to avoid artificial coupling between logical entities and physical realization. In this section, we discuss a key abstraction that enables such disentanglement: the file system cube. We then discuss the key principles underlying a file system that realizes disentanglement: no shared physical resources, no access dependencies, and no bundled transactions.

3.1 The Cube Abstraction

We propose a new file system abstraction, the *cube*, that enables applications to specify which files and directories are logically related. The file system can safely combine the performance and reliability properties of groups of files and their metadata that belong to the same cube; each cube is physically isolated from others and is thus

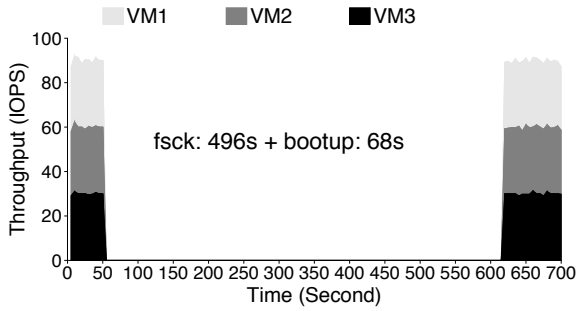

Figure 3: **Global Failure for Virtual Machines.** *This figure shows how a fault in Ext3 affects all three virtual machines (VMs). Each VM runs a workload that writes 4KB blocks randomly to a 1GB file and calls* `fsync()` *after every 10 writes. We inject a fault at 50s, run e2fsck after the failure, and reboot all three VMs.*

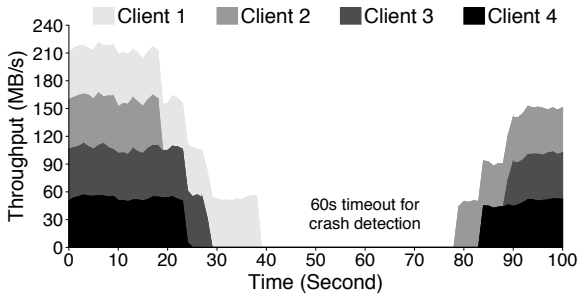

Figure 4: **Impact of Machine Crashes in HDFS.** *This figure shows the negative impact of physical entanglement within local file systems on HDFS. A kernel panic caused by a local file system leads to a machine failure, which negatively affects the throughput of multiple clients.*

completely independent at the file system level.

The cube abstraction is easy to use, with the following operations:

Create a cube: A cube can be created on demand. A default global cube is created when a new file system is created with the `mkfs` utility.

Set cube attributes: Applications can specify customized attributes for each cube. Supported attributes include: failure policy (e.g., read-only or crash), recovery policy (e.g., online or offline checking) and journaling mode (e.g., high or low consistency requirement).

Add files to a cube: Users can create or move files or directories into a cube. By default, files and directories inherit the cube of their parent directory.

Delete files from a cube: Files and directories can be removed from the cube via `unlink`, `rmdir`, and `rename`.

Remove a cube: An application can delete a cube completely along with all files within it. The released disk space can then be used by other cubes.

The cube abstraction has a number of attractive properties. First, each cube is *isolated* from other cubes both logically and physically; at the file system level, each cube is independent for failure, recovery, and journaling. Second, the use of cubes can be *transparent* to applications; once a cube is created, applications can interact with the file system without modification. Third, cubes are *flexible*; cubes can be created and destroyed on demand, similar to working with directories. Fourth, cubes are *elastic* in storage space usage; unlike partitions, no storage over-provision or reservation is needed for a cube. Fifth, cubes can be *customized* for diverse requirements; for example, an important cube may be set with high consistency and immediate recovery attributes. Finally, cubes are *lightweight*; a cube does not require extensive memory or disk resources.

3.2 Disentangled Data Structures

To support the cube abstraction, key data structures within modern file systems must be disentangled. We discuss three principles of disentangled data structures: no shared physical resources, no access dependencies, and no shared transactions.

3.2.1 No Shared Physical Resources

For cubes to have independent performance and reliability, multiple cubes must not share the same physical resources within the file system (e.g., blocks on disk or pages in memory). Unfortunately, current file systems freely co-locate metadata from multiple files and directories into the same unit of physical storage.

In classic Ext-style file systems, storage space is divided into fixed-size *block groups*, in which each block group has its own metadata (i.e., a group descriptor, an inode bitmap, a block bitmap, and inode tables). Files and directories are allocated to particular block groups using heuristics to improve locality and to balance space. Thus, even though the disk is partitioned into multiple block groups, any block group and its corresponding metadata blocks can be shared across any set of files. For example, in Ext3, Ext4 and Btrfs, a single block is likely to contain inodes for multiple unrelated files and directories; if I/O fails for one inode block, then all the files with inodes in that block will not be accessible. As another example, to save space, Ext3 and Ext4 store many group descriptors in one disk block, even though these group descriptors describe unrelated block groups.

This false sharing percolates from on-disk blocks up to in-memory data structures at runtime. Shared resources directly lead to global failures, since a single corruption or I/O failure affects multiple logically-independent files. Therefore, to isolate cubes, a disentangled file system must partition its various data structures into smaller independent ones.

3.2.2 No Access Dependency

To support independent cubes, a disentangled file system must also ensure that one cube does not contain references to or need to access other cubes. Current file systems often contain a number of data structures that violate this principle. Specifically, *linked lists* and *trees* encode dependencies across entries by design. For example, Ext3 and Ext4 maintain an orphan inode list in the super block to record files to be deleted; Btrfs and XFS use Btrees extensively for high performance. Unfortunately, one failed entry in a list or tree affects all entries following or below it.

The most egregious example of access dependencies in file systems is commonly found in the implementation of the *hierarchical directory structure*. In Ext-based systems, the path for reaching a particular file in the directory structure is implicitly encoded in the physical layout of those files and directories on disk. Thus, to read a file, all directories up to the root must be accessible. If a single directory along this path is corrupted or unavailable, a file will be inaccessible.

3.2.3 No Bundled Transactions

The final data structure and mechanism that must be disentangled to provide isolation to cubes are transactions. To guarantee the consistency of metadata and data, existing file systems typically use journaling (e.g., Ext3 and Ext4) or copy-on-write (e.g., Btrfs and ZFS) with transactions. A transaction contains temporal updates from many files within a short period of time (e.g., 5s in Ext3 and Ext4). A shared transaction batches multiple updates and is flushed to disk as a single atomic unit in which either all or none of the updates are successful.

Unfortunately, transaction batching artificially tangles together logically independent operations in several ways. First, if the shared transaction fails, updates to all of the files in this transaction will fail as well. Second, in physical journaling file systems (e.g., Ext3), a `fsync()` call on one file will force data from other files in the same transaction to be flushed as well; this falsely couples performance across independent files and workloads.

4 The Ice File System

We now present IceFS, a file system that provides cubes as its basic new abstraction. We begin by discussing the important internal mechanisms of IceFS, including novel directory independence and transaction splitting mechanisms. Disentangling data structures and mechanisms enables the file system to provide behaviors that are localized and specialized to each container. We describe three major benefits of a disentangled file system (localized reactions to failures, localized recovery, and specialized journaling performance) and how such benefits are realized in IceFS.

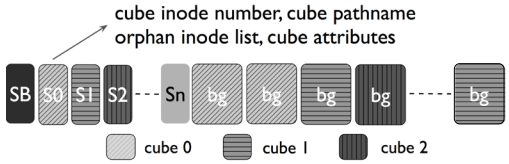

Figure 5: **Disk Layout of IceFS.** *This figure shows the disk layout of IceFS. Each cube has a sub-super block, stored after the global super block. Each cube also has its own separated block groups. Si: sub-super block for cube i; bg: a block group.*

4.1 IceFS

We implement a prototype of a disentangled file system, IceFS, as a set of modifications to Ext3, a standard and mature journaling file system in many Linux distributions. We disentangle Ext3 as a proof of concept; we believe our general design can be applied to other file systems as well.

4.1.1 Realizing the Cube Abstraction

The cube abstraction does not require radical changes to the existing POSIX interface. In IceFS, a cube is implemented as a special directory; all files and sub-directories within the cube directory belong to the same cube.

To create a cube, users pass a cube flag when they call `mkdir()`. IceFS creates the directory and records that this directory is a cube. When creating a cube, customized cube attributes are also supported, such as a specific journaling mode for different cubes. To delete a cube, only `rmdir()` is needed.

IceFS provides a simple mechanism for filesystem isolation so that users have the freedom to define their own policies. For example, an NFS server can automatically create a cube for the home directory of each user, while a VM server can isolate each virtual machine in its own cube. An application can use a cube as a data container, which isolates its own data from other applications.

4.1.2 Physical Resource Isolation

A straightforward approach for supporting cubes is to leverage the existing concept of a block group in many existing file systems. To disentangle shared resources and isolate different cubes, IceFS dictates that a block group can be assigned to only one cube at any time, as shown in Figure 5; in this way, all metadata associated with a block group (e.g., bitmaps and inode tables) belongs to only one cube. A block group freed by one cube can be allocated to any other cube. Compared with partitions, the allocation unit of cubes is only one block group, much smaller than the size of a typical multiple GB partition.

When allocating a new data block or an inode for a cube, the target block group is chosen to be either an empty block group or a block group already belonging to the cube. Enforcing the requirement that a block group

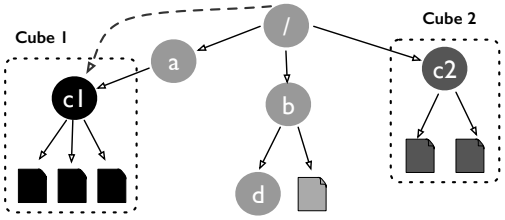

Figure 6: **An Example of Cubes and Directory Indirection.** *This figure shows how the cubes are organized in a directory tree, and how the directory indirection for a cube is achieved.*

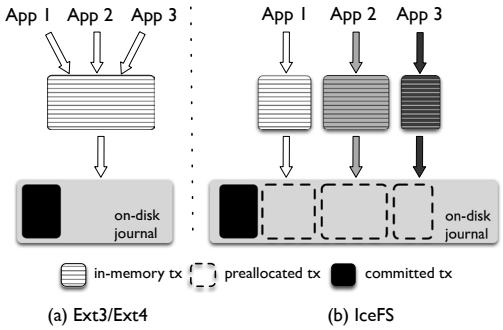

Figure 7: **Transaction Split Architecture.** *This figure shows the different transaction architectures in Ext3/4 and IceFS. In IceFS, different colors represent different cubes' transactions.*

is devoted to a single cube requires changing the file and directory allocation algorithms such that they are cube-aware without losing locality.

To identify the cube of a block group, IceFS stores a cube ID in the group descriptor. To get the cube ID for a file, IceFS simply leverages the static mapping of inode numbers to block groups as in the base Ext3 file system; after mapping the inode of the file to the block group, IceFS obtains the cube ID from the corresponding group descriptor. Since all group descriptors are loaded into memory during the mount process, no extra I/O is required to determine the cube of a file.

IceFS trades disk and memory space for the independence of cubes. To save memory and reduce disk I/O, Ext3 typically places multiple contiguous group descriptors into a single disk block. IceFS modifies this policy so that only group descriptors from the same cube can be placed in the same block. This approach is similar to the meta-group of Ext4 for combining several block groups into a larger block group [35].

4.1.3 Access Independence

To disentangle cubes, no cube can reference another cube. Thus, IceFS partitions each global list that Ext3 maintains into per-cube lists. Specifically, Ext3 stores the head of the global orphan inode list in the super block. To isolate this shared list and the shared super block, IceFS uses one *sub-super* block for each cube; these sub-super blocks are stored on disk after the super block and each references its own orphan inode list as shown in Figure 5. IceFS preallocates a fixed number of sub-super blocks following the super block. The maximum number of sub-super blocks is configurable at `mkfs` time. These sub-super blocks can be replicated within the disk similar to the super block to avoid catastrophic damage of sub-super blocks.

In contrast to a traditional file system, if IceFS detects a reference from one cube to a block in another cube, then it knows that reference is incorrect. For example, no data block should be located in a different cube than the inode of the file to which it belongs.

To disentangle the file namespace from its physical representation on disk and to remove the naming dependencies across cubes, IceFS uses *directory indirection*, as shown in Figure 6. With directory indirection, each cube records its top directory; when the file system performs a pathname lookup, it first finds a longest prefix match of the pathname among the cubes' top directory paths; if it does, then only the remaining pathname within the cube is traversed in the traditional manner. For example, if the user wishes to access `/home/bob/research/paper.tex` and `/home/bob/research/` designates the top of a cube, then IceFS will skip directly to parsing `paper.tex` within the cube. As a result, any failure outside of this cube, or to the `home` or `bob` directories, will not affect accessing `paper.tex`.

In IceFS, the path lookup process performed by the VFS layer is modified to provide directory indirection for cubes. The inode number and the pathname of the top directory of a cube are stored in its sub-super block; when the file system is mounted, IceFS pins in memory this information along with the cube's dentry, inode, and pathname. Later, when a pathname lookup is performed, VFS passes the pathname to IceFS so that IceFS can check whether the pathname is within any cube. If there is no match, then VFS performs the lookup as usual; otherwise, VFS uses the matched cube's dentry as a shortcut to resolve the remaining part of the pathname.

4.1.4 Transaction Splitting

To disentangle transactions belonging to different cubes, we introduce *transaction splitting*, as shown in Figure 7. With transaction splitting, each cube has its own running transaction to buffer writes. Transactions from different cubes are committed to disk in parallel without any waiting or dependencies across cubes. With this approach, any failure along the transaction I/O path can be attributed to the source cube, and the related recovery action can be triggered only for the faulty cube, while other healthy cubes still function normally.

IceFS leverages the existing generic journaling mod-

ule of Ext3, JBD. To provide specialized journaling for different cubes, each cube has a virtual journal managed by JBD with a potentially customized journaling mode. When IceFS starts an atomic operation for a file or directory, it passes the related cube ID to JBD. Since each cube has a separate virtual journal, a commit of a running transaction will only be triggered by its own `fsync()` or timeout without any entanglement with other cubes.

Different virtual journals share the physical journal space on disk. At the beginning of a commit, IceFS will first reserve journal space for the transaction of the cube; a separate committing thread will flush the transaction to the journal. Since transactions from different cubes write to different places on the journal, IceFS can perform multiple commits in parallel. Note that, the original JBD uses a shared lock to synchronize various structures in the journaling layer, while IceFS needs only a single shared lock to allocate transaction space; the rest of the transaction operations can now be performed independently without limiting concurrency.

4.2 Localized Reactions to Failures

As shown in Section 2, current file systems handle serious errors by crashing the whole system or marking the entire file system as read-only. Once a disentangled file system is partitioned into multiple independent cubes, the failure of one cube can be detected and controlled with a more precise boundary. Therefore, failure isolation can be achieved by transforming a global failure to a local per-cube failure.

4.2.1 Fault Detection

Our goal is to provide a new fault-handling primitive, which can localize global failure behaviors to an isolated cube. This primitive is largely orthogonal to the issue of detecting the original faults. We currently leverage existing detection mechanism within file systems to identify various faults.

For example, file systems tend to detect metadata corruption at the I/O boundary by using their own semantics to verify the correctness of file system structures; file systems check error conditions when interacting with other subsystems (e.g., failed disk read/writes or memory allocations); file systems also check assertions and invariants that might fail due to concurrency problems.

IceFS modifies the existing detection techniques to make them cube-aware. For example, Ext3 calls `ext3_error()` to mark the file system as read-only on an inode bitmap read I/O fault. IceFS instruments the fault-handling and crash-triggering functions (e.g., `BUG_ON()`) to include the ID of the responsible cube; pinpointing the faulty cube is straightforward as all metadata is isolated. Thus, IceFS has cube-aware fault detectors.

One can argue that the incentive for detecting problems in current file systems is relatively low because many of the existing recovery techniques (e.g., calling `panic()`) are highly pessimistic and intrusive, making the entire system unusable. A disentangled file system can contain faults within a single cube and thus provides incentive to add more checks to file systems.

4.2.2 Localized Read-Only

As a recovery technique, IceFS enables a single cube to be made read-only. In IceFS, only files within a faulty cube are made read-only, and other cubes remain available for both reads and writes, improving the overall availability of the file system. IceFS performs this per-cube reaction by adapting the existing mechanisms within Ext3 for making all files read-only.

To guarantee read-only for all files in Ext3, two steps are needed. First, the transaction engine is immediately shut down. Existing running transactions are aborted, and attempting to create a new transaction or join an existing transaction results in an error code. Second, the generic VFS super block is marked as read-only; as a result, future writes are rejected.

To localize read-only failures, a disentangled file system can execute two similar steps. First, with the transaction split framework, IceFS individually aborts the transaction for a single cube; thus, no more transactions are allowed for the faulty cube. Second, the faulty cube alone is marked as read-only, instead of the whole file system. When any operation is performed, IceFS now checks this per-cube state whenever it would usually check the super block read-only state. As a result, any write to a read-only cube receives an error code, as desired.

4.2.3 Localized Crashes

Similarly, IceFS is able to localize a crash for a failed cube, such that the crash does not impact the entire operating system or operations of other cubes. Again, IceFS leverages the existing mechanisms in the Linux kernel for dealing with crashes caused by `panic()`, `BUG()`, and `BUG_ON()`. IceFS performs the following steps:

- *Fail the crash-triggering thread:* When a thread fires an assertion failure, IceFS identifies the cube being accessed and marks that cube as crashed. The failed thread is directed to the failure path, during which the failed thread will free its allocated resources (e.g., locks and memory). IceFS adds this error path if it does not exist in the original code.
- *Prevent new threads:* A crashed cube should reject any new file-system request. IceFS identifies whether a request is related to a crashed cube as early as possible and return appropriate error codes to terminate the related system call. Preventing new accesses consists of blocking the entry point functions and the directory indirection functions. For example, the state of a cube is checked at all the callbacks provided by Ext3, such as super block

operations (e.g., `ext3_write_inode()`), directory operations (e.g., `ext3_readdir()`), and file operations (e.g., `ext3_sync_file()`). One complication is that many system calls use either a pathname or a file descriptor as an input; VFS usually translates the pathname or file descriptor into an inode. However, directory indirection in IceFS can be used to quickly prevent a new thread from entering the crashed cube. When VFS conducts the directory indirection, IceFS will see that the pathname belongs to a crashed cube and VFS will return an appropriate error code to the application.

- *Evacuate running threads:* Besides the crash-triggering thread, other threads may be accessing the same cube when the crash happens. IceFS waits for these threads to leave the crashed cube, so they will free their kernel and file-system resources. Since the cube is marked as crashed, these running threads cannot read or write to the cube and will exit with error codes. To track the presence of on-going threads within a cube, IceFS maintains a simple counter for each cube; the counter is incremented when a system call is entered and decremented when a system call returns, similar to the system-call gate [38].
- *Clean up the cube:* Once all the running threads are evacuated, IceFS cleans up the memory states of the crashed cube similar to the unmount process. Specifically, dirty file pages and metadata buffers belonging to the crashed are dropped without being flushed to disk; clean states, such as cached dentries and inodes, are freed.

4.3 Localized Recovery

As shown in Section 2, current file system checkers do not scale well to large file systems. With the cube abstraction, IceFS can solve this problem by enabling per-cube checking. Since each cube represents an independent fault domain with its own isolated metadata and no references to other cubes, a cube can be viewed as a basic checking unit instead of the whole file system.

4.3.1 Offline Checking

In a traditional file-system checker, the file system must be offline to avoid conflicts with a running workload. For simplicity, we first describe a per-cube offline checker.

Ext3 uses the utility e2fsck to check the file system in five phases [39]. IceFS changes e2fsck to make it cube-aware; we call the resulting checker ice-fsck. The main idea is that IceFS supports partial checking of a file system by examining only faulty cubes. In IceFS, when a corruption is detected at run time, the error identifying the faulty cube is recorded in fixed locations on disk. Thus, when ice-fsck is run, erroneous cubes can be easily identified, checked, and repaired, while ignoring the rest of the file system. Of course, ice-fsck can still perform a full file system check and repair, if desired.

Specifically, ice-fsck identifies faulty cubes and their corresponding block groups by reading the error codes recorded in the journal. Before loading the metadata from a block group, each of the five phases of ice-fsck first ensures that this block group belongs to a faulty cube. Because the metadata of a cube is guaranteed to be self-contained, metadata from other cubes not need to be checked. For example, because an inode in one cube cannot point to an indirect block stored in another cube (or block group), ice-fsck can focus on a subset of the block groups. Similarly, checking the directory hierarchy in ice-fsck is simplified; while e2fsck must verify that every file can be connected back to the root directory, ice-fsck only needs to verify that each file in a cube can be reached from the entry points of the cube.

4.3.2 Online Checking

Offline checking of a file system implies that the data will be unavailable to important workloads, which is not acceptable for many applications. A disentangled file system enables on-line checking of faulty cubes while other healthy cubes remain available to foreground traffic, which can greatly improve the availability of the whole service.

Online checking is challenging in existing file systems because metadata is shared loosely by multiple files; if a piece of metadata must be repaired, then all the related files should be frozen or repaired together. Coordinating concurrent updates between the checker and the file system is non-trivial. However, in a disentangled file system, the fine-grained isolation of cubes makes online checking feasible and efficient.

We note that online checking and repair is a powerful recovery mechanism compared to simply crashing or marking a cube read-only. Now, when a fault or corruption is identified at runtime with existing detection techniques, IceFS can unmount the cube so it is no longer visible, and then launch ice-fsck on the corrupted cube while the rest of the file system functions normally. In our implementation, the on-line ice-fsck is a user-space program that is woken up by IceFS informed of the ID of the faulty cubes.

4.4 Specialized Journaling

As described previously, disentangling journal transactions for different cubes enables write operations in different cubes to proceed without impacting others. Disentangling journal transactions (in conjunction with disentangling all other metadata) also enables different cubes to have different consistency guarantees.

Journaling protects files in case of system crashes, providing certain consistency guarantees, such as metadata

or data consistency. Modern journaling file systems support different modes; for example, Ext3 and Ext4 support, from lowest to highest consistency: *writeback*, *ordered*, and *data*. However, the journaling mode is enforced for the entire file system, even though users and applications may desire differentiated consistency guarantees for their data. Transaction splitting enables a specialized journaling protocol to be provided for each cube.

A disentangled file system is free to choose customized consistency modes for each cube, since there are no dependencies across them; even if the metadata of one cube is updated inconsistently and a crash occurs, other cubes will not be affected. IceFS supports five consistency modes, from lowest to highest: *no fsync*, *no journal*, *writeback journal*, *ordered journal* and *data journal*. In general, there is an incentive to choose modes with lower consistency to achieve higher performance, and an incentive to choose modes with higher consistency to protect data in the presence of system crashes.

For example, a cube that stores important configuration files for the system may use data journaling to ensure both data and metadata consistency. Another cube with temporary files may be configured to use *no journal* (i.e., behave similarly to Ext2) to achieve the highest performance, given that applications can recreate the files if a crash occurs. Going one step further, if users do not care about the durability of data of a particular application, the *no fsync* mode can be used to ignore `fsync()` calls from applications. Thus, IceFS gives more control to both applications and users, allowing them to adopt a customized consistency mode for their data.

IceFS uses the existing implementations within JBD to achieve the three journaling modes of writeback, ordered, and data. Specifically, when there is an update for a cube, IceFS uses the specified journaling mode to handle the update. For *no journal*, IceFS behaves like a non-journaled file system, such as Ext2, and does not use the JBD layer at all. Finally, for *no fsync*, IceFS ignores `fsync()` system calls from applications and directly returns without flushing any related data or metadata.

4.5 Implementation Complexity

We added and modified around 6500 LOC to Ext3/JBD in Linux 3.5 for the data structures and journaling isolation, 970 LOC to VFS for directory indirection and crash localization, and 740 LOC to e2fsprogs 1.42.8 for file system creation and checking. The most challenging part of the implementation was to isolate various data structures and transactions for cubes. Once we carefully isolated each cube (both on disk and in memory), the localized reactions to failures and recovery was straightforward to achieve.

Workload	Ext3 (MB/s)	IceFS (MB/s)	Difference
Sequential write	98.9	98.8	0%
Sequential read	107.5	107.8	+0.3%
Random write	2.1	2.1	0%
Random read	0.7	0.7	0%
Fileserver	73.9	69.8	-5.5%
Varmail	2.2	2.3	+4.5%
Webserver	151.0	150.4	-0.4%

Table 3: **Micro and Macro Benchmarks on Ext3 and IceFS.** *This table compares the throughput of several micro and macro benchmarks on Ext3 and IceFS. Sequential write/read are writing/reading a 1GB file in 4KB requests. Random write/read are writing/reading 128MB of a 1GB file in 4KB requests. Fileserver has 50 threads performing creates, deletes, appends, whole-file writes, and whole-file reads. Varmail emulates a multi-threaded mail server. Webserver is a multi-threaded read-intensive workload.*

5 Evaluation of IceFS

We present evaluation results for IceFS. We first evaluate the basic performance of IceFS through a series of micro and macro benchmarks. Then, we show that IceFS is able to localize many failures that were previously global. All the experiments are performed on machines with an Intel(R) Core(TM) i5-2500K CPU (3.30 GHz), 16GB memory, and a 1TB Hitachi Deskstar 7K1000.B hard drive, unless otherwise specified.

5.1 Overall Performance

We assess the performance of IceFS with micro and macro benchmarks. First, we mount both file systems in the default ordered journaling mode, and run several micro benchmarks (sequential read/write and random read/write) and three macro workloads from Filebench (Fileserver, Varmail, and Webserver). For IceFS, each workload uses one cube to store its data. Table 3 shows the throughput of all the benchmarks on Ext3 and IceFS. From the table, one can see that IceFS performs similarly to Ext3, indicating that our disentanglement techniques incur little overhead.

IceFS maintains extra structures for each cube on disk and in memory. For each cube IceFS creates, one sub-super block (4KB) is allocated on disk. Similar to the original super block, sub-super blocks are also cached in memory. In addition, each cube has its own journaling structures (278 B) and cached running states (104 B) in memory. In total, for each cube, its disk overhead is 4 KB and memory overhead is less than 4.5 KB.

5.2 Localize Failures

We show that IceFS converts many global failures into local, per-cube failures. We inject faults into core file-system structures where existing checks are capable of detecting the problem. These faults are selected from

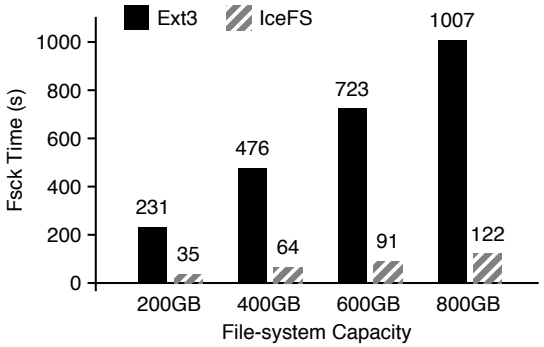

Figure 8: **Performance of IceFS Offline Fsck.** *This figure compares the running time of offline fsck on ext3 and on IceFS with different file-system size.*

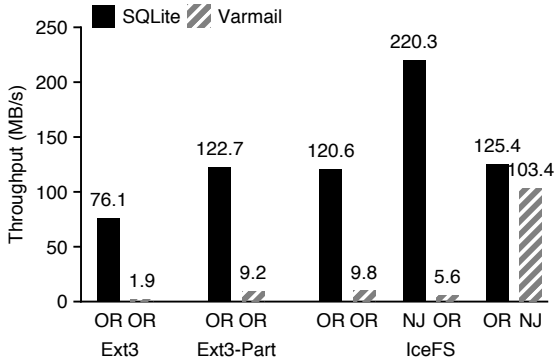

Figure 9: **Running Two Applications on IceFS with Different Journaling Mode.** *This figure compares the performance of simultaneously running SQLite and Varmail on Ext3, partitions and IceFS. In Ext3, both applications run in ordered mode (OR). In Ext3-Part, two separated Ext3 run in ordered mode (OR) on two partitions. In IceFS, two separate cubes with different journaling modes are used: ordered mode (OR) and no-journal mode (NJ).*

Table 2 and they cover all different fault types, including memory allocation failures, metadata corruption, I/O failures, `NULL` pointers, and unexpected states. To compare the behaviors, the faults are injected in the same locations for both Ext3 and IceFS. Overall, we injected nearly 200 faults. With Ext3, in every case, the faults led to global failures of some kind (such as an OS panic or crash). IceFS, in contrast, was able to localize the triggered faults in every case.

However, we found that there are also a small number of failures during the mount process, which are impossible to isolate. For example, if a memory allocation failure happens when initializing the super block during the mount process, then the mount process will exit with an error code. In such cases, both Ext3 and IceFS will not be able to handle it because the fault happens before the file system starts running.

5.3 Fast Recovery

With localized failure detection, IceFS is able to perform offline fsck only on the faulted cube. To measure fsck performance on IceFS, we first create file system images in the same way as described in Figure 1, except that we make 20 cubes instead of directories. We then fail one cube randomly and measure the fsck time. Figure 8 compares the offline fsck time between IceFS and Ext3. The fsck time of IceFS increases as the capacity of the cube grows along with the file system size; in all cases, fsck on IceFS takes much less time than Ext3 because it only needs to check the consistency of one cube.

5.4 Specialized Journaling

We now demonstrate that a disentangled journal enables different consistency modes to be used by different applications on a shared file system. For these experiments, we use a Samsung 840 EVO SSD (500GB) as the underlying storage device. Figure 9 shows the throughput of running two applications, SQLite and Varmail, in Ext3, two separated Ext3 on partitions (Ext3-Part) and IceFS. When running with Ext3 and ordered journaling (two leftmost bars), both applications achieve low performance because they share the same journaling layer and both workloads affect the other. When the applications run with IceFS on two different cubes, their performance increases significantly since `fsync()` calls to one cube do not force out dirty data to the other cube. Compared with Ext3-Part, we can find that IceFS achieves great isolation for cubes at the file system level, similar to running two different file systems on partitions.

We also demonstrate that different applications can benefit from different journaling modes; in particular, if an application can recover from inconsistent data after a crash, the no-journal mode can be used for much higher performance while other applications can continue to safely use ordered mode. As shown in Figure 9, when either SQLite or Varmail is run on a cube with no journaling, that application receives significantly better throughput than it did in ordered mode; at the same time, the competing application using ordered mode continues to perform better than with Ext3. We note that the ordered competing application may perform slightly worse than it did when both applications used ordered mode due to increased contention for resources outside of the file system (i.e., the I/O queue in the block layer for the SSD); this demonstrates that isolation must be provided at all layers of the system for a complete solution. In summary, specialized journaling modes can provide great flexibility for applications to make trade-offs between their performance and consistency requirements.

5.5 Limitations

Although IceFS has many advantages as shown in previous sections, it may perform worse than Ext3 in certain

Device	Ext3 (MB/s)	Ext3-Part (MB/s)	IceFS (MB/s)
SSD	40.8	30.6	35.4
Disk	2.8	2.6	2.7

Table 4: **Limitation of IceFS On Cache Flush.** *This table compares the aggregated throughput of four Varmail instances on Ext3 and IceFS. Each Varmail instance runs in a directory of Ext3, an Ext3 partition (Ext3-Part), or a cube of IceFS. We run the same experiment on both a SSD and hard disk.*

extreme cases. The main limitation of our implementation is that IceFS uses a separate journal commit thread for every cube. The thread issues a device cache `flush` command at the end of every transaction commit to make sure the cached data is persistent on device; this cache flush is usually expensive [21]. Therefore, if many active cubes perform journal commits at the same time, the performance of IceFS may be worse than Ext3 that only uses one journal commit thread for all updates. The same problem exists in separated file systems on partitions.

To show this effect, we choose Varmail as our testing workload. Varmail utilizes multiple threads; each of these threads repeatedly issues small writes and calls `fsync()` after each write. We run multiple instances of Varmail in different directories, partitions or cubes to generate a large number of transaction commits, stressing the file system.

Table 4 shows the performance of running four Varmail instances on our quad-core machine. When running on an SSD, IceFS performs worse than Ext3, but a little better than Ext3 partitions (Ext3-Part). When running on a hard drive, all three setups perform similarly. The reason is that the cache flush time accounts for a large percentage of the total I/O time on an SSD, while the seeking time dominates the total I/O time on a hard disk. Since IceFS and Ext3-Part issue more cache flushes than Ext3, the performance penalty is amplified on the SSD.

Note that this style of workload is an extreme case for both IceFS and partitions. However, compared with separated file systems on partitions, IceFS is still a single file system that can utilize all the related semantic information of cubes for further optimization. For example, IceFS can pass per-cube hints to the block layer, which can optimize the cache flush cost and provide other performance isolation for cubes.

5.6 Usage Scenarios

We demonstrate that IceFS improves overall system behavior in the two motivational scenarios initially introduced in Section 2.3: virtualized environments and distributed file systems.

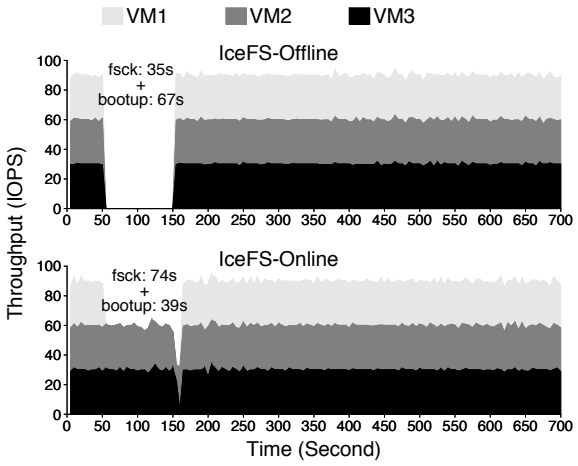

Figure 10: **Failure Handling for Virtual Machines.** *This figure shows how IceFS handles failures in a shared file system which supports multiple virtual machines.*

5.6.1 Virtual Machines

To show that IceFS enables virtualized environments to isolate failures within a particular VM, we configure each VM to use a separate cube in IceFS. Each cube stores a 20GB virtual disk image, and the file system contains 10 such cubes for 10 VMs. Then, we inject a fault to one VM image that causes the host file system to be read-only after 50 seconds.

Figure 10 shows that IceFS greatly improves the availability of the VMs compared to that in Figure 3 using Ext3. The top graph illustrates IceFS with offline recovery. Here, only one cube is read-only and crashes; the other two VMs are shut down properly so the offline cube-aware check can be performed. The offline check of the single faulty cube requires only 35 seconds and booting the three VMs takes about 67 seconds; thus, after only 150 seconds, the three virtual machines are running normally again.

The bottom graph illustrates IceFS with online recovery. In this case, after the fault occurs in VM1 (at roughly 50 seconds) and VM1 crashes, VM2 and VM3 are able to continue. At this point, the online fsck of IceFS starts to recover the disk image file of VM1 in the host file system. Since fsck competes for disk bandwidth with the two running VMs, checking takes longer (about 74 seconds). Booting the single failed VM requires only 39 seconds, but the disk activity that arises as a result of booting competes with the I/O requests of VM2 and VM3, so the throughput of VM2 and VM3 drops for that short time period. In summary, these two experiments demonstrate that IceFS can isolate file system failures in a virtualized environment and significantly reduce system recovery time.

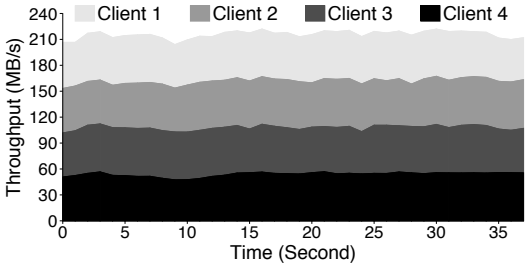

Figure 11: **Impact of Cube Failures in HDFS.** *This figure shows the throughput of 4 different clients when a cube failure happens at time 10 second. Impact of the failure to the clients' throughput is negligible.*

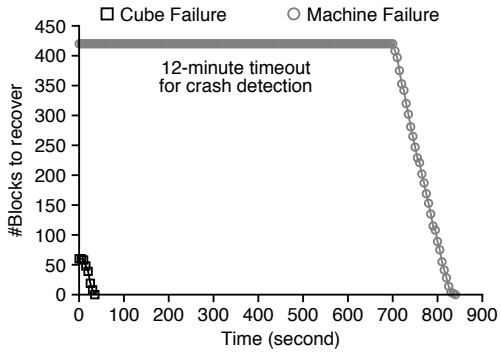

Figure 12: **Data Block Recovery in HDFS.** *The figure shows the number of lost blocks to be regenerated over time in two failure scenarios: cube and whole machine failure. Cube failure results into less blocks to recover in less time.*

5.6.2 Distributed File System

We illustrate the benefits of using IceFS to provide flexible fault isolation in HDFS. Obtaining fault isolation in HDFS is challenging, especially in multi-tenant settings, primarily because HDFS servers are not aware of the data they store, as shown in Section 2.3.2. IceFS provides a natural solution for this problem. We use separate cubes to store different applications' data on HDFS servers. Each cube isolates the data from one application to another; thus, a cube failure will not affect multiple applications. In this manner, IceFS provides end-to-end isolation for applications in HDFS. We added 161 lines to storage node code to make HDFS IceFS-compatible and aware of application data. We do not change any recovery code of HDFS. Instead, IceFS turns global failures (e.g., kernel panic) into partial failures (i.e., cube failure) and leverages HDFS recovery code to handle them. This facilitates and simplifies our implementation.

Figure 11 shows the benefits of IceFS-enabled application-level fault isolation. Here, four clients concurrently access different files stored in HDFS when a cube that stores data for Client 2 fails and becomes inaccessible. Other clients are completely isolated from the cube failure. Furthermore, the failure negligibly impacts the throughput of the client as it does not manifest as machine failure. Instead, it results in a soft error to HDFS, which then immediately isolates the faulty cube and returns an error code the client. The client then quickly fails over to other healthy copies. The overall throughput is stable for the entire workload, as opposed to 60-second period of losing throughput as in the case of whole machine failure described in Section 2.3.2.

In addition to end-to-end isolation, IceFS provides scalable recovery as shown in Figure 12. In particular, IceFS helps reduce network traffic required to regenerate lost blocks, a major bandwidth consumption factor in large clusters [47]. When a cube fails, IceFS again returns an error code to the host server, which then immediately triggers a block scan to find out data blocks that are under-replicated and regenerates them. The number of blocks to recover is proportional to the cube size. Without IceFS, a kernel panic in local file system manifests as whole machine failure, causing a 12-minute timeout for crash detection and making the number of blocks lost and to be regenerated during recovery much larger. In summary, IceFS helps improve not only flexibility in fault isolation but also efficiency in failure recovery.

6 Related Work

IceFS has derived inspiration from a number of projects for improving file system recovery and repair, and for tolerating system crashes.

Many existing systems have improved the reliability of file systems with better recovery techniques. Fast checking of the Solaris UFS [43] has been proposed by only checking the working-set portion of the file system when failure happens. Changing the I/O pattern of the file system checker to reduce random requests has been suggested [14, 34]. A background fsck in BSD [38] checks a file system snapshot to avoid conflicts with the foreground workload. WAFL [29] employs Wafliron [40], an online file system checker, to perform online checking on a volume but the volume being checked cannot be accessed by users. Our recovery idea is based on the cube abstraction which provides isolated failure, recovery and journaling. Under this model, we only check the faulty part of the file system without scanning the whole file system. The above techniques can be utilized in one cube to further speedup the recovery process.

Several repair-driven file systems also exist. Chunkfs [28] does a partial check of Ext2 by partitioning the file system into multiple chunks; however, files and directory can still span multiple chunks, reducing the independence of chunks. Windows ReFS [50] can automatically recover corrupted data from mirrored

storage devices when it detects checksum mismatch. Our earlier work [32] proposes a high-level design to isolate file system structures for fault and recovery isolation. Here, we extend that work by addressing both reliability and performance issues with a real prototype and demonstrations for various applications.

Many ideas for tolerating system crashes have been introduced at different levels. Microrebooting [18] partitions a large application into rebootable and stateless components; to recover a failed component, the data state of each component is persistent in a separate store outside of the application. Nooks [54] isolates failures of device drivers from the rest of the kernel with separated address spaces for each target driver. Membrane [53] handles file system crashes transparently by tracking resource usage and the requests at runtime; after a crash, the file system is restarted by releasing the in-use resources and replaying the failed requests. The Rio file cache [20] protects the memory state of the file system across a system crash, and conducts a warm reboot to recover lost updates. Inspired by these ideas, IceFS localizes a file system crash by microisolating the file system structures and microrebooting a cube with a simple and light-weight design. Address space isolation technique could be used in cubes for better memory fault isolation.

7 Conclusion

Despite isolation of many components in existing systems, the file system still lacks physical isolation. We have designed and implemented IceFS, a file system that achieves physical disentanglement through a new abstraction called cubes. IceFS uses cubes to group logically related files and directories, and ensures that data and metadata in each cube are isolated. There are no shared physical resources, no access dependencies, and no bundled transactions among cubes.

Through experiments, we demonstrate that IceFS is able to localize failures that were previously global, and recover quickly using localized online or offline fsck. IceFS can also provide specialized journaling to meet diverse application requirements for performance and consistency. Furthermore, we conduct two cases studies where IceFS is used to host multiple virtual machines and is deployed as the local file system for HDFS data nodes. IceFS achieves fault isolation and fast recovery in both scenarios, proving its usefulness in modern storage environments.

Acknowledgments

We thank the anonymous reviewers and Nick Feamster (our shepherd) for their tremendous feedback. We thank the members of the ADSL research group for their suggestions and comments on this work at various stages. We thank Yinan Li for the hardware support, and Ao Ma for discussing fsck in detail.

This material was supported by funding from NSF grants CCF-1016924, CNS-1421033, CNS-1319405, and CNS-1218405 as well as generous donations from Amazon, Cisco, EMC, Facebook, Fusion-io, Google, Huawei, IBM, Los Alamos National Laboratory, Mdot-Labs, Microsoft, NetApp, Samsung, Sony, Symantec, and VMware. Lanyue Lu is supported by the VMWare Graduate Fellowship. Samer Al-Kiswany is supported by the NSERC Postdoctoral Fellowship. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and may not reflect the views of NSF or other institutions.

References

[1] Apache HBase. https://hbase.apache.org/.

[2] Docker: The Linux Container Engine. https://www.docker.io.

[3] Filebench. http://sourceforge.net/projects/filebench.

[4] Firefox 3 Uses fsync Excessively. https://bugzilla.mozilla.org/show_bug.cgi?id=421482.

[5] Fsyncers and Curveballs. http://shaver.off.net/diary/2008/05/25/fsyncers-and-curveballs/.

[6] HBase User Mailing List. http://hbase.apache.org/mail-lists.html.

[7] Linux Containers. https://linuxcontainers.org/.

[8] Solving the Ext3 Latency Problem. http://lwn.net/Articles/328363/.

[9] SQLite. https://sqlite.org.

[10] Remzi H. Arpaci-Dusseau and Andrea C. Arpaci-Dusseau. *Operating Systems: Three Easy Pieces*. Arpaci-Dusseau Books, 0.8 edition, 2014.

[11] Lakshmi N. Bairavasundaram, Garth R. Goodson, Shankar Pasupathy, and Jiri Schindler. An Analysis of Latent Sector Errors in Disk Drives. In *Proceedings of the 2007 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems (SIGMETRICS '07)*, San Diego, California, June 2007.

[12] Lakshmi N. Bairavasundaram, Garth R. Goodson, Bianca Schroeder, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. An Analysis of Data Corruption in the Storage Stack. In *Proceedings of the 6th USENIX Symposium on File and Storage Technologies (FAST '08)*, pages 223–238, San Jose, California, February 2008.

[13] Gaurav Banga, Peter Druschel, and Jeffrey C. Mogul. Resource containers: a new facility for resource management in server systems. In *Proceedings of the 3rd Symposium on Operating Systems Design and Implementation (OSDI '99)*, New Orleans, Louisiana, February 1999.

[14] Eric J. Bina and Perry A. Emrath. A Faster fsck for BSD Unix. In *Proceedings of the USENIX Winter Technical Conference (USENIX Winter '89)*, San Diego, California, January 1989.

[15] Silas Boyd-Wickizer and Nickolai Zeldovich. Tolerating Malicious Device Drivers in Linux. In *Proceedings of the USENIX Annual Technical Conference (USENIX '10)*, Boston, Massachusetts, June 2010.

[16] Edouard Bugnion, Scott Devine, and Mendel Rosenblum. Disco: Running Commodity Operating Systems on Scalable Multiprocessors. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles (SOSP '97)*, pages 143–156, Saint-Malo, France, October 1997.

[17] Calton Pu and Tito Autrey and Andrew Black and Charles Consel and Crispin Cowan and Jon Inouye and Lakshmi Kethana and Jonathan Walpole and Ke Zhang. Optimistic Incremental Specialization: Streamlining a Commercial Operating System. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles (SOSP '95)*, Copper Mountain Resort, Colorado, December 1995.

[18] George Candea, Shinichi Kawamoto, Yuichi Fujiki, Greg Friedman, and Armando Fox. Microreboot – A Technique for Cheap Recovery. In *Proceedings of the 6th Symposium on Operating Systems Design and Implementation (OSDI '04)*, pages 31–44, San Francisco, California, December 2004.

[19] Remy Card, Theodore Ts'o, and Stephen Tweedie. Design and Implementation of the Second Extended Filesystem. In *First Dutch International Symposium on Linux*, Amsterdam, Netherlands, December 1994.

[20] Peter M. Chen, Wee Teck Ng, Subhachandra Chandra, Christopher Aycock, Gurushankar Rajamani, and David Lowell. The rio file cache: Surviving operating system crashes. In *Proceedings of the 7th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS VII)*, Cambridge, Massachusetts, October 1996.

[21] Vijay Chidambaram, Thanumalayan Sankaranarayana Pillai, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Optimistic Crash Consistency. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP '13)*, Nemacolin Woodlands Resort, Farmington, Pennsylvania, October 2013.

[22] Andy Chou, Junfeng Yang, Benjamin Chelf, Seth Hallem, and Dawson Engler. An Empirical Study of Operating System Errors. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP '01)*, pages 73–88, Banff, Canada, October 2001.

[23] Christopher Clark, Keir Fraser, Steven Hand, Jacob Gorm Hansen, Eric Jul, Christian Limpach, Ian Pratt, and Andrew Warfield. Live migration of virtual machines. In *Proceedings of the 2nd Symposium on Networked Systems Design and Implementation (NSDI '05)*, Boston, Massachusetts, May 2005.

[24] Boris Dragovic, Keir Fraser, Steve Hand, Tim Harris, Alex Ho, Ian Pratt, Andrew Warfield, Paul Barham, and Rolf Neugebauer. Xen and the Art of Virtualization. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP '03)*, Bolton Landing, New York, October 2003.

[25] Ian Goldberg, David Wagner, Randi Thomas, and Eric A. Brewer. A Secure Environment for Untrusted Helper Applications. In *Proceedings of the 6th USENIX Security Symposium (Sec '96)*, San Jose, California, 1996.

[26] Diwaker Gupta, Ludmila Cherkasova, Rob Gardner, and Amin Vahdat. Enforcing Performance Isolation Across Virtual Machines in Xen. In *Proceedings of the ACM/IFIP/USENIX 7th International Middleware Conference (Middleware'2006)*, Melbourne, Australia, Nov 2006.

[27] Tyler Harter, Dhruba Borthakur, Siying Dong, Amitanand Aiyer, Liyin Tang, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Analysis of HDFS Under HBase: A Facebook Messages Case Study. In *Proceedings of the 12th USENIX Symposium on File and Storage Technologies (FAST '14)*, Santa Clara, California, February 2014.

[28] Val Henson, Arjan van de Ven, Amit Gud, and Zach Brown. Chunkfs: Using divide-and-conquer to improve file system reliability and repair. In *IEEE 2nd Workshop on Hot Topics in System Dependability (HotDep '06)*, Seattle, Washington, November 2006.

[29] Dave Hitz, James Lau, and Michael Malcolm. File System Design for an NFS File Server Appliance. In *Proceedings of the USENIX Winter Technical Conference (USENIX Winter '94)*, San Francisco, California, January 1994.

[30] Shvetank Jain, Fareha Shafique, Vladan Djeric, and Ashvin Goel. Application-Level Isolation and Recovery with Solitude. In *Proceedings of the EuroSys Conference (EuroSys '08)*, Glasgow, Scotland UK, March 2008.

[31] Poul-Henning Kamp and Robert N. M. Watson. Jails: Confining the omnipotent root. In *Second International System Administration and Networking Conference (SANE '00)*, May 2000.

[32] Lanyue Lu, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Fault Isolation And Quick Recovery in Isolation File Systems. In *5th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage '13)*, San Jose, CA, June 2013.

[33] Lanyue Lu, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, and Shan Lu. A Study of Linux File System Evolution. In *Proceedings of the 11th USENIX Symposium on File and Storage Technologies (FAST '13)*, San Jose, California, February 2013.

[34] Ao Ma, Chris Dragga, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. ffsck: The Fast File System Checker. In *Proceedings of the 11th USENIX Symposium on File and Storage Technologies (FAST '13)*, San Jose, California, February 2013.

[35] Avantika Mathur, Mingming Cao, Suparna Bhattacharya, Alex Tomas Andreas Dilge and, and Laurent Vivier. The New Ext4 filesystem: Current Status and Future Plans. In *Ottawa Linux Symposium (OLS '07)*, Ottawa, Canada, July 2007.

[36] Avantika Mathur, Mingming Cao, Suparna Bhattacharya, Andreas Dilger, Alex Tomas, Laurent Vivier, and Bull S.A.S. The New Ext4 Filesystem: Current Status and Future Plans. In *Ottawa Linux Symposium (OLS '07)*, Ottawa, Canada, July 2007.

[37] Marshall K. McKusick, William N. Joy, Sam J. Leffler, and Robert S. Fabry. A Fast File System for UNIX. *ACM Transactions on Computer Systems*, 2(3):181–197, August 1984.

[38] Marshall Kirk McKusick. Running 'fsck' in the Background. In *Proceedings of BSDCon 2002 (BSDCon '02)*, San Fransisco, California, February 2002.

[39] Marshall Kirk McKusick, Willian N. Joy, Samuel J. Leffler, and Robert S. Fabry. Fsck - The UNIX File System Check Program. Unix System Manager's Manual - 4.3 BSD Virtual VAX-11 Version, April 1986.

[40] NetApp. Overview of WAFL_check. http://uadmin.nl/init/?p=900, Sep. 2011.

[41] Oracle Inc. Consolidating Applications with Oracle Solaris Containers. http://www.oracle.com/technetwork/server-storage/solaris/documentation/consolidating-apps-163572.pdf, Jul 2011.

[42] Nicolas Palix, Gael Thomas, Suman Saha, Christophe Calves, Julia Lawall, and Gilles Muller. Faults in Linux: Ten Years Later. In *Proceedings of the 15th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XV)*, Newport Beach, California, March 2011.

[43] J. Kent Peacock, Ashvin Kamaraju, and Sanjay Agrawal. Fast Consistency Checking for the Solaris File System. In *Proceedings of the USENIX Annual Technical Conference (USENIX '98)*, pages 77–89, New Orleans, Louisiana, June 1998.

[44] Vijayan Prabhakaran, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Analysis and Evolution of

Journaling File Systems. In *Proceedings of the USENIX Annual Technical Conference (USENIX '05)*, pages 105–120, Anaheim, California, April 2005.

[45] Vijayan Prabhakaran, Lakshmi N. Bairavasundaram, Nitin Agrawal, Haryadi S. Gunawi, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. IRON File Systems. In *Proceedings of the 20th ACM Symposium on Operating Systems Principles (SOSP '05)*, pages 206–220, Brighton, United Kingdom, October 2005.

[46] Feng Qin, Joseph Tucek, Jagadeesan Sundaresan, and Yuanyuan Zhou. Rx: Treating Bugs As Allergies. In *Proceedings of the 20th ACM Symposium on Operating Systems Principles (SOSP '05)*, Brighton, United Kingdom, October 2005.

[47] K. V. Rashmi, Nihar B. Shah, Dikang Gu, Hairong Kuang, Dhruba Borthakur, and Kannan Ramchandran. A solution to the network challenges of data recovery in erasure-coded distributed storage systems: A study on the facebook warehouse cluster. In *5th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage '13)*, San Jose, CA, June 2013.

[48] Constantine P. Sapuntzakis, Ramesh Chandra, Ben Pfaff, Jim Chow, Monica S. Lam, and Mendel Rosenblum. Optimizing the Migration of Virtual Computers. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation (OSDI '02)*, Boston, Massachusetts, December 2002.

[49] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. The Hadoop Distributed File System. In *Proceedings of the 26th IEEE Symposium on Mass Storage Systems and Technologies (MSST '10)*, Incline Village, Nevada, May 2010.

[50] Steven Sinofsky. Building the Next Generation File System for Windows: ReFS. http://blogs.msdn.com/b/b8/archive/2012/01/16/building-the-next-generation-file-system-for-windows-refs.aspx, Jan. 2012.

[51] Mark Sullivan and Ram Chillarege. Software defects and their impact on system availability-a study of field failures in operating systems. In *Proceedings of the 21st International Symposium on Fault-Tolerant Computing (FTCS-21)*, pages 2–9, Montreal, Canada, June 1991.

[52] Mark Sullivan and Ram Chillarege. A Comparison of Software Defects in Database Management Systems and Operating Systems. In *Proceedings of the 22st International Symposium on Fault-Tolerant Computing (FTCS-22)*, pages 475–484, Boston, USA, July 1992.

[53] Swaminathan Sundararaman, Sriram Subramanian, Abhishek Rajimwale, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, and Michael M. Swift. Membrane: Operating System Support for Restartable File Systems. In *Proceedings of the 8th USENIX Symposium on File and Storage Technologies (FAST '10)*, San Jose, California, February 2010.

[54] Michael M. Swift, Brian N. Bershad, and Henry M. Levy. Improving the Reliability of Commodity Operating Systems. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP '03)*, Bolton Landing, New York, October 2003.

[55] Theodore Ts'o. http://e2fsprogs.sourceforge.net, June 2001.

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

[57] Satyam B. Vaghani. Virtual Machine File System. *ACM SIGOPS Operating Systems Review*, 44(4):57–70, Dec 2010.

[58] Ben Verghese, Anoop Gupta, and Mendel Rosenblum. Performance Isolation: Sharing and Isolation in Shared-Memory Multiprocessors. In *Proceedings of the 8th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS VIII)*, San Jose, California, October 1998.

[59] VMware Inc. VMware Workstation. http://www.vmware.com/products/workstation, Apr 2014.

[60] Matthew Wachs, Michael Abd-El-Malek, Eno Thereska, and Gregory R. Ganger. Argon: Performance Insulation for Shared Storage Servers. In *Proceedings of the 5th USENIX Symposium on File and Storage Technologies (FAST '07)*, San Jose, California, February 2007.

[61] Feng Zhou, Jeremy Condit, Zachary Anderson, Ilya Bagrak, Rob Ennals, Matthew Harren, George Necula, and Eric Brewer. SafeDrive: Safe and Recoverable Extensions Using Language-Based Techniques. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation (OSDI '06)*, Seattle, Washington, November 2006.