A Logic of File Systems

Muthian Sivathanu*, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, Somesh Jha
Google Inc. *Computer Sciences Department, University of Wisconsin, Madison*

muthian@google.com, {dusseau, remzi, jha}@cs.wisc.edu

Abstract

Years of innovation in file systems have been highly successful in improving their performance and functionality, but at the cost of complicating their interaction with the disk. A variety of techniques exist to ensure consistency and integrity of file system data, but the precise set of correctness guarantees provided by each technique is often unclear, making them hard to compare and reason about. The absence of a formal framework has hampered detailed verification of file system correctness.

We present a logical framework for modeling the interaction of a file system with the storage system, and show how to apply the logic to represent and prove correctness properties. We demonstrate that the logic provides three main benefits. First, it enables reasoning about existing file system mechanisms, allowing developers to employ aggressive performance optimizations without fear of compromising correctness. Second, the logic simplifies the introduction and adoption of new file system functionality by facilitating rigorous proof of their correctness. Finally, the logic helps reason about smart storage systems that track semantic information about the file system.

A key aspect of the logic is that it enables *incremental modeling*, significantly reducing the barrier to entry in terms of its actual use by file system designers. In general, we believe that our framework transforms the hitherto esoteric and error-prone "art" of file system design into a readily understandable and formally verifiable process.

1 Introduction

Reliable data storage is the cornerstone of modern computer systems. File systems are responsible for managing persistent data, and it is therefore essential to ensure that they function correctly.

Unfortunately, modern file systems have evolved into extremely complex pieces of software, incorporating sophisticated performance optimizations and features. Because disk I/O is the key bottleneck in file system performance, most optimizations aim at minimizing disk access, often at the cost of complicating the interaction of the file system with the storage system; while early file systems adopted simple update policies that were easy to reason about [11], modern file systems have significantly more complex interaction with the disk, mainly stemming from asynchrony in updates to metadata [2, 6, 8, 12, 18, 22, 23].

Reasoning about the interaction of a file system with disk is paramount to ensuring that the file system never corrupts or loses data. However, with complex update policies, the precise set of guarantees that the file system provides is obscured, and reasoning about its behavior often translates into a manual intuitive exploration of various scenarios by the developers; such *ad hoc* exploration is arduous [23], and possibly error-prone. For example, recent work [24] has found major correctness errors in widely used file systems such as ext3, ReiserFS and JFS.

In this paper, we present a formal logic for modeling the interaction of a file system with the disk. With formal modeling, we show that reasoning about file system correctness is simple and foolproof. The need for such a formal model is illustrated by the existence of similar frameworks in many other areas where correctness is paramount; existing models for authentication protocols [4], database reliability [7], and database recovery [9] are a few examples. While general theories for modeling concurrent systems exist [1, 10], such frameworks are too general to model file systems effectively; a domain-specific logic greatly simplifies modeling [4].

A logic of file systems serves three important purposes. First, it enables us to prove properties about existing file system designs, resulting in better understanding of the set of guarantees and enabling aggressive performance optimizations that preserve those guarantees. Second, it significantly lowers the barrier to providing new mechanisms or functionality in the file system by enabling rigorous reasoning about their correctness; in the absence of such a framework, designers tend to stick with "time-tested" alternatives. Finally, the logic helps design functionality in new class of storage systems [20] by facilitating precise characterization and proof of their properties.

A key goal of the logic framework is *simplicity*; in order to be useful to general file system designers, the barrier to entry in terms of applying the logic should be low. Our logic achieves this by enabling *incremental modeling*. One need not have a complete model of a file system before starting to use the logic; instead, one can simply model a particular piece of functionality or mechanism in isolation and prove properties about it.

Through case studies, we demonstrate the utility and efficacy of our logic in reasoning about file system cor-

*Work done while at the University of Wisconsin-Madison rectness properties. First, we represent and prove the soundness of important guarantees provided by existing techniques for file system consistency, such as soft updates and journaling. We then use the logic to prove that the Linux ext3 file system is needlessly conservative in its transaction commits, resulting in sub-optimal performance; this case study demonstrates the utility of the logic in enabling aggressive performance optimizations.

To illustrate the utility of the logic in developing new file system functionality, we propose a new file system mechanism called *generation pointers* to enable *consistent undelete* of files. We prove the correctness of our design by incremental modeling of this mechanism in our logic, demonstrating the simplicity of the process. We then implement the mechanism in the Linux ext3 file system, and verify its correctness. As the logic indicates, we empirically show that inconsistency does indeed occur in undeletes in the absence of our mechanism.

The rest of the paper is organized as follows. We first present an extended motivation (§2), and a background on file systems (§3). We present the basic entities in our logic (§4) and the formalism (§5), and represent some common file system properties using the logic (§6). We then use the logic to prove consistency properties of existing systems (§7), prove the correctness of an unexploited performance optimization in ext3 (§8), and reason about a new technique for consistent undeletes (§9). We then apply our logic to semantic disks (§10). Finally, we present related work (§11) and conclude (§12).

2 Extended Motivation

A systematic framework for reasoning about the interaction of a file system with the disk has multifarious benefits. We describe three key applications of the framework.

2.1 Reasoning about existing file systems

An important usage scenario for the logic is to model existing file systems. There are three key benefits to such modeling. First, it enables a clear understanding of the precise guarantees that a given mechanism provides, and the assumptions under which those guarantees hold. Such an understanding enables correct implementation of functionality at *other* system layers such as the disk system by ensuring that they do not adversely interact with the file system assumptions. For example, write-back caching in disks often results in reordering of writes to the media; this can negate the assumptions journaling is based on.

Second, the logic enables aggressive performance optimizations. When reasoning about complex interactions becomes hard, file system developers tend to be conservative (*e.g.*, perform unnecessarily more waits). Our logic helps remove this barrier, enabling developers to be aggressive in their performance optimizations while still being confident of their correctness. In Section 8, we analyze a real example of such an opportunity for optimization in the Linux ext3 file system, and show that the logic framework can help prove its correctness.

The final benefit of the logic framework is its potential use in implementation-level model checkers [24]; having a clear model of expected behavior against which to validate an existing file system would perhaps enable more comprehensive and efficient model checking, instead of the current technique of relying on the *fsck* mechanism which is quite expensive; the cost of an fsck on every explored state limits the scalability of such model checking.

2.2 Building new file system functionality

Recovery and consistency are traditionally viewed as "tricky" issues to reason about and get right. A classic illustration of this view arises in database recovery; the widely used ARIES [13] algorithm pointed to correctness issues with many earlier proposals. Ironically, the success of ARIES stalled innovation in database recovery, due to the difficulty in proving the correctness of new techniques.

Given that most innovation within the file system deals with its interaction with the disk and can have correctness implications, this inertia against changing "time-tested" alternatives stifles the incorporation of new functionality in file systems. A systematic framework to reason about a new piece of functionality can greatly reduce this barrier to entry. In Section 9, we propose new file system functionality and use our logic to prove its correctness. To further illustrate the efficacy of the logic in reasoning about new functionality, we examine in Section 7.2.1 a common file system feature, *i.e.*, journaling, and show that starting from a simple logical model of journaling, we can systematically arrive at the various corner cases that need to be handled, some of which involve complex interactions as described by the developers of Linux Ext3 [23].

2.3 Designing semantically-smart disks

The logic framework also significantly simplifies reasoning about a new class of storage systems called *semantically-smart disk systems* that provide enhanced functionality by inferring file system operations [20]. Inferring information accurately underneath modern file systems is known to be quite complex [21], especially because it is dependent on dynamic file system properties. In Section 10, we show that the logic can simplify reasoning about a semantic disk; this can in turn enable aggressive functionality in them.

3 Background

A file system organizes disk blocks into logical files and directories. In order to map blocks to logical entities such as files, the file system tracks various forms of *metadata*. In this section, we first describe the forms of metadata that file systems track, and then discuss the issue of file system consistency. Finally, we describe the asynchrony of file systems, a major source of complexity in its interaction with disk.

3.1 File system metadata

File system metadata can be classified into three types:

Directories: Directories map a logical file name to per-file metadata. Since the file mapped for a name can be a directory itself, directories enable a hierarchy of files. When a user opens a file specifying its *path name*, the file system locates the per-file metadata for the file, reading each directory in the path if required.

File metadata: File metadata contains information about a specific file. Examples of such information are the set of disk blocks that comprise the file, file size, and so on. In certain file systems such as FAT, file metadata is embedded in the directory entries, while in most other file systems, file metadata is stored separately (*e.g.*, inodes) and is pointed to by the directory entries. The pointers from file metadata to the disk blocks can sometimes be indirected through *indirect pointer* blocks in the case of large files.

Allocation structures: File systems manage various resources on disk such as the set of free blocks that can be allocated to new files. To track such resources, file systems maintain structures (*e.g.*, bitmaps, free lists) that point to free resource instances.

In addition, file systems track other metadata (*e.g.*, super block), but we mainly focus on the above three types.

3.2 File system consistency

For proper operation, the internal metadata of the file system and its data blocks should be in a *consistent* state. By *metadata consistency*, we mean that the state of the various metadata structures obeys a set of invariants that the file system relies on. For example, a directory entry should only point to a valid file metadata structure; if a directory points to file metadata that is uninitialized (*i.e.*, marked free), the file system is said to be *inconsistent*.

Most file systems provide metadata consistency, since that is crucial to correct operation. A stronger form of consistency is *data consistency*, where the file system guarantees that data block contents always correspond to the file metadata structures that point to them. We discuss this issue in Section 7.1. Many modern file systems such as Linux ext3 and ReiserFS provide data consistency.

3.3 File system asynchrony

An important characteristic of most modern file systems is the *asynchrony* they exhibit during updates to data and metadata. Updates are simply buffered in memory and are written to disk only after a certain delay interval, with possible reordering among those writes. While such asynchrony is crucial for performance, it complicates consistency management. Due to asynchrony, a system crash leads to a state where an arbitrary subset of updates has been applied on disk, potentially leading to an inconsistent on-disk state. Asynchrony of updates is the principal reason for complexity in the interaction of a file system with the disk, and hence the *raison d'etre* of our logic.

4 Basic entities and notations

In this section, we define the basic entities that constitute a file system in our logic, and present their notations. In the next section, we build upon these entities to present our formalism of the operation of a file system.

4.1 Basic entities

The basic entities in our model are *containers*, *pointers*, and *generations*. A file system is simply a collection of containers. Containers are linked to each other through pointers. Each file system differs in the exact types of containers it defines and the relationship it allows between those container types; we believe that this abstraction based on containers and pointers is general to describe any file system.

Containers in a file system can be *freed* and *reused*; a container is considered to be free when it is not pointed to by any other container; it is *live* otherwise. The instance of a container between a reuse and the next free is called a *generation*; thus, a generation is a specific incarnation of a container. Generations are never reused. When a container is reused, the previous generation of that container is freed and a new generation of the container comes to life. A generation is thus fully defined by its container plus a logical *generation number* that tracks how many times the container was reused. Note that generation does *not* refer to the *contents* of a container, but is an abstraction for its current incarnation; contents can change without affecting the generation.

We illustrate the notion of containers and generations with a simple example from a typical UNIX-based file system. If the file system contains a fixed set of designated *inodes*, each inode slot is a *container*. At any given point, an inode slot in use is associated with an inode *generation* that corresponds to a specific file. When the file is deleted, the corresponding inode generation is deleted (forever), but the inode container is simply marked free. A different file created later can reuse the same inode container for a logically different inode generation.

Note that a single container (*e.g.*, an inode) can point to multiple containers (*e.g.*, data blocks). A single container can also be sometimes pointed to by multiple containers (*e.g.*, hard links in UNIX file systems).

4.2 Notations

The notations used to depict the basic entities and the relationships across them are listed in Table 1. Note that many notations in the table are defined only later in the section. Containers are denoted by upper case letters, while generations are denoted by lower case letters. An "entity" in the description represents a container or a generation.

Symbol	Description
$\&A$	set of entities that point to container A
$*A$	set of entities pointed to by container A
$\|A\|$	container that tracks if container A is live
$\&a$	set of entities that point to generation a
$*a$	set of entities pointed to by generation a
$A \rightarrow B$	denotes that container A has a pointer to B
$\&A = \emptyset$	denotes that no entity points to A
A^k	the k^{th} epoch of container A
$t(A^k)$	type of k^{th} epoch of container A
$g(A^k)$	generation of the k^{th} epoch of container A
$C(a)$	container associated with generation a
A_k	generation k of container A

Table 1: **Notations on containers and generations.**

A pointer is denoted by the $\rightarrow$ symbol; $A \rightarrow B$ indicates that container A has a pointer to container B, *i.e.*, $(A \in \&B) \wedge (B \in *A)$. For most of this paper, we only consider pointers from and to containers that are live. In Section 9, we will relax this assumption and introduce a new notation for pointers involving dead containers.

4.3 Attributes of containers

To make the logic expressive for modern file systems, we extend its vocabulary with attributes on a container; a generation has the same attributes as its container.

4.3.1 Epoch

The *epoch* of a container is defined as follows: every time the *contents* of a container change *in memory*, its epoch is incremented. For example, if the file system sets different fields in an inode one after the other, each step results in a new epoch of the inode container. Since the file system can batch multiple changes to the contents due to buffering, the set of epochs visible at the disk is a subset of the total set of epochs a container goes through. We denote an epoch by the superscript notation; A^k denotes the k^{th} epoch of A. Note that our definition of epoch is only used for expressivity of our logic; it does not imply that the file system tracks such an epoch. Also note the distinction between an *epoch* and a *generation*; a generation change occurs only on a reuse of the container, while an epoch changes on every change in contents *or* when the container is reused.

4.3.2 Type

Containers can have a certain *type* associated with them. The type of a container can either be *static*, *i.e.*, it does not change during the lifetime of the file system, or can be *dynamic*, where the same container can belong to different types at different points in time. For example, in FFS-based file systems, *inode* containers are statically typed, while block containers may change their type between data, directory, and indirect pointers. We denote the type of a container A by the notation $t(A)$.

4.3.3 Shared vs. unshared

A container that is pointed to by more than one container is called a *shared container*; a container that has exactly one pointer leading into it is unshared. By default, we assume that containers are shared. We denote unshared containers with the $\oplus$ operator. $\oplus A$ indicates that A is unshared. Note that being unshared is a property of the container *type* that the file system always ensures; a container belonging to a type that is unshared will always have only one pointer pointing into it. For example, most file systems designate data block containers to be unshared.

4.4 Memory and disk versions of containers

A file system needs to manage its structures across two domains: volatile memory and disk. Before accessing the contents of a container, the file system needs to *read* the on-disk version of the container into memory. Subsequently, the file system makes modifications to the in-memory copy of the container, and such modified contents are periodically written to disk. Thus, until the file system writes a modified container to disk, the contents of the container in memory will be different from that on disk.

5 The Formalism

We now present our formal model of the operation of a file system. We first formulate the logic in terms of *beliefs* and *actions*, and then introduce the operators in the logic, our proof system, and the basic axioms in the logic.

5.1 Beliefs

The state of the system is modeled using *beliefs*. A belief represents a certain state in memory or disk.

Any statement enclosed within $\{\}$ represents a belief. Beliefs can be either *in memory* beliefs or *on disk* beliefs, and are denoted as either $\{\}_M$ or $\{\}_D$ respectively. For example $\{A \rightarrow B\}_M$ indicates that $A \rightarrow B$ is a belief in the file system memory, *i.e.*, container A currently points to B in memory, while $\{A \rightarrow B\}_D$ means it is a disk belief. The timing of when such a belief begins to hold is determined in the context of a *formula* in our logic, as we describe in the next subsection; in brief terms, the timing of a belief is defined *relative to* other beliefs or actions specified in the formula. An isolated belief in itself thus has no temporal dimension.

While memory beliefs just represent the state the file system tracks in memory, on-disk beliefs are defined as follows: a belief holds on disk at a given time, if on a crash, the file system can conclude with the same belief purely based on a scan of on-disk state at that time. On-disk beliefs are thus solely dependent on on-disk data.

Since the file system manages free and reuse of containers, its beliefs can be in terms of *generations*; for example $\{A_k \rightarrow B_j\}_M$ is valid (note that A_k refers to generation k of container A). However, on-disk beliefs can only deal with containers, since generation information is lost at the disk. In Sections 9 and 10, we propose techniques to expose generation information to the disk, and show that it enables improved guarantees.

5.2 Actions

The other component of our logic is *actions*, which result in changes to system state; actions thus alter the set of beliefs that hold at a given time. There are two actions defined in our logic:

- $read(A)$ – This operation is used by the file system to read the contents of an on-disk container (and thus, its current generation) into memory. The file system needs to have the container in memory before it can modify it. After a $read$, the contents of A in memory and on-disk are the same, *i.e.*, $\{A\}_M = \{A\}_D$.
- $write(A)$ – This operation results in flushing the current contents of a container to disk. After this operation, the contents of A in memory and on-disk are the same, *i.e.*, $\{A\}_D = \{A\}_M$.

5.3 Ordering of beliefs and actions

A fundamental aspect of the interaction of a file system with disk is the *ordering* among its actions. The ordering of actions also determines the order in which beliefs are established. To order actions and the resulting beliefs, we use the *before* ($\ll$) and *after* ($\gg$) operators. Thus, $\alpha \ll \beta$ means that α occurred before β in time. Note that by *ordering* beliefs, we are using the $\{\}$ notation as both a way of indicating the *event* of creation of the belief, and the *state* of existence of a belief. For example, the belief $\{B \rightarrow A\}_M$ represents the event where the file system assigns A as one of the pointers from B.

We also use a special ordering operator called *precedes* ($\prec$). Only a belief can appear to the left of a $\prec$ operator. The $\prec$ operator is defined as follows: $\alpha \prec \beta$ means that belief α occurs before β (*i.e.*, $\alpha \prec \beta \Rightarrow \alpha \ll \beta$); further, it means that belief α holds at least until β occurs. This implies there is no intermediate action or event between α and β that invalidates belief α.

The operator $\prec$ is *not* transitive; $\alpha \prec \beta \prec \gamma$ does not imply $\alpha \prec \gamma$, because belief α needs to hold only until β and not necessarily until γ (note that $\alpha \prec \beta \prec \gamma$ is simply a shortcut for $(\alpha \prec \beta) \wedge (\beta \prec \gamma)$ (note that this implies $\alpha \ll \gamma$).

Beliefs can be grouped using parantheses, which has the following semantics with precedes:

$$(\alpha \prec \beta) \prec \gamma \quad \Rightarrow \quad (\alpha \prec \beta) \wedge (\alpha \prec \gamma) \wedge (\beta \prec \gamma) \tag{1}$$

If a group of beliefs precedes a certain other belief α, *every* belief within the parantheses precedes belief α.

5.4 Proof system

Given our primitives for sequencing beliefs and actions, we can define *rules* or *formulas* in our logic in terms of an *implication* of one event sequence given another sequence. We use the traditional operators: $\Rightarrow$ (implication) and $\Leftrightarrow$ (double implication, *i.e.*, if and only if). We also use logical AND ($\wedge$) and OR ($\vee$) to combine sequences.

An example of a logical rule is: $\alpha \ll \beta \Rightarrow \gamma$. This notation means that *every time* an event or action β occurs after α, event γ occurs *at the point of occurrence of* β. The rule does not say anything about *when* α or β occurs in absolute time; all it says is whenever they occur in that order, γ occurs. Thus, the above rule would be valid if $\alpha \ll \beta$ never occured at all. In general, if the left hand side of the rule involves a more complex expression, say a disjunction of two components, the belief on the RHS holds at the point of occurrence of the first event that makes the LHS true; in the example above, the occurence of β makes the sequence $\alpha \ll \beta$ true.

Another example of a rule is $\alpha \ll \beta \Rightarrow \alpha \ll \gamma \ll \beta$; this rule denotes that every time β occurs after α, γ should have occurred sometime between α and β. Note that in such a rule where the same event occurs in both sides, the event constitutes a temporal reference point by referring to the same time instant in both the LHS and RHS. This temporal interpretation of identical events is crucial to the above rule serving the intended implication; otherwise the RHS could refer to some other instant where $\alpha \ll \beta$.

Rules such as the above can be used in logical proofs by *event sequence substitution*; for example, with the rule $\alpha \ll \beta \Rightarrow \gamma$, whenever the subsequence $\alpha \ll \beta$ occurs in a sequence of events, it logically implies the event γ. We could then apply the above rule to any event sequence by replacing any subsequence that matches the left half of the rule, with the right half; thus, with the above rule, we have the following postulate: $\alpha \ll \beta \ll \delta \Rightarrow \gamma \ll \delta$. Thus, our proof system enables deriving new invariants about the file system, building on basic axioms.

5.5 Basic axioms

In this subsection, we present the axioms that govern the transition of beliefs across memory and disk.

- If a container B points to A in memory, its current generation also points to A in memory.

$$\{B^x \rightarrow A\}_M \quad \Leftrightarrow \quad \{g(B^x) \rightarrow A\}_M \tag{2}$$

- If B points to A in memory, a $write$ of B will lead to the disk belief that B points to A.

$$\{B \rightarrow A\}_M \prec write(B) \quad \Rightarrow \quad \{B \rightarrow A\}_D \tag{3}$$

  The converse states that the disk belief implies that the same belief first occurred in memory.

$$\{B \rightarrow A\}_D \quad \Rightarrow \quad \{B \rightarrow A\}_M \ll \{B \rightarrow A\}_D \tag{4}$$

- Similarly, if B points to A on disk, a $read$ of B will result in the file system inheriting the same belief.

$$\{B \to A\}_D \prec read(B) \quad \Rightarrow \quad \{B \to A\}_M \qquad (5)$$

- If the on-disk contents of container A pertain to epoch y, some generation c should have pointed to generation $g(A^y)$ in memory, followed by $write(A)$. The converse also holds:

$$\{A^y\}_D \Rightarrow \quad \{c \to g(A^y)\}_M \prec write(A) \ll \{A^y\}_D \quad (6)$$

$$\{c \to A_k\}_M \prec write(A) \Rightarrow \{A^y\}_D \wedge (g(A^y) = k) \qquad (7)$$

Note that A_k refers to some generation k of A, and is used in the above rule to indicate that the generation c points to is the same as that of A^y.

- If $\{b \to A_k\}$ and $\{c \to A_j\}$ hold in memory at two different points in time, container A should have been freed between those instants.

$$\{b \to A_k\}_M \ll \{c \to A_j\}_M \wedge (k \neq j)$$
$$\Rightarrow \{b \to A_k\}_M \ll \{\&A = \emptyset\}_M \prec \{c \to A_j\}_M (8)$$

Note that the rule includes the scenario where an intermediate generation A_l occurs between A_k and A_j.

- If container B pointed to A on disk, and subsequently the file system removes the pointer from B to A in memory, a write of B will lead to the disk belief that B does not point to A.

$$\{B \to A\}_D \prec \{A \notin *B\}_M \prec write(B)$$
$$\Rightarrow \{A \notin *B\}_D \qquad (9)$$

Further, if A is an unshared container, the write of B will lead to the disk belief that no container points to A, *i.e.*, A is *free*.

$$\oplus A \wedge (\{B \to A\}_D \prec \{\&A = \emptyset\}_M \prec write(B))$$
$$\Rightarrow \{\&A = \emptyset\}_D \qquad (10)$$

- If A is a dynamically typed container, and its type at two instants are different, A should have been freed in between.

$$(\{t(A) = x\}_M \ll \{t(A) = y\}_M) \wedge (x \neq y)$$
$$\Rightarrow \{t(A) = x\}_M \ll \{\&A = \emptyset\}_M \prec \{t(A) = y\}_M \qquad (11)$$

5.6 Completeness of notations

The various notations we have discussed in this section cover a wide range of the set of behaviors that we would want to model in a file system. However, this is by no means a *complete* set of notations that can model every aspect of a file system. As we show in Section 7.2 and Section 9, certain specific file system features may require new notations. The main contribution of this paper lies in putting forth a framework to formally reason about file system correctness. Although new notations may sometimes need to be introduced for certain specific file system features, much of the framework will apply without any modification.

5.7 Connections to Temporal Logic

Our logic bears some similarity to linear temporal logic. The syntax of *Linear Temporal Logic (LTL)* [5, 15] is defined as follows:

- A formula $p \in AP$ is an LTL formula, where AP is a set of atomic propositions.
- Given two LTL formulas f and g, $\neg f$, $f \wedge g$, $f \vee g$, $\mathbf{X}\ f$, $\mathbf{F}\ f$, $\mathbf{G}\ f$, $f\ \mathbf{U}\ g$, and $f\ \mathbf{R}\ g$ are LTL formulas.

In the definition given above $\mathbf{X}$("next time"), $\mathbf{F}$("in the future"), $\mathbf{G}$("always"), $\mathbf{U}$("until"), and $\mathbf{R}$("release") are temporal operators. Our formalism is a fragment of LTL, where the set of atomic propositions AP consists of memory and disk beliefs and actions and only temporal operators $\mathbf{F}$ and $\mathbf{U}$ are allowed. In our formalism, $\alpha \ll \beta$ and $\alpha \prec \beta$ are equivalent to $\alpha\ \mathbf{F}\ \beta$ and $\alpha\ \mathbf{U}\ \beta$, respectively.

Given an execution π, which is a sequence of states, and an LTL formula f, $\pi \models f$ denotes that f is true in the execution π. A system S satisfies an LTL formula f if all its executions satisfy f. The precise semantics of the satisfaction relation (the meaning of $\models$) can be found in [5, Chapter 3]. Thus the semantics for our formalism follows from the standard semantics of LTL.

In our proof system, we are given set of axioms $\mathcal{A}$ (given in Section 5.5) and a desired property f (such as the data consistency property described in Section 7.1), and we want to prove that f follows from the axioms in $\mathcal{A}$ (denoted by $\mathcal{A} \to f$), i.e., if a file system satisfies all properties in the set $\mathcal{A}$, it will also satisfy property f.

6 File System Properties

Various file systems provide different guarantees on their update behavior. Each guarantee translates into new rules to the logical model of the file system, and can be used to complement our basic rules when reasoning about that file system. In this section, we discuss three such properties.

6.1 Container exclusivity

A file system exhibits *container exclusivity* if it guarantees that for every on-disk container, there is at most one dirty copy of the container's contents in the file system cache. It also requires the file system to ensure that the in-memory contents of a container do not change while the container is being written to disk. Many file systems such as BSD FFS, Linux ext2 and VFAT exhibit container exclusivity; some journaling file systems like ext3 do not exhibit this property. In our equations, when we refer to containers in memory, we refer to the latest epoch of the container in memory, in the case of file systems that do not obey container exclusivity. For example, in eq. 10, $\{\&A = \emptyset\}_M$ means that at that time, there is no container whose latest epoch in memory points to A; similarly, $write(B)$ means that the latest epoch of B at that time is being written. When referring to a specific version, we use the epoch notation. Of course, if container exclusivity holds, only one epoch of any container exists in memory.

Under container exclusivity, we have a stronger converse for eq. 3:

$$\{B \to A\}_D \quad \Rightarrow \quad \{B \to A\}_M \prec \{B \to A\}_D \tag{12}$$

If we assume that A is unshared, we have a stronger equation following from equation 12, because the only way the disk belief $\{B \to A\}_D$ can hold is if B was written by the file system. Note that many containers in typical file systems (such as data blocks) are unshared.

$$\begin{aligned}\{B \to A\}_D \quad \Rightarrow \quad & \{B \to A\}_M \prec \\ & (write(B) \ll \{B \to A\}_D)\end{aligned} \tag{13}$$

6.2 Reuse ordering

A file system exhibits *reuse ordering* if it ensures that before reusing a container, it commits the freed state of the container to disk. For example, if A is pointed to by generation b in memory, later freed (*i.e.*, $\&A = \emptyset$), and then another generation c is made to point to A, the freed state of A (*i.e.*, the container of generation b, with its pointer removed) is written to disk before the reuse occurs.

$$\begin{aligned}&\{b \to A\}_M \prec \{\&A = \emptyset\}_M \prec \{c \to A\}_M \\ &\Rightarrow \{\&A = \emptyset\}_M \prec write(C(b)) \ll \{c \to A\}_M\end{aligned}$$

Since every reuse results in such a commit of the freed state, we could extend the above rule as follows:

$$\begin{aligned}&\{b \to A\}_M \ll \{\&A = \emptyset\}_M \prec \{c \to A\}_M \\ &\Rightarrow \{\&A = \emptyset\}_M \prec write(C(b)) \ll \{c \to A\}_M\end{aligned} \tag{14}$$

FFS with soft updates [6] and Linux ext3 are two examples of file systems that exhibit reuse ordering.

6.3 Pointer ordering

A file system exhibits *pointer ordering* if it ensures that before writing a container B to disk, the file system writes all containers that are pointed to by B.

$$\begin{aligned}&\{B \to A\}_M \prec write(B) \\ &\Rightarrow \{B \to A\}_M \prec (write(A) \ll write(B))\end{aligned} \tag{15}$$

FFS with soft updates is an example of a file system that exhibits pointer ordering.

7 Modeling Existing Systems

Having defined the basic formalism of our logic, we proceed to using the logic to model and reason about file system behaviors. In this section, we present proofs for two properties important for file system consistency. First, we discuss the *data consistency* problem in a file system. We then model a journaling file system and reason about the *non-rollback* property in a journaling file system.

7.1 Data consistency

We first consider the problem of *data consistency* of the file system after a crash. By data consistency, we mean that the contents of data block containers have to be consistent with the metadata that references the data blocks. In other words, a file should not end up with data from a different file when the file system recovers after a crash. Let us assume that B is a file metadata container (i.e. contains pointers to the data blocks of the respective file), and A is a data block container. Then, if the disk belief that B^x points to A holds, and the on-disk contents of A were written when k was the generation of A, then epoch B^x should have pointed (at some time in the past) exactly to the k^{th} generation of A in memory, and not a different generation. The following rule summarizes this:

$$\begin{aligned}\{B^x \to A\}_D \wedge \{A^y\}_D \quad \Rightarrow \quad & (\{B^x \to A_k\}_M \ll \{B^x \to A\}_D) \\ & \wedge (k = g(A^y))\end{aligned}$$

We prove below that if the file system exhibits reuse ordering and pointer ordering, it never suffers a data consistency violation. We also show that if the file system does not obey any such ordering, data consistency could be compromised on crashes.

For simplicity, let us make a further assumption that the data containers in our file system are nonshared ($\oplus A$), *i.e.*, different files do not share data block pointers. Let us also assume that the file system obeys the container exclusivity property. Many modern file systems such as ext2 and VFAT have these properties. Since under block exclusivity $\{B^x \to A\}_D \Rightarrow \{B^x \to A\}_M \prec \{B^x \to A\}_D$ (by eq. 12), we can rewrite the above rule as follows:

$$\begin{aligned}&(\{B^x \to A_k\}_M \prec \{B^x \to A\}_D) \wedge \{A^y\}_D \\ &\Rightarrow (k = g(A^y))\end{aligned} \tag{16}$$

If this rule does not hold, it means that the file represented by the generation $g(B^x)$ points to a generation k of A, but the contents of A were written when its generation was $g(A^y)$, clearly a case of data corruption.

To show that this rule does not always hold, we assume the negation and prove that it is reachable as a sequence of valid file system actions ($\alpha \Rightarrow \beta \equiv \neg(\alpha \wedge \neg\beta)$).

From eq. 6, we have $\{A^y\}_D \Rightarrow \{c \to g(A^y)\}_M \prec write(A)$. Thus, we have two event sequences implied by the LHS of eq. 16:

$$\begin{aligned}&i. \quad \{B^x \to A_k\}_M \prec \{B^x \to A\}_D \\ &ii. \quad \{c \to g(A^y)\}_M \prec write(A)\end{aligned}$$

Thus, in order to prove eq. 16, we need to prove that every possible interleaving of the above two sequences, together with the clause $(k \neq g(A^y))$ is invalid. To disprove eq. 16, we need to prove that at least one of the interleavings is valid.

Since $(k \neq g(A^y))$, and since $\{B^x \to A_k\}_M \prec \{B^x \to$

$A\}_D$, the event $\{c \rightarrow g(A^y)\}_M$ cannot occur in between those two events, due to container exclusivity and because A is unshared. Similarly $\{B^x \rightarrow A_k\}_M$ cannot occur between $\{c \rightarrow g(A^y)\}_M \prec write(A)$. Thus, we have only two interleavings:

1. $\{B^x \rightarrow A_k\}_M \prec \{B^x \rightarrow A\}_D \ll \{c \rightarrow g(A^y)\}_M \prec write(A)$
2. $\{c \rightarrow g(A^y)\}_M \prec write(A) \ll \{B^x \rightarrow A_k\}_M \prec \{B^x \rightarrow A\}_D$

Case 1:

Applying eq. 2,

$$\Rightarrow \quad \{g(B^x) \rightarrow A_k\}_M \prec \{B^x \rightarrow A\}_D \ll \{c \rightarrow g(A^y)\}_M \prec write(A) \wedge (k \neq g(A^y))$$

Applying eq. 8,

$$\Rightarrow \quad \{g(B^x) \rightarrow A_k\}_M \prec \{B^x \rightarrow A\}_D \ll \{\&A = \emptyset\}_M \prec \{c \rightarrow g(A^y)\}_M \prec write(A) \tag{17}$$

Since step 17 is a valid sequence in file system execution, where generation A^k could be freed due to a delete of the file represented by generation $g(B^x)$ and then a subsequent generation of the block is reallocated to the file represented by generation c in memory, we have shown that this violation could occur.

Let us now assume that our file system obeys reuse ordering, *i.e.*, equation 14. Under this additional constraint, equation 17 would imply the following:

$$\Rightarrow \quad \{g(B^x) \rightarrow A_k\}_M \prec \{B^x \rightarrow A\}_D \prec \{\&A = \emptyset\}_M \prec write(B) \ll \{c \rightarrow g(A^y)\}_M \prec write(A)$$

By eq. 10,

$$\Rightarrow \quad \{g(B^x) \rightarrow A_k\}_M \prec \{B^x \rightarrow A\}_D \prec \{\&A = \emptyset\}_D \ll \{c \rightarrow g(A^y)\}_M \prec write(A)$$

$$\Rightarrow \quad \{\&A = \emptyset\}_D \wedge \{A_c\}_D \tag{18}$$

This is however, a contradiction under the initial assumption we started off with, i.e. $\{\&A = B\}_D$. Hence, under reuse ordering, we have shown that this particular scenario does not arise at all.

Case 2: $\{c \rightarrow g(A^y)\}_M \prec write(A) \ll \{B^x \rightarrow A_k\}_M \prec \{B^x \rightarrow A\}_D \wedge (k \neq g(A^y))$

Again, applying eq. 2,

$$\Rightarrow \quad (k \neq g(A^y)) \wedge \{c \rightarrow g(A^y)\}_M \prec write(A) \ll \{g(B^x) \rightarrow A_k\}_M \prec \{B^x \rightarrow A\}_D$$

By eqn 8,

$$\Rightarrow \quad \{c \rightarrow g(A^y)\}_M \prec write(A) \ll \{\&A = \emptyset\}_M \prec \{g(B^x) \rightarrow A_k\}_M \prec \{B^x \rightarrow A\}_D \tag{19}$$

Again, this is a valid file system sequence where file generation c pointed to data block generation $g(A^y)$, the generation $g(A^y)$ gets deleted, and a new generation k of container A gets assigned to file generation $g(B^x)$. Thus, consistency violation can also occur in this scenario.

Interestingly, when we apply eq. 14 here, we get

$$\Rightarrow \quad \{c \rightarrow g(A^y)\}_M \prec write(A) \ll \{\&A = \emptyset\}_M \prec write(C(c)) \ll \{g(B^x) \rightarrow A_k\}_M \prec \{B^x \rightarrow A\}_D$$

However, we cannot apply eq. 10 in this case because the belief $\{C \rightarrow A\}_D$ need not hold. Even if we did have a rule that led to the belief $\{\&A = \emptyset\}_D$ immediately after $write(C(c))$, that belief will be overwritten by $\{B^x \rightarrow A\}_D$ later in the sequence. Thus, eq. 14 does not invalidate this sequence; reuse ordering thus does not guarantee data consistency in this case.

Let us now make another assumption, that the file system also obeys pointer ordering (eq. 15).

Since we assume that A is unshared, and that container exclusivity holds, we can apply eq. 13 to equation 19.

$$\Rightarrow \quad \{c \rightarrow g(A^y)\}_M \prec write(A) \ll \{\&A = \emptyset\}_M \prec \{g(B^x) \rightarrow A_k\}_M \prec write(B) \ll \{B^x \rightarrow A\}_D \tag{20}$$

Now applying the pointer ordering rule (eqn 15.),

$$\Rightarrow \quad \{c \rightarrow g(A^y)\}_M \prec write(A) \ll \{\&A = \emptyset\}_M \prec \{g(B^x) \rightarrow A_k\}_M \prec write(A) \ll write(B) \ll \{B^x \rightarrow A\}_D$$

By eq. 7,

$$\Rightarrow \quad \{c \rightarrow A\}_M \prec write(A) \ll \{\&A = \emptyset\}_M \prec \{A^y\}_D \ll write(B) \ll \{B^x \rightarrow A\}_D \wedge (k = g(A^y))$$

$$\Rightarrow \quad \{A^y\}_D \wedge \{B^x \rightarrow A\}_D \wedge (k = g(A^y)) \tag{21}$$

This is again a contradiction, since this implies that the contents of A on disk belong to the same generation A_k, while we started out with the assumption that $g(A^y) \neq k$.

Thus, under reuse ordering and pointer ordering, the file system never suffers a data consistency violation. If the file system does not obey any such ordering (such as ext2), data consistency could be compromised on crashes. Note that this inconsistency is fundamental, and cannot be fixed by scan-based consistency tools such as *fsck*. We also verified that this inconsistency occurs in practice; we were able to reproduce this case experimentally on an ext2 file system.

7.2 Modeling file system journaling

We now extend our logic with rules that define the behavior of a journaling file system. We then use the model to reason about a key property in a journaling file system.

Journaling is a technique commonly used by file systems to ensure metadata consistency. When a single file system operation spans multiple changes to metadata structures, the file system groups those changes into a *transaction* and guarantees that the transaction commits atomically, thus preserving consistency. To provide atomicity, the file system first writes the changes to a *write-*

ahead log (WAL), and propagates the changes to the actual on-disk location only after the transaction is *committed* to the log. A transaction is committed when all changes are logged, and a special "commit" record is written to log indicating completion of the transaction. When the file system recovers after a crash, a checkpointing process *replays* all changes that belong to committed transactions.

To model journaling, we consider a logical "transaction" object that determines the set of *log record* containers that belong to that transaction, and thus logically contains pointers to the log copies of all containers modified in that transaction. We denote the log copy of a journaled container by the ˆ symbol on top of the container name; $\hat{A^x}$ is thus a container in the *log*, *i.e.*, *journal* of the file system (note that we assume *physical logging*, such as the block-level logging in ext3). The physical realization of the transaction object is the "commit" record, since it logically points to all containers that changed in that transaction. For the WAL property to hold, the commit container should be written only after the log copy of all modified containers that the transaction points to are written.

If T is the commit container, the WAL property leads to the following two rules:

$$\{T \to \hat{A^x}\}_M \prec write(T) \quad \Rightarrow \quad \{T \to \hat{A^x}\}_M \prec (write(\hat{A^x}) \ll write(T)) \tag{22}$$

$$\{T \to \hat{A^x}\}_M \prec write(A^x) \quad \Rightarrow \quad \{T \to \hat{A^x}\}_M \prec (write(T) \ll write(A^x)) \tag{23}$$

The first rule states that the transaction is not committed (*i.e.*, commit record not written) until all containers belonging to the transaction are written to disk. The second rule states that the on-disk home copy of a container is written only after the transaction in which the container was modified, is committed to disk. Note that unlike the normal pointers considered so far that point to containers or generations, the pointers from container T in the above two rules point to *epochs*. These *epoch pointers* are used because a commit record is associated with a specific epoch (*e.g.*, snapshot) of the container.

The replay or checkpointing process can be depicted by the following two rules.

$$\{T \to \hat{A^x}\}_D \wedge \{T\}_D \;\Rightarrow\; write(A^x) \ll \{A^x\}_D \tag{24}$$

$$\{T_1 \to \hat{A^x}\}_D \wedge \{T_2 \to \hat{A^y}\}_D \wedge (\{T_1\}_D \ll \{T_2\}_D) \Rightarrow write(A^y) \ll \{A^y\}_D \tag{25}$$

The first rule says that if a container is part of a transaction and the transaction is committed on disk, the on-disk copy of the container is updated with the logged copy pertaining to that transaction. The second rule says that if the same container is part of multiple committed transactions, the on-disk copy of the container is updated with the copy pertaining to the last of those transactions.

The following belief transitions hold:

$$(\{T \to \hat{B^x}\}_M \wedge \{B^x \to A\}_M) \prec write(T) \Rightarrow \{B^x \to A\}_D \tag{26}$$

$$\{T \to \hat{A^x}\}_M \prec write(T) \Rightarrow \{A^x\}_D \tag{27}$$

Rule 26 states that if B^x points to A and $\hat{B^x}$ belongs to transaction T, the commit of T leads to the disk belief $\{B^x \to A\}_D$. Rule 27 says that the disk belief $\{A^x\}_D$ holds immediately on commit of the transaction which $\hat{A^x}$ is part of; creation of the belief does not require the checkpoint write to happen. As described in §5.1, a disk belief pertains to the belief the file system would reach, if it were to start from the current disk state.

In certain journaling file systems, it is possible that only containers of certain types are journaled; updates to other containers directly go to disk, without going through the transaction machinery. In our proofs, we will consider the cases of both complete journaling (where all containers are journaled) and selective journaling (only containers of a certain type). In the selective case, we also address the possibility of a container changing its type from a journaled type to a non-journaled type and vice versa. For a container B that belongs to a journaling type, we have the following converse of equation 26:

$$\{B^x \to A\}_D \quad \Rightarrow \quad (\{T \to \hat{B^x}\}_M \wedge \{B^x \to A\}_M) \prec write(T) \ll \{B^x \to A\}_D \tag{28}$$

We can show that in complete journaling, data inconsistency never occurs; we omit this due to space constraints.

7.2.1 The non-rollback property

We now introduce a new property called *non-rollback* that is pertinent to file system consistency. We first formally define the property and then reason about the conditions required for it to hold in a journaling file system.

The non-rollback property states that the contents of a container on disk are never overwritten by *older* contents from a previous epoch. This property can be expressed as:

$$\{A^x\}_D \ll \{A^y\}_D \quad \Rightarrow \quad \{A^x\}_M \ll \{A^y\}_M \tag{29}$$

The above rule states that if the on-disk contents of A move from epoch x to y, it should logically imply that epoch x occurred before epoch y in memory as well. The non-rollback property is crucial in journaling file systems; absence of the property could lead to data corruption.

In the proof below, we logically derive the corner cases that need to be handled for this property to hold, and show that *journal revoke records* effectively ensure this.

If the disk believes in the x^{th} epoch of A, there are only two possibilities. If the type of A^x was a journaled type, A^x should have belonged to a transaction and the disk must have observed the commit record for the transaction; as indicated in eq 27, the belief of $\{A^x\}_D$ occurs immediately after the commit. However, at a later point the actual contents of A^x will be written by the file system as part of its checkpoint propagation to the actual on-disk location, thus re-establishing belief $\{A^x\}_D$. If J is the set of all journaled types,

$$\{A^x\}_D \wedge \{t(A^x) \in J\}_M \Rightarrow \begin{array}{l}(\{A^x\}_M \wedge \{T \rightarrow \hat{A^x}\}_M) \\ \prec write(T) \ll \{A^x\}_D \\ \ll write(A^x) \ll \{A^x\}_D\end{array} \quad (30)$$

The second possibility is that A^x is of a type that is not journaled. In this case, the only way the disk could have learnt of it is by a prior commit of A^x.

$$\{A^x\}_D \wedge \{t(A^x) \notin J\}_M \quad \Rightarrow \quad \begin{array}{l}\{A^x\}_M \prec write(A^x) \\ \ll \{A^x\}_D\end{array} \quad (31)$$

A^x and A^y are journaled:

Let us first assume that both A^x and A^y belong to a journaled type. To prove the non-rollback property, we consider the LHS of eq. 29: $\{A^x\}_D \ll \{A^y\}_D$; since both A^x and A^y are journaled, we have the following two sequence of events that led to the two beliefs (by eq. 30):

$$(\{A^x\}_M \wedge \{T_1 \rightarrow \hat{A^x}\}_M) \prec write(T_1) \ll \{A^x\}_D \ll write(A^x) \ll \{A^x\}_D$$

$$(\{A^y\}_M \wedge \{T_2 \rightarrow \hat{A^y}\}_M) \prec write(T_2) \ll \{A^y\}_D \ll write(A^y) \ll \{A^y\}_D$$

Omitting the $write$ actions in the above sequences for simplicity, we have the following sequences of events:

$$\begin{array}{ll} i. & \{A^x\}_M \ll \{A^x\}_D \ll \{A^x\}_D \\ ii. & \{A^y\}_M \ll \{A^y\}_D \ll \{A^y\}_D \end{array}$$

Note that in each sequence, there are two instances of the *same* disk belief being created: the first instance is created when the corresponding transaction is committed, and the second instance when the checkpoint propagation happens at a later time. In snapshot-based coarse-grained journaling systems (such as ext3), transactions are always committed in order. Thus, if epoch A^x occurred before A^y, T_1 will be committed before T_2 (*i.e.*, the first instance of $\{A^x\}_D$ will occur before the first instance of $\{A^y\}_D$). Another property true of such journaling is that the checkpointing is in-order as well; if there are two committed transactions with different copies of the same data, only the version pertaining to the later transaction is propagated during the checkpoint.

Thus, the above two sequences of events lead to only two interleavings, depending on whether epoch x occurs before epoch y or vice versa. Once the ordering between epoch x and y is fixed, the rest of the events are constrained to a single sequence:

Interleaving 1:

$$\begin{array}{ll} & (\{A^x\}_M \ll \{A^y\}_M) \wedge (\{A^x\}_D \ll \{A^y\}_D \ll \{A^y\}_D) \\ \Rightarrow & \{A^x\}_M \ll \{A^y\}_M \end{array}$$

Interleaving 2:

$$\begin{array}{ll} \Rightarrow & (\{A^y\}_M \ll \{A^x\}_M) \wedge (\{A^y\}_D \ll \{A^x\}_D \ll \{A^x\}_D) \\ \Rightarrow & \{A^y\}_D \ll \{A^x\}_D \end{array}$$

Thus, the second interleaving results in a contradiction from our initial statement we started with (*i.e.*, $\{A^x\}_D \ll \{A^y\}_D$). Therefore the first interleaving is the only legal way the two sequences of events could be combined. Since the first interleaving implies that $\{A^x\}_M \ll \{A^y\}_M$, we have proved that if the two epochs are journaled, the non-rollback property holds.

A^y is journaled, but A^x is not:

We now consider the case where the type of A changes between epochs x and y, such that A^y belongs to a journaled type and A^x does not.

We again start with the statement $\{A^x\}_D \ll \{A^y\}_D$. From equations 30 and 31, we have the following two sequences of events:

$$\begin{array}{ll} i. & (\{A^y\}_M \wedge \{T \rightarrow \hat{A^y}\}_M) \prec write(T) \\ & \ll \{A^y\}_D \ll write(A^y) \ll \{A^y\}_D \\ ii. & \{A^x\}_M \prec write(A^x) \ll \{A^x\}_D \end{array}$$

Omitting the $write$ actions for the sake of readability, the sequences become:

$$\begin{array}{ll} i. & \{A^y\}_M \ll \{A^y\}_D \ll \{A^y\}_D \\ ii. & \{A^x\}_M \ll \{A^x\}_D \end{array}$$

To prove the non-rollback property, we need to show that every possible interleaving of the above two sequences where $\{A^y\}_M \ll \{A^x\}_M$ results in a contradiction, *i.e.*, cannot co-exist with $\{A^x\}_D \ll \{A^y\}_D$.

The interleavings where $\{A^y\}_M \ll \{A^x\}_M$ are:

1. $\{A^y\}_M \ll \{A^x\}_M \ll \{A^x\}_D \ll \{A^y\}_D \ll \{A^y\}_D$
2. $\{A^y\}_M \ll \{A^y\}_D \ll \{A^x\}_M \ll \{A^x\}_D \ll \{A^y\}_D$
3. $\{A^y\}_M \ll \{A^y\}_D \ll \{A^y\}_D \ll \{A^x\}_M \ll \{A^x\}_D$
4. $\{A^y\}_M \ll \{A^x\}_M \ll \{A^y\}_D \ll \{A^x\}_D \ll \{A^y\}_D$
5. $\{A^y\}_M \ll \{A^x\}_M \ll \{A^y\}_D \ll \{A^y\}_D \ll \{A^x\}_D$
6. $\{A^y\}_M \ll \{A^y\}_D \ll \{A^x\}_M \ll \{A^y\}_D \ll \{A^x\}_D$

Scenarios 3, 5, and 6 imply $\{A^y\}_D \ll \{A^x\}_D$ and are therefore invalid interleavings. Scenarios 1, 2, and 4 are valid interleavings that do not contradict our initial assumption of disk beliefs, but at the same time, imply $\{A^y\}_M \ll \{A^x\}_M$; these scenarios thus violate the non-rollback property. Therefore, under dynamic typing, the above journaling mechanism does not guarantee non-rollback. Due to this violation, file contents can be corrupted by stale metadata generations.

Scenario 2 and 4 occur because the checkpoint propagation of earlier epoch A^y which was journaled, occurs *after* A was overwritten as a non-journaled epoch. To prevent this, we need to impose that the checkpoint propagation of a container in the context of transaction T does not happen if the on-disk contents of that container were updated *after* the commit of T. The *journal revoke records* in ext3 precisely guarantee this; if a revoke record is encountered during log replay (during a pre-scan of the log), the corresponding block is not propagated to the actual disk location.

Scenario 1 happens because a later epoch of A is committed to disk before the transaction which modified an earlier epoch is committed. To prevent this, we need a form of *reuse ordering*, which imposes that before a container changes type (i.e. is reused in memory), the transaction that freed the previous generation be committed. Since transactions commit in order, and the freeing transaction should occur *after* transaction T which used A^y in the above example, we have the following guarantee:

$$\{t(A^y) \in J\}_M \wedge \{t(A^x) \notin J\}_M \wedge (\{A^y\}_M \ll \{A^x\}_M) \Rightarrow \{A^y\}_M \prec write(T) \ll \{A^x\}_M$$

With this rule, Scenario 1 becomes the same as 2 and 4 and is handled by the revoke record solution. Thus, under these two properties, the non-rollback property holds.

8 Redundant Synchrony in Ext3

We examine a performance problem with the ext3 file system where the transaction commit procedure artificially limits parallelism due to a redundant synchrony in its disk writes [16]. The *ordered mode* of ext3 guarantees that a newly created file will never point to stale data blocks after a crash. Ext3 ensures this guarantee by the following ordering in its commit procedure: when a transaction is committed, ext3 first writes to disk the data blocks allocated in that transaction, waits for those writes to complete, then writes the journal blocks to disk, waits for those to complete, and then writes the commit block. If I is an inode container, F is a file data block container, and T is the transaction commit container, the commit procedure of ext3 can be expressed by the following equation:

$$\begin{aligned}&(\{I^x \to F_k\}_M \wedge \{T \to \hat{I^x}\}_M) \prec write(T)\\ &\Rightarrow (\{I^x \to F_k\}_M \wedge \{T \to \hat{I^x}\}_M)\\ &\quad \prec write(F) \ll write(\hat{I^x}) \ll write(T)\end{aligned} \tag{32}$$

To examine if this is a necessary condition to ensure the no-stale-data guarantee, we first formally depict the guarantee that the ext3 ordered mode seeks to provide, in the following equation:

$$\{I^x \to F_k\}_M \ll \{I^x \to F\}_D \quad \Rightarrow \quad \{F_y\}_D \ll \{I^x \to F\}_D \wedge (g(F^y) = k) \tag{33}$$

The above equation states that if the disk acquires the belief that $\{I^x \to F\}$, then the contents of the data container F on disk should *already* pertain to the generation of F that I^x pointed to in memory. Note that because ext3 obeys reuse ordering, the ordered mode guarantee only needs to cater to the case of a *free* data block container being allocated to a new file.

We now prove equation 33, examining the conditions that need to hold for this equation to be true. We consider the LHS of the equation:

$$\{I^x \to F_k\}_M \ll \{I^x \to F\}_D$$

Applying equation 28 to the above, we get

$$\begin{aligned}\Rightarrow \quad &(\{I^x \to F_k\}_M \wedge \{T \to \hat{I^x}\}_M) \prec\\ &write(T) \ll \{I^x \to F\}_D\end{aligned}$$

Applying equation 32, we get

$$\begin{aligned}\Rightarrow \quad &(\{I^x \to F_k\}_M \wedge \{T \to \hat{I^x}\}_M) \prec\\ &write(F) \ll write(\hat{I^x}) \ll\\ &write(T) \ll \{I^x \to F\}_D\end{aligned} \tag{34}$$

By equation 7,

$$\begin{aligned}\Rightarrow \quad &(\{I^x \to F_k\}_M \wedge \{T \to \hat{I^x}\}_M) \prec\\ &\{F^y\}_D \ll write(\hat{I^x}) \ll\\ &write(T) \ll \{I^x \to F\}_D \wedge (g(F^y) = k)\\ \Rightarrow \quad &\{F^y\}_D \ll \{I^x \to F\}_D \wedge (g(F^y) = k)\end{aligned}$$

Thus, the current ext3 commit procedure (equation 32) guarantees the no-stale-data property. However, to see if all the waits in the above procedure are required, let us reorder the two actions $write(F)$ and $write(\hat{I^x})$ in eq. 34:

$$\begin{aligned}\Rightarrow \quad &(\{I^x \to F_k\}_M \wedge \{T \to \hat{I^x}\}_M) \prec\\ &write(\hat{I^x}) \ll write(F) \ll\\ &write(T) \ll \{I^x \to F\}_D\end{aligned}$$

Once again, applying equation 7,

$$\Rightarrow \quad \{F^y\}_D \ll \{I^x \to F\}_D \wedge (g(F^y) = k)$$

Thus, we can see that the ordering between the actions $write(F)$ and $write(\hat{I^x})$ is inconsequential to the guarantee that ext3 ordered mode attempts to provide. We can hence conclude that the wait that ext3 employs after the write to data blocks is redundant, and unnecessarily limits parallelism between data and journal writes. This can have especially severe performance implications in settings where the log is stored on a separate disk, as illustrated in previous work [16].

We believe that this specific example points to a general problem with file system design. When developers do not have rigorous frameworks to reason about correctness, they tend to be *conservative*. Such conservatism often translates into unexploited opportunities for performance optimization. A systematic framework enables aggressive optimizations while ensuring correctness.

9 Support for Consistent Undelete

In this section, we demonstrate that our logic enables one to quickly formulate and prove properties about new file system features and mechanisms. We explore a functionality that is traditionally not considered a part of core file system design: the ability to *undelete* deleted files with certain consistency guarantees. The ability to recover deleted files is useful, as demonstrated by the large number of tools available for the purpose [17, 19]. Such tools try to rebuild deleted files by scavenging through on-disk metadata; this is possible to an extent because file systems do not normally zero out freed metadata containers (they are simply marked free). For example, in a UNIX file system, the block pointers in a deleted inode would indicate the blocks that used to belong to that deleted file.

However, none of the existing tools for undelete can guarantee *consistency* (*i.e.*, assert that the recovered contents are valid). While undelete is fundamentally only best-effort (files cannot be recovered if the blocks were subsequently reused in another file), the user needs to know how trustworthy the recovered contents are. We demonstrate using our logic that with existing file systems, such *consistent* undelete is impossible. We then provide a simple solution, and prove that the solution guarantees consistent undelete. Finally, we present an implementation of the solution in ext3.

9.1 Undelete in existing systems

To model undelete, the logic needs to express pointers from containers holding a *dead* generation. We introduce the $\rightsquigarrow$ notation to indicate such a pointer, which we call a *dead pointer*. We also define a new operator $\tilde{\&}$ on a container that denotes the set of all dead *and* live entities pointing to the container. Let $undel(B)$ be the undelete action on container B. The undelete process can be summarized by the following equation:

$$undel(B) \wedge \{B^x \rightsquigarrow A\}_D \wedge \{\tilde{\&}A = \{B\}\}_D$$
$$\Leftrightarrow \{B^x \rightsquigarrow A\}_D \prec \{B^y \rightarrow A\}_D \wedge (g(B^x) = g(B^y)) \quad (35)$$

In other words, if the dead (free) container B^x points to A on disk, and is the only container (alive or dead) pointing to A, the undelete makes the generation $g(B^x)$ live again, and makes it point to A.

The guarantee we want to hold for consistency is that if a dead pointer from B^x to A is brought alive, the on-disk contents of A at the time the pointer is brought alive must correspond to the same generation that epoch B^x originally pointed to in memory (similar to the data consistency formulation in §7.1):

$$\{B^x \rightarrow A_k\}_M \ll \{B^x \rightsquigarrow A\}_D \prec \{B^y \rightarrow A\}_D$$
$$\wedge (g(B^x) = g(B^y))$$
$$\Rightarrow \{B^x \rightsquigarrow A\}_D \wedge \{A^z\}_D \wedge (g(A^z) = k)$$

Note that the clause $g(B^x) = g(B^y)$ is required in the LHS to cover only the case where the *same* generation is brought to life, which would be true only for undelete.

To show that the above guarantee does not hold necessarily, we consider the negation of the RHS, *i.e.*, $\{A^z\}_D \wedge (g(A^z) \neq k)$, and show that this condition can co-exist with the conditions required for undelete as described in equation 35. In other words, we show that $undel(B) \wedge \{B^x \rightsquigarrow A\}_D \wedge \{\tilde{\&}A = \{B\}\}_D \wedge \{A^z\}_D \wedge (g(A^z) \neq k)$ can arise from valid file system execution.

We utilize the following implications for the proof:

$$\{B^x \rightsquigarrow A\}_D \quad \Leftrightarrow \quad \{B^x \rightarrow A_k\}_M \prec \{\&A = \emptyset\}_M \prec write(B)$$
$$\{A^z\}_D \quad \Rightarrow \quad \{c \rightarrow g(A^z)\}_M \prec write(A) \text{ (eq. 6)}$$

Let us consider one possible interleaving of the above two event sequences:

$\{c \rightarrow g(A^z)\}_M \prec write(A) \ll \{B^x \rightarrow A_k\}_M \prec \{\&A = \emptyset\}_M \prec write(B)$

This is a valid file system sequence where a file represented by generation c points to $g(A^z)$, A^z is written to disk, then block A is freed from c thus killing the generation $g(A^z)$, and a new generation A_k of A is then allocated to the generation $g(B^x)$. Now, when $g(B^x)$ is deleted, and B is written to disk, the disk has both beliefs $\{B^x \rightsquigarrow A\}_D$ and $\{A^z\}_D$. Further, if the initial state of the disk was $\{\tilde{\&}A = \emptyset\}_D$, the above sequence would also simultaneously lead to the disk belief $\{\tilde{\&}A = \{B\}\}_D$. Thus we have shown that the conditions $\{B^x \rightsquigarrow A\}_D \wedge \{\tilde{\&}A = \{B\}\}_D \wedge \{A^z\}_D \wedge (k \neq g(A^z))$ can hold simultaneously. An undelete of B at this point would lead to violation of the consistency guarantee, because it would associate a stale generation of A with the undeleted file $g(B^x)$. It can be shown that neither reuse ordering nor pointer ordering would guarantee consistency in this case.

9.2 Undelete with generation pointers

We now propose the notion of *generation pointers* and show that with such pointers, consistent undelete is guaranteed. So far, we have assumed that *pointers* on disk point to *containers* (as discussed in Section 4). If instead, each pointer pointed to a specific *generation*, it leads to a different set of file system properties. To implement such "generation pointers", each on-disk container contains a generation number that gets incremented every time the container is reused. In addition, every on-disk pointer will embed this generation number in addition to the container name. With generation pointers, the on-disk contents of a container will implicitly indicate its generation. Thus, $\{B_k\}_D$ is a valid belief; it means that the disk knows the contents of B belong to generation k.

Under generation pointers, the criterion for doing undelete (eq. 35) becomes:

$$undel(B) \wedge \{B^x \rightsquigarrow A_k\}_D \wedge \{A_k\}_D$$
$$\Leftrightarrow \{B^x \rightsquigarrow A_k\}_D \prec \{B^y \rightarrow A_k\}_D \quad (36)$$

Let us introduce an additional constraint $\{A^z\}_D \wedge (k \neq g(A^z))$ into the left hand side of the above equation (as we did in the previous subsection):

$$\{B^x \rightsquigarrow A_k\}_D \wedge \{A_k\}_D \wedge \{A^z\}_D \wedge (k \neq g(A^z)) \quad (37)$$

Since $k \neq g(A^z)$, let us denote $g(A^z)$ as h. Since every on-disk container holds the generation number too, we have $\{A_h\}_D$. Thus, the above equation becomes:

$$\{B^x \rightsquigarrow A_k\}_D \wedge \{A_k\}_D \wedge \{A_h\}_D \wedge (k \neq h)$$

This is clearly a contradiction, since it means the on-disk container A has the two different generations k and h simultaneously. Thus, it follows that an undelete would not occur in this scenario (or alternatively, this would be flagged as inconsistent). Thus, all undeletes occurring under generation pointers are consistent.

9.3 Implementation of undelete in ext3

Following on the proof for consistent undelete, we implemented the generation pointer mechanism in Linux ext3. Each block has a generation number that gets incremented every time the block is reused. Although the generation numbers are maintained in a separate set of blocks, ensuring atomic commit of the generation number and the block data is straightforward in the data journaling mode of ext3, where we simply add the generation update to the create transaction. The block pointers in the inode are also extended with the generation number of the block. We implemented a tool for undelete that scans over the on-disk structures, restoring all files that can be undeleted *consistently*. Specifically, a file is restored if the generation information in all its metadata block pointers match with the corresponding block generation of the data blocks.

We ran a simple microbenchmark creating and deleting various directories from the linux kernel source tree, and observed that out of roughly 12,200 deleted files, 2970 files (roughly 25%) were detected to be inconsistent and not undeletable, while the remaining files were successfully undeleted. This illustrates that the scenario proved in Section 9.1 actually occurs in practice; an undelete tool without generation information would wrongly restore these files with corrupt or misleading data.

10 Application to Semantic Disks

An interesting application of a logic framework for file systems is that it enables reasoning about a recently proposed class of storage systems called "semantically-smart" disk systems (SDS). An SDS exploits file system information within the storage system, to provide better functionality [20]. However, as admitted by the authors [21], reasoning about the correctness of knowledge tracked in a semantic disk is quite hard. Our formalism of memory and disk beliefs fits the SDS model, since the extra file system state tracked by an SDS is essentially a disk belief. In this section, we first use our logic to explore the feasibility of tracking block type within a semantic disk. We then show that the usage of generation pointers by the file system simplifies information tracking within an SDS.

10.1 Block typing

An important piece of information required within a semantic disk is the *type* of a disk container [21]. While identifying the type of statically-typed containers is straightforward, dynamically typed containers are hard to deal with. The type of a dynamically typed container is determined by the contents of a *parent* container; for example, an indirect pointer block can be identified only by observing a parent inode that has this block in its indirect pointer field. Thus, tracking dynamically typed containers requires correlating type information from a type-determining parent, and then using the information to interpret the contents of the dynamic container.

For accurate type detection in an SDS, we want the following guarantee to hold:

$$\{t(A^x) = k\}_D \quad \Rightarrow \quad \{t(A^x) = k\}_M \quad (38)$$

In other words, if the disk interprets the contents of an epoch A^x to be belonging to type k, those contents should have belonged to type k in memory as well. This guarantees, for example, that the disk would not wrongly interpret the contents of a normal data block container as an indirect block container. Note however that the equation does not impose any guarantee on *when* the disk identifies the type of a container; it only states that whenever it does, the association of type with the contents is correct.

To prove this, we first state an algorithm of how the disk arrives at a belief about a certain type [21]. An SDS snoops on metadata traffic, looking for type-determining containers such as inodes. When such a container is written, it observes the pointers within the container and concludes on the type of each of the pointers. Let us assume that one such pointer of type k points to container A. The disk then examines if container A was written since the last time it was freed. If yes, it interprets the current contents of A as belonging to type k. If not, when A is written at a later time, the contents are associated with type k. We have the following equation:

$$\{t(A^x) = k\}_D \quad \Rightarrow \quad \{B^y \to A\}_D \wedge (f(B^y, A) = k) \\ \wedge \{A^x\}_D \quad (39)$$

In other words, to interpret A^x as belonging to type k, the disk must believe that some container B points to A, and the current on-disk epoch of B (*i.e.*, B^y) must indicate that A is of type k; the function $f(B^y, A)$ abstracts this indication. Further, the disk must contain the contents of epoch A^x in order to associate the contents with type k.

Let us explore the logical events that should have led to each of the components on the right side of equation 39. Applying eq. 12,

$$\{B^y \to A\}_D \wedge (f(B^y, A) = k)$$

$$\Rightarrow (\{B^y \rightarrow A\}_M \wedge\ (f(B^y, A) = k)) \prec\ \{B^y \rightarrow A\}_D$$
$$\Rightarrow (\{B^y \rightarrow A\}_M \wedge\ \{t(A) = k\}_M) \prec\ \{B^y \rightarrow A\}_D \quad (40)$$

Similarly for the other component $\{A^x\}_D$,

$$\{A^x\}_D \quad \Rightarrow \quad write(A^x) \ll \{A^x\}_D \qquad (41)$$

To verify the guarantee in equation 38, we assume that it does not hold, and then observe if it leads to a valid scenario. Thus, we can add the clause $\{t(A^x) = j\}_M \wedge (j \neq k)$ to equation 39, and our equation to prove is:

$$\{B^y \rightarrow A\}_D \wedge\ (f(B^y, A) = k) \wedge\ \{A^x\}_D \wedge\ \{t(A^x) = j\}_M$$

We thus have two event sequences (from eq. 40 and 41):

1. $(\{B^y \rightarrow A\}_M \wedge\ \{t(A) = k\}_M) \prec\ \{B^y \rightarrow A\}_D$
2. $\{t(A^x) = j\}_M \wedge\ write(A^x)$

Since the type of an epoch is unique, and a $write$ of a container implies that it already has a type,
$\{t(A^x) = j\}_M \wedge\ write(A^x) \Rightarrow\ \{t(A^x) = j\}_M \prec\ write(A^x)$.

These sequences can only be interleaved in two ways. The epoch A^x occurs either before or after the epoch in which $\{t(A) = k\}_M$.

Interleaving 1:

$$(\{B^y \rightarrow A\}_M \wedge\ \{t(A) = k\}_M) \prec\ \{B^y \rightarrow A\}_D$$
$$\ll\ \{t(A^x) = j\}_M \prec\ write(A^x)$$

By eq. 11,

$$\Rightarrow \quad (\{B^y \rightarrow A\}_M \wedge\ \{t(A) = k\}_M) \prec\ \{B^y \rightarrow A\}_D$$
$$\ll\ \{\&A = \emptyset\}_M \prec\ \{t(A^x) = j\}_M \prec\ write(A^x)$$

This is a valid sequence where the container A is freed after the disk acquired the belief $\{B \rightarrow A\}$ and a later version of A is then written when its actual type has changed to j in memory, thus leading to incorrect interpretation of A^x as belonging to type k.

However, in order to prevent this scenario, we simply need the reuse ordering rule (eq. 14). With that rule, the above sequence would imply the following:

$$\Rightarrow \quad (\{B^y \rightarrow A\}_M \wedge\ \{t(A) = k\}_M) \prec\ \{B^y \rightarrow A\}_D$$
$$\ll\ \{\&A = \emptyset\}_M \prec\ write(B) \ll\ \{t(A^x) = j\}_M \prec\ write(A^x)$$
$$\Rightarrow \quad (\{B^y \rightarrow A\}_M \wedge\ \{t(A) = k\}_M) \prec\ \{B^y \rightarrow A\}_D$$
$$\ll\ \{\&A = \emptyset\}_D \prec\ \{t(A^x) = j\}_M \prec\ write(A^x)$$

Thus, when A^x is written, the disk will be treating A as free, and hence will not wrongly associate A with type k.

Interleaving 2:

Proceeding similarly with the second interleaving where epoch A^x occurs before A is assigned type k, we arrive at the following sequence:

$$\Rightarrow \quad \{t(A^x) = j\}_M \prec\ write(A^x) \ll\ \{\&A = \emptyset\}_M$$
$$\prec\ (\{B^y \rightarrow A\}_M \wedge\ \{t(A) = k\}_M) \prec\ \{B^y \rightarrow A\}_D$$

We can see that simply applying the reuse ordering rule does not prevent this sequence. We need a stronger form of reuse ordering where the "freed state" of A includes not only the containers that pointed to A, but also the allocation structure $|A|$ tracking liveness of A. With this rule, the above sequence becomes:

$$\Rightarrow \quad \{t(A^x) = j\}_M \prec\ write(A^x) \ll\ \{\&A = \emptyset\}_M$$
$$\prec\ write(|A|) \ll\ (\{B^y \rightarrow A\}_M \wedge\ \{t(A) = k\}_M)$$
$$\prec\ \{B^y \rightarrow A\}_D \qquad (42)$$

We also need to add a new behavior to the SDS which states that when the SDS observes an allocation structure indicating that A is free, it inherits the belief that A is free.

$$\{\&A = \emptyset\}_M \prec\ write(|A|) \quad \Rightarrow \quad \{\&A = \emptyset\}_D$$

Applying the above SDS operation to eqn 42, we get

$$\Rightarrow \quad \{t(A^x) = j\}_M \prec\ write(A^x) \ll\ \{\&A = \emptyset\}_D$$
$$\ll\ (\{B^y \rightarrow A\}_M \wedge\ \{t(A) = k\}_M) \prec\ \{B^y \rightarrow A\}_D$$

In this sequence, because the SDS does not observe a write of A since it was treated as "free", it will not associate type k to A until A is subsequently written.

Thus, we have shown that an SDS cannot accurately track dynamic type underneath a file system without any ordering guarantees. We have also shown that if the file system exhibits a strong form of reuse ordering, dynamic type detection can indeed be made reliable within an SDS.

10.2 Utility of generation pointers

In this subsection, we explore the utility of file system-level "generation pointers" (§ 9.2) in the context of an SDS. To illustrate their utility, we show that tracking dynamic type in an SDS is straightforward if the file system tracks generation pointers.

With generation pointers, equation 39 becomes:

$$\{t(A_g) = k\}_D \quad \Rightarrow \quad \{B^y \rightarrow A_g\}_D \wedge\ (f(B^y, A_g) = k)$$
$$\wedge\ \{A_g\}_D$$

The two causal event sequences (as explored in the previous subsection) become:

$$(\{B^y \rightarrow A_g\}_M \wedge\ \{t(A_g) = k\}_M) \prec\ \{B^y \rightarrow A_g\}_D$$

$$\{t(A_g) = j\}_M \wedge\ write(A_g)$$

Since the above sequences imply that the same generation g had two different types, it violates rule 11. Thus, we straightaway arrive at a contradiction that proves that violation of rule 38 can never occur.

11 Related Work

Previous work has recognized the need for modeling complex systems with formal frameworks, in order to facilitate proving correctness properties about them. The logical framework for reasoning about authentication protocols, proposed by Burrows *et al.* [4], is the most related to our work in spirit; in that paper, the authors formulate a domain-specific logic and proof system for authentication, showing that protocols can be verified through simple logical derivations. Other domain-specific formal models exist in the areas of database recovery [9] and database reliability [7].

A different body of related work involves generic frameworks for modeling computer systems. The well-known TLA framework [10] is an example. The I/O automaton [1] is another such framework. While these frameworks are general enough to model most complex systems, their generality is also a curse; modeling various aspects of a file system to the extent we have in this paper, is quite tedious with a generic framework. Tailoring the framework by using domain-specific knowledge makes it simpler to reason about properties using the framework, thus significantly lowering the barrier to entry in terms of adopting the framework [4]. Specifications and proofs in our logic take 10 to 20 lines in contrast to the thousands of lines that TLA specifications take [25]. However, automated theorem-proving through model checkers is one of the benefits of using a generic framework such as TLA.

Previous work has also explored verification of the correctness of system *implementations*. The recent body of work on using model checking to verify implementations is one example [14, 24]. We believe that this body of work is complementary to our logic framework; our logic framework can be used to build the model and the invariants that should hold in the model, which the implementation can be verified against.

Finally, the file system properties we have listed in Section 6 have been identified in previous work on soft updates [6] and more recent work on semantic disks [20].

12 Conclusions

As the need for dependability of computer systems becomes more important than ever, it is essential to have systematic formal frameworks to verify and reason about their correctness. Despite file systems being a critical component of system dependability, formal verification of their correctness has been largely ignored. Besides making file systems vulnerable to hidden errors, the absence of a formal framework also stifles innovation, because of the skepticism towards the correctness of new proposals, and the proclivity to stick to "time-tested" alternatives. In this paper, we have taken a step towards bridging this gap in file system design by showing that a logical framework can substantially simplify and systematize the process of verifying file system correctness.

Acknowledgements

We would like to thank Lakshmi Bairavasundaram, Nathan Burnett, Timothy Denehy, Rajasekar Krishnamurthy, Florentina Popovici, Vijayan Prabhakaran, and Vinod Yegneswaran for their comments on earlier drafts of this paper. We also thank the anonymous reviewers for their excellent feedback and comments, many of which have greatly improved this paper.

This work is sponsored by NSF CCR-0092840, CCR-0133456, CCR-0098274, NGS-0103670, ITR-0325267, IBM, Network Appliance, and EMC.

References

[1] P. C. Attie and N. A. Lynch. Dynamic Input/Output Automata, a Formal Model for Dynamic Systems. In *ACM PODC*, 2001.

[2] S. Best. JFS Overview. www.ibm.com/developerworks/library/l-jfs.html, 2004.

[3] N. Bjorner, A. Browne, M. Colon, B. Finkbeiner, Z. Manna, H. Sipma, and T. E. Uribe. Verifying temporal properties of reactive systems: A STeP tutorial. *Formal Methods in System Design (FMSD)*, 16(3):227–270, 2000.

[4] M. Burrows, M. Abadi, and R. Needham. A Logic of Authentication. In *ACM SOSP*, pages 1–13, 1989.

[5] E. Clarke, O. Grumberg, and D. Peled. *Model Checking*. The MIT Press, 2000.

[6] G. R. Ganger, M. K. McKusick, C. A. Soules, and Y. N. Patt. Soft Updates: A Solution to the Metadata Update Problem in File Systems. *ACM TOCS*, 18(2), May 2000.

[7] V. Hadzilacos. A Theory of Reliability in Database Systems. *J. ACM*, 35(1):121–145, 1988.

[8] R. Hagmann. Reimplementing the Cedar File System Using Logging and Group Commit. In *SOSP '87*, Nov. 1987.

[9] D. Kuo. Model and Verification of a Data Manager Based on ARIES. *ACM Trans. Database Systems*, 21(4):427–479, 1996.

[10] L. Lamport. The Temporal Logic of Actions. *ACM Trans. Program. Lang. Syst.*, 16(3):872–923, 1994.

[11] M. K. McKusick, W. N. Joy, S. J. Leffler, and R. S. Fabry. A Fast File System for UNIX. *ACM Transactions on Computer Systems*, 2(3):181–197, August 1984.

[12] J. C. Mogul. A Better Update Policy. In *USENIX Summer '94*, Boston, MA, June 1994.

[13] C. Mohan, D. Haderle, B. Lindsay, H. Pirahesh, and P. S. z. ARIES: A Transaction Recovery Method Supporting Fine-Granularity Locking and P artial Rollbacks Using Write-Ahead Logging. *ACM TODS*, 17(1):94–162, March 1992.

[14] M. Musuvathi, D. Y. Park, A. Chou, D. R. Engler, and D. L. Dill. CMC: A pragmatic approach to model checking real code. In *OSDI '02*, Dec. 2002.

[15] A. Pnueli. The temporal semantics of concurrent programs. *Theoretical Computer Science (TCS)*, 13:45–60, 1981.

[16] V. Prabhakaran, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Analysis and Evolution of Journaling File Systems. In *USENIX '05*, 2005.

[17] R-Undelete. R-Undelete File Recovery Software. http://www.r-undelete.com/.

[18] H. Reiser. ReiserFS. www.namesys.com, 2004.

[19] Restorer2000. Restorer 2000 Data Recovery Software. http://www.bitmart.net/.

[20] M. Sivathanu, L. Bairavasundaram, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Life or Death at Block Level. In *OSDI '04*, pages 379–394, San Francisco, CA, December 2004.

[21] M. Sivathanu, V. Prabhakaran, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Improving Storage System Availability with D-GRAID. In *FAST04*, 2004.

[22] T. Ts'o and S. Tweedie. Future Directions for the Ext2/3 Filesystem. In *FREENIX '02*, Monterey, CA, June 2002.

[23] S. C. Tweedie. EXT3, Journaling File System. http://olstrans.sourceforge.net/release/OLS2000-ext3/OLS2000-ext3.html, July 2000.

[24] J. Yang, P. Twohey, D. Engler, and M. Musuvathi. Using Model Checking to Find Serious File System Errors. In *OSDI '04*, Dec. 2004.

[25] Y. Yu, P. Manolios, and L. Lamport. Model Checking TLA+ Specifications. *Lecture Notes in Computer Science*, (1703):54–66, 1999.