Robust Consistency Checking for Modern Filesystems

by

Kuei (Jack) Sun

A thesis submitted in conformity with the requirements
for the degree of Master of Applied Science
Graduate Department of Electrical and Computer Engineering
University of Toronto

Abstract

Robust Consistency Checking for Modern Filesystems

Kuei (Jack) Sun

Master of Applied Science

Graduate Department of Electrical and Computer Engineering

University of Toronto

2013

A runtime file system checker protects file-system metadata integrity. It checks the consistency of file system update operations before they are committed to disk, thus preventing corrupted updates from reaching the disk. Previously, we had designed a runtime checker for the widely-deployed Linux Ext3 file system that follows a traditional Unix file system design. In this thesis, we describe our experiences with building Brunch, a runtime checker for an emerging Linux file system called Btrfs. Btrfs is a copy-on-write file system that supports many modern file-system features that pose challenges in designing a robust checker. We find that the runtime consistency checks need to be expressed clearly so that they can be reasoned about and implemented reliably, and thus we propose writing the checks declaratively. This approach makes it easier to reason about the correctness of the checker in several ways. Expressing each check as a logical assertion reduces the complexity of the checks and ensures their independence, thereby improving their clarity. It also helps clearly identify the abstractions for representing file system metadata updates, which is challenging because Btrfs uses many data structures with complex relationships between them. The declarative approach showed that the structure of the file system should be checked before performing any consistency checks, which ensures that the latter do not fail unpredictably. Our results show that runtime consistency checking is still viable for complex, modern file systems.

Acknowledgements

I would like to first thank the committee members, Professor Al Leon Gracia, Professor Cristiana Amza, and Professor Ding Yuan for their timely support and valuable comments. I want to acknowledge the hard works of my colleagues, Daniel Fryer, Mike Qin, and Dhaval Giani during sleepless nights before paper deadlines. Their deep insight helped materialize the many interesting concepts in this thesis. I would also like to thank my wife, Judy, for her dedication during my hardest hours. She really took care of me when I was not able to take care of myself. I appreciate the support and encouragement my family has shown me over the last few years. I am indebted to my supervisors, Professor Ashvin Goel and Professor Angela Demke Brown. Their commitment to ensuring the success and well-being of their students is unparalleled. I would like a take a moment to especially thank Professor Ashvin Goel for his patience and guidance. Not only has he been an invaluable mentor, but he has also proven to be my greatest role model.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Many studies have shown that file systems contain bugs that are hard to detect, even under heavy testing [26, 36]. These bugs can result in data corruption, persistent application crashes, or even security exploits [35]. Unfortunately, existing file-system reliability methods, such as transactional updates, checksums, and redundancy, primarily focus on storage hardware failures and provide limited defenses against file-system bugs [26, 25] or random memory corruption [37]. For example, transactional update techniques such as journaling [16], copy-on-write [18], and soft updates [13] are designed to address crash failures. Checksums and redundancy methods reduce the probability of data loss due to storage hardware or low-level software failures [14]. All these methods assume that the file system or the operating system software is bug free and random corruption of in-memory kernel data structures does not occur [37]. For example, a mirroring system offers no protection against a buggy file-system write, which would be reliably replicated to multiple disks.

When a file system bug corrupts file-system metadata, the entire file system must be checked for possible corruption. This consistency check process is typically performed offline, causing significant downtime for large storage systems [17]. Upon detecting corruption, the checker may attempt repair, but this operation is itself complex and error-prone [15, 1]. Alternatively,

an administrator can use a backup to revert a file system to a consistent state, but a backup risks losing important recent data.

To avoid downtime and data loss, file system metadata corruption must be detected before it propagates to disk. A *runtime* file-system consistency checker can protect the integrity of file-system metadata by checking the consistency of file-system update operations before they are committed to disk. As a result, buggy file-system metadata updates can be detected before they corrupt the disk, minimizing the need for an offline checker.

We have previously described the design and implementation of a runtime checker for the Linux Ext3 file system, a traditional Unix-like file system [12]. Runtime checking involves two operations, *metadata interpretation* and *invariant checking*. The checker observes file system metadata blocks at the block layer, as they are read or written, and interprets the types of file system data structures in these blocks, similar to semantically smart disks [31]. Then, it enforces file-system consistency at runtime by checking a set of rules called *consistency invariants*. These invariants are expressed in terms of the inferred file system data structures. For example, a consistency invariant in the Linux Ext3 journaling file system is that a transaction that makes a data block live (i.e., by adding a pointer to the block) must also contain a corresponding bit-flip (from 0 to 1) in the block allocation bitmap (and vice versa) within the same transaction. If this invariant is not met, then the file system would be inconsistent after the transaction, because the block could be doubly allocated in the future (or it would be lost).

In this thesis, we describe our experiences with building a runtime checker, called Brunch[1], for a next generation Linux file system called Btrfs. Btrfs is still under active development, and so a runtime checker that limits the damage caused by bugs in the file system software can both serve as a powerful debugging tool and help encourage adoption of newer versions of the file system. Btrfs is a copy-on-write, B-tree based file system that supports many modern file-system features such as dynamic allocation of file system structures for handling large files and large numbers of files, extent-based allocation, writable snapshots and logical volumes.

[1] **B**trfs **Run**time **Ch**ecker

To implement these features, Btrfs uses a large number of file system metadata structures with complex relationships among them, which complicate the consistency invariants considerably, raising challenges in designing a *robust* checker. It is vital that a runtime checker, designed to enhance file system reliability, is itself reliable and does not destabilize the system. Specifically, it should work correctly and predictably in the presence of arbitrary file system corruption failures, and it should detect all consistency violations.

We found that it was hard to reason about these correctness and completeness properties in our original Brunch implementation, which required writing the invariants using hand-crafted C code. Consistency invariants need to be *expressed* clearly so that they can be reasoned about and implemented reliably. The language used to write invariants should enhance our confidence that the invariants are correct and complete. Unfortunately, using a low-level language to express high-level invariants in our checker is fundamentally error-prone, because it is hard to enforce a clean separation between the metadata interpretation code and the invariant checking code, when both are written in the same low-level language. For example, Gunawi et al. show that the Linux `e2fsck` file system checker, which intermingles metadata interpretation and checks (both written in C), has bugs that cause additional file-system damage when repairing an inconsistent file system [15]. Their solution is SQCK, an offline consistency checker that translates the checks and repairs performed by e2fsck into SQL.

In this work, we propose using a declarative language to express the consistency invariants in a runtime checker. This approach makes it easier to reason about the correctness of the runtime checker in three ways. First, declarative languages are naturally designed for making runtime assertions. For example, each consistency invariant can be written as a set of declarative statements and run independently of the other invariants. The invariant code is thus easier to reason about than invariants written in a low-level language that are intermingled with each other and with the metadata interpretation code. Second, it helps identify the appropriate abstractions for representing file system metadata updates. For example, our original metadata interpretation code would represent updates at the granularity of data structure fields, but this

representation was either too fine-grained or too coarse-grained for certain invariants, complicating the implementation of these invariants. With declarative invariants, the conceptual invariants are written as clearly as possible, and the metadata is interpreted accordingly. Finally, the declarative specification showed that the structure of the file system should be checked before performing any consistency checks. This ordering ensures that invariant checks do not fail unpredictably, because they can depend on the structural integrity of the file system.

This thesis describes our experiences with building a robust runtime checker for a modern file system. We implemented the C and the declarative version of Brunch concurrently. We found that the declarative invariants were often more than an order of magnitude smaller than the C invariants, enhancing our confidence in the correctness of the invariants. More importantly, as we gained more familiarity with Btrfs, we needed to occasionally modify the invariants. Modifying the declarative invariants was generally dramatically simpler than the C invariants. The declarative implementation forced us to reconsider the abstraction between metadata interpretation and invariant checking, helping make the checker more robust to file system corruption.

Our performance results show that runtime consistency checking using the C implementation is still viable for a modern file system, despite its added complexity. However, our current declarative implementation is slow because we are unable to create and use efficient indexes for invariant checking. In the future, we plan to implement better indexing and a compiler for the declarative implementation. In the meantime, the Brunch implementors of the C invariants have started looking at the declarative invariants and are rewriting some of the C invariants to match the declarative specification.

1.1 Thesis Contributions

This thesis makes the following contributions:

1. We describe the design and implementation of a runtime file system checker called Brunch that uses declarative consistency invariants. These invariants run independent of the rest of the file system checker and other invariants, making it easier to reason about their correctness.

2. We describe the appropriate abstractions for representing file system updates that simplify the writing of file system consistency invariants.

3. We show that the structural integrity of the file system should be checked before checking any consistency invariants. This ordering allows writing invariants more concisely, and invariant violations provide more meaningful information.

4. We evaluate the robustness of the declarative file system checker in two ways: 1) as a qualitative comparison, we compare how invariants are written declaratively versus in a low-level language like C, and 2) we inject file system corruption and compare the detection accuracy of the declarative checker with the offline file system checker.

The rest of the thesis describes our approach in more detail. Chapter 2 provides background on the Btrfs file system. Chapter 3 describes the benefits of specifying consistency invariants in a declarative language and then presents our experiences with designing a robust consistency checker for the Btrfs file system. Chapter 4 describes the two implementations of Brunch and then Chapter 5 evaluates both the implementations, examining their benefits and drawbacks.

Chapter 2

An Overview of Btrfs

This chapter provides background information on the Btrfs file system, which is needed to understand the types of consistency invariants in Btrfs and how they can be checked. Like other modern file systems, such as ZFS [4], Btrfs supports many features that were not available in older file systems [27], such as extent-based allocation, back references, writable snapshots, checksums and logical volume management. Btrfs uses extent-based allocation to efficiently support both large and small files. Extents allow a single allocation record to cover multiple blocks in contrast to the block-based allocation schemes of FFS-like file systems. Btrfs extent records contain back references, which allow efficiently looking up all pointers to the given extent. This information is useful for tasks like online defragmentation, deduplication and volume resizing.

Btrfs supports light-weight, writable snapshots, allowing a user to instantly create a copy of the file system state. The snapshot is isolated from the original version using copy-on-write semantics. After the snapshot is created, both the snapshot and the original can be modified independently. The copy-on-write mechanism in Btrfs is also integral to its crash consistency model.

Btrfs uses checksums to ensure the integrity of metadata and optionally data extents. It supports logical volume management functionality, such as grouping multiple disks together

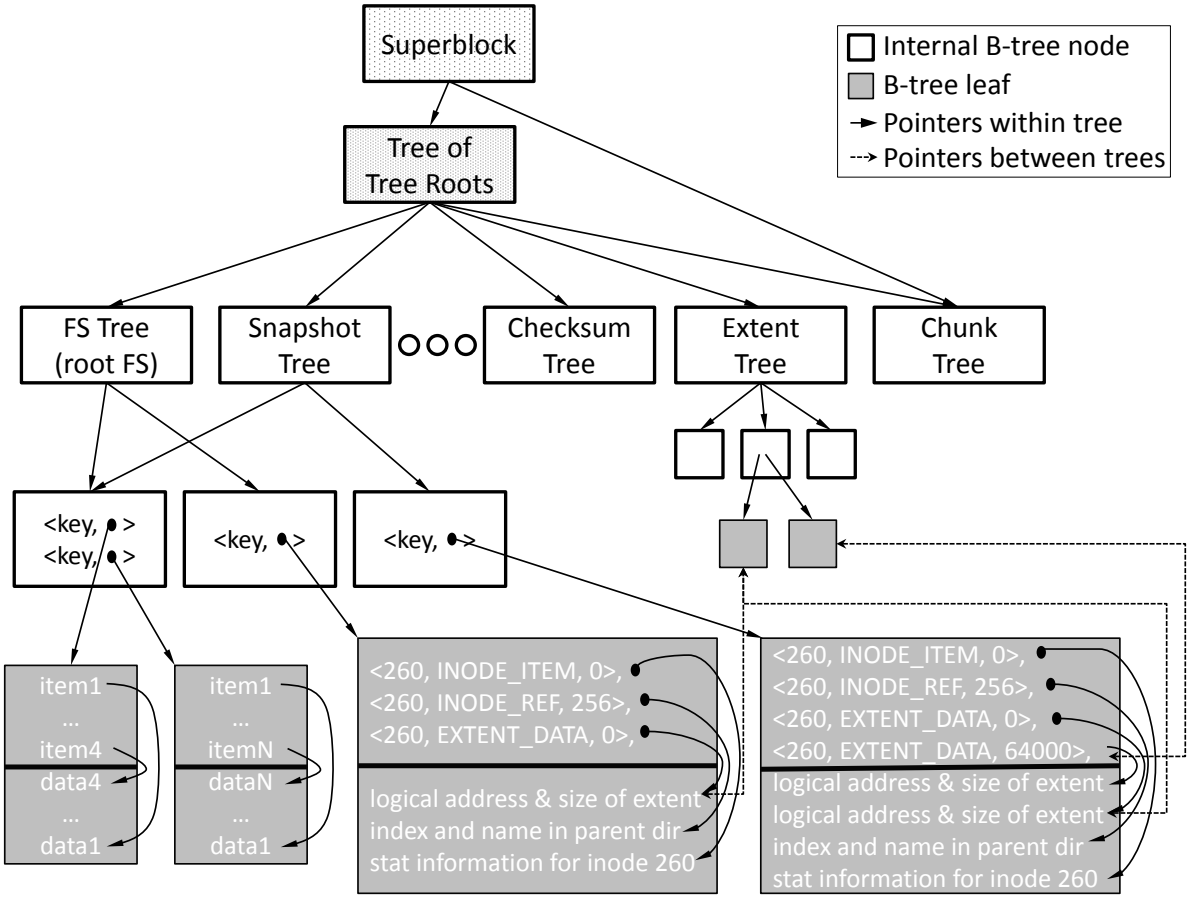


Figure 2.1: Btrfs trees and metadata items.

into a logical volume, mirroring of data between disks or between regions of a disk, and various RAID levels.

2.1 Btrfs File System Format

Figure 2.1 provides a simplified view of the Btrfs file system format.[1] As hinted by the name, Btrfs uses B-trees to store its metadata. A separate B-tree is associated with the root file system and with each snapshot (or "subvolume"). In addition, there are several B-trees with special semantics, such as the checksum tree. The superblock contains pointers to two B-trees, the

[1] Our description of the Btrfs file system format and data structures is based on the Btrfs code included in the Linux 2.6.35 kernel and btrfs-tools version 0.19, which we use in our implementation.

chunk tree and the tree of tree roots. The chunk tree maps logical block numbers to physical block numbers for supporting logical volume management. The tree of tree roots contains pointers to all the other B-trees, including the (root) FS tree, the snapshot trees, the checksum tree, and the extent tree, which records allocation information for logical extents.

All the Btrfs B-trees use the same structure, consisting of internal nodes and leaves. Internal nodes contain an array of key/block-pointer pairs. Like any B-tree, the key represents the smallest key stored in the pointed-to node or leaf, and the block pointer helps locate the child node or leaf on disk. Btrfs leaves contain file system metadata using an array of items and a data section. Items consist of a Btrfs key and the location and size of their data region within the leaf's data section. Data regions are variable sized and store a file system object such as an inode. For example, the Btrfs leaf at the bottom-left of Figure 2.1 contains four items. Every node and leaf in a Btrfs metadata tree has a header as well, with a common header format, including a checksum and volume id that is useful for reconstructing a severely damaged file system. The header also identifies which tree owns the block, and whether or not the block is a node or a leaf.

Items in Btrfs leaves are identified by a Btrfs key, consisting of a tuple [`objectid`, `type`, `offset`]. The meaning of objectid and offset depends on the type of the item that the key represents. For example, if the type indicates that it is an extent item belonging in the extent tree, the objectid denotes the start address of the allocated extent, and the offset gives the length. We can write this key as [start, `EXTENT_ITEM`, len]. The extent structure attached to that key (in the data region in the leaf) contains additional information about the allocated extent, such as whether it contains data or metadata, and an array of back references to all extents which contain pointers to that extent.

Figure 2.1 shows an FS tree and a single snapshot tree. The FS tree has a leaf that contains an inode indicated by a key of the form [260, `INODE_ITEM`, 0], where 260 is the inode number. The corresponding inode structure is stored at the bottom of the corresponding data section and has fields for the inode mode, size, and other fields required by the stat system call.

Unlike Unix-based file systems, the pointers to the file data are not located inside this inode structure. Instead, there can be multiple objects in the tree with the same objectid (the inode number), a type identifying the object as a pointer to file data (EXTENT_DATA), and an offset indicating the starting position in the file corresponding to this data extent. Since B-trees keep items sorted by key, related items with the same objectid are co-located in the tree. The attached structure for an EXTENT_DATA item stores the location of the extent. Figure 2.1 shows that inode 260 in the FS tree has a single data extent [260, EXTENT_DATA, 0] associated with it, and this file is contained in a directory with inode number 256 (as indicated by the offset in the INODE_REF item).

2.2 Btrfs Transaction Model and Snapshots

The Btrfs metadata trees are kept crash-consistent using shadow paging or a copy-on-write mechanism. When a leaf is modified, it is written to a new location on disk, and this necessitates an update to the parent, which must also be written to a new block. This continues recursively up to an item in a leaf of the tree of tree roots, and then up to the superblock, which has a fixed location. Updating the superblock with a pointer to the new root of the tree of tree roots represents a commit point. Allocation of metadata extents results in updates to the extent tree as well. Allocating extents for updates to the extent tree results in further updates to the extent tree, but since an extent record is much smaller than a leaf, this process converges quickly.

The Btrfs copy-on-write (COW) semantics, and the ability to store multiple tree root pointers in the tree of tree roots, allows supporting snapshots. If a new root item is created and points to an existing FS or snapshot root, then with the COW model, modifications to either of the (overlapping) "trees" will not affect the other. Figure 2.1 shows a snapshot created from the FS tree. The file with inode number 260 has data appended in the snapshot, causing an allocation of a new extent for the file (which updates the extent tree), and a new copy of the

leaf with the metadata items for this file. The new copy is linked to the snapshot tree while the parent FS tree continues to be linked to the original leaf.

2.3 Btrfs Data Structures

Btrfs includes many other data structures with complex relationships between them. The directory metadata includes Btrfs items to map the file name to an object id (e.g. the inode number), as well as two indexes, one using a hash of the filename for fast lookup, and the other using a sequence number for iterating over all directory entries efficiently.

Btrfs uses back references extensively. For example, the metadata for a file includes an `INODE_REF` item, the key of which stores the inode number of the parent directory (e.g., inode 260 stores the parent directory inode 256 in [260, `INODE_REF`, 256] in Figure 2.1). The data for this item replicates the filename and the sequence number in the parent directory, as described above. Thus, each file has a back reference to its containing directory. Similarly, `EXTENT_DATA` items record the extents used by a given file, and each extent records a list of back references to these `EXTENT_DATA` items.

Chapter 3

Robust Consistency Checking

This chapter describes the design of a runtime file system consistency checker called Brunch for the Btrfs file system. Our design has two goals that help improve the robustness of the checker: 1) it should work correctly and predictably in the presence of arbitrary file system corruption failures, and 2) it should detect all consistency violations. We meet these goals with three design principles. First, the file system consistency invariants must be written declaratively and concisely, making it easier to reason about their correctness. Second, the abstractions between the checking of invariants and the rest of the checker should be chosen carefully to minimize the complexity of the invariants. Finally, the structure of the file system should be checked before performing any semantic checks so that the latter can depend on the structural integrity of the file system. In the following sections, we first provide background on checking file system consistency at runtime, and then describe our design in the following sections in detail.

3.1 Runtime Consistency Checking

A runtime checker operates below the file system layer and checks a set of consistency invariants before permitting block writes to reach the disk, as shown in Figure 3.1. The next subsections describe how these invariants are specified and checked.

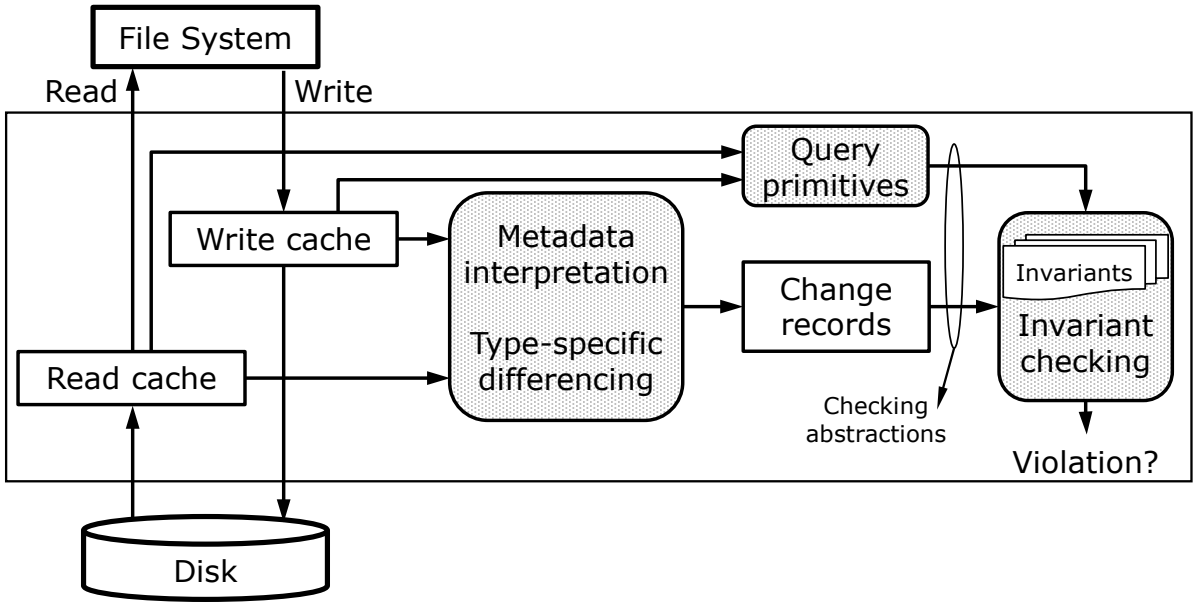


Figure 3.1: A runtime file system checker.

3.1.1 Specifying Invariants

Identifying file system consistency invariants is challenging because a formal specification of file system behaviour is rarely available. Fortunately, the source code of an offline file system consistency checker provides a comprehensive set of consistency properties. For example, Gunawi et al. found that e2fsck checks 121 basic properties for ext2 and ext3 file systems [15]. Ideally, a runtime checker should be able to check all consistency properties. Unfortunately, these properties can be global statements about disk data. For example, a consistency property in traditional Unix file systems is that data blocks must not be doubly allocated. Checking this property requires a full disk scan, making it infeasible to perform at runtime.

Instead, each global consistency property must be transformed into a local consistency invariant, which is an assertion that must hold for updated blocks to preserve metadata consistency. In the example above, the consistency invariant is that if a block points to a newly allocated block, then the new block must be marked allocated in the updated block allocation bitmap. A runtime checker can enforce this invariant by examining only the updated blocks, i.e., the updated pointer block and the updated block allocation bitmap block. Each global con-

sistency property can be transformed to a local consistency invariant because the file system itself relies on limited information (i.e., not the entire metadata on disk) to preserve metadata consistency at runtime. After the invariants have been specified for a file system, the checker performs two operations at runtime, metadata interpretation and invariant checking, as described below.

3.1.2 Metadata Interpretation

Invariants can only be checked when the file system itself claims that its data is consistent on disk. For example, journaling file systems group writes to disk blocks from one or more operations, such as the creation of a directory and a file write, into transactions. Similarly, the copy-on-write Btrfs file system updates its superblock to commit groups of file system operations. Transaction commits are well-defined points at which the file system believes itself to be consistent. Invariants are checked at these commit points, and a consistency violation at these points indicates a bug or a memory corruption.

The consistency invariants used by a runtime checker are expressed in terms of logical file-system data structures, such as the values of block pointers and bits in the block allocation bitmap in the Ext3 example above. However, a buggy file system cannot be trusted to provide the correct logical data structure information. As a result, a runtime checker cannot have any direct dependencies on the file system code or data. Instead, the checker performs metadata interpretation by observing I/O operations below the file system layer, such as at the block layer, and then it independently interprets the types of file system metadata blocks as they are read or written, similar to semantically smart disks [31]. With the known block types, the checker can interpret the block contents and derive the logical file-system data structures. We also infer certain file system operations of interest, such as a transaction commit point, based on interpreting metadata writes (e.g., writes to Btrfs super block, or writes to a block that has the format of a commit block in the Ext3 journal).

Metadata interpretation allows the checker to compare the file-system data structures in the updated metadata blocks, cached in the *write cache*, with the previous version of the corresponding data structures, cached in the *read cache* (these metadata caches are shown in Figure 3.1). This type-specific differencing operation generates records of logical changes, such as a bit flip in an Ext3 allocation bitmap, for all the updated file-system data structures in a transaction.

3.1.3 Invariant Checking

The invariant checking operation verifies that the logical file system changes when applied to consistent, pre-transaction file system state (as observed from the read cache) will result in consistent, post-transaction file system state (as observed from the write cache). Invariant checks are expressed in terms of changes to file system objects such as directories, inodes and extents, but they may also involve querying the state of objects that have not changed. These checks use two abstractions, 1) *change records* that are generated by metadata interpretation and the type-specific differencing operation, and 2) *query primitives* that allow accessing the write cache or unmodified file system data, as shown in Figure 3.1.

Change records capture any modified states of file system objects, such as the addition of a new object, update to an existing object, and the removal of an object in a transaction. Query primitives are used to access objects or object fields that may or may not have changed in a transaction, and thus may not appear in the set of change records. Together, they are sufficient for representing and accessing the state of any file system object that has been added, updated, removed, or remains unchanged.

A change record in Btrfs is expressed as follows:

```
change(TREE, ID, FIELD, OLD, NEW)
```

The `TREE` is the Btrfs B-tree within which the object resides. The `ID` is the unique identifier of the object that is being changed (e.g. a Btrfs key for an inode, indicating that the inode has

changed). The TREE and ID uniquely identify Btrfs objects, similar to the way that a record in SQL is uniquely defined by its table and its primary key. The `FIELD` is a specific part of the object (e.g. inode size). The values `OLD` and `NEW` are the old and new values of the corresponding field. When an object is created or destroyed, the old and the new arguments are set to `null`, respectively. Some Btrfs objects do not contain any data. For example, all information about a Btrfs orphan item is stored within its identifier: `(constant_value, orphan_item, INODE_NR)`. In this case, the field in the change record is set to `null`. We describe query primitives in more detail in Section 3.3.5.

3.2 Expressing Invariants

Gunawi et al. demonstrated that rewriting consistency check and repair operations declaratively in SQL revealed bugs in fsck, an offline file system checker [15]. When the checks and repairs performed by the checker are expressed in a language that more closely matches the domain, it is easier to reason about their effects. As a result, in our previous work, we wrote the Ext3 invariants in C [12] by looking at the Ext3 consistency properties as expressed in SQL [15]. However, when we started working on Btrfs invariants in C, we faced several challenges. First, Btrfs is a vastly more complex file system compared to Ext3 and thus has more complex consistency properties. Second, these properties are hard to extract from btrfsck, the file system checker source code for Btrfs written in C, because they are implemented piecemeal, intermingled with the metadata interpretation performed by the checker. Third, after we converted the consistency properties to their corresponding invariants, and implemented the invariants in C, we found that it was hard to reason about the correctness of these invariants because their implementation was complex, with many corner cases.

We realized that we needed to restart from scratch and write the invariants declaratively so that they can be reasoned about and implemented reliably. The declarative invariants revealed that the complexity of the C implementation resulted in part from a mismatch in the checking

abstractions provided by the Btrfs metadata interpretation code and the abstractions needed by the invariants. As a result, we have restructured the checker to provide the appropriate abstractions, as described in Section 3.3.

3.2.1 Datalog

We chose to express consistency invariants in the Datalog declarative language because it is well suited for checking runtime assertions. Datalog is a logic programming language that is often used as a query language in deductive databases [6]. Datalog programs consist of statements that are expressed in terms of relations, represented as a database of facts and rules. Rules take the form of *conclusion* $\vdash$ *premise*, where *premise* consists of one or more predicates joined by conjunction (comma) or disjunction (semicolon). Predicates can contain variables, written in uppercase, or constants, written in lowercase or in numeric form. Rules can be defined recursively (e.g., the conclusion and the premise can contain the same predicate), which fits well with operating over graph-like file system data structures. Datalog is derived from (and typically implemented in) Prolog. By restricting certain uses of negation and recursion, it guarantees termination and can leverage memoization for fast query execution.

A Datalog computation is initiated by running a query over these relations. A query is a predicate that is satisfied and returns *true* when all of its arguments can be *bound*, and *false* otherwise. For example, suppose our database contains one fact: `father(joe, bob)`. The query `father(joe, bob)` would return *true*; the query `father(X, Y)` would return *true*, binding `X` to `joe` and `Y` to `bob`; the query `father(joe, john)` would return *false*; and the query `father(greg, X)` would also return false, since our database does not have a record of greg's father.

We express all the change records that are generated from a file system transaction as Datalog facts. Consistency invariants are statements that must hold true for a consistent file system. We express these invariants in Datalog in a negated form to reach the conclusion that an invariant has been violated. For example, for a consistency invariant $A \Rightarrow B$, the corresponding

```
% the btrfs key for an extent is [start, extent_item, size]
violation(6, TREE_ID, k(EXTENT, extent_item, SIZE)) :-
  add(TREE_ID, k(EXTENT, extent_item, SIZE)),
  previous(TREE_ID, k(EXTENT, extent_item, SIZE),
    k(EXTENT_PREV, extent_item, SIZE_PREV)),
  EXTENT < EXTENT_PREV + SIZE_PREV.

% the underscore '_' is an "don't care" or wildcard variable
violation(6, TREE_ID, k(EXTENT, extent_item, SIZE)) :-
  add(TREE_ID, k(EXTENT, extent_item, SIZE)),
  next(TREE_ID, k(EXTENT, extent_item, SIZE),
    k(EXTENT_NEXT, extent_item, _)),
  EXTENT_NEXT < EXTENT + SIZE.
```

Figure 3.2: The Btrfs invariant "If a new extent item is added, the extent must not overlap the previous or next extents" expressed in Datalog.

Datalog statement is $violation \vdash A, \neg B$, where A is a condition which will trigger the check B. The predicate A matches a change record because it looks for a change in the file system. The matching is performed based on the attributes of the change record. The predicate B can match change records or invoke primitives to access objects that may or may not have changed.

3.2.2 Invariant Example

Figure 3.2 shows a simple example of a declarative invariant. A consistency property in Btrfs is that extents must not overlap. The corresponding consistency invariant is that if a new extent item is added to a tree, then the extent must not overlap with the previous or next extents in this tree. The `add`(TREE, ID) clause looks for an `extent_item` object with the Btrfs key ID that has been added[1] to the file system and binds the `TREE_ID`, `EXTENT` and `SIZE` variables to its values. The `previous`() and `next`() clauses are primitives that query the metadata caches and bind the previous and next extents in the tree to their second argument, respectively. We need to query the caches in this case because the adjacent extents may not have changed, and thus may not be available as change records. The final clause checks for overlap between the

[1] If the extent size is updated, the extent item key would change, and so the change records would indicate that an extent has been deleted and another has been added.

```
change(TREE, k(INODE_NR, inode_item, 0), flags, _, NEW), NEW \= null
```

Figure 3.3: Sample query for extracting the new value of an inode's `flags` field from change records.

new extent and the previous or next extents returned by the primitives. When an extent does not have a previous or next extent, the `previous()` and `next()` query will fail, which indicates that the invariant has not been violated.

Note that this invariant operates completely independently of the metadata interpretation code and any other consistency invariants, thus making it is easier to about its correctness. Later in Section 5.1, we compare invariants written in Datalog versus C.

3.3 Choice of Abstractions

In this section, we show that the consistency invariants of a file system can be written more concisely and clearly when the checking abstractions are chosen carefully based on the requirements of these invariants. These abstractions consist of change records and query primitives, as described previously in Section 3.1.3.

3.3.1 Wrappers for Change Records

Change records can capture all file system state modifications, but it is often easier to write invariants using higher-level abstractions. For example, invariants on objects that are added are generally different from invariants on objects that are deleted. Suppose that an invariant needs to obtain the `flags` field of a newly created or updated Btrfs inode. The Datalog query would be expressed as shown in Figure 3.3.

For the query in Figure 3.3, we need to ensure that `NEW` is not null or else the change record can match deleted inodes as well, which may cause false violations. To avoid such errors, we provide some simple wrapper rules for change records, such as `add/update/delete`, that return newly added objects, updated objects and deleted objects. Similarly, the `new/old` rules

```
% bind SIZE to the change in the size of the file
violation(15, TREE_ID, k(OBJECTID, inode_item, 0)) :-
  default_zero(TREE_ID, k(OBJECTID, inode_item, 0), size, OLD, NEW),
  SIZE is NEW - OLD, % ...
```

Figure 3.4: Example use of `default_zero`

return objects that are added/deleted *or* have been updated. The query above would use the `new` rule to obtain the new value of the `flags` field for an inode that has been added or updated.

Change records use the Datalog `null` value for the `OLD` and the `NEW` fields to indicate a newly created or destroyed field. However, sometimes a different default value is more appropriate. For example, a default value of 0 is more meaningful for integer and bit fields. We use the `default_zero` rule to simplify invariants on integer and bit fields as shown in Figure 3.4.

Many declarative invariants in Btrfs need to test whether an object, rather than an individual field, has been added, updated or deleted. We could use a wildcard on the field argument of the change record for this test. For example, we can detect that an inode is created or updated by replacing the `flags` field with a wildcard in the example shown in Figure 3.3. However, this causes redundant computation in Datalog, which searches for all possible ways to prove the truth of a query. As a result, Datalog would redo the same check for all change records for the same object. Instead, we generate a single `add/update/delete` per-object record to improve the performance of the object-level queries.

3.3.2 Compound Values in Change Records

Datalog does not natively support complex terms as arguments of predicates, e.g., `father(joe, bob)` is permissible but `father(father(joe, X), bob)` is not allowed, because complex terms can cause infinite recursion. This restriction was not an issue for our Ext3 Datalog invariants because the identity, type and fields of Ext3 data structures are simple values. However, Btrfs data structures are more complicated because they use compound values. For exam-

```
struct btrfs_disk_key {
  __le64 objectid;
  u8 type;
  __le64 offset;
} __attribute__ ((__packed__));

struct btrfs_dir_item {
  struct btrfs_disk_key location;
  __le64 transid;
  // ...
} __attribute__ ((__packed__));
```

Figure 3.5: Data structures for a Btrfs key and a Btrfs directory item.

ple, the Btrfs key, which serves as an ID, for items in the Btrfs nodes or leaves is the tuple [`objectid`, `type`, `offset`]. Initially, we flattened these values in the Datalog facts, as shown below:

```
change(5, 256, dir_index, 1, location, objectid, null, 258)
```

In this change record, the tree id is 5, and the key of a Btrfs directory item is [256, `dir_index`, 1]. The data associated with this item is a `struct btrfs_dir_item` shown in Figure 3.5. The `location` field is a Btrfs key for the inode associated with this directory entry. The change record above shows that a directory item was created and the inode number associated with this directory item (i.e., the `objectid` in the Btrfs key) is 258. Unfortunately, these variable length facts make the invariants much more complicated because of the ambiguity in the arguments of the change record. For example, it is difficult to tell whether `location` in the example above is part of an ID or a field.

The example above can be more easily expressed using compound terms and fixed-length change records as follows:

```
change(5, k(256, dir_index, 1), f(location, objectid), null, 258)
```

In this change record, we express a Btrfs key as the predicate `k(OBJECTID, TYPE, OFFSET)`, which represents the identity of the object. Data structure fields for aggregate types, such as

```
1 violation(16, TREE, k(INODE_NR, dir_item, CRC)) :-
2     new(TREE, k(INODE_NR, dir_item, CRC), type, DIR_ITEM_TYPE),
3     query(TREE, k(INODE_NR, dir_item, CRC), location, LOCATION),
4     not(query(TREE, LOCATION, f(mode, s_ifmt), INODE_FILE_TYPE),
5         DIR_ITEM_TYPE =:= INODE_FILE_TYPE).
```

Figure 3.6: Btrfs invariant "Directory entry type is the same as the type of the inode".

```
violation(16, TREE, INODE_NR, dir_item, CRC) :-
    new(TREE, k(INODE_NR, dir_item, CRC), type, DIR_ITEM_TYPE),
    query(TREE, INODE_NR, dir_item, CRC, location, objectid,
          OBJECTID),
    query(TREE, INODE_NR, dir_item, CRC, location, type, TYPE),
    query(TREE, INODE_NR, dir_item, CRC, location, offset, OFFSET),
    not(query(TREE, OBJECTID, TYPE, OFFSET, mode, s_ifmt,
              INODE_FILE_TYPE),
        DIR_ITEM_TYPE =:= INODE_FILE_TYPE).
```

Figure 3.7: Btrfs invariant "Directory entry type is the same as the type of the inode" without compound value support.

a `btrfs_dir_item` that contains a `btrfs_disk_key` are expressed as the predicate `f(FIELD, SUBFIELD, ...)`. Fortunately, there are several extensions to Datalog that support compound terms [20]. In our implementation, we guarantee that queries terminate because we do not create a change record fact that includes another change record as an argument.

Besides removing the ambiguity of arguments in change records, compound terms can help simplify invariants, as shown in Figure 3.6. This invariant checks that a directory entry's file type is the same as the type of the inode pointed to by the entry. For example, both the types are directories or both are files. The predicate on Line 2 returns the file type in a directory item when a directory item is changed (created or updated). The `query(TREE, ID, FIELD, VALUE)` predicate is a primitive that queries the metadata caches for the object `ID` and returns its `FIELD` value. Notice that `LOCATION` is a Btrfs key, and this compound value is passed directly as the second argument of the `query` statement on Line 4, which specifies the identifier of the inode object. Without compound terms, the query on Line 3 would need to be written thrice, once for each field of the Btrfs key, as shown in Figure 3.7.

3.3.3 Granularity of Change Records

Invariant checking becomes much simpler when change records are generated at the granularity expected by invariants. Otherwise, invariants require additional logic, either to stitch facts together, or to break them down further, reducing the clarity of the invariant. Previously, the type-differencing operation, shown in Figure 3.1, generated change records for the old and new values of each *changed* field of each updated object. If a field of an object is an aggregate (i.e., a structure or an array), it would be broken down until each subfield is of a primitive C data type (we assume a 64 bit integer to be the largest primitive data type). While this differencing implementation is simple, it complicates invariants that use aggregate fields or bit fields.

For example, suppose there is an invariant on the `location` field (a Btrfs key) of a `dir_item` object, shown in Figure 3.5. This invariant could check that if this location changes, then the corresponding inode and a reference to this directory item from the inode exist in the file system. The original differencing implementation provided the changed fields of objects. For example, suppose the `location` field of a `dir_item` object changed from `k(258, inode_item, 0)` to `k(258, root_item, 12)`. Figure 3.8(a) shows the change records that would have been generated when the directory object ID is 257. These change records do not provide the entire location key value because the objectid in the location field (258) did not change. As a result, obtaining this value requires querying the metadata cache using the `query()` predicate. Figure 3.8(b) shows the complicated predicates needed to aggregate the `location` field when any of its subfields change.

Instead, if a change record was generated for the entire location field, as shown in Figure 3.9(a), then the more intuitive `new` rule can be used to obtain the new location value, as shown in Figure 3.9(b).

Figure 3.10(a) shows the reverse problem for invariants requiring bit fields. In this case, the `flags` field is modified and thus available in a change record, but the bits in the field need to be extracted, for example, to check whether the `inode_nodatasum` bit has been changed

```
change(5, k(257, dir_item, -1), f(location, offset), 0, 12).
change(5, k(257, dir_item, -1), f(location, type), inode_item,
    root_item)
```

(a) Change records are too fine grained.

```
% Logical OR: ';', Logical AND: ','
% we need to query for unchanged subfields of location
violation(...) :-
  ((new(TREE, ID, f(location, objectid), OBJECTID),
    query(TREE, ID, f(location, type), TYPE),
    query(TREE, ID, f(location, offset), OFFSET));
   ( new(TREE, ID, f(location, type), TYPE),
    query(TREE, ID, f(location, objectid), OBJECTID),
    query(TREE, ID, f(location, offset), OFFSET));
   ( new(TREE, ID, f(location, offset), OFFSET),
    query(TREE, ID, f(location, objectid), OBJECTID),
    query(TREE, ID, f(location, offset), OFFSET))),
  LOCATION = k(OBJECTID, TYPE, OFFSET), % ...
```

(b) The invariant uses complex queries to aggregate the compound value of the Btrfs key structure.

Figure 3.8: Invariants are more complicated when change records are too fine-grained.

```
change(5, k(257, dir_item, -1), location, k(258, inode_item, 0),
     k(258, root_item, 12)).
```

(a) Change record at correct abstraction level for the Btrfs key structure.

```
% this is what the user wishes to do
violation(...) :-
  new(TREE, ID, location, LOCATION), % ...
```

(b) An invariant that uses a Btrfs key.

Figure 3.9: Invariants can be written more intuitively when change records are generated at the correct abstraction level.

```
% Bitwise AND: '/\', Not equal: '\='
violation(...) :-
  default_zero(TREE, ID, flags, OLD_FLAGS, NEW_FLAGS),
  OLD_NODATASUM = OLD_FLAGS /\ nodatasum,
  NEW_NODATASUM = NEW_FLAGS /\ nodatasum,
  OLD_NODATASUM \= NEW_NODATASUM, % ...
```

(a) Change records are too coarse grained, so the invariant needs to extract the bit fields.

```
violation(...) :-
  new(TREE, ID, f(flags, nodatasum), BIT), % ...
```

(b) Change record at correct abstraction level for bit fields.

Figure 3.10: Invariants are more complicated when change records are too coarse-grained.

within the `flags` field. Figure 3.10(b), shows the more intuitive and concise invariant if bit field modifications are available as separate change records.

These examples show that invariants are much easier to express when the change records are generated at the granularity expected by invariants. This insight led us to modify our approach for expressing and checking consistency invariants. Previously, we had designed the metadata interpretation and differencing operation independently of the invariants, but this led to invariants being much harder to write correctly. Instead, it is easier to write invariants logically, based on the consistency properties of the file system, and then the granularity of the change records should be matched to the needs of the invariants. This approach significantly enhances our confidence that the invariants are correct and complete.

We implement this approach in the type-specific differencing operation by allowing programmers to explicitly specify the granularity for representing the identifier and fields of an object when the default data structure definition (e.g., the C struct definition) is not matched with the invariant specification. This extra programming effort is outweighed by the gains in clarity in the invariant specification. The Btrfs invariants make use of three levels of granularity other than the default field within a C structure: flag bits within a byte, structure fields (including integers and strings), and entire structures (either by themselves or as a part of a containing structure). To avoid generating duplicate changes at multiple granularities (e.g., bit fields of a

byte and the byte itself), we currently generate change records for only the granularity that is explicitly specified.

3.3.4 Choice of Identifier

The change records require identifiers for each object. For certain objects, such as the Btrfs key and the Ext3 inode number, the file system has its own identifier. Other file system objects, such as Btrfs backrefs and directory entries, do not have an explicit identifier. However, the file system stores them as distinct objects in a set. In this case, we construct an identifier for these objects. For example, a directory entry object in Ext3 can be uniquely identified by the parent inode number and the name in the entry. Similarly, the Btrfs `inode_ref` item, described in Section 2.3, stores multiple backrefs (as an array of objects) when there are multiple hard links from within the same directory to an inode. These backrefs can be uniquely identified by the `inode_ref` item key and either the filename or the file index. In this case, we chose to use the index because it was easier to process an integer in Datalog than a string. Section 3.4 describes our approach for checking the uniqueness of object identifiers in the face of file system bugs.

3.3.5 Query Primitives

The second type of abstraction used by invariants are query primitives. These primitives are used to access objects or object fields that may or may not have changed in a transaction, and thus may not appear as change records. Primitives operate on the write and read caches and hence return the most recent version of the object. In other words, they return either the new version if it has been changed in a transaction, or else the old version from the read cache. If the old version does not exist in the read cache, then we query the file system data structure on disk.

Format	Description
`query(TREE, ID, FIELD, VALUE)`	1. Given `TREE, ID` and `FIELD`, binds `VALUE` to the value of the field. 2. Given `TREE, ID, FIELD`, and `VALUE`, returns `true` if the value of the field is equal to `VALUE`.
`query(TREE, ID)`	3. Tests for the existence of an object in TREE with identifier `ID`.
`query(TREE, k(OID, TYPE, OFFSET))`	4. Given `TREE`, `OID` and `TYPE`, binds `OFFSET`. 5. Given `TREE and OID`, binds `TYPE` and `OFFSET`.
`previous(TREE, ID, ID_PREV)`	Binds `ID_PREV` to the identifier of the previous object in the same `TREE`.
`next(TREE, ID, ID_NEXT)`	Binds `ID_NEXT` to the identifier of the next object in the same `TREE`.

Table 3.1: Brunch primitives.

The Brunch primitives are shown in Table 3.1. These primitives allow retrieving an object by key, testing whether an object exists, and finding the previous or next Btrfs key in a tree. They return `true` on success (e.g., object is found), `false` otherwise.

The `query` primitive can be invoked in several ways. The first version that returns the value of a field of an object is used most often. The second version tests for the value of a field of an object. The third version tests for the existence of an object. The fourth version helps retrieve all objects with the same object identifier `OID` and a type. The last version is similar to the previous version, except that only an `OID` is specified. For example, the first clause of Rule 12 in Figure 3.11 shows that when an inode is deleted, no Btrfs item with the inode's number remains within the metadata tree. We need to use a primitive for this invariant because these items may not have changed.

The query primitives operate at the granularity of an object (versions 1, 2 and 3) and the `OID` (version 4). The `previous` and `next` primitives allow retrieving objects with adjacent object identifier `OID` (Figure 3.2 shows an example). This physical adjacency information is not available in change records. For example, assume that there are two adjacent extents, with identifiers 10 and 12. Even if both extents are modified, we do not know about the status of extent 11 from change records, because this extent may not have been modified or it may

```
violation(12, TREE_ID, k(INODE_NUMBER, TYPE, OFFSET)) :-
  delete(TREE_ID, k(INODE_NUMBER, inode_item, _)),
  file_tree(TREE_ID),
  query(TREE_ID, k(INODE_NUMBER, TYPE, OFFSET)).

violation(12, TREE_ID, k(INODE_NUMBER, TYPE, OFFSET)) :-
  add(TREE_ID, k(INODE_NUMBER, TYPE, OFFSET)),
  file_tree(TREE_ID), TYPE \= inode_item,
  not(query(TREE_ID, k(INODE_NUMBER, inode_item, 0))).
```

Figure 3.11: Rule 12: "If an inode is removed, make sure no objects with that inode number remain in the tree. If an item is added, and it's not an inode, verify that the corresponding inode exists."

not exist. The `previous` and `next` primitives provide this information. While the primitives shown in Table 3.1 are used to access Btrfs metadata, we believe that primitives with similar interfaces would be needed for checking invariants in any file system [32].

3.4 Checking Structure before Semantics

The previous sections have focused on expressing invariants clearly and concisely so that consistency violations can be detected reliably. Our second goal is to ensure that the checker works predictably in the presence of arbitrary file system failures. To do so, we need to ensure that the three components of the checker, shown in Figure 3.1, 1) metadata interpretation, 2) query primitives, and 3) invariant checking, are robust to metadata corruption. Invariant checking is expressed declaratively, operates on well-formed change records generated by metadata interpretation, and uses query primitives. Hence, its robustness depends entirely on the first two components. Both these components access the read and the write caches. The read cache contains previously checked file system state that is known to be consistent. The write cache may contain corrupt data caused by file system bugs and thus any code accessing this cache must perform careful validation. Below, we describe how this validation can be performed.

Intuitively, the structure of the file system needs to be checked when interpreting metadata. These checks need to ensure that the file system data structures are typed correctly, so that

they can be interpreted correctly. For example, these checks will prevent following a stray or corrupt pointer. In addition to correct typing, the primitives, which take an identifier as input, need to operate on the data structure associated with this identifier. These structural integrity requirements consist of three integrity properties that need to be checked in order:

Type Safety: Type safety ensures that interpreting metadata in the write cache is robust to data corruption. Consider a query primitive `query(TREE, ID, VALUE)` that binds `VALUE` to the object with identifier `ID`, with the `ID` incorporating the type of the object (e.g., the type in the Btrfs key).[2] Then, type safety ensures that `query(TREE, ID, VALUE)` will bind *some* object to `VALUE` that is of the same type as the type specified in `ID`. This ensures that the metadata interpretation code will operate on correctly typed objects.

Unfortunately, type safety is hard to enforce dynamically because file system data structures do not necessarily provide type information, e.g., a tag associated with each type. Even if such type information was available, it could have been corrupted, possibly to another known type. Instead, we ensure type safety by validating or range checking all primitive data types that are accessed during metadata interpretation. For example, absolute disk pointers need to lie within the file system partition, while extent-relative pointers should lie within the extent. Similarly, enumerated values (enum in C) need to be valid instances, and any length fields in structures should lie within expected bounds. Any time these checks fail, we raise a type-safety violation.

Reachability Constraints: While the type safety property ensures robust metadata interpretation, it is not sufficient for ensuring the correctness of the primitives. For example, if an object is misplaced in a B-tree, `query(TREE, ID, VALUE)` would not return an object that exists because it assumes that keys are ordered, or else it would need to perform an expensive, full tree search. In Btrfs, we enforce reachability by ensuring that keys are

[2] This primitive is similar to the third query primitive in Table 3.1, except that it returns the object.

sorted correctly in the updated B-trees. These constraints need to be checked before the rest of the constraints, as described below.

Uniqueness Constraints: The primitives expect that all objects are uniquely identified by an identifier. If multiple objects have the same identity then several problems can arise. First, the primitives may not provide such duplicate objects deterministically, which could lead to invariant violations that are hard to analyze, or worse, allow corruption to propagate to disk. Second, duplicate change records may be generated (e.g., two objects with the same identity are modified), but since Datalog ignores duplicate facts, such duplicates would not be detected. It is important to check reachability before checking uniqueness. If an object is reachable, it is easy to test for uniqueness by first searching for the object.

After the three structural integrity properties have been checked, we are assured that `query(TREE, ID, VALUE)` will bind `VALUE` to the object associated with `ID`. At this point, the semantic consistency invariants can depend on well-formed change records being generated (even though their contents may be corrupt) and the primitives working correctly.

Next, we show two examples that illustrate subtle issues that can arise when uniqueness constraints are not enforced before checking consistency invariants. Consider the "extents must not overlap" invariant, shown in Figure 3.2. This invariant assumes that extents are identified uniquely. Suppose that the previous extent was of zero size (a possible corner case). In that case, if an extent that started at the same location as the previous extent was added mistakenly, then the first invariant in Figure 3.2 would not detect the uniqueness violation because the current extent OID would not be less than the previous extent OID. To detect a uniqueness violation, we would need to modify the last predicate to `(EXTENT =:= EXTENT_PREV ; EXTENT < EXTENT_PREV + SIZE_PREV)`, which is subtle because the first condition seems to be covered by the second condition (assuming SIZE_PREV ≥ 1).

Consider the Btrfs "directory entry type and inode type must match" invariant, shown in Figure 3.6. Suppose a file system, while creating a directory, creates a directory entry, and

mistakenly creates two inodes, one of which is of the wrong type (e.g., a file type). The second `query` primitive in Figure 3.6 that returns the type of the inode would match the inode change records because the inodes have been recently created. However, the `INODE_TYPE` value that is bound will depend on the order in which the Datalog engine performs its matching, so the corruption may or may not be detected.

The main benefit of checking structural integrity before checking consistency invariants is that the invariants can be made simpler because they can depend on the preceding integrity properties. Another benefit is that it allows the consistency invariants to be run in any order, independently of each other. For example, the order in which the invariants are run has no effect on the correctness of the primitives, which has been established by the structural integrity properties. Finally, this approach raises structural violations as early as possible, thus providing more accurate debugging information.

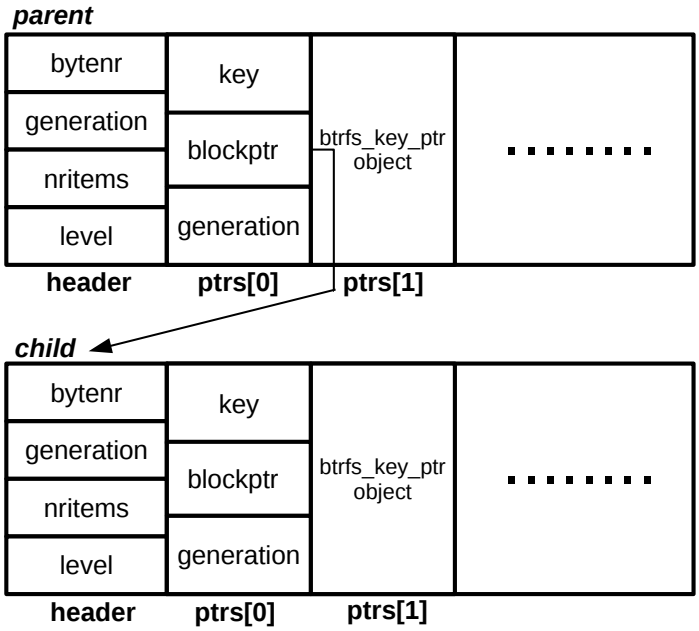
Chapter 4

Implementation

In this chapter, we will describe the technical aspects of designing and implementing our invariant checking framework. We will first give an overview of the design of Brunch and the scope of our work in Section 4.1. In Section 4.2, a brief background on Xen is given, and we explain how Brunch is incorporated into the Xen architecture. Section 4.3 will provide the implementation detail of supporting invariant checking in Datalog. We describe our C implementation in Section 4.4, and discuss the limitations of our implementation in Section 4.5.

4.1 Overview

We have implemented two online checking systems for Btrfs, one with invariant checks in C (BrunchC) and one in Datalog (BrunchD). Both of them are designed to check a block I/O stream generated from a Xen-hosted guest OS. Both versions of Brunch have an identical architecture as depicted in Figure 3.1. BrunchC and BrunchD share most of their code base with the exception that BrunchD has an additional Datalog module, and that each version has its own transaction checking code. BrunchD invokes Datalog queries through the Datalog engine's C interface to perform invariant checking, while BrunchC directly calls a C function. The entire framework, except for the Datalog invariants, is written in C.



Object Type	Field	Description
`btrfs_node`	`header`	A `btrfs_header` structure (described below). Each block shown in the figure above is of this type.
	`ptrs`	An array of `btrfs_key_ptr` objects
`btrfs_header`	`bytenr`	The byte offset of the current block with respect to the start of the partition
	`generation`	The generation number of the current block
	`nritems`	Number of valid `btrfs_key_ptr` objects in the current block
	`level`	The B-tree level of the current node
`btrfs_key_ptr`	`key`	A `btrfs_disk_key` structure (i.e. [`objectid`, `type`, `offset`])
	`blockptr`	The byte offset of the child node
	`generation`	The generation number of the child block

B-tree Invariants

1.	`nritems != 0 && nritems < BTRFS_NODEPTRS_PER_BLOCK`
2.	`ptr[i].key < ptr[i+1].key`
3.	`parent.ptr[i].key == child.ptr[0].key`
4.	`parent.ptr[i].blockptr == child.header.bytenr`
5.	`parent.ptr[i].generation == child.header.generation`

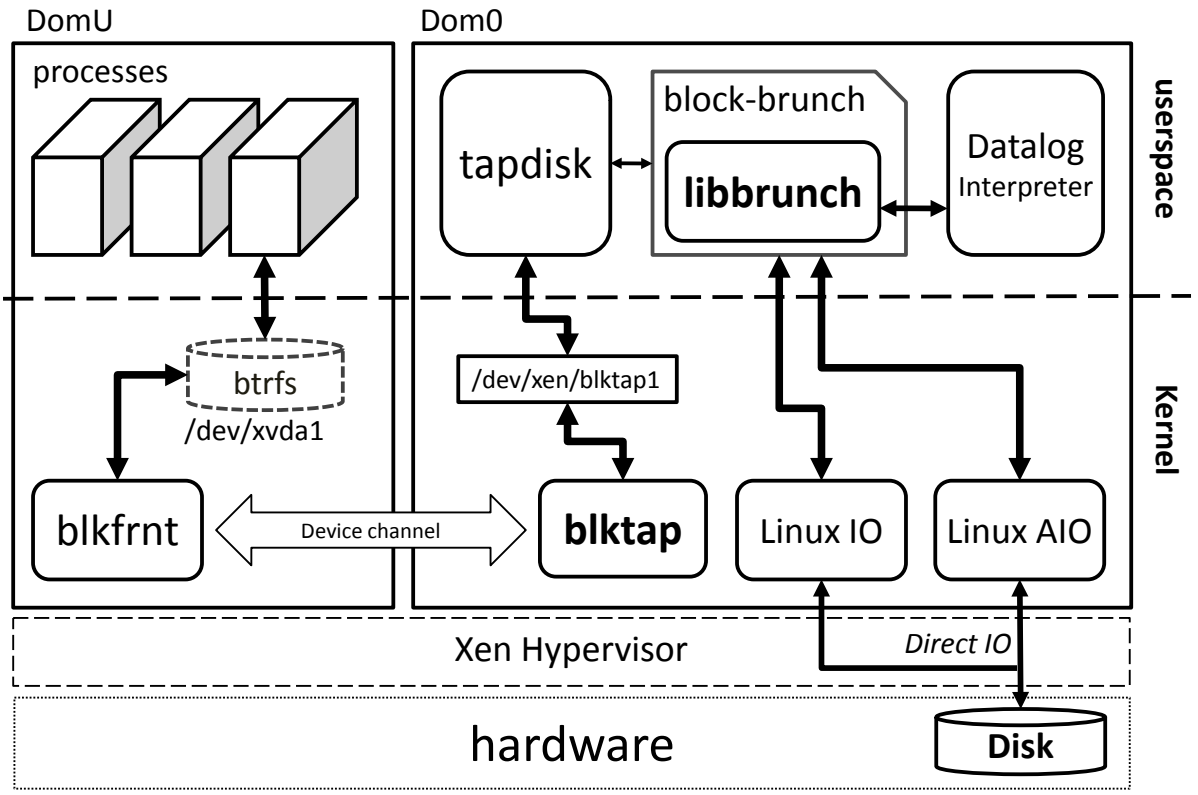
Table 4.1: A description of Btrfs internal B-tree node data structures and a list of integrity invariants checked by Btrfsck for internal B-tree nodes. In the C declaration of `btrfs_header`, some fields were intentionally left out for brevity.

The Brunch invariants are based on the behaviour of btrfsck v0.19, and the implementation of Btrfs in the linux-2.6.35 kernel. We have identified 30 properties to check. Of the 30 invariants, there are 25 semantic invariants. The remaining 5 were identified to be structural integrity constraints necessary to ensure correct traversal of the B-tree. All structural integrity constraints are checked in C before change records are generated. The integrity constraints pertaining to the correctness of the B-tree data structure are shown in Table 4.1. Invariant 1 is a type-safety constraint because not range checking the value of `nritems` may cause the metadata interpreter to dereference outside the bounds of the block. Invariant 2 is both a reachability and a uniqueness constraint. All keys in a valid B-tree node must be monotonically increasing, a requirement that is necessary to provide both reachability and uniqueness. Invariant 3 to 5 are reachability constraints. They verify that the parent points to the correct child node, thus ensuring that B-tree items are ordered correctly.

The original version of our runtime checker was implemented within the same kernel in which the file system was running [12]. This checker used the Linux device mapper to interpose on block I/O. We have ported the checker to the user level, and are using the Xen hypervisor to monitor the block I/O requests of the target file system. This approach has two benefits. First, it provides stronger isolation between the checker's code and its data, and the file system being checked, which helps detect metadata corruption caused not just by file system bugs but *arbitrary* kernel bugs. Second, the user-level checker can use a standard Datalog interpreter, without requiring a port of the interpreter to the kernel.

4.2 Recon via Hypervisor

Figure 4.1 shows Brunch framework implemented using the Xen virtual environment [2]. Xen is a Type 1 hypervisor that runs directly on physical hardware. The hypervisor runs one or more guest virtual machines (VMs). One virtual machine, Dom0, is privileged and has direct access to the hardware, and can create additional unprivileged virtual machines.



Block I/O from DomU is captured by the blktap kernel module in Dom0 and sent to tapdisk, which invokes Brunch's block module API to process the requests. Brunch intercepts and processes all I/O made from DomU before redirecting the I/O to the underlying device through the Linux AIO module. libbrunch implements the runtime file system checker shown in Figure 3.1.The Datalog interpreter is responsible for performing invariant checking when a commit block is detected. Query primitives that request data not available in the Brunch cache use synchronous I/O requests to fetch blocks from the disk.

Figure 4.1: The Brunch implementation using the Xen hypervisor.

In order to intercept file system requests at the block layer, Brunch makes use of Xen's blktap[1] I/O infrastructure [34]. Blktap is a kernel module that channels block I/O from DomU to tapdisk, a user-level process in Dom0 that contains several block modules which implement a standard storage interface. All read and write requests from the guest are directed through tapdisk and fed into a block module, which may perform additional operations on the data before sending the request to an underlying physical or virtual device. Brunch is currently a library which is statically linked to Xen's tapdisk2[2] process.

We copied the existing Linux AIO (Asynchronous I/O) block module and re-factored it so that the data for a write request is first intercepted by Brunch before being sent to the AIO module. For read requests, we intercept the AIO read callback and process the data before returning it to the guest. We picked the AIO module as our starting point primarily due to its high performance characteristics over other block modules implemented by Xen developers. The behaviour of intercepting write requests and blocking the write from reaching disk is important for the correctness of Brunch. If Brunch detects a violation during transaction checking, it must prevent the commit block from reaching the disk.

One of the problems we had to solve with this implementation, however, was that during transaction checking, a query primitive may issue a request for data not available in the Brunch caches. Therefore, we have to fetch the data from the disk. However, since the transaction checking procedure is not run as a separate thread, it will block any further requests from being accepted and processed by the AIO module. We resolve the issue by opening the block device twice, as shown in Figure 4.1. One file descriptor opens the block device with the O_DIRECT | O_RDWR flags and is passed to the AIO module. The other file descriptor opens the same device with the O_RDONLY flag. We use the latter to perform synchronous I/O, which will retrieve the requested block independent of the AIO module. Any synchronization required between the two file descriptors is performed by the Dom0 kernel. We do not read

[1] pronounced "block tap"
[2] tapdisk2 is a newer version of tapdisk which is faster and more reliable.

stale data because Brunch also has its own write cache, which stores all recently written blocks. Therefore, even if there are block write requests still pending in the AIO module, we will not have had to request these blocks because they would still be available in our internal cache.

4.3 Supporting Datalog

Datalog is a subset of Prolog that restricts certain operations so that it can guarantee termination and the absence of side-effects. It uses an caching technique called memoization for improving performance. With memoization, the results of previously performed queries are cached so that any subsequent query with the same arguments can directly return the cached result.

Brunch has two requirements that are not available in all Datalog implementations. First, it must be able to handle nested tuples, which we use to encode structures being passed through change records. Second, it needs an interface to C so that we can use the query primitives. Despite there being several extensions to Datalog which support constructs such as tuples and sets [20], none of them provide an interface to C. Instead, we use a Prolog interpreter for running our Datalog queries. Prolog allows expressing nested tuples using functors, which meets our first requirement. Secondly, many Prolog implementations offer a well-documented C interface. The main drawback of using Prolog is that we must implement memoization for our queries manually. Prolog does not memoize queries by default, because some queries, such as writing to standard output, have side-effects. Although we use a Prolog interpreter, our declarative invariants conform to the standard Datalog syntax, with the exception of nested tuples.

We chose SWI-Prolog [33] for our implementation because it has a well-documented C interface. We link the SWI-Prolog library statically to tapdisk2 via the swipl-ld utility program. We pre-compile the Prolog files, which contain the consistency rules, and link Brunch and the SWI-Prolog library into a single executable, as shown in Figure 4.1.

```
1  property(17, TREE_ID, PARENT_ID, CHILD_ID) :-
2    query(TREE_ID, k(PARENT_ID, dir_item, HASH), location,
3      k(CHILD_ID, inode_item, 0)),
4    query(TREE_ID, k(PARENT_ID, dir_item, HASH), name_len, LEN_1),
5    query(TREE_ID, k(PARENT_ID, dir_item, HASH), name, NAME_1),
6    query(TREE_ID, k(PARENT_ID, dir_index, INDEX), location,
7      k(CHILD_ID, inode_item, 0)),
8    query(TREE_ID, k(PARENT_ID, dir_index, INDEX), name_len, LEN_2),
9    query(TREE_ID, k(PARENT_ID, dir_index, INDEX), name, NAME_2),
10   query(TREE_ID, k(CHILD_ID, inode_ref, a(PARENT_ID, INDEX)),
11     name_len, LEN_3),
12   query(TREE_ID, k(CHILD_ID, inode_ref, a(PARENT_ID, INDEX)),
13     name, NAME_3),
14   LEN_1 = LEN_2, LEN_2 = LEN_3,
15   NAME_1 = NAME_2, NAME_2 = NAME_3, crc32c(NAME_1, HASH).
16
17 violation(17, TREE_ID, k(OBJECTID, TYPE, OFFSET)) :-
18   ( TYPE = dir_index ; TYPE = dir_item ),
19   new(TREE_ID, k(OBJECTID, TYPE, OFFSET), location,
20     k(CHILD_ID, inode_item, 0)),
21   verify(property(17, TREE_ID, OBJECTID, CHILD_ID)).
22
23 violation(17, TREE_ID, k(OBJECTID, inode_ref, a(PARENT_ID, INDEX))) :-
24   new(TREE_ID, k(OBJECTID, inode_ref, a(PARENT_ID, INDEX))),
25   verify(property(17, TREE_ID, PARENT_ID, OBJECTID)).
```

Figure 4.2: Invariant "For every `dir_index`, `dir_item`, and `inode_ref` triplet, their `name` and `index` must match, in addition, the value of `dir_item`'s offset field in its key must equal to the crc32c hash value of `name`", written in Datalog. The two `violation` predicates separately invoke the property 17 predicate through the `verify` query, which performs memoization if the property holds (i.e., returns *true*). The property 17 predicates starts off by fetching the location field of a `dir_item`, which contains the information necessary to find its back reference (i.e., the `inode_ref` object). After which, it obtains the `name` and `name_len` field of all three objects, and ensures that all length fields are equal, and all names are equal, plus that the crc32c value of the `name` field is equal to the offset field in the `dir_item`'s key.

```
1 change(5, k(256, dir_index, 2), location, null,
2   k(286, inode_item, 0)).
3 change(5, k(256, dir_item, 1732636812), location, null,
4   k(286, inode_item, 0)).
5 change(5, k(286, inode_ref, a(256, 2)), name, null, 'brunch').
```

Figure 4.3: Sample change records that are used during a Rule 17 check.

We improve the performance of invariant checking in Prolog by manually adding memoization to cache the results of invariant checks that pass. We motivate the need for memoization by presenting an invariant that would perform poorly without it. The property 17 predicate, as shown on line 1 in Figure 4.2, can be invoked by matching `dir_index`, `dir_item`, or `inode_ref` objects. Exactly one check is sufficient to ensure that the invariant holds, since property 17 fetches all of the related change records (i.e., if the rule is triggered by a `dir_index`, it would fetch the associated `dir_item` and `inode_ref` during a query to the property 17 predicate to perform the check). However, due to the backtracking behaviour of Datalog, it will exhaustively search all possible outcomes. For example, assume we have the three change records shown in Figure 4.3. The `dir_index` change record will first be matched by the violation clause on line 17 of Figure 4.2, which invokes the property 17 predicate. It first queries the `dir_item`'s `location` field, which is the change record on line 3 of Figure 4.3, and later queries the `inode_ref`'s `name` field, which is the change record on line 5. When the check succeeds, violation 17 fails (i.e., there is no violation). A backtrack ensues, causing a redo on the violation 17 predicate with the next matching change record in the database (i.e. `dir_item`). Eventually, the property 17 predicate would be executed again with the exact same arguments. Without memoization, the same invariant, triggered by different change records, will be re-run three times.

We implement memoization with a Prolog library function called `verify` that caches its argument query when the query succeeds. Therefore, on all subsequent calls to violation 17, the property 17 predicate would immediately return true since memoization would have cached the result of the first successful check. Currently, 5 out of the 25 structural invariants take advantage of memoization. Other invariants do not benefit from memoization because they are not triggered by multiple change records. By adding memoization, we have reduced the total transaction check time by half on average.

Module	LOC
Prolog Adapter	1712
C/Prolog Translator	2964
C/Prolog Translator Generator	902
Primitive Module	1034
Invariant Checker	214

Table 4.2: BrunchD's Prolog modules. The line of code count for each module includes empty lines, but does not include the LOC of their corresponding header files. The Prolog Adapter is responsible for making API calls to SWI-Prolog's API. The C/Prolog Translator is responsible for converting change records encoded in C data structures to Prolog facts, and vice versa. The C/Prolog Translator Generator is a Python script which automatically generates C code. The generated code is the C/Prolog Translator. The Primitive module contains implementation of all primitives shown in Table 3.1. It uses the C/Prolog Translator to convert Prolog terms back to C representation. The invariant checker is a simple module that invokes invariant queries but does not include the invariants.

4.3.1 Transaction Checking

In this section, we describe the transaction checking process in BrunchD. This process consists of four steps: 1) change record generation, 2) set differencing, 3) invariant checking, and 4) clean up. Next, we describe these steps. We will then describe the implementation of the query primitives and some of the challenges in using Datalog for invariant checking. Table 4.2 presents the total line of code count for each of the Prolog module that was implemented for BrunchD. We will describe the purpose of each module in this section.

Prolog queries can be invoked in C by calling a set of SWI-Prolog's C interface API functions. When Btrfs commits, Brunch's type-specific differencing module generates change records which are fed into the C/Prolog Translator module, written in C. Its purpose is to convert change records from C data structures to Prolog facts. We build the Prolog database by using the built-in *assert* predicate, which adds a new Prolog fact to the database.

Once all change records have been generated, we perform set differencing. The purpose of set differencing is to merge change records that were generated for objects that are part of an unordered linked list on disk (e.g., Ext3 directory entries and Btrfs `inode_ref`). For example, we may observe the change records shown in Figure 4.4(a). The two change records are expressed as one after set differencing is performed, as shown in Figure 4.4(b). We needed

to perform set differencing in Prolog because our change record generation module currently does not support it. As is shown in our evaluation, the cost of set differencing is high. We are planning to move it to the change record generation phase where it can be performed much more efficiently. Next, we execute each invariant in Prolog. Finally, at the end of transaction checking, we wipe the database via the *retractall* built-in predicate, which removes all the dynamically added facts. A wipe is necessary because the change records for the previous transaction are no longer valid for the next transaction.

SWI-Prolog allows the Prolog interpreter to call arbitrary C functions. We take advantage of this functionality to implement our query primitives. We add wrapper code around the same query functions that the C invariants use to lookup arbitrary file system objects, given a key. We also added wrapper code, written in Prolog, for the query primitives. These wrappers memoize query results, and they use relevant change record values, if they exist, rather than querying the Brunch caches. Table 4.2 shows the LOC required to implement all primitives listed in Table 3.1. The bulk of the source code needed to implement primitives marshals data between C and Prolog.

One of the challenges in converting change records from C data structures to Prolog facts is the sheer number of transformations that needs to be performed. Every integer needs to be converted to Prolog integer, and every enumerated type, such as the name of the data type (i.e. `inode_ref`) and the name of every field (i.e. `name`), needs to be converted to textual format. To facilitate this, we created a Python script that generates this conversion code in C. We show the effort required to implement the Python script in Table 4.2.

```
change(5, k(257, inode_ref, 321, 0), name, null, bob).
change(5, k(257, inode_ref, 321, 0), name, joe, null).
```

(a) Change records generated before set differencing.

```
change(5, k(257, inode_ref, 321, 0), name, joe, bob).
```

(a) Change record generated after set differencing.

Figure 4.4: Set differencing for unordered sets.

In an earlier version of BrunchD, we generated change records for every single field of every single object for completeness. Now, we prune the set of change records by disabling the generation of any change record that we know the invariants are not going to check (e.g., `transid` field of every Btrfs metadata object). This optimization reduced our invariant checking time by a factor of four.

4.4 Rule Checking in C

Figure 4.5 shows a typical invariant written in C. As one may observe, the logic is very scattered and verbose, when compared its Datalog counterpart, which is shown in Figure 4.2.

Unlike our Datalog implementation, which builds a database of all change records, the C implementation processes the change records in a streaming manner, without storing each record. Therefore, all invariants written in C must keep track of data that they potentially care about. For every change record type, rule-specific code is responsible for extracting the relevant values and storing them in a hashtable. Figure 4.5(a) shows the code segment for initializing the hashtable. Figure 4.5(b) shows the code for allocating memory for the rule-specific data structure via `tx_alloc`, filling the data structure with relevant data, and placing the structure into a hash table via `rv_hashtable_insert`. This process builds a set of rule-specific indexes on the change records.

Once all the change records have been processed in this manner, each rule is executed in turn. Typically each rule will iterate over its index structure, executing queries or asserting the conditions of the invariant on the indexed values. Figure 4.5(c) shows how each rule-specific index structure is checked for potential invariant violation. This segment of the invariant first checks whether an old `DIR_INDEX` object (i.e., it existed in the pre-update state of the file system) has a corresponding removed `INODE_REF` object that refers to the same inode. This is done by first creating a key object to search the `RULE_17_INODE_REF_BYIDX` hashtable. If a corresponding `INODE_REF` object is not found, `INVARIANT(names, 1701)` will trigger a

```
case label_hash_rule_17_ref_pair:
  rule_storage[i] = rv_create_hashtable(16, hash_rule_17_ref_pair,
    cmp_rule_17_ref_pair);
    break;
```

(a) Line 291 in **initialize_tables()**

```
case BTRFS_DIR_INDEX_KEY:
  switch (field) {
    case DIR_LOCATION: {
      struct btrfs_disk_key* new_location = uint2ptr(newval);
      struct btrfs_disk_key* old_location = uint2ptr(oldval);
      if (IS_FS_TREE(tree_id)) {  //RULE 17
        if (new_location->type == BTRFS_INODE_ITEM_KEY ||
          old_location->type == BTRFS_INODE_ITEM_KEY) {
          struct rule_17_ref_pair * hash_key = tx_alloc(rv,sizeof(*hash_key));
          struct inum_pair * locs = tx_alloc(rv,sizeof(*locs));
          hash_key->tree_id = tree_id;
          hash_key->dir_inode = item_key->objectid;
          hash_key->ref = item_key->offset;
          if(new_location->type == BTRFS_INODE_ITEM_KEY)
            locs->new_inum = new_location->objectid;
          else
            locs->new_inum=0;
          if(old_location->type == BTRFS_INODE_ITEM_KEY)
            locs->old_inum = old_location->objectid;
          else
            locs->old_inum=0;
            rv_hashtable_insert(rule_storage[RULE_17_DIR_INDEX_INUM],
              hash_key, locs);
        }
```

(b) Line 1319 in **process_change()**. Similar code exists on Line 690, 1226, 1472 and 1530.

```
foreach_hash(rule_storage[RULE_17_DIR_INDEX_INUM]) {
  struct inum_pair * inums = (struct inum_pair*) value;
  struct rule_17_ref_pair * idx_info = (struct rule_17_ref_pair*) key;
  if(inums->old_inum) {
    /* Verify we saw something disappear */
    struct rule_17_inode_ref search_key= {
      .tree_id=idx_info->tree_id,
      .dir_inode=idx_info->dir_inode,
      .child_inode=inums->old_inum,
      .ref=idx_info->ref
    };
    struct name_pair * names= (struct name_pair*)
      rv_hashtable_search(rule_storage[RULE_17_INODE_REF_BYIDX], &search_key);
    INVARIANT(names, 1701);
    if(names) {
      INVARIANT(names->newname==0,1702);
    }
  }
  // ... other parts of the checks are not shown here ...
```

(c) Line 2089 to Line 2260, in **check_transaction()**

Figure 4.5: "For every `dir_index`, `dir_item`, and `inode_ref` triplet, their `name` and `index` must match, in addition to the hash value of `dir_item` being equal to the crc32c hash value of the `name`", implemented in C.

violation, since the two objects are supposed to have been removed together. Otherwise, we make sure that the `INODE_REF` object is in fact removed by ensuring that it did not update its name.

Because the rule-specific code is distributed between the different change record types relevant to the invariant, the C invariant implementations are disjoint. Also, the logic required to piece together different parts of an invariant is difficult to conceptualize in C. There are a total of eight code segments in `check_transaction()` for Rule 17,[3] with each segment dealing with a very specific subset of the rule.

As observed from the previous example, memory for the index structures has to be manually managed, whereas Datalog manages all memory allocation for the change records that are generated. The debugging effort required to ensure that the C invariant implementation is bug-free is high, because when we observe an invariant violation during testing, we do not know for sure whether it is our understanding of the invariant or the C implementation that is incorrect.

4.5 Limitations

While there are conceivably more consistency properties that Brunch could have checked, they are either not checked by Btrfsck, missed during code inspection, or the consistency properties pertain to features that we do not support. Brunch does not support the Logical Volume Management (LVM) features of Btrfs, either within a single volume or across multiple volumes. We currently disable the LVM features of Btrfs during `mkfs` by passing in the `-d single -m single` options. We plan to support Btrfs LVM features in the future.

[3]The code segment shown in Figure 4.5(c) is 1 of the 8 segments.

Chapter 5

Evaluation

In this chapter, we evaluate our declarative approach for expressing file system consistency invariants. First, we describe our experiences with using Datalog and discuss the advantage of writing invariants in Datalog over C. Next, we show the correctness of our invariants by running a comprehensive set of corruption tests and comparing the corruption detection accuracy of our runtime checker with the Btrfsck offline file system checker. Lastly, we analyze the performance of our implementation and discuss the implications of our results.

5.1 Experiences with Datalog

In this section, we compare our declarative Datalog-based checker (BrunchD) with the C-based checker (BrunchC) and describe the benefits and drawbacks of each. In general, the Datalog implementation trades performance for clarity when compared to the C implementation.

Clear Invariants A significant advantage of Datalog invariants over BrunchC is clarity, for two reasons. The first is simply brevity. The Btrfs Datalog invariants, including helper functions, and written in 584 lines of code (this number doesn't include comments or empty lines). The invariant checking code in BrunchC uses 1851 lines of code. Some invariants are not completely implemented in BrunchC, and so this number is an underestimate.

The second reason is that Datalog's model matches the way we reason about invariants. Consider Invariant 12, which enforces the property that there exists an inode item for every distinct objectid in a file system tree. A plain statement of the invariant might be "If an inode is removed, make sure that no objects with that inode number remain in the tree. If an item is added, and it's not an inode, verify that a corresponding inode exists." The corresponding BrunchD invariant reflects this statement, as shown in Figure 3.11, improving our confidence in the correctness of the implementation.

The equivalent Rule 12 in C involves about 45 lines. While the Datalog code is consolidated into one place, the C is split across several locations, as mentioned in Section 4.4. First, there are several declarations - a struct to hold the index values along with appropriate hash and comparison functions, and two entries in an enumeration to allocate two hashtables to the rule. This takes about 10 lines, scattered between an enum, a list of structures in a header file, and a set of hash function declarations. Second, there are two different types of change records which have to be matched - the delete of an inode, and the creation of any item in an FS tree. When either of these is found, an entry recording this fact is allocated, initialized, and inserted into the hash table. This takes about 5 lines in two different locations. Finally, there is a loop which iterates over each table. For all deleted inodes, it must query the tree to ensure that nothing with the same objectid still exists; for newly created items which are not inodes, it must query the tree to ensure that the corresponding inode exists. This takes about 30 lines.

Freedom from Low-level Mistakes The high-level nature of Datalog also insulates the invariant writer from simple mistakes that can impact stability. It allows the programmer to focus solely on pattern matching, and does not distract the programmer with the burden of low-level implementation details. For instance, the C invariants have to manage memory, perform pattern matching on change records, and avoid dereferencing null pointers. Additionally, while the representation of structures like tuples and strings in Datalog is easy, the details cannot be ignored at the C level. Some of these responsibilities must be handled in the BrunchD imple-

mentation as well, but they are handled by the interpreter or the C-Datalog translation module. The translation module, which converts data types between C and Prolog, needs to be written once and will work, with minor modifications, for other file systems. In contrast, writing invariants is much harder because file system developers generally do not document them.

Rapid Prototyping Prototyping invariants is also easier in Datalog. One invariant we had difficulty with was Rule 17, shown in Figure 4.2. Rule 17 governs the relationship between the two items for a directory entry, `DIR_ITEM` and `DIR_INDEX`, and the backref item `INODE_REF`. We attempted several times to implement it in C before we had written it in Datalog, but every time our understanding changed, we had to rewrite a large amount of code. Thinking about an invariant in C is difficult because the language is not in the same domain as the problem. Once we got the BrunchD invariant right, we returned to BrunchC and implemented the equivalent version in C. The final invariant in Datalog uses 45 lines, as partly shown in Figure 4.2. The changes to the C version of the invariant added about 250 lines to the existing invariant, including changes to the change record generation for `INODE_REF` objects, as shown in Figure 4.5.

Similar to prototyping, when a bug is discovered in an invariant, it is easy to patch the existing invariant in Datalog. Our original understanding of Rule 25 was that it ensures that all file data extents in a file are contiguous. What we did not realize is that this is only true up to the file's logical size. A Btrfs file is permitted to have discontiguous extents beyond its logical end-of-file. Once we understood the problem, it took about 25 minutes to fix and completely test the Datalog invariant. The fix required adding a single line of Datalog, as shown in Figure 5.1(a). It took roughly 3 hours just to implement a correct fix, with about 20 lines of code in C, as shown in Figure 5.1(b).

```
violation(25, TREE_ID, k(OBJECTID, extent_data, OFFSET)) :-
  new(TREE_ID, k(OBJECTID, extent_data, OFFSET), num_bytes, NUM_BYTES),
  query(TREE_ID, k(OBJECTID, inode_item, 0), size, I_SIZE),
  OFFSET < I_SIZE,
  not(property(25, TREE_ID, OBJECTID, OFFSET, NUM_BYTES)).
```

(a) The line `OFFSET < I_SIZE` was added to Datalog Invariant 25 after we found that file data extents may be discontiguous after the logical end of the file.

```
/* Check to see if we grew. If we did, start with minimum i_size
 * Keep searching till we cover new i_size */
if(fs->start > fs->size) {
    struct btrfs_disk_key k=fe_key;
    u64 cur_pos = fs->size;
    k.offset=fs->size;
    while(1) {
        fe = (void*) btrfs_query_cache(rv, inode_hashkey->tree_id,
            &fe_key, FLAG_GLB | WRITE_CACHE | READ_CACHE, 0, &item);

        INVARIANT(fe && item->key.type==BTRFS_EXTENT_DATA_KEY, 2505);

        if(!fe || item->key.type != BTRFS_EXTENT_DATA_KEY)
            break;
        if(fe->type == BTRFS_FILE_EXTENT_INLINE) {
            fe_size = fe->ram_bytes;
        } else {
            fe_size = fe->num_bytes;
        }

        /* We made it! */
        if(item->key.offset+fe_size >= file_size)
            break;

        INVARIANT(item->key.offset + fe_size > cur_pos, 2506);

        if(item->key.offset + fe_size <= cur_pos)
            break;
        cur_pos=item->offset+fe_size;
        k.offset=cur_pos+1;
    }
}
```

(b) This C code was added so that file extents are not checked beyond the logical file size. The C code shown here is only a small portion of the implementation for Invariant 25.

Figure 5.1: Part of Invariant 25 in Datalog and C.

5.2 Correctness

In this section, we demonstrate the correctness of the invariants written in Datalog and C through a set of comprehensive tests, as described below. While we cannot formally verify our code, we have run extensive tests, comprising of building the Linux kernel, creating snapshots, and running a series of file system operations (such as creating and deleting file and directories) on both the main and the snapshot trees. During this process, we assumed that the file system worked correctly and did not introduce any inconsistency. We made several changes to the invariants as we discovered false alarms. Many bugs, including bugs in the type-specific differencing module and the C-to-Datalog conversion module, were discovered and fixed.

To assess whether our invariants are also free of false negatives, we corrupted specific file system metadata fields such as `INODE_ITEM`'s `mode` field and observe whether Brunch or Btrfsck, the offline file system checker for Btrfs, would detect the corruption. We facilitated the testing by running a modified version of the guest operating system that includes a corruption framework. This framework intercepts file system I/O at the block layer, similar to Recon, and corrupts file metadata objects of the specified type. Brunch intercepts the corruption in Dom0 outside the kernel.

Note that since we cannot run BrunchD and BrunchC simultaneously, we cannot directly compare the ability to detect errors for BrunchD and BrunchC. Therefore, each version of Brunch has an independent set of corruption tests, but both of which are compared against Btrfsck.

In Figure 5.2, we show the results of our corruption test. We corrupted a total of 36 fields across 7 Btrfs metadata objects. We exclude corruption results for 13 fields that neither Btrfsck nor BrunchD caught (e.g., the `transid` field of every Btrfs metadata object). In Table 5.1, we show the Datalog invariants which were triggered by the corruption tests performed for BrunchD. Figure 5.2 BrunchD is able to detect all violations that Btrfsck caught.

We also notice that Btrfsck sometimes crashes due to corrupted fields. We will analyze each scenario in the following paragraphs.

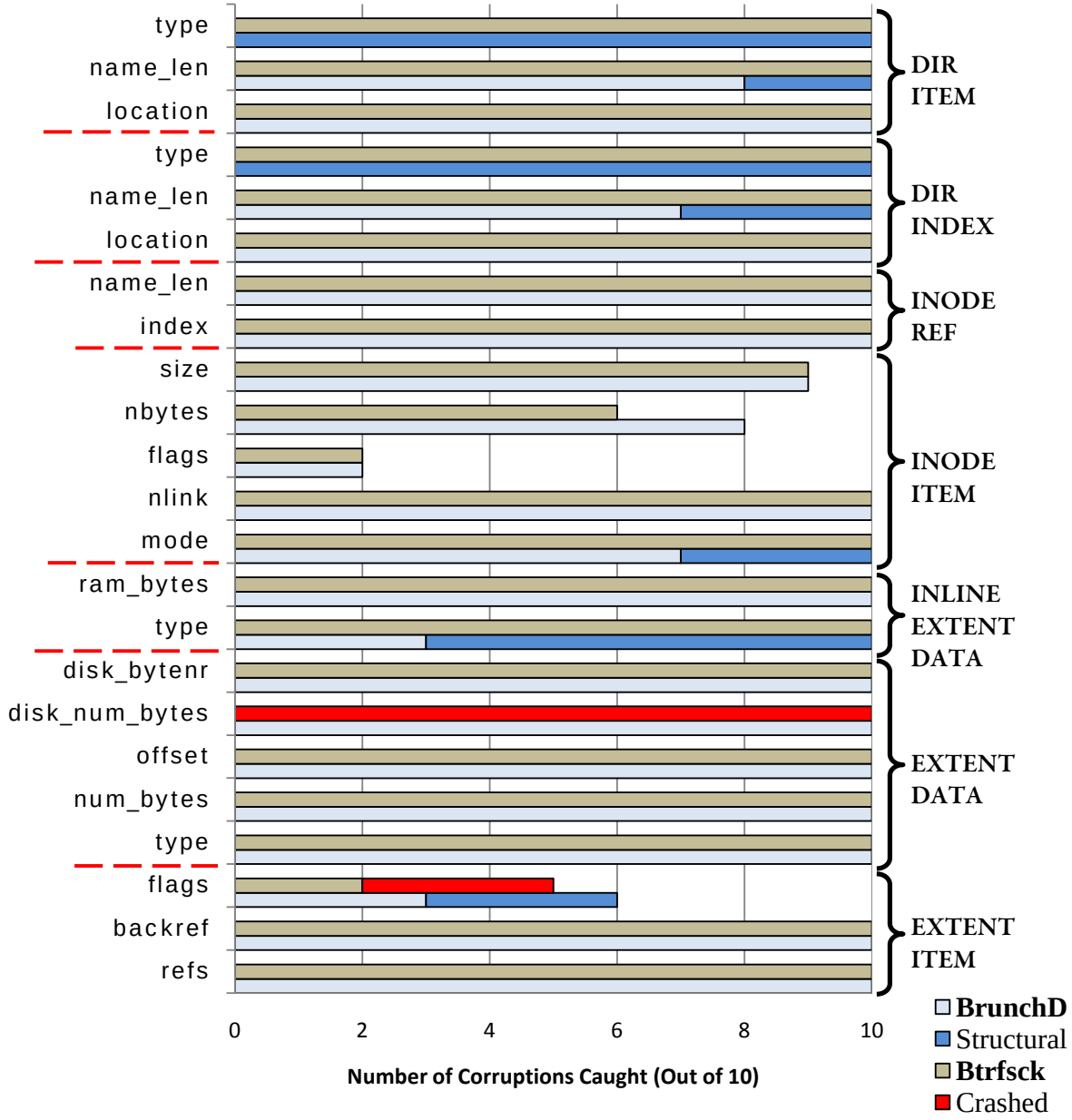


Figure 5.2: Corruption results for BrunchD vs. Btrfsck. The top line in each pair is the Btrfsck result, and the lower line for each pair is BrunchD's result. BrunchD is able to detect more corruptions than Btrfsck, and does not crash when structural invariants are violated.

#	Description	Triggered by corruption to
1	If a `tree_block_ref` object is added to an `extent_item`, an internal tree node, a `root_item`, or the super block should point to the `extent_item`.	`extent_item.backref`
5	If an `extent_data`'s `disk_num_bytes` is updated and is non-zero, then it be must equal to the offset of the `extent_item` that it references.	`extent_data.disk_num_bytes`
7	The `extent_data_ref`'s `count` field must be equal to the number of `extent_data` points to the `extent_item` that owns the `extent_data_ref`.	`extent_data.offset`
9	The `extent_item`'s `refs` field must be equal to the number of all its backrefs.	`extent_item.refs`
13	If a inode is not a directory, it should not have any associated `dir_item` or `dir_index`.	`inode_item.mode`
14	The `nlink` field must be equal to the number of `inode_ref`s that points to the inode.	`inode_item.nlink`
15a	For a directory, its `inode`'s `size` field must be equal to sum of all `name_len` field of its directory entries (`dir_index` and `dir_item`).	`inode.size`
		`dir_item.name_len`
		`dir_index.name_len`
15b	A directory inode's `nbyte` field must be 0.	`inode.nbytes`
16	If an inode's `mode` changes file type, then the directory entries that references it must also update their `type` field.	`inode_item.mode`
17	The name and index of all `dir_item`, `dir_index`, and `inode_ref` that refers to the same inode must be equal. In addition, the offset of the `dir_item`'s key must be equal to the crc32c hash value of the file name.	`dir_index.location`
		`inode_ref.name_len`
		`dir_item.location`
		`inode_ref.index`
21	Checksum should not exist for all of an inode's `extent_data` if its `nodatasum` bit is set.	`inode_item.flags`
23	Checksum should exist for all `extent_data` whose inode's `nodatasum` bit is not set.	`extent_data.disk_bytenr`
		`extent_data.offset`
24	A inode's `nbytes` field is equal to the sum of all its `extent_data`'s `num_bytes` field and `inline_extent_data`'s `ram_bytes` field.	`inline_extent_data.ram_bytes`
		`extent_data.num_bytes`
		`inode_item.nbytes`
25	There should not be holes in a regular file: i.e. the range of all its extent_data must cover the full size of the inode.	`extent_data.num_bytes`
		`inode_item.size`

Table 5.1: List of Datalog invariants that were triggered by type-specific corruption.

```
if (btrfs_extent_flags(eb, ei) & BTRFS_EXTENT_FLAG_TREE_BLOCK)
  ptr += sizeof(struct btrfs_tree_block_info);
```

Figure 5.3: Btrfsck bug.

The BTRFS_EXTENT_FLAG_TREE_BLOCK bit of the flags field of an EXTENT_ITEM determines the offset from which to interpret the list of backrefs that follows the structure. if the bit were flipped, it can cause misinterpretation. The code in Btrfsck which uses the bit field is shown in Figure 5.3. Brunch is able to catch the problem early because it checks the unused bits of the flag field to detect potential corruption. However, it would not be able to find a problem early on if only the BTRFS_EXTENT_FLAG_TREE_BLOCK bit were flipped during corruption. Nonetheless, for either Btrfsck and Brunch, crash or other undefined behaviour can be avoided by proper sanity and type checking while parsing the backrefs. This observation further motivates the need for robust structural integrity checks during metadata interpretation.

Btrfsck also terminates on an assertion failure on the disk_num_bytes field of EXTENT_DATA. We traced the cause of this assertion in the Btrfsck source code and discovered that it was because Btrfsck depends on the field to perform other checks (i.e., the backref consistency check similar to Rule 9 shown in Table 5.1). The disk_num_bytes field, together with disk_bytenr, are used to uniquely identify an EXTENT_ITEM. However, Btrfsck decides to terminate rather than gracefully failing the check when it is unable to locate the reference EXTENT_ITEM. The main problem with this approach is that Btrfsck mixes metadata interpretation with rule checking. Thus when metadata interpretation fails unexpectedly, it cannot continue with its checks. On the other hand, Brunch does not depend on the value of disk_num_bytes to complete metadata interpretation, since EXTENT_DATA is not a structural metadata in Btrfs. Brunch also completes all metadata interpretation and change record generation before executing invariant checks.

BrunchD was able to detect more corruption than Btrfsck for the nbytes field of an INODE_ITEM. This exceptional case occurs because Btrfsck did not ensure that the nbytes field of an inode remains 0 when the inode is a directory. However, an excerpt from the Btrfs

wiki page explicitly states that `nbytes` "... is the sum of the offset fields of all `EXTENT_DATA` items for this inode. *For a directory, this is 0*" [9]. Fortunately, a corrupted `nbytes` field for directories does not seem to affect the functionality of the file system. Even then, we suspect that most Btrfs developers lack a complete understanding of all consistency properties due to the lack of formal specification and documentation of the consistency properties in declarative form.

There are some fields which, depending on the corruption, would either remain a valid value or be converted to an invalid value. An example is `INODE_ITEM`'s `mode` field, which stores the type of the file as an enumerated type. When the `mode` is corrupted, it either changes its file type to the wrong one, or it would change the field to an uninterpretable value. In the latter case, our structural invariant would detect the problem and halt the metadata interpreter. In the former case, there are many invariants that ensure the correctness of each type of file. Rule 13 and 16 listed in Table 5.1 are good examples of such checks. The `name_len` field is an example of another structural invariant which may not be violated if the corrupted value stays within reasonable range. However, Rule 17 would notice a discrepancy between the `dir_item` and others.

A corruption on the `flags` field of an inode is rarely detected because most bits of the `flags` field are unused.[1] Lastly, both Btrfsck and BrunchD missed a corruption of an inode's `size` field because the corrupted inode was an orphan item (i.e., it's link count is 0).

A list of the remaining semantic invariants which did not trigger any violation during our test is shown in Table 5.2. The structural invariants shown in Table 4.1 were not triggered because we do not corrupt internal B-tree structures at this stage of our corruption tests. More explanation on the limitations of the corruption test is given in Section 5.2.1. Here, we define false-negatives as injected faults that are detectable (i.e., a true violation of a consistency property). We believe that no false-negative have occurred because Btrfsck is likely to have caught the corruption if we did not. If that were to occur, we would have found out and corrected the

[1] Only 3 out of 64 bits in the `flags` field are currently used.

#	Description	Why not corrupted?
2	`ref` field of a `root_item` must be equal to the sum of all `dir_item` and `dir_index` that points it it.	*b, c*
3	`ref` field of a root_item must be greater than 0 unless it is an `orphan_item`.	*b*
4	Every `root_item` of snapshot tree must be referenced by a `root_ref`, a `root_backref`, a `dir_item` and a `dir_index`.	*b, c*
6	`extent_item` must be non-overlapping.	*c*
8	The `offset` field of `shared_data_ref` and `shared_block_ref` must be equal to the old `blockptr` value of an updated `key_ptr`.	*a, c*
10	Every `extent_data_ref` should have an associated `extent_data`.	*c*
11	`nlinks` field of an inode must be greater than 0 unless the inode is an `orphan_item`.	*d*
12a	An `inode_item` must exist for every `objectid` that exists in a file system tree.	*c*
12b	When an `inode_item` is deleted, all objects with the same `objectid` should also be deleted.	*c*
18	`dir_item`, `dir_index`, and `inode_ref` that refers to the same inode must be deleted together.	*c*
19	`super_block`' s `csum_type` field must be zero.	*b*
20	A directory inode cannot have any associated `extent_data` or `inline_extent_data`.	*d*
22	If `nodatasum` of an inode is set, its associated `extent_data` should not be checksummed.	*d*

Reason	Explanation
a	We do not corrupt B-tree internal nodes and leaves (i.e., `key_ptr` or `btrfs_header`)
b	We do not corrupt `root_item` or `super_block`.
c	We do not add spurious objects or intentionally delete objects (subset of reason *a*), e.g., by changing the Btrfs key.
d	It could have happened, but the corruption tests didn't cover this case (it may be rare, such as corrupting to a specific value like zero)

Table 5.2: List of semantic invariants that were not triggered by type-specific corruption tests for BrunchD. A list of possible reasons and their explanation is given for each invariant.

invariant. A limitation to this approach is that if Btrfsck also did not catch a false-negative, then we would not know about it. One possible scenario of an uncaught false-negative would be correlated violation, where two or more invariants should trigger on the same corruption, but one does not and that is masked by the fact that the other invariant(s) triggered. In that case, we would not detect the false-negative unless we manually inspect the corruption logs in great detail. However, the chance of a correlated failure that also causes false-negatives is low, and can be eliminated by running more corruption tests using edge-case values that would specifically target the invariant under scrutiny.

The corruption experiment results for C are shown in Figure 5.4. We attribute the difference in result when compared to BrunchD with three causes: 1) Bugs in the implementation of the invariants, 2) unfinished invariant implementation, and 3) lack of consolidation of the integrity invariants in the shared code base.

There are many invariants that do not get triggered correctly in BrunchC. In Datalog, there are three different types of change records that trigger the same property 17 shown in Figure 4.2. While it may appear simple in Datalog, since there is only one version of the code to maintain, in C, a different triggering change record would alter the order in which the check must be made, or the type of check that is to be made. We illustrated the problem of multiple and disjoint code segments in Section 4.4. Bugs in the implementation of Rule 24 and Rule 25 prevented the full detection of corruption on the `ram_bytes` field of `INLINE_EXTENT_DATA` and `num_bytes` field of `EXTENT_DATA`.

Mainly due to time constraint, BrunchC did not implement Rule 21 to 23; therefore, it was not able to catch the rare case when the `nodatasum` bit of the `flags` field in an `INODE_ITEM` object was flipped to 0. It also had an incomplete version of Rule 24, preventing it from detecting all corruptions in the `nbytes` field of `INODE_ITEM` object.

BrunchC also completely missed the `type` field corruption of `DIR_INDEX`. As we later found out, the structural invariant that checks the validity of the `type` field was placed inside the Datalog to C conversion module, and thus was not available to BrunchC. We intend to

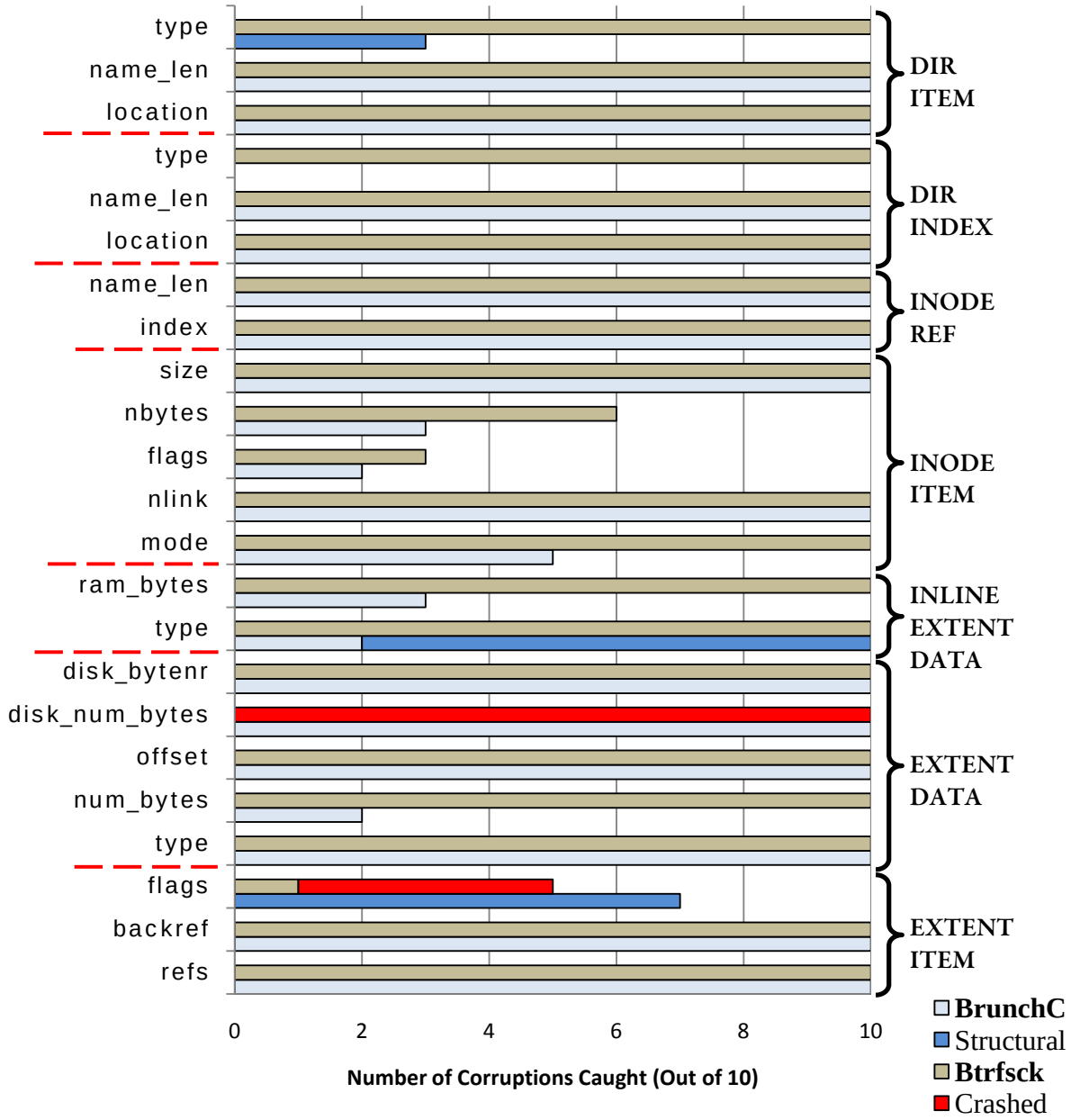


Figure 5.4: Corruption results for BrunchC vs. Btrfsck. The top line in each pair is the Btrfsck result, and the lower line for each pair is BrunchC's result. BrunchC is on par with Btrfsck.

consolidate all integrity invariants scattered across the code base and enumerate them as future work.

5.2.1 Limitations

A summary of possible reasons of why our corruption test did not cover all semantic invariants is shown in Table 5.2.

We do not perform corruption on the structural metadata of Btrfs (i.e., B-tree nodes and pointers), which also implies that we are unable to perform corruptions which would add spurious objects or wrongfully delete existing objects. The type of corruption described would require correct re-structuring of the B-tree. We consider it as future work to extend our corruptor to carry out such types of corruption.

We currently also do not corrupt the `ROOT_ITEM` structure nor do we attempt to create snapshots during the corruption tests. The reason for the exclusion is because Btrfsck v0.19 has a bug that falsely identifies an error with valid snapshot trees. As such, it would taint the result of all corruption workloads that create snapshots.

5.3 Performance

To get an idea of the costs of using a Prolog interpreter, we consider three configurations: Xen without Recon active ("Native"), Xen-Brunch with rules implemented in C, and Xen-Brunch with rules written in Datalog. All experiments are performed in the guest virtual machine, running in para-virtualized mode.

Our test machine has a Intel Q8200 Quad-core CPU clocked at 2.33GHz. We configure the guest VM with 1GB of RAM and one physical CPU. The guest storage is set up so that the operating system is stored on a virtual disk separate from our testbed, which excludes any operating system I/O from affecting our experiment. Our testbed virtual disk is linked to Brunch via blktap, with a capacity of 300GB. Only one guest was running on each machine.

Initially, we chose the Filebench benchmark suite because of its configurable and realistic workload. However, currently BrunchD cannot handle the load generated by Filebench because it does not scale well with large numbers of change records. We discuss this issue in more detail in Section 5.3.1.

Instead, we ran a simpler workload that creates files of varying size and builds a tree of directories. the size of the file is chosen at random from an exponential distribution with a mean of 16K bytes. The workload is setup such that it maintains at least 1,000 files at all times (except for at the beginning of the workload). We mimic the behaviour of modifying files by either resizing the file or updating one or more segments of the file. This is achieved through various options provided by the Linux `dd` command. This workload is much more sustainable for BrunchD, where the peak number of change records per transaction is about 40,000, and transaction checking takes roughly 1 minute to complete. We show a breakdown of the time it takes to check each invariant, as well as the time it takes to perform set differencing in Figure 5.5.

Rule 23 took the longest time and is responsible for one third of the total checking time because the rule requires a scan of all checksum entries in the checksum tree for each newly added extent. This behaviour is due to the inability to impose constraints on the inputs to the query primitive, even though we have the information to limit the search range. As future work, we would like to be able to give hints to the query primitive. Similarly, Rule 12, which is shown in Figure 3.11, also suffers from having to scan through the whole file system tree for potentially undeleted items.

For the experiment, we ran 10,000 commands per workload. Table 5.3 shows the result of our performance evaluation. We ran the workload for each configuration 22 times, and removed the worst and the best results from each data set, for a total of 20 experiments per configuration, shown in Table 5.3.

As is expected, our C implementation comes close to native performance, similar to the result obtained in our previous work [12], while our Datalog implementation performs about

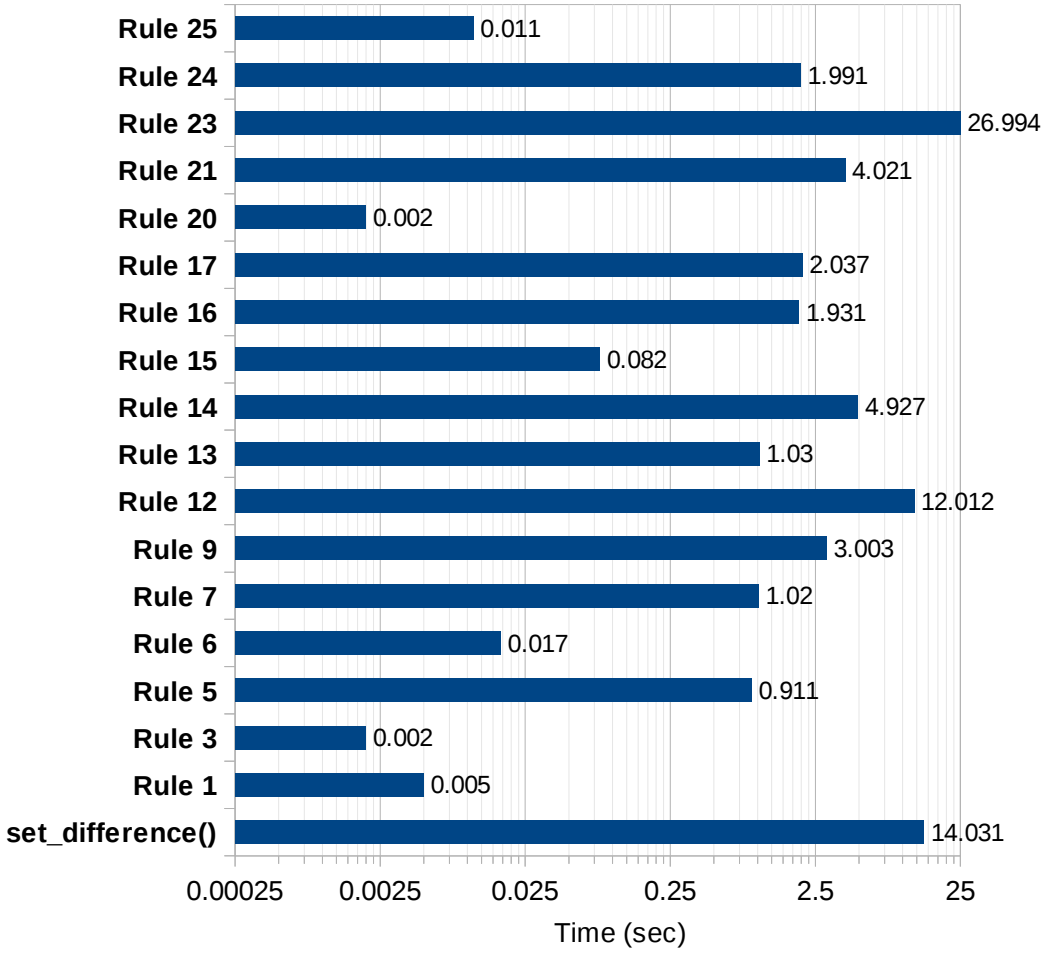


Figure 5.5: Breakdown of invariant checking cost for a transaction with 40877 change records. Rules that are not shown in the graph took less than 0.001 seconds to complete. The total checking time, including set differencing in Prolog, was 74.046 seconds. The total time to check a transaction, including generating change records, was 74.070 seconds.

	Native	**BrunchC**	**BrunchD**
Host Wall Clock Time	241.7±7.9s	244.8±11.0s	440.5±22.3s
tapdisk2 CPU Time	$<$ 1s	$<$ 1s	216.8±23.4s

	BrunchC	**BrunchD**
Overhead	1.28%	82.28%

Table 5.3: Cost of Invariant Checking. The overhead is calculated as $\frac{WallTime(Brunch)}{WallTime(Native)} - 1$, expressed in percentages. The total run time of each workload is measured from the host VM. tapdisk2's CPU time was obtained through the Linux `ps` command.

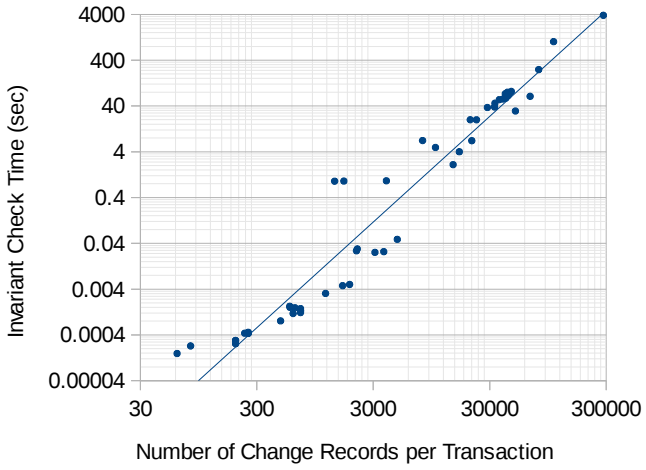


Figure 5.6: Total invariant checking time with respect to the number of change records per transaction. The slope of the trend line is 2.3 (more than O(n^2)). As the number of change record increases, the time required to check a transaction takes much longer. At 284,315 change records, it took 3816 seconds (63.6 minutes) to complete invariant checking.

twice as slow on average in terms of wall clock time. Below, we analyze the reasons for this slowdown.

5.3.1 Performance Analysis

The set differencing operation in Prolog is expensive, taking approximately 20% of the total transaction checking time, as shown in Figure 5.5. We would like to move the set differencing operation into the change record generation phase, written in C, where it can be performed more efficiently.

We believe a significant portion of the BrunchD overhead arises from inefficient matching in the Prolog interpreter. In particular, we found that the transaction checking time scales more than quadratically with the number of change records in the transaction, as shown in Figure 5.6. We believe that the matching performance of Datalog can be improved with better indexing in Prolog. We are also exploring other alternatives for improving BrunchD's performance, as we will discuss in Section 7.

Chapter 6

Related Work

In this chapter, we discuss research topics that are similar in goal to our work. We focus on three major topics: using declarative languages for the purpose of clarity, file system consistency checking and verification.

6.1 Using Declarative Languages

In this section, we present research that demonstrates the use of declarative languages to improve the conciseness and clarity of rules and policies.

SQCK [15] inspired our work with the concept of writing file system consistency properties declaratively. It demonstrated that by expressing consistency properties as compact SQL queries, the checks and repairs can be more easily understood, which in turn provides a stronger guarantee on their correctness. Gunawi et al. were able to improve upon the repairs made by e2fsck by correcting the order in which certain checks and repairs were performed and also by using more information that is provided redundantly by the file system. The main difference between SQCK and Brunch is that SQCK performs offline checks and repairs while Brunch performs online checks.

Datalog and its variants are popular in the field of networking for specifying routing policies [21, 23, 22, 19]. DAWN [19] is a platform for creating adaptive policy-based routing

protocols for mobile ad-hoc networks. Liu et al. demonstrates the protocol can be expressed concisely by writing the policies declaratively in a variant of Datalog named *NDLog*. The protocol specifications are orders of magnitude smaller than its C/C++ counterparts, which realistically allow for formal analysis of complex protocols. They found that the technique enables quick prototyping of hybrid protocols, similar to our experience with writing semantic invariants.

Declarative languages are also widely used for specifying and maintaining security policies [3, 8, 7]. Debar et al. [7] describes an architecture for specifying generic security policies in Datalog. Whenever new threats are detected, the policies may react to the threat and enable security rules such as prohibiting TCP/IP traffic for its web servers when syn-flooding attack occurs. The enabling of security rules holds some similarity to our query primitive in that it must be implemented outside the context of Datalog (i.e., the Datalog engine must invoke predicates written in foreign languages).

Schüpbach et al. [28] introduced a novel concept of encoding hardware configuration logic in a Prolog-like language for Linux device drivers. They reported that declarative logic programming provides clear separation of policy and mechanism, as well as separation of special cases. Similar to their finding, one of the improvements we noticed from writing declarative invariants is that each of them are independent of the others.

6.2 Consistency Checking and Verification

ZFS [4, 5, 24] has a consistency checker similar to our work that is able to perform consistency checking without taking the file system offline. The particular repair tool, named *scrub*, is able to detect metadata and data corruption via checksumming for blocks that have redundant copies. However, this technique may not be able to detect software bugs. For example, if the metadata was corrupted by the file system itself, it would have passed the checksum test.

Type-safe disks [29] extends the disk interface by exposing additional APIs to the file system for block allocation and pointer management. Such a design helps with enforcing basic invariants, for example, all allocated blocks must be referenced. Our work expands on this concept and we are able to check many more consistency invariants.

Our previous work was built upon many concepts introduced by semantically-smart disk system (SDS) [31], which performs metadata interpretation in the block layer to avoid modification of the target file system. SDS uses additional knowledge about the file system to improve performance or enhance functionality, such as performing secure delete and caching frequently updated blocks on high performance persistent media. This work, however, assumes an update-in-place file system such as Ext2 and implicitly assumes that the file system will not face arbitrary corruption, since it relies on the correctness of the values in the super block, which contains information on where to locate the inode blocks and bitmap blocks.

The Xok exokernel's XN storage system [11] is designed to support library file systems. One of the APIs that the file system library must implement is the `reboot` function, which would detect invariant violations. The `reboot` function is run after every disk write to ensure the file system is crash consistent. While it is not practical to check the full file system after every disk write, the author claims that with state partitioning and checkable hints from the file system, the `reboot` can be checkpointed. This concept is similar to our assumption that given a consistency pre-update state of the file system image, we can perform consistency checking without having to examine the entire disk by verifying the consistency of the update of the post-write stable state.

Chunkfs [17] attempts to reduce the time to check consistency by breaking the file system into chunks that can presumably be checked independent of each others. In practice, however, the chunks are not truly independent because path names may be extended to other chunks, and Chunkfs uses cross-chunk references to handle files that are spread across multiple chunks. Thus, the inter-chunk consistency properties would still have to be checked as a whole.

Sivathanu et al. [30] presented formal methods for proving the soundness of file system operations. However, it is restricted to Ext3-like file systems. For example, the pointer exclusivity property, which states that no two block pointers can point to the same block, does not hold for copy-on-write file systems that supports snapshots, such as Btrfs.

Chapter 7

Conclusions and Future Work

Our experience with the use of declarative languages to express and check consistency invariants has shown that it is a valuable technique to elucidate the otherwise enigmatic file system consistency properties. A major portion of our contribution comes from extracting these invariants and ensuring their correctness through rigorous testing. We found that it was much faster to conceptualize invariants, and to make corrections to them, in a declarative language when compared to our attempts of same challenge in C. We showed that Datalog is a natural fit for this problem. Invariants expressed in Datalog are clearer, easier to write and reason about than their counterparts in a low-level language.

While we were writing declarative invariants, we found that change records must be generated at a proper level of granularity and expressed in a proper format in order to improve the clarity and conciseness of invariants. To this end, we realized that in order to simplify their expression, the invariants must drive the format of the change records.

We discovered that there exists classes of integrity constraints which must be checked in a certain order. Structural constraints must be checked first to ensure that metadata interpretation will not fail in the face of arbitrary file system corruption. Reachability and uniqueness constraints must be checked next such that we allow the semantic invariants to make more assumptions about the correctness of the file system update. Checking invariants in this order

allows the semantic invariants, which are the most complex class of invariants, to be expressed in a concise and yet clear manner. Furthermore, we would detect a problem as close to its root cause as possible, which makes debugging easier.

In the future, we envision ways to translate invariants written in Datalog to C procedures automatically. Another possible direction to explore is to implement a Datalog parser optimized for file systems, similar to the Datalog parser that Loo et al. implemented for their declarative routing policies [21]. We are also investigating whether certain invariants can be automatically derived by using learning algorithms, similar to the approach of Engler et al. for inferring bugs from deviant behaviours [10]. Lastly, we are looking into designing a specification language that will automatically generate metadata interpretation and type-specific differencing module for any file system. Such a tool would significantly reduce the time to use our framework by eliminating the need to manually implement the module, which in itself is an error prone and time-consuming process.

Bibliography

[1] Jo ao Carlos Menezes Carreira, Rodrigo Rodrigues, George Candea, and Rupak Majumdar. Scalable testing of file system checkers. In *Proceedings of the ACM SIGOPS European Conference on Computer Systems (Eurosys)*, pages 239–252, 2012. 1

[2] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. In *Proceedings of the Symposium on Operating Systems Principles (SOSP)*, pages 164–177, October 2003. 4.2

[3] K. Bhargavan, C. Fournet, and A.D. Gordon. Verifying policy-based security for web services. In *Proceedings of the 11th ACM conference on Computer and communications security*, pages 268–277. ACM, 2004. 6.1

[4] J. Bonwick and B. Moore. ZFS - The Last Word in File Systems. `http://opensolaris.org/os/community/zfs/docs/zfs_last.pdf`. 2, 6.2

[5] Jim Bonwick. ZFS Block Allocation. `http://blogs.sun.com/bonwick/entry/zfs_block_allocation` Accessed Feb. 19, 2009. 6.2

[6] S. Ceri, G. Gottlob, and L. Tanca. What you always wanted to know about datalog (and never dared to ask). *Knowledge and Data Engineering, IEEE Transactions on*, 1(1):146–166, 1989. 3.2.1

[7] H. Debar, Y. Thomas, F. Cuppens, and N. Cuppens-Boulahia. Enabling automated threat response through the use of a dynamic security policy. *Journal in Computer Virology*, 3(3):195–210, 2007. 6.1

[8] J. DeTreville. Binder, a logic-based security language. In *Security and Privacy, 2002. Proceedings. 2002 IEEE Symposium on*, pages 105–113. IEEE, 2002. 6.1

[9] Btrfs Developers. On-disk Format, btrfs Wiki. https://btrfs.wiki.kernel.org/index.php/On-disk_Format. 5.2

[10] Dawson Engler, David Yu Chen, Seth Hallem, Andy Chou, and Benjamin Chelf. Bugs as deviant behavior: a general approach to inferring errors in systems code. In *Proceedings of the Symposium on Operating Systems Principles (SOSP)*, pages 57–72, 2001. 7

[11] D.R. Engler. *The Exokernel operating system architecture*. PhD thesis, Massachusetts Institute of Technology, 1998. 6.2

[12] Daniel Fryer, Jack Sun, Rahat Mahmood, TingHao Cheng, Shaun Benjamin, Ashvin Goel, and Angela Demke Brown. Recon: Verifying file system consistency at runtime. In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST)*, February 2012. 1, 3.2, 4.1, 5.3

[13] Gregory R. Ganger, Marshall Kirk McKusick, Craig A. N. Soules, and Yale N. Patt. Soft updates: a solution to the metadata update problem in file systems. *ACM Transactions on Computer Systems*, 18(2):127–153, 2000. 1

[14] Haryadi S. Gunawi, Vijayan Prabhakaran, Swetha Krishnan, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Improving file system reliability with I/O shepherding. In *Proceedings of the Symposium on Operating Systems Principles (SOSP)*, pages 293–306, 2007. 1

[15] Haryadi S. Gunawi, Abhishek Rajimwale, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. SQCK: A declarative file system checker. In *Proceedings of the Operating Systems Design and Implementation (OSDI)*, December 2008. 1, 3.1.1, 3.2, 6.1

[16] R. Hagmann. Reimplementing the Cedar file system using logging and group commit. In *Proceedings of the Symposium on Operating Systems Principles (SOSP)*, November 1987. 1

[17] Val Henson, Arjan van de Ven, Amit Gud, and Zach Brown. Chunkfs: Using divide-and-conquer to improve file system reliability and repair. In *Proceedings of the Workshop on Hot Topics in System Dependability (HotDep)*, 2006. 1, 6.2

[18] Dave Hitz, James Lau, and Michael Malcolm. File system design for an NFS file server appliance. In *Proceedings of the USENIX Technical Conference*, 1994. 1

[19] C. Liu, R. Correa, X. Li, P. Basu, B.T. Loo, and Y. Mao. Declarative policy-based adaptive mobile ad hoc networking. *IEEE/ACM Transactions on Networking (ToN)*, 20(3):770–783, 2012. 6.1

[20] M. Liu. Overview of datalog extensions with tuples and sets. *GMD–Forschungszentrum Informationstechnik GmbH*, page 99, 1998. 3.3.2, 4.3

[21] Boon Thau Loo, Tyson Condie, Joseph M. Hellerstein, Petros Maniatis, Timothy Roscoe, and Ion Stoica. Implementing declarative overlays. In *Proceedings of the Symposium on Operating Systems Principles (SOSP)*, pages 75–90, 2005. 6.1, 7

[22] B.T. Loo, T. Condie, M. Garofalakis, D.E. Gay, J.M. Hellerstein, P. Maniatis, R. Ramakrishnan, T. Roscoe, and I. Stoica. Declarative networking: language, execution and optimization. In *Proceedings of the 2006 ACM SIGMOD international conference on Management of data*, pages 97–108. ACM, 2006. 6.1

[23] B.T. Loo, J.M. Hellerstein, I. Stoica, and R. Ramakrishnan. Declarative routing: extensible routing with declarative queries. In *ACM SIGCOMM Computer Communication Review*, volume 35, pages 289–300. ACM, 2005. 6.1

[24] Sun Microsystems. Zfs. http://opensolaris.org/os/community/zfs. 6.2

[25] Vijayan Prabhakaran, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Model-based failure analysis of journaling file systems. In *Proceedings of the IEEE Dependable Systems and Networks (DSN)*, pages 802–811, 2005. 1

[26] 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 Symposium on Operating Systems Principles (SOSP)*, pages 206–220, 2005. 1

[27] Ohad Rodeh, Josef Bacik, and Chris Mason. Btrfs: The linux b-tree filesystem. Technical Report RJ10501 (ALM1207-004), IBM Research, July 2012. 2

[28] Adrian Schüpbach, Andrew Baumann, Timothy Roscoe, and Simon Peter. A declarative language approach to device configuration. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 119–132, March 2011. 6.1

[29] Gopalan Sivathanu, Swaminathan Sundararaman, and Erez Zadok. Type-safe disks. In *Proceedings of the Operating Systems Design and Implementation (OSDI)*, pages 15–28, 2006. 6.2

[30] Muthian Sivathanu, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, and Somesh Jha. A logic of file systems. In *FAST'05: Proceedings of the 4th conference on USENIX Conference on File and Storage Technologies*, pages 1–1, Berkeley, CA, USA, 2005. USENIX Association. 6.2

[31] Muthian Sivathanu, Vijayan Prabhakaran, Florentina I. Popovici, Timothy E. Denehy, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Semantically-smart disk systems. In *USENIX Conference on File and Storage Technologies (FAST)*, pages 73–88, 2003. 1, 3.1.2, 6.2

[32] Jack Sun, Daniel Fryer, Ashvin Goel, and Angela Demke Brown. Expressing invariants for protecting file-system integrity. In *Proceedings of the Workshop on Programming Languages and Operating Systems (PLOS)*, 2011. 3.3.5

[33] Jan Wielemaker, Tom Schrijvers, Markus Triska, and Torbjörn Lager. SWI-Prolog. *Theory and Practice of Logic Programming*, 12(1-2):67–96, 2012. 4.3

[34] Xen Developers. blktap - Xen Wiki. http://wiki.xensource.com/xenwiki/blktap. 4.2

[35] Junfeng Yang, Can Sar, Paul Twohey, Cristian Cadar, and Dawson Engler. Automatically generating malicious disks using symbolic execution. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 243–257, 2006. 1

[36] Junfeng Yang, Paul Twohey, Dawson Engler, and Madanlal Musuvathi. Using model checking to find serious file system errors. *ACM Transactions on Computer Systems*, 24(4):393–423, 2006. 1

[37] Yupu Zhang, Abhishek Rajimwale, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. End-to-end data integrity for file systems: a ZFS case study. In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST)*, 2010. 1