Using Declarative Invariants for Protecting File-System Integrity

Jack Sun, Daniel Fryer, Ashvin Goel and Angela Demke Brown
University of Toronto

ABSTRACT

We have been developing a framework, called Recon, that uses runtime checking to protect the integrity of file-system metadata on disk. Recon performs consistency checks at commit points in transaction-based file systems. We define declarative statements called consistency invariants for a file system, which must be satisfied by each transaction being committed to disk. By checking each transaction before it commits, we prevent any corruption to file-system metadata from reaching the disk.

Our prototype system required writing the consistency invariants in C. In this paper, we argue that using a declarative language to express and check these invariants improves the clarity of the rules, making them easier to reason about, verify, and port to new file systems. We describe how file system invariants can be written and checked using the Datalog declarative language in the Recon framework.

1. INTRODUCTION

Existing file-system reliability methods, such as checksums, redundancy, or transactional updates, provide limited defenses against in-memory file-system corruption [10, 9, 15]. Whether the corruption occurs due to bugs in the file system or operating system code or random memory errors, it can result in data loss, the return of corrupt data to the user, persistent application crashes, or even security exploits [13].

We have been developing a framework, called Recon, that aims to protect file-system metadata integrity in the face of *arbitrary* operating system system bugs or memory errors. Recon interposes at the block level between the file system and the storage device, and uses a set of high-level rules called *consistency invariants* that help enforce file-system consistency at runtime. For example, a consistency invariant in the Linux ext3 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 bitmap within the same transaction. Consistency invariants differ across file systems because each file system provides unique features and uses different metadata structures. Recon therefore provides a general framework for interpreting file-system specific metadata. For example, when data is appended to a file, Recon helps determine that one of the updated blocks is an ext3 indirect block and a pointer value has been updated in it, and another updated block is a bitmap block and certain bits have been flipped in the block. Our previous work describes how the framework addresses three issues: when to check metadata consistency, what properties to check, and how to check them [1]. Section 2 provides an overview of the framework.

This paper focuses on *expressing* consistency invariants correctly. It is vital that a software layer designed to enhance reliability, such as the Recon framework, does not itself destabilize the system. Consistency invariants need to be stated 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. Our previous implementation of the Recon framework, for the ext3 file-system, required writing invariants in hand-crafted C code. Using a low-level language to express high-level invariants 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` utility, which intermingles metadata interpretation and checks (both written in C), has bugs that cause additional file-system damage when repairing an inconsistent file system [5]. Their solution is SQCK, an offline consistency check tool that translates the checks and repairs performed by e2fsck into SQL. In fact, our C implementation of the ext3 invariants was based on the SQCK statements, since they were much easier to reason about than the e2fsck code from which they were derived.

In this work, we explore using a declarative language for *online* file-system consistency checking because declarative languages are naturally designed for making runtime assertions. For example, each consistency invariant is written as a set of declarative statements and run independently of the other invariants. The invariant code is thus much easier to reason about than invariants written in a low-level language that are intermingled with each other and with the metadata interpretation code. We chose to use Datalog (a derivative of Prolog) for checking invariants because it has been used successfully for specifying correct behavior in several systems [8, 3, 11]. Moreover, it is a better match for checking

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
PLOS '11, October 23, 2011, Cascais, Portugal.
Copyright 2011 ACM 978-1-4503-0979-0/11/10 ...$10.00.

file systems than SQL. For example, Datalog supports recursion and thus allows expressing properties of recursive file-system data structures more easily. By choosing a language that fits the nature of the problem, we can improve the *clarity* of the invariants that we wish to express. This feature is particularly important since we expect that it will make it easier to identify commonality in the invariants used by different file systems.

We evaluate the feasibility of using declarative invariants in the Recon framework. To facilitate this evaluation, we have created a simplified user-space implementation of an FFS-like file system called TestFS, and a user-level implementation of Recon that uses the Datalog language for the specification of its invariants. We contrast the simplicity of the Datalog invariants against those written in C and discuss the extent to which invariants written for our test file system are applicable to ext3 in Section 3. The main drawback of using declarative languages for checking invariants is performance overhead, which we examine in Section 4.

2. BACKGROUND

In this section, we review existing techniques for improving file system reliability. We then describe our approach for checking file-system consistency at runtime.

2.1 File System Reliability

File-system corruption can be caused by crash failures, storage hardware errors, file-system/OS bugs and memory corruption. Storage systems have primarily focused on the first two problems. Journaling [6], copy-on-write [7], and soft updates [2] address crash failures. Checksums and redundancy reduce the probability of data loss due to hardware or low-level software failure [4].

The complexity of modern file systems leads to bugs that can be hard to detect, even under heavy testing [10, 14]. When a file or operating system bug corrupts file-system data, it requires complex recovery procedures. Current solutions fall in two categories, both of which are unsatisfactory. Disaster recovery methods, such as backups and snapshots, can result in loss of recent data. The alternative is to use an offline consistency check mechanism, provided by most file systems, for restoring file system consistency. Since a consistency checker operates on a current snapshot of the file system, and a corruption may have occurred a while ago, correct repair may simply be impossible. Both solutions result in significant downtime, and put data at risk because they are run after the failure may have corrupted significant file-system state. To truly prevent downtime and data loss, corruption must be prevented from propagating to disk.

2.2 Checking Consistency at Runtime

The Recon system achieves its goal of preventing file-system metadata corruption by interposing between the file system and the storage device at the block layer and checking a set of consistency invariants before permitting writes to reach the disk. Next, we describe the three challenges that arise when checking file-system consistency at runtime, and how we address them in the Recon framework.

When to check consistency:. The in-memory copies of metadata may be temporarily inconsistent during file system operation, and so it is non-trivial to check consistency properties at arbitrary times. In a journaling file system, transaction commits are well-defined points at which the file system believes itself to be consistent. Hence, transaction boundaries serve as convenient vantage points for verifying consistency properties. A consistency violation at these points indicates a bug or a memory corruption.

What properties to check:. We can derive an informal specification of metadata consistency properties from offline file-system consistency checkers, such as the Linux e2fsck program. These *consistency properties* define what it means to have consistent metadata on disk. Our aim is to ensure that any metadata committed to disk will maintain these same consistency properties. Unfortunately, consistency properties are *global* statements about the file system. For example, a simple check implemented by e2fsck is that *"all* live data blocks are marked in the block bitmap". Checking these global properties requires a full disk scan.

Instead, we manually derive a *consistency invariant* from each consistency property. The invariant is a local assertion that must hold for a transaction to preserve the corresponding file system consistency property. For example, the consistency invariant for the "all live data blocks are marked in the block bitmap" property 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 bitmap within the same transaction, i.e., the invariant is "block pointer set to N from 0 $\Leftrightarrow$ bit N set in bitmap". This invariant can be checked by examining only the updated blocks, i.e., the updated pointer block, the allocated block and the updated block bitmap must all be part of the same transaction. It is possible to transform consistency properties into invariants because file systems keep themselves consistent without examining the entire disk state. In other words, our invariant checking should not require much more data than the file system itself needs for its operations. This is backed up by our experience with implementing invariants in C for the ext3 and btrfs file systems.

How to check invariants:. Checking consistency invariants requires examining the current state of the file system and the metadata updates at each transaction commit point. The invariants are expressed in terms of logical file-system data structures (e.g., current and updated values of block pointers, bits in block bitmap). However, Recon observes physical block updates below the file system, because it cannot trust a buggy file system to provide the correct logical data structure information. We bridge this semantic gap by reinterpreting the metadata from the updated physical blocks, similar to semantically smart disks [12].

Figure 1 shows the architecture of the Recon system for the ext3 and btrfs file systems. Recon uses the Linux device mapper framework to interpose on all file system requests at the block layer, allowing it to track all file-system metadata accessed from disk. The read cache allows accessing the pre-update file-system state (i.e., disk state) efficiently. The post-update state is derived by superposing the write cache on the read cache. When a transaction commits, Recon interprets metadata by invoking file-system specific Recon API calls. The file-system specific code determines the types of the updated blocks and the data structures within them by starting from known block types and determining the types of the unknown blocks iteratively. This process is possible because the file-system has a known root and is

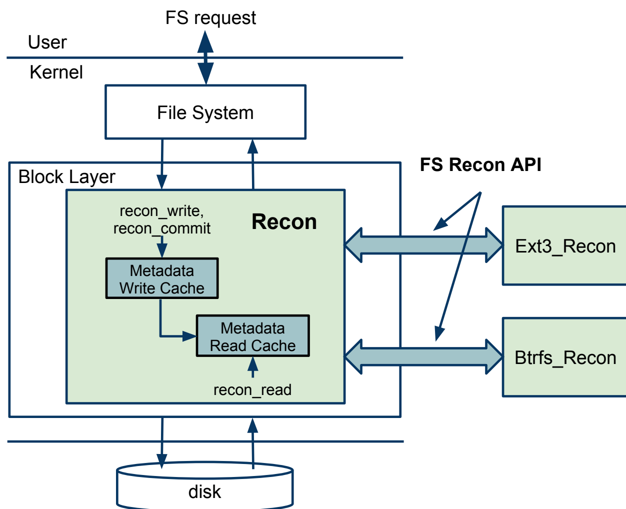

Figure 1: The Recon Architecture

arranged in a connected graph. If a corruption breaks these assumptions, we would detect it as an invariant violation.

Once the type for each block in the write cache is known, the file-system specific code performs a data-structure level diff between the block in the write cache and its corresponding version in the read cache to generate *change records* at the granularity of data-structure fields.[1] A change record has the following general format:

[type, id, field, oldval, newval]

The *type* specifies a data structure (e.g., inode, directory block), the *id* is the identifier of a specific object of the given type (e.g. inode number), and *field* is a field in the structure (e.g. inode size). The values *oldval* and *newval* are the old and new values of the corresponding field. The change records are used for checking invariants, as described in detail in the next section. When all invariants are checked successfully, the transaction is allowed to commit, after which the write cache is merged with the read cache, updating Recon's view of file-system state.

3. INVARIANT CHECKING IN DATALOG

The consistency invariants in the Recon framework are structured as assertions on a set of facts. These facts come from two sources, the Recon read cache that describes the pre-update state of the file system, and change records that describe the logical metadata updates made by a transaction.

Datalog is a natural fit for expressing assertions on these facts, because it is built around the notion of facts and predicates. Its declarative approach allows the specification to more closely match the domain, making it easier to reason about the checks. In addition, Datalog can handle recursive predicates, which fits naturally when operating over file-system data structures. Our eventual aim is to gain a broad understanding of file system consistency. We expect that high-level, compact representations of consistency invariants will help clarify the commonalities between file systems, even though they may have significantly different data structures.

[1]Change records are also generated for metadata blocks that were valid in the read cache but are not valid after the commit.

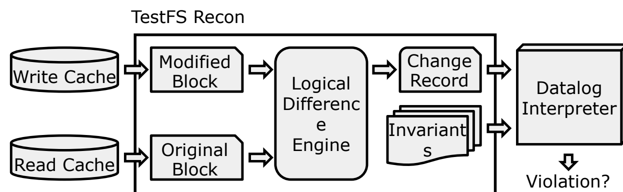

Figure 2: Integrating Datalog in the Recon Framework

To assess the feasibility of our approach, we have implemented a simplified version of ext3 as a user-space file system called TestFS. A user-space file system avoids the need to port the Datalog interpreter to the kernel. Most of the consistency invariants that apply to TestFS are derived directly from those of ext3. The main differences concern ext3 orphan list handling and corner cases involving unused inodes. TestFS consists of four main structures, a superblock, allocation bitmaps, inodes, and directory entries. There are two bitmaps in TestFS, the block and inode allocation freemaps.

When TestFS Recon generates change records, they are added as facts to the Datalog environment, as shown in Figure 2. We encode a change record [type, id, field, oldval, newval] as a straightforward Datalog fact in the format: `change(type, id, offset, old, new)`. A directory consists of a list of directory entries, requiring a slight modification: `change(type, action, dir_inode_id, name, inode_id, name_size)`, where *action* is *add* or *remove*.

Before a file system commits a transaction, Recon invokes the interpreter to evaluate all the invariants. The invariants can use the facts generated from the change records, or they can use two custom primitives, written in C, to query the read cache. These custom primitives are: 1) dir_get_entry(DIN, NAME, IN), which given as input a directory inode DIN, and name NAME, returns the inode number of the directory entry in the IN argument, and 2) inode_get(IN, FIELD, VALUE), which given as input an inode IN, and a field number FIELD in the inode, returns the value of the field in the read cache in VALUE. TestFS requires 12 consistency invariants, while ext3 requires 33, mainly because it has more features. However, each of the TestFS Datalog invariants corresponds to one of the ext3 C invariants.

If the transaction does not violate any consistency invariants, then the change-record based facts in the Datalog environment are discarded before the next transaction is checked, because these facts are incorporated in the read cache (because the write cache is merged with the read cache). Next, we present two examples to show the benefits of specifying invariants declaratively.

3.1 Block Bitmap and Block Pointers

We express invariants in Datalog with predicates that return a true value when an invariant violation occurs. Recall that the ext3 block allocation invariant, described in Section 2.2, is "block pointer set to N from 0 ⇔bit N set in bitmap". We express this biconditional invariant in a straightforward way using 2 predicates, R1_violation and

```
R1_violation(IN,BN) :- block_allocated(IN,BN), not(change(b_freemap,_,BN,_,1)).
R2_violation(BN) :- change(b_freemap,_,BN,0,1), not(block_allocated(_,BN)).
```

Figure 3: Block Allocation Invariants

```
get_block_type(IN, TYPE) :- change(inode, IN, 0, _, TYPE).
get_block_type(IN, TYPE) :- inode_get(IN, 0, TYPE).

dir_get_parent(IN, PIN) :- change(dir_block, add, IN, '..', PIN, _).
dir_get_parent(IN, PIN) :- IN \= 0, dir_get_entry(IN, '..', PIN).

path(IN,PIN) :- dir_get_parent(IN, PIN).
path(IN,AIN) :- dir_get_parent(IN, PIN), path(PIN, AIN).

cycle(IN) :- path(IN, IN).

cycle_violation(PIN, NAME, IN) :- change(dir_block, add, PIN, NAME, IN, _),
                                  NAME \= '.', NAME \= '..',
                                  get_block_type(IN, TYPE), TYPE = "DIR",
                                  cycle(PIN).
```

Figure 4: Directory Cycle Invariant

R2_violation, in which IN is an inode number and BN is a block number, as shown in Figure 3. When either predicate succeeds, the allocation invariant is violated. The R1_violation predicate succeeds when 1) a block BN is allocated to inode IN, and 2) a change record fact, stating that a bit corresponding to block BN is set to 1 in the block bitmap, is *not* seen in the transaction.[2] Block deallocation requires similar invariants.

In contrast to the Datalog invariants, it is hard to reason about the correctness of our C implementation of this consistency invariant for ext3 because the code is divided into several parts. Invariant checking in C generally starts by using custom code that pattern matchs change records based on the premise of the invariant. When such a match occurs, some invariants accumulate bookkeeping information until the end of the transaction, as described in the example below. Finally, the assertion is implemented in a final processing phase.

In the example above, the C implementation obtains allocated blocks by matching [_, _ , block_pointer_field, 0, X] change records. When such a match occurs, it inserts a flag with key X into a rule-specific bookkeeping table indicating a new pointer has been set to X. It also matches for [block bitmap, Y, _, 0, 1] change records. In this case, it inserts a different flag into the same table with key Y, indicating bit Y in the allocation bitmap is set. During final processing, the implementation verifies that for each key in the table, both flags are set. Otherwise the invariant has been violated. While none of this code is complex, it is scattered across several phases of processing, making it difficult to infer this rule from the code, or be assured that the rule is implemented correctly.

3.2 Directories

Another consistency property in ext3 and TestFS is that a directory must not be part of a cycle (or else it may not be connected to the root). The corresponding invariant must be checked whenever a directory's parent changes, since it is possible that the directory was made the child of one of its children, creating a cycle.

Figure 4 defines a predicate called cycle(), which returns true if a directory with inode number IN is part of a cycle. This predicate uses the recursive path(IN, AIN) predicate, which returns true if a path (with edges consisting of ".." entries) exists between inode IN and an ancestor inode AIN. The path predicate uses dir_get_parent() to find the parent directory inode of an inode. The dir_get_parent() predicate matches an appropriate dir_block change record or uses the dir_get_entry() primitive to access the Recon read cache.

The cycle_violation() invariant looks for cycles in newly created or moved directories. It ensures that we do not inspect a newly created '.' or '..' entry (which are created when a new directory is created), and that the entry is in fact a directory (which uses the inode_get primitive to access the Recon read cache). Other directory invariants, e.g., a directory must remain connected to the root, or has exactly one '.' and one '..' entry, can also be expressed easily. The C implementation of these invariants requires much more code, and is spread across different processing phases.

	Original	With Checking	Overhead
User	17.8±0.2s	36.4±0.1s	2.04X
System	22.5±0.1s	23.1±0.1s	1.03X
Sleep	545.4±9.1s	604.9±9.0s	1.11X
Total	585.8±9.2s	664.4±9.1s	1.13X

Table 1: Cost of Datalog Invariant Checks

4. EVALUATION

To get an idea of the costs of using Datalog, we stressed TestFS by running a metadata update-intensive workload, consisting of roughly 203K directory and file create, file write, remove, and directory change operations. We expect to see higher overheads than a full implementation would in-

[2]The _ character in the argument of a Datalog predicate matches any value. Also, the block_allocated() predicate, consisting of two simple clauses, is not shown.

cur for two reasons. First, TestFS performs much less work than a full-featured file system. Second, we have embedded the DES Datalog interpreter in TestFS-Recon, without making any optimizations for the types of invariants used in our implementation.

TestFS uses a regular file as its storage device; we open this file with the O_SYNC flag to mimic a write-heavy file system since the entire file system is small enough to fit in memory. Our test machine has an Intel Xeon X430 quad-core processor at 2.4 GHz with 4GB of RAM; it is running Debian Linux version 6.0, kernel version 2.6.32.

Table 1 shows the average of 28 trials, both with and without checking (we performed a total of 30 trials and removed the fastest and slowest run from each set), including 95% confidence intervals.[3]

As expected, checking invariants has a large impact on user time while the effect on system time is negligible, since we have a user-space implementation. The inflation in sleep time (which includes I/O wait time) is surprising, and we are still investigating the cause. The overall 13% overhead is encouraging, and we expect that an optimized datalog interpreter developed for Recon will reduce overhead further.

5. CONCLUSIONS

Our ongoing research towards improving file system reliability is exploring the use of declarative languages to express and check consistency invariants. We showed that Datalog is a natural fit for this problem. The prior file-system state and metadata updates are facts and the consistency invariants are assertions on these facts. Invariants expressed in Datalog are clearer, easier to write, and reason about than their counterparts in a low-level language like C.

Currently, we are tuning the Datalog interpreter to reduce its overheads. We are implementing the full set of ext3 and btrfs invariants in Datalog, which will help determine whether Datalog provides sufficient expressiveness for implementing invariants for widely-used file systems. We are also investigating whether certain invariants can be automatically derived by using learning algorithms.

6. REFERENCES

[1] Daniel Fryer, Rahat Mahmood, Ashvin Goel, and Angela Demke Brown. Verifying file system consistency at runtime. Technical report, University of Toronto, 2011. `http://hdl.handle.net/1807/27754`.

[2] 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.

[3] Haryadi S. Gunawi, Thanh Do, Pallavi Joshi, Peter Alvaro, Joseph M. Hellerstein, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, Koushik Sen, and Dhruba Borthakur. Fate and destini: a framework for cloud recovery testing. In *Proceedings of the Networked Systems Design and Implementation (NSDI)*, April 2011.

[4] 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.

[5] 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.

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

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

[8] 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.

[9] 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*, pages 802–811, 2005.

[10] 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.

[11] 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*, pages 119–132, March 2011.

[12] 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*, pages 73–88, 2003.

[13] 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.

[14] 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.

[15] 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*, 2010.

[3] Our C implementation of the invariants exists for the kernel Recon implementation for ext3 and btrfs but not for the user-level Recon implementation for TestFS, and hence we are unable to compare overheads with a C implementation.