Robust Consistency Checking for Modern Filesystems

Kuei Sun, Daniel Fryer, Dai Qin, Angela Demke Brown, and Ashvin Goel

University of Toronto

Abstract. We describe our approach to building a runtime file system checker for the emerging Linux Btrfs file system. Such checkers verify the consistency of file system metadata update operations before they are committed to disk, thus preventing corrupted updates from becoming durable. The consistency checks in Btrfs are complex and need to be expressed clearly so that they can be reasoned about and implemented reliably, thus we propose writing the checks declaratively. This approach reduces the complexity of the checks, ensures their independence, and helps identify the correct abstractions in the checker. It also shows how the checker can be made robust against arbitrary file system corruption.

Keywords: Runtime file system checker, Btrfs, Datalog, Consistency invariants.

1 Introduction

A *runtime* file-system consistency checker verifies the consistency of file-system update operations before they are committed to disk. File system metadata corruption can thus be detected *before* it propagates to disk, which minimizes data loss. In contrast, traditional offline checkers [1,4] require the file system to be taken offline to be checked for possible corruption, which can incur significant downtime [5]. Recon [3] enforces the consistency of the Linux Ext3 file system at runtime by checking that metadata updates conform to a set of rules called *consistency invariants*. These invariants are expressed in terms of the file system data structures, which are inferred outside the file system at the block layer using *metadata interpretation*, similar to semantically smart disks [6].

We describe the challenges with designing and building a robust, accurate and complete runtime checker for the Linux Btrfs file system. Since Btrfs is still under active development, 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 the new file system. Compared to Ext3, Btrfs uses many more file system data structures with vastly complex relationships, which complicate both the metadata interpretation and the consistency invariants considerably. Thus, it is of paramount importance that consistency invariants for Btrfs are *expressed* clearly and concisely so that they can be reasoned about and implemented reliably.

We use a declarative language to express the Btrfs consistency invariants, which is similar in spirit to Gunawi et al.'s [4] offline consistency checker written in SQL. This approach makes it easier to reason about the runtime checker's correctness in three ways. First, each consistency invariant can be written as a set of declarative statements and run independently of the other invariants. Second, the declarative style helps to identify the appropriate abstractions for representing file system metadata updates; the conceptual invariants are written as clearly as possible, and the metadata is interpreted accordingly. Last, the declarative approach clarified two distinct categories of invariants: the first expresses constraints on structural properties of the metadata (e.g., bounds checking) and the second expresses semantic properties (e.g., the agreement between directory entries and inode link counts).

2 Robust Consistency Checking

Our Btrfs runtime consistency checker has two goals: 1) it should detect all consistency violations, and 2) it should work correctly and predictably in the presence of arbitrary file system corruption failures. We meet these goals with two design principles. First, the semantic invariants must be written declaratively and concisely, making it easier to reason about their correctness. Second, the file system's structural invariants should be checked before performing any semantic checks so that the latter can depend on the structural integrity of the file system.

2.1 Abstractions for Runtime Checking

Here we provide an overview of how invariants are checked in a runtime file system checker. 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. The checking operation verifies that when the logical file system changes are applied to consistent, pre-transaction file system state, they will result in consistent, post-transaction file system state.

Invariant checks are performed using two abstractions. The first is the *change record*, which captures any modifications to file system objects, such as the addition of a new object, an update to an existing object, or the removal of an object in a transaction. For example, a change record for Btrfs can be expressed as: `change(TREE, ID, FIELD, OLD, NEW)`. Here, `TREE` is the Btrfs B-tree within which the object resides, while `ID` is the unique identifier of the object that is being changed (e.g., a Btrfs key for an inode). The TREE and ID uniquely identify Btrfs objects. The `FIELD` is a specific part of the object (e.g., inode size). Finally, `OLD` and `NEW` are the old and new values of the corresponding field.

The second abstraction is the query primitive, which is 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. The primitives return the most recent version of the object, from either the checking framework's internal caches or the disk. There are two types of primitives, `query()` for retrieving an object by key, and `prev()/next()` for finding the previous or next Btrfs key in a tree, as shown in Figure 1.

2.2 Expressing Invariants

Btrfs is a highly complex file system with correspondingly complex consistency properties. These properties are hard to extract from the C source code of btrfsck, the file

```
% 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)),
    prev(TREE_ID, k(EXTENT, extent_item, SIZE),
        k(EXTENT_PREV, extent_item, SIZE_PREV)),
    EXTENT < EXTENT_PREV + SIZE_PREV.

% the underscore '_' is a "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.
```

Fig. 1. Btrfs invariant "If a new extent item is added, it must not overlap previous or next extents"

system checker for Btrfs, because they are implemented piecemeal and intermingled with the checker's metadata interpretation code. When we converted the consistency properties to their corresponding runtime invariants and implemented them 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.

Instead, we chose to express consistency invariants in Datalog, a declarative logic programming language [2]. 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). We express the change records generated from a file system transaction as Datalog facts. Semantic invariants are statements that must hold true for a consistent file system. In Datalog, we negate these invariants to reach the conclusion that an invariant has been violated. For example, for an invariant $A \Rightarrow B$, the corresponding Datalog statement is $violation \vdash A, \neg B$ where A is a condition which will trigger the check B. The predicate A looks for a change in the file system by matching on the attributes of a change record. The predicate B can match change records or invoke primitives to access unmodified objects.

Figure 1 shows the Datalog invariant that checks for extent overlap. The `add(TREE, ID)` clause looks for an `extent_item` object with the Btrfs key ID that has been added to the file system and binds the `TREE_ID`, `EXTENT` and `SIZE` variables to its values. The `prev()` and `next()` clauses are primitives that query the file system state and bind the previous and next items in the tree to their second argument, respectively. We need a query 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 new extent and the previous or next extents returned by the primitives. When an extent does not have a previous or next extent, the relevant query will fail, indicating that the invariant has not been violated. Note that this invariant is independent of the metadata interpretation code and other consistency invariants, making it easier to reason about.

2.3 Checking Structure before Semantics

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 (metadata interpretation, query primitives, and invariant checking) are robust to metadata corruption. Invariant checking operates on change records generated by metadata interpretation and uses query primitives. Hence, its robustness depends on the first two components. Both metadata interpretation and query primitives access the current file system state, including the possibly corrupt metadata blocks that need to be checked. Thus, these components must perform careful validation.

Metadata interpretation requires checks to ensure that file system data structures are correctly typed, so that they can be interpreted correctly (e.g., 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 requirements lead to three invariants that need to be checked in order:

Type Safety: Type safety ensures that interpretation of updated metadata is robust to data corruption. Consider a query primitive `query(TREE, ID, VALUE)` that binds `VALUE` to a given object with identifier `ID` within tree TREE. Here `ID` incorporates the type of the object (e.g., the type in the Btrfs key). Type safety ensures that the object bound to `VALUE` will be of the same type as that specified in `ID`. The metadata interpretation code will therefore operate on correctly typed objects. Type safety is hard to enforce dynamically because file system data structures do not usually provide type information (e.g., a tag associated with each type). Even if they did, 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 must lie within the extent. Similarly, enumerated values (enum in C) need to be valid instances, and any length fields in structures must lie within expected bounds. If these checks fail, we raise a type-safety violation.

Reachability Invariants: The query primitives require more than type safety. For example, `query(TREE, ID, VALUE)` would not return an existing object that has been misplaced in a B-tree, because it assumes that keys are ordered (otherwise it would need to perform an expensive full tree search). In Btrfs, we enforce reachability by checking that a parent points to the correct child node, and keys are sorted correctly. Reachability invariants also ensure that primitives will not encounter an infinite loop in the B-tree.

Uniqueness Invariants: The primitives expect that all objects are uniquely identified by an identifier. If multiple objects have the same identity 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, only one of the changes would be checked. We check reachability before uniqueness, because if an object is reachable, it is easy to test for uniqueness by first searching for the object.

```
1. nr_items != 0 && nr_items < PTRS_PER_BLOCK
2. p.ptr[i].key == c.ptr[0].key
3. p.ptr[i].blockptr == c.header.bytenr
4. p.ptr[i].generation == c.header.generation
5. ptr[i].key < ptr[i+1].key
```

Fig. 2. The structural invariants on an internal B-tree node in Btrfs (p and c are parent and child)

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

Fig. 3. Btrfs invariant "Directory entry type is the same as the type of the inode"

After the three types of structural invariants 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 invariants can depend on well-formed change records being generated (even though their contents may be corrupt) and the primitives working correctly.

Figure 2 shows the five structural invariants that we check for B-tree internal nodes. An internal node consists of a header and an array of key pointers. The `header` contains the number of key pointers in the node (`nr_items`), the location of the node on disk (`bytenr`), and the generation number of the node. A key pointer (`ptr[]`) contains a Btrfs key, the location of the node pointed to by the key (`blockptr`) and the generation of the pointed-to node. Invariant 1 is a type-safety check on the key pointer array. Invariants 2 to 4 are reachability invariants that verify that the parent points to the correct child node. Invariant 5 checks that all keys in a valid B-tree node must be monotonically increasing, a requirement that provides both reachability and uniqueness. Together, Invariants 2-5 ensure that B-tree items are ordered correctly. Similar structural invariants exist for B-tree leaf nodes. The file system metadata in the leaf nodes also has additional structural invariants such as type safety requirements for all data types.

A simple example shows the need to check structural invariants before semantic ones. Figure 3 shows the Btrfs invariant that checks that a directory entry's file type is the same as the type of the inode to which it points (e.g., both are directories or both are files). The `new` predicate returns the file type in a changed directory item. Suppose while creating a directory, the file system creates a directory entry and mistakenly creates two inodes with the same inode number, one of which has the `file` type. The second `query` primitive in Figure 3 (within the `not` clause), which returns the type of the inode, would match the two inode change records. However, the `INODE_TYPE` value that is bound depends on the Datalog engine, so the corruption may not be detected.

Semantic invariants can be made simpler when structural invariants are checked first, because they can depend on structural correctness. The semantic invariants can also be checked independently of each other, because the correctness of the primitives has been established by the structural invariants, rather than by the order in which semantic invariants are checked. Finally, this approach raises structural violations as early as possible, thus providing more accurate debugging information.

```
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))).
```

Fig. 4. Invariant 12: An inode item must exist for every distinct `objectid` in a file system tree

3 Experiences with Invariants

The declarative approach allows the invariants in our runtime checker to match the programmer's intent, enhancing our confidence in the correctness of the implementation. The programmer can focus on pattern matching, without worrying about the correctness of other code such as memory management. We share three examples illustrating the benefits of a declarative approach over an imperative one.

Invariant 12, shown in Figure 4, can be simply stated as "If an inode is removed, ensure 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 Datalog invariant reflects this statement in two rules, each written in 4 lines. The corresponding implementation in C consists of 45 lines, spread across several locations.

Declarative invariants also support rapid prototyping. The Btrfs directory metadata includes Btrfs items that map the file name to an object id (i.e., inode number) and two indexes for fast lookup and iterating over all entries; each inode stores back references to all the directory entries pointing to it. The invariant that checks the consistency of the directory entries, the indexes and the back references is complicated. Its C implementation is spread in 13 locations, 1 for initializing hash tables, 4 for initializing data structures based on the different change records, and 8 for invariant checking based on different hash tables. As our understanding of the invariant evolved, significant amounts of the C code needed to be re-written. We found it simpler to reason about the invariant in Datalog, and then reimplement the equivalent version in C. The final Datalog invariant consists of 45 lines, while just the rewrite of the C invariant added 250 lines.

Fixing bugs in invariants is also easier in Datalog. Our original understanding was that all the data extents in a file must be contiguous, however, we learned that Btrfs files can have discontiguous extents beyond their logical file size. The fix for this invariant required adding a single line of Datalog to check if the offset was less than the size. The corresponding fix took roughly 20 lines (and several hours) to implement in C.

4 Conclusions

We have designed and implemented a declarative online file system checker for Btrfs, a modern file system that supports a rich set of features. The most significant challenge lies in reasoning about the correctness of the checker in the face of arbitrary file system corruption failures. A key takeaway is that the invariants should be expressed as concisely and intuitively as possible, using a declarative language such as Datalog. The rest of the checker, such as the metadata interpretation, should then be designed to support the invariants. This approach makes prototyping invariants and fixing bugs easier, significantly enhancing our confidence in their correctness. We also identified the need to check structural invariants before semantic invariants, so that arbitrary file system structural violations are caught early, and the semantic invariants can depend on the structural integrity of the file system.

References

1. Carreira, J.C.M., Rodrigues, R., Candea, G., Majumdar, R.: Scalable testing of file system checkers. In: Proc. of the 7th EuroSys, pp. 239–252 (2012)
2. Ceri, S., Gottlob, G., Tanca, L.: What you always wanted to know about datalog (and never dared to ask). IEEE Transactions on Knowledge and Data Engineering 1(1), 146–166 (1989)
3. Fryer, D., Sun, K., Mahmood, R., Cheng, T., Benjamin, S., Goel, A., Brown, A.D.: Recon: Verifying file system consistency at runtime. ACM Trans. on Storage 8(4), 15:1–15:29 (2012)
4. Gunawi, H.S., Rajimwale, A., Arpaci-Dusseau, A.C., Arpaci-Dusseau, R.H.: SQCK: A declarative file system checker. In: Proc. of the 8th USENIX OSDI (December 2008)
5. Henson, V., van de Ven, A., Gud, A., Brown, Z.: Chunkfs: Using divide-and-conquer to improve file system reliability and repair. In: Proc. of the 2nd HotDep (2006)
6. Sivathanu, M., Prabhakaran, V., Popovici, F.I., Denehy, T.E., Arpaci-Dusseau, A.C., Arpaci-Dusseau, R.H.: Semantically-smart disk systems. In: Proc. the 2nd FAST, pp. 73–88 (2003)