
# WAFL Iron: Repairing Live Enterprise File Systems

Ram Kesavan, *NetApp, Inc.*; Harendra Kumar, *Composewell Technologies*;
Sushrut Bhowmik, *NetApp, Inc.*

https://www.usenix.org/conference/fast18/presentation/kesavan

This paper is included in the Proceedings of the
16th USENIX Conference on File and Storage Technologies.
February 12–15, 2018 • Oakland, CA, USA

ISBN 978-1-931971-42-3

Open access to the Proceedings of
the 16th USENIX Conference on
File and Storage Technologies
is sponsored by USENIX.

# WAFL Iron: Repairing Live Enterprise File Systems

Ram Kesavan
NetApp, Inc.
ram.kesavan@gmail.com

Harendra Kumar*
Composewell Technologies
harendra.kumar@gmail.com

Sushrut Bhowmik
NetApp, Inc.
sushrut@netapp.com

## Abstract

Consistent and timely access to an arbitrarily damaged file system is an important requirement of enterprise-class systems. Repairing file system inconsistencies is accomplished most simply when file system access is limited to the repair tool. Checking and repairing a file system while it is open for general access present unique challenges. In this paper, we explore these challenges, present our online repair tool for the NetApp® WAFL® file system, and show how it achieves the same results as offline repair even while client access is enabled. We present some implementation details and evaluate its performance. To the best of our knowledge, this publication is the first to describe a fully functional online repair tool.

## 1 Introduction

File system state can be corrupted by hardware failures [9, 38, 41, 45, 35, 53] including misdirected or lost writes and media errors, software bugs [4, 14, 51], and even human errors such as inserting a device in the wrong shelf or slot. Journaling [25, 49, 37, 44, 10], shadow paging [29, 13, 50], and soft updates [22] are all techniques that provide file system crash consistency. Recon [21], incremental checksum, and digest-based transaction audit [34] are well-understood mechanisms to prevent some hardware and software bugs from corrupting the file system. Despite such defenses, corruptions are still an unfortunate reality for file systems, and our customer deployment confirms that reality.

Corruption can affect both user data and metadata. A corrupted user data block affects client access only to it and not to the rest of the file system. However, a corrupted metadata block not only affects access to user data but can also compromise other metadata when client operations are processed. Repair of user data is limited to recovery from backup, whereas metadata can be repaired using consistency properties of the file system. Corruption in metadata is detected either when a metadata block fails some checks as it is read into memory or later when some operation detects a violation of a file system invariant. Some corruption can be fixed on-the-fly using techniques such as RAID or erasure coding. If not, the file system is placed in a restricted mode while it is repaired by an offline repair tool, such as *fsck* [26]. Restricting access serves two purposes: it greatly simplifies the task of repair, and it prevents the inconsistency from causing further damage.

Several approaches have been proposed to speed up or to improve *offline* repair [36, 12, 27, 24] but the time to completion remains proportional to the amount of metadata that must be checked, which in turn is a function of the size of the file system. Enterprises require continuous access to data; any disruption outside of scheduled maintenance windows is highly undesirable. The damaged file system must be made completely available to clients as soon as possible, and repair must therefore not preclude client access. Repair should also not invalidate any data that is accessed or modified by clients after repair is initiated. Additionally, the impact to client performance must be within acceptable limits.

The NetApp® WAFL® file system [29, 20] validates the consistency of its data structures during normal operation. In the rare event of a detected inconsistency that can be neither trivially recovered by RAID nor tolerated, the file system is marked as inconsistent and taken offline. *WAFLIron* [31] (henceforth called *Iron*) repairs the file system even while allowing full access to clients. This paper presents the challenges inherent to this mode of file system repair. It describes the high-level design and implementation of Iron, and evaluates its performance. Most importantly, it explains the theory behind Iron to show that it fixes file system inconsistencies with practically the same assurances as offline repair. Our field experience has shown that Iron is extremely reliable, and meets high performance goals. Although online repair is available for ReFS [30], to the best of our knowledge, there is no prior published work on this topic.

*Research performed while working at NetApp.

# 2 Motivation

A file system can be damaged in several ways [52], but a repair tool is required in only some cases.

## 2.1 When Is Repair Required?

NetApp is a storage and data management company that offers software, systems, and services to manage and store data, including the proprietary NetApp ONTAP® software, which is built on the WAFL file system. Although fsck [40] was originally designed to fix inconsistencies created by an unclean shutdown, WAFL and other file systems use well-understood techniques such as copy-on-write (COW) and journaling to guarantee file system consistency after a crash. WAFL logs recent client operations in a stand-alone nonvolatile journal, and those operations are replayed after a crash to recover them [29]. Because the replay of each operation re-creates all necessary file system state, a corruption of the journal cannot corrupt the persistent file system; at worst, it might result in the loss of logged operations. Furthermore, this loss is limited because WAFL's transaction mechanism ensures the journal typically has client operations only from the past few seconds.

Each block in WAFL is written out to storage media together with a checksum and with some file system specific context that helps further identify the block [8, 13, 48, 47]. If a write is misdirected or lost by the device or if a previously-persisted block is damaged, a subsequent read results in a context or checksum mismatch. The damaged block can be recomputed and fixed by using the underlying RAID [43, 15]. This fixup is done on-the-fly when servicing a read or through a periodic background *scrub* [3]. Other file systems such as ZFS and Btrfs also leverage RAID or data mirroring [1, 5, 46] to provide similar protection. In the rare case of multidevice failures, reconstruction of damaged blocks can become impossible. If such blocks contain metadata that are critical to the functioning of WAFL, the file system is marked as inconsistent and is brought offline, so that it can be repaired.

Despite rigorous testing and prevention mechanisms, rarely occurring software bugs [4, 14, 51] and hardware errors [9, 38, 41, 45, 35, 53] might corrupt a block before its checksum is computed. Such faults cannot be detected by using the persistent checksum or context, and they cannot be repaired by using underlying redundancy [52]. WAFL detects such corruptions when it reads or uses metadata, and if the code path is unable to navigate past it, the file system is marked as inconsistent and is brought offline, so that it can be repaired.

## 2.2 Traditional Offline Repair

Exclusive access to the file system greatly simplifies offline repair, which walks the metadata of the file system exhaustively, checks them for inconsistencies, lists out each inconsistency with a recommended fix, and provides the choice to commit each fix [40]. Such an audit requires accounting metadata to track progress. Repair tools were designed to avoid writing to the physical storage that hosts the file system under repair until the administrator chooses to commit the recommended changes. Thus, the tools keep all their accounting data structures in memory until that time. In general, the amount of metadata increases with file system size, which means an increase in the memory that is required by the tool. This increase is typically offset either by breaking the file system into disjointed chunks of storage [28] or by the tool making multiple passes of the file system, thereby lowering memory requirements. *WAFLCheck*, the first and now obsolete offline tool for repairing the WAFL file system, suffered from similar drawbacks.

## 2.3 Enterprise Needs

Enterprise file systems are usually hundreds of TiB in size, and depending on the features supported their metadata can be both large (several GiB) and complex. Thus, the repair of a 100 TiB file system can take hours or even weeks depending on the I/O capability of the underlying media. Businesses require uninterrupted data availability; an hour-long outage can cost millions in lost revenue. Furthermore, an enterprise storage system typically hosts and exports multiple file systems. Because CPU and memory on such a system are shared resources, the repair of one file system can affect the performance of the others. Therefore, NetApp invested in building online repair instead of making incremental improvements to WAFLCheck. NetApp support staff get involved when a WAFL file system is marked as inconsistent and is taken offline. Under their supervision, the file system is brought online with an option to enable Iron. Clients gain full access to the data even while the persistent file system is checked and repaired. Iron logs its progress and completion, at which point the file system is marked as consistent. The time required for completion depends on several factors, such as file system size and client load on the system. All client ops that were logged in the nonvolatile journal are replayed; as implied earlier, Iron does not repair any corruptions in the journal.

One version of "online" repair [39] argues that orphaned blocks and inodes are the primary outcomes of file system inconsistency. Hence, a snapshot of the file system is taken, the file system is made available, and background

fsck runs on the snapshot to reclaim orphaned blocks and inodes into the active version of the file system. In general, the WAFL file system easily survives orphaned blocks and inodes, and is taken offline only when it encounters an inconsistency that prevents continued operation. Therefore, this approach does not apply.

### 2.4 Considerations With Online Repair

**[1] Unconditional commit:** Iron fixes corruptions as it encounters them so that the file system can continue operations. This means, unlike fsck, the administrator is not given the option to accept or decline repairs. With fsck, if the damage is truly extensive it is likely that the administrator would choose not to commit and restore the entire file system from backup. Online repair does not preclude this option, but all intervening client mutations are lost when the entire file system is restored from backup. There is one scenario in which offline repair is preferable. A customer with poor practices might have no (recent) backup of the file system, and might want to conservatively use offline repair and carefully choose which of the recommended fixes are committed.

**[2] Speed of repair:** An ONTAP system hosts multiple file systems. An aggressive repair process can affect the performance of the clients of all those file systems. A customer might prioritize the completion of Iron because the full repair of the dataset is more important to their business than are the IOPS made available to the applications on that storage system—especially if the backup copies of the corrupted dataset are not sufficiently recent. The ability to control the speed of the repair process is therefore important.

## 3 Metadata and Inconsistencies

This section presents a simplified version of the WAFL file system (persistent) metadata, and the the inconsistencies that can affect them.

### 3.1 Persistent Metadata

WAFL is a UNIX-style file system that uses inodes to represent its files, which are the basic construct for storing both metadata and user data. An inode stores metadata (permissions, timestamps, block count) about the file and its data in a symmetric tree of fixed-size blocks, henceforth called the inode's *blocktree*. Only leaf nodes ($L_0$s) of the blocktree hold the file's data; interior nodes are called *indirect* blocks. An inode that stores file system metadata in its leaves is called a *metafile*. Inodes themselves are stored in the leaves of the *inodefile* metafile, and its blocktree is rooted in the superblock of the file system. All together, they constitute the WAFL file system tree [29].

Each directory is stored in a file that contains a list of entries, where each entry is the name of a file or subdirectory and its corresponding inode number; the root directory is stored in a well-known inode. The reference count (*refcnt*) metafile stores a list of integers where the $i^{th}$ integer tracks the number of references to the $i^{th}$ block of the file system[1]. Multiple references occur because of features such as deduplication that result in block-sharing. The file system stores several counters, some that reside in structures such as inodes, and others that are global.

From the viewpoint of repair, we classify WAFL file system metadata into two broad categories.

**[1] Primary metadata** constitute the blocks of the file system required to read user data. In WAFL, this comprises the superblock, the inodefile blocktree, directories, inodes (for user files) and their blocktrees. WAFL stores copies of some key data structures primarily to protect against storage media failures; corruption due to most software bugs will damage both copies. Importantly, corrupted primary metadata in WAFL cannot be reconstructed by using other metadata. A damaged indirect block in a blocktree cannot be repaired, and therefore, at best, its child sub-tree can be recovered in the *lost+found* folder [40] on completion of the repair process. Similarly, the corresponding inode of a damaged directory entry can be recovered only in lost+found. It should be noted that, to avoid single points of failure, a file system could build redundancy into its primary metadata—each block could encode its location in the file system tree; however, that comes with additional complexity and the run-time cost of maintaining it. To protect against storage media failures, ONTAP uses efficient redundancy techniques: dual-parity RAID [15], triple-parity RAID [2], and remote synchronous mirroring [6].

**[2] Derived metadata** track the usage of resources, such as blocks and inodes, by the file system and can be recomputed by walking the primary metadata. They are typically maintained by the file system software for its efficient functioning, or for enabling specific features, such as file system quotas. The block count of an inode, the refcnt file, and various global counters are all examples of such metadata in WAFL. Damage to derived metadata can usually be repaired based on primary or other derived metadata.

Note that although derived metadata can eventually be reconstituted, they are needed for basic file system op-

[1] In reality, a bitmap tracks the first and the refcnt tracks additional references to blocks [32, 33]. Without loss of generality, the bitmap is subsumed into the refcnt metafile for the purposes of this paper.

eration. For example, the file system must consult and update the refcnt metadata to process new mutations, but that metadata is not fully validated until repair completes. Therefore, the primary complexity of online repair centers around the repair of derived metadata even while the metadata are used by file system operations. The refcnt metafile is the largest and most complex derived metadata in WAFL, and is therefore deliberately used as a running example in this paper.

### 3.2 Inconsistencies

The enablement of Iron does not change how corruption in WAFL is detected; only the action precipitated by such detection. If Iron is not enabled and the software cannot navigate past the inconsistency, the file system is marked as inconsistent and is taken offline. Otherwise, it is repaired. Metadata can be corrupted in one of two ways.

**[1] Manifest corruption:** This form of corruption is detected either when the block is read into memory—checksum or context mismatch—or when some of its contents are used for the first time—well-known signatures appear wrong or some data structures are outside acceptable bounds. Such a block needs repair only if it cannot be recomputed by the underlying RAID, which can happen either because multiple hardware elements have failed or the block was corrupted before the associated parity was computed.

**[2] Latent corruption:** File system invariants typically define relationships across different metadata. A latent corruption violates a relationship even while each participant block is devoid of manifest corruption. The relationship might involve primary metadata only. For example, a directory $L_0$ and an inodefile $L_0$ might each be independently reliable, but the former maps a dir entry to an inode that is marked as free in the latter. Sec. 4.5 presents examples of latent corruption across primary and/or derived metadata. Latent corruption is detected only when a metadata consistency invariant in the code trips up. Before its detection, it can create further inconsistencies if used by the file system; Sec. 4.5 has more details.

Both forms of corruption can be caused by bugs in the file system logic or memory scribbles. Device failures and media errors typically result in manifest corruption only; the block will appear to be unreliable.

## 4 Basics of WAFL Iron

Much as offline repair would, Iron walks all primary metadata, checks consistency with other metadata, and makes repairs where necessary. However, full client access is enabled early on. After a file's blocktree has been completely checked, all derived metadata for the file (such as its block count) is verified. As mentioned earlier, WAFL stores all user data and metadata (both primary and derived) in files. Therefore, after all files in a file system have been checked, all derived metadata is verified and the file system is marked as consistent.

The first version of Iron (circa 2003) focused on mitigating the main drawbacks of WAFLCheck: (1) scaling of the metadata needed for checking the file system, and (2) allowing early file system access while still providing the same assurances as WAFLCheck, aside from the unconditional commit highlighted in Sec. 2.4. This paper focuses primarily on proving parity in functionality with offline repair, and therefore does not do justice to the details of implementation. This section presents the rules for addressing the complications from allowing client access, and presents the main design.

### 4.1 Rules for Iron

**Rule #1, Interposition:** *Every block is processed by Iron before the rest of the file system software can use it.* This rule lets Iron make repairs early, which prevents the rest of the file system software from making decisions based on inconsistencies. This rule necessitates a filter in the read-from-storage code path so that all blocks are examined by Iron first.

**Rule #2, Irreversibility:** *After Iron starts, any state that is exposed to the client cannot be revoked by any future repair done by Iron.* Practicality requires that the data served to a client, as well as the results of any client mutation, not be revoked by subsequent repair[2]. To satisfy this rule, a file system op (client or internal) waits when loading a block until Iron validates any metadata required to ensure that block's continued survival through the completion of Iron. This approach has two obvious implications: (1) The latency of an op can be significantly affected; later sections explore this impact. (2) Iron needs a definition of the metadata required to ensure a block's survival; we look at that implication next.

Let the relationship $b \rightarrow b_i$ define a WAFL consistency invariant where metadata block $b_i$ must exist and contain the "right" information to ensure that block $b$ belongs to the file system; $b$ can be user data or metadata. In other words, Iron must either move $b$ to lost+found, or create or modify $b_i$ to preserve the relationship. This relationship is obviously transitive, i.e., if $b \rightarrow b_i$ and $b_i \rightarrow b_j$, then $b \rightarrow b_j$. We define the *essential set*, $\Psi(b)$, of all metadata blocks, such that $b_i \in \Psi(b) \implies b \rightarrow b_i$. When an op loads $b$, Iron uses the filter described in Rule #1

[2] In fact, this invariant is extended to include state exposed to all internal file system ops. It simplifies the interaction of Iron with several file system modules.

to load, check, and potentially repair all metadata blocks in $\Psi(b)$ before allowing the op to proceed. Thereafter (with help from Rule #3), Iron does not change anything in $\Psi(b)$ that revokes that state of $b$, thereby preserving Rule #2[3]. This is true even if $b$ is modified by the op.

Let's look a little closer at the essential set. All ancestor blocks of $b$ in the file system tree (Sec. 3.1) trivially belong to $\Psi(b)$. This includes any ancestor indirect blocks of $b$ within its inode, the corresponding inodefile $L_0$, the ancestor indirect blocks of the inodefile, including the superblock. Iron is invoked through the *mount* command, and so no blocks of the file system are in memory at the start of Iron. Thus, Rule #2 is trivially satisfied for an ancestor block because it is always loaded before its child. *In fact, all primary metadata in $\Psi(b)$ can be exhaustively shown to satisfy this rule*; for brevity, we do not list them here. However, as an example, if $b$ belongs to a user file, the directory block $L_0$ with the corresponding directory entry also belongs in $\Psi(b)$, and it is loaded and accessed before $b$. Derived metadata associated with $b$, such as the refcnt $L_0$ with its refcnt entry, also must be loaded and checked for consistency. Sec. 4.3 explores an important complication with the essential set.

**Rule #3, Convergence:** *As Iron incrementally checks and repairs metadata, it monotonically expands the portion of the file system metadata that is self-consistent.* Iron ensures that file system metadata is never checked more than once, and therefore the extra cost of checking the essential set when loading any block diminishes with the progression of Iron. For example, when a second child of block $b$ is loaded, Iron does not repeat the checking of all primary metadata performed on the first load of a child of $b$. Rule #2 ensures that all metadata associated with new mutations to the file system have also been checked and are included in the portion of the file system metadata that is considered self-consistent. Thus, convergence is guaranteed. This rule implies that Iron maintains data structures to track its progress, which leads us to the next rule.

**Rule #4, Scalability:** *Data structures that Iron needs to track progress must scale with file system size without requiring additional system memory.* The previous two rules make it clear that Iron makes a single pass over the file system metadata. Iron scales its data structures with file system size by storing them in files that are paged in and out of the WAFL buffer cache [19] much like any other metafile; they are called *Iron status files*. Much like any other file in the file system, all previous rules apply to the creation, consultation, and mutation of the status files. In other words, the ever expanding portion of self-consistent metadata (of Rule #3) includes all status files.

[3] A new corruption introduced to a block after it has been checked may violate this statement; Sec. 5 discusses that topic in more detail.

## 4.2 Iron Status Files

Status files are created and used by Iron for each file system that it repairs. All status files are deleted upon completion of Iron. Status files can be broadly classified into *progress indicator metafiles* and *derived shadow metafiles*.

**Progress indicator metafiles:** This class of status files tracks the progress of Iron and avoids repeated work, both of which are necessary for ensuring Rule #3. One example is the *checked bitmap* status file, which is a vector of bits where the $i^{th}$ bit indicates that the $i^{th}$ block of the file system has been checked, and repaired if necessary. Because a metadata block can be scavenged from the buffer cache and subsequently re-read from storage, this bitmap ensures a given block is processed exactly once.

**Derived shadow metafiles:** Iron computes shadow versions of some derived metadata as it walks the file system. On completion, Iron compares the shadow version with the original version, repairs both manifest and latent corruption in that derived metadata, and discovers any orphaned resources that are tracked by that metadata. One example is the *claimed refcnt* status file, a list of shadow integers where the $i^{th}$ integer tracks the references to block $b_i$ that Iron encounters. On completion, Iron replaces each refcnt integer with its claimed refcnt counterpart; a count that changes to zero represents an orphaned block. Thus, Iron can ignore the corresponding refcnt block when it processes the essential set for a block; the refcnt $L_0$ in $\Psi(b_i)$ is replaced by the corresponding claimed refcnt $L_0$. Until Iron completes, the WAFL write allocator [18] consults both refcnt integers to decide if a block is free, and freeing a block requires decrementing both refcnt integers. The claimed refcnt integer can never underflow because Rule #2 guarantees that Iron claims a block before freeing it. Sec. 4.4.2 and Sec. 4.5 discuss the underflow and overflow of a damaged refcnt integer. Iron uses other shadow derived metadata in a similar fashion, but all of them are smaller in size and in complexity than the claimed refcnt file, and are therefore not discussed here.

## 4.3 Recursivity Within the Essential Set

Let's say that $b_i \in \Psi(b)$. When Iron loads $b_i$ on behalf of $b$, Rule #2 forces a recursive load of all metadata blocks from $\Psi(b_i)$. Although this recursivity implies indefinitely long response times for client ops, we will show why that is not true in reality; let us look at each type of metadata in $\Psi(b)$.

**[1] Primary metadata:** For every primary metadata block $b_i \in \Psi(b)$, all primary metadata blocks in $\Psi(b_i)$

also belong in $\Psi(b)$ (and will therefore have already been loaded and checked earlier). For example, if $b_i$ is an ancestor of $b$, then all ancestors of $b_i$ are also ancestors of $b$. A similar argument can be made for directory blocks of $\Psi(b)$ when walking a pathname.

**[2] Derived metadata:** Depending on the specific derived metadata, let's say $M$, Iron breaks this recursion in one of two ways: (1) It loads and checks all of $M$ during mount and before client access is allowed, and Rule #3 ensures $M$ is not checked again. This approach is taken for smaller metadata. (2) Iron does not load or consult $M$ when it processes the essential set. Instead, it maintains and updates a derived shadow metafile that corresponds to $M$; $M$ is checked only when it is loaded for other file system activity. We illustrate this approach by using the refcnt file example.

Let $b_j$ be the refcnt $L_0$ block that contains the $i^{th}$ refcnt integer; clearly, $b_j \in \Psi(b_i)$. Then, the $L_0$ block of the refcnt file with the $j^{th}$ integer, let's say $b_k$, belongs in $\Psi(b_j)$. This means that Iron would need to check $b_j$, $b_k$, and so on, possibly checking the entire refcnt file to load $b_i$, which would result in unpredictable operational latencies. As described in Sec. 4.2, instead of loading and checking $b_j$, Iron increments the $i^{th}$ claimed refcnt integer[4]. This breaks the recursion, and $b_j$ is checked later.

Thus, the number of yet-to-be-checked blocks in $\Psi(b)$ is quite small in practice. The analysis in this section can be used to prove freedom from deadlocks even when essential sets of concurrent client ops happen to overlap; we cannot present a formal proof due to lack of space.

## 4.4 Repairing Manifest Corruption

Depending on the type of metadata, Iron chooses from two techniques—tombstoning and quarantining—to handle manifest corruption.

### 4.4.1 Tombstoning Primary Metadata

Sec. 3.1 explains why damaged primary metadata in WAFL cannot be repaired, which is true with offline repair as well. Before making the file system available to clients, Iron checks the higher part of the file system tree hierarchy, such as the superblock, the inodefile blocktree, and the root directory. Iron aborts if corruption is detected there; the file system is not considered repairable. The customer can then choose between restoration from a recent snapshot (stored locally or remotely) or manual stitch-up of the file system by skilled technicians with direct access to the storage media.

Data structures in the lower part of the file system tree hierarchy with manifest corruptions are *tombstoned* by setting them to a corresponding zero value or to a special value that WAFL code paths recognize. If the whole block is unreliable, its entire content is tombstoned. A client read op that encounters it—say, a tombstoned child pointer in an indirect block of a user file—returns an appropriate error. A subsequent mutation can change the tombstone to a legal value. For example, a client op that writes to an offset in that file corresponding to the range covered by that child pointer replaces the tombstone. Tombstoning a child pointer results in an orphaned sub-tree, which is eventually recovered and placed into lost+found by Iron. Much as in traditional repair, the administrator can choose to stitch it back into the file system, but in concert with the application accessing it. If the administrator chooses otherwise, the data structures remain tombstoned until they are overwritten or deleted by new mutations. Given Rule #2, and that WAFL eschews redundancy within primary metadata (to avoid the associated performance overhead), we conclude:

**Conclusion 1.** *The repair of manifest corruption in primary metadata of WAFL by offline repair is no better than repair by Iron.*[5]

### 4.4.2 Quarantining Derived Metadata

If Iron encounters manifest corruption in a derived data structure, it *quarantines* the data structure by setting it to a corresponding well-known and conservative value that protects the resource that it tracks. The well-known value never overflows or underflows, which allows WAFL code paths to navigate past invariants that use it. If an entire block of derived metadata is deemed to be unreliable, then every data structure in it is quarantined. On completion, all quarantined structures are set to their corresponding values computed by Iron. Thus, given that all damage to derived metadata is quarantined before consulted by file system operations, and that the quarantined value conservatively protects the resource that it tracks, we conclude:

**Conclusion 2.** *Iron guarantees that mutations to the file system can never cause new or additional corruption due to existing manifest corruption in derived metadata.*

[4] Because WAFL is a COW file system, a claimed refcnt increment results in that claimed refcnt $L_0$ getting written to a new location, let's say block $b_n$, which in turn requires the $n^{th}$ claimed refcnt integer to be incremented, and so on. This is a different type of recursion that impacts most allocation bitmaps in WAFL. The WAFL block allocator finds free blocks colocated in the block number space, and therefore this recursion converges very quickly. Previous publications [32, 33] detail how recursion during decrements (due to frees) converge.

[5] Offline repair in a file system with redundancy in primary metadata could stitch an orphaned subtree back into its correct location, repair the damaged child pointer, and avoid data loss. Online repair for such a file system would need to suspend client access to the tombstoned structure until the orphaned subtree is found.

The following example helps illustrate this conclusion. Let's say that Iron determines a refcnt file $L_0$ to be unreliable when it is first loaded, and let's say that $L_0$ stores refcnt integers for blocks $b_i$ through $b_{(i+n-1)}$ of the file system. Iron then sets each of those refcnt integers in the $L_0$ to the quarantined value, thereby ensuring all potential references to blocks $b_i$ to $b_{i+n-1}$ are conservatively protected. In other words, the WAFL write allocator considers them unavailable for new mutations; note that WAFL uses COW, and no block is ever written in place. On completion, Iron resets each quarantined integer to its claimed refcnt counterpart and returns any unused blocks back to the free space in the file system.

Given sufficient damage to a specific derived metadata, Iron might decide that the file system has run out of the resource tracked by that derived metadata. Sec. 4.6 describes how this case is handled.

## 4.5 Repairing Latent Corruption

As Sec. 3.2 explained, a latent corruption is a violation of some specific file system invariant, and is detected when a code-path trips on it. We reason about latent corruption by discussing the different permutations of metadata that are involved in the violated invariant.

**Primary metadata only:** Let's say that all the blocks involved in the violated invariant are of primary metadata. In WAFL, all relationships between primary metadata are captured in $\Psi(b)$ for a given primary metadata block $b$. In the example from Sec. 3.2, a client op can access the damaged inode only after accessing that directory. Therefore, a violation of such an invariant is conveniently detected as and when each primary metadata block is loaded, which leads to:

**Conclusion 3.** *The offline repair of a latent corruption that violates an invariant across primary metadata of WAFL can be no better than repair by Iron.*

**Derived and derived/primary metadata:** Derived metadata typically track persistent resources consumed by the file system, such as inodes and blocks. We explore this problem for blocks, and then extend the results to other derived metadata. The refcnt integer $r_i$ tracks the consumption of the $i^{th}$ block by the file system. Thus, $r_i$ encodes a relationship with block $b_i$; $b_i$ may be user data or metadata (primary or derived).

Let's say that a latent corruption had made $r_i$ incorrect. As described next, several mutations might be persisted to the file system before this corruption is eventually detected. WAFL relies exclusively on the child pointer in $b_i$'s parent block when it frees $b_i$ (say due to a file truncation) and decrements $r_i$. On the other hand, the WAFL write allocator relies exclusively on $r_i$ to check if block $b_i$ is free. Thus, code paths that allocate and free blocks depend exclusively on derived and primary metadata, respectively, and expect them to be consistent. This split-brain behaviour can result in the morphing of this latent corruption even before it is detected. The corrupted $r_i$ might be (A) higher or (B) lower than the true value. If (A), WAFL might eventually leak $b_i$ when all references to it have been dropped and $r_i$ remains non-zero. The leaked block will be detected by a subsequent run of Iron. Two possibilities exist with (B): In case (B1): an eventual decrement causes $r_i$ to underflow, which is detected as a violation. In case (B2), the WAFL write allocator might incorrectly assign $b_i$ to a new write when $r_i$ becomes zero, and the original contents of $b_i$ are lost to the file system. If $b_i$ originally contained metadata, any access through an older reference would detect manifest corruption (signature and context mismatch)[6]. In this case as well as case (B1), the file system is marked as inconsistent, is taken offline, and repair is invoked.

Conclusions 1 and 2 show that Iron handles the manifest corruption of case (B2) no worse than offline repair does. In case (B1), if the decrement has been triggered by a client op, Iron increments the $i^{th}$ claimed refcnt integer almost immediately after mount because WAFL replays all client ops after any disruption. Thus, the subsequent decrement finds a zero value $r_i$ but a nonzero claimed refcnt integer. Iron prevents any file system activity from underflowing a refcnt integer as long as it can decrement the corresponding claimed refcnt integer. If the decrement has not been triggered by a client op, $b_i$ remains unclaimed until Iron gets to the file that refers to it. During this window, both $r_i$ and its claimed refcnt counterpart are zero, and the WAFL write allocator may use $b_i$ for a new block; that scenario is subsumed by case (B2).

Although offline repair averts the previously mentioned window because no new blocks are being written to the file system, it should be noted that case (B2) may also occur during runtime before the latent corruption is detected and recovery is initiated. Thus, in practice, the use of Iron does not introduce significant additional risk beyond what existed earlier.

This entire argument can be replicated for any resource that is similarly tracked by derived metadata and tracked separately by Iron shadow metadata. Latent corruption in derived metadata that is checked before client access is allowed to the file system can be found and repaired early on. This leads to:

**Conclusion 4.** *The repair by Iron of latent corruption*

[6] If $b_i$ contained user data, it is lost. Independent of whether $b_i$ contained user data or metadata, its re-allocation does not create a security risk because access via the original parent of $b_i$ will fail the context check, and return an error instead of the new content stored in $b_i$.

*that violates a relationship across derived and/or primary metadata is no worse than that by offline repair.*

**Miscellaneous metadata:** WAFL maintains extensive auxiliary metadata that are computed using information from other derived metadata. Such auxiliary metadata typically are used to enable specific features or better file system performance. The WAFL file system can typically survive corruption to such metadata, but when Iron is invoked, it can repair these structures while the file system runs with decreased performance or with those specific features disabled. We have found that customers are willing to tolerate such temporal deficiencies for continued data availability.

### 4.6 Running Out of a Resource

The end of Sec. 4.4.2 discussed a problem scenario that relates to the quarantining of a sufficiently large amount of derived metadata. It might result in premature exhaustion of the file system resource tracked by that metadata, before Iron completes. For example, sufficient quarantining of the refcnt metadata might cause the file system to run out of space. The WAFL block allocator is designed to offline the file system gracefully in this case. Because the file system is still marked as corrupt (Iron never completed), the file system is now repaired by using Iron in offline mode (more information in Sec. 6). It is important to note that no mutations are lost in this scenario. To the best of our knowledge, this problem has not been encountered in the field thus far.

### 4.7 Three Phases of Iron

**[1] Mount:** ONTAP mounts the inconsistent WAFL file system when it is brought online with the Iron option. To allow faster access to clients, Iron limits the amount of metadata that is checked at mount. As described in Sec. 4.4.1, key metadata in the upper part of the file system tree hierarchy are checked. Based on the physical storage devices, various limits on file system resources are computed (such as number of blocks) and are used as ceilings for various global counters. Auxiliary metadata, such as hints for speeding up the search for free space, are checked. Based on those hints, the block allocator is primed by prefetching refcnt file blocks. Detection of manifest corruption results in the quarantining of refcnt integers and further loading of refcnt blocks until sufficient free space has been confirmed. The Iron status files are created and updated to reflect the checking performed thus far. As described in Sec. 4.4.1, Iron aborts if mount-time checks do not complete. Otherwise, client access is allowed.

**[2] File system scan:** Metadata are checked on-demand (based on client access) and through background scans, each of which selects an inode and walks its blocktree. Leaf nodes of metafiles are also checked. Progress indicator status files ensure that each block is checked once.

As client mutations are processed, the WAFL write allocator prefetches more blocks of the refcnt file to find more free space. The background walk of crucial derived metadata, such as the refcnt file, typically completes early, and all quarantining that affects free space accounting is in place. Recall that Iron status files track both validated and new data written by clients, so the validated portion of the file system continually increases. Due to space constraints, we do not describe our implementation in more detail.

**[3] Completion:** After the entire file system has been walked, the derived shadow metadata—which, now accurately represent all resource consumption—are swapped with their counterparts; quarantined structures are removed. Status files are deleted subsequently and the file system is marked as consistent.

## 5 Analysis and Some History

This section describes some deficiencies in the initial version of Iron and some improvements that were made over the years.

**[1] New corruption:** Regular file system access is allowed during online repair, which means that if the file system was originally corrupted by a software bug, it could reoccur during the repair process. Therefore, it is difficult for any efficient online repair tool to *guarantee* file system consistency on completion. Earlier versions of Iron also have this "flaw". Although it has never been observed, it is possible for the Iron status metadata to be corrupted by such a bug, which might have a bigger impact on the guarantees that Iron provides. Sec. 5.1 addresses this issue.

**[2] Mount-time performance:** The first version of Iron (circa 2003) checked the indirect blocks of the blocktrees of all derived metadata during mount. However, this meant longer mount times, and therefore a longer wait for restoration of client access to the file system. The mount phase was subsequently thinned, and larger derived metadata (that scale linearly with file system size) are now checked asynchronously post mount. Quarantining occurs at any level of the indirect blocktree of a derived metafile. Latent corruption in derived metadata is addressed by the hardening techniques of Sec. 5.1.

**[3] Performance of client ops:** In its original version, Iron checked the entire indirect blocktree of a given in-

ode before a client (or internal) op could access any block of that file. Thus, the original definition of $\Psi(b)$ included all indirect blocks in the blocktree of any inode in the ancestory hierarchy of $b$, thereby ensuring Rule #2 when exposing file attribute state to clients, such as size or block count. In its early days, WAFL was primarily optimized for homedir-style engineering workloads with many small to medium-sized files, and so the time to first-access of a file was not too significant. With the deployment of critical database and virtualization workloads on WAFL, GiB- and TiB-sized files became increasingly common. Irreversibility of attributes such as block count for files that host databases or VM disks is not a strict requirement. Thereafter, the definition of $\Psi(b)$ was refined (to that in Sec. 4.1) to exclude non-ancestor blocks of $b$ in the file system tree, but without any risk to the repair process. Sec. 5.2 describes additional performance improvement.

### 5.1 WAFL Metadata Integrity Protection

Two techniques, incremental checksums and digest-based auditing [34], were introduced circa 2012 to protect much of the WAFL file system metadata from memory scribbles and logic bugs [23]. Sec. 7.5 of [34] shows the resultant drop in corruption incidents in customer systems, thereby dramatically reducing the need for Iron. In addition, there are two crucial implications for Iron: (1) Iron status files are now protected by these techniques, which squarely addresses the first deficiency described in Sec. 5; (2) it removes one key reason for the on-demand update of shadow derived metadata while processing client ops; more in the next section.

### 5.2 Lazy Block Claiming (LBC)

After mount-time outages were reduced, the one important remaining problem with Iron was the impact to client ops. As explained earlier, Iron must process the corresponding essential set before a file system op can be given access to a block. Because recursivity in derived metadata has been solved, the cost of processing any block in $\Psi$ is dominated by: (1) loading and consulting/updating checked bitmap blocks, and (2) loading and updating claimed refcnt blocks. These steps require additional CPU cycles and random I/Os to storage; the randomness is also a function of client access patterns. *Lazy Block Claiming (LBC)* was introduced to address this overhead.

The on-demand update of claimed refcnt metadata (or any derived metadata) guarantees that resources accessed by a client op are henceforth protected, thereby preserving Rule #2. Thus, when a client op accesses block $b_i$, the on-demand increment of the $i^{th}$ claimed refcnt integer protects $b_i$ from being re-allocated due to a corrupted refcnt integer. Latent corruption in the refcnt metadata is eliminated by the integrity techniques of Sec. 5.1. And, given that manifest corruption results in quarantining, Rule #2 is now preserved even without on-demand updates of derived metadata, such as claimed refcnts. Thus, LBC avoids the afore-mentioned costs and enables improved client performance, independent of file size.

This means, after the file system is mounted, the claiming of the references to each block in the file system occurs only through the background scans. More importantly, knobs are provided that control the speed of those background scans. Thus, the customer can choose between reducing the impact of Iron scans to client workloads and how quickly Iron completes processing the entire file system.

### 5.3 Additional Enhancements to Iron

This section outlines in-progress and productized improvements; a future paper will cover them. First, concurrent access of Iron status files within the WAFL parallelism model [17] is required to truly minimize the impact of Iron on client performance. Second, the customer still experiences outage from the time the WAFL aggregate is offlined until Iron is started. Incremental Autoheal Iron [11] builds on the principles described in Sec. 4.1 to provide true zero disruption. When Autoheal detects a corruption at runtime, it tombstones or quarantines, simulates a minimal emptying of the buffer cache, and optionally kicks off a background scan to check and repair a defined set of metadata based on the corruption. Depending on the results of the scan, a larger subset of the metadata might be scanned next. Such incremental granular repair is ideal because, as mentioned earlier, only a few metadata blocks are typically damaged in WAFL. ONTAP 9.1 introduced FlexGroup technology [7, 42]—it allows a file system to span multiple physical nodes in a cluster. Offlining an entire FlexGroup on the detection of a corruption is obviously not an option; ONTAP 9.1 includes an early version of Autoheal.

## 6 Topics in Practice

This section presents some selected topics that relate to the implementation of Iron.

**Location of status files:** The implementation allows for Iron status files to be stored within the file system being repaired or in a different WAFL file system; both choices are equally safe. By storing it remotely, the customer can isolate to a separate set of storage devices the extra I/Os

required to read and update status files.

**Offline mode for Iron:** In this mode, Iron provides WAFLCheck-like behaviour. Thus, client access is disallowed and Iron cannot write to the physical storage of the file system. Iron stores its status files remotely. Corruptions that Iron fixes are appended as a sequence of tuples to a log file stored in a different file system. Each tuple includes the contents of the file system block and its physical location that the WAFL write allocator chose. Eventually, the administrator is given the choice to commit all or none of the repair. If the former choice is made, the log file is "replayed" and the content of the log is written out at the appropriate locations in the file system.

**Aggregates and FlexVols:** ONTAP hosts and exports hundreds of FlexVol® volumes on a shared pool of of physical storage called an *aggregate* [20]. Each FlexVol and aggregate is a WAFL file system. When corruption is detected, the aggregate file system is tagged as corrupt, is offlined, and is eventually remounted with Iron. Hundreds of applications hosted on the FlexVols gain early access to their data. Our field data show that typically a handful of blocks from a few FlexVols are damaged. At worst, a few of the applications might be halted if they access tombstoned structures; they can be restarted after that data is recovered from backup or from lost+found. However, other applications see minimal disruption.

**Field data:** In an analysis of corruptions seen across ~250,000 customer systems during a recent six-month period, approximately a third were attributed to software bugs, another third to media errors (while RAID was running in degraded mode), and the rest to a mix of manual configuration error or unknown reasons. In each case, the total number of corrupted metadata blocks was less than 10. In very rare cases when hundreds of blocks are damaged (due to silent hardware failures), customers typically restore from backup/snapshots or use offline repair.

# 7 Evaluation

In this section, we present the performance characteristics of Iron. As explained earlier, a WAFL file system being repaired has at worst tens of damaged metadata blocks. The extra cost of repairing those blocks is undetectable compared with the cost of checking the entire file system.. Therefore, no actual corruption is required in the datasets of the following experiments. We discuss client outage times, the overhead of running Iron, and how it interferes with a real-world workload. Unless otherwise mentioned, all experiments were conducted on a lower-end system with 16 Intel Sandy Bridge cores and 64 GiB DRAM to accentuate the impact of Iron.

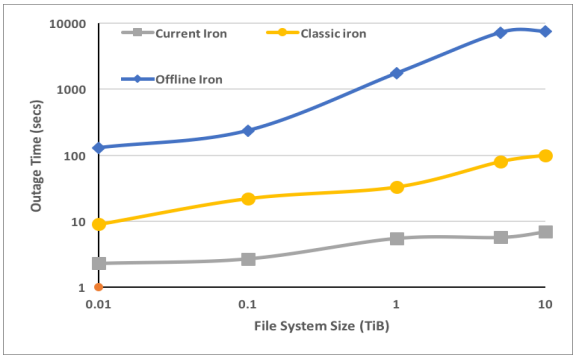


**Figure 1:** Client outage time in seconds on a logarithmic scale with increasing file system size.

## 7.1 Memory and Storage Overhead

Iron metafiles are paged in and out of the buffer cache like any other file in the file system. The storage space that the metafiles consume is approximately 32 MiB (checked bitmap) and 0.5 GiB (claimed refcnt) per TiB of file system size, and is 4 MiB (link count) per million inodes in the file system. Together with other metafiles (not presented in this paper), it adds up to around 0.05% of the file system size. The in-memory data structures that Iron uses add up to a few KiB of extra memory. This extra requirement in memory and storage is negligible on all configurations of ONTAP.

## 7.2 Outage Time with Iron

Fig. 1 plots on a logarithmic scale client outage time with increasing file system size (used space) on the lower-end system. Outage was measured from when the command to online the WAFL file system (with the Iron option) was issued to when the file system was first exposed to clients. Thus, this experiment measured the time taken to check metadata during mount. Numerous drives were used to avoid I/O bottlenecks, and no other load was applied on the system. The experiment was run thrice: with Iron in offline mode, with classic Iron (mentioned in Sec. 5) in which derived metadata were checked during mount, and with the current version of Iron in which indirect blocks of all large derived metafiles were checked post-mount. Iron employs the same level of parallelism for checking metadata in each mode, thereby ensuring fairness.

Iron in offline mode performs similarly to the now-obsolete WAFLCheck tool, and outage time is obviously the time to complete Iron. Offline repair is clearly impractical for enterprises—a 10 TiB file system takes 2+ hours to repair. Outage times with the current version of Iron are an order of magnitude less than the times reported by classic Iron; 6.9s and 100s, respectively, for 10

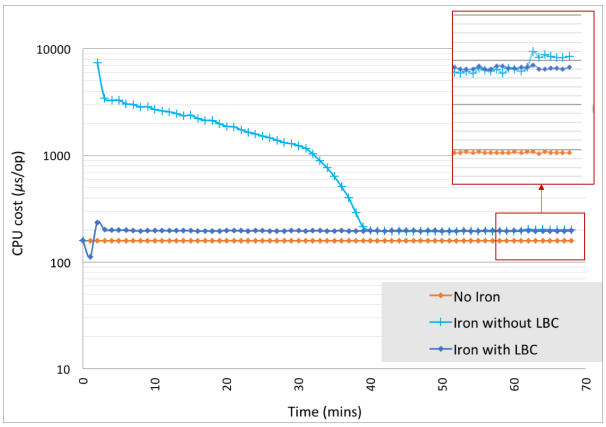


**Figure 2:** CPU cost ($\mu$s/op) of random reads

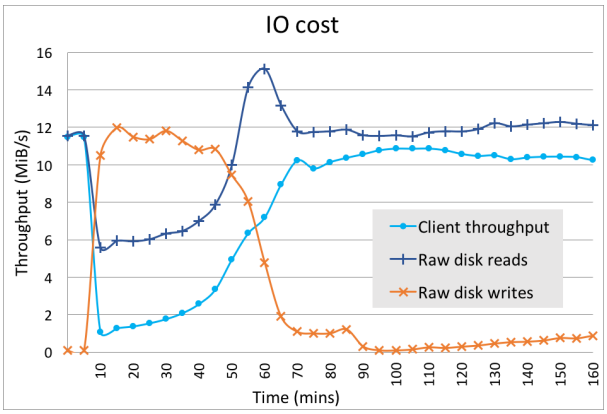


**Figure 3:** Throughput in MiB/s when random reads are directed to a repairing (without LBC) file system

TiB. Outage times with the current Iron tool are almost independent of the file system size; mount without Iron (not shown) takes around 1s for any file system size.

## 7.3 Performance Overhead of Iron

Next, we present the overhead associated with Iron in terms of CPU cycles and storage I/Os—the two important metrics that IT architects use for sizing storage systems and applications. LBC was instrumental in making Iron's overhead more predictable and therefore practical. The worst case overhead on low-end systems is 25%. Note that most of our systems do not experience that level of overhead. We used a worst-case random read workload—it reduced the overlap between the essential set for a given client read with that of a previous one—thereby maximizing the amount of on-demand work that Iron performs. A read-only workload was used to keep the analysis simple; there was no material change in the results when client writes were added in.

### 7.3.1 Cost in CPU Cycles

Several NFS clients directed a random read workload of 25 MiB/s to a 18 TiB dataset that comprised 450 files, each of size 40 GiB on the lower-end system. To avoid perturbation from I/O bottlenecks, the storage was all-SSD. The experiment was run without Iron, with Iron and LBC disabled, and with current Iron (LBC enabled). To make the comparison fair, the background Iron scan was disabled for the first one hour of the run without LBC—so all Iron work was triggered only on-demand by the client reads.

CPU cost is computed by adding up all the cycles that the file system code paths (including Iron) use and dividing that value by the number of client operations serviced for a given time interval. Fig. 2 compares that cost (measured as $\mu$s/op) on a logarithmic scale. The baseline read op cost averages 160 $\mu$s' worth of CPU cycles; it includes the processing for misses in the buffer cache.

In the Iron run without LBC, about 4 to 8 primary metadata blocks are checked for each client read in the early portion of the experiment. Each checked indirect block may have up to 256 children that are unlikely to be colocated, and requires random updates to the claimed refcnt metafile. In the early portion of the experiment, this costs an extra 7.8 ms' worth of cycles. Over time, a client read finds that more of its essential set is already checked, and the Iron overhead drops. Almost all metadata has been checked by the 40-min mark, and the cost flattens to about 190 $\mu$s; the extra 30 $\mu$s is the unavoidable cost of consulting the checked bitmap. The small bump at the 60-min mark coincides with the start of the background scan (which completes soon after because on-demand checking has already done the job). With LBC, the overhead of random I/Os to the claimed refcnt metafile is moved from client ops to the slow-running background scan. Thus, after the initial spike to check important metadata, such as inodes and directory entries, the cost flattens to 200 $\mu$s; the still-running background scan costs the extra 10 $\mu$s.

### 7.3.2 Cost in I/O Bandwidth

The previous experiment was modified to use SATA hard drives (so storage I/Os were no longer "cheap"), and the client load was lowered to 11.5 MiB/s which is commensurate with the I/O capability of that storage media.

Fig. 3 shows the client throughput and the raw reads and writes to the drives in MiB/s. To simplify the analysis, Iron without LBC was started at the 10-min mark of the experiment, and the background walk of the file system was disabled (until the 110-min mark). The client

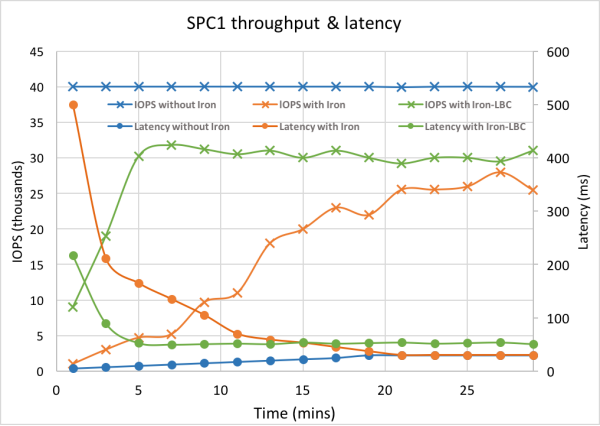


**Figure 4:** Impact of Iron with and without LBC on latency (right-side y-axis) and on throughput (left-side y-axis) at a steady applied load of 40k SPC-1 IOPS.

throughput and raw reads are identical until the 10-min mark. The remount (to start Iron) at the 10-min mark empties the buffer cache. This is followed by a spike in writes to storage as Iron status metafiles undergo frequent updates. The remaining disk bandwidth is divided between reading the user inode leaf nodes and Iron status files. Thus, for the first 40 mins of Iron only about 10% to 18% of the disk bandwidth is used for reading the leaf nodes of user files. By 90 mins the essential set for most client reads has already been checked, but checked bitmap consultations require a continual and fixed amount of read bandwidth. The impact due to background scans is seen after the 110-min mark. With LBC enabled (not shown here), the drop in client throughput is mostly a function of the rate at which Iron background scans run, which is typically set to a low default.

### 7.4 Impact on Clients

Several clients were used to apply a steady load of reads/writes to model the query/update ops of a database/OLTP application; this was based on the Storage Performance Council Benchmark-1 (SPC-1) [16]. Iron was started soon after. LBC was designed primarily for helping database/OLTP applications, which are quite latency-sensitive to any additional CPU or I/O overhead. To accentuate the impact of Iron, the experiment was configured on a low-end system with 8 AMD Opteron cores, 32 GiB DRAM, and SATA HDDs. The background Iron scans were allowed to run in this experiment.

Fig. 4 shows that without Iron the clients achieve the entire applied throughput of 40k IOPS with average latencies under 30 ms. With Iron, both metrics improve as a larger portion of the metadata is checked. With LBC, these metrics are 2 to 5 times better early on, and they soon converge to a steady 75% of the applied throughput. In theory, WAFL parallelism should be unaffected by Iron because checking (both on-demand and by way of scan) can run concurrently with other client operations in the WAFL MP model [17]. However, one last project to achieve full parallelism is still in progress, after which we expect the impact of Iron (with LBC) to be much smaller. Because the background scans limit some of the potential parallelism, client operations in the run with LBC show poorer latencies past the half-way point. In the run without LBC, the on-demand work has finished much of the scan's job by that point. The slow increase in latency in the baseline run (until 20 min) is due to the initial aging of the file system.

Many of our customer systems use SSDs and are not low-end, and therefore see less impact with Iron. As mentioned earlier, Iron is run under the supervision of NetApp support, and customers are aware that an inconsistent file system is being recovered. We find that they greatly appreciate the continued uptime for their applications, even with reduced performance. As improvements to Iron have reduced the impact to clients over the years, we also find that customers have become less concerned with Iron completion times as long as progress indicators provide a time estimate. But, as an example, Iron completion time with a default scan speed on the lower-end system is 0.48 hour per TiB dataset resident on SSDs and 1 hour per TiB dataset resident on hard drives, even while sustaining a random-read client workload of 470 MiB/s and 255 MiB/s, respectively. Impact on home-directory style workloads is not presented due to lack of space. The impact is typically less than that on database/OLTP workloads because the files themselves are small, and each indirect block has few children. However, datasets with very large directories (millions of entries) are affected to a greater extent; future work is planned to make the checking of directories truly asynchronous to client operations.

## 8 Conclusion

This paper explains the importance of online repair to enterprises. It explains how Iron provides the same quality of repair as offline repair does, even while allowing client access to the file system. It presents some implementation detail, history, and performance evaluation. To the best of our knowledge, this publication is the first to present fully functional enterprise quality online repair. A follow-up paper will present implementation details and the enhancements mentioned in Sec. 5.3. We thank the many WAFL engineers who contributed to Iron over the years; they are too many to list. We also thank our reviewers and shepherd for their invaluable feedback.

## References


[1] Checking ZFS file system integrity. https://docs.oracle.com/cd/E23823_01/html/819-5461/gbbwa.html#scrolltoc.

[2] Disks and aggregates power guide. https://library.netapp.com/ecm/ecm_download_file/ECMLP2496263.

[3] How you schedule automatic raid-level scrubs. https://library.netapp.com/ecmdocs/ECMP1196912/html/GUID-A2F3A870-5C8D-4A68-AC8C-912946CECAC0.html.

[4] Kernel Bug Tracker. http://bugzilla.kernel.org/.

[5] Linux btrfs blog posts. http://marc.merlins.org/perso/btrfs/post_2014-03-19_Btrfs-Tips_-Btrfs-Scrub-and-Btrfs-Filesystem-Repair.html.

[6] Metrocluster for clustered data ontap 8.3.2. https://storageconsortium.de/content/sites/default/files/WP_NetApp%20Metrocluster%20for%20Clustered%20Data%20ONTAP%208.3.2.pdf.

[7] Scalability and performance using flexgroup volumes power guide. http://docs.netapp.com/ontap-9/index.jsp?topic=%2Fcom.netapp.doc.pow-fg-mgmt%2FGUID-A304BBC1-C00C-4E7A-989E-7C5A0E505146.html.

[8] Wendy Bartlett and Lisa Spainhower. Commercial Fault Tolerance: A Tale of Two Systems. *IEEE Transactions on Dependable and Secure Computing*, 1(1):87–96, 2004.

[9] Robert Baumann. Soft errors in advanced computer systems. *IEEE Design & Test of Computers*, 22(3):258–266, 2005.

[10] Steve Best. JFS Overview. http://www.ibm.com/developerworks/library/l-jfs.html, 2000.

[11] Sushrut Bhowmik, Vinay Kumar, Sreenath Korrakuti, Arun Pandey, and Sateesh Pola. Automatic incremental repair of granular filesystem objects. pending patent application.

[12] Eric J. Bina and Perry A. Emrath. A Faster fsck for BSD Unix. In *Proceedings of the USENIX Winter Conference*, January 1989.

[13] Jeff Bonwick and Bill Moore. ZFS: The Last Word in File Systems. http://opensolaris.org/os/community/zfs/docs/zfs_last.pdf, 2007.

[14] John Chapin, Mendel Rosenblum, Scott Devine, Tirthankar Lahiri, Dan Teodosiu, and Anoop Gupta. Hive: Fault Containment for Shared-Memory Multiprocessors. In *Proceedings of the fifteenth ACM Symposium on Operating Systems Principles (SOSP)*, pages 12–25, 1995.

[15] Peter Corbett, Bob English, Atul Goel, Tomislav Grcanac Steven Kleiman, James Leong, and Sunitha Sankar. Row-diagonal parity for double disk failure correction. In *Proceedings of Conference on File and Storage Technologies (FAST)*, 2004.

[16] Storage Performance Council. Storage performance council-1 benchmark. www.storageperformance.org/results/#spc1_overview.

[17] Matthew Curtis-Maury, Vinay Devadas, Vania Fang, and Aditya Kulkarni. To waffinity and beyond: A scalable architecture for incremental parallelization of file system code. In *Proceedings of USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 419–434, 2016.

[18] Matthew Curtis-Maury, Ram Kesavan, and Mrinal K. Bhattacharjee. Scalable write allocation in the WAFL file system. In *Proceedings of the International Conference on Parallel Processing (ICPP)*, August 2017.

[19] Peter Denz, Matthew Curtis-Maury, and Vinay Devadas. Think global, act local: A buffer cache design for global ordering and parallel processing in the WAFL file system. In *Proceedings of the International Conference on Parallel Processing (ICPP)*, August 2016.

[20] John K. Edwards, Daniel Ellard, Craig Everhart, Robert Fair, Eric Hamilton, Andy Kahn, Arkady Kanevsky, James Lentini, Ashish Prakash, Keith A. Smith, and Edward Zayas. FlexVol: flexible, efficient file volume virtualization in WAFL. In *Proceedings of the 2008 USENIX Annual Technical Conference*, pages 129–142, Jun 2008.

[21] Daniel Fryer, Kuei Sun, Rahat Mahmood, Ting-Hao Cheng, Shaun Benjamin, Ashvin Goel, and Angela Demke Brown. Recon: Verifying file system consistency at runtime. In *Proceedings of 10th USENIX Conference on File and Storage Technologies (FAST)*, February 2012.

[22] Gregory R. Ganger and Yale N. Patt. Metadata Update Performance in File Systems. In *Proceedings of 1st USENIX Conference on Operating Systems Design and Implementation (OSDI)*, November 1994.

[23] Jim Gray. Why do computers stop and what can be done about it? Tandem Technical Report 85.7, June 1985.

[24] Haryadi S. Gunawi, Abhishek Rajimwale, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-

Dusseau. SQCK: A Declarative File System Checker. In *Proceedings of 8th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2008.

[25] Robert Hagmann. Reimplementing the Cedar File System Using Logging and Group Commit. In *Proceedings of the 11th ACM Symposium on Operating Systems Principles (SOSP)*, November 1987.

[26] Val Henson. The Many Faces of fsck. http://lwn.net/Articles/248180/, 2007.

[27] Val Henson, Zach Brown, Theodore Ts'o, and Arjan van de Ven. Reducing fsck time for ext2 file systems. In *Linux Symposium*, pages 395–407, 2006.

[28] Val Henson, Arjan van de Ven, Amit Gud, and Zach Brown. Chunkfs: Using divide-and-conquer to improve file system reliability and repair. In *Proceedings of the 2nd Conference on Hot Topics in System Dependency (HotDep)*, 2006.

[29] Dave Hitz, James Lau, and Michael Malcolm. File system design for an NFS file server appliance. In *Proceedings of USENIX Winter 1994 Technical Conference*, pages 235–246, Jan 1994.

[30] Microsoft Inc. Building the next generation file system for windows: Refs. https://blogs.msdn.microsoft.com/b8/2012/01/16/building-the-next-generation-file-system-for-windows-refs/, 2012.

[31] NetApp Inc. Overview of wafliron. `https://kb.netapp.com/support/index?page=content\&id=3011877`, 2016.

[32] Ram Kesavan, Rohit Singh, Travis Grusecki, and Yuvraj Patel. Algorithms and data structures for efficient free space reclamation in wafl. In *15th USENIX Conference on File and Storage Technologies (FAST)*, 2017.

[33] Ram Kesavan, Rohit Singh, Travis Grusecki, and Yuvraj Patel. Efficient free space reclamation in WAFL. *ACM Transactions on Storage (TOS)*, 13, October 2017.

[34] Harendra Kumar, Yuvraj Patel, Ram Kesavan, and Sumith Makam. High performance metadata integrity protection in the WAFL copy-on-write file system. In *15th Usenix Conference on File and Storage Technologies (FAST)*, 2017.

[35] Xin Li, Kai Shen, Michael C. Huang, and Lingkun Chu. A memory soft error measurement on production systems. In *USENIX Annual Technical Conference (ATC)*, June 2007.

[36] Ao Ma, Chris Dragga, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, and Marshall Kirk Mckusick. Ffsck: The fast file-system checker. *ACM Transactions on Storage (TOS)*, 10(1):2:1–2:28, January 2014.

[37] Joshua MacDonald, Hans Reiser, and Alex Zarochentcev. http://www.namesys.com/txn-doc.html, 2002.

[38] T. C. May and M. H. Woods. Alpha-particle-induced soft errors in dynamic memories. *IEEE Tranactions on Electron Devices*, 26(1), 1979.

[39] Marshall Kirk McKusick. Running 'fsck' in the Background. In *BSDCon '02*, 2002.

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

[41] T. J. O'Gorman, J. M. Ross, A. H. Taber, J. F. Ziegler, H. P. Muhlfeld, C. J. Montrose, H. W. Curtis, and J. L. Walsh. Field testing for cosmic ray soft errors in semiconductor memories. *IBM Journal of Research and Development*, 40(1):41–50, 1996.

[42] Justin Parisi. Netapp flexgroup volumes: An evolution of nas. `https://blog.netapp.com/blogs/netapp-flexgroup-volumes-an-evolution-of-nas/`.

[43] David Patterson, Garth Gibson, and Randy Katz. A Case for Redundant Arrays of Inexpensive Disks (RAID). In *ACM SIGMOD International Conference on Management of Data*, pages 109–116, June 1988.

[44] Mendel Rosenblum and John Ousterhout. The Design and Implementation of a Log-Structured File System. *ACM Transactions on Computer Systems*, 10(1), 1992.

[45] Bianca Schroeder, Eduardo Pinheiro, and Wolf-Dietrich Weber. DRAM errors in the wild: A Large-Scale Field Study. In *Proceedings of the 2009 Joint International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS/Performance '09)*, Seattle, Washington, June 2007.

[46] Thomas J.E. Schwarz, Qin Xin, Ethan L. Miller, Darrell D.E. Long, Andy Hospodor, and Spencer Ng. Disk Scrubbing in Large Archival Storage Systems. In *IEEE 12th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems*, 2004.

[47] Christopher A. Stein, John H. Howard, and Margo I. Seltzer. Unifying File System Protection. In *Proceedings of USENIX Annual Technical Conference*, pages 79–90, June 2001.

[48] Rajesh Sundaram. The Private Lives of Disk Drives. http://www.netapp.com/go/techontap/matl/sample/0206tot_resiliency.html, 2006.

[49] Stephen C. Tweedie. Journaling the Linux ext2fs

File System. In *The Fourth Annual Linux Expo*, Durham, North Carolina, 1998.

[50] Wikipedia. Btrfs. en.wikipedia.org/wiki/Btrfs, 2009.

[51] Yichen Xie, Andy Chou, and Dawson Engler. ARCHER: using symbolic, path-sensitive analysis to detect memory access errors. In *Proceedings of the 9th European software engineering conference (FSE)*, pages 327–336, September 2003.

[52] 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 8th USENIX Conference on File and Storage Technologies (FAST)*, 2010.

[53] J. F. Ziegler and W. A. Lanford. Effect of cosmic rays on computer memories. *Science*, 206(4420):776–788, 1979.