Recording and replay of block device operations for file system consistency analysis

Technical University of Vienna

Dominik Süß (01429781)

March 9, 2022

Abstract

In this work, we develop a tool to record and replay block device operations. The tool serves as a proof of concept and basis for further research in the area of file system consistency. The tool saves a time series of device operations and allows for restoration of the disk state at any point in time. To do this, a virtual block device is provisioned which records the performed operations. During replay, the recorded operations are played back up to a specified point in time, allowing researchers to inspect the file system consistency in specific states. As a side effect, this recording also serves as a interchange format of file system traces which can be utilized to further specify bug reports.

Afterwards, a framework for automated analysis of existing file system implementations is presented. This framework is then used to perform simple tests against the `xfs`, `ext4`, `btrfs`, `fat` and `ntfs` file systems.

Contents

1 Introduction **3**

1.1 File Systems . . . 3

1.2 Linux Storage Architecture . . . 5

1.3 Previous work . . . 6

2 Approach **7**

2.1 Communication Protocol . . . 8

3 Implementation **8**

3.1 Kernel Module . . . 8

3.2 Transfering data out of the kernel . . . 8

3.3 Writing to the kernel module . . . 9

4 Userspace **10**

4.1 Extensibility . . . 10

5 Filesystem evaluations **11**

5.1 mkfs . . . 11

5.2 Partial Writes . . . 11

6 Automated Analysis **13**

6.1 Test harness structure . . . 13

6.2 Test methods . . . 16

6.3 Running the tests . . . 17

6.4 Test results . . . 17

7 Future research **21**

8 Conclusion **22**

9 Bibliography **23**

1 Introduction

Files are the basic storage unit on modern systems. They are organized using file systems which in turn manage the layout of records and directory structures on physical devices. File systems are usually looked at from a performance and memory efficiency standpoint which shows stark differences between popular formats. This work however approaches file systems from a reliablity perspective.

When deciding which file system to use for a system, its behaviour in certain failure states is often a critical factor. A corrupted file system can break the system entirely or, even worse, introduce inconsistencies during seemingly normal operations.

Power loss or physical disconnects happen from time to time (either from natural causes like power outages or mistakes) and especially in critical environments, data loss is not an option. A lot of research in this research area is done to provide safeguards (Gunawi et al. 2007) or develop new file systems (Bonwick et al. 2003). However, as already shown by *Recon* (Fryer et al. 2012), these conceptual improvements only work as long as the implementation is correct.

For large scale distributed systems, corruption of a single file system is not critical as they are built with redundancy in mind at a higher level. The focus of our research is in providing help in the development process and continously analyze existing implementations for errors. Since server workloads are mostly virtualized and rely on complex distributed file systems, we focus on desktop user workloads.

Analyzing file system consistency of specific implementations is a very involved process, often requiring physical activity to disconnect power, reboot machines or to configure hardware. As such, it is hard to run experiments rapidly and reproducable across multiple filesystems. It is also immensly difficult to time the operations.

We present a system called *Sitelen* which aims to aid researchers by allowing the recording and following replay of block device operations. This way, arbitrary data can be written to the file system and inspected in retrospect. By relying on replay after the data has already been writen, Sitelen offers a more ergonomic approach which does not rely on catching the exact call to interrupt. Another big advantage is the ability to run the same consistency checks at *any* point in time.

By jumping to a specific state, a disconnect of the underlying block device is simulated, without having to perform hard to time actions during runtime. Since the information written to the disk is the only persistent information the file system stores, our tooling can also be used to find bugs not related to hardware failure/disconnects. If the filesystem writes corrupted information, the exact steps which lead to this can be replayed at will. By cropping specific writes from the log, a minimal set of writes can be used in bug reports.

1.1 File Systems

File systems offer a standardized way to write and retrieve files. The prevalent type of file system is called hierarchial as they structure their contents using directories containing files. Hierarchical file systems are far from the only type of file system available (Seltzer and Murphy 2009). They are however the most common ones.

The Portable Operating System Interface (POSIX) represents a common denominator across file systems. It describes the information to be stored for each file and standardized interfaces to access them. Without this interface, each file system would have to provide their own set of rules and methods which complicates portability across platforms.

1.1.1 File System Implementation Types

To realise the goals of file systems, a multitude of options is availalble to the developers. While early file systems relied on management utilities like `fsck`, the amount of data soon became too overwhelming for these tools to continue to be a viable option from a performance perspective. Nonetheless, modern file system still ship with support tooling to repair a file system after a crash. For example: *xfs* provides `xfs_repair` and *btrfs* uses a command with the same name to check and repair the file system. The file systems in question are structured around the idea that consistency is implicit and does not have to be verified or repaired on each mount.

On top of their structure, file systems also offer some other features as unique selling points. Many file systems offer transparent encryption or compression while others optimize for amount of writes performed. The reason why minimizing write operations is a goal for some file systems is to increase the longlivety of flash devices which only offer a limited amount of write cycles. During creation, the parameters of a file system are also configurable. Most commonly this includes options like sector size, compression level or tuning information for specific hardware setups.

Bornholt et al. split the most prevalent structural approaches into the following four categories: journaling, log-structured, copy-on-write as well as soft updates. The different aproaches are described more in depth below.

1. Journaling

   The idea of journaling is keeping a *write-ahead-log* on disk to describe the operations about to be performed. This information is kept in the journal and allows the file system to know if a crash occured and if it can recover itself. Most of the time, the journal is kept on the same device as the rest of the files. However, many journaling file systems also support the option of storing the journal on a separate device. This introduces an additional layer of complexity but increases the failure tolerance.

   Popular examples include: `ext4`, `ntfs` as well as `xfs`.

2. log-structured

   These file systems operate on the principle that the entire disk is a large, sequential log (Rosenblum and Ousterhout 1992). The log also contains indexing information to speed up operations and recovery. This type of file system can be seen as an early precursor to the copy-on-write file systems outlined below. Rosenblum and Ousterhout also provide an implementation for a log-structured file system called *Sprite LFS*.

Usually these kinds of file systems also include a *checkpoint* mechanism which records states in which the disk is consistent. This information can then be used to discard non-consistent data on recovery.

A popular log structured file system in use today is *F2FS* (Lee et al. 2015). It was developed by Lee et al. for Samsung Electronics with the goal of optimizing for flash storage. Another unusual feature of F2FS is the option to treat file formats differently. The file system allows for different treatment of *hot* and *cold* files. This information is used for performance fine-tuning.

3. Copy-on-write

   COW file systems operate similar to log-structured ones but do not strictly rely on a log to keep information about entries. Rodeh, Bacik, and Mason specify *BTRFS* (pronounced *better FS* or sometimes *butter FS*), a file system which is now adopted as the default file system for some popular linux distributions.

   It works by writing data to fresh locations and redirecting the location information to these new blocks. This allows for easy snapshot operations in which the old superblock is protected from being overwritten. Another popular file system using this aproach is ZFS (Bonwick et al. 2003) which is mainly used in BSD installations but now also has its own linux implementation (Ahrens 2014).

4. soft-updates

   Soft update file systems work by ordering the operations so that each operation by its own does not result in an inconsistent file system. On mounting the file system after a crash, the dependency graph of operations is inspected, rolled-back and then rolled-forward. This implies a form of journal in some capacity which is why this aproach is sometimes reffered to as *JournalLite* (McKusick and Roberson 2010).

   Currently, only the BSD family of operating systems ships with support for soft-update file systems. As our implementation focuses on Linux, we did not analyze the behaviour of soft updates in this work.

1.2 Linux Storage Architecture

To figure out where to inject our code, we must first discuss the Linux Storage Architecture. The Linux storage architecture can be split up in separate layers as seen in figure **??**.

Applications use a set of user space APIs to write to files. In the case of linux these APIs are based on the POSIX standard. However, each file system has its own way of accessing contents. Having the operating

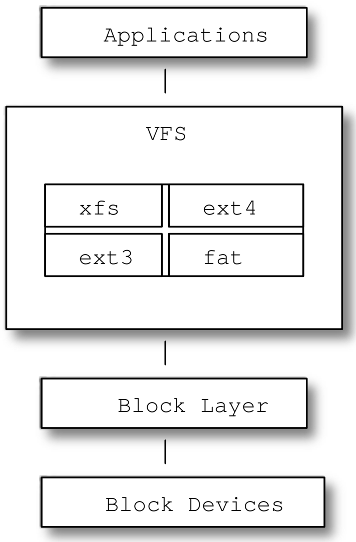


Figure 1: Linux Storage Architecture

system implement every file system directly, would thus result in a bottleneck. Instead, the linux developers introduced another abstraction layer called VFS. This also allows the kernel to mount multiple different file systems into the same directory tree. The VFS also serves as point of reference for Directory, inode and Page Caches, allowing for better end-user performance across file systems (Kroeger and Long 2001).

Below the VFS, the *Block Layer* handles ordering and queueing of so called *requests*. If an operation needs to access multiple locations in storage, the block layer splits this up into multiple requests to contigous regions. The requests are then put into the dispatch queue from which the next layer picks up the requests. This is also the layer on which concurrent access gets ordered into sequential requests.

At the lowest level we find the *Block Device Drivers*. The drivers receive requests from the block layer via the aformentioned dispatch queue. The requests are taken from the queue and written to the block device. Since persistence is only available at this point, the block device layer is a sensible entrypoint for our tool to record the written data. Other than responding to requests, the block device driver also offers information about the hardware device itself. This includes capacity, flags and physical block size. The information can then be used to optimize performance for this specific physical device.

For our implementation, the only flag set is `QUEUE_FLAG_NONROT` which tells the block layer that this disk is not a rotational device. The block layer can then infer that no performance can be gained by allocating data physically close together.

1.3 Previous work

Replaying of block device traces is not a new concept. Tools like `blktrace`(Axboe et al., n.d.) and `blkreplay`(Schöbel-Theuer 2016) exist to load test file systems and simulate real workloads captured beforehand. One noteable difference is the amount of data recorded during tracing. The beforementioned tooling only records the time and amount of data written but not the specific data. They are not meant to split up the traces or travel to a specific point.

On a higher conceptual level, there exists `r2`(Guo et al. 2008) which uses a different architecture. Instead of recording writes as they happen to a live kernel, they provide their own application-level kernel with custom system calls. This however requires library injection or other modifications to the existing code which could in turn produce unintended side effects or even introduce bugs on its own.

In their paper, Bornholt et al. describe how to model and analyze different consistency models. This allows for a detailed description of consistency guarantees in the case of a crash and a tool called *Ferrite* which checks these models.

2 Approach

Due to Linux' prevailing market share, excelent documentation and filesystem availability, Sitelen is developed to work with the Linux storage architecture. Implementation of the system for other platforms such as BSD, NT or Plan9 is out of scope but there are no inherent properties of this approach which would prevent the implementation on these architectures.

Our system is split in two parts:

- A kernel module simulating a block device
- A userspace program storing and analyzing the writes passed by the kernel module

This way, a clear seperation of concerns can be made and the underlying kernel module could theoretically be replaced by equivalencies on other platforms. A diagram of the archictecture can be found in Figure ??. When playing back the recorded writes, the flow of data is reversed (Figure ??).

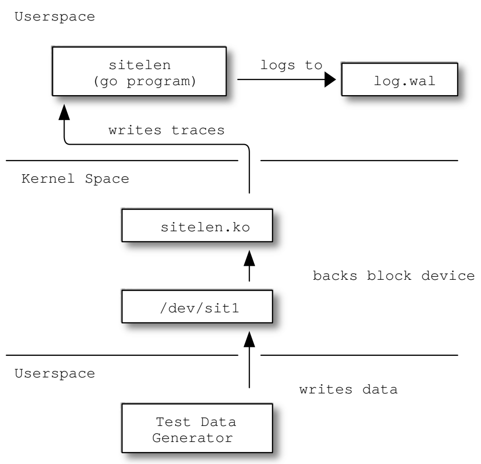


Figure 2: Sitelen Recording Architecture

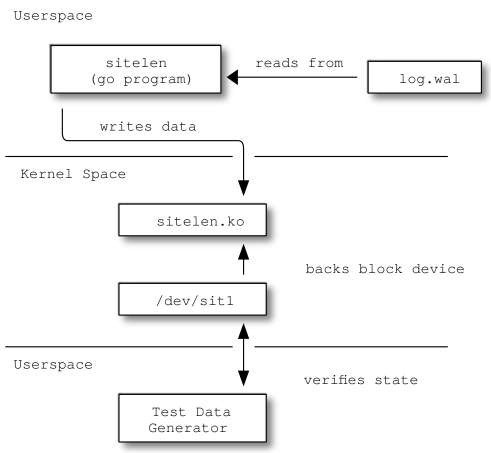


Figure 3: Sitelen Replay Architecture

2.1 Communication Protocol

To keep the overhead to a minimum, a simplified TLV (Type-Length-Value) encoding is used. The format is used for all communications between the kernel module and the userspace agent. The first 4 Bytes contain the length of the record written while the next 8 Bytes contain the sector index. After these twelve bytes of metadata, follow exactly `len` bytes of raw sector data. This approach optimizes throughput while still being easy to implement. If we only have a single data pipe available, a more complex protocol would be required as the packets could be intertwined when a process switch to another CPU is made. However, the way we implemented the data transfer, no such state is possible.

3 Implementation

3.1 Kernel Module

As we do not need persistent storage longer than the lifetime of the kernel module, an in memory block device is the simplest way to allow RW operations. First we looked into implementing this from scratch as seemingly there exist a lot of guides (Corbet, Rubini, and Kroah-Hartman 2005). Sadly, most of the guides are based on an outdated kernel version. Beginning in 2002, the Linux block device layer underwent a major rewrite to allow for more flexible performance tuning (Axboe, Bhattacharya, and Piggin 2002). This meant that the information presented in most guides still used the old architecture and concepts. Fortunately the Linux Kernel ships with an open source in-memory block device driver called `brd`[1]. The open license of the Linux kernel allows us to take the existing code and adapt it to our needs as long as we keep our implementation open source.

The block layer submits information as `BIO` (Block Device IO) structs. These contain a set of Operations (read or write) to the block device. Each of these Operations is performed in sequence which makes them an ideal place to record the information written by the file system.

Our tracing implementation is located in the `do_bvec` function of the block device driver. Here we already have access to the data to be written as well as the target sector. This is also the place where the original in memory block device driver performs the memory copy to the datastructure.

3.2 Transfering data out of the kernel

Before we look at the recording of information, we have to figure out a way to store the traces. It is considered bad practice to write files directly from kernel space [2] because of permissions and separation of concerns.

The kernel offers different facilities to interact with userspace, each with its own advantages and drawbacks. For our usecase `relay` (Wisniewski 2003) is the best option as it allows for streaming of data without having to keep it all in memory. This way, the kernel module just needs to write to the queue and an userspace agent can pick up

[1]Linux BRD Driver: `https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/tree/drivers/block/brd.c`

[2]`https://kernelnewbies.org/FAQ/WhyWritingFilesFromKernelIsBad`

the data and transform or store it.

Due to the fact that the relay is scoped per CPU, our usperspace agent needs to listen on each file in a seperate thread. Another valid approach would be to run the kernel module in a Virutal machine with only one cpu available.

Using a tool like fio [3] we can benchmark our implementation against the pure `brd` driver. Since our module only modifies the writing of data and sequential access or locality is not changed, we will perform a `randwrite` benchmark for 60 seconds. In the case of tracing and replaying information, the metric of interest is the IO Operations per second (IOPS)

Blockdevice	IOPS
`brd`	1307k
sitelen	553k
NVME-SSD	74.3k

These results show, that by simply extracting the data out of the kernel, the module encounters a severe perfomance penalty. It can be reasoned that this is because every write performs *two* memory copy operations instead of one. The second operation is caused by writing to the relay.

Even though the writes to the sitelen device were recorded to the same NVME-SSD, it outperforms it by quite a lot. This is due to buffering on the writing side. The IO Operation completes when the data has been written to the relay and is not affected by the userspace agent.

[3]`https://github.com/axboe/fio` Version 3.26

3.3 Writing to the kernel module

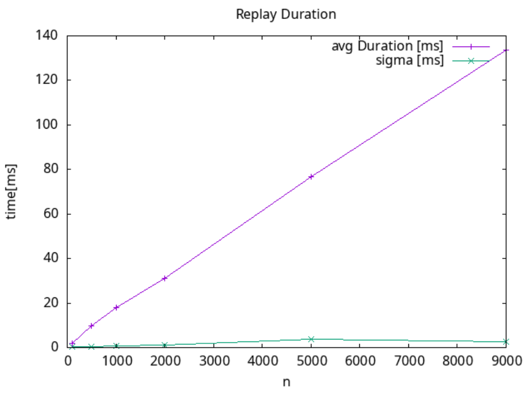
To replay the recorded information, we can utilize the existing debugfs structure. We create a new file called `replay` and provide a custom file operations structure which in turn replays the writes to the in-memory data structure.

The protocol used to write to the `replay` file is the same used during recording. To benchmark the replay procedure, we recorded a series of writes and replayed them up to a certain point. The writes used for this were repeated calls of `mkfs.ext4` as this consistently performs writes to all sectors while keeping the written data easily compressible.

To obtain detailed information, we used `hyperfine`[4] with five warmup runs. The table below shows the results of these benchmarks with n signifying the amount of replayed writes.

n	avg Duration [ms]	sigma [ms]
100	1.8	0.2
500	9.6	0.5
1000	17.9	0.7
2000	31.0	1.0
5000	76.7	3.8
9000	133.8	2.5

[4]Hyperfine Project: `https://github.com/sharkdp/hyperfine`



As expected, the performance of replaying files is linear and does not vary much between runs. Sub-linear performance is not possible as we need to perform the write operations in order. This also means that it is not possible to split the work onto multiple cores as is the case during recording.

Performance could be increase by writing boilerplate contents to the file system directly through tools like dd. Using this method, removes the ability to replay a subset of writes but offers performance on the same level as shown in section ??.

4 Userspace

The userspace agent is used to record and analyze the data transfered out of the kernel module. As the kernel relay facility works by channeling the data respective to the current CPU core, the userspace agents needs to run a separate routine for each CPU core. The chosen programming language *go* uses goroutines, a lightweight userspace thread implementation to handle concurrency.

The agent listens on each CPU relay and pushes information to a centralized logging facility. In its simplest form, the logging facility writes to a file in the following format:

seq. id	sector	data hash
1	0	C9B3B57AF5BB42...

By storing the data seperately and only referencing it by its hash, we optimize for storage space. It also allows us to edit the logfile without having to load the entire history into the editor.

As an example, formating the blockdevice using `mkfs.ext4` writes the same data to 399 sectors. With a sector size of `4096`, this way of storing data saves around 1 MB of disk space.

Another big advantage of this method is the reusal of data blocks across many traces.

4.1 Extensibility

The way the recording system is structured, allows for extensibility in many directions. The interface for a complete storage implementation can be seen below:

```
type Log interface {
  Write(
    sector uint64,
    hash []byte
  ) (uint64, error)
  Iterate(
    limit uint64,
    callback IteratorCallback
  ) error
}
type DataStore interface {
  Get(h []byte) ([]byte, error)
  Put(h []byte, data []byte) error
}
```

The `DataStore` is used for blob storage while the `Log` structure keeps a sequential log of events. Example extensions could include a database backend to share blobs and events across multiple workstations or a SQLite backend to have a self-contained file instead of a directory and acompanying log file.

5 Filesystem evaluations

Now that we have a way to trace operations, we can perform preliminary analysis of popular file systems to show differnces in crash handling. All the tests below were carried out on a sitelen disk of size `16` Mebibyte. The operations were replayed and partially omited manually.

5.1 mkfs

The `mkfs` utility is used to format block devices or partitions with a filesystem. Each filesystem provides its own extension like `mkfs.xfs` or `mkfs.ext4`. In this experiment we used our tool to record the number of writes to the block device during formating. By further analyzing the written sectors, we can detect any sectors written multiple times. This could point to potential performance increases since file system initialization should not require multiple writes to the same disk location.

FS	WR Operations	Dupl. writes
`ntfs`	5356	0
`xfs`	1063	1
`exfat`	516	0
`ext4`	315	1
`fat32`	68	0
`fat16`	13	0

The results show, that the amount of duplicate writes is much lower than assumed. Furthermore, the duplicate writes all occur in sector 0 and at the end of formating which could be caused by the file system acknowledging the completion of formating. Many file systems zeroed out the entire block device, which could be optimized by checking the contents beforehand. The only advantage of this would be reducing the amount of write cycles for flash devices. Reasons for this are laid out in section **??**.

5.2 Partial Writes

This section contains information about tests performed manually, for the automated test results see the Partial Write testcase instead.

The next test is performed by creating file containing random data to the formated file system. The `32Ki` file is written using the `dd` program reading from `/dev/urandom` and writing to a regular file in the root of the filesystem. To verify the contents of the file, we note its `md5sum`. Afterwards, we replay the recorded writes one by one and inspect the intermediate states for file presence or content. The file systems tested were `fat16`, `ext4` and `btrfs` as these file systems are some of the most commonly used file systems. These file systems also represent different classes, with `fat16` being a simple file allocation table, `ext4` providing journaling support and `btrfs` offering Copy-On-Write.

All files were written using the following command: `dd if=/dev/urandom of=/mnt/file bs=4096 count=8`. A block size of 4096 bytes was used since this is the sector size of the virtual block device. To avoid state kept in memory, the recording was started *before* mounting the file system.

5.2.1 fat16

During mount and unmount, the fat16 file system writes information to sector 0. This information is irrelevant for file system consistency and can be ommited. The data is written in three separate steps. Depending on the amount of writes replayed, this results in two different failure states:

1. Write the file metadata to the FAT
   - Failure → The file is present but empty
2. Write the data to the disk
   - Failure → The file is present but still empty (Data is recoverable by recovery tools)
3. Write the amount of data written to the FAT
   - Success → The file is present and valid

By reordering or omitting some writes, a third failure state can be reached. If only the final write succedes, the file is listed as its expected size, but the contents are invalid.

5.2.2 ext4

An interresting observation made during analysis of the ext4 traces is the creation of the `lost+found` folder. This folder is *not* created during formating but on the first mount. This creation also causes a lot of zero writes to sectors which have previously been cleared during formating, which shows a potential performance improvement.

Like with fat16, ext4 writes information on mount and unmount which is does not cause inconsistency if ommited. The initial creation of the `lost+found` folder can also be omited as it will be created on the next mount.

By replaying the activity up to the point of writing to the superblock, the filesystem does not show any files. As the superblock spans multiple sectors, a partial write causes the mount syscall to fail with the error *Structure needs cleaning.* Using the `fsck.ext4` utility, the mount will succed but the file and its contents are gone.

The only way to induce a failure state is by erasing data written previously while leaving the journal and superblock unaffected. This leads to the respective parts of the file being empty. In a real environment, this can only occur when the hard drive somehow loses specific write operations while allowing others to succeed.

5.2.3 btrfs

In contrast to the other file systems inspected, `btrfs` does not handle ommiting of mount operations gracefully. When removing the writes happening during mount, the file system becomes corroputed and is unable to be restored by using the *check* command. Granted, this is something which will never realistically happen unless the device is silently dropping write operations. Ommiting the unmount operations does have no effect on the resulting file system.

By stopping the writes at specific locations it is possible to reach two failure states. The first one occurs when data has been written, but no metadata updates took place. In this case, the file system will be empty as no reference to the file can be found. If the metadata writes finish before all data has been written, read operations on the file will fail with a general *Input/Output* error instead of returning garbage. Using the default `btrfs-check` tool, the file system *cannot* be repaired.

6 Automated Analysis

As shown in the previous section, even a manual analysis can yield some interresting insights into the different file systems and their inner workings. However, the actions performed do not sufficently describe every action taken by file systems in regular operations. It does not take metadata updates, overwriting, deleting or copying into account.

More involved analysis is very time-consuming and labour-intensive. This is why we implemented a simple test harness to automate the testing of file systems.

6.1 Test harness structure

Following the Unix principle of doing one thing and doing it well, the decision was made to implement the test harness as an extra tool instead of extending the existing go program to include testing capabilities. The harness is written in bash and allows for customization by providing scripts which will be called at the appropriate points in time.

To verify if file systems fulfill our expectations of consistency, we must first define the concept of a consistent file system further.

6.1.1 Defining consistency

Current operating systems adhere to a set of APIs defined by POSIX. A common misunderstanding is, that these calls are intended to be atomic. POSIX only specifies atomicity when talking about FIFOs, pipes and multiple threads in the same process. When talking about file access across multiple processes, the interfaces do not neccesarily have to be atomic (Siebenmann 2020).

In the context of our work, this becomes important as we now have to be more lax on our definition of consinstency. We declare a state consistent if the state is a logical consequence of a sequence of write operations which occured until the interruption. Taking a look at an example:

```
echo "content" > file1
sync
echo "updated" > file2
mv file2 file1
sync
```

Consistent states for this sequence of events include:

- `file1` does not exist
- `file1` exists with the contents `content`
- `file1` exists with the contents `updated`

The contents of `file2` and whether it exists or not are not relevant to the consistency. In the syntax chosen for the testcases, calls to `sync` constitute fixed points after which the program has succesfully written information to disk.

All other states would provide the user with incorrect data or result in the loss of existing data.

6.1.2 Modeling failure states

In our test framework, the verification is written as a `bash` script as well. This also means that any assertions have to be made explicit.

An example verification script for the sequence of events above could look something like this:

```
if [ -f file1 ]; then
  if [[ $(< file1) != "content"
  && $(< file1) != "updated" ]]
  then
    echo "invalid file contents"
    exit 1
  fi
fi
```

6.1.3 Dealing with reordering

As explained in the introduction, the operating system might buffer writes or perform reordering of writes. This can cause our test harness to miss possible failure states. In the scope of this paper, we just accept this and run each test case multiple times for the same file system. If one would like to further reduce the scope of analysis, the kernel module can be adapted to circumvent the write queue.

Another level of reordering can be introduced by specific block device drivers and hardware vendors but this cannot be analyzed using the approach chosen.

6.1.4 Check and repair

Some file systems (notably *xfs* and *btrfs*) refuse to mount alltogether if they detect a crash. In this case, we want to run the respective tooling. When the mount command in the verification step fails, it calls a script called `fsck` which in turn contains file system specific instructions on how to repair a broken structure. We also track the amount of times this was needed in our test results. Modern init systems like systemd provide their own utilities (e.g `systemd-mount`) which run a file system check on every mount to be on the safe side. Due to an open issue in systemd [5], as of the time of writing, this cannot be tested automaticaly.

6.1.5 Report structure

In the current software development industry landscape there is no real standard in reporting testcases. The nearest standard is found in JUnit XML which comes from the java unit testing library with the same name [6]. Some languages provide tooling to convert their own test reports to the JUnit test format to be consumed by continous integration tools [7]. It is also worth noting that the XML schema itself is not part of junit but Apache Ant. Most integrations do expect the junit specific fields to present which is the reason why it is commonly refered to as JUnit XML report.

For our usecase the XML format is not

[5]https://github.com/systemd/systemd/issues/12493

[6]https://junit.org/junit5/

[7]https://github.com/jstemmer/go-junit-report

suitable as it does not allow more involved failure analytics and statistics. Our test harness outputs an SQL file in addition to a SQLite database file. By providing both the SQL instructions and the evaluated database file, we allow for a simple transition to other database system as well as enable researchers to combine multiple test results into a single database for larger statistical analysis.

At first glance, SQLite does not look like a suitable format to provide static reports of test runs but closer inspection yields a lot of benefits. The well established SQL query language offers familiarity for application developers and allows for explorative queries into the past test results. Additionally, tooling support for this format is prevalent in the data analysis space [8].

The chosen schema is structured as a sequence of events with additional information for different kinds of events. A graphical representation can be found in figure ??. To simplify queries, a view called `testresult` is provided as well. This view joins the events with their testruns and groups by testcase and retry. The result is a summary of the respective testcases and their success rate.

1. Base events

   Every event inherits the base event. The base event type contains the name of the testcase, a unique event ID and the numer of operations replayed.

2. Testrun events

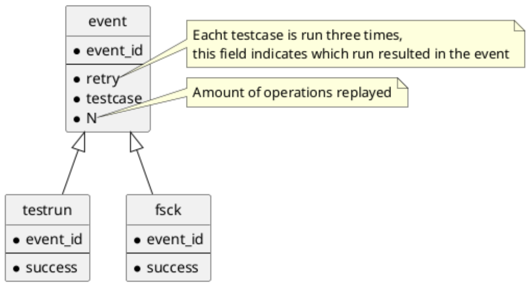

event
• event_id
• retry
• testcase
• N
Eacht testcase is run three times, this field indicates which run resulted in the event
Amount of operations replayed
testrun
• event_id
• success
fsck
• event_id
• success

Figure 4: Database Structure

   A `testrun` event signifies the completion of a replay-and-verify cycle. In addition to the fields for the base event, we also store the success of the test run.

3. fsck events

   A `fsck` event is emited whenever a manual fsck step had to be taken to mount the file system. If the file system check still left the device in a corrupted state, the failure flag is set to true.

6.1.6 Implementation

As is the case in every software project, the implementation of the test framework was not without its own problems. The first version of the test harness simply recorded the results of a certain shell script and replayed it while staticaly checking for consistency of a hardcoded file. This approach does not scale as every testcase has a different way to validate. The next iteration split the test scripts into two parts: `XXX.test` and `XXX.verify`. While at first everything seemed to work fine, a large portion of testcases failed because the verify script aborted and the file system staid mounted across test runs.

Bash offers a way to perform operations on exit using a `trap`. After implementing

[8]These topics are also highlighted by the SQLite developers themselves: `https://www.sqlite.org/appfileformat.html`

a trap which performs unmount operations, the first prototype of the test harness was complete.

Since some file systems refuse to mount in specific states, a mechanism to run a file system repair operation has to be devised. As up until this point, each testcase implemented mounting and unmounting separately, this was also a good time to implement a common boilerplate. This boilerplate is sourced by every testcase and performs the following actions: set common shell options, set up the exit trap as described previously and mount the file system. By extending this common base to run a file system specific repair script on failed mount, more testcases succeded.

Whether a repair operation has been performed or not is an interresting datapoint, so we need to communicate this back to the test harness. Communicating via the standard input and output streams is not an option since the test and verify scripts call thrid party programs which might fill the streams with garbage data. Another option would have been setting up named pipes or writing to predefined files but the simplest and most integrated solution is to set the exit code of the script depending on whether the file system performed a repair operation. This leads our test cases to exit with one of four exit codes:

- `0`: Successful without a file system repair
- `42`: Successful, file system was repaired
- `43`: Testcase or file system repair failed
- `other`: Testcase failed without a file system repair

Since errors can occur at every step of the verification, we have to keep track of the result of the file system check. This complicates error handling a bit but as everything can be done in the base, which is sourced by the testcases, the testcase developers do not have to worry about this and simply exit with a non-zero exit code for the test harness to detect the correct type of failure.

6.2 Test methods

In the scope of this paper, we modeled a list of testcases to represent various access patterns occuring in regular use. We run each testcase three times on every examined file system. The file systems chosen, represent different approaches to consistency.

6.2.1 Testcases

This section documents the test cases implemented in our work. The source code for these cases can be found in the tests/testcases directory of the project repository.

1. Partial Write (**PW**) This testcase describes the same procedure as taken in section **??**. A file spanning multiple sectors is created. Consistency is only reached if the file is present and the contents are correct. As described above, this consistency is not guaranteed by POSIX so low success rates are to be expected.
2. Write and rename (**WR**) This testcase is the source of the example shown in

section ??. It tests the atomicity of rename operations.

3. Partial Overwrite (**PO**) This testcase performes a write similar to the first testcase but then overwrites sections of the file afterwards. Consistent states include the file with its original contents or the new contents. A failure occurs when the contents represent an intermediate state.

4. Directory operations (**DO**) Here we examine metadata updates. A directory tree is created and moved around. The deepest level contains a file. The failure state we're looking for here is missing directories and the consistency of the file.

5. Sequential Consistency (**SC**)

   This testcase checks the implementation for consistency on a temporal scale. Two files are created in sequence. If the second file becomes available before the first one, the state is considered inconsistent.

6. Git Commit (**GC**)

   Git offers distributed versioning of source code and as such adds another level of redundancy above the file system. A common workflow is staging a file and then commiting it. This testcase examines the behaviour of git, if a crash occurs during the commit. An inconsistency occurs when the `git commit` command fails or the updated file does not contain the correct data.

6.3 Running the tests

With the testcases defined, the last step is to actually run the tests. The flow of the test cases is described in pseudocode below.

```
mkfs = record_mkfs()
for t in testcases:
    # run each test case three times
    for i in 1..3:
      replay(mkfs)
      test_record = record(t.test)
      if t.prep?:
          prep = record(t.prep)
      # validate each write separately
      for l in 1..(len(test_record)):
          replay(mkfs)
          replay(prep)
          replay(test_record,l)
          validate(t.validate)
```

Tests were performed on a Lenovo `P52`, outfitted with an `Intel Core i7-8850H` CPU and 32 Gigabytes of memory, running the `5.15.13-200.fc35.x86_64` version of the Linux kernel. The kernel is configured with the default kernel parameters for Fedora Linux 35. To accomodate the minimum volume size of btrfs, the sitelen device was created with a size of 256 Mebibyte.

6.4 Test results

The test results show a large variety of access patterns across different file systems. While some file systems performed barely any writes at all, the XFS file system performed the most writes of all by far.

Table ?? shows a very basic view on the test results. In the case of this table, the test runs are not split by testcase or run which can skew the results. Still it provides a general perspective on the overall performance of the respective file systems across workloads. Success Rate is calculated as number of successful tests divided by the amount of tests run in total. The column labeled *Successful FSCK* describes the rate of success when a manual file system check had to be performed. A special value of `n/a` denotes that no file system check had to be performed, as the file system was always able to be mounted without any errors. As an example, `xfs` was unable to mount the file system 75 times. Out of these, in 32 cases the `xfs_repair` tool was able to reach a consistent state. This results in a success rate of 32/75 $\simeq$ 0.43. A detailed look at the test results for the examined file systems can be found in section **??** and onwards. As each test case is run three times, this is separated in the detailed test result tables as signified by the *Retry* column.

Table 1: Summarized Test Results

FS	Success Rate	Successful FSCK
`xfs`	0.975	0.43
`ext4`	0.964	n/a
`btrfs`	0.962	0.00
`fat32`	0.438	n/a
`ntfs`	0.122	0.00

When confronted with the data, it seems like the logical conlusion would be that file system corruptions and data loss are a regular occurance. This is not the case, as the failure states produced by our approach are extraordinarily rare during regular use. To validate this hypothsis, we used the `blktrace` software and recored the number of block device write events over 6 minutes. During this time, the machine was used to write this paper, listen to music, browse the web and even perform some database operations. This resulted in a total number of 10811 write operations. Spread out across the timespan, this amounts to 30 events per second. On its own this still seems like a lot of operations. However, write operations usually only take around 10 microseconds[9], so the probability of interrupting one specific write operation is calculated to be 0.0003 which equals 0.03%. These numbers vary by workload and only serve to put the failure rate into perspective. For more conclusive results, a large scale benchmark would have to be performed.

As briefly mentioned in the introduction, for critical workloads, file systems should be kept redundant. This can either be acomplished by abstracting a single file system over the network or performing higher level consistency analysis on an application layer, as is the case with many database systems.

6.4.1 xfs

The XFS file system is the primary file system in use by CentOS and Redhat Enterprise Linux. It is journal based and offers many optimization features. For this work, we only examined the default parameters.

Out of all the file systems tested, XFS performed the most amount of write operations. This is due to the fact that by simply mounting and unmounting the block device, around 500 write operations are performed. During initial testing, this caused a lot of issues and falsified test results. When mount-

[9]This was calculated using the IOPS from Table in section **??**

Table 2: XFS test results

Testcase	Retry	Success Rate
PW	0	0.97
	1	0.96
	2	0.99
WR	0	0.98
	1	0.98
	2	0.99
PO	0	1.0
	1	0.99
	2	1.0
DO	0	0.93
	1	0.93
	2	0.92
SC	0	1.0
	1	1.0
	2	1.0
GC	0	0.96
	1	0.99
	2	0.98

ing a block device and immediately writing to it, the operations might get reordered by the block layer, causing an early corruption of the log which is irrecoverable. This also occured for other file systems but due to the sheer amount of writes, it only became apparent through the inconsistent test results of the XFS file system.

The performance of the `xfs_repair` utility is also acceptable. It correctly brought back the file system to a consistent state 42% of the time. This not only means that the file system was able to be mounted. It also signifies that the testcase was successful after mounting. It is important to keep in mind, that in our case, we always called the tool with the same arguments, regardles of file system corruption present. This means that the performance of the tool might exceed the numbers stated here when applied correctly.

6.4.2 ext4

Table 3: ext4 test results

Testcase	Retry	Success Rate
PW	0	0.64
	1	0.55
	2	0.48
WR	0	0.93
	1	1.0
	2	1.0
PO	0	1.0
	1	0.98
	2	0.92
DO	0	0.76
	1	0.84
	2	0.74
SC	0	1.0
	1	1.0
	2	1.0
GC	0	0.99
	1	0.98
	2	0.5

In our testruns, the journaling file system ext4 came second when comparing by average of aggregated test runs. ext4 had above 95% success with 9 out of 18 performed runs. As expected, the WR and SC testcases performed very well, as this subject was a controversial topic [10] with the result of supporting this model of consistency.

Potential areas for improvement have been identified in the area of partial writes as signified by the PW testcase.

The results for this file system show, that it is well suited for end user workloads due to its high success rate and low complexity.

Table 4: btrfs test results

Testcase	Retry	Success Rate
PW	0	1.0
	1	1.0
	2	0.97
WR	0	0.92
	1	0.81
	2	0.89
PO	0	1.0
	1	0.81
	2	0.98
DO	0	0.99
	1	0.99
	2	1.0
SC	0	1.0
	1	1.0
	2	1.0
GC	0	1.0
	1	1.0
	2	1.0

6.4.3 btrfs

BTRFS is classified as a Copy-on-Write file system. Since 2014 it is the default file system for OpenSUSE and was recently also introduced to be the default for desktop editions of Fedora. It is also in use by Facebook due to its snapshotting capabilities [11].

In our testcases, it was only slightly outperformed by xfs and ext4. It does however present a median success rate of 1 with its average being dragged down by the WR testcase. It seems like btrfs chose performance over consistency when handling metadata updates.

When inspecting a common developer workflow like git, btrfs performed extraordinarily well which shows a promising future for btrfs.

[10] `https://lore.kernel.org/lkml/1238742067-30814-1-git-send-email-tytso@mit.edu/`

[11] `https://facebookmicrosites.github.io/btrfs/docs/btrfs-facebook.html`

6.4.4 fat32

Table 5: fat32 test results

Testcase	Retry	Success Rate
PW	0	0.63
	1	0.27
	2	0.38
WR	0	1.0
	1	0.85
	2	1.0
PO	0	0.58
	1	0.5
	2	0.08
DO	0	0.68
	1	0.48
	2	0.55
SC	0	1.0
	1	1.0
	2	1.0
GC	0	0.06
	1	0.0
	2	0.0

Designed in 1977, the *File Allocation Table* format is one of the oldest file systems. It offers basically no consistency guarantees and keeps metadata stored in a central file allocation table. The only testcase in which fat32 excelled was sequential consistency. Otherwise, the test results varied widely and no strong conclusion can be drawn from these results alone.

6.4.5 NTFS

The *New Technology File System* is the default file system in use by Microsoft Windows. It is a journaled file system and as such offers modest consistency guarantees. With it being the most prominent file system in desktop use (where system crashes and power outages are a frequent

Table 6: ntfs test results

Testcase	Retry	Success Rate
PW	0	0.04
	1	0.04
	2	0.04
WR	0	0.06
	1	0.04
	2	0.04
PO	0	0.03
	1	0.03
	2	0.08
DO	0	0.08
	1	0.58
	2	0.48
SC	0	0.04
	1	0.04
	2	0.04
GC	0	0.08
	1	0.08
	2	0.08

occurance), the overall success rate of 0.122 seems very concerning at first glance. It can be explained by the simple fact that we used the Linux implementation of the file system which is of course different from the Windows implementation. This drastically worsens results and also prohibits a effective file system repair operation. The `fsck.ntfs` program instructs the user to mount the file system in windows and run a windows specific tool to repair the file system. This also shows in the success rate of file system repair operations where not a single one resulted in a success.

7 Future research

The goal of this paper is to aid researchers in examining file system implementations. The analysis performed in this paper is only scratching the surface in that regard. Future research topics include the extension of the system for other architectures (NT, BSD) as well as analyzing more file systems. During testing, we only examined the default file system parameters. Future research can be devoted to inspecting the effects of various tuning options on file system consistency.

Improvements can also be made to the way corrupted file systems are handled in our test harness. Currently, a one-fits-all forceful `fsck` approach is taken. It is trivial to see that this will not always produce the best possible results for various file systems. The current test harness also disregards the possibility of running a file system repair operation when the operating system performed a successful mount.

Another major tradeoff taken in this paper is the realization of the test framework as scripts. The tools called by these scripts embed a high degree of complexity and are points of failure themselves. By rewriting the testsuite in a programming language closer to the system layer, writing operations could be more granular. This in turn enables more complex test cases without including the noise and failure potential of third party tools. To gain this level of control, a new set of APIs must be made available from the kernel module. At least a way to reset the kernel module state has to be implemented.

During development of the testing framework, an interresting failure state was discovered which occurs when directly writing to the file system after it has been mounted. If data is written in quick succession to the mount operation, the block layer might re-order the writes and corrupt the log in the event of a crash. In this work, we worked around the issue by idling for a certain amount of time before writing.

While this paper focuses on specific existing implementations, a formal model of file systems can allow their consistency to be proven by model checkers. Insight gained here can also aid in the specificaton of consistency models during file system development.

8 Conclusion

The unexpected crash or power loss of operating systems is a real threat to data consinstency. The `sitelen` tool presented in this work is able to record and replay operations as they would be performed on a real device. Through designing and implementation of an automated test suite, we examined popular file system implementations from a standpoint of consistency after crash recovery. Notably XFS and ext4 performed extraordinary well for typical end user usage patterns. As expected, simpler file systems like fat16 performed considerably worse. The high complexity of BTRFS also caused a lot of testcases to fail.

Even though, many popular file systems exhibit non-optimal success rates, one should not come to the conclusion that a power loss always results in a corrupted file system. To reach these failure states, very precise timing is be needed.

The report format proposed can offer file system developers a way to exchange state information which leads to more concise bug reports and easier reproducability.

Source code and raw test results are available at `git.sr.ht/~thesuess/sitelen`.

9 Bibliography

Ahrens, Matthew. 2014. "OpenZFS: A Community of Open Source ZFS Developers.'" *AsiaBSDCon 2014*, 27.

Axboe, Jens, Suparna Bhattacharya, and Nick Piggin. 2002. "Notes on the Generic Block Layer Rewrite in Linux 2.5." 2002. `https://www.kernel.org/doc/html/latest/block/biodoc.html`.

Axboe, Jens, Alan D. Brunelle, Nathan Scott, and Tom Zanussi. n.d. "blktrace." `https://git.kernel.dk/cgit/blktrace/`.

Bonwick, Jeff, Matt Ahrens, Val Henson, Mark Maybee, and Mark Shellenbaum. 2003. "The Zettabyte File System." In *Proc. Of the 2nd USENIX Conference on File and Storage Technologies*. Vol. 215.

Bornholt, James, Antoine Kaufmann, Jialin Li, Arvind Krishnamurthy, Emina Torlak, and Xi Wang. 2016. "Specifying and Checking File System Crash-Consistency Models." In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*, 83–98. ASPLOS '16. Atlanta, Georgia, USA: Association for Computing Machinery. `https://doi.org/10.1145/2872362.2872406`.

Corbet, Jonathan, Alessandro Rubini, and Greg Kroah-Hartman. 2005. *Linux Device Drivers, 3rd Edition*. O'Reilly Media, Inc.

Fryer, Daniel, Kuei Sun, Rahat Mahmood, Tinghao Cheng, Shaun Benjamin, Ashvin Goel, and Angela Demke Brown. 2012. "Recon: Verifying File System Consistency at Runtime." *ACM Trans. Storage* 8 (4). `https://doi.org/10.1145/2385603.2385608`.

Gunawi, Haryadi S., Vijayan Prabhakaran, Swetha Krishnan, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2007. "Improving File System Reliability with I/O Shepherding." In *Proceedings of Twenty-First ACM SIGOPS Symposium on Operating Systems Principles*, 293–306. SOSP '07. Stevenson, Washington, USA: Association for Computing Machinery. `https://doi.org/10.1145/1294261.1294290`.

Guo, Zhenyu, Xi Wang, Jian Tang, Xuezheng Liu, Zhilei Xu, Ming Wu, M. Frans Kaashoek, and Zheng Zhang. 2008. "R2: An Application-Level Kernel for Record and Replay." In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, 193–208. OSDI'08. San Diego, California: USENIX Association.

Kroeger, Thomas M., and Darrell D. E. Long. 2001. "Design and Implementation of a Predictive File Prefetching Algorithm." In *2001 USENIX Annual Technical Conference (USENIX ATC 01)*. Boston, MA: USENIX Association. `https://www.usenix.org/conference/2001-usenix-annual-technical-conference/design-and-implementation-predictive-file`.

Lee, Changman, Dongho Sim, Jooyoung Hwang, and Sangyeun Cho. 2015. "F2FS: A New File System for Flash Storage." In *13Th USENIX Conference on File and Storage Technologies (FAST 15)*, 273–86. Santa Clara, CA: USENIX Association. `https://www.usenix.org/conference/fast15/technical-sessions/presentation/lee`.

McKusick, Marshall Kirk, and Jeffery Roberson. 2010. "Journaled Soft-Updates." *BSD-Can, Ottawa, Canada.*

Rodeh, Ohad, Josef Bacik, and Chris Mason. 2013. "BTRFS: The Linux B-Tree Filesystem." *ACM Trans. Storage* 9 (3). `https://doi.org/10.1145/2501620.2501623`.

Rosenblum, Mendel, and John K. Ousterhout. 1992. "The Design and Implementation of a Log-Structured File System." *ACM Trans. Comput. Syst.* 10 (1): 26–52. `https://doi.org/10.1145/146941.146943`.

Schöbel-Theuer, Thomas. 2016. "blkreplay." `https://github.com/schoebel/blkreplay`.

Seltzer, Margo, and Nicholas Murphy. 2009. "Hierarchical File Systems Are Dead." In *Proceedings of the 12th Conference on Hot Topics in Operating Systems*, 1. HotOS'09. Monte Verità, Switzerland: USENIX Association.

Siebenmann, Chris. 2020. "POSIX write() Is Not Atomic in the Way That You Might like." 2020. `https://utcc.utoronto.ca/~cks/space/blog/unix/WriteNotVeryAtomic`.

Wisniewski, Robert W. 2003. "relayfs: An Efficient Unified Approach for Transmitting Data from Kernel to User Space." `https://landley.net/kdocs/ols/2003/ols2003-pages-494-506.pdf`.