Faust

Failure-Atomic User-Space Transactions

Eric Lee

ericlee123@utexas.edu

Advised by:
Vijay Chidambaram

Department of Computer Science

The University of Texas at Austin

Abstract

System failures, like those resulting from hardware defects or software crashes, can interrupt filesystem operations and leave persistent storage in corrupt states. Even though many popular filesystems, like ext4 and btrfs, provide some form of crash recovery, the recovered state can be semantically meaningless to user-level applications. Beyond that, there exist implementations that allow user-level applications to compose transactions for the filesystem, but all of them require nontrivial changes to the development environment.

Faust serves as a minimally invasive, system-call wrapper library that allows developers to easily wrap existing file I/O code in failure-atomic and durable transactions. Faust operates efficiently by only storing and persisting data when necessary. Benchmarks reveal that Faust incurs a practical overhead, within the range of 2-10x, for the transactional features it provides. Finally, we managed to easily include Faust in Vim's file-saving process as a show of Faust's usability.

Acknowledgments

I'd like to thank Vijay Chidambaram for advising me throughout the development of this project. He has been very useful in terms of availability and helping me refine my ideas regarding Faust. I would also like to thank Simon Peter for providing useful feedback on this report.

Contents

1 Introduction

When programs crash in the middle of performing filesystem operations, the filesystem state can exhibit many undesireable effects, such as a file being partially modified or written data being forgotten after loss of power. To combat this, people have turned to transactions that provide certain guarantees that alleviate the issues that arise in the event of system failure.

Transactional storage is not a novel idea. At the time of creating Faust, there already exist many persistent, transactional solutions that have been developed over the past few decades, such as SQLite, transactional NTFS, failure-atomic `msync()`, TxOS, and so on. However, none of these provide seamless integration for wrapping existing file I/O code within transactions. For example, most popular relational databases provide transactional support, but these databases do not have the same semantics as a POSIX filesystem. Other transactional solutions necessitate kernel modifications, which could be undesirable in the case when a system administrator wishes to keep up with the latest kernel updates for security purposes. Finally, some solutions simply have become outdated as demand for higher storage capacity and file sizes have scaled with time.

Filling the aforementioned void, Faust is a lightweight, POSIX-compliant, transactional library that focuses on simplicity and performance. So that developers do not have to learn and reason about the semantics of a new interface, Faust wraps around glibc storage I/O system calls. With Faust intercepting calls between the kernel and user-level applications, developers only need to mark where transactions begin and end in their existing code. The underlying implementation utilizes write-ahead and redo logging to provide a subset of ACID transactional properties. As an optimization, only data necessary to perform recovery is stored to reduce overhead whenever possible. It attempts to be minimally invasive yet efficient to offer developers an economic option for easily making existing filesystem operations transactional.

From our benchmarks, we estimate that using Faust transactions result in an overhead of around 2-10x, depending on the workload. More importantly, in line with simplicity, we managed to wrap Vim's file-saving code within a Faust transaction in just two lines, leaving the original codebase untouched.

This paper is organized as follows. Section 2 discusses background information regarding Faust. Section 3 details implementation and design decisions. Section 4 explains our testing methodology and performance benchmarks. Lastly, section 5 brings the report to a conclusion by discussing encountered problems and future extensions.

2 Background

To properly discuss the design and implementation of Faust, we must first cover relevant background knowledge. Faust provides failure-atomicity and durability from ACID, a commonly referenced set of properties for database transactions. Related work, such as ext4, SQLite, and TxOS will also be discussed.

2.1 ACID

ACID is an acronym for the set of properties ideally provided by a transaction. Taking a step back, a transaction is simply defined as a sequence of changes to some set of files. The four parts of ACID are described below:

- **Atomicity**: For a transaction to be atomic, the entire transaction should be executed as if it were a single operation. At any point in time, the filesystem needs to reflect one of two possible states, either before or after the transaction. The filesystem should never transition to an intermediate state where the transaction has only been partially applied.
- **Consistency**: Consistency means that the underlying storage will not be in an invalid state after having applied a transaction. This definition can vary based on storage type. In the context of filesystems, data structures, like inodes, data blocks, and bitmaps, cannot point to or contain nonsensical data.
- **Isolation**: Providing isolation implies that concurrent execution of multiple transactions is equivalent to as if they were executed serially.

In other words, transactions should be processed in a non-overlapping manner.

- **Durability**: This property states that once a transaction is complete, the effects of the transaction are durable, as in they can survive crashes, power outages, and other system errors.

Combining these properties with POSIX filesystem semantics, we designed Faust to provide failure atomicity and durability. The failure atomicity that Faust provides is different than the atomicity defined above. When Faust is applying the effects of a transaction onto underlying storage, the filesystem will still go through intermediate steps where only part of the transaction has occurred. As suggested by the name, Faust provides atomicity in the case of failure; if a crash were to occur at any point in the transaction, the filesystem would either remain unchanged or reflect a state where the entire transaction had been processed. For consistency, because Faust sits in user space, filesystem consistency is out of the scope of its control, so it must rely on the actual filesystem implementation. This is not an issue as most modern filesystems have some mechanism of maintaining consistency during filesystem operations. It could also be argued that Faust provides application-level consistency. Specifically, by wrapping transactions around certain strings of file I/O, a developer is marking what should be considered valid filesystem state. As for isolation, we developed Faust only in the context of single-threaded applications. To provide isolation for multithreaded usage would require all filesystem access to go through Faust. Without this guarantee, Faust would not be able to support isolation. Even with this guarantee, Faust would need to be redesigned to perform bookkeeping around simultaneous transactions.

2.2 POSIX Filesystem Atomicity

POSIX filesystems only offer atomicity in very limited cases. For example, writes are atomic at the granularity of the size of the pipe buffer (usually 4 kilobytes in Linux). Additionally, deletions and renames are also atomic. However, for the scope of an entire filesystem, without some modification to the kernel like Strata [5], it is not possible to have a large string of filesystem operations be applied atomically.

A very common and basic pattern for modifying the contents of a file is shown below.

```
char *data[65536];
int fd = open("out.txt", O_CREAT | O_RDWR , 0644);
write(fd, data, sizeof(data));
close(fd);
```

Figure 2.1: Common file access pattern

If a crash occurs at any point in this code, the underlying storage can be left in a corrupt or inconsistent state. Usually, filesystems have some mechanism to restore these data structures to a valid state, but even after recovery, the files and their contents could be semantically meaningless to the user-level application that was modifying them to begin with.

Imagine the scenario where some image manipulation software is persisting a set of changes to an image file, sitting in storage. Because the `write()` system call is not atomic, the image file must go through some transition states before arriving at its final, intended state. To extend this example, shown below in figure 2.2, imagine that a user wishes to edit an image of his cat by changing the colors of its fur. In his image editing software, he has painted the paws and mane burnt orange, in true UT fashion. After he is done making his changes, he wishes to save his modifications to the original image. Right after he clicks "save", the very same cat from the image disconnects the power supply to his desktop. Upon rebooting, the operating system on his computer performs some sort of recovery algorithm on the filesystem. He opens the image to find that it has weird, unintentional artifacts, of which he is not skilled enough to fix. As that was the only copy of his image, the memory is forever ruined. In this case, if the data saving code was wrapped within a Faust transaction, after recovery, the image would either be untouched, or the cat would have orange fur.

(a) Original cat photo

(b) Edited cat photo

Figure 2.2: Cat photo: before and after

(a) Acceptable corrupted photo

(b) Unacceptable corrupted photo

Figure 2.3: Possible image states after crash

The key point here is that even though the user-level application, the image manipulation software, is aware of how the underlying data, RGB and alpha values, is formatted, the kernel is not. As a result, desipte the fact that the recovered state of the image file is consistent in the context of the kernel, it is corrupt from the perspective of the user.

2.3 Existing Solutions

As stated above, there already exist many solutions that allow applications to transactionally persist data to storage; however, they are not all the same in terms of features and use cases. We will discuss some of the more popular solutions to highlight their differences.

2.3.1 ext4

The first and most obvious difference between Faust and ext4 [1] is that ext4 is an entire filesystem implementation and Faust is a library. ext4 provides low-level consistency by processing every modification as a transaction. Each transaction first logs changes to inodes, bitmaps, and data blocks before actually applying the modifications. ty[2] Similar to the problem highlighted by the image manipulation example, this level of consistency is only useful to the kernel and not user space.

2.3.2 SQLite

SQLite [9], under the hood, is a relational database management system. Although Faust and SQLite are both libraries that provide transactions, the structure of the underlying data is very different. One consequence of this disparity is that the interfaces to both libraries are also different. Faust complies with the POSIX definition for a filesystem while SQLite parses SQL statements to process data. Because of this, SQLite forgoes POSIX filesystem semantics, such as the hierarchical structure of directories and files. The utility of the tradeoff is largely dependent on the needs of an application and the experience of the developer.

2.3.3 Failure-atomic `msync()`

Failure-atomic `msync()` [7] is a modification to the `mmap()` and `msync()` use case. By passing a flag, `MAP_ATOMIC`, while memory mapping a file, failure-atomic `msync()` is able to offer similar features to Faust when it comes to crash recovery. When `msync()` is called on a memory-mapped file and a crash occurs, failure-atomic `msync()` provides the same failure atomicity around the `msync()` call that Faust does around its transactions. One big difference between the two transactional solutions is that

failure-atomic `msync()` is bound to the granularity of one file, while Faust can wrap any arbitrary string of filesystem operations within a transaction.

2.3.4 TxOS

TxOS [8] is a variant to the Linux kernel in the form of three system calls that allow for full ACID transactions on any operating system resource. Given that the filesystem falls under this category, users should be able to transactionally modify persistent storage. Additionally, because it is a kernel modification, its performance is indistinguishable from Linux in certain cases.

When it comes to providing transactions around a resource modified by the kernel, it makes a lot of sense to implement a solution within the kernel as opposed to user space to reduce the overhead of system calls. However, installing a custom kernel may not always be suitable in a development environment. Possibly for security or performance reasons, a system administrator may wish to keep up with the latest kernel versions, and it may not always be trivial to include the same modifications as parts of the kernel may change over time.

2.3.5 Transactional NTFS

Transactional NTFS [6] was developed by Microsoft to provide transactions for filesystem modifications. It was utilized in parts of the operating system where filesystem integrity is crucial, like installing software updates or performing system restores. Despite being developed by a company like Microsoft, they themselves have encouraged its users to find alternatives for the near future, quoting reasons such as "limited developer interest" and "complexity and various nuances which developers need to consider".

Taking lessons from all of these transactional solutions, we made our design decisions for Faust with usability as the strongest priority and performance as a close second.

3 Implementation

3.1 Goals and Assumptions

The main goal of Faust was to provide a unique yet practical combination of simplicity and performance. We placed a stronger emphasis on simplicity because we believed the existing transactional solutions were too invasive. Ideally, a developer should be able to identify a block of file I/O code and simply wrap it with `begin_txn()` and `end_txn()`. The logic and correctness of the existing code should remain unaffected. With that in mind, we designed Faust to be as space-efficient and performant as possible.

To be clear, one assumption we made was that Faust was designed for POSIX filesystems. When wrapping glibc system calls, we made sure that the behavior defined by POSIX stayed the same. One benefit of aligning with the POSIX standard is that, compared to solutions that modify the kernel or are entire filesystem implementations themselves, Faust will age better over time. The idea is that as the scale of computing grows, such as demand for larger files, more processing power, etc., implementations may need to be redesigned to keep up, while standards, ideally, do not. Another benefit is that developers who wish to utilize Faust would not need to reason about a new interface; their pre-existing knowledge about their current development environment should be sufficient. Finally, another assumption we made was that all filesystem operations would go through wrapped glibc calls. If files are handled using other file I/O libraries, then the control is out of the scope of Faust and the behavior is undefined.

3.2 Design

To achieve our goal of simplicity, we decided to place Faust in between user space and the kernel, so that it could intercept system calls and have the ability to log filesystem operations and redirect when necessary. The idea was that for each intercepted system call, Faust would be able to complete the required bookkeeping so that failure-atomicity could be provided in the case of a crash. A simple visualization is shown below in figure 3.1.

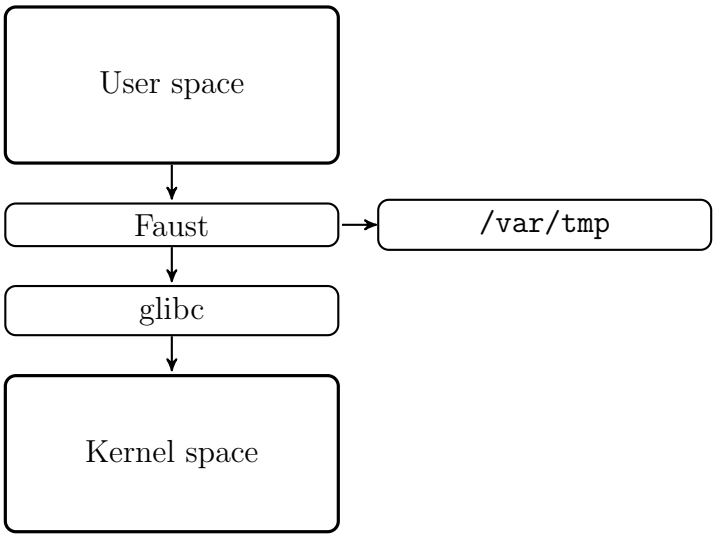

Figure 3.1: Position of Faust between user and kernel space

3.2.1 Undo vs. redo logging

Undo and redo logging are similar algorithms for performing recovery around transactions. In short, as a transaction proceeds, information is logged for each step that would allow the operation to be undone or redone, depending on implementation.

The first and most important decision we had to make was whether to implement undo or redo logging. In key-value databases, the two implementations are symmetric, but in the case of a filesystem, the nuances and the hierarchical structure bring out some rather important distinctions.

The main difference between undo and redo logging is whether or not the filesystem changes during the transaction, before the commit. For undo logging, because the filesystem reflects operations in the transaction as it is processed, operations

Undo logging	Redo logging
Must ensure durability with each system call	Only need to persist data on commit
Do not need to handle filesystem semantics	Need to mirror filesystem behavior and logic
Filesystem state changes along with transaction	Files and directories do not actually change during transaction
Easier implementation	More complex implementation
Slow	Fast

Table 3.1: Undo vs. redo logging distinctions

must first be durably logged before any change occurs. This is called write-ahead logging. [2] That way, if a crash occurs, Faust has the ability to restore the filesystem to its original point before the transaction by undoing any operation that could have been executed before the crash. As for redo logging, because Faust would only know how to move forward in the transaction, the filesystem cannot change until all operations are logged and the transaction is committed. At that point, the log would first be persisted, to ensure durability, and then the transaction would be applied after. This difference creates a performance vs. programmability tradeoff. Because undo logging requires durability for every system call, overhead compared to not using a transaction is around 1000x due to the disparity between writing to disk versus memory. The benefit is that Faust would not have to handle any filesystem logic, as the filesystem is actually modified as the transaction proceeds. For redo logging, because the filesystem does not actually change until commit, persisting the log is not necessary at every step. Durability is still provided, but its cost is amortized across the entire transaction. Therefore, the performance of redo logging is much closer to transactionless file I/O, around 2x-10x. Again, the tradeoff is now that Faust has to provide the abstraction that the underlying filesystem is changing with the transaction, meaning it must handle and mirror filesystem state.

Given this tradeoff, we decided to go with redo logging, as paying 1000x overhead is too impractical an expense for making file I/O transactional.

3.2.2 Wrapping glibc system calls

To begin, we wrapped Faust around the some basic file I/O system calls, such as `open()`, `mkdir()`, `rename()`, `remove()`, `write()`, and `ftruncate()`. A combination of `LD_PRELOAD`, `LD_LIBRARY_PATH`, and `dlsym()` allowed us to wrap any system call we deemed necessary. For each wrapper, the control flow follows the same general pattern.

1. Perform recovery if necessary.
2. Check if current function was called within transaction.
   (a) If so, log necessary data for redoing operation and modify underlying data structures to reflect the effects of the call.
   (b) If not, simply call the corresponding glibc function.

The data logged at each point varies depending on the system call. For example, for `open()` with the `O_CREAT` flag, Faust simply tracks the path of the new file and its permissions. For slightly more complex system calls like `write()`, Faust logs the path, position, total data written, and the location of the redirected data. In general, with each system call, the underlying data structures should be modified in a way that would reflect the filesystem state as if it were actually modified.

3.2.3 Mirroring filesystem semantics

We decided to wrap Faust around system calls that used file descriptors, such as `read()`, `write()`, `ftruncate()`, etc., as these are the building blocks for all filesystem operations. To match how the kernel handles file descriptors, we created two structs, `file_desc` and `vfile`. A `vfile` would be created for each file touched within a transaction. The idea is to keep track of how a file has changed in memory, so that the actual file does not need to be modified. Each `vfile` struct points to a corresponding file in `/var/tmp` in which all redirected writes go to. For `file_desc`, we wanted to create a layer of indirection for file descriptors between user-level applications and the kernel.

In `vfile`, the `path` field represents where the file "virtually" exists. For example, if a file is renamed within a transaction, the `path` field is updated to reflect

```
struct file_desc {
    int redirect_fd;
    struct vfile *file;
};
```

```
struct vfile {
    char path[4096+1];
    char src[4096+1];
    char redirect[4096+1];
    struct range *writes;
};
```

Table 3.2: Implementation of `file_desc` and `vfile`

it. `src` points to where the original file sits in the actual filesystem. This is necessary for `read` operations to return original data. `redirect` contains the path for the redirected file in `/var/tmp`. Finally, `writes` is a list of ranges that indicate between which indices of the file contain newly written data. This is necessary for `read` to reflect the latest `write`. Just like with normal file descriptors, multiple `file_desc` structs can point to the same `vfile`.

3.2.4 Transactions from beginning to end

A transaction begins with a call to `begin_txn()`. Until `end_txn()`, all file I/O calls wrapped by Faust are intercepted and redirected away from their actual destinations. Once `end_txn()` is called, Faust first persists all logged data, including redirected writes and the redo log itself. At the very end of persisting all data, Faust writes "commit" to the end of the redo log. It is important to persist all necessary data before executing the transaction as specified by write-ahead logging. Once the transaction is committed, the log is replayed from the beginning. After the redo log has been completely replayed and the touched files and directories have been persisted, the redo log is deleted to indicate that transaction has been durably processed.

Utilizing a Faust transaction should not result in a different program state after the transaction is finished. In other words, data buffers, return values, validity of file descriptors, etc., should be equivalent to as if no transaction had been used at all.

3.2.5 Crash and recovery

For each wrapped system call, before anything is performed, Faust checks to see whether the filesystem needs to be recovered. A recovery is necessary if a redo log

exists and the last line is a commit message. There are three possible states that could be observed when checking for recovery:

1. **A redo log does not exist.** This could imply that a transaction had been fully committed and executed, resulting in the deletion of the redo log. It is also possible that, up to this point, no transactions have been utilized. In any case, there is nothing to recover.

2. **A redo log exists, but the last line is not "commit".** This could result from two situations.

   (a) If check is being performed within a transaction, then it simply implies that Faust is in the middle of a transaction, building up the redo log.

   (b) If Faust is not within a transaction, then the system must have crashed during a transaction but before the commit. Even so, there is nothing to recover as the filesystem is not modified until a transaction is committed. Failure-atomicity is still provided.

3. **A redo log exists, and its last line is "commit".** This implies that a crash occurred during a transaction, specifically after the transaction had been commmitted but before the redo log had finished replaying. Now, Faust must perform recovery as the filesystem may be in an inconsistent state.

To perform recovery, Faust takes the same steps as if it had just completed committing a transaction. The redo log is replayed, and upon finishing, it is deleted.

In the process of replaying the redo log, each log entry begins with the operation name, `mkdir`, `create`, `write`, `remove`, `rename`, and `truncate`, followed by parameters for the corresponding operation. As a side note, Faust only needs to log information for system calls that modify persistent filesystem state, so functions such as `read`, `lseek`, and `fstat` were excluded. It is important to note that application of the redo log must be idempotent, meaning repeated replays of the redo log will always result in the same filesystem state. This is because, if a system crashes after only replaying half of the log, replaying the same half again on recovery should not result in a different state than if the system had not crashed to begin with. On the topic of idempotence, it is not sufficient for an individual recovery operation to be idempotent, but the log as a whole must satisfy this property. We did not run into

this issue for redo logging, but for undo logging, there were cases where backed up data was lost after crashing during certain parts of the recovery.

3.2.6 Optimizations

Although redo logging could be considered a very significant optimization from undo logging, there were some additional steps we took for Faust to scale better and gain more performance. All of these optimizations were applied on top of a very naïve implementation of redo logging. In implementing this naïve version, we made decisions that prioritized ease of programmability, such as open and closing files whenever needed, using linked lists, etc. The one optimization we did keep in this vanilla optimization was to avoid calling `fsync()` each time an entry was applied from the redo log. Excluding this from redo logging would result in the same overhead as undo logging as an `fsync()` call would accompany each system call. The optimizations we made and their reasons are listed below:

- Writing to the redo log happens very frequently. At first, upon writing an entry, Faust would open, write, then close the redo log. Instead, we simply kept an open file descriptor corresponding to the redo log for the duration of the transaction. This was significant because, in certain cases, `open()` and `write()` have the same order of magnitude in terms of runtime.

- At the beginning of every system call, a check is made to see whether recovery needs to be performed. Initially, we simply called `open()` on the redo log and checked whether the returned file descriptor was valid. This was very inefficent as any call, within a transaction or not, to a function wrapped by Faust would have to bear the cost of an `open()` system call. Instead, we optimized Faust to first check whether the file descriptor corresponding to the redo log is valid. As a secondary check, we then use the `access()` system call, which is significantly faster than `open()`, to check for the existence of the redo log before proceeding.

- All files touched within a transaction have corresponding files within `/var/tmp` that hold buffered data. Similar to log writing, instead of just opening and closing the redirect files as they were written to,

we changed the implementation to keep them open until `close()` was called. This change also simplified also our implementation of mirroring filesystem semantics. At first, within the `file_desc` struct, we planned on adding fields to track all properties of file descriptors within the kernel. Now, the only field needed is a file descriptor. All semantics, like position and permissions, are nicely handled by the kernel itself.

- Like the optimization above, we created a file descriptor cache to be used when replaying the redo log. We did not want to pay the cost of an `open()` everytime a file needed to be written to. This helps when a redo log has multiple operations changing the same file.

- The last major optimization we had was related to storing `vfile` structs. For some system calls, the existence of a file needed to be checked before continuing. For example, in a transaction, a user can create a file and then write to it. It is not sufficient to check the actual filesystem, as the file does not yet exist, so we must check our own data structures. Using linked lists, though easy to program, scales very poorly. We used sets [3] instead, for efficient lookup. This optimization allows Faust to scale better when a large number of files are touched within one transaction.

The results of these optimizations are shown in the benchmarking section later in the report.

4 Results

4.1 Testing for Correctness

The testing framework we created for Faust validates failure-atomicity through the use of random crashes and multiple iterations. The multiple layers of a single test are described below:

1. A test is composed of multiple transactions to be executed one after another. The initial filesystem state simply contains a folder and an empty file. Throughout a test, three folders are updated as transactions are processed: `before/`, `after/`, and `txn/`. `before/` represents the state of the filesystem before the a transaction is processed; `after/` represents the result of a transaction; lastly, `txn/` is the working directory to where the transaction is applied.

    (a) A transaction is randomly generated from a set of filesystem operations, including creating a new file/directory, removing, writing, truncating, and renaming.

    (b) Using the generated transaction, `after/` is generated by applying the transaction to a copy of `before/`.

    (c) Next, a redo log is generated for `txn/`.

    (d) With everything set in place, a child process, whose purpose is to solely apply the redo log, is spawned and killed by the parent process for a random number of iterations.

        i. After each time the child process is killed, the redo log is applied once again, and `txn/` is then compared to `after/` to check for any disparities that would indicate a failure to provide failure-atomicity.

ii. After the check succeeds, `txn/` is reset to the state of before the transaction by copying from `before/`.

As we were developing Faust, running this test framework for a few hours, usually overnight, would either convince us of the correctness of our implementation or reveal a bug. Additionally, we used `srand()` so that failed test runs could be replicated during debugging.

4.2 Benchmarks

To properly show the effects of our design choices and optimizations, we ran benchmarks on our implementations of undo logging, redo logging, and optimized redo logging. All tests were run on a i7-7700HQ processor paired with a 512 GB NVMe SSD. Between each iteration of each test, we ran `sync` followed by `echo 3 > /proc/sys/vm-/drop_caches` to reset the page cache in an attempt to obtain consistent results across runs.

Figure 4.1 shows the results of a benchmark that tested how `redo()` runtimes scaled with the amount of data written in a transaction. This test would first write a certain amount of data to a file, save the generated redo log, delete the resulting file, and then call `redo()` and measure its runtime. For some reason, the optimized version of redo logging consistently performed worse when the write size exceeded 1 GB (2^{30} bytes). We managed to determine that the culprit was the file descriptor cache maintained during `redo()`; however, the cause of the behavior is still unclear and needs further investigation.

Another test we performed involved combining fsx and Faust. fsx, short for "file system exerciser", is a commonly used tool for stress testing and validating filesystems. [4] To test Faust, we wrapped the testing loop within a transaction and configured fsx to only use write and truncate operations. Within each test, each operation was performed 10000 times. Table 4.1 shows the measured overhead of performing fsx tests within a transaction. The results here are as expected. Both versions of redo logging are significantly faster than undo logging, and optimized redo logging is slightly faster than our naïve implementation.

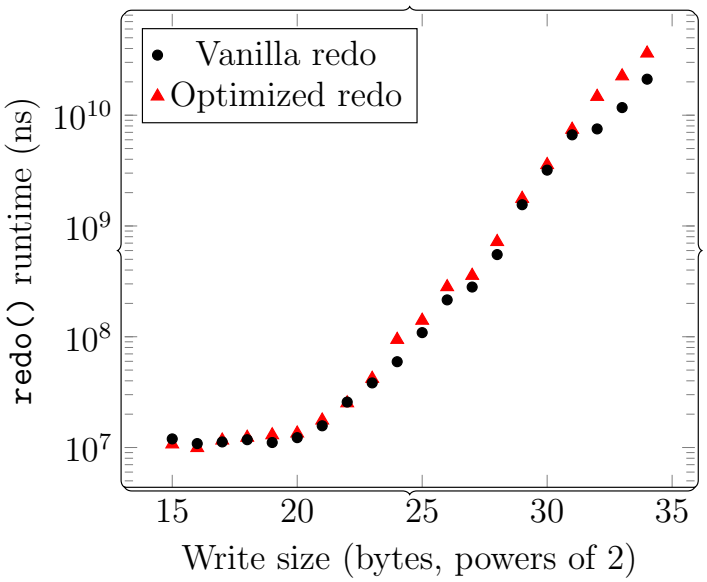

Figure 4.1: Comparison of `redo()` runtimes for increasing write sizes

	Writes	Truncates
Undo	2.64x	24.71x
Redo	1.30x	2.72x
Optimized redo	1.11x	1.84x

Table 4.1: Ratio of runtimes for transactional fsx test to transactionless fsx test (operations: 10000)

To provide some comparison against other transactional solutions, we created a benchmark that compares writing an equal amount of data in both Faust and SQLite. In SQLite, our test would write a set amount of data, 16 MB in our case, in the form of reals, 8 bytes each, inside a transaction. In Faust, our test would write an equal amount of data, also 8 bytes at a time. Within the Faust workload, we also tried writing data 16 bytes at a time while halving the number of `write()` calls. This resulted in optimized redo logging being faster than SQLite for our 16 MB workload. The runtime of a Faust transaction over an equivalent SQLite transaction is shown in table 4.2.

	8B	16B
Redo	2.65x	1.38x
Optimized redo	1.54x	0.78x

Table 4.2: Comparison to equivalent SQLite workloads (size: 16MB)

We also ran benchmarks on the Faust endpoint `remove()`. We decided upon `remove()` as it is a relatively simple and lightweight wrapper in terms of bookkeeping. For each test, we would create 20,000 files or directories, and then delete them all within a transaction. As seen in table 4.3, there is a significant performance increase from naïve to optimized redo logging. This is due the use of a different data structure to store `vfile` structs. In unoptimized redo logging, a simple linked list is used, which resulted in slower lookup times as the number of files touched within a transaction increased. The set in optimized redo logging provided much faster lookup, and therefore, a much smaller overhead.

	File	Directory
Undo	1401.71x	868.38x
Redo	54.78x	9.58x
Optimized redo	5.57x	2.02x

Table 4.3: `remove()` overhead benchmarks (count: 20000)

For obvious reasons, we also ran benchmarks on the `write()` endpoint, to gauge Faust's throughput. First, we tested all three implementations. Within each implementation, a transaction containing, 1, 10, or 100 writes would be compared to an equal amount of in-memory writes. The transaction would also be compared to an equal amount of writes followed by an `fsync()` for durability. Whether a file was being appended to or overwritten was also tested as a variable. Each test was run for 1000 iterations, and all writes were 4 KB of data. Overheads of the Faust transactions compared to in-memory and durable writes are shown in table 4.4. Some key differences to highlight are all transactions are much slower than their in-memory counterparts. This is because durability requires disk access, which is around three orders of magnitude slower than memory. Also, the overhead for undo logging increases with longer transactions because each call requires durability, which is another call to `fsync()`.

Undo	In-memory		append	overwrite
		1	6299.77x	7144.49
		10	5763.58x	7446.84x
		100	5801.48x	6177.40x
	Ending `fsync()`		append	overwrite
		1	6.47x	3.99x
		10	27.51x	23.10x
		100	193.23x	200.64x
Redo	In-memory		append	overwrite
		1	2444.92x	2840.84x
		10	865.79x	1158.04x
		100	120.28x	139.72x
	Ending `fsync()`		append	overwrite
		1	3.93x	6.52x
		10	3.90x	20.76x
		100	3.13x	6.29x
Optimized redo	In-memory		append	overwrite
		1	2547.82x	3263.42x
		10	1007.70x	1180.03x
		100	129.96x	131.26x
	Ending `fsync()`		append	overwrite
		1	3.93x	5.80x
		10	4.55x	6.85x
		100	3.13x	5.93x

Table 4.4: `write()` overhead benchmarks (count: 1000, size: 4KB)

Lastly, we wanted to compare Faust to a very simple alternative for preventing file corruption. In the process of saving a file, vim, like other many popular text editors, makes a copy of the working file, applies the changes to the copy, and then renames the copy to the original. This method has many benefits. For example, it is very easy to understand and implement. Also, its failure case is tolerable to most users. If a crash were to occur in the middle of saving, only the copied file would be corrupted. Users can simply delete the corrupted file and continue working. Fortunately for us, this method becomes less practical with larger files as the time of copying a file becomes longer. Figure 4.2 shows the results of our comparisons. Each test was run for 100 iterations. The green dotted line represents when the two methods, Faust and swapping, have equal runtimes. For redo logging, the two

methods have the same runtime around a file size of 2 to 4 MB, and for undo logging, the point is around 32 MB. We believe the crossover point could exist at smaller file sizes with better optimizations for Faust.

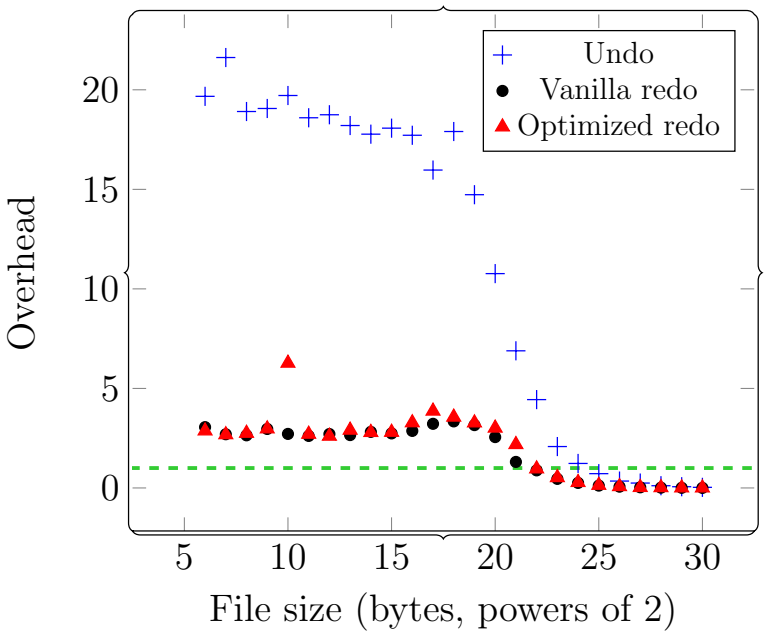

Figure 4.2: Comparison of Faust and copy-write-then-rename (count: 100)

For all of the benchmarks we ran, redo logging has an explicit advantage over undo logging. However, when it comes to the effects of our optimizations, the variance across runs makes evaluation difficult. In the end, we believe the best optimization for Faust would be those that reduced the number of `fsync()` calls within a transaction because the expense of an `fsync()` call dominate most other bookkeeping operations within Faust.

One way to reduce the number of `fsync()` calls would be to merge the redo log and the redirected data into one file. In our optimized implementation, redirected data sits in many separate files, one for each file touched within a transaction. In order to commit a transaction, every redirect file needs to be persisted, resulting in many calls to `fsync()`. Merging redirected data with the redo log would result in only one call to `fsync()` on commit. Unfortunately, we did not have time to experiment with this optimization.

4.3 Porting Vim

As stated previously, our first priority in designing Faust was its usability. As proof of such, we decided upon making vim, a very popular command-line text editor, transactional. By default, when vim saves a file, it first makes a copy of the current file, applies the changes to the copy, and then renames the copy to replace the original. This process prevents corruption in the original file, as renames are atomic. If the saving process were to be interrupted, only the copy would be in a corrupt state. As shown in our benchmarks, one downside to this method of saving is that it does not scale well with file size. With larger file sizes, the cost of copying the original becomes impractical, especially in the case where only a small fraction of the file is actually modified. This mechanism, in addition to vim's highly portable, well-estabilished codebase, makes vim a suitable candidate for the features that Faust provides.

In short, we managed to port vim in 2 lines, one for beginning the transaction and one for ending it. We also had to add `set noswapfile`, `set backupcopy=yes`, and `set backupdir=''` to `.vimrc` in order to configure vim to modify the working file in place. It should be noted that we were not able to wrap the full `:write` command within a transaction because, at the time of writing this report, Faust does not wrap enough glibc functions nor does it account for certain behavior, like calling `open()` on process numbers. Regardless, we believe our current port is sufficient proof of Faust's usability.

5 Conclusion

In conclusion, we believe that we accomplished all of the goals we had set at the beginning of developing Faust. We managed to provide Faust in a non-invasive manner, as it is easily used as a shared library. We made a convincing argument for Faust's guarantee of failure atomicity by running our test framework for extended periods of time. We also maintain a reasonable overhead in comparison to similar solutions. Finally, and most importantly, the ease at which we ported vim shows that we stayed true to our first priority, usability. As a result, we believe there is a legitimate argument for the utility of Faust for a large range of I/O intensive applications.

5.1 Encountered Problems

Most of the problems that we encountered while developing Faust resulted from our position in the operating stack. Because Faust intercepts system calls between user space applications and the kernel, we were forced to work in C. Two main issues stemmed from this.

First, the lack of data structures from a C standard library was rather troublesome. As a result, we had to implement the set data structure ourselves. Undoubtedly, there exist better implementations of sets that handle collision and hashing much better than ours does. It would have been nice for the set data structures used within Faust to have better performance than the most naïve implementations.

Secondly, there were some instances where we were presented with two options, use some system call not available to C through `system()` or implement the entire system call from scratch. Using `system()` gave us weird behavior in terms of reentry and crashing. We suspected that `system()` performs a process fork where a new instance of Faust was copied with the new process. This caused some issues because Faust would then falsely detect a need to perform recovery. It should also be noted

that this behavior only occurred with certain calls to `system()`. We avoided this issue by setting up a bypass mechanism. We would create an empty file to use as global state. The existence of that file would indicate to Faust that recovery should explicilty be avoided, and wrapper functions should simply reroute to their corresponding system calls. Using `system()` also resulted in weird errors in our testing framework. Our best guess is that `system()` is not tolerant to random crashing. As an example, for undo logging, we needed to reverse the undo log as a step leading up to recovery. There is a simple Linux tool that does it in one line; however, using it would sometimes result in recovery failures in our testing framework. We wrote our own method to reverse a text file line by line and used that instead. Using our own method, the recovery failures no longer showed up.

The final main issue we ran into was obtaining consistent benchmarks. Running some benchmark multiple times would show a degree of variance. This could be due to background activity of other applications or possibly the operating system having slightly different load with each run. For example, in a benchmark measuring the runtime of multiple writes, we compared Faust to a string of writes followed by an `fsync()`. The reported runtimes would vary anywhere from 1,000,000 to 4,500,000 nanoseconds. This variance made it very difficult to evaluate of the effectiveness of our optimizations. Switching from undo logging to redo logging showed obvious benefits as the new overhead had a much smaller order of magnitude. However, for our small optimizations, we were not sure whether the observed differences were due to chance.

5.2 Future Work

The most obvious and useful next steps for Faust would be to increase its robustness. At the time of writing this, Faust only imitates underlying system calls for the most part. For example, we did not take into account permissions, error codes, and all the tricky edge cases that accompany the system calls wrapped by Faust. In addition to more closely mirroring filesystem behavior, it would also be beneficial for Faust to wrap system calls used for different data access patterns, such as `mmap()` and `msync()`. Handling these cases may call for a more general design in how Faust performs its bookkeeping on files within a transaction. All in all, increasing robustness would help increase the number of applications that could be easily ported.

Some other beneficial future work would be to consider Faust in the context of multiple threads. Usually in the context of combining transactions with multithreaded programming, correctness is not enough, as performance is equally important. Referring back to ACID, this would add isolation to the set of transaction guarantees provided by Faust.

Lastly, another way of improving Faust would be to reduce its overhead even further. At the moment, Faust still performs some repetitive work. For example, when a user attempts to create a file under a nonexistent directory, an entry is still written to the log. It might be better to maintain a tree to reflect filesystem state, so that faster, in-memory checks can be performed instead writing to the log. Though no optimization, to our knowledge, can result in a different order of magnitude for overhead, we still believe there exists room for better performance.

6 Appendix

Faust API		
Endpoint name	Method signature	Explanation
begin_txn	`int begin_txn()`	Begins a new transaction. Returns transaction id.
end_txn	`int end_txn(int txn_id)`	Ends a transaction.
redo	`int redo()`	Process the redo log if necessary.
rollback	`void rollback()`	Rolls back the effects of a transaction.
save_log	`void save_log(const char *dest)`	Saves redo log to destination after log is replayed.
delete_log	`void delete_log()`	Deletes log.
set_bypass	`void set_bypass(int set)`	Creates/deletes bypass file based on `set`.

Table 6.1: List of Faust endpoints

Wrapped glibc system calls	
open	`int open(const char pathname, int flags, ...)`
close	`int close(int fd)`
mkdir	`int mkdir(const char pathname, mode_t mode)`
rename	`int rename(const char oldpath, const char newpath)`
remove	`int remove(const char pathname)`
read	`ssize_t read(int fd, void buf, size_t count)`
write	`ssize_t write(int fd, const void buf, size_t count)`
ftruncate	`int ftruncate(int fd, off_t length)`
fstat	`int fstat(int fd, struct stat *statbuf)`
lseek	`off_t lseek(int fd, off_t offset, int whence)`

Table 6.2: List of glibc functions wrapped by Faust

References

[1] Ext4 filesystem. URL: https://www.kernel.org/doc/Documentation/filesystems/ext4.txt.

[2] Remzi H. Arpaci-Dusseau and Andrea C. Arpaci-Dusseau. *Operating Systems: Three Easy Pieces*, chapter 42. Arpaci-Dusseau Books, 0.90 edition, 2015.

[3] Dan Bernstein. djb2. URL: http://www.cse.yorku.ca/~oz/hash.html.

[4] Dave Jones. File system exerciser. URL: http://codemonkey.org.uk/projects/fsx/.

[5] Youngjin Kwon, Henrique Fingler, Tyler Hunt, Simon Peter, Emmett Witchel, and Thomas Anderson. Strata: A cross media file system, 2017. URL: https://www.cs.utexas.edu/~simon/sosp17-final207.pdf.

[6] Microsoft. Transactional NTFS. URL: https://msdn.microsoft.com/en-us/library/windows/desktop/aa363764(v=vs.85).aspx.

[7] Stan Park, Terrence Kelly, and Kai Shen. Failure-atomic msync(): A simple and efficient mechanism for preserving the integrity of durable data. URL: https://www.cs.rochester.edu/u/kshen/papers/eurosys2013.pdf.

[8] Donald Porter, Owen Hoffman, Christopher Rossbach, Alexander Benn, and Emmett Witchel. Operating system transactions, 2009. URL: http://www.cs.utexas.edu/~porterde/pubs/sosp063-porter.pdf.

[9] SQLite, 2000. URL: https://www.sqlite.org/index.html.