Rethink the Sync

Edmund B. Nightingale, Kaushik Veeraraghavan, Peter M. Chen, and Jason Flinn
Department of Electrical Engineering and Computer Science
University of Michigan

Abstract

We introduce *external synchrony*, a new model for local file I/O that provides the reliability and simplicity of synchronous I/O, yet also closely approximates the performance of asynchronous I/O. An external observer cannot distinguish the output of a computer with an externally synchronous file system from the output of a computer with a synchronous file system. No application modification is required to use an externally synchronous file system: in fact, application developers can program to the simpler synchronous I/O abstraction and still receive excellent performance. We have implemented an externally synchronous file system for Linux, called xsyncfs. Xsyncfs provides the same durability and ordering guarantees as those provided by a *synchronously* mounted ext3 file system. Yet, even for I/O-intensive benchmarks, xsyncfs performance is within 7% of ext3 mounted *asynchronously*. Compared to ext3 mounted synchronously, xsyncfs is up to two orders of magnitude faster.

1 Introduction

File systems serve two opposing masters: durability and performance. The tension between these goals has led to two distinct models of file I/O: synchronous and asynchronous.

A synchronous file system (e.g., one mounted with the `sync` option on a Linux system) guarantees durability by blocking the calling application until modifications are committed to disk. Synchronous I/O provides a clean abstraction to users. Any file system operation they observe to complete is durable — data will not be lost due to a subsequent OS crash or power failure. Synchronous I/O also guarantees the ordering of modifications; if one operation causally precedes another, the effects of the second operation are never visible unless the effects of first operation are also visible. Unfortunately, synchronous I/O can be very slow because applications frequently block waiting for mechanical disk operations. In fact, our results show that blocking due to synchronous I/O can degrade the performance of disk-intensive benchmarks by two orders of magnitude.

In contrast, an asynchronous file system does not block the calling application, so modifications are typically committed to disk long after the call completes. This is fast, but not safe. Users view output that depends on uncommitted modifications. If the system crashes or loses power before those modifications commit, the output observed by the user was invalid. Asynchronous I/O also complicates applications that require durability or ordering guarantees. Programmers must insert explicit synchronization operations such as `fsync` to enforce the guarantees required by their applications. They must sometimes implement complex group commit strategies to achieve reasonable performance. Despite the poor guarantees provided to users and programmers, most local file systems provide an asynchronous I/O abstraction by default because synchronous I/O is simply too slow.

The tension between durability and performance leads to surprising behavior. For instance, on most desktop operating systems, even executing an explicit synchronization command such as `fsync` does *not* protect against data loss in the event of a power failure [13]. This behavior is not a bug, but rather a conscious design decision to sacrifice durability for performance [27]. For example, on `fsync`, the Linux 2.4 kernel commits data to the volatile hard drive cache rather than to the disk platter. If a power failure occurs, the data in the drive cache is lost. Because of this behavior, applications that require stronger durability guarantees, such as the MySQL database, recommend disabling the drive cache [15]. While MacOS X and the Linux 2.6 kernel provide mechanisms to explicitly flush the drive cache, these mechanisms are not enabled by default due to the severe performance degradation they can cause.

We show that a new model of local file I/O, which we term *external synchrony*, resolves the tension between durability and performance. External synchrony provides the reliability and simplicity of synchronous I/O, while closely approaching the performance of asynchronous I/O. In external synchrony, we view the abstraction of synchronous I/O as a set of guarantees that are provided to the clients of the file system. In con-

trast to asynchronous I/O, which improves performance by substantially weakening these guarantees, externally synchronous I/O provides the same guarantees, but it changes the clients to which the guarantees are provided.

Synchronous I/O reflects the *application-centric* view of modern operating systems. The return of a synchronous file system call guarantees durability to the application since the calling process is blocked until modifications commit. In contrast, externally synchronous I/O takes a *user-centric* view in which it guarantees durability not to the application, but to any external entity that observes application output. An externally synchronous system returns control to the application before committing data. However, it subsequently buffers all output that causally depends on the uncommitted modification. Buffered output is only externalized (sent to the screen, network, or other external device) after the modification commits.

From the viewpoint of an external observer such as a user or an application running on another computer, the guarantees provided by externally synchronous I/O are identical to the guarantees provided by a traditional file system mounted synchronously. An external observer never sees output that depends on uncommitted modifications. Since external synchrony commits modifications to disk in the order they are generated by applications, an external observer will not see a modification unless all other modifications that causally precede that modification are also visible. However, because externally synchronous I/O rarely blocks applications, its performance approaches that of asynchronous I/O.

Our externally synchronous Linux file system, xsyncfs, uses mechanisms developed as part of the Speculator project [17]. When a process performs a synchronous I/O operation, xsyncfs validates the operation, adds the modifications to a file system transaction, and returns control to the calling process without waiting for the transaction to commit. However, xsyncfs also taints the calling process with a *commit dependency* that specifies that the process is not allowed to externalize any output until the transaction commits. If the process writes to the network, screen, or other external device, its output is buffered by the operating system. The buffered output is released only after all disk transactions on which the output depends commit. If a process with commit dependencies interacts with another process on the same computer through IPC such as pipes, the file cache, or shared memory, the other process inherits those dependencies so that it also cannot externalize output until the transaction commits. The performance of xsyncfs is generally quite good since applications can perform computation and initiate further I/O operations while waiting for a transaction to commit. In most cases, output is delayed by no more than the time to commit a single transaction — this is typically less than the perception threshold of a human user.

Xsyncfs uses *output-triggered commits* to balance throughput and latency. Output-triggered commits track the causal relationship between external output and file system modifications to decide when to commit data. Until some external output is produced that depends upon modified data, xsyncfs may delay committing data to optimize for throughput. However, once some output is buffered that depends upon an uncommitted modification, an immediate commit of that modification is triggered to minimize latency for any external observer.

Our results to date are very positive. For I/O intensive benchmarks such as Postmark and an Andrew-style build, the performance of xsyncfs is within 7% of the default asynchronous implementation of ext3. Compared to current implementations of synchronous I/O in the Linux kernel, external synchrony offers better performance *and* better reliability. Xsyncfs is up to an order of magnitude faster than the default version of ext3 mounted synchronously, which allows data to be lost on power failure because committed data may reside in the volatile hard drive cache. Xsyncfs is up to two orders of magnitude faster than a version of ext3 that guards against losing data on power failure. Xsyncfs sometimes even improves the performance of applications that do their own custom synchronization. Running on top of xsyncfs, the MySQL database executes a modified version of the TPC-C benchmark up to three times faster than when it runs on top of ext3 mounted asynchronously.

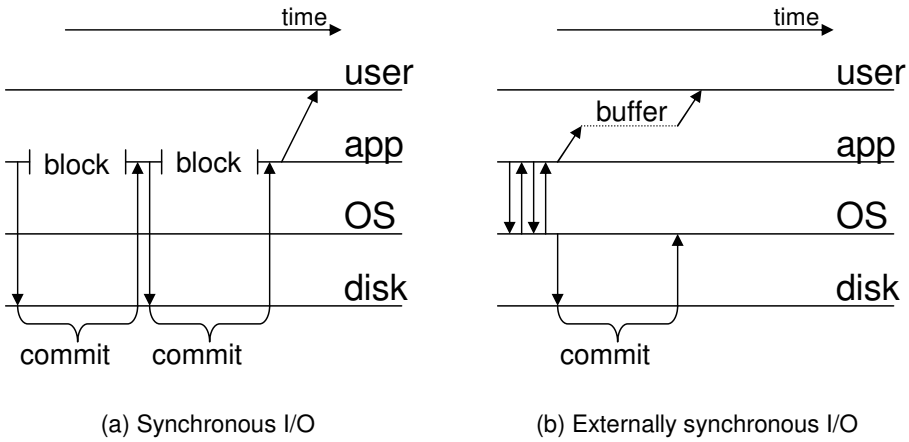
2 Design overview

2.1 Principles

The design of external synchrony is based on two principles. First, we define externally synchronous I/O by its externally observable behavior rather than by its implementation. Second, we note that application state is an internal property of the computer system. Since application state is not directly observable by external entities, the operating system need not treat changes to application state as an external output.

Synchronous I/O is usually defined by its implementation: an I/O is considered synchronous if the calling application is blocked until after the I/O completes [26]. In contrast, we define externally synchronous I/O by its observable behavior: we say that an I/O is externally synchronous if the external output produced by the computer system cannot be distinguished from output that could have been produced if the I/O had been synchronous.

The next step is to precisely define what is considered external output. Traditionally, the operating system takes



(a) Synchronous I/O

(b) Externally synchronous I/O

This figure shows the behavior of a sample application that makes two file system modifications, then displays output to an external device. The diagram on the left shows how the application executes when its file I/O is synchronous; the diagram on the right shows how it executes when its file I/O is externally synchronous.

Figure 1. Example of externally synchronous file I/O

an *application-centric* view of the computer system, in which it considers applications to be external entities observing its behavior. This view divides the computer system into two partitions: the kernel, which is considered internal state, and the user level, which is considered external state. Using this view, the return from a system call is considered an externally visible event.

However, users, not applications, are the true observers of the computer system. Application state is only visible through output sent to external devices such as the screen and network. By regarding application state as internal to the computer system, the operating system can take a *user-centric* view in which only output sent to an external device is considered externally visible. This view divides the computer system into three partitions, the kernel and applications, both of which are considered internal state, and the external interfaces, which are considered externally visible. Using this view, changes to application state, such as the return from a system call, are not considered externally visible events.

The operating system can implement user-centric guarantees because it controls access to external devices. Applications can only generate external events with the cooperation of the operating system. Applications must invoke this cooperation either directly by making a system call or indirectly by mapping an externally visible device.

2.2 Correctness

Figure 1 illustrates these principles by showing an example single-threaded application that makes two file system modifications and writes some output to the screen. In the diagram on the left, the file modifications made by the application are synchronous. Thus, the application blocks until each modification commits.

We say that external output of an externally synchronous system is equivalent to the output of a synchronous one if (a) the values of the external outputs are the same, and (b) the outputs occur in the same causal order, as defined by Lamport's *happens before* relation [9]. We consider disk commits external output because they change the stable image of the file system. If the system crashes and reboots, the change to the stable image is visible. Since the operating system cannot control when crashes occur, it must treat disk commits as external output. Thus, in Figure 1(a), there are three external outputs: the two commits and the message displayed on the screen.

An externally synchronous file I/O returns the same result to applications that would have been returned by a synchronous I/O. The file system does all processing that would be done for a synchronous I/O, including validation and changing the volatile (in-memory) state of the file system, except that it does not actually commit the modification to disk before returning. Because the results that an application sees from an externally synchronous I/O are equivalent to the results it would have seen if the I/O had been synchronous, the external output it produces is the same in both cases.

An operating system that supports external synchrony must ensure that external output occurs in the same causal order that would have occurred had I/O been performed synchronously. Specifically, if an external output causally follows an externally synchronous file I/O, then that output cannot be observed before the file I/O has been committed to disk. In the example, this means that the second file modification made by the application cannot commit before the first, and that the screen output cannot be seen before both modifications commit.

2.3 Improving performance

The externally synchronous system in Figure 1(b) makes two optimizations to improve performance. First, the two modifications are group committed as a single file system transaction. Because the commit is atomic, the effects of the second modification are never seen unless the effects of the first are also visible. Grouping multiple modifications into one transaction has many benefits: the commit of all modifications is done with a single sequential disk write, writes to the same disk block are coalesced in the log, and no blocks are written to disk at all if data writes are closely followed by deletion. For example, ext3 employs value logging — when a transaction commits, only the latest version of each block is written to the journal. If a temporary file is created and deleted within a single transaction, none of its blocks are written to disk. In contrast, a synchronous file system cannot group multiple modifications for a single-threaded application because the application does not begin the second modification until after the first commits.

The second optimization is buffering screen output. The operating system must delay the externalization of the output until after the commit of the file modifications to obey the causal ordering constraint of externally synchronous I/O. One way to enforce this ordering would be to block the application when it initiates external output. However, the asynchronous nature of the output enables a better solution. The operating system instead buffers the output and allows the process that generated the output to continue execution. After the modifications are committed to disk, the operating system releases the output to the device for which it was destined.

This design requires that the operating system track the causal relationship between file system modifications and external output. When a process writes to the file system, it inherits a commit dependency on the uncommitted data that it wrote. When a process with commit dependencies modifies another kernel object (process, pipe, file, UNIX socket, etc.) by executing a system call, the operating system marks the modified objects with the same commit dependencies. Similarly, if a process observes the state of another kernel object with commit dependencies, the process inherits those dependencies. If a process with commit dependencies executes a system call for which the operating system cannot track the flow of causality (e.g., an `ioctl`), the process is blocked until its file systems modifications have been committed. Any external output inherits the commit dependencies of the process that generated it — the operating system buffers the output until the last dependency is resolved by committing modifications to disk.

2.4 Deciding when to commit

An externally synchronous file system uses the causal relationship between external output and file modifications to trigger commits. There is a well-known tradeoff between throughput and latency for group commit strategies. Delaying a group commit in the hope that more modifications will occur in the near future can improve throughput by amortizing more modifications across a single commit. However, delaying a commit also increases latency — in our system, commit latency is especially important because output cannot be externalized until the commit occurs.

Latency is unimportant if no external entity is observing the result. Specifically, until some output is generated that causally depends on a file system transaction, committing the transaction does not change the observable behavior of the system. Thus, the operating system can improve throughput by delaying a commit until some output that depends on the transaction is buffered (or until some application that depends on the transaction blocks due to an `ioctl` or similar system call). We call this strategy *output-triggered commits* since the attempt to generate output that is causally dependent upon modifications to be written to disk triggers the commit of those modifications.

Output-triggered commits enable an externally synchronous file system to maximize throughput when output is not being displayed (for example, when it is piped to a file). However, when a user could be actively observing the results of a transaction, commit latency is small.

2.5 Limitations

One potential limitation of external synchrony is that it complicates application-specific recovery from catastrophic media failure because the application continues execution before such errors are detected. Although the kernel validates each modification before writing it to the file cache, the physical write of the data to disk may subsequently fail. While smaller errors such as a bad disk block are currently handled by the disk or device driver, a catastrophic media failure is rarely masked at these levels. Theoretically, a file system mounted synchronously could propagate such failures to the application. However, a recent survey of common file systems [20] found that write errors are either not detected by the file system (ext3, jbd, and NTFS) or induce a kernel panic (ReiserFS). An externally synchronous file system could propagate failures to applications by using Speculator to checkpoint a process before it modifies the file system. If a catastrophic failure occurs, the process would be rolled back and notified of the failure. We rejected this solution because it would both greatly increase the complexity

of external synchrony and severely penalize its performance. Further, it is unclear that catastrophic failures are best handled by applications — it seems best to handle them in the operating system, either by inducing a kernel panic or (preferably) by writing data elsewhere.

Another limitation of external synchrony is that the user may have some temporal expectations about when modifications are committed to disk. As defined so far, an externally synchronous file system could indefinitely delay committing data written by an application with no external output. If the system crashes, a substantial amount of work could be lost. Xsyncfs therefore commits data every 5 seconds, even if no output is produced. The 5 second commit interval is the same value used by ext3 mounted asynchronously.

A final limitation of external synchrony is that modifications to data in two different file systems cannot be easily committed with a single disk transaction. Potentially, we could share a common journal among all local file systems, or we could implement a two-phase commit strategy. However, a simpler solution is to block a process with commit dependencies for one file system before it modifies data in a second. Speculator would map each dependency to a specific file system. When a process writes to a file system, Speculator would verify that the process depends only on the file system it is modifying; if it depends on another file system, Speculator would block it until its previous modifications commit.

3 Implementation

3.1 External synchrony

We next provide a brief overview of Speculator [17] and how it supports externally synchronous file systems.

3.1.1 Speculator background

Speculator improves the performance of distributed file systems by hiding the performance cost of remote operations. Rather than block during a remote operation, a file system predicts the operation's result, then uses Speculator to checkpoint the state of the calling process and speculatively continue its execution based on the predicted result. If the prediction is correct, the checkpoint is discarded; if it is incorrect, the calling process is restored to the checkpoint, and the operation is retried.

Speculator adds two new data structures to the kernel. A *speculation* object tracks all process and kernel state that depends on the success or failure of a speculative operation. Each speculative object in the kernel has an *undo log* that contains the state needed to undo speculative modifications to that object. As processes interact with kernel objects by executing system calls, Speculator uses these data structures to track causal dependencies. For example, if a speculative process writes to a pipe, Speculator creates an entry in the pipe's undo log that refers to the speculations on which the writing process depends. If another process reads from the pipe, Speculator creates an undo log entry for the reading process that refers to all speculations on which the pipe depends.

Speculator ensures that speculative state is never visible to an external observer. If a speculative process executes a system call that would normally externalize output, Speculator buffers its output until the outcome of the speculation is decided. If a speculative process performs a system call that Speculator is unable to handle by either transferring causal dependencies or buffering output, Speculator blocks it until it becomes non-speculative.

3.1.2 From speculation to synchronization

Speculator ties dependency tracking and output buffering to other features, such as checkpoint and rollback, that are not needed to support external synchrony. Worse yet, these unneeded features come at a substantial performance cost. This led us to factor out the functionality in Speculator common to both speculative execution and external synchrony. We modified the Speculator interface to allow each file system to specify the additional Speculator features that it requires. This allows a single computer to run both a speculative distributed file system and an externally synchronous local file system.

Both speculative execution and external synchrony enforce restrictions on when external output may be observed. Speculative execution allows output to be observed based on *correctness*; output is externalized after all speculations on which that output depends have proven to be correct. In contrast, external synchrony allows output to be observed based on *durability*; output is externalized after all file system operations on which that output depends have been committed to disk.

In external synchrony, a commit dependency represents the causal relationship between kernel state and an uncommitted file system modification. Any kernel object that has one or more associated commit dependencies is referred to as *uncommitted*. Any external output from a process that is uncommitted is buffered within the kernel until the modifications on which the output depends have been committed. In other words, uncommitted output is never visible to an external observer.

When a process writes to an externally synchronous file system, Speculator marks the process as uncommitted. It also creates a commit dependency between the process and the uncommitted file system transaction that contains the modification. When the file system commits the transaction to disk, the commit dependency is removed.

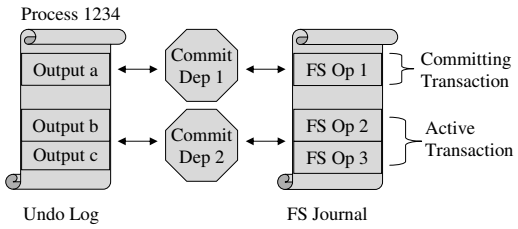
Once all commit dependencies for buffered output have been removed, Speculator releases that output to the external device to which it was written. When the last commit dependency for a process is discarded, Speculator marks the process as committed.

Speculator propagates commit dependencies among kernel objects and processes using the same mechanisms it uses to propagate speculative dependencies. However, since external synchrony does not require checkpoint and rollback, the propagation of dependencies is considerably easier to implement. For instance, before a process inherits a new speculative dependency, Speculator must checkpoint its state with a copy-on-write fork. In contrast, when a process inherits a commit dependency, no checkpoint is needed since the process will never be rolled back. To support external synchrony, Speculator maintains the same many-to-many relationship between commit dependencies and undo logs as it does for speculations and undo logs. Since commit dependencies are never rolled back, undo logs need not contain data to undo the effects of an operation. Therefore, undo logs in an externally synchronous system only track the relationship between commit dependencies and kernel objects and reveal which buffered output can be safely released. This simplicity enables Speculator to support more forms of interaction among uncommitted processes than it supports for speculative processes. For example, checkpointing multi-threaded processes for speculative execution is a thorny problem [17, 21]. However, as discussed in Section 3.5, tracking their commit dependencies is substantially simpler.

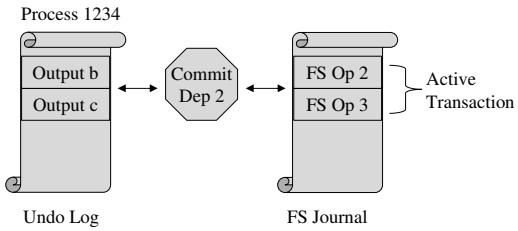
3.2 File system support for external synchrony

We modified ext3, a journaling Linux file system, to create xsyncfs. In its default *ordered* mode, ext3 writes only metadata modifications to its journal. In its *journaled* mode, ext3 writes both data and metadata modifications. Modifications from many different file system operations may be grouped into a single compound journal transaction that is committed atomically. Ext3 writes modifications to the *active* transaction — at most one transaction may be active at any given time. A commit of the active transaction is triggered when journal space is exhausted, an application performs an explicit synchronization operation such as `fsync`, or the oldest modification in the transaction is more than 5 seconds old. After the transaction starts to commit, the next modification triggers the creation of a new active transaction. Only one transaction may be committing at any given time, so the next transaction must wait for the commit of the prior transaction to finish before it commits.

Figure 2 shows how the external synchrony data structures change when a process interacts with xsyncfs. In



(a) Data structures with a committing and active transaction



(b) Data structures after the first transaction commits

Figure 2. The external synchrony data structures

Figure 2(a), process 1234 has completed three file system operations, sending output to the screen after each one. Since the output after the first operation triggered a transaction commit, the two following operations were placed in a new active transaction. The output is buffered in the undo log; the commit dependencies maintain the relationship between buffered output and uncommitted data. In Figure 2(b), the first transaction has been committed to disk. Therefore, the output that depended upon the committed transaction has been released to the screen and the commit dependency has been discarded.

Xsyncfs uses journaled mode rather than the default ordered mode. This change guarantees ordering; specifically, the property that if an operation A causally precedes another operation B, the effects of B should never be visible unless the effects of A are also visible. This guarantee requires that B never be committed to disk before A. Otherwise, a system crash or power failure may occur between the two commits — in this case, after the system is restarted, B will be visible when A is not. Since journaled mode adds all modifications for A to the journal before the operation completes, those modifications must already be in the journal when B begins (since B causally follows A). Thus, either B is part of the same transaction as A (in which case the ordering property holds since A and B are committed atomically), or the transaction containing A is already committed before the transaction containing B starts to commit.

In contrast, the default mode in ext3 does not provide ordering since data modifications are not journaled. The kernel may write the dirty blocks of A and B to disk in any order as long as the data reaches disk before the metadata in the associated journal transaction commits.

Thus, the data modifications for B may be visible after a crash without the modifications for A being visible.

Xsyncfs informs Speculator when a new journal transaction is created — this allows Speculator to track state that depends on the uncommitted transaction. Xsyncfs also informs Speculator when a new modification is added to the transaction and when the transaction commits.

As described in Section 1, the default behavior of ext3 does not guarantee that modifications are durable after a power failure. In the Linux 2.4 kernel, durability can be ensured only by disabling the drive cache. The Linux 2.6.11 kernel provides the option of using *write barriers* to flush the drive cache before and after writing each transaction commit record. Since Speculator runs on a 2.4 kernel, we ported write barriers to our kernel and modified xsyncfs to use write barriers to guarantee that all committed modifications are preserved, even on power failure.

3.3 Output-triggered commits

Xsyncfs uses the causal relationship between disk I/O and external output to balance the competing concerns of throughput and latency. Currently, ext3 commits its journal every 5 seconds, which typically groups the commit of many file system operations. This strategy optimizes for throughput, a logical behavior when writes are asynchronous. However, latency is an important consideration in xsyncfs since users must wait to view output until the transactions on which that output depends commit. If xsyncfs were to use the default ext3 commit strategy, disk throughput would be high, but the user might be forced to wait up to 5 seconds to see output. This behavior is clearly unacceptable for interactive applications.

We therefore modified Speculator to support output-triggered commits. Speculator provides callbacks to xsyncfs when it buffers output or blocks a process that performed a system call for which it cannot track the propagation of causal dependencies (e.g., an `ioctl`). Xsyncfs uses the ext3 strategy of committing every 5 seconds unless it receives a callback that indicates that Speculator blocked or buffered output from a process that depends on the active transaction. The receipt of a callback triggers a commit of the active transaction.

Output-triggered commits adapt the behavior of the file system according to the observable behavior of the system. For instance, if a user directs output from a running application to the screen, latency is reduced by committing transactions frequently. If the user instead redirects the output to a file, xsyncfs optimizes for throughput by committing every 5 seconds. Optimizing for throughput is correct in this instance since the only event the user can observe is the completion of the application (and the completion would trigger a commit if it is a visible event). Finally, if the user were to observe the contents of the file using a different application, e.g., `tail`, xsyncfs would correctly optimize for latency because Speculator would track the causal relationship through the kernel data structures from `tail` to the transaction and provide callbacks to xsyncfs. When `tail` attempts to output data to the screen, Speculator callbacks will cause xsyncfs to commit the active transaction.

3.4 Rethinking sync

Asynchronous file systems provide explicit synchronization operations such as `sync` and `fdatasync` for applications with durability or ordering constraints. In a synchronous file system, such synchronization operations are redundant since ordering and durability are already guaranteed for all file system operations. However, in an externally synchronous file system, some extra support is needed to minimize latency. For instance, a user who types "sync" in a terminal would prefer that the command complete as soon as possible.

When xsyncfs receives a synchronization call such as `sync` from the VFS layer, it creates a commit dependency between the calling process and the active transaction. Since this does not require a disk write, the return from the synchronization call is almost instantaneous. If a visible event occurs, such as the completion of the `sync` process, Speculator will issue a callback that causes xsyncfs to commit the active transaction.

External synchrony simplifies the file system abstraction. Since xsyncfs requires no application modification, programmers can write the same code that they would write if they were using a unmodified file system mounted synchronously. They do not need explicit synchronization calls to provide ordering and durability since xsyncfs provides these guarantees by default for all file system operations. Further, since xsyncfs does not incur the large performance penalty usually associated with synchronous I/O, programmers do not need complicated group commit strategies to achieve acceptable performance. Group commit is provided transparently by xsyncfs.

Of course, a hand-tuned strategy might offer better performance than the default policies provided by xsyncfs. However, as described in Section 3.3, there are some instances in which xsyncfs can optimize performance when an application solution cannot. Since xsyncfs uses output-triggered commits, it knows when no external output has been generated that depends on the current transaction; in these instances, xsyncfs uses group commit to optimize throughput. In contrast, an application-specific commit strategy cannot determine the visibility

of its actions beyond the scope of the currently executing process; it must therefore conservatively commit modifications before producing external messages.

For example, consider a client that issues two sequential transactions to a database server on the same computer and then produces output. Xsyncfs can safely group the commit of both transactions. However, the database server (which does not use output-triggered commits) must commit each transaction separately since it cannot know whether or not the client will produce output after it is informed of the commit of the first transaction.

3.5 Shared memory

Speculator does not propagate speculative dependencies when processes interact through shared memory due to the complexity of checkpointing at arbitrary states in a process' execution. Since commit dependencies do not require checkpoints, we enhanced Speculator to propagate them among processes that share memory.

Speculator can track causal dependencies because processes can only interact through the operating system. Usually, this interaction involves an explicit system call (e.g., `write`) that Speculator can intercept. However, when processes interact through shared memory regions, only the sharing and unsharing of regions is visible to the operating system. Thus, Speculator cannot readily intercept individual reads and writes to shared memory.

We considered marking a shared memory page inaccessible when a process with write permission inherits a commit dependency that a process with read permission does not have. This would trigger a page fault whenever a process reads or writes the shared page. If a process reads the page after another writes it, any commit dependencies would be transferred from the writer to the reader. Once these processes have the same commit dependencies, the page can be restored to its normal protections. We felt this mechanism would perform poorly because of the time needed to protect and unprotect pages, as well as the extra page faults that would be incurred.

Instead, we decided to use an approach that imposes less overhead but might transfer dependencies when not strictly necessary. We make a conservative assumption that processes with write permission for a shared memory region are continually writing to that region, while processes with read permission are continually reading it. When a process with write permission for a shared region inherits a new commit dependency, any process with read permission for that region atomically inherits the same dependency.

Speculator uses the same mechanism to track commit dependencies transfered through memory-mapped files. Similarly, Speculator is conservative when propagating dependencies for multi-threaded applications — any dependency inherited by one thread is inherited by all.

4 Evaluation

Our evaluation answers the following questions:

- How does the durability of xsyncfs compare to current file systems?
- How does the performance of xsyncfs compare to current file systems?
- How does xsyncfs affect the performance of applications that synchronize explicitly?
- How much do output-triggered commits improve the performance of xsyncfs?

4.1 Methodology

All computers used in our evaluation have a 3.02 GHZ Pentium 4 processor with 1 GB of RAM. Each computer has a single Western Digital WD-XL40 hard drive, which is a 7200 RPM 120 GB ATA 100 drive with a 2 MB on-disk cache. The computers run Red Hat Enterprise Linux version 3 (kernel version 2.4.21). We use a 400 MB journal size for both ext3 and xsyncfs. For each benchmark, we measured ext3 executing in both journaled and ordered mode. Since journaled mode executed faster in every benchmark, we report only journaled mode results in this evaluation. Finally, we measured the performance of ext3 both using write barriers and with the drive cache disabled. In all cases write barriers were faster than disabling the drive cache since the drive cache improves read times and reduces the frequency of writes to the disk platter. Thus, we report only results using write barriers.

4.2 Durability

Our first benchmark empirically confirms that without write barriers, ext3 does not guarantee durability. This result holds in both journaled and ordered mode, whether ext3 is mounted synchronously or asynchronously, and even if `fsync` commands are issued by the application after every write. Even worse, our results show that, despite the use of journaling in ext3, a loss of power can corrupt data and metadata stored in the file system.

We confirmed these results by running an experiment in which a test computer continuously writes data to its local file system. After each write completes, the test computer sends a UDP message that is logged by a remote computer. During the experiment, we cut power to the test computer. After it reboots, we compare the state of its file system to the log on the remote computer.

File system configuration	Data durable on `write`	Data durable on `fsync`
Asynchronous	No	Not on power failure
Synchronous	Not on power failure	Not on power failure
Synchronous with write barriers	Yes	Yes
External synchrony	Yes	Yes

This figure describes whether each file system provides durability to the user when an application executes a `write` or `fsync` system call. A "Yes" indicates that the file system provides durability if an OS crash or power failure occurs.

Figure 3. When is data safe?

Our goal was to determine when each file system guarantees durability and ordering. We say a file system fails to provide durability if the remote computer logs a message for a write operation, but the test computer is missing the data written by that operation. In this case, durability is not provided because an external observer (the remote computer) saw output that depended on data that was subsequently lost. We say a file system fails to provide ordering if the state of the file after reboot violates the temporal ordering of writes. Specifically, for each block in the file, ordering is violated if the file does not also contain all previously-written blocks.

For each configuration shown in Figure 3, we ran four trials of this experiment: two in journaled mode and two in ordered mode. As expected, our results confirm that ext3 provides durability only when write barriers are used. Without write barriers, synchronous operations ensure only that modifications are written to the hard drive cache. If power fails before the modifications are written to the disk platter, those modifications are lost.

Some of our experiments exposed a dangerous behavior in ext3: unless write barriers are used, power failures can corrupt both data and metadata stored on disk. In one experiment, a block in the file being modified was silently overwritten with garbage data. In another, a substantial amount of metadata in the file system, including the superblock, was overwritten with garbage. In the latter case, the test machine failed to reboot until the file system was manually repaired. In both cases, corruption is caused by the commit block for a transaction being written to the disk platter before all data blocks in that transaction are written to disk. Although the operating system wrote the blocks to the drive cache in the correct order, the hard drive reorders the blocks when writing them to the disk platter. After this happens, the transaction is committed during recovery even though several data blocks do not contain valid data. Effectively, this overwrites disk blocks with uninitialized data.

Our results also confirm that ext3 without write barriers writes data to disk out of order. Journaled mode alone is insufficient to provide ordering since the order of writing transactions to the disk platter may differ from the order of writing transactions to the drive cache. In contrast, ext3 provides both durability and ordering when write barriers are combined with some form of synchronous operation (either mounting the file system synchronously or calling `fsync` after each modification). If write barriers are not available, the equivalent behavior could also be achieved by disabling the hard drive cache.

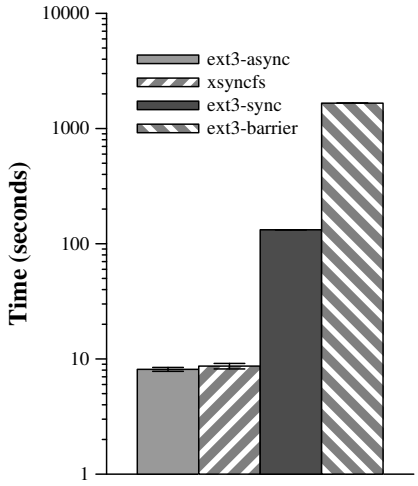
The last row of Figure 3 shows results for xsyncfs. As expected, xsyncfs provides both durability and ordering.

4.3 The PostMark benchmark

We next ran the PostMark benchmark, which was designed to replicate the small file workloads seen in electronic mail, netnews, and web-based commerce [8]. We used PostMark version 1.5, running in a configuration that creates 10,000 files, performs 10,000 transactions consisting of file reads, writes, creates, and deletes, and then removes all files. The PostMark benchmark has a single thread of control that executes file system operations as quickly as possible. PostMark is a good test of file system throughput since it does not generate any output or perform any substantial computation.

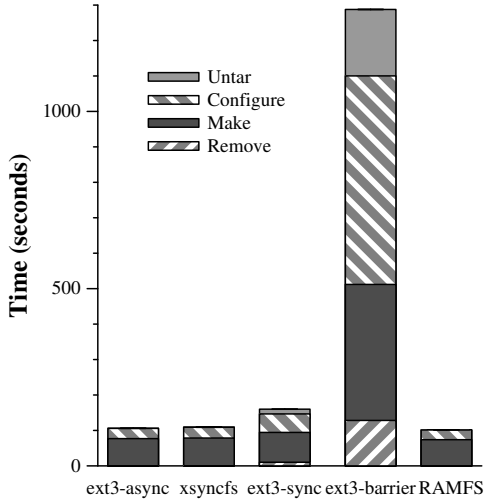
Each bar in Figure 4 shows the time to complete the PostMark benchmark. The y-axis is logarithmic because of the substantial slowdown of synchronous I/O. The first bar shows results when ext3 is mounted asynchronously. As expected, this offers the best performance since the file system buffers data in memory up to 5 seconds before writing it to disk. The second bar shows results using xsyncfs. Despite the I/O intensive nature of PostMark, the performance of xsyncfs is within 7% of the performance of ext3 mounted asynchronously. After examining the performance of xsyncfs, we determined that the overhead of tracking causal dependencies in the kernel accounts for most of the difference.

The third bar shows performance when ext3 is mounted synchronously. In this configuration the writing process is blocked until its modifications are committed to the drive cache. Ext3 in synchronous mode is over an order of magnitude slower than xsyncfs, even though xsyncfs provides stronger durability guarantees. Throughput is limited by the size of the drive cache; once the cache fills, subsequent writes block until some data in the cache is written to the disk platter.



This figure shows the time to run the PostMark benchmark — the y-axis is logarithmic. Each value is the mean of 5 trials — the (relatively small) error bars are 90% confidence intervals.

Figure 4. The PostMark file system benchmark



This figure shows the time to run the Apache build benchmark. Each value is the mean of 5 trials — the (relatively small) error bars are 90% confidence intervals.

Figure 5. The Apache build benchmark

The last bar in Figure 4 shows the time to complete the benchmark when ext3 is mounted synchronously and write barriers are used to prevent data loss when a power failure occurs. Since write barriers synchronously flush the drive cache twice for each file system transaction, ext3's performance is over two orders of magnitude slower than that of xsyncfs.

Due to the high cost of durability, high end storage systems sometimes use specialized hardware such as a non-volatile cache to improve performance [7]. This eliminates the need for write barriers. However, even with specialized hardware, we expect that the performance of ext3 mounted synchronously would be no better than the third bar in Figure 4, which writes data to a volatile cache. Thus, use of xsyncfs should still lead to substantial performance improvements for synchronous operations even when the hard drive has a non-volatile cache of the same size as the volatile cache on our drive.

4.4 The Apache build benchmark

We next ran a benchmark in which we untar the Apache 2.0.48 source tree into a file system, run `configure` in an object directory within that file system, run `make` in the object directory, and remove all files. The Apache build benchmark requires the file system to balance throughput and latency; it displays large amounts of screen output interleaved with disk I/O and computation.

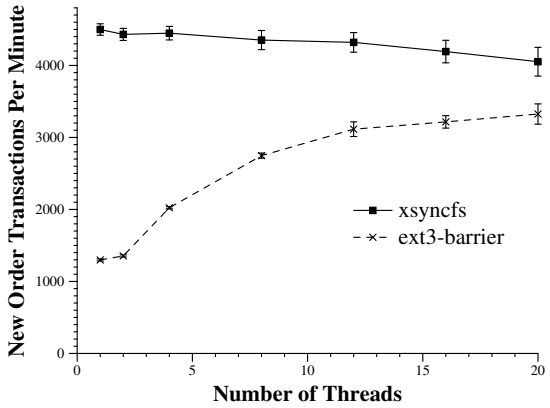
Figure 5 shows the total amount of time to run the benchmark, with shadings within each bar showing the time for each stage. Comparing the first two bars in the graph, xsyncfs performs within 3% of ext3 mounted asynchronously. Since xsyncfs releases output as soon as the data on which it depends commits, output appears promptly during the execution of the benchmark.

For comparison, the bar at the far right of the graph shows the time to execute the benchmark using a memory-only file system, RAMFS. This provides a lower bound on the performance of a local file system, and it isolates the computation requirements of the benchmark. Removing disk I/O by running the benchmark in RAMFS improves performance by only 8% over xsyncfs because the remainder of the benchmark is dominated by computation.

The third bar in Figure 5 shows that ext3 mounted in synchronous mode is 46% slower than xsyncfs. Since computation dominates I/O in this benchmark, any difference in I/O performance is a smaller part of overall performance. The fourth bar shows that ext3 mounted synchronously with write barriers is over 11 times slower than xsyncfs. If we isolate the cost of I/O by subtracting the cost of computation (calculated using the RAMFS result), ext3 mounted synchronously is 7.5 times slower than xsyncfs while ext3 mounted synchronously with write barriers is over two orders of magnitude slower than xsyncfs. These isolated results are similar to the values that we saw for the PostMark experiments.

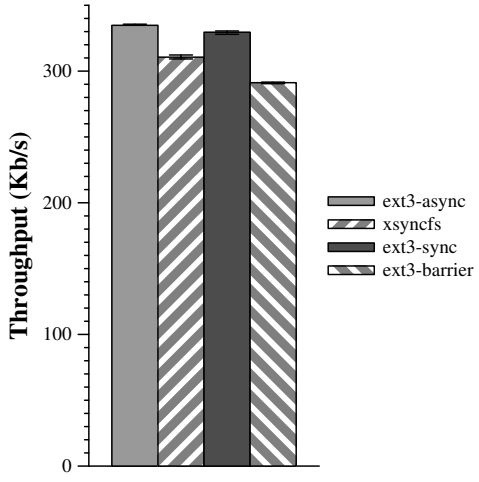
4.5 The MySQL benchmark

We were curious to see how xsyncfs would perform with an application that implements its own group commit strategy. We therefore ran a modified version of the OSDL TPC-C benchmark [18] using MySQL version 5.0.16 and the InnoDB storage engine. Since both MySQL and the TPC-C benchmark client are



This figure shows the New Order Transactions Per Minute when running a modified TPC-C benchmark on MySQL with varying numbers of clients. Each result is the mean of 5 trials — the error bars are 90% confidence intervals.

Figure 6. The MySQL benchmark



This figure shows the mean throughput achieved when running the SPECweb99 benchmark with 50 simultaneous connections. Each result is the mean of three trials, with error bars showing the highest and lowest result.

Figure 7. Throughput in the SPECweb99 benchmark

multi-threaded, this benchmark measures the efficacy of xsyncfs's support for shared memory. TPC-C measures the New Order Transactions Per Minute (NOTPM) a database can process for a given number of simultaneous client connections. The total number of transactions performed by TPC-C is approximately twice the number of New Order Transactions. TPC-C requires that a database provide ACID semantics, and MySQL requires either disabling the drive cache or using write barriers to provide durability. Therefore, we compare xsyncfs with ext3 mounted asynchronously using write barriers. Since the client ran on the same machine as the server, we modified the benchmark to use UNIX sockets. This allows xsyncfs to propagate commit dependencies between the client and server on the same machine. In addition, we modified the benchmark to saturate the MySQL server by removing any wait times between transactions and creating a data set that fits completely in memory.

Figure 6 shows the NOTPM achieved as the number of clients is increased from 1 to 20. With a single client, MySQL completes 3 times as many NOTPM using xsyncfs. By propagating commit dependencies to both the MySQL server and the requesting client, xsyncfs can group commit transactions from a single client, significantly improving performance. In contrast, MySQL cannot benefit from group commit with a single client because it must conservatively commit each transaction before replying to the client.

When there are multiple clients, MySQL can group the commit of transactions from different clients. As the number of clients grows, the gap between xsyncfs and ext3 mounted asynchronously with write barriers shrinks. With 20 clients, xsyncfs improves TPC-C performance by 22%. When the number of clients reaches 32, the performance of ext3 mounted asynchronously with write barriers matches the performance of xsyncfs. From these results, we conclude that even applications such as MySQL that use a custom group commit strategy can benefit from external synchrony if the number of concurrent transactions is low to moderate.

Although ext3 mounted asynchronously without write barriers does not meet the durability requirements for TPC-C, we were still curious to see how its performance compared to xsyncfs. With only 1 or 2 clients, MySQL executes 11% more NOTPM with xsyncfs than it executes with ext3 without write barriers. With 4 or more clients, the two configurations yield equivalent performance within experimental error.

4.6 The SPECweb99 benchmark

Since our previous benchmarks measured only workloads confined to a single computer, we also ran the SPECweb99 [29] benchmark to examine the impact of external synchrony on a network-intensive application. In the SPECweb99 benchmark, multiple clients issue a mix of HTTP GET and POST requests. HTTP GET requests are issued for both static and dynamic content up to 1 MB in size. A single client, emulating 50 simultaneous connections, is connected to the server, which runs Apache 2.0.48, by a 100 Mb/s Ethernet switch. As we use the default Apache settings, 50 connections are sufficient to saturate our server.

We felt that this benchmark might be especially challenging for xsyncfs since sending a network message externalizes state. Since xsyncfs only tracks causal dependencies on a single computer, it must buffer each message

Request size	ext3-async	xsyncfs
0–1 KB	0.064 (±0.025)	0.097 (±0.002)
1–10 KB	0.150 (±0.034)	0.180 (±0.001)
10–100 KB	1.084 (±0.052)	1.094 (±0.003)
100–1000 KB	10.253 (±0.098)	10.072 (±0.066)

The figure shows the mean time (in seconds) to request a file of a particular size during three trials of the SPECweb99 benchmark with 50 simultaneous connections. 90% confidence intervals are given in parentheses.

Figure 8. SPECweb99 latency results

until the file system data on which that message depends has been committed. In addition to the normal log data written by Apache, the SPECweb99 benchmark writes a log record to the file system as a result of each HTTP POST. Thus, small file writes are common during benchmark execution — a typical 45 minute run has approximately 150,000 file system transactions.

As shown in Figure 7, SPECweb99 throughput using xsyncfs is within 8% of the throughput achieved when ext3 is mounted asynchronously. In contrast to ext3, xsyncfs guarantees that the data associated with each POST request is durable before a client receives the POST response. The third bar in Figure 7 shows that SPECweb99 using ext3 mounted synchronously achieves 6% higher throughput than xsyncfs. Unlike the previous benchmarks, SPECweb99 writes little data to disk, so most writes are buffered by the drive cache. The last bar shows that xsyncfs achieves 7% better throughput than ext3 mounted synchronously with write barriers.

Figure 8 summarizes the average latency of individual HTTP requests during benchmark execution. On average, use of xsyncfs adds no more than 33 ms of delay to each request — this value is less than the commonly cited perception threshold of 50 ms for human users [5]. Thus, a user should perceive no difference in response time between xsyncfs and ext3 for HTTP requests.

4.7 Benefit of output-triggered commits

To measure the benefit of output-triggered commits, we also implemented an *eager commit* strategy for xsyncfs that triggers a commit whenever the file system is modified. The eager commit strategy still allows for group commit since multiple modifications are grouped into a single file system transaction while the previous transaction is committing. The next transaction will only start to commit once the commit of the previous transaction completes. The eager commit strategy attempts to minimize the latency of individual file system operations.

We executed the previous benchmarks using the eager commit strategy. Figure 9 compares results for the two strategies. The output-triggered commit strategy performs better than the eager commit strategy in every benchmark except SPECweb99, which creates so much output that the eager commit and output-triggered commit strategies perform very similarly. Since the eager commit strategy attempts to minimize the latency of a single operation, it sacrifices the opportunity to improve throughput. In contrast, the output-triggered commit strategy only minimizes latency after output has been generated that depends on a transaction; otherwise it maximizes throughput.

5 Related work

To the best of our knowledge, xsyncfs is the first local file system to provide high-performance synchronous I/O without requiring specialized hardware support or application modification. Further, xsyncfs is the first file system to use the causal relationship between file modifications and external output to decide when to commit data.

While xsyncfs takes a software-only approach to providing high-performance synchronous I/O, specialized hardware can achieve the same result. The Rio file cache [2] and the Conquest file system [31] use battery-backed main memory to make writes persistent. Durability is guaranteed only as long as the computer has power or the batteries remain charged.

Hitz et al. [7] store file system journal modifications on a battery-backed RAM drive cache, while writing file system data to disk. We expect that synchronous operations on Hitz's hybrid system would perform no better than ext3 mounted synchronously without write barriers in our experiments. Thus, xsyncfs could substantially improve the performance of such hybrid systems.

eNVy [33] is a file system that stores data on flash-based NVRAM. The designers of eNVy found that although reads from NVRAM were fast, writes were prohibitively slow. They used a battery-backed RAM write cache to achieve reasonable write performance. The write performance issues seen in eNVy are similar to those we experienced writing data to commodity hard drives. Therefore, it is likely that xsyncfs could also improve performance for flash file systems.

Xsyncfs's focus on providing both strong durability and reasonable performance contrasts sharply with the trend in commodity file systems toward relaxing durability to improve performance. Early file systems such as FFS [14] and the original UNIX file system [22] introduced the use of a main memory buffer cache to hold writes until they are asynchronously written to disk. Early file systems suffered from potential corruption when a computer lost power or an operating system crashed. Recovery often required a time consuming

Benchmark	Eager Commits	Output-Triggered Commits	Speedup
PostMark (seconds)	9.879 (±0.056)	8.668 (±0.478)	14%
Apache (seconds)	111.41 (±0.32)	109.42 (±0.71)	2%
MySQL 1 client (NOTPM)	3323 (±60)	4498 (±73)	35%
MySQL 20 clients (NOTPM)	3646 (±217)	4052 (±200)	11%
SPECweb99 (Kb/s)	312 (±1)	311(±2)	0%

This figure compares the performance of output-triggered commits with an eager commit strategy. Each result shows the mean of 5 trials, except SPECweb99, which is the mean of 3 trials. 90% confidence intervals are given in parentheses.

Figure 9. Benefit of output-triggered commits

examination of the entire state of the file system (e.g., running `fsck`). For this reason, file systems such as Cedar [6] and LFS [23] added the complexity of a write-ahead log to enable fast, consistent recovery of file system state. Yet, as was shown in our evaluation, journaling data to a write-ahead log is insufficient to prevent file system corruption if the drive cache reorders block writes. An alternative to write-ahead logging, Soft Updates [25], carefully orders disk writes to provide consistent recovery. Xsyncfs builds on this prior work since it writes data after returning control to the application and uses a write-ahead log. Thus, external synchrony could improve the performance of synchronous I/O with other journaling file systems such as JFS [1] or ReiserFS [16].

Fault tolerance researchers have long defined consistent recovery in terms of the output seen by the outside world [3, 11, 30]. For example, the *output commit* problem requires that, before a message is sent to the outside world, the state from which that message is sent must be preserved. In the same way, we argue that the guarantees provided by synchronous disk I/O should be defined by the output seen by the outside world, rather than by the results seen by local processes.

It is interesting to speculate why the principle of outside observability is widely known and used in fault tolerance research yet new to the domain of general purpose applications and I/O. We believe this dichotomy arises from the different *scope* and *standard* of recovery in the two domains. In fault tolerance research, the scope of recovery is the entire process; hence not using the principle of outside observability would require a synchronous disk I/O at every change in process state. In general purpose applications, the scope of recovery is only the I/O issued by the application (which can be viewed as an application-specific recovery protocol). Hence it is feasible (though still slow) to issue each I/O synchronously. In addition, the standard for recovery in fault tolerance research is well defined: a recovery system should lose no visible output. In contrast, the standard for recovery in general purpose systems is looser: asynchronous I/O is common, and even synchronous I/O is usually committed synchronously only to the volatile hard drive cache.

Our implementation of external synchrony draws upon two other techniques from the fault tolerance literature. First, buffering output until the commit is similar to deferring message sends until commit [12]. Second, tracking causal dependencies to identify what and when to commit is similar to causal tracking in message logging protocols [4]. We use these techniques in isolation to improve performance and maintain the appearance of synchronous I/O. We also use these techniques in combination via output-triggered commits, which automatically balance throughput and latency.

Transactions, provided by operating systems such as QuickSilver [24], TABS [28], and Locus [32], and by transactional file systems [10, 19], also give the strong durability and ordering guarantees that are provided by xsyncfs. In addition, transactions provide atomicity for a set of file system operations. However, transactional systems typically require that applications be modified to specify transaction boundaries. In contrast, use of xsyncfs requires no such modification.

6 Conclusion

It is challenging to develop simple and reliable software systems if the foundations upon which those systems are built are unreliable. Asynchronous I/O is a prime example of one such unreliable foundation. OS crashes and power failures can lead to loss of data, file system corruption, and out-of-order modifications. Nevertheless, current file systems present an asynchronous I/O interface by default because the performance penalty of synchronous I/O is assumed to be too large.

In this paper, we have proposed a new abstraction, external synchrony, that preserves the simplicity and reliability of a synchronous I/O interface, yet performs approximately as well as an asynchronous I/O interface. Based on these results, we believe that externally synchronous file systems such as xsyncfs can provide a better foundation for the construction of reliable software systems.

Acknowledgments

We thank Manish Anand, Evan Cooke, Anthony Nicholson, Dan Peek, Sushant Sinha, Ya-Yunn Su, our shepherd, Rob Pike, and the anonymous reviewers for feedback on this paper. The work has been supported by the National Science Foundation under award CNS-0509093. Jason Flinn is supported by NSF CAREER award CNS-0346686, and Ed Nightingale is supported by a Microsoft Research Student Fellowship. Intel Corp. has provided additional support. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of NSF, Intel, Microsoft, the University of Michigan, or the U.S. government.

References

[1] BEST, S. JFS overview. Tech. rep., IBM, http://www-128.ibm.com/developerworks/linux/library/l-jfs.html, 2000.

[2] CHEN, P. M., NG, W. T., CHANDRA, S., AYCOCK, C., RAJAMANI, G., AND LOWELL, D. The Rio file cache: Surviving operating system crashes. In *Proceedings of the 7th International Conference on Architectural Support for Programming Languages and Operating Systems* (Cambridge, MA, October 1996), pp. 74–83.

[3] ELNOZAHY, E. N., ALVISI, L., WANG, Y.-M., AND JOHNSON, D. B. A survey of rollback-recovery protocols in message-passing systems. *ACM Computing Surveys 34*, 3 (September 2002), 375–408.

[4] ELNOZAHY, E. N., AND ZWAENEPOEL, W. Manetho: Transparent Rollback-Recovery with Low Overhead, Limited Rollback, and Fast Output Commit. *IEEE Transactions on Computers C-41*, 5 (May 1992), 526–531.

[5] FLAUTNER, K., AND MUDGE, T. Vertigo: Automatic performance-setting for Linux. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation* (Boston, MA, December 2002), pp. 105–116.

[6] HAGMANN, R. Reimplementing the Cedar file system using logging and group commit. In *Proceedings of the 11th ACM Symposium on Operating Systems Principles* (Austin, TX, 1987), pp. 155–162.

[7] HITZ, D., LAU, J., AND MALCOLM, M. File system design for an NFS file server appliance. In *Proceedings of the Winter 1994 USENIX Technical Conference* (1994).

[8] KATCHER, J. PostMark: A new file system benchmark. Tech. Rep. TR3022, Network Appliance, 1997.

[9] LAMPORT, L. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM 21*, 7 (1978), 558–565.

[10] LISKOV, B., AND RODRIGUES, R. Transactional file systems can be fast. In *Proceedings of the 11th SIGOPS European Workshop* (Leuven, Belgium, September 2004).

[11] LOWELL, D. E., CHANDRA, S., AND CHEN, P. M. Exploring failure transparency and the limits of generic recovery. In *Proceedings of the 4th Symposium on Operating Systems Design and Implementation* (San Diego, CA, October 2000).

[12] LOWELL, D. E., AND CHEN, P. M. Persistent messages in local transactions. In *Proceedings of the 1998 Symposium on Principles of Distributed Computing* (June 1998), pp. 219–226.

[13] MCKUSICK, M. K. Disks from the perspective of a file system. *;login: 31*, 3 (June 2006), 18–19.

[14] MCKUSICK, M. K., JOY, W. N., LEFFLER, S. J., AND FABRY, R. S. A fast file system for unix. *ACM Transactions on Computer Systems (TOCS) 2*, 3 (August 1984), 181–197.

[15] MYSQL AB. *MySQL Reference Manual.* http://dev.mysql.com/.

[16] NAMESYS. *ReiserFS.* http://www.namesys.com/.

[17] NIGHTINGALE, E. B., CHEN, P. M., AND FLINN, J. Speculative execution in a distributed file system. In *Proceedings of the 20th ACM Symposium on Operating Systems Principles* (Brighton, United Kingdom, October 2005), pp. 191–205.

[18] OSDL. *OSDL Database Test 2*. http://www.osdl.org/.

[19] PAXTON, W. H. A client-based transaction system to maintain data integrity. In *Proceedings of the 7th ACM Symposium on Operating Systems Principles* (1979), pp. 18–23.

[20] PRABHAKARAN, V., BAIRAVASUNDARAM, L. N., AGRAWAL, N., GUNAWI, H. S., ARPACI-DUSSEAU, A. C., AND ARPACI-DUSSEAU, R. H. IRON file systems. In *Proceedings of the 20th ACM Symposium on Operating Systems Principles* (Brighton, United Kingdom, October 2005), pp. 206–220.

[21] QIN, F., TUCEK, J., SUNDARESAN, J., AND ZHOU, Y. Rx: Treating bugs as allergies—a safe method to survive software failures. In *Proceedings of the 20th ACM Symposium on Operating Systems Principles* (Brighton, United Kingdom, October 2005), pp. 235–248.

[22] RITCHIE, D. M., AND THOMPSON, K. The UNIX time-sharing system. *Communications of the ACM 17*, 7 (1974), 365–375.

[23] ROSENBLUM, M., AND OUSTERHOUT, J. K. The design and implementation of a log-structured file system. *ACM Transactions on Computer Systems (TOCS) 10*, 1 (February 1992), 26–52.

[24] SCHMUCK, F., AND WYLIE, J. Experience with transactions in QuickSilver. In *Proceedings of the 13th ACM Symposium on Operating Systems Principles* (October 1991), pp. 239–53.

[25] SELTZER, M. I., GANGER, G. R., MCKUSICK, M. K., SMITH, K. A., SOULES, C. A. N., AND STEIN, C. A. Journaling versus soft updates: Asynchronous meta-data protection in file systems. In *USENIX Annual Technical Conference* (San Diego, CA, June 2000), pp. 18–23.

[26] SILBERSCHATZ, A., AND GALVIN, P. B. *Operating System Concepts (5th Edition).* Addison Wesley, February 1998. p. 27.

[27] SLASHDOT. *Your Hard Drive Lies to You.* http://hardware.slashdot.org/article.pl?sid=05/05/13/0529252.

[28] SPECTOR, A. Z., DANIELS, D., DUCHAMP, D., EPPINGER, J. L., AND PAUSCH, R. Distributed transactions for reliable systems. In *Proceedings of the 10th ACM Symposium on Operating Systems Principles* (Orcas Island, WA, December 1985), pp. 127–146.

[29] STANDARD PERFORMANCE EVALUATION CORPORATION. *SPECweb99.* http://www.spec.org/web99.

[30] STROM, R. E., AND YEMINI, S. Optimistic Recovery in Distributed Systems. *ACM Transactions on Computer Systems 3*, 3 (August 1985), 204–226.

[31] WANG, A.-I. A., REIHER, P., POPEK, G. J., AND KUENNING, G. H. Conquest: Better performance through a disk/persistent-RAM hybrid file system. In *Proceedings of the 2002 USENIX Annual Technical Conference* (Monterey, CA, June 2002).

[32] WEINSTEIN, M. J., THOMAS W. PAGE, J., LIVEZEY, B. K., AND POPEK, G. J. Transactions and synchronization in a distributed operating system. In *Proceedings of the 10th ACM Symposium on Operating Systems Principles* (Orcas Island, WA, December 1985), pp. 115–126.

[33] WU, M., AND ZWAENEPOEL, W. eNVy: A non-volatile, main memory storage system. In *Proceedings of the 6th International Conference on Architectural Support for Programming Languages and Operating Systems* (San Jose, CA, 1994), pp. 86–97.