Operating System Transactions

Donald E. Porter, Owen S. Hofmann, Christopher J. Rossbach, Alexander Benn, and Emmett Witchel
Department of Computer Sciences, The University of Texas at Austin, Austin, TX 78712
Technical Report 09-13
{*porterde,osh,rossbach,abenn1,witchel*}*@cs.utexas.edu*

Abstract

The programming demands of multi-core processors require support for applications to synchronize access to operating system resources. Operating systems should provide *system transactions* that let users group updates to heterogeneous system resources with atomicity, consistency, isolation, and durability (ACID). System transactions can eliminate time-of-check-to-time-of-use (TOCTTOU) race conditions in the file system, a class of security vulnerability that are difficult to eliminate with other techniques. New processes can be created and run as part of a system transaction, allowing a software installation script to execute transactionally. A failed installation can be rolled back without having to roll back concurrent, independent updates to the file system.

This paper describes TxOS, a variant of Linux 2.6.22. TxOS is the first operating system to implement system transactions on commodity hardware with strong isolation and fairness between transactional and non-transactional system calls. The prototype demonstrates that a mature OS can provide system transactions at a reasonable performance cost, and has minimal performance cost for processes that do not use transactions. For instance, a transactional installation of OpenSSH incurs only 40% overhead. Having the OS support transactions gives users and system developers a powerful new tool to provide innovative services. For example, with TxOS, one developer transformed ext3 into a transactional file system in less than one month.

1 Introduction

Operating systems manage resources for user programs, but those programs currently lack the ability to group operations into logically consistent updates. The POSIX system call API, while providing support for atomic and isolated execution of individual system calls, provides no support for composing multiple calls into a consistent update to OS-managed resources. As a result, the consistency guarantees provided by the POSIX API are difficult, if not impossible, to extend to operations that are too complex to fit into a single system call.

With the current proliferation of multi-core processors, concurrent processing is becoming more prevalent, exposing the POSIX API's weakness at managing consistent updates. This paper proposes *system transactions* to provide atomicity, consistency, isolation, and durability (ACID) for system state. System transactions are easy to use: code regions with consistency constraints need only be enclosed within the appropriate system calls, `sys_xbegin()` and `sys_xend()`. The user can abort an in-progress transaction with `sys_xabort()`. Placing system calls within a transaction alters the semantics of when and how their results are published to the rest of the system. Outside of a transaction, actions on system resources are visible as soon as the relevant internal kernel locks are released. Within a transaction, all updates are kept isolated until commit time, when they are atomically published to the rest of the system. System transactions provide a simple and powerful way for applications to express consistency requirements for concurrent operations to the OS.

Applications currently struggle to make consistent updates to system resources. In simple cases, programmers can serialize operations by using a single system call, such as using `rename` to atomically replace the contents of a file. Unfortunately, more complex operations, such as software installation or upgrade, cannot be condensed to a single system call. For example, software install alters the file system and also creates, kills and signals processes. Executing an entire software install atomically and in isolation from running tasks would be a powerful tool for the system administrator, but would require different types of system resources to be updated consistently. No mainstream operating system provides a combination of system abstractions to express that consistent update.

In the presence of concurrency, applications must ensure consistency by isolating a series of modifications to important data from interference by other tasks. Concurrency control mechanisms currently available to user programs (e.g., file locking) are clumsy and difficult to program. Moreover, they are often insufficient for protecting a series of system calls from interference by other applications, especially when the other applications are buggy or malicious.

In practice, addressing the lack of concurrency control in the system call API has been *ad hoc*: new, semantically rich system calls are added piecemeal to solve new problems that arise. Eliminating file system race conditions is a vital problem that has motivated the Linux developers to

add thirteen new system calls in 2006 (the `openat` family of system calls, also supported by Solaris). The close-on-exec flag was added to fifteen system calls in the 2.6.27 version of Linux [12] to eliminate a race condition between calls to `open` and `fcntl`. Individual file systems have introduced new operations to address consistency needs: the Google File System supports atomic append operations [15], while Windows has recently adopted support for transactions in NTFS [35]. Rather than requiring users to lobby OS developers for new system calls or file system features, why not allow users to solve their own problems by supporting composition of multiple system calls into arbitrary atomic and isolated units?

System transactions allow programmers to continue using the POSIX API in the presence of increased concurrency. They directly solve the problem of consistent updates to system state, eliminating the need for many of the complicated API changes that have been recently introduced to modern operating systems.

This paper describes an implementation of system transactions on Linux called **TxOS**, which provides transactional semantics for 134 out of 323 system calls. To efficiently provide strong guarantees, the TxOS implementation redesigns several key OS data structures and internal subsystem interfaces. TxOS supports transactional semantics for all of its internal data structures, which translates directly to considerable freedom for users and system implementors to create powerful applications and services. Given an initial implementation of TxOS, a single developer needed less than a month to convert `ext3` into a transactional file system.

This paper makes the following contributions.

1. Describes a new approach to OS implementation that supports transactions with strong atomicity and isolation guarantees, while maintaining low performance overheads on commodity hardware.
2. Demonstrates the utility of system transactions to eliminate race conditions in the system call API and to increase OS scalability. Transactional `link`/`unlink` on TxOS can outperform `rename` on Linux by a 3.9× at 8 CPUs.
3. Demonstrates the utility of process-independent system transactions to provide complete recovery for a failed software install without losing concurrent updates to the file system.
4. Demonstrates that a transactional OS with an integrated transactional file system performs provides a lightweight alternative to a database for concurrency management and crash consistency purposes, while being far simpler to maintain. OpenLDAP using flat files and system transactions outperforms the BerkeleyDB on a write-mostly workload by a factor of 2.
5. Demonstrates that the open problem of supporting system calls within user-level transactional applications can be cleanly solved with system transactions.

The remainder of the paper is structured as follows. Section 2 provides an overview of the system transaction API and motivating use-cases. Section 3 positions TxOS in related work and Section 4 describes its design principles. Section 5 describes how certain key subsystems provide transactional semantics. Section 6 provides kernel implementation details, while Section 7 measures the performance overhead of system transactions and evaluates TxOS in a number of application case studies. Section 8 concludes.

2 Overview and motivation

This section illustrates the utility of system transactions with two use cases. System transactions support failure atomicity for software installs without disrupting concurrent, independent updates to the file system and can eliminate race conditions in the file system API.

2.1 Software install

Installing new software or software patches is an increasingly common system activity as time to market pressures and good network connectivity combine to make software updates frequent for users. Yet software upgrade remains a dangerous activity. For example, Microsoft pulled a prerequisite patch for Vista service pack 1 because it caused an endless cycle of boots and reboots [27]. Unfortunately, a partial upgrade can leave a system in an unusable state.

Microsoft Windows, and other systems provide a checkpoint-based solution to the software update problem [29]. Users can take a checkpoint of disk state before they install software: if something goes wrong, they roll back to the checkpoint. However, any updates to the file system that are concurrent but independent from the software install are lost, which significantly decreases the usability of the feature. Moreover, the user or the system must create and manage the disk-based checkpoints to make sure a valid image is always available, further decreasing usability. Finally, if the install problem affects volatile system state, the system can corrupt files unrelated to the install.

TxOS provides a simple interface to address these issues. The software install or update can be executed within a transaction, which isolates the rest of the system until the install is judged successful either by a human or by software. An additional advantage to this approach is that independent updates made concurrently with the install are not reverted as a side-effect.

2.2 Eliminating races for security

Figure 1 depicts a scenario in which an application wants to make a single, consistent update to the file system by checking the access permissions of a file and conditionally writing it. This pattern is common in `setuid` programs, and the source of a major and persistent security problem in modern operating systems. An attacker can change the file system name space using symbolic links between the victim's `access` control check and the file `open`, perhaps tricking a `setuid` program into overwriting a sensitive system file like the password database. The API provides

```
    Victim                    Attacker
if(access('foo')){
                          symlink('secret','foo');
  fd=open('foo');
  write(fd,...);
  ...
}
```

```
    Victim                    Attacker
                          symlink('secret','foo');
sys_xbegin();
if(access('foo')){
  fd=open('foo');
  write(fd,...);
  ...
}
sys_xend();
                          symlink('secret','foo');
```

Figure 1: An example of a TOCTTOU attack, followed by an example that eliminates the race using system transactions. The attacker's symlink is serialized (ordered) either before or after the transaction, and the attacker cannot see partial updates from the victim's transaction, such as changes to `atime`.

no way for the application to express this requirement to the operating system.

Although most common in the file system, system API races, or time-of-check-to-time-of-use (TOCTTOU) races, in other OS resources can be exploited. Local sockets used for IPC are vulnerable to a similar race between creation and connection. Versions of OpenSSH before 1.2.17 suffered from a socket race exploit that allowed a user to steal another's credentials [1]; the Plash sandboxing system suffers a similar vulnerability [2]. Zalewski demonstrates how races in signal handlers can be used to crack applications, including sendmail, screen, and wu-ftpd [53].

While conceptually simple, TOCTTOU vulnerabilities are pervasive in deployed software and are difficult to eliminate. At the time of writing, a search of the U.S. national vulnerability database for the term "symlink attack" yields over 600 hits [34]. Further, recent work by Cai et al. [6] exploits fundamental flaws to defeat two major classes of TOCTTOU countermeasures: dynamic race detectors in the kernel [49] and probabilistic user-space race detectors [48]. This continuous arms race of measure and countermeasure illustrates that TOCTTOU attacks can only be eliminated with change to the API.

In practice, these races are addressed with *ad hoc* extension of the system API. Linux has added a new close-on-exec flag to fifteen different system calls to eliminate a race condition between calls to `open` and `fcntl`. Tsafrir et al. [47] demonstrate how programmers can use the *openat()* family of system calls to construct deterministic countermeasures for many races by traversing the directory tree and checking user permissions in the application. However, these techniques cannot protect against all races without even more API extensions. In particular, Tsafrir's technique is incompatible with the `O_CREAT` flag to `open` that is used to prevent exploits on temporary file creation [8].

Such a reactive approach to fixing race conditions is not likely to be effective long term. Complicating the API in the name of security is a risky: code complexity is often the enemy of code security [4]. Because system transactions provide deterministic safety guarantees and a natural programming model, they are an easy-to-use, general mechanism that can eliminate *all* API race conditions.

3 Related Work

In this section we contrast TxOS with previous research in OS transactions, transactional memory, Speculator, transactional file systems, and distributed transactions.

Previous transactional operating systems. Locus [51] and QuickSilver [19,41] are historical systems that provide some system support for transactions. Both systems use database implementation techniques for transactions, isolating data structures with two-phase locking and rolling back failed transactions from an undo log. A shortcoming of this approach is that simple locks, and even reader-writer locks, do not capture the semantics of container objects, such as a directory. Multiple transactions can concurrently and safely create files in the same directory so long as none of them use the same file name and none of them read the directory. Unfortunately, creating a file in these historical systems requires a write lock on the directory, which serializes the writing transactions and eliminates concurrency. To compensate for the poor performance of reader-writer locks, both systems allow directory contents to change during a transaction, which reintroduces the possibility of the TOCTTOU race conditions that system transactions ought to eliminate.

Transactional Memory. The system transactions supported by TxOS solve a fundamentally different problem from those solved by TxLinux [40]. TxLinux is a variant of Linux that uses hardware transactional memory (HTM) as a synchronization primitive to protect OS data structures within the kernel, whereas TxOS exports a transactional API to user programs. The techniques used to build TxLinux support short critical regions that enforce consistency for accessing memory: these techniques are insufficient to implement TxOS, which must guarantee consistency across heterogeneous system resources, and which must support arbitrarily large transactions. TxOS runs on currently available hardware, unlike TxLinux.

Volos et al. [50] extend the Intel STM compiler with xCalls, which support deferral or rollback of common system calls when performed in a memory transaction. Because xCalls are implemented in a single, user-level application, they cannot isolate transaction effects from other applications on the system, ensure durable updates to a file, or support multi-process transactions, all of which are needed to perform a transactional software install and are supported by TxOS.

Speculator. Speculator [32] applies an isolation and rollback mechanism to the operating system that is very similar to transactions, allowing the system to speculate past

Feature	Amino	TxF	Valor	TxOS
Low overhead kernel implementation	No	Yes	Yes	Yes
Can be root fs?	No	Yes*	Yes	Yes
Framework for transactionalizing other file systems	No	No	Yes	Yes
Transactional and non-transactional access	No	No	Yes	Yes
Simple programmer interface	Yes	No	No	Yes
Other kernel resources in a transaction	No	No	No	Yes

Figure 2: A summary of features supported by recent transactional file systems.

Function Name	Description
int sys_xbegin (int restart, int durable)	Begin a transaction. If restart is true, OS automatically restarts the transaction after an abort. Durable sets whether the results should be on stable storage, where appropriate. Returns status code.
int sys_xend()	End of transaction. Returns whether commit succeeded.
void sys_xabort (int no_restart)	Aborts a transaction. If the transaction was started with restart, setting no_restart overrides that flag and does not restart the transaction.

Table 1: TxOS API

high-latency remote file system operations. The transactional semantics TxOS provides to user programs is a more complicated endeavor. In TxOS, transactions must be isolated from each other, while applications in Speculator share speculative results. Speculator does not eliminate TOCTTOU vulnerabilities. If a TOCTTOU attack occurred in Speculator, the attacker and victim would be part of the same speculation, allowing the attack to succeed. Speculator has been extended to parallelize security checks [33] and to debug system configuration [46], but does not provide a general-purpose interface for users to explicitly delimit speculative code regions, it is insufficient for applications like atomic software install/update.

Transactional file systems. TxOS simplifies the task of writing a transactional file system by detecting conflicts and versioning data in the virtual filesystem layer. Some previous work such as OdeFS [14], Inversion [36], and DBFS [31] provide a file system interface to a database, implemented as a user-level NFS server. These systems do not address the problem of atomic isolated operations on local disk, and cannot address the problem of coordinating access to OS-managed resources. Berkeley DB and Stasis [42] are transactional libraries, not file systems. Amino [52] supports transaction file operation semantics by interposing on system calls using `ptrace`, using a user-level database to store and manage file system data and metadata. Another approach to transactional file system implementation, exemplified by Valor [45] and Transactional NTFS [35], and others [13, 35, 41, 43] implement all transactional semantics directly in the file system.

Figure 2 lists several desirable properties for a transactional file system and compares TxOS with recent systems. Because Amino's database must be hosted on a native file system, it cannot be used as the root file system. TxF can be used as the root file system, but the programmer must ensure that the local system is the two-phase commit coordinator for any distributed transactions it participates in.

Unlike TxF, TxOS allows the same resources to be accessed by transactional and non-transactional threads. For example, TxF does not let a non-transactional thread open a file that is open in a transactional thread. The importance of interoperability is discussed in Section 4.2. Like TxOS, Valor provides kernel support in the page cache to simplify the task of adding transactions to new file systems. Valor supports transactions larger than memory, which TxOS does not currently do. Valor primarily provides logging and coarse-grained locking for files; moreover, because directory operations require locking the directory, Valor, like QuickSilver, is more conservative than necessary with respect to concurrent directory updates.

TxOS provides programmers with a simple, natural interface, augmenting the POSIX API with only three system calls (`sys_xbegin()`, `sys_xend()`, `sys_xabort()`). Other transactional file systems require application programmers to understand implementation details, such as two-phase commit (TxF) and the logging and locking mechanism (Valor).

Distributed transactions. A number of systems, including TABS [44], Argus [25], and Sinfonia [3], provide support for distributed transactional applications at the language or library level. Because these systems implement transactions at user level, they cannot isolate system resources, while TxOS system transactions can.

4 TxOS Design

System transactions are designed to meet the key implementation goals of strong isolation guarantees for transactions, while retaining good performance and simple interfaces. This section outlines how the TxOS design achieves these goals.

4.1 Overview

System transactions in TxOS provide programmers with ACID semantics for system state, as opposed to application state. System state includes OS data structures and device state stored in the OS's address space, whereas application state is the data structures stored in the application's address space. When using system transactions, the application must be able to restore its pre-transaction state if a system transaction aborts. Application state can be managed in several ways: the application state may need no explicit management (as in the TOCTTOU example), the OS can automatically checkpoint and restore the application's address space (as in Speculator [32]), or the

application can implement its own checkpoint and recovery mechanism, perhaps using hardware or software transactional memory. See Section 4.5 for information about how TxOS coordinates with existing hardware and software transactional memory systems.

TxOS uses existing OS memory buffers and kernel data structures to isolate data read and written in a transaction. When an application writes data to a file system or device, the updates generally go into an OS buffer first, allowing the OS to optimize device accesses. By making these buffers copy-on-write for transactions, TxOS isolates transactional data accesses until commit.

TxOS isolates updates to kernel data structures, including objects that represent file system metadata or a process's address space, using recent implementation techniques from object-based software transactional memory systems. These techniques are a departure from the logging and two-phase locking approaches of databases and historic transactional operating systems.

Buffering updates in memory during transactions limits the size of transactions, and restricts the transactional model. For instance, if an application's outgoing message is buffered, it cannot receive a response within the same transaction. Future work could examine using secondary storage to help buffer changes and extending transactions over the network. This paper argues for a more expressive system call framework that can serve as the interface for future enhancements.

4.2 Interoperability and fairness

TxOS allows flexible interaction between transactional and non-transaction kernel threads. Allowing users to freely mix transactional and non-transactional accesses to the same resources is crucial to maintaining a simple interface to system resources. Most programs are not written to use transactions. TxOS can run all of these programs alongside programs that do take advantage of transactions, even if the two programs access the same resources, such as files.

TxOS provides a total order for all operations on system resources, transactional and non-transactional. A total order makes all operations serializable, which matches the intuitive semantics programmers expect [17]. We call the ordering of transactional and non-transactional updates to the same resources strong isolation.

TxOS provides strong isolation efficiently by requiring all threads to use the same locking discipline, and by requiring that transactions annotate accessed objects. When a thread, transactional or non-transactional, accesses an object for the first time, it must check for a conflicting annotation. The scheduler arbitrates conflicts when they are detected. In many cases, this check is performed at the same time as a thread acquires a lock for the object.

Interoperability is a weak spot for previous transactional systems. In most transactional systems, a conflict between a transaction and a non-transactional thread (called an **asymmetric conflict**) must be resolved by aborting the transaction. This approach undermines fairness. In TxOS, because asymmetric conflicts are often detected before a non-transactional thread enters a critical region, the scheduler has the option of suspending the non-transactional thread, allowing for fairness between transactions and non-transactional threads.

4.3 State management

Databases and historical transactional operating systems typically adopt **eager version management** [24], which updates data in place and maintains an undo log. These systems isolate transactions by locking all data encountered and holding the lock until commit (two-phase locking). Because data accesses are not ordered by applications, these systems can deadlock.

The possibility of deadlock complicates the programing model of transactional systems. Deadlock is commonly addressed by exposing a timeout parameter to users. Setting the timeout properly is a challenge. If it is too short, it can starve long-running transactions. If it is too long, it can destroy the performance of the system.

Eager version management degrades responsiveness in ways that are not acceptable for an operating system. If an interrupt handler, high priority thread, or real-time thread aborts a transaction, it must wait for the transaction to walk its undo log before it can safely proceed. This wait can jeopardize the system's ability to meet its timing requirements.

TxOS, in contrast, uses **lazy version management**, where transactions operate on private copies of a data structure. Because lazy versioning requires TxOS to hold locks only long enough to make a private copy of the relevant data structure, deadlock is avoided. Applications never hold kernel locks across system calls. Transactions can be aborted instantly, and no latency is incurred walking the undo log.

The primary disadvantage of lazy versioning is that at commit time transactional updates are copied from the speculative version to the stable version of the data structures. As we discuss in Section 6, TxOS minimizes this overhead by splitting up objects, turning a `memcpy` of the entire object into a pointer copy.

4.4 Precise conflict semantics

A common performance problem with transactional implementations derives from overly conservative, read/write conflict semantics. For instance, linked-lists are heavily used to organize data structures in the Linux kernel, and in many cases can safely permit multiple concurrent writes. TxOS isolates list updates with a lock and defines conflicts according to the compatibility lattice described in Table 2. For instance, a A list in the `write` state can allow concurrent transactional and non-transactional writers, as long as they do not access the same entry. Individual entries that are transactionally added or removed are annotated with a transaction pointer and used to detect conflicts. If a writing transaction also attempts to read the list contents, it

State	Description
`exclusive`	Any attempt to access the list is a conflict with the current owner
`write`	Any number of insertions and deletions are allowed, provided they do not access the same entries. Reads (iterations) are not allowed. Writers may be transactions or non-transactional tasks.
`read`	Any number of readers, transactional or non-transactional, are allowed, but insertions and deletions are conflicts.
`notx`	There are no active transactions, and a non-transactional thread may perform any operation. A transaction must first upgrade to `read` or `write` mode.

Table 2: The states for a transactional list in TxOS. Having multiple states allows TxOS lists to tolerate access patterns that would be conflicts in previous transactional systems.

must upgrade the list to `exclusive` mode by aborting all other writers. The `read` state behaves similarly. This design allows maximal list concurrency while preserving correctness.

4.5 Integrating with user transactions

System transactions protect system state, not application state. For multi-threaded programs, the OS has no efficient mechanism to save and restore the memory state of an individual thread. User-level transactional memory (TM) systems do provide atomic and isolated updates to application data structures. Transactional memory is implemented either in hardware (building on cache coherence) [18, 30], in software [11], or as a hybrid of the two [7, 9]. User and system transactions can coordinate to create a simple and complete transactional programming model.

One of the most troublesome limitations of transactional memory systems is the lack of support for system calls within transactions. For example, a file append inside of a transaction can occur an arbitrary number of times when executed on current TM systems (both hardware and software), depending on how often the user-level transaction has to abort and retry. Because transactional semantics do not extend to the system call, there is no way to rollback previous appends when the transactions retries.

System transactions complete the transactional semantics of user-level transactional systems. When a TM application makes a system call, the runtime begins a system transaction. The user-level transactional memory system handles buffering and possibly rolling back the user's memory state, and the system transaction buffers updates to system state. The updates to system state are committed or aborted by the kernel atomically with the commit or abort of the user-level transaction. We can coordinate user and system transactions by modifying the TM runtime libraries; programmers need only write atomic blocks. The implementation is discussed in Section 6.6.

5 Design of kernel subsystems

This subsection discusses how we extend different kernel subsystems to support ACID semantics. We note that adding transactional semantics often required extending functionality already present in the subsystem, rather than developing it from scratch. We use the journal in `ext3` to provide true, multi-operation durability. We use Linux's ability to defer signal delivery to manage signals sent from transactional threads.

We also observe that having the kernel provide transactional semantics for system resources makes it easy to extend the kernel to provide transactional services. The one-month turnaround for converting `ext3` into a transactional filesystem is a good example. While the journal support of `ext3` helps, having the kernel's data structures support atomicity and isolation provides the heavy lifting of transactional semantics for the subsystems.

One benefit of TxOS is to make it easier for kernel developers to bring transactional semantics to new kernel services. Transactions have proven to be a useful abstraction for concurrent programming, and such abstractions will be necessary to ease the task of programming the proliferating cores on modern CPUs.

5.1 Transactional file system

TxOS simplifies the task of writing a transactional file system by detecting conflicts and versioning data in the virtual filesystem layer. The OS provides the transactional semantics—isolating updates and detecting conflicts. The file system merely has to provide the ability to atomically commit updates (e.g., via a journal), and then it can function as a transactional file system for the user.

By simply ensuring that all committed changes are written in a single journal transaction, `ext3` was converted into a transactional file system. Memory-only file systems, such as `proc` and `tmpfs`, are automatically transactional when used within TxOS.

Durability is only relevant for some system resources, like file systems on non-volatile storage. Providing durability can slow performance because of the increased latency of stable storage, so users should have the option of relaxing it when they do not need it. For instance, eliminating the TOCTTOU race does not require durable updates.

5.2 Multi-process transactions

The paradigm for UNIX application development is to compose more complex tasks from simple, powerful utility programs. Programs might wish to transactionally fork a number of child processes to execute utilities and wait for the results to be returned through a pipe. To support this programming paradigm in a natural way, TxOS allows multiple tasks to be part of the same transaction. Processes in the same transaction share and synchronize on speculative state.

When a process forks a child inside a transaction, the child process is in the same transaction until it performs an `sys_xend()` or it exits. The transaction com-

mits when all tasks in the transaction have issued an `sys_xend()`. The exit system call is considered an implicit `sys_xend()`. This method of process management allows transactional programs to call high-level convenience functions like `system` to easily create a process with a complicated command line using the full compliment of shell functionality. The `exec`ed program runs with transactional semantics, though it might not contain any explicitly transactional code. After a child process commits, it is no longer part of the transaction and subsequent `sys_xbegin()` calls will begin transactions that are completely independent from the parent.

TxOS does not need to checkpoint task state for transactionally forked processes. Because an abort will ultimately terminate them, no rollback is required. A range of system calls can update task state, ranging from memory allocation to installing a new file descriptor. These calls do not incur the performance overhead of checkpointing private task data in a transactionally forked task.

5.3 Signal delivery

Signal semantics in TxOS derive from the need to provide isolation between processes in different transactions, as well as isolation between non-transactional and transactional processes. Transactionally forked processes enlisted in the same multi-process transaction can send and receive signals freely with other processes in the same transaction. Signals sent by transactional processes are deferred until commit by placing them in a deferred queue, regardless of whether the receiving process is transactional. Signals in the queue are delivered in order if the transaction commits, and discarded if the transaction aborts.

Signals sent to a process that has an active transaction can either be deferred until commit (*deferred receive*), or speculatively handled in the transaction (*speculative receive*). When signals are received speculatively, they must be logged. If the transaction aborts, these signals must be re-delivered to the receiving process so that from the sender's perspective the signals do not disappear. When a transaction that has speculatively received a signal commits, logged signals are discarded. When received signals are deferred, incoming signals are placed in a queue and delivered in order when the transaction commits.

Deferred receive addresses atomicity vulnerabilities in signal handlers [53]. Enclosing signal handling code in a transaction ensures that system calls in handler are atomic, and forces calls to the same handler to serialize. This eliminates atomicity vulnerabilities without the need for the additional API complexity of `sigaction`. While the `sigaction` API can address signal handler atomicity within a single process by making handlers non-reentrant, the API does not make signal handlers atomic with respect to other processes.

The `SIGSTOP` and `SIGKILL` signals cannot be blocked or ignored outside of a transaction. TxOS does not alter these semantics, and these signals are delivered to transactional threads and handled, even if the transaction was started in deferred receive mode.

Subsystem	Tot.	Part.	Examples
Credentials	35	5	getuid, getcpu, setrlimit (partial)
Processes	4	1	fork, vfork, clone, exit, exec (partial)
Signals	8	0	rt_sigaction, rt_sigprocmask, kill
Filesystem	56	2	link, access, stat, chroot, dup
File Access	11	4	open, close, write, lseek, truncate
Other	4	4	time, nanosleep, ioctl (partial), mmap2 (partial)
Totals	118	16	Grand total: 134

Figure 3: Summary of system calls that TxOS completely supports (Tot.) and partially supports (Part.) in transactions.

6 TxOS Kernel Implementation

This section describes how system transactions are implemented in the TxOS kernel. TxOS provides transactional semantics to 134 of 323 system calls in Linux, presented in Table 3. The classes of system calls include process creation/termination, sending and receiving signals, and file system operations.

System transactions in TxOS add roughly 3,300 lines of code for transaction management, and 5,300 lines for object management. TxOS also requires about 14,000 lines of minor changes to convert kernel code to use the new object type system and to insert checks for asymmetric conflicts when executing non-transactionally.

6.1 Versioning data

TxOS must maintain multiple versions of kernel data structures for system transactions to isolate the effects of system calls until the transaction commits and to undo the effects of a transaction if it cannot complete. Data structures private to a process, such as the current user id or the file descriptor table, can be versioned with a simple checkpoint and restore scheme. For shared kernel data structures, however, TxOS implements a versioning system that borrows techniques from modern software transactional memory systems [20] and other recent concurrent programming systems [23].

When a transaction accesses a shared kernel object, such as an `inode`, it acquires a private copy of the object, called a **shadow** object. For the rest of the transaction, this shadow object is used in place of the **stable** object. Shadow objects ensure that transactions always have a consistent view of the system state. When the transaction commits, the shadow objects replace their stable counterpart. If a transaction cannot complete, the shadow objects are simply discarded.

Any given kernel object may be the target of pointers from several other objects, presenting a challenge to replacing a stable object with a newly-committed shadow

```
struct inode_header {
  atomic_t      i_count; // Reference count
  spinlock_t    i_lock;
  inode_data    *data;   // Data object
  // Other objects
  address_space i_data;  // Cached data pages
  tx_data xobj;   // used for conflict detection
  list i_sb_list; // kernel bookkeeping };

struct inode_data {
  inode_header *header;
  // Common inode data fields
  unsigned long i_ino;
  loff_t        i_size; // etc. };
```

Figure 4: A simplified `inode` structure, decomposed into header and data objects in TxOS. The header contains the reference count, locks, kernel bookkeeping data, and the objects that are managed transactionally. The `inode_data` object contains the fields commonly accessed by system calls, such as `stat`, and can be updated by a transaction by replacing the pointer in the header.

object. A naïve approach might update those pointers at commit, but the pointer writes will abort any transactions using the objects being updated.

Splitting objects into header and data. TxOS addresses object replacement by decomposing the object into a stable **header** component and a volatile, transactional **data** component. Figure 4 provides an example of this decomposition for an `inode`. The object header contains a pointer to the object's data; this pointer can be changed during commit to point to a new copy of the data object. The header itself is never replaced by a transaction, which eliminates the need to update pointers in other objects; pointers point to headers. The header can also contain data that is not accessed by transactions. For instance, the kernel garbage collection thread (kswapd) periodically scans the `inode` and `dentry` (directory entry) caches looking for cached file system data that can be reused. By keeping the data for kernel bookkeeping, such as the reference counter and the superblock lists (`i_sb_list` in Figure 4), in the header, these scans never need to access the associated `inode_data` objects and avoid restarting active transactions.

Decomposing objects into headers and data also provides the advantage of the type system ensuring that transactional code always has a speculative object. For instance, in Linux, `vfs_link` takes pointers to `inodes` and `dentries`, but in TxOS these pointers are converted to the shadow types `inode_data` and `dentry_data`. When converting a large code base such as Linux, using the type system allows the compiler to find all of the code that needs to acquire a speculative object, ensuring completeness. The type system also allows the use of interfaces that minimize the time spent looking up shadow objects. For example, the `vfs_link` and `vfs_unlink` helper functions are generally called after path name resolution, which acquires the shadow data object and passes it on. Shadow objects are acquired upon entry to the virtual file system code, eliminating the need for filesystem-specific code to acquire shadow objects.

Multiple data objects. An object can also be decomposed into multiple data payloads when it stores data that can be accessed disjointly. For instance, the `inode_header` houses both file metadata (owner, permissions, etc.) and the mapping of file blocks to cached pages in memory (`i_data`). A process may often read or write a file without updating the metadata. TxOS versions these objects separately, allowing metadata operations and data operations on the same file to execute concurrently when it is safe.

Read-only objects. In the common case, many kernel objects are only read in a transaction, such as the parent directories in a path lookup. To avoid the cost of making shadow copies, a transaction can mark object data as read-only for the length of the transaction. Each data object has a transactional reader reference count. If the reader count is non-zero, a non-transactional writer must create a new copy of the object and install it as the new stable version. The old copy of the data is garbage collected via read-copy update [28] when all transactional readers release it and after all non-transactional tasks have been descheduled. This ensures that all active references to the old, read-only version have been released before it is freed and all tasks see a consistent view of kernel data. The only caveat is that a non-transactional task that blocks must re-acquire any data objects it was using after waking, as they may have been replaced and freed by a transaction commit. Although it complicates the kernel programming model slightly, marking data objects as transactionally read is a structured way to eliminate substantial overhead for memory allocation and copying.

6.2 Conflict detection and resolution

As discussed in Section 4.2, it is important to properly serialize transactions with other transactions as well as with processes executing system calls outside of a transaction. To detect conflicts, TxOS leverages the current locking practice in Linux and augments stable objects with information about transactional readers and writers. Transactional and non-transactional threads can detect conflicts when they acquire a kernel object for writing.

Conflicts occur when a thread writes an object that is being written by a non-transactional thread, or that is read or written by a transactional thread. A locked object indicates a non-transactional writer. TxOS embeds a `tx_data` object in all shared objects that can be accessed transactionally. The `tx_data` object includes a pointer to a transactional writer and a reader list. A non-null writer pointer indicates an active transactional writer, and an empty reader list indicates there are no readers. By locking and testing the transactional readers and writer fields, TxOS detects transactional and asymmetric conflicts. When a thread detects a conflict, it calls the contention manager to arbitrate.

6.2.1 Contention Management

When a conflict is detected between two transactions or between a transaction and non-transactional thread, TxOS invokes the contention manager to resolve the conflict. The contention manager implements a policy to arbitrate conflicts among transactions, dictating which of the conflicting transactions may continue. All other conflicting transactions must abort.

As a default policy, TxOS adopts the *osprio* policy [40]. Osprio always selects the higher priority process as the winner of a conflict, eliminating priority and policy inversion in transactional conflicts. When processes with the same priority conflict, the older transaction wins (a policy known as timestamp [37]), guaranteeing liveness within a given priority level.

6.2.2 Asymmetric conflicts

A conflict between a transactional and non-transactional thread is called an **asymmetric conflict** [38]. Unlike transactional threads, non-transactional threads cannot be rolled back, so the system has fewer options when dealing with these conflicts. TxOS must have the freedom to resolve an asymmetric conflict in favor of either the transactional or non-transactional thread. Otherwise, asymmetric conflicts will undermine fairness in the system, possibly starving transactions.

While non-transactional threads cannot be rolled back, they can often be preempted, which allows them to lose conflicts with transactional threads. Kernel preemption is a recent feature of Linux that allows processes to be preemptively descheduled while executing system calls inside the kernel, unless they are inside of certain critical regions. In TxOS , non-transactional threads detect conflicts with transactional threads before they actually update state, usually when they acquire a lock for a kernel data structure. A non-transactional thread can simply deschedule itself if it loses a conflict and is in a preemptible state. If a non-transactional, non-preemptible process aborts a transaction too many times, the kernel can still prevent it from starving the transaction by placing the non-transactional process on a wait queue the next time it makes a system call. The kernel wakes it up only after the transaction commits.

Within Linux, a kernel thread can be preempted if it is not holding a spinlock and is not in an interrupt handler. TxOS has the additional restriction that it will not preempt a thread that holds one or more mutexes (or semaphores). Otherwise, TxOS risks a deadlock with the committing transaction, which might need that lock to commit. By using kernel preemption and lazy version management, TxOS has more flexibility to coordinate transactional and non-transactional threads than was possible in previous transactional operating systems.

6.3 Managing transaction state

To manage transactional state, TxOS adds transaction objects to the kernel, which store metadata and statistics for a transaction. The transaction object, shown in Figure 5, is pointed to by the kernel thread's control block (the `task_struct` in Linux). A process can have at most one active transaction, though transactions can flat nest, meaning that all nested transactions are subsumed into the enclosing transaction. Multiple tasks may also share a transaction, as discussed in the next section.

```
struct transaction {
  atomic_t tx_status;  // live, aborted, inactive
  uint64 tx_start_time;// timestamp
  uint32 retry_count;
  struct pt_regs *checkpointed_registers;
  workset_hlist  *workset_hashtable;
  deferred_ops; // operations done at commit
  undo_ops;     // operations undone at abort
};
```

Figure 5: Data contained in a system transaction object, which is pointed to by the user area (`task_struct`).

Figure 5 summarizes the fields of the transaction object. The transaction includes a status word (`tx_status`). If another thread wins a conflict with this thread it will update this word atomically (e.g. using a compare-and-swap instruction. The kernel checks the status word when attempting to add a new shadow object to its workset and checks it before commit.

If a transactional system call reaches a point where it cannot complete because of a conflict with another operation, it must immediately abort execution. This abort is required because Linux is written in an unmanaged language and cannot safely follow pointers if it does not have a consistent view of memory. To allow roll-back at arbitrary points during execution, the transaction stores the register state on the stack at the beginning of the current system call in the `checkpointed_registers` field. If the transaction is aborted midway through a system call, it restores the register state and jumps back to the top of the kernel stack (like the C library function `longjmp`). Because a transaction can hold a lock or other resource when it aborts, supporting the `longjmp`-style abort involves a small overhead to track certain events within a transaction so that they can be cleaned up on abort.

Transactions must defer certain operations until commit time, such as freeing memory, delivering signals and `dnotify` events. The `deferred_ops` field stores these events. Similarly, some operations must be undone if a transaction is aborted, such as releasing the locks it holds and freeing the memory it allocates. These are stored in the `undo_ops` field. The `tx_start_time` field is used by the contention manager (see section 6.2.1), while the `retry_count` field stores the number of times the transaction aborted.

The `workset_hashtable` is a private hashtable that stores references to all of the objects for which the transaction has private copies. Each entry in the workset contains a pointer to the stable object, a pointer to the shadow copy, information about whether the object is read-only or read-write, and a set of type-specific methods (commit, abort,

lock, unlock, release). When a transactional thread adds an object to its workset, the thread increments the reference count on the stable copy. This increment prevents the object from being unexpectedly freed while the transaction still has an active reference to it. Kernel objects are not dynamically relocatable, so ensuring a non-zero reference count is sufficient for guaranteeing that memory addresses remain consistent for the duration of the transaction.

6.4 Commit protocol

When a system transaction calls `sys_xend()`, it is ready to begin the commit protocol. The flow of the commit protocol is shown in Figure 6. In the first step, the transaction acquires all of the locks in the conflict table that protect objects in its workset. The transaction collects the locks in the following order:

1. Sorts the workset in accordance with the locking discipline (i.e., kernel virtual address).
2. Acquires all blocking locks on objects in its workset.
3. Acquires any needed global locks (e.g., the `dcache_lock`).
4. Acquires all non-blocking locks (e.g., spinlocks) on objects in its workset.

After acquiring all locks, the transaction does a final check of its status word. If it has not been set to ABORTED, then the transaction can successfully commit (this is the transactions' linearization point [22]). The committing process holds all relevant object locks during commit, thereby excluding any transactional or non-transactional threads that would compete for the same objects.

After acquiring all locks, the transaction copies its updates to the stable objects. The transaction references are removed from the objects and locks are released in the opposite order they were acquired. Between releasing spinlocks and mutexes, the transaction performs deferred operations (like memory allocations/frees and delivering fsnotify events) and performs any pending writes to stable storage. TxOS is careful to acquire blocking locks before spinlocks. Acquiring or releasing a mutex or semaphore can cause a process to sleep, and sleeping with a held spinlock can deadlock the system.

During commit, TxOS holds locks that are not otherwise held at the same time in the kernel. As a result, it extends the locking discipline slightly, for instance by requiring that `rename` locks inodes entries in order of kernel virtual address. TxOS also introduces additional fine-grained locking on objects, such as lists, that are not locked in Linux. Although these additional constraints complicate the locking discipline, they also allow TxOS to elide coarse-grained locks such as the `dcache_lock`, yielding improved performance scalability.

6.5 Abort Protocol

If a transaction detects that it loses a conflict, it must abort. The abort protocol is similar to the commit protocol, but simpler because it does not require all objects to be locked at once. If the transaction is holding any locks, it first releases them to avoid stalling other processes. The transaction then iterates over its working set and locks each object, removes any references to itself from the object's transactional state, and then unlocks the object. This process allows other transactions to access the objects in its working set. Next, the transaction frees its shadow objects and decrements the reference count on their stable counterparts. The transaction walks its undo log to release any other resources, such as memory allocated within the transaction.

6.6 User-level transactions

This section discusses the protocols that coordinate user and system transactions.

6.6.1 Lock-based STM requirements

TxOS uses a simplified variant of the two-phase commit protocol (2PC) [16] to coordinate a lock-based user-level software transactional memory (STM) with a system transaction. The TxOS commit consists of the following steps.

1. The user prepares a transaction.
2. The user requests that the system commit the transaction through the `sys_xend()` system call.
3. The system commits or aborts.
4. The system communicates the outcome to the user through the `sys_xend()` return code.
5. The user commits or aborts in accordance with the outcome of the system transaction.

This protocol naturally follows the flow of control between the user and kernel, but requires the user transaction system to support the prepared state. We define a prepared transaction as being finished (it will add no more data to its working set), safe to commit (it has not currently lost any conflicts with other threads), and guaranteed to remain able to commit (it will win all future conflicts until the end of the protocol). In other words, once a transaction is prepared, another thread must stall or rollback if it tries to perform a conflicting operation. In a system that uses locks to protect a commit, prepare is accomplished by simply holding all of the locks required for the commit during the `sys_xend()` call. On a successful commit, the system commits its state before the user, but any competing accesses to the shared state are serialized after the user commit.

6.6.2 HTM and obstruction-free STM requirements

Hardware transactional memory (HTM) and obstruction-free STM systems [21] use a single instruction (`xend` and compare-and-swap, respectively), to perform their commits. For these systems, a prepare stage is unnecessary. Instead, the commit protocol should have the kernel issue the commit instruction on behalf of the user once the kernel has validated its workset. Both the system and user level transaction now commit or abort depending upon the result of this specific commit instruction.

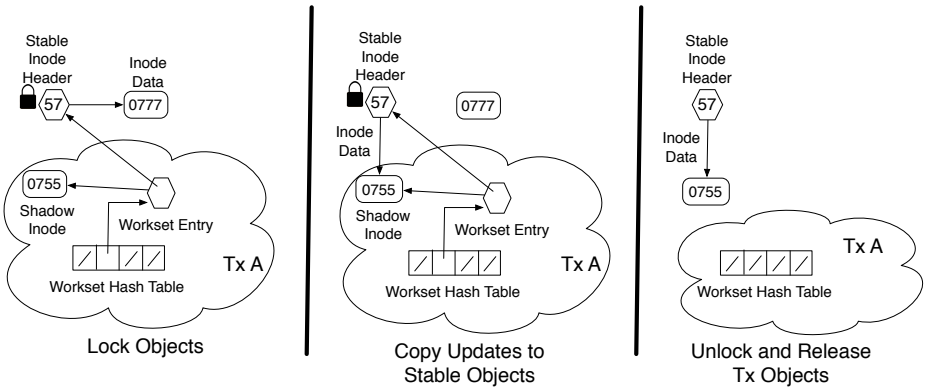

Figure 6: The major steps involved in committing Transaction A with inode 57 in its workset, changing the mode from 0777 to 0755. The commit code first locks the inode. It then replaces the inode header's data pointer to the shadow inode. Finally, Transaction A frees the resources used for transactional bookkeeping and unlocks the inode.

For HTM support, TxOS requires that the hardware allow the kernel to suspend user-initialized transactions on entry to the kernel. Every HTM proposal that supports an OS [30,40,54] contains the ability to suspend user-initiated transactions so that user and kernel addresses do not enter the same hardware transaction. Doing so would create a security vulnerability in most HTM proposals. Also, the kernel needs to be able to issue an `xend` instruction on behalf of the application.

Though TxOS supports user-level HTM, it runs on commodity hardware and does not require any special HTM support itself.

7 Evaluation

This section evaluates the overhead of system transactions in TxOS, as well as the behavior for several case studies: transactional software installation, a transactional LDAP server, a transactional `ext3` file system, eliminating TOCTTOU races, scalable atomic operations, and integration with hardware and software transactional memory.

All of our experiments are performed on a server with with 1 or 2 quad-core Intel X5355 processors (total of 4 or 8 cores) running at 2.66 GHz and 4 GB. All single-threaded experiments were performed on the 4-core machine, and scalability measurements were taken on the 8 core machine. TxOS is compared against an unmodified Linux kernel, version 2.6.22.6—the same version extended to create TxOS .

The hardware transactional memory experiments were run using MetaTM [38] on Simics version 3.0.27 [26]. The simulated machine has 16 1000 MHz CPUs, each with a 32k L1 and 4 MB L2 cache. An L1 miss costs 24 cycles and an L2 miss costs 350 cycles. We use the timestamp contention management policy and linear backoff on restart.

7.1 Transaction overhead

TxOS serializes non-transactional tasks with transactions, introducing overhead for non-transactional system calls. TxOS was designed to minimize this overhead, so that applications benefitting from transactions incur the majority of the cost. Table 7 compares the average execution times of common file system operations on TxOS to Linux. The "NoTx" is the cost of non-transactional system calls when no transactions are active, reflecting single-thread overhead increases (0-50%) due to code reorganization. The "Bgnd Tx" column shows the overhead for a non-transactional system call to check for conflicts with a concurrent transaction on the system, generally an extra 10%.

Call	Linux	NoTx	Bgnd Tx	Tx
access	2.1	2.9 (1.3×)	3.2 (1.5×)	8.1 (3.8×)
stat	2.4	3.2 (1.3×)	3.5 (1.4×)	8.3 (3.4×)
open	2.6	3.7 (1.4×)	3.8 (1.4×)	13.0 (4.9×)
unlink	6.4	9.6 (1.5×)	10.0 (1.6×)	16.0 (2.6×)
link	7.7	11.0 (1.5×)	13.0 (1.6×)	38.0 (4.9×)
mkdir	64.0	68.0 (1.1×)	72.0 (1.1×)	90.0 (1.4×)
read	2.7	3.4 (1.3×)	3.5 (1.3×)	8.7 (3.3×)
write	120.0	120.0 (1.0×)	120.0 (1.0×)	140.0 (1.2×)

Figure 7: Execution time in thousands of processor cycles of common system calls on TxOS and performance relative to Linux. **NoTX** indicates the speed of non-transactional system calls on TxOS. **Bgnd Tx** indicates the speed of non-transactional system calls when another process is running a transaction in the background, and **Tx** is the cost of a system call inside a transaction.

Figure 7 shows the worst-case performance of TxOS transactions (1-4.9×), as transaction with a single system call has no opportunity to amortize the cost of creating shadow objects and commit. The "Tx" column shows just the time for the system call, excluding the `sys_xbegin()` and `sys_xend()` system calls. In practice, a user would not write a transaction consisting of a single system call, as a single system call is already atomic.

Figure 8 shows the performance of TxOS on a range of micro-benchmarks as well as software installation. Postmark is version 1.51 with the same transactions as used by Amino [52]. The LFS small benchmark operates on 10,000 100 bytes files, and the large benchmark reads and

Bench	ACI			ACID		
	Linux	TxOS		Linux	TxOS	
postmark	.04	.12	3.0×	.05	.06	1.2 ×
lfs small						
create	4.5	5.5	1.2×	26.0	32.0	1.2×
read	1.2	1.0	.8×	63.2	62.0	1.0×
delete	.1	1.2	12.0×	3.7	3.7	1.0×
lfs large						
write seq	1.38	.34	.2×	1.58	1.67	1.1×
read seq	.04	.13	3.2×	.04	.07	1.7×
write rnd	1.60	.36	.2×	1.29	1.44	1.1×
read rnd	.07	.14	2.0×	.06	.15	2.5×
MAB						
1	.00017	.00041	2.4×	.00028	.00031	1.1×
2	.0011	.0034	2.9×	.0018	.053	2.9×
3	.54	1.10	2.0×	.51	1.61	3.2×
4	.58	.91	1.5×	.56	2.69	4.8×
dpkg	.63	.84	1.3×	2.7	3.8	1.4×
make	5.48			4.9	5.0	1.0×

Figure 8: Execution time in seconds for several transactional benchmarks on TxOS and slowdown relative to Linux. ACI represents non-durable transactions, with a baseline of ext2, and ACID represents durable transactions with a baseline of ext3 with full data journaling. Data for make ACI was not available at the time of submission.

writes a 100MB file. We scaled back the LFS small file sizes because durable transactions pathologically exacerbate a memory leak present in the Linux 2.6.22.6 ext3 journaling code. The Modified Andrew Benchmark (MAB) wraps each phase, excluding compilation, in a transaction. Dpkg and Make are software installation benchmarks that wrap the entire installation in a transaction, as discussed in the following subsection.

As the workloads get larger, the overhead of system transactions decreases, often 0–50% for workloads that run for more than a second. Benchmarks that repeatedly write files in a transaction, such as the LFS large benchmark, are more efficient than Linux because transaction commit groups the writes and presents them to the I/O scheduler at once, improving disk arm scheduling. Write-intensive workloads out-perform non-transactional writers by as much as a factor of 5×.

7.2 Software installation

By wrapping system commands in a transaction, we extend both `make install` and `dpkg`, the Debian package manager, to provide ACID properties to software installation. We test `make` with an installation of the Subversion revision control system, version 1.4.4, and test `dpkg` installing the package for OpenSSH version 4.6. The OpenSSH package was modified not to restart the daemon as the script responsible sends a signal and waits for the running daemon to die, whereas TxOS defers the signal until commit. This script could be rewritten to match the TxOS signal API in a production system.

As Figure 8 shows, the overhead for adding transactions is quite reasonable (1.0-1.4×), especially considering the qualitative benefits. For instance, by checking the return code of dpkg, our transactional wrapper was able to automatically roll back a broken Ubuntu build of OpenSSH (4.6p1-5ubuntu0.3), and no concurrent tasks were able to access the invalid package files during the installation.

Back end	Search Throughput	Add/Delete Throughput
BDB	969.3	57.6
LDIF	744.6	416.7
LDIF-TxOS	781.4	116.3

Figure 9: Throughput in queries per second of OpenLDAP's slapd server (higher is better) for a read-only and write-mostly workload. LDIF is a back end that uses flat files and provides no consistency guarantees. The BDB back end uses Berkeley DB, and LDIF-TxOS augments the LDIF back end to use system transactions on a flat file. LDIF-TxOS provides the same crash consistency guarantees as BDB with double the write throughput.

7.3 Transactional LDAP server

Many applications, such as OpenLDAP, have fairly modest concurrency control requirements for their stable data storage, yet use heavyweight solutions, such as a database server. System transactions provide a simple, lightweight solution for such applications. To demonstrate this, we modified the `slapd` server in OpenLDAP 2.3.35's LDIF (flat file) back end to use system transactions.

Table 9 shows throughput for the unmodified Berkeley DB back end, the unmodified LDIF back end, and the LDIF back end using TxOS. We used `SLAMD` to exercise the server, running in single-thread mode. While LDIF-TxOS shows a 24% slowdown over the BDB version on a read-only workload, on a write-mostly workload it improves over the BDB version by 2×. LDIF-TxOS provides higher read throughput than LDIF because of a few simple optimizations, such as caching file contents in memory. Unmodified LDIF provides higher write throughput, but provides no consistency guarantees in the presence of a crash or concurrency. LDIF-TxOS provides the same guarantees as the BDB back end with respect to concurrency and recoverability after a crash.

7.4 Transactional ext3

In addition to measuring the overheads of durable transactions, we validated the correctness of our transactional `ext3` implementation by powering off the machine during a series of transactions. After the machine is powered back on, we ran an `fsck` on the disk to validate that it was in a consistent state. To facilitate scripting, we performed these checks using Simics. At the time of submission, our system has successfully passed 763 trials, giving us a high degree of confidence that TxOS transactions correctly provide atomic, durable updates to stable storage.

7.5 Eliminating race attacks

System transactions provide a simple, deterministic method for eliminating races on system resources. To qualitatively validate this claim, we reproduced several race attacks from recent literature on Linux and validated

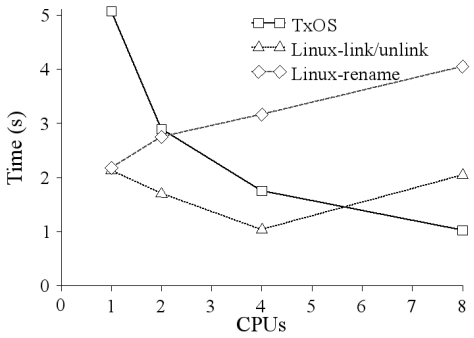

Figure 10: Time to perform 500,000 renames divided across a number of threads (lower is better). TxOS implements its renames as calls to `sys_xbegin()`, `link`, `unlink`, and `sys_xend()`, using 4 system calls for every Linux `rename` call. Despite higher single-threaded overhead, TxOS provides better scalability, outperforming Linux by 3.9× at 8 CPUs. At 8 CPUs, TxOS also outperforms a simple, non-atomic `link`/`unlink` combination on Linux by 1.9×.

that TxOS prevented the exploit.

We downloaded the attacker code used by Borisov et al. [5] to defeat Dean and Hu's probabilistic countermeasure [10]. This attack code creates memory pressure on the file system cache to force the victim to deschedule for disk I/O, thereby lengthening the amount of time spent between checking the path name and using it. This additional time allows the attacker to win nearly every time on Linux.

TxOS successfully resists this attacker by reading a consistent view of the directory structure and opening the correct file. The attacker's attempt to interpose a symbolic link creates a conflicting update that occurs after the transactional `access` check starts, so TxOS puts the attacker to sleep on the asymmetric conflict. The performance of the safe victim code on TxOS is statistically indistinguishable from the vulnerable victim on Linux.

To demonstrate that TxOS improves robustness while preserving simplicity for signal handlers, we reproduced two of the attacks described by Zalewksi [53]. The first attack, which is representative of a vulnerability present in `sendmail` up to 8.11.3 and 8.12.0.Beta7, in which an attacker can induce double-free in a signal handler. The second attack, representative of a vulnerability in the `screen` utility, which exploits lack of signal handler atomicity. Both attacks lead to root compromise; the first can be fixed by using the `sigaction` API rather than `signal`, while the second cannot. We modified the signal handlers in these attacks by wrapping handler code in `sys_xbegin`, `sys_xend` pair, which provides signal handler atomicity without requiring the programmer to change code use `sigaction`. In our experiments, TxOS serializes handler code with respect to other system operations, and is therefore able to withstand both attacks.

7.6 Concurrent performance

System calls like `rename` and `open` have been used as *ad hoc* solutions for the lack of general-purpose atomic actions. These system calls have become semantically heavy, resulting in complex implementations whose performance does not scale. As an example in Linux, `rename` has to serialize all cross-directory renames on a single file-system-wide mutex because finer-grained locking would risk deadlock.

Transactions allow simpler, semantically lighter system calls to be combined to perform heavier weight operations yielding better performance scalability and a simpler implementation. Figure 10 compares the unmodified Linux implementation of `rename` to calling `sys_xbegin()`, `link`, `unlink`, and `sys_xend()` in TxOS. TxOS has worse single-thread performance because it makes four system calls for each Linux system call. But TxOS quickly recovers the performance, performing within 6% at 2 CPUs and out-performing `rename` by 3.9× 8 CPUs. We also compare the TxOS atomic link/unlink to a non-atomic link/unlink pair on Linux, in which TxOS outperforms Linux by a factor of 1.9× at 8CPUs. The scalability is directly due to TxOS using fine-grained locking to implement transactions, while Linux must use conservative, coarse-grained locks in order to keep implementation complexity reasonable.

7.7 HTM with system calls

To evaluate integration with hardware transactional memory, we use the genome benchmark from the STAMP benchmark suite [7], to fix a memory leak in the distributed version. Genome allocates memory during a transaction, and the allocation sometimes calls `mmap`. When the transaction restarts, `mmap` gets called repeatedly, leaking memory. With TxOS, the `mmap` is made part of a system transaction and is properly rolled back when the user-level transaction aborts.

Table 3 shows the execution time, number of system calls within a transaction, and the number of allocated pages at the end of the benchmark for both TxOS and unmodified Linux running on MetaTM. TxOS rolls back `mmap` in unsuccessful transactions, allocating 3× less heap memory to the application, without effecting performance. No source code or libc changes are required for TxOS to detect that `mmap` is transactional.

The possibility of an `mmap` leak is a known problem [54], with several proposed solutions, including open nesting and a transactional pause instruction. All solutions complicate the programming model, the hardware, or both. System transactions address the memory leak with the simplest hardware model and user API.

7.8 Integration with software transactional memory

We integrated a Java-based software transactional memory system (DATM-J [39]) with system transactions. We extend DATM-J to use the system call API provided by TxOS. The only modification to the STM is to follow the

Execution Time		System Calls		Allocated Pages	
TxOS	Linux	TxOS	Linux	TxOS	Linux
.05	.05	1,084	1,024	8,755	25,876

Table 3: Execution Time, number of system calls, and allocated pages for the genome benchmark on the MetaTM HTM simulator with 16 processors.

commit protocol when committing a user level transaction that invokes a system call, as outlined in Section 4.5.

We tested the integration of DATM-J with TxOS by modifying Tornado, a multi-threaded web server that is publicly available on sourceforge, to use transactions. Tornado protects its data structures with STM transactions, and the STM transparently protects concurrent reads and writes to its data files from interfering with each other. Compared to the original code, augmented to use file locking to protect against concurrent file writes, transactions perform up to 47% better with 7 threads.

8 Conclusion

Adding efficient transactions to the Linux system call API provides a general-purpose, natural way for programmers to synchronize access to system resources, a problem currently solved in an *ad hoc* manner. This paper demonstrates how system transactions can solve a number of important, long-standing problems from a number of domains, including file system races and supporting system calls within transactional memory, while maintaining scalable performance.

References

[1] http://www.employees.org/ satch/ssh/faq/TheWholeSSHFAQ.html.

[2] http://plash.beasts.org/wiki/PlashIssues/ConnectRaceCondition.

[3] M. K. Aguilera, A. Merchant, M. Shah, A. Veitch, and C. Karamanolis. Sinfonia: a new paradigm for building scalable distributed systems. In *SOSP*, New York, NY, USA, 2007. ACM.

[4] D. J. Bernstein. Some thoughts on security after ten years of qmail 1.0. In *CSAW*, 2007.

[5] N. Borisov, R. Johnson, N. Sastry, and D. Wagner. Fixing races for fun and profit: How to abuse atime. In *USENIX Security*, 2005.

[6] X. Cai, Y. Gui, and R. Johnson. Exploiting unix file-system races via algorithmic complexity attacks. *Oakland*, 2009.

[7] C. Cao Minh, M. Trautmann, J. Chung, A. McDonald, N. Bronson, J. Casper, C. Kozyrakis, and K. Olukotun. An effective hybrid transactional memory system with strong isolation guarantees. In *ISCA*. Jun 2007.

[8] C. Cowan, S. Beattie, C. Wright, and G. Kroah-Hartman. RaceGuard: Kernel protection from temporary file race vulnerabilities. In *USENIX Security 2001*, pages 165–176.

[9] P. Damron, A. Fedorova, Y. Lev, V. Luchangco, M. Moir, and D. Nussbaum. Hybrid transactional memory. In *ASPLOS*, 2006.

[10] D. Dean and A. J. Hu. Fixing races for fun and profit: how to use access(2). In *USENIX Security*, pages 14–26, 2004.

[11] D. Dice, O. Shalev, and N. Shavit. Transactional locking II. In *DISC*, pages 194–208, 2006.

[12] U. Drepper. Secure file descriptor handling. In *LiveJournal*, 08.

[13] E. Gal and S. Toledo. A transactional flash file system for microcontrollers. In *USENIX*, 2005.

[14] N. Gehani, H. V. Jagadish, and W. D. Roome. Odefs: A file system interface to an object-oriented database. In *VLDB*, 1994.

[15] S. Ghemawat, H. Gobioff, and S.-T. Leung. The google file system. *SOSP*, 2003.

[16] J. Gray. Notes on data base operating systems. In *Operating Systems, An Advanced Course*. Springer-Verlag, 1978.

[17] J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann, 1993.

[18] L. Hammond, V. Wong, M. Chen, B. Carlstrom, J. Davis, B. Hertzberg, M. Prabhu, H. Wijaya, C. Kozyrakis, and K. Olukotun. Transactional memory coherence and consistency. In *ISCA*, June 2004.

[19] R. Haskin, Y. Malachi, and G. Chan. Recovery management in QuickSilver. *ACM Trans. Comput. Syst.*, 6(1):82–108, 1988.

[20] M. Herlihy and E. Koskinen. Transactional boosting: A methodology for highly-concurrent transactional objects. In *PPoPP*, 2008.

[21] M. Herlihy, V. Luchangco, M. Moir, and I. William N. Scherer. Software transactional memory for dynamic-sized data structures. In *PODC*, 2003.

[22] M. P. Herlihy and J. M. Wing. Linearizability: a correctness condition for concurrent objects. *ACM TOPLAS*, 12(3), 1990.

[23] M. Kulkarni, K. Pingali, B. Walter, G. Ramanarayanan, K. Bala, and L. P. Chew. Optimistic parallelism requires abstractions. In *PLDI*, New York, NY, USA, 2007. ACM Press.

[24] J. Larus and R. Rajwar. *Transactional Memory*. Morgan & Claypool, 2006.

[25] B. Liskov, D. Curtis, P. Johnson, and R. Scheifer. Implementation of Argus. *SOSP*, 1987.

[26] P. Magnusson, M. Christianson, and J. E. et al. Simics: A full system simulation platform. In *IEEE Computer*, Feb 2002.

[27] P. McDougall. Microsoft pulls buggy windows vista sp1 files. In *Information Week*. `http://www.informationweek.com/story/showArticle.jhtml?articleID=206800819`.

[28] P. E. McKenney. *Exploiting Deferred Destruction: An Analysis of Read-Copy Update Techniques in Operating System Kernels*. PhD thesis, 2004.

[29] Microsoft. What is system restore. 2008. http://support.microsoft.com/kb/959063.

[30] K. E. Moore, J. Bobba, M. J. Moravan, M. D. Hill, and D. A. Wood. LogTM: Log-based transactional memory. In *HPCA*, 2006.

[31] N. Murphy, M. Tonkelowitz, and M. Vernal. The design and implementation of the database file system, 2002.

[32] E. B. Nightingale, P. M. Chen, and J. Flinn. Speculative execution in a distributed file system. In *SOSP*, 2005.

[33] E. B. Nightingale, D. Peek, P. M. Chen, and J. Flinn. Parallelizing security checks on commodity hardware. In *ASPLOS*, 2008.

[34] NIST. National Vulnerability Database. `http://nvd.nist.gov/`, 2008.

[35] J. Olson. Enhance your apps with file system transactions. *MSDN Magazine*, July 2007. http://msdn2.microsoft.com/en-us/magazine/cc163388.aspx.

[36] M. A. Olson. The design and implementation of the inversion file system. In *USENIX*, 1993.

[37] R. Rajwar and J. R. Goodman. Transactional lock-free execution of lock-based programs. *ASPLOS*, 2002.

[38] H. Ramadan, C. Rossbach, D. Porter, O. Hofmann, A. Bhandari, and E. Witchel. MetaTM/TxLinux: Transactional memory for an operating system. In *ISCA*, 2007.

[39] H. E. Ramadan, I. Roy, M. Herlihy, and E. Witchel. Committing conflicting transactions in an STM. *PPoPP*, 2009.

[40] C. Rossbach, O. Hofmann, D. Porter, H. Ramadan, A. Bhandari, and E. Witchel. TxLinux: Using and managing transactional memory in an operating system. In *SOSP*, 2007.

[41] F. Schmuck and J. Wylie. Experience with transactions in QuickSilver. In *SOSP*. ACM, 1991.

[42] R. Sears and E. Brewer. Stasis: Flexible transactional storage. In *OSDI*, 2006.

[43] M. I. Seltzer. Transaction support in a log-structured file system. In *9th International Conference on Data Engineering*, 1993.

[44] A. Z. Spector, D. Daniels, D. Duchamp, J. L. Eppinger, and R. Pausch. Distributed transactions for reliable systems. In *SOSP*, 1985.

[45] R. Spillane, S. Gaikwad, M. Chinni, E. Zadok, and C. P. Wright. Enabling transactional file access via lightweight kernel extensions. *FAST*, 2009.

[46] Y.-Y. Su, M. Attariyan, and J. Flinn. Autobash: improving configuration management with operating system causality analysis. In *SOSP*, 2007.
[47] D. Tsafrir, T. Hertz, D. Wagner, and D. D. Silva. Portably preventing file race attacks with user-mode path resolution. Technical report, IBM Research Report, 2008.
[48] D. Tsafrir, T. Hertz, D. Wagner, and D. D. Silva. Portably solving file TOCTTOU races with hardness amplification. In *FAST*, 2008.
[49] E. Tsyrklevich and B. Yee. Dynamic detection and prevention of race conditions in file accesses. In *USENIX Security*, 2003.
[50] H. Volos, A. J. Tack, N. Goyal, M. M. Swift, and A. Welc. xCalls: Safe I/O in memory transactions. In *EuroSys*, 2009.
[51] M. J. Weinstein, J. Thomas W. Page, B. K. Livezey, and G. J. Popek. Transactions and synchronization in a distributed operating system. In *SOSP*, 1985.
[52] C. P. Wright, R. Spillane, G. Sivathanu, and E. Zadok. Extending ACID semantics to the file system. *Trans. Storage*, 3(2):4, 2007.
[53] M. Zalewski. Delivering signals for fun and profit. 2001.
[54] C. Zilles and L. Baugh. Extending hardware transactional memory to support non-busy waiting and non-transactional actions. In *TRANSACT*, Jun 2006.