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
{porterde,osh,rossbach,abenn1,witchel}@cs.utexas.edu

ABSTRACT

Applications must be able to synchronize accesses to operating system resources in order to ensure correctness in the face of concurrency and system failures. *System transactions* allow the programmer to specify updates to heterogeneous system resources with the OS guaranteeing atomicity, consistency, isolation, and durability (ACID). System transactions efficiently and cleanly solve persistent concurrency problems that are difficult to address with other techniques. For example, system transactions eliminate security vulnerabilities in the file system that are caused by time-of-check-to-time-of-use (TOCTTOU) race conditions. System transactions enable an unsuccessful software installation to roll back without disturbing concurrent, independent updates to the file system.

This paper describes TxOS, a variant of Linux 2.6.22 that implements system transactions. TxOS uses new implementation techniques to provide fast, serializable transactions with strong isolation and fairness between system transactions and non-transactional activity. The prototype demonstrates that a mature OS running on commodity hardware can provide system transactions at a reasonable performance cost. For instance, a transactional installation of OpenSSH incurs only 10% overhead, and a non-transactional compilation of Linux incurs negligible overhead on TxOS. By making transactions a central OS abstraction, TxOS enables new transactional services. For example, one developer prototyped a transactional `ext3` file system in less than one month.

Categories and Subject Descriptors

D.4.1 [**Operating Systems**]: Process Management—*synchronization*; D.1.3 [**Programming Techniques**]: Concurrent Programming; D.4.7 [**Operating Systems**]: Organization and Design

General Terms

Design, Performance, Security

Keywords

Transactions, Operating Systems, TxOS, Race Conditions, Transactional Memory

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
SOSP'09, October 11–14, 2009, Big Sky, Montana, USA.
Copyright 2009 ACM 978-1-60558-752-3/09/10 ...$10.00.

1. INTRODUCTION

Applications often need to group accesses to operating system resources (such as files and signals) into logical units, just as multithreaded applications must group accesses to shared data structures into critical regions. For example, local user and group accounts on Linux and similar operating systems are stored across three files that need to be mutually consistent: `/etc/passwd`, `/etc/shadow`, and `/etc/group`.

Applications currently struggle to make consistent updates to system resources. In current operating systems, individual system calls are generally atomic and isolated from the rest of the system, but it is difficult, if not impossible, to condense complex operations into a single system call. In simple cases, programmers can use a powerful, single system call like `rename`, which atomically replaces the contents of a file. For more complex updates, options like file locking are clumsy and difficult to program. In the presence of concurrency, the problem is exacerbated because existing interfaces are often insufficient to protect a series of system calls from interference by buggy or malicious applications. With the current proliferation of multi-core processors, concurrent processing is becoming ubiquitous, exposing the inability of the traditional system call interface to ensure consistent accesses.

In the example of managing local user accounts, developers spend substantial effort creating tools that minimize, but fail to eliminate consistency problems. The `vipw` and `useradd` utilities help ensure that user account databases are formatted correctly and mutually consistent. To address concurrency in the system, these tools create lock files for mutual exclusion. A careless administrator, however, can corrupt the files by simply editing them directly. The tools also use the `sync()` and `rename` commands to ensure that an individual file is not corrupted if the system crashes, but cannot ensure that an update to multiple files is consistently propagated. For instance, suppose a system crashes after `useradd` writes `/etc/passwd` but before it writes `/etc/shadow`. After rebooting the system, the new user will not be able to log on, yet `useradd` will fail because it thinks the user already exists, leaving the system administrator to manually repair the database files. The proliferation of tools to mitigate such a simple problem, as well as the tools' incompleteness, indicate that developers need a better API for consistent system accesses.

In practice, OS maintainers address the lack of concurrency control in the system call API in an *ad hoc* manner: new system calls and complex interfaces are added to solve new problems as they arise. The critical problem of eliminating file system race conditions has motivated Solaris and Linux developers to add over a dozen new system calls, such as `openat`, over the last seven years. Linux maintainers added a close-on-exec flag to fifteen system calls in a recent version of Linux [13] to eliminate a race condition be-

tween calls to open and fcntl. Individual file systems have introduced new operations to address consistency needs: the Google File System supports atomic append operations [16], while Windows recently adopted support for transactions in NTFS and the Windows registry [44]. Users should not be required to lobby OS developers for new system calls and file system features to meet their concurrent programming needs. Why not allow users to solve their own problems by supporting composition of multiple system calls into arbitrary atomic and isolated units?

This paper proposes *system transactions* to allow programmers to group accesses to system resources into logical units, which execute with atomicity, consistency, isolation, and durability (ACID). System transactions are easy to use: code regions with consistency constraints are enclosed within the 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 accesses 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.

This paper describes an implementation of system transactions on Linux called **TxOS**, which provides transactional semantics for OS resources, including the file system, memory management, signals, and process creation. To efficiently provide strong guarantees, the TxOS implementation redesigns several key OS data structures and internal subsystem interfaces. By making transactions a core OS abstraction, TxOS enables user and OS developers to create powerful applications and services. For example, given an initial implementation of TxOS, a single developer needed less than a month to prototype a transactional ext3 file system.

This paper makes two primary contributions. First, it describes a new approach to OS implementation that supports efficient transactions on commodity hardware with strong atomicity and isolation guarantees. Secondly, it demonstrates a prototype implementation of system transactions (TxOS) whose strong guarantees and good performance enable new solutions to systems problems such as:

1. Eliminating security vulnerabilities exploited by file system race conditions.
2. Rolling back an unsuccessful software install or upgrade without disturbing concurrent, unrelated updates. A transactional dpkg install adds only 10% overhead for this increase in safety.
3. Providing a lightweight alternative to a database for concurrency management and crash consistency, yielding simpler application code and system administration. Replacing Berkeley DB with flat files and system transactions as the storage back-end for the OpenLDAP directory service improves performance on write-mostly workloads by 2–4×.
4. Allowing user-level transactional programs to make system calls during a transaction.

The remainder of the paper is structured as follows. Section 2 provides motivating use-cases for system transactions and Section 3 describes programming with system transactions and their implementation in TxOS. Section 4 describes the design of TxOS, Section 5 provides kernel implementation details, and Section 6 describes how certain key subsystems provide transactional semantics. Section 7 measures the performance overhead of system transactions and evaluates TxOS in a number of application case studies. Section 8 positions TxOS in related work and Section 9 concludes.

2. MOTIVATING EXAMPLES

A range of seemingly unrelated application limitations share a root cause—the lack of a general mechanism to ensure consistent access to system resources. This section reviews two common application consistency problems and how system transactions remedy those problems. System transactions allow software installations to recover from failures without disrupting concurrent, independent updates to the file system. System transactions also eliminate race conditions inherent in the file system API, which can be exploited to undermine security.

2.1 Software installation or upgrade

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 recalled a prerequisite patch for Vista service pack 1 because it caused an endless cycle of boots and reboots [28]. More generally, a partial upgrade can leave a system in an unusable state.

Current systems are adopting solutions that mitigate these problems, but each has its own drawbacks. Microsoft Windows and other systems provide a checkpoint-based solution to the software update problem. Users can take a checkpoint of disk state before they install software: if something goes wrong, they roll back to the checkpoint. Windows checkpoints certain key structures, like the registry and some system files [30]. Other systems, like ZFS's apt-clone, checkpoint the entire file system. If the software installation fails, the system restores the pre-installation file system image, erasing file system updates that are concurrent but independent from the software installation. Partial checkpointing mitigates this problem, but loses the ability to recover from application installations that corrupt files not checkpointed by the system. Moreover, the user or the system must create and manage the disk-based checkpoints to make sure a valid image is always available. Finally, if a bad installation affects volatile system state, errant programs can corrupt files unrelated to the failed installation. Collectively, these problems severely decrease the usability of checkpoint-based solutions.

System transactions provide a simple interface to address these software installation problems. A user executes the software installation or update within a transaction, which isolates the rest of the system until the installation successfully completes. If the installation or upgrade needs to be rolled back, independent updates made concurrently remain undisturbed.

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. Common in setuid programs, this pattern is 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 OS API provides no way for the application to tell the operating system that it needs a consistent view of the file system's name space.

Although most common in the file system, system API races, or time-of-check-to-time-of-use (TOCTTOU) races, can be exploited in other OS resources. 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 al-

```
     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`.

lowed 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 [57].

While TOCTTOU vulnerabilities are conceptually simple, they pervade 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 [37]. Further, recent work by Cai et al. [7] exploits fundamental flaws to defeat two major classes of TOCTTOU countermeasures: dynamic race detectors in the kernel [53] and probabilistic user-space race detectors [52]. This continuous arms race of measure and countermeasure suggests that TOCTTOU attacks can be eliminated only by changing the API.

In practice, such 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. [51] 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, they are incompatible with the `O_CREAT` flag to `open` that is used to prevent exploits on temporary file creation [9].

Fixing race conditions as they arise is not an effective long-term strategy. Complicating the API in the name of security is 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 eliminates API race conditions.

3. OVERVIEW

System transactions are designed to provide programmers with a natural abstraction for ensuring consistent access to system resources. This section describes the API, semantics, and behavior of system transactions, followed by an overview of how system transactions are supported in TxOS, our prototype implementation of system transactions within Linux.

3.1 System transactions

System transactions provide ACID semantics for updates to OS resources, such as files, pipes, and signals. In this programming model, both transactional and non-transactional system calls may access the same system state; the OS is responsible for ensuring that these accesses are correctly serialized and contention is arbitrated fairly. The interface for system transactions is intuitive and simple, allowing a programmer to wrap a block of unmodified code in a transaction simply by adding `sys_xbegin()` and `sys_xend()`.

3.1.1 System transaction semantics

System transactions share several properties developers are likely familiar with from database transactions. System transactions are serializable and recoverable. Reads are only allowed to committed data and are repeatable, which corresponds to the highest database isolation level (level 3 [18]). Transactions are atomic (the system can always roll back to a pre-transaction state) and durable (transaction results, once committed, survive system crashes).

To ensure isolation, the kernel enforces the invariant that a kernel object may only have one writer at a time, excepting containers, which allow multiple writers to disjoint entries. Two concurrent system transactions cannot both successfully commit if they access the same kernel objects and at least one of the accesses is a write. Such transactions are said to **conflict** and the system will detect the conflict and abort one of the transactions. Non-transactional updates to objects read or written by an active system transaction are also prevented by the system. Either the system suspends the non-transactional work before the update, or it aborts the transaction. By preventing conflicting accesses to the same kernel object, the system provides conflict serializability, which is commonly used to enforce serializability efficiently.

System transactions make durability optional because durability often increases transaction commit latency and the programmer does not always need it. The increased commit latency comes from flushing data to a slow block storage device, like a disk. Eliminating the TOCTTOU race in the file system namespace is an example of a system transaction that does not require durability. Durability for system transactions in TxOS is under the control of the programmer, using a flag to `sys_xbegin()` (Table 2).

Each kernel thread may execute a system transaction. Transactional updates are isolated from all other kernel threads, including threads in different processes. We call a kernel thread executing a system transaction a transactional kernel thread.

3.1.2 Interaction of transactional and non-transactional threads

The OS serializes system transactions and non-transactional system calls, providing the strongest guarantees and most intuitive semantics [18] to the programmer. The serialization of transactional and non-transactional updates to the same resources is called **strong isolation** [5]. Previous OS transaction designs have left the interaction of transactions with non-transactional activity semantically murky. Intuitive semantics for mixing transactional and non-transactional access to the same resources is crucial to maintaining a simple interface to system resources. Strong isolation prevents unexpected behavior due to non-transactional and transactional applications accessing the same system resources.

The presence of system transactions does not change the behavior of non-transactional activity in the underlying operating system. While most system calls are already isolated and atomic, there are important exceptions. For example, Linux does not serialize `read` with `write`. On an OS with system transactions, non-

transactional system calls can still exhibit non-serializable behavior with respect to each other, but non-transactional system calls serialize with transactions. For example, one or more calls to `read` in a system transaction will correctly serialize with a non-transactional `write`.

3.1.3 System transaction progress

The operating system guarantees that system transactions do not livelock with other system transactions. When two transactions, A and B, cannot both commit, the system selects one to restart (let's say B in this example), and ensures its decision remains consistent. If A continues and B restarts and again conflicts with A, the OS will again restart B. See § 5.2.1 for details.

Guaranteeing progress for transactional threads in the presence of non-transactional threads requires support from the OS. If an OS supports preemption of kernel threads (present in Linux 2.4 and 2.6 since 2004), then it can guarantee progress for long running transactions by preempting non-transactional threads that would impede progress of the transaction.

The OS has several mechanisms to regulate the progress of transactions, but the use of these mechanisms is a matter of policy. For instance, allowing a long running transaction to isolate all system resources indefinitely is undesirable, so the OS may want a policy that limits the size of a transaction. Limiting a transaction that over-consumes system resources is analogous to controlling any process that abuses system resources, such as memory, disk space, or kernel threads.

3.1.4 System transactions for system state

Although system transactions provide ACID semantics for system state, they do not provide these semantics for application state. System state includes OS data structures and device state stored in the operating system's address space, whereas application state includes only the data structures stored in the application's address space. When a system transaction aborts, the OS restores the kernel state to its pre-transaction state, but it does not revert application state.

For most applications, we expect programmers will use a library or runtime system that transparently manages application state as well as system transactions. In simple cases, such as the TOCTTOU example, the developer could manage application state herself. TxOS provides single-threaded applications with an automatic checkpoint and restore mechanism for the application's address space that marks the pages copy-on-write (similar to Speculator [35]), which can be enabled with a flag to `sys_xbegin()` (Table 2). In Section 4.3, we describe how system transactions integrate with hardware and software transactional memory, providing a complete transactional programming model for multi-threaded applications.

3.1.5 Communication model

Code that communicates outside of a transaction and requires a response cannot be encapsulated into a single transaction. Communication outside of a transaction violates isolation. For example, a transaction may send a message to a non-transactional thread over an IPC channel and the system might buffer the message until commit. If the transaction waits for a reply on the same channel, the application will deadlock. The programmer is responsible for avoiding this send/reply idiom within a transaction.

Communication among threads within the same transaction is unrestricted. This paper only considers system transactions on a single machine, but future work could allow system transactions to span multiple machines.

Subsystem	Tot.	Part.	Examples
Credentials	34	1	getuid, getcpu, setrlimit (partial)
Processes	13	3	fork, vfork, clone, exit, exec (partial)
Communication	15	0	rt_sigaction, rt_sigprocmask, pipe
Filesystem	61	4	link, access, stat, chroot, dup, open, close, write, lseek
Other	13	6	time, nanosleep, ioctl (partial), mmap2 (partial)
Totals	136	14	Grand total: 150

Unsupported		
Processes	33	nice, uselib, iopl, sched_yield, capget
Memory	15	brk, mprotect, mremap, madvise
Filesystem	31	mount, sync, flock, setxattr, io_setup, inotify
File Descriptors	14	splice, tee, sendfile, select, poll
Communication	8	socket, ipc, mq_open, mq_unlink
Timers/Signals	12	alarm, sigaltstack, timer_create
Administration	22	swapon, reboot, init_module, settimeofday
Misc	18	ptrace, futex, times, vm86, newuname
Total	153	

Table 1: Summary of system calls that TxOS completely supports (Tot.) and partially supports (Part.) in transactions, followed by system calls with no transaction support. Partial support indicates that some (but not all) execution paths for the system call have full transactional semantics. Linux 2.6.22.6 on the i386 architecture has 303 total system calls.

3.2 TxOS overview

TxOS implements system transactions by isolating data read and written in a transaction using existing kernel memory buffers and data structures. 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. In TxOS, transactions must fit into main memory, although this limit could be raised in future work by swapping uncommitted transaction state to disk.

TxOS isolates updates to kernel data structures 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 (§4.2). TxOS's isolation mechanisms are optimistic, allowing concurrent transactions on the assumption that conflicts are rare.

Table 1 summarizes the system calls and resources for which TxOS supports transactional semantics, including the file system, process and credential management, signals, and pipes. A partially supported system call means that some processing paths are fully transactional, and some are not. For example, `ioctl` is essentially a large switch statement, and TxOS does not support transactional semantics for every case. When a partially supported call cannot support transactional semantics, or an unsupported call is issued, the system logs a warning or aborts the transaction, depending on the flags passed to `sys_xbegin()`.

Ideal support for system transactions would include every reasonable system call. TxOS supports a subset of Linux system calls as shown in Table 1. The count of 150 supported system calls shows the relative maturity of the prototype, but also indicates that it is incomplete. The count of unsupported system calls does not proportionately represent the importance or challenge of the re-

Function Name	Description
int sys_xbegin (int flags)	Begin a transaction. The flags specify transactional behavior, including automatically restarting the transaction after an abort, ensuring that committed results are on stable storage (durable), and aborting if an unsupported system call is issued. 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 2: TxOS API

maining work because many resources, such as network sockets, IPC, etc., primarily use the common file system interfaces. For instance, extending transactions to include networking (a real challenge) would increase the count of supported calls by 5, whereas transaction support for extended file attributes (a fairly straightforward extension) would add 12 supported system calls. The remaining count of system calls falls into three categories: substantial extensions (memory management, communication), straightforward, but perhaps less common or important (process management, timers, most remaining file interfaces), and operations that are highly unlikely to be useful inside a transaction (e.g., `reboot`, `mount`, `init_module`, etc.). TxOS supports transactional semantics for enough kernel subsystems to demonstrate the power and utility of system transactions.

4. TXOS DESIGN

System transactions guarantee strong isolation for transactions, while retaining good performance and simple interfaces. This section outlines how the TxOS design achieves these goals.

4.1 Interoperability and fairness

TxOS allows flexible interaction between transactional and non-transaction kernel threads. TxOS efficiently provides strong isolation inside the kernel by requiring all system calls to follow the same locking discipline, and by requiring that transactions annotate accessed kernel objects. When a thread, transactional or non-transactional, accesses a kernel 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** [41]) 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.

4.2 Managing transactional state

Databases and historical transactional operating systems typically update data in place and maintain an undo log. This approach is called **eager version management** [25]. These systems isolate transactions by locking data when it is accessed and holding the lock until commit. This technique is called two-phase locking, and it usually employs locks that distinguish read and write accesses. Because applications generally do not have a globally consistent order for data accesses, these systems can deadlock. For example, one thread might read file A then write file B, while a different thread might read file B, then write file A.

The possibility of deadlock complicates the programming model of eager versioning 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 process its undo log (to restore the pre-transaction state) before it can safely proceed. This wait jeopardizes 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. Applications never hold kernel locks across system calls. Lazy versioning requires TxOS to hold locks only long enough to make a private copy of the relevant data structure. By enforcing a global ordering for kernel locks, TxOS avoids deadlock. TxOS can abort transactions instantly—the winner of a conflict does not incur latency for the aborted transaction to process its undo log.

The primary disadvantage of lazy versioning is the commit latency due to copying transactional updates from the speculative version to the stable version of the data structures. As we discuss in Section 5, TxOS minimizes this overhead by splitting objects, turning a `memcpy` of the entire object into a pointer copy.

4.3 Integration with transactional memory

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, however, are designed to provide efficient transactional semantics to memory modifications by a thread, but cannot isolate or roll back system calls. Integrating user and system transactions creates a simple and complete transactional programming model.

System transactions fix one of the most troublesome limitations of transactional memory systems—that system calls are disallowed during user transactions because they violate transactional semantics. System calls on traditional operating system are not isolated, and they cannot be rolled back if a transaction fails. For example, a file append performed inside a hardware or software user transaction can occur an arbitrary number of times. Each time the user-level transaction aborts and retries, it repeats the append.

On a TM system integrated with TxOS, 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 application'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. The programmer sees the simple abstraction of an atomic block that can contain updates to user data structures and system calls. See Section 5.6 for implementation details and Sections 7.8 and 7.9 for evaluation.

5. TxOS KERNEL IMPLEMENTATION

This section describes how system transactions are implemented in the TxOS kernel. TxOS provides transactional semantics for 150 of 303 system calls in Linux, presented in Table 1. The supported system calls include process creation and termination, cre-

```
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 pages
  tx_data xobj;   // 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 2: 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.

dential management operations, 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.

5.1 Versioning data

TxOS maintains multiple versions of kernel data structures so that system transactions can isolate the effects of system calls until transactions commit, and in order to undo the effects of transactions if they cannot complete. Data structures private to a process, such as the current user id or the file descriptor table, are versioned with a simple checkpoint and restore scheme. For shared kernel data structures, however, TxOS implements a versioning system that borrows techniques from software transactional memory systems [21] and other recent concurrent programming systems [24].

When a transaction accesses a shared kernel object, such as an `inode`, it acquires a private copy of the object, called a **shadow** object. All system calls within the transaction use this shadow object in place of the **stable** object until the transaction commits or aborts. The use of shadow objects ensures that transactions always have a consistent view of the system state. When the transaction commits, the shadow objects replace their stable counterparts. If a transaction cannot complete, it simply discards its shadow objects.

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 object. A naïve system might update the pointers to an object when that object is committed. Unfortunately, updating the pointers means writing the objects that contain those pointers. By writing to the pointing objects, the transaction may create conflicting accesses and abort otherwise non-conflicting transactions. For two concurrent transactions to successfully commit in TxOS, they must write disjoint objects.

Splitting objects into header and data.

In order to allow efficient commit of lazy versioned data, TxOS decomposes objects into a stable **header** component and a volatile, transactional **data** component. Figure 2 provides an example of this decomposition for an `inode`. The object header contains a pointer to the object's data; transactions commit changes to an object by replacing this pointer in the header to a modified 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 to reuse. By keeping the data for kernel bookkeeping, such as the reference count and the superblock list (`i_sb_list` in Figure 2), in the header, these scans never 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, the virtual file system function `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 modifying 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, when the path name resolution code initially acquires shadow data objects, it then passes these shadow objects directly to helper functions such as `vfs_link` and `vfs_unlink`. The virtual file system code acquires shadow objects once on entry and passes them to lower layers, minimizing the need for filesystem-specific code to reacquire the shadow objects.

Multiple data objects.

TxOS decomposes an object into multiple data payloads when it houses data that can be accessed disjointly. For instance, the `inode_header` contains 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.

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, kernel code can specify read-only access to an object, which marks the object data as read-only for the length of the transaction. Each data object has a transactional reader reference count. If a writer wins a conflict for an object with a non-zero reader count, it must create a new copy of the object and install it as the new stable version. The OS garbage collects the old copy via read-copy update (RCU) [29] when all transactional readers release it and after all non-transactional tasks have been descheduled. This constraint 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 read-only in a transaction is a structured way to eliminate substantial overhead for memory allocation and copying. Special support for read-mostly transactions is a common optimization in transactional systems, and RCU is a technique to support efficient, concurrent access to read-mostly data.

5.2 Conflict detection and resolution

As discussed in Section 4.1, TxOS serializes transactions with non-transactional activity as well as with other transactions. TxOS serializes non-transactional accesses to kernel objects with transactions by leveraging the current locking practice in Linux and augmenting stable objects with information about transactional readers and writers. Both transactional and non-transactional threads use this information to detect accesses that would violate conflict serializability when they acquire a kernel object.

Conflicts occur when a transaction attempts to write an object that has been read or written by another transaction. An asymmetric conflict is defined similarly: a non-transactional thread attempts to write an object a transaction has read or written, or vice versa. TxOS embeds a `tx_data` object in the header portion of all shared kernel objects that can be accessed within a transaction. 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. Locks prevent transactions from acquiring an object that is concurrently accessed by a non-transactional thread. When a thread detects a conflict, TxOS uses these fields to determine which transactions are in conflict; the conflict is then arbitrated by the contention manager (§5.2.1). Note that the reader list is attached to the stable header object, whereas the reader count (§5.1) is used for garbage collecting obsolete data objects. By locking and testing the transactional readers and writer fields, TxOS detects transactional and asymmetric conflicts.

5.2.1 Contention Management

When a conflict is detected between two transactions or between a transaction and a non-transactional thread, TxOS invokes the contention manager to resolve the conflict. The contention manager is kernel code that 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 [43]. Osprio always selects the process with the higher scheduling priority 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 [40]), guaranteeing liveness for transactions within a given priority level.

5.2.2 Asymmetric conflicts

A conflict between a transactional and non-transactional thread is called an asymmetric conflict. Transactional threads can always be aborted and rolled back, but non-transactional threads cannot be rolled back. TxOS must have the freedom to resolve an asymmetric conflict in favor of the transactional thread, otherwise asymmetric conflicts will always win, undermining fairness in the system and 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 the kernel to preemptively deschedule threads 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 reschedules the non-transactional process only after the transaction commits.

Linux can preempt a kernel thread if the thread is not holding a spinlock and is not in an interrupt handler. TxOS has the additional restriction that it will not preempt a conflicting thread that holds one or more mutexes (or semaphores). Otherwise, TxOS risks a deadlock with a transaction that 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 previous transactional operating systems.

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 3: 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.

5.2.3 Minimizing conflicts on lists

The kernel relies heavily on linked list data structures. When applied to lists, simple read/write conflict semantics produce a number of false positives. For instance, two transactions should both be allowed to add elements to the same list, even though adding an element is a list write. TxOS adopts techniques from previous transactional memory systems to define conflicts on lists more precisely [21].

TxOS isolates list updates with a lock and defines conflicts according to the states described in Table 3. For instance, a list in the `write` state allows concurrent transactional and non-transactional writers, so long as they do not access the same entry. Individual entries that are transactionally added or removed are annotated with a transaction pointer that is used to detect conflicts. If a writing transaction also attempts to read the list contents, it 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.

5.3 Managing transaction state

To manage transactional state, TxOS adds transaction objects to the kernel, which store metadata and statistics for a transaction. The kernel thread's control block (the `task_struct` in Linux) points to the transaction object, shown in Figure 3. A thread can have at most one active transaction, though transactions can flat nest, meaning that all nested transactions are subsumed into the enclosing transaction. Each thread in a multithreaded application can have its own transaction, and multiple threads (even those in different processes) may share a transaction, as we discuss in Section 6.2.

Figure 3 summarizes the fields of the transaction object. The transaction includes a status word (`status`). If another thread wins a conflict with this thread, it will update this word atomically with a compare-and-swap instruction. The kernel checks the status

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

Figure 3: **Data contained in a system transaction object, which is pointed to by the user area** (`task_struct`)**.**

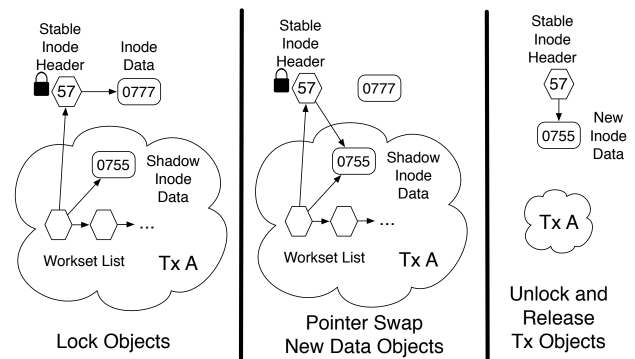

Figure 4: **The major steps involved in committing Transaction A with inode 57 in its workset, changing the mode from 0777 to 0755. The commit code first locks the inode. It then replaces the inode header's data pointer to the shadow inode. Finally, Transaction A frees the resources used for transactional bookkeeping and unlocks the inode.**

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 thread, 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 system aborts the transaction 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 file system monitoring events (i.e., `inotify` and `dnotify`). 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 operations are stored in the `undo_ops` field. The `tx_start_time` field is used by the contention manager (see Section 5.2.1), while the `retry_count` field stores the number of times the transaction aborted.

The `workset_list` is a skip list [39] that stores references to all of the objects for which the transaction has private copies. The workset list is sorted by the kernel locking discipline for fast commit. 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 unchanged for the duration of the transaction.

5.4 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 4. In the first step, the transaction acquires locks for all items in its workset. The workset is kept sorted according to the kernel locking discipline to enable fast commit and eliminate the possibility of deadlock among committing transactions. Specifically, objects are sorted by the kernel virtual address of the header, followed by lists sorted by kernel virtual address. Lists are locked last to maintain an ordering with the directory traversal code.

TxOS iterates over the objects twice, once to acquire the blocking locks and a second time to acquire non-blocking locks. TxOS is careful to acquire blocking locks before spinlocks, and to release spinlocks before blocking locks. Acquiring or releasing a mutex or semaphore can cause a process to sleep, and sleeping with a held spinlock can deadlock the system.

After acquiring all locks, the transaction does a final check of its status word with an atomic compare-and-swap instruction. If it has not been set to ABORTED, then the transaction can successfully commit (this CAS instruction is the transaction's linearization point [23]). 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's bookkeeping data are removed from the objects, then the locks are released. Between releasing spinlocks and mutexes, the transaction performs deferred operations (like memory allocations/frees and delivering file system monitoring events) and performs any pending writes to stable storage.

During commit, TxOS holds locks that are not otherwise held at the same time in the kernel. As a result, TxOS 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`, which protects updates to the hash table of directory entries cached in memory. By eliminating these coarse-grained locks, TxOS improves performance scalability for individual system calls.

5.5 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 kernel 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. 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.

5.6 User-level transactions

In order for a user-level transactional memory system to use system transactions, the TM system must coordinate commit of application state with commit of the system transaction. This section provides commit protocols for the major classes of TM implementations.

5.6.1 Lock-based STM requirements

TxOS uses a simplified variant of the two-phase commit protocol (2PC) [17] to coordinate commit of a lock-based user-level software (STM) transaction 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.

Depending on the implementation details of the user TM implementation, additional integration effort may be required of the STM implementation. For instance, a lazy versioned STM needs to ensure that a transactional `write` system call is issued with the correct version of the buffer. As an optimization, the STM runtime can check the return code on system calls within a transaction to detect an aborted system transaction sooner. For the TM systems we examined, coordinating commit and adding extra return checks were sufficient.

5.6.2 HTM and obstruction-free STM requirements

Hardware transactional memory (HTM) and obstruction-free software TM systems [22] 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 commit or abort together depending upon the result of this specific commit instruction.

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 [32,43,58] supports mechanisms that suspend user-initiated transactions, avoiding the mixture of user and kernel addresses in the same hardware transaction. Mixing user and kernel address creates 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.

6. TxOS KERNEL SUBSYSTEMS

This section discusses how various kernel subsystems support ACI[D] semantics in TxOS. In several cases, transactional semantics need not be developed from scratch, but are implemented by extending functionality already present in the subsystem. For example, we use the journal in `ext3` to provide true, multi-operation durability. We leverage Linux's support for deferring signal delivery to manage signals sent to and from transactional threads.

6.1 Transactional file system

TxOS simplifies the task of writing a transactional file system by detecting conflicts and managing versioned data in the virtual filesystem layer. The OS provides the transactional semantics—versioning updates and detecting conflicts. The file system need only provide the ability to atomically commit updates to stable storage (e.g., via a journal). By ensuring that all committed changes are written in a single journal transaction, we converted `ext3` into a transactional file system. Memory-only file systems, such as `proc` and `tmpfs`, are automatically transactional when used within system transactions.

6.2 Multi-process transactions

A dominant paradigm for UNIX application development is the composition of simple but powerful utility programs into more complex tasks. Following this pattern, applications may 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 threads to participate in the same transaction. The threads in a transaction may share an address space, as in a multithreaded application, or the threads may reside in different address spaces. Threads in the same transaction share and synchronize access to speculative state.

When a process forks a child inside a transaction, the child process executes within the active transaction until it performs a `sys_xend()` or it exits (where an exit is considered an implicit `sys_xend()`). The transaction commits when all tasks in the transaction have issued a `sys_xend()`. This method of process management allows transactional programs to call high-level convenience functions, like `system`, to easily create processes using the full complement of shell functionality. Such `exec`ed programs run with transactional semantics, though they 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.

System calls that modify process state, for example by allocating memory or installing signal handlers, are faster in transactionally forked tasks because they do not checkpoint the process's system state. An abort will simply terminate the process; no other rollback is required.

6.3 Signal delivery

Signal semantics in TxOS provide isolation among threads in different transactions, as well as isolation between non-transactional and transactional threads. Any signal sent to a thread not part of the source's transaction is deferred until commit by placing it in a deferral queue, regardless of whether the receiving thread is transactional. Signals in the queue are delivered in order if the transaction commits, and discarded if the transaction aborts.

When a thread begins a transaction, a flag to `sys_xbegin()` specifies whether incoming signals should be delivered speculatively within the transaction (*speculative delivery*) or deferred until

commit (*deferred delivery*). Speculative delivery enables transactional applications to be more responsive to input. When signals are delivered speculatively, they must be logged. If the transaction aborts, these signals are re-delivered to the receiving thread so that from the sender's perspective the signals do not disappear. When a transaction that has speculatively received a signal commits, the logged signals are discarded.

When signal delivery is deferred, incoming signals are placed in a queue and delivered in order when the transaction commits or aborts. Deferring signals allows transactions to ensure that they are atomic with respect to signal handlers [57]. Enclosing signal handling code in a transaction ensures that system calls in the handler are atomic, and forces calls to the same handler to serialize. Transactional handling of signals eliminates race conditions without the need for the additional API complexity of `sigaction`. While the `sigaction` API addresses signal handler atomicity within a single thread by making handlers non-reentrant, the API does not make signal handlers atomic with respect to other threads.

An application cannot block or ignore the `SIGSTOP` and `SIGKILL` signals outside of a transaction. TxOS preserves the special status of these signals, delivering them directly to transactional threads, even if the transaction started in deferred delivery mode.

Speculative and deferred delivery apply only to delivery of incoming signals sent from non-transactional threads or from a different transaction once it commits. When a transaction sends a signal to a thread outside of the transaction, it is buffered until commit. Threads in the same transaction can send and receive signals freely with other threads in the same transaction.

6.4 Future work

TxOS does not yet provide transactional semantics for several classes of OS resources. Currently, TxOS either logs a warning or aborts a transaction that attempts to access an unsupported resource: the programmer specifies the behavior via a flag to `sys_xbegin()`. This subsection considers some challenges inherent in supporting these resources, which we leave for future work.

Networking.

The network is among the most important resources to transactionalize. Within a system transaction, some network communication could be buffered and delayed until commit, while others could be sent and logically rolled back by the communication protocol if the transaction aborts. Network protocols are often written to explicitly tolerate the kinds of disruptions (e.g., repeated requests, dropped replies) that would be caused by restarting transactions. The open challenge is finding a combination of techniques that is high performance across a wide range of networking applications, while retaining a reasonably simple transaction API.

Interprocess communication.

While TxOS currently supports IPC between kernel threads in the same system transaction, and supports transactional signals and pipes, a range of IPC abstractions remain that TxOS could support. These abstractions include System V shared memory, message queues, and local sockets. IPC has much in common with networking, but presents some additional opportunities because the relevant tasks are on the same system. IPC on the same system admits more creative approaches, such as aborting a transaction that receives a message from a transaction that later aborts.

User interfaces.

Exchanging messages with a user while inside a transaction is unlikely to become a popular paradigm (although TABS implemented a transaction GUI by crossing out text dialogs from aborted transactions [48]), because the I/O-centric nature of user interfaces is not a natural fit with the transactional programming model. Like other communication channels, however, the OS could naturally support transactions that only read from or write to a user I/O device by buffering the relevant data. Maintaining a responsive user interface will likely mandate that developers keep transactions involving interfaces short.

Logging.

Applications may wish to explicitly exempt certain output from isolation while inside a transaction, primarily for logging. Logging is useful for debugging aborted transactions, and it is also important for security sensitive applications. For instance, an authentication utility may wish to log failed attempts to minimize exposure to password guessing attacks. An attacker should not be able to subvert this policy by wrapping the utility in a transaction that aborts until the password is guessed.

Most system resources can be reasonably integrated with system transactions. However, extending transactions to these resources may complicate the programming interface and slow the implementation. Future work will determine if system transactions for these resources are worth the costs.

7. EVALUATION

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

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

The hardware transactional memory experiments use MetaTM [41] on Simics version 3.0.27 [27]. The simulated machine has 16 1000 MHz CPUs, each with a 32 KB level 1 and 4 MB level 2 cache. An L1 miss costs 24 cycles and an L2 miss costs 350 cycles. The HTM uses the timestamp contention management policy and linear backoff on restart.

7.1 Single-thread system call overheads

A key goal of TxOS is to make transaction support efficient, taking special care to minimize the overhead non-transactional applications incur. To evaluate performance overheads for substantial applications, we measured the average compilation time across three non-transactional builds of the Linux 2.6.22 kernel on unmodified Linux (3 minutes, 24 seconds), and on TxOS (3 minutes, 28 seconds). This slowdown of less than 2% indicates that for most applications, the non-transactional overheads will be negligible. At the scale of a single system call, however, the average overhead is currently 29%, and could be cut to 14% with improved compiler support.

Table 4 shows the performance of common file system system calls on TxOS. We ran each system call 1 million times, discarding the first and last 100,000 measurements and averaging the remaining times. The elapsed cycles were measured using the `rdtsc` instruction. The purpose of the table is to analyze transaction overheads in TxOS, but it is not a realistic use case, as most system calls are already atomic and isolated. Wrapping a single system call in a

Call	Linux	Base		Static		NoTx		Bgnd Tx		In Tx		Tx	
access	2.4	2.4	1.0×	2.6	1.1×	3.2	1.4×	3.2	1.4×	11.3	4.7×	18.6	7.8×
stat	2.6	2.6	1.0×	2.8	1.1×	3.4	1.3×	3.4	1.3×	11.5	4.1×	20.3	7.3×
open	2.9	3.1	1.1×	3.2	1.2×	3.9	1.4×	3.7	1.3×	16.5	5.2×	25.7	8.0×
unlink	6.1	7.2	1.2×	8.1	1.3×	9.4	1.5×	10.8	1.7×	18.1	3.0×	31.9	7.3×
link	7.7	9.1	1.2×	12.3	1.6×	11.0	1.4×	17.0	2.2×	57.1	7.4×	82.6	10.7×
mkdir	64.7	71.4	1.1×	73.6	1.1×	79.7	1.2×	84.1	1.3×	297.1	4.6×	315.3	4.9×
read	2.6	2.8	1.1×	2.8	1.1×	3.6	1.3×	3.6	1.3×	11.4	4.3×	18.3	7.0×
write	12.8	9.9	0.7×	10.0	0.8×	11.7	0.9×	13.8	1.1×	16.4	1.3×	39.0	3.0×
geomean			1.03×		1.14×		1.29×		1.42×		3.93×		6.61×

Table 4: Execution time in thousands of processor cycles of common system calls on TxOS and performance relative to Linux. Base is the basic overhead introduced by data structure and code modifications moving from Linux to TxOS, without the overhead of transactional lists. Static emulates compiling two versions of kernel functions, one for transactional code and one for non-transactional code, and includes transactional list overheads. These overheads are possible with compiler support. NoTX indicates the current 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. In Tx is the cost of a system call inside a transaction, excluding `sys_xbegin()` and `sys_xend()`, and Tx includes these system calls.

transaction is the worst case for TxOS performance because there is very little work across which to amortize the cost of creating shadow objects and commit.

The **Base** column shows the base overhead from adding transactions to Linux. These overheads have a geometric mean of 3%, and are all below 20%, including a performance improvement for `write`. Overheads are incurred mostly by increased locking in TxOS and the extra indirection necessitated by data structure reorganization (e.g., separation of header and data objects). These low overheads show that transactional support does not significantly slow down non-transactional activity.

TxOS replaces simple linked lists with a more complex transactional list (§5.2.3). The transactional list allows more concurrency, both by eliminating transactional conflicts and by introducing fine-grained locking on lists, at the expense of higher single-thread latency. The **Static** column adds the latencies due to transactional lists to the base overheads (roughly 10%, though more for `link`).

The **Static** column assumes that TxOS can compile two versions of all system calls: one used by transactional threads and the other used by non-transactional threads. Our TxOS prototype uses dynamic checks, which are frequent and expensive. With compiler support, these overheads are achievable.

The **NoTx** column presents measurements of the current TxOS prototype, with dynamic checks to determine if a thread is executing a transaction. The **Bgnd Tx** column are non-transactional system call overheads for TxOS while there is an active system transaction in a different thread. Non-transactional system calls need to perform extra work to detect conflicts with background transactions. The **In Tx** column shows the overhead of the system call in a system transaction. This overhead is high, but represents a rare use case. The **Tx** column includes the overheads of the `sys_xbegin()` and `sys_xend()` system calls.

7.2 Applications and micro-benchmarks

Table 5 shows the performance of TxOS on a range of applications and micro-benchmarks. Each measurement is the average of three runs. The slowdown relative to Linux is also listed. Postmark is a file system benchmark that simulates the behavior of an email, network news, and e-commerce client. We use version 1.51 with the same transaction boundaries as Amino [56]. The LFS small file benchmark operates on 10,000 1024 bytes files, and the large file benchmark reads and writes a 100MB file. The Reimplemented Andrew Benchmark (RAB) is a reimplementation of the Modified Andrew Benchmark, scaled for modern computers. Initially, RAB creates 500 files, each containing 1000 bytes of pseudo-random printable-ASCII content. Next, the benchmark measures execution time of four distinct phases: the `mkdir` phase creates 20,000 directories; the `cp` phase copies the 500 generated files into 500 of these directories, resulting in 250,000 copied files; the `du` phase calculates the disk usage of the files and directories with the `du` command; and the `grep`/`sum` phase searches the files for a short string that is not found and checksums their contents. The sizes of the `mkdir` and `cp` phases are chosen to take roughly similar amounts of time on our test machines. In the transactional version, each phase is wrapped in a transaction. Make wraps a software compilation in a transaction. Dpkg and Install are software installation benchmarks that wrap the entire installation in a transaction, as discussed in the following subsection.

Across most workloads, the overhead of system transactions is quite reasonable (1–2×), and often system transactions speed up the workload (e.g., postmark, LFS small file create, RAB `mkdir` and `cp` phases). Benchmarks that repeatedly write files in a transaction, such as the LFS large file benchmark sequential write or the LFS small file create phase, are more efficient than Linux. Transaction commit groups the writes and presents them to the I/O scheduler all at once, improving disk arm scheduling and, on `ext2` and `ext3`, increasing locality in the block allocations. Write-intensive workloads outperform non-transactional writers by as much as a factor of 29.7×.

TxOS requires extra memory to buffer updates. We surveyed several applications' memory overheads, and focus here on the LFS small and large benchmarks as two representative samples. Because the utilization patterns vary across different portions of physical memory, we consider low memory, which is used for kernel data structures, separately from high memory, which can be allocated to applications or to the page cache (which buffers file contents in memory). High memory overheads are proportional to the amount data written. For LFS large, which writes a large stream of data, TxOS uses 13% more high memory than Linux, whereas LFS small, which writes many small files, introduced less than 1% space consumption overhead. Looking at the page cache in isolation, TxOS allocates 1.2–1.9× as many pages as unmodified Linux. The pressure on the kernel's reserved portion of physical memory, or low memory, is 5% higher for transactions across all benchmarks. This overhead comes primarily from the kernel slab allocator, which allocates 2.4× as much memory. The slab allo-

Bench	Linux ext2	TxOS ACI		Linux ext3	TxOS ACID	
postmark	38.0	7.6	0.2×	180.9	154.6	0.9×
lfs small						
create	4.6	0.6	0.1×	10.1	1.4	0.1×
read	1.7	2.2	1.2×	1.7	2.1	1.3×
delete	0.2	0.4	2.0×	0.2	0.5	2.4×
lfs large						
write seq	1.4	0.3	0.2×	3.4	2.0	0.6×
read seq	1.3	1.4	1.1×	1.5	1.6	1.1×
write rnd	77.3	2.6	0.03×	84.3	4.2	0.005×
read rnd	75.8	71.8	0.9×	70.1	70.2	1.0×
RAB						
mkdir	8.7	2.3	0.3×	9.4	2.2	0.2×
cp	14.2	2.5	0.2×	13.8	2.6	0.2×
du	0.3	0.3	1.0×	0.4	0.3	0.8×
grep/sum	2.7	3.9	1.4×	4.2	3.8	0.9×
dpkg	.8	.9	1.1×	.8	.9	1.1×
make	3.2	3.3	1.0×	3.1	3.3	1.1×
install	1.9	2.7	1.4×	1.7	2.9	1.7×

Table 5: 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.

cator is used for general allocation (via `kmalloc`) and for common kernel objects, like inodes. TxOS's memory use indicates that buffering transactional updates in memory is practical, especially considering the trend in newer systems toward larger DRAM and 64-bit addresses.

7.3 Software installation

By wrapping system commands in a transaction, we extend `make`, `make install`, and `dpkg`, the Debian package manager, to provide ACID properties to software installation. We first test `make` with a build of the text editor nano, version 2.0.6. Nano consists of 82 source files totaling over 25,000 lines of code. Next, we test `make install` with an installation of the Subversion revision control system, version 1.4.4. Finally, we test `dpkg` by 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 exit, but TxOS defers the signal until commit. This script could be rewritten to match the TxOS signal API in a production system.

As Table 5 shows, the overhead for adding transactions is quite reasonable (1.1–1.7×), 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.

7.4 Transactional LDAP server

Many applications have fairly modest concurrency control requirements for their stable data storage, yet use heavyweight solutions, such as a database server. An example is Lightweight Directory Access Protocol (LDAP) servers, which are commonly used to authenticate users and maintain contact information for large organizations. System transactions provide a simple, lightweight storage solution for such applications.

To demonstrate that system transactions can provide lightweight concurrency control for server applications, we modified the `slapd` server in OpenLDAP 2.3.35's flat file storage module (called LDIF)

Back end	Search Single	Search Subtree	Add	Del
BDB	3229	2076	203	172
LDIF	3171	2107	1032 (5.1×)	2458 (14.3×)
LDIF-TxOS	3124	2042	413 (2.0×)	714 (4.2×)

Table 6: Throughput in queries per second of OpenLDAP's slapd server (higher is better) for a read-only and write-mostly workload. For the Add and Del workloads, the increase in throughput over BDB is listed in parentheses. The BDB storage module uses Berkeley DB, LDIF uses a flat file with no consistency for updates, and LDIF-TxOS augments the LDIF storage module use system transactions on a flat file. LDIF-TxOS provides the same crash consistency guarantees as BDB with more than double the write throughput.

to use system transactions. The OpenLDAP server supports a number of storage modules; the default is Berkeley DB (BDB). We used the `SLAMD` distributed load generation engine[1] to exercise the server, running in single-thread mode. Table 6 shows throughput for the unmodified Berkeley DB storage module, the LDIF storage module augmented with a simple cache, and LDIF using system transactions. The "Search Single" experiment exercises the server with single item read requests, whereas the "Search Subtree" column submits requests for all entries in a given directory subtree. The "Add" test measures throughput of adding entries, and "Del" measures the throughput of deletions.

The read performance (search single and search subtree) of each storage module is within 3%, as most reads are served from an in-memory cache. LDIF has 5–14× the throughput of BDB for requests that modify the LDAP database (add and delete). However, the LDIF module does not use file locking, synchronous writes or any other mechanism to ensure consistency. LDIF-TxOS provides ACID guarantees for updates. Compared to BDB, the read performance is similar, but workloads that update LDAP records using system transactions outperform BDB by 2–4×. LDIF-TxOS provides the same guarantees as the BDB storage module with respect to concurrency and recoverability after a crash.

7.5 Transactional ext3

In addition to measuring the overheads of durable transactions, we validate 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 mount the disk to replay any operations in the ext3 journal and run `fsck` on the disk to validate that it is in a consistent state. We then verify that all results from committed transactions are present on the disk, and that no partial results from uncommitted transactions are visible. To facilitate scripting, we perform these checks using Simics. Our system successfully passes over 1,000 trials, giving us a high degree of confidence that TxOS transactions correctly provide atomic, durable updates to stable storage.

7.6 Eliminating race attacks

System transactions provide a simple, deterministic method for eliminating races on system resources. To qualitatively validate this claim, we reproduce several race attacks from recent literature on Linux and validate that TxOS prevents the exploit.

We downloaded the symlink TOCTTOU attacker code used by Borisov et al. [6] to defeat Dean and Hu's probabilistic countermeasure [11]. This attack code creates memory pressure on the file

[1] http://www.slamd.com/

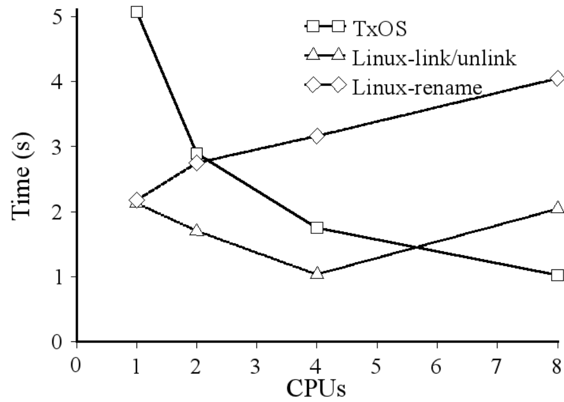

Figure 5: Time to perform 500,000 renames divided across a number of threads (lower is better). TxOS implements its renames as calls to `sys_xbegin()`, `link`, `unlink`, and `sys_xend()`, using 4 system calls for every Linux `rename` call. Despite higher single-threaded overhead, TxOS provides better 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×.

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.

On TxOS, the victim successfully resists the 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 [57]. The first attack is representative of a vulnerability present in `sendmail` up to 8.11.3 and 8.12.0.Beta7, in which an attacker induces a double-free in a signal handler. The second attack, representative of a vulnerability in the `screen` utility, 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 a `sys_xbegin, sys_xend` pair, which provides signal handler atomicity without requiring the programmer to change the code to use `sigaction`. In our experiments, TxOS serializes handler code with respect to other system operations, and therefore defeats both attacks.

7.7 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 strong semantics (a rename is atomic within a file system), 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. The problem is not that performance tuning `rename` is difficult, but it would substantially increase the implementation complexity of the entire file system, including unrelated system calls.

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

Table 7: Execution Time, number of system calls, and allocated pages for the genome benchmark on the MetaTM HTM simulator with 16 processors.

Transactions allow the programmer to combine simpler system calls to perform more complex operations, yielding better performance scalability and a simpler implementation. Figure 5 compares the unmodified Linux implementation of `rename` to calling `sys_xbegin()`, `link`, `unlink`, and `sys_xend()` in TxOS. In this micro-benchmark, we divide 500,000 cross-directory renames across a number of threads.

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× at 8 CPUs. The difference in scalability is directly due to TxOS using fine-grained locking to implement transactions, whereas Linux must use coarse-grained locks to maintain the fast path for `rename` and keep its implementation complexity reasonable. While this experiment is not representative of real workloads, it shows that solving consistency problems with modestly complex system calls like `rename` will either harm performance scalability or introduce substantial implementation complexity. Because of Linux's coarse-grained locks, TxOS' atomic `link`/`unlink` pair outperforms the Linux non-atomic `link`/`unlink` pair by a factor of 1.9× at 8 CPUs.

7.8 Integration with software TM

We qualitatively verify that system transactions can be integrated with existing transactional memory systems by extending a software and hardware TM implementation to use system transactions. We integrated DATM-J [42], a Java-based STM, with system transactions. The only modifications to the STM are to follow the commit protocol when committing a user level transaction that invokes a system call and to add return code checks for aborted system transactions, as outlined in Section 4.3.

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. The original code uses file locking. For one synthetic workload, the STM version is 47% faster at 7 threads.

7.9 Integration with hardware TM

In the genome benchmark from the STAMP transactional memory benchmark suite [31], the lack of integration between the hardware TM system and the operating system results in an unavoidable memory leak. Genome allocates memory during a transaction, and the allocation sometimes calls `mmap`. When the transaction restarts, it rolls back the allocator's bookkeeping for the `mmap`, but not the results of the `mmap` system call, thereby leaking memory. When the MetaTM HTM system [41] is integrated with TxOS, the `mmap` is made part of a system transaction and is properly rolled back when the user-level transaction aborts.

Table 7 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. Benchmark performance is not affected. 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 [58], with several proposed solutions, including open nesting [33] and a transactional pause instruction [58]. All previously proposed solutions complicate the programming model, the hardware, or both. System transactions address the memory leak with the simplest hardware requirements and user API.

8. 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 [55] and QuickSilver [20, 45] are operating systems that support 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 or read the directory. Unfortunately, creating a file in these 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. In contrast, TxOS implements system transactions with lazy version management, more sophisticated containers, and asymmetric conflict detection, allowing it to provide higher isolation levels while minimizing performance overhead.

Transactional Memory.

Transactional Memory (TM) systems provide a mechanism to provide atomic and isolated updates to application data structures. Transactional memory is implemented either as modifications to cache coherence hardware (HTM) [19,32], in software (STM) [12], or as a hybrid of the two [8, 10].

Volos et al. [54] 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 kernel threads in different processes, ensure durable updates to a file, or support multi-process transactions, all of which are needed to perform a transactional software installation and are supported by TxOS.

The system transactions supported by TxOS solve a fundamentally different problem from those solved by TxLinux [43]. TxLinux is a variant of Linux that uses hardware transactional memory 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 enforce consistency for kernel memory accesses within short critical regions. However, these techniques are insufficient to implement TxOS, which must guarantee consistency across heterogeneous system resources, and which must support system transactions spanning multiple system calls. TxLinux requires hardware transactional memory support, whereas TxOS runs on currently available commodity hardware.

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 [2]	Yes	Yes
Simple programmer interface	Yes	No	No	Yes
Other kernel resources in a transaction	No	Yes [3]	No	Yes

Table 8: A summary of features supported by recent transactional file systems.

Speculator.

Speculator [35] applies an isolation and rollback mechanism to the operating system that is very similar to transactions, allowing the system to speculate past 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 Speculator is designed for applications to share speculative results when they access the same data. 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 [36] and to debug system configuration [50], but does not provide ACID semantics for user-delimited speculation, and is thus insufficient for applications like atomic software installation/update.

Transactional file systems.

TxOS simplifies the task of writing a transactional file system by detecting conflicts and versioning data in the virtual file system layer. Some previous work such as OdeFS [15], Inversion [38], and DBFS [34] provide a file system interface to a database, implemented as a user-level NFS server. These systems do not provide atomic, isolated updates to local disk, and cannot address the problem of coordinating access to OS-managed resources. Berkeley DB and Stasis [46] are transactional libraries, not file systems. Amino [56] supports transactional file operation semantics by interposing on system calls using ptrace and relying on a user-level database to store and manage file system data and metadata. Other file systems implement all transactional semantics directly in the file system, as illustrated by Valor [49], Transactional NTFS (also known as TxF) [44], and others [14, 45, 47].

Table 8 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 if it participates in a distributed transaction.

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 currently does not. Valor primarily provides logging and coarse-grained locking for files. Because directory operations require locking the directory, Valor, like QuickSilver, is more conservative than necessary with respect to concurrent directory updates.

[2] Windows provides a kernel transaction manager, which coordinates commits across transactional resources, but each individual filesystem is still responsible for implementing checkpoint, rollback, conflict detection, etc.

[3] Windows supports a transactional registry.

In addition to TxF, Windows Vista introduced a transactional registry (TxR) and a kernel transaction manager (KTM) [44]. KTM allows applications to create kernel transaction objects that can coordinate transactional updates to TxF, TxR, and user-level applications, such as a database or software updater. KTM coordinates transaction commit, but each individual filesystem or other kernel component must implement its own checkpoint, rollback, conflict detection, etc. In contrast, TxOS minimizes the work required to transactionalize a file system by providing conflict detection and isolation in shared virtual filesystem code.

TxOS and KTM also represent different points in the design space of transactional application interfaces. KTM requires that all transactional accesses be explicit, whereas TxOS allows unmodified libraries or applications to be wrapped in a transaction. Requiring each system call to be explicitly transactional is a more conservative design because unsupported operations do not compile, whereas TxOS detects these dynamically. A key downside to KTM's low-level interface is that it requires individual application developers to be aware of accesses that can deadlock with completely unrelated applications on the same system (such as accessing two files in opposite order), and implement their own timeout and backoff system. In contrast, transactions in TxOS cannot deadlock and TxOS can arbitrate conflicts according to scheduler policies (Section 5.2.1) without any expert knowledge from the developer.

TxOS provides programmers with a simple, natural interface, augmenting the POSIX API with only three system calls (Table 2). Other transactional file systems require application programmers to understand implementation details, such as deadlock detection (TxF) and the logging and locking mechanism (Valor).

Distributed transactions.

A number of systems, including TABS [48], Argus [26], and Sinfonia [3], provide support for distributed transactional applications at the language or library level. User-level services cannot isolate system resources without OS support.

9. CONCLUSION

This paper argues for system transactions as a general-purpose, natural way for programmers to synchronize access to system resources, a problem currently solved in an *ad hoc* manner. TxOS supports transactions for system resources, including the file system, signals, memory allocation, and process management. Although system transactions in TxOS are limited in size and the scope of resources involved, they can solve a number of important, long-standing problems from a number of domains, including file system race conditions. This paper describes novel implementation techniques for system transactions that are efficient and minimize the effort required to extend transaction support to additional resources, such as converting a given file system implementation to a transactional file system.

10. ACKNOWLEDGMENTS

We extend thanks to Michael Bond, Michael Dahlin, Kathryn McKinley, Edmund Nightingale, Vitaly Shmatikov, Michael Walfish, Jean Yang, and the anonymous reviewers for careful reading of drafts, and to our shepherd Paul Barham for valuable feedback and suggestions. We also thank Indrajit Roy and Andrew Matsuoka for help developing TxOS. This research is supported by NSF Career Award 0644205 and the DARPA computer science study panel, phase 1. We thank Virtutech for their Simics academic site license program and Intel for an equipment donation.

11. 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] C. Blundell, E. C. Lewis, and M. M. K. Martin. Deconstructing transactions: The subtleties of atomicity. In *Fourth Annual Workshop on Duplicating, Deconstructing, and Debunking*. Jun 2005.

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

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

[8] 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.

[9] 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.

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

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

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

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

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

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

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

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

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

[19] 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.

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

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

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

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

[24] 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.

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

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

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

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

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

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

[31] C. C. Minh, J. Chung, C. Kozyrakis, and K. Olukotun. Stamp: Stanford transactional applications for multi-processing. In *IISWC*, 2008.

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

[33] M. J. Moravan, J. Bobba, K. E. Moore, L. Yen, M. D. Hill, B. Liblit, M. M. Swift, and D. A. Wood. Supporting nested transactional memory in LogTM. In *ASPLOS*, 2006.

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

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

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

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

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

[39] W. Pugh. Skip lists: a probabilistic alternative to balanced trees. *Communications of the ACM*, 33:668–676, 1990.

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

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

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

[43] 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.

[44] M. Russinovich and D. Solomon. *Windows Internals*. Microsoft Press, 2009.

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

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

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

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

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

[50] Y.-Y. Su, M. Attariyan, and J. Flinn. Autobash: improving configuration management with operating system causality analysis. In *SOSP*, 2007.

[51] 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.

[52] D. Tsafrir, T. Hertz, D. Wagner, and D. D. Silva. Portably solving file TOCTTOU races with hardness amplification. In *FAST*, 2008.

[53] E. Tsyrklevich and B. Yee. Dynamic detection and prevention of race conditions in file accesses. In *USENIX Security*, 2003.

[54] H. Volos, A. J. Tack, N. Goyal, M. M. Swift, and A. Welc. xCalls: Safe I/O in memory transactions. In *EuroSys*, 2009.

[55] 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.

[56] C. P. Wright, R. Spillane, G. Sivathanu, and E. Zadok. Extending ACID semantics to the file system. *Trans. Storage*, 3(2):4, 2007.

[57] M. Zalewski. Delivering signals for fun and profit. 2001.

[58] C. Zilles and L. Baugh. Extending hardware transactional memory to support non-busy waiting and non-transactional actions. In *TRANSACT*, Jun 2006.