Transactional system calls on Linux

Donald E. Porter
The University of Texas at Austin
porterde@cs.utexas.edu

Emmett Witchel
The University of Texas at Austin
witchel@cs.utexas.edu

Abstract

Have you ever had to manually back out an unsuccessful software install? Has a machine ever crashed on you while adding a user, leaving the group, password and shadow files inconsistent? Have you struggled to eliminated time-of-check-to-time-of-use (TOCTTOU) race conditions from an application? All of these problems have a single underlying cause: programmers cannot group multiple system calls into a single, consistent operation. If users (and kernel developers) had this power, there are a variety of innovative services they could build and problems they could eliminate. This paper describes system transactions and a variety of applications based on system transactions. We add system calls to begin, end, and abort a transaction. A system call that executes within a transaction is isolated from the rest of the system. The effects of a system transaction are undone if the transaction fails.

This paper describes a research project that developed transactional semantics for 152 Linux system calls and abstractions including signals, process creation, files, and pipes. The paper also describes the practical challenges and trade-offs in implementing transactions in Linux. The code changes needed to support transactions are substantial, but so are the benefits. With no modifications to dpkg itself, we were able to wrap an installation of OpenSSH in a system transaction. The operating system rolls back failed installations automatically, preventing applications from observing inconsistent files during the installation, and preserving unrelated, concurrent updates to the file system. Overheads for using transactions in an application like software installation range from 10-70%.

1 Introduction

A number of programming tasks are impossible to write robustly using the POSIX API. For example, backing out a failed software installation or upgrade is a major hassle for system administrators because the software spans multiple files with tightly coupled dependences. For instance, a new version of a binary may expect a new configuration file format or a new binary may not link with previous versions of the supporting libraries. If a software installation fails or the machine crashes during installation, these tight dependences are broken, often rendering the software unusable. If the failed upgrade is for a core system utility, such as the shell, the entire system may stop working. Software installation tools, such as `yum` and `apt`, have evolved to provide sophisticated support for tracking package dependences and automatically uninstalling libraries that are no longer needed, but even these systems fail—truly robust failure recovery remains elusive.

Even simple changes to system settings, such as adding user accounts, are prone to subtle errors or race conditions with other administrators. Local user accounts are stored across three files that need to be mutually consistent: `/etc/passwd`, `/etc/shadow`, and `/etc/group`. Utilities like `vipw` and `useradd` help ensure that these account files are formatted correctly and mutually consistent. These utilities create lock files to prevent concurrent modifications, but this cannot prevent a careless administrator from ignoring the lock file and editing the password files directly. Moreover, these utilities cannot ensure that updates to these files are mutually consistent if the system crashes during an operation. For instance, suppose the 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.

Race conditions for OS-managed resources, including the file system namespace, can cause security problems for programs that run as root. Despite their conceptual simplicity, time-of-check-to-time-of-use, or TOCT-

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`.

TOU, races [7] have created over 600 vulnerabilities in real, deployed applications [10]. A TOCTTOU race most commonly occurs as depicted in Figure 1, when a malicious program changes the file system namespace with a `symlink`, just between the check (`access`) and the use (`open`) in an application with root privilege. This is a common attack vector for privilege escalation, as these race conditions can trick a process with root-privilege to overwrite a sensitive file, such as the password database.

Each of these seemingly unrelated problems share an underlying cause: developers cannot group multiple system calls into a single, consistent operation. The ideal software installer would be able to atomically replace multiples files on the system at once; when the installation finished, either all of the updates take effect or they are all rolled back. Similarly, adding a user should atomically update each relevant configuration file, and prevent a concurrent user from interfering with the updates. Finally, an application with root privileges should be able to request that a permissions check and subsequent file open be executed in isolation from potentially interfering applications.

Some of these problems, such as TOCTTOU races, are being addressed in the kernel by adding more functionality to existing system calls. The `open` system call has acquired a number of flags that bundle in tasks like checking that the file doesn't exist and conditionally creating the file. The current `open` implementation is more complex than a simple `open`, `create`, and `stat` combined. Similarly, the `rename` system call has been heavily used by applications, such as editors, to atomically replace a single file. The `rename` implementation is so complex that Linux uses a single, file-system wide mutex to synchronize renames in all but the simplest cases, harming system scalability. In order to address TOCTTOU specifically, `openat` and over a dozen similar variants have been added to Linux. These calls essentially allow applications to reimplement their own private `dcache`, at a substantial performance and complexity cost to the application [16].

As an alternative, *system transactions* allow developers to compose a series of simple system calls into a more complex operation. The kernel guarantees that a system transaction appears to execute as one isolated, atomic operation. System transactions eliminate the need for complex work-arounds in applications, and even obviate the need for such semantically heavy system calls as `rename`. Windows Vista and later have already adopted a transactional file system and registry to address problems arising from crashes during software installation [14]. Rather than having to petition kernel developers for a point solution to the next race condition or crash-consistency issue, system transactions give developers the tools to solve their own problems.

This paper describes ongoing research at the University of Texas at Austin to develop transactional system calls on a variant of Linux, called TxOS. The work has appeared in previous research venues [11, 12]; this paper reviews the design of the system with a focus on the needed changes to the Linux source code and the rationale for the design decisions. Section 2 provides an overview of the TxOS design and Section 3 describes the implementation in more detail. Sections 4 and 5 measure the performance of system transactions. Section 6 describes why system transactions are a better solution than file locking or a transactional file system, Section 7 describes ongoing and future directions for the project, and Section 8 concludes.

2 TxOS Overview

System transactions provide ACID semantics for updates to OS resources, such as files, pipes, and signals.

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	63	4	link, access, stat, chroot, dup, open, close, write, lseek
Other	13	6	time, nanosleep, ioctl (partial), mmap2 (partial)
Totals	138	14	Grand total: 152

Unsupported		
Processes	33	nice, uselib, iopl, sched_yield, capget
Memory	15	brk, mprotect, mremap, madvise
Filesystem	29	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	151	

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.

In this programming model, both transactional and non-transactional system calls may access the same system state; the OS imposes a global order for all accesses and arbitrates contention 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()`.

TxOS implements system transactions by isolating data read and written in a transaction (making it invisible to unrelated kernel threads) 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 memory buffer first, allowing the OS to batch updates to the underlying device. 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, such as QuickSilver [15] and Locus [17] (Section 2.3). 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 the user makes an unsupported system call or a partially supported call cannot support transactional semantics, 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 152 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 remaining 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 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

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

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.

2.1 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 [9]), which can be enabled with a flag to `sys_xbegin()` (Table 2). In a recent paper [11], we describe how system transactions integrate with hardware and software transactional memory, providing a complete transactional programming model for multi-threaded applications.

2.2 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, which the system will buffer until commit. If the code waits for a reply to the buffered message, 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.

2.3 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** [5]. 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 follow 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 OS kernel. 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.

In contrast, transactions in TxOS operate on private copies of data structures, known as **lazy version management**. Transactions 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 wait 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 3, TxOS minimizes this overhead by splitting objects, turning a `memcpy` of the entire object into a pointer copy.

2.4 Interoperability and fairness

TxOS allows flexible interaction between transactional and non-transaction kernel threads. TxOS efficiently orders transactions with non-transactional accesses inside the kernel by requiring all system calls 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** [13]) 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.

3 Implementation

This section describes how system transactions are implemented in the TxOS kernel and the reasons why the TxOS implementation deviates from the Linux kernel. TxOS provides transactional semantics for 152 of 303 system calls in Linux, presented in Table 1. The supported system calls include process creation and termination, credential 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. Compared to the overall size of the kernel, these changes are small; however, some changes are invasive at points and this section explains why the changes were necessary and potential alternatives.

3.1 Object versioning

TxOS maintains multiple versions of kernel data structures so that system transactions can isolate the effects of system calls until transactions commit (i.e., hide the effects from other kernel threads), 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 [3] and recent concurrent programming systems [4].

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 approach might update the pointers to an object when that object is committed. This naïve approach is impractical, as some objects (e.g., inodes) are pointed to by a substantial number of other data structures which the object itself doesn't reference.

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

```
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.

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) [6] 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.

```
static inline struct inode_data *
tx_get_inode(struct inode *inode,
       enum access_mode mode){
   if(!aborted_tx())
      return error;
   else if(!live_transaction()){
      return inode->inode_data;
   else {
      contend_for_object(inode, mode);
      return get_private_copy(inode);
   }
}
```

```
struct inode *inode;
// Replace idata = inode->inode_data with
inode_data *idata = tx_get_inode(inode, RW);
```

Figure 3: Pseudo-code for the hook used to acquire an inode's data object, and an example of its use in code.

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.

3.2 Impact of data structure changes

The largest source of lines changed in TxOS comes from splitting objects such as inodes into multiple data structures. After a small amount of careful design work in the headers, most of the code changes needed to split objects was rather mechanical.

A good deal of design effort went into assessing which fields might be modified transactionally and must be placed in the data object, and which can remain in the header, including read-only data, kernel-private bookkeeping, or pointers to other data structures that are independently versioned. A second design challenge was assessing when a function should accept a header object as an argument and when it should accept a data object. The checks to acquire a data object are relatively expensive and would ideally occur only once per object per system call. Thus, once a system call path has acquired a data object, it would be best to pass the data object directly to all internal functions rather than reacquire it. This has to be balanced against forcing needless object acquisition in order to call a shared function that only uses the data object in the uncommon case.

Once the function signatures and data structure definitions are in place, the remaining work is largely mechanical. The primary change that must be propagated through the code is replacing certain pointer dereferences with hooks (Figure 3), so that TxOS can redirect requests for a data object to the transaction's private copy where appropriate. It is in this hook code where TxOS checks for conflicts between transactions. By encapsulating this work in a macro, we hide much of the complexity of managing private copies from the rest of the kernel code, reducing the chances for error.

A benefit of changing the object definitions is that it gives us confidence in the completeness of our hook placement. In order to dereference a field that can be modified in a transaction, the code must acquire a reference to a data object through the hook function. If the hook is not placed properly, the code will not compile. A question for future work is assessing to what degree these changes can be automatically applied during compilation using a tool like CIL [8]. This "header crawl" technique leads to more lines of code changed, but increases our confidence that the changes were made throughout the large codebase that is the Linux kernel.

3.3 Lists

Linked lists are a key data structure in the Linux kernel, and they present key implementation challenges. Simple read/write conflict semantics for lists throttle concurrent performance, especially when the lists contain directory entries. For instance, two transactions should both be allowed to add distinct directory entries to a single list, even though each addition is a list write. TxOS adopts techniques from previous transactional memory

systems to avoid conflicts on list updates that do not semantically conflict [3]. 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.

A second implementation challenge for linked lists is that an object may be speculatively moved from one list to another. This requires a record of membership in both the original list (marked as speculatively deleted) and the new list (marked as speculatively added). Ideally, one would simply embed a second `list_head` in each object for speculatively adding an entry to a new list; however, if multiple transactions are contending for a list entry, it is difficult to coordinate reclaiming the second embedded entry from an aborted transaction. For this reason, if a transaction needs to speculatively add an object to a list, it dynamically allocates a second `list_head`, along with some additional bookkeeping. Dynamic allocation of speculative list entries allows a transaction to defer clean-up of speculatively added entries from an aborted transaction until a more convenient time (i.e., one that does not further complicate the locking discipline for lists).

Although TxOS dynamically allocates `list_head` structures for transactions, the primary `list_head` for an object is still embedded in the object. During commit, a transaction replaces any dynamically allocated, speculative entries with the embedded list head. Thus, non-transactional code never allocates or frees memory for list traversal or manipulation.

A final issue with lists and transactional scalability is that most lists in the Linux kernel are protected by coarse locks, such as the `dcache_lock`. Ideally, two transactions that touch disjoint data should be able to commit concurrently, yet acquiring a coarse lock will cause needless performance loss. Thus, we implemented fine-grained locking on lists, at the granularity of a list. This improves scalability (§ 5), but complicates the locking discipline. Locks in TxOS are ordered by kernel virtual address, except that list locks must be acquired after other object locks. This discipline roughly matches the paradigm in the directory traversal code.

4 Evaluation

This section evaluates the overhead of system transactions in TxOS, as well as its behavior for several case studies, including a transactional software installation and a transactional LDAP server. We perform all of our experiments on a server with 1 or 2 quad-core Intel X5355 processors (for a 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.

4.1 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 does not reflect how a programmer would use system transactions because most system calls are already atomic and isolated. Wrapping a single system call in a 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 $\sim$3%, and are all below 20%, including 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.

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). Transaction support in the kernel does not significantly slow down non-transactional activity.

TxOS replaces simple linked lists with a more complex transactional list (§3.3). 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`).

A key overhead in the TxOS prototype is dynamic checks whether a system call is executing inside a transaction or not. An alternative implementation might provide two versions of each function, one transactional and one non-transactional, and convert the dynamic checks into compile-time checks. This optimization would require installing a second system call table for transactions and more sophisticated compilation support. We capture the benefits in the **Static** column, which reduces the average non-transactional system call overhead to 14% over Linux.

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.

4.2 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 [18]. 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

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	0.05×
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.

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 below (§ 4.3).

The overhead of system transactions for most workloads 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 sequential write phase 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 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 allocator 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.

4.3 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 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. A production system could rewrite the script to match the TxOS signal API.

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 automatically rolled 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.

4.4 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) 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.

4.5 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.

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

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.

4.6 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. [1] to defeat Dean and Hu's probabilistic countermeasure [2]. This attack code creates memory pressure on the file system cache to force the victim to deschedule for disk I/O, thereby lengthening the amount of time spent between checking the path name and using it. This additional time allows the attacker to win nearly every time on Linux.

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 conflict with the transactional `access` check, which TxOS resolves by putting

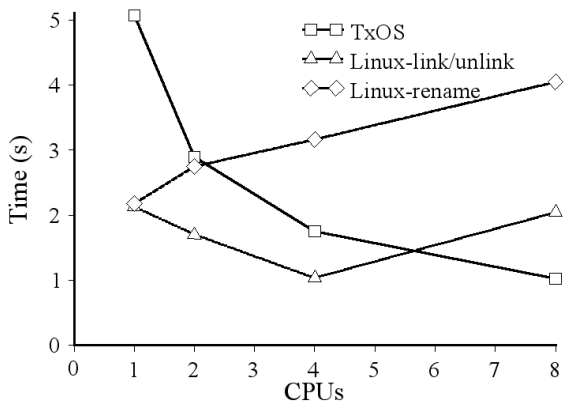

Figure 4: Time to perform 500,000 renames divided across a number of threads (lower is better). TxOS implements its renames as calls to `sys_xbegin()`, `link`, `unlink`, and `sys_xend()`, using 4 system calls for every Linux `rename` call. Despite higher single-threaded overhead, TxOS provides better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

the attacker to sleep until the victim commits. 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 [19]. 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, preventing both attacks.

5 Toward simpler, scalable system calls

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.

Transactions allow the programmer to combine simpler system calls to perform more complex operations, yielding better performance scalability and a simpler implementation. Figure 4 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. TxOS quickly recovers, performing within 6% at 2 CPUs and out-performing `rename` by 3.9× at 8 CPUs. The difference in scalability is directly due to implementing transactions with fine-grained locking, 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 1.9× at 8 CPUs.

A kernel that provides a smaller set of simple calls as well as a facility to compose them into more complex operations will be more maintainable than a kernel that supports a wide array of point solutions. Moreover, the complexity of managing fine-grained locking inside of transactions is encapsulated inside a small code base. In TxOS, the locking code inside a given system call is generally not complicated by transaction support. While adding transactions to the kernel may seem to increase the complexity of the system at first blush, TxOS demonstrates that the complexity can be tightly encapsulated and transactions can obviate the need for other complex or poor-performing code.

6 Design alternatives

The problems that system transactions solve are currently addressed to some degree by file locking and transactional file systems. However, neither approach is a complete solution, as explained in this section.

File locking File locking in Linux takes many forms: mandatory locking, advisory locking, and lock files. Lock files and advisory locking both provide concurrency control at the system level when all programs respect the locks; however, one cannot prevent buggy or malicious applications from ignoring these locks and accessing the data concurrently.

The reason advisory locking is popular is that mandatory locking can lead to denial of service on the system. The OS can revoke a mandatory lock, but likely at the cost of corrupting the underlying file.

File locking in Linux is associated with a file inode; this means that file locking cannot protect the file system namespace against TOCTTOU attacks. Finally, it is worth emphasizing that file locking only addresses concurrency control for system resources. File locking does not provide the ability to recover from a failed operation, which is useful for problems like software installation.

Transactional file systems The examples in the paper introduction focus on the file system, which is the source of the largest pain points. A transactional file system, such as TxF adopted by Windows Vista [14], can address some of the key issues. Unfortunately, implementing transactions within a particular file system (below the virtual filesystem (VFS) layer) undermines API simplicity and leads to usability problems.

When transactions are implemented in a specific file system, the key problem is file system state propagating into other volatile resources which cannot be rolled back by the file system if a transaction fails. State flowing from the file system to other resources leads to either conservative restrictions on transactions or speculative state leaking from an aborted transaction. For example, memory mappings are not under the control of the filesystem, and therefore most transactional file systems cannot allow a transactionally written file to also be memory mapped and executed. Linux software installers commonly unpack a set of files and then configure the software with post-installation scripts included in the software package. The conservative prohibition against executing transactionally written binaries, common in transactional file systems, prevents rolling back an install that can't be configured properly. As a second example, file handles are not visible to a file system and they do not roll back even if the backing store rolls back. For this reason, the Windows transactional file system requires that file handles used in a transaction be closed when a transaction ends and then subsequently re-opened when they are again needed. Finally, running processes can observe transactional file system data and propagate it (or results computed using it as inputs) through a pipe to new child process. Returning again to the post-installation script example, rolling back modifications to the file system does not kill the running post-installation script nor does it stop any daemons it may have launched or undo requests it sent to another running service. Compensating for these actions in userspace is difficult; the simplest programming model requires more robust kernel support.

Implementing transactions as a first-class kernel primitive simplifies the programming model for developers; treating transactions as a core kernel abstraction better encapsulates implementation details. For example, most transactional file systems expose locking details to users, often in the form of forcing them to reason about the risk of deadlock between completely unrelated programs. The TxOS prototype encapsulates all locking details within the kernel, guaranteeing the user that transactions cannot deadlock with each other. Transaction isolation in TxOS can still lead to starvation in some pathological cases, such as a long-running transaction denying access to a file or short transactions repeatedly aborting a longer transaction before it can commit. Rather than expose locking or other low-level system details to users as a way to enforce transaction scheduling policy, TxOS allows system administrators to set high-level policies through a file in `/proc`. For instance, the default policy is to favor the transaction with the highest scheduling priority, but this can be replaced with a policy that favors the oldest transaction. Selecting a high-level contention management policy is much less error-prone than managing kernel locks in userspace.

A final argument for generalized transaction support in the kernel is that transactions are a useful feature for all Linux file systems. The TxOS design implements file system transactions primarily in the VFS layer, leaving minimal adoption work for a specific file system. The

main onus on a specific file system is ensuring atomic commit of data to disk, which is already provided by common techniques such as journaling. As a proof point, we implemented a transactional `ext3` file system in one developer month on TxOS. Transaction support can be generalized in the VFS layer while imposing minimal development effort on individual file systems.

7 Limitations and 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()`. Among the unsupported resources are the network, certain classes of inter-process communication, and user interfaces.

Ongoing work on TxOS is focused in three directions. First, we plan to study additional applications that can benefit from transactions. Second, we plan to add support for additional kernel abstractions and resources. Third, we would like to find additional optimizations to improve the performance of the TxOS prototype.

8 Conclusion

TxOS demonstrates that transactions are a practical abstraction a widely-deployed, commodity OS. The code changes required are substantial, but so are the benefits. The source code for TxOS is available at `http://txos.code.csres.utexas.edu`.

References

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

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

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

[4] M. Kulkarni, K. Pingali, B. Walter, G. Ramanarayanan, K. Bala, and L. P. Chew. Optimistic parallelism requires abstractions. In *PLDI*, 2007.

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

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

[7] W. S. McPhee. Operating system integrity in OS/VS2. *IBM Systems Journal*, 13(3):230–252, 1974.

[8] G. C. Necula, S. Mcpeak, S. P. Rahul, and W. Weimer. CIL: Intermediate language and tools for analysis and transformation of C programs. In *ICCC*, pages 213–228, 2002.

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

[10] NIST. National Vulnerability Database. `http://nvd.nist.gov/`, 2010.

[11] D. E. Porter, O. S. Hofmann, C. J. Rossbach, A. Benn, and E. Witchel. Operating system transactions. In *SOSP*, 2009.

[12] D. E. Porter and E. Witchel. Operating systems should provide transactions. In *HotOS*, 2009.

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

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

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

[16] 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.

[17] 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.

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

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