Operating Systems Should Provide Transactions

Donald E. Porter, Indrajit Roy, Andrew Matsuoka, Emmett Witchel
Department of Computer Sciences, The University of Texas at Austin, Austin, TX 78712
{porterde,indrajit,matsuoka,witchel}@cs.utexas.edu
TR-08-30, June 17, 2008

Abstract

Operating systems can efficiently provide *system transactions* to user applications, in which user-level processes can execute a series of system calls atomically and in isolation from other processes on the system. The effects of system calls performed during a system transaction are not visible to the rest of the system (other threads or hardware devices) until the transaction commits. This paper describes TxOS, a variant of Linux 2.6.22, which is the first operating system to implement system transactions on commodity hardware with recent techniques from the transactional memory literature. The paper demonstrates that system transactions can solve problems in a wide range of domains, including security, isolating extensions, and user-level transactional memory. We also show that combining semantically lightweight system calls to perform heavyweight operations can yield better performance scalability: for example, enclosing `link` and `unlink` within a system transaction outperforms `rename` on Linux by 14% at 8 CPUs.

1 Introduction

The challenge of system API design is finding a small set of easily understood abstractions that compose naturally and intuitively to solve diverse programming and systems problems. Using the file system as the interface for everything from data storage to character devices and inter-process pipes is a classic triumph of the Unix API that has enabled large and robust applications. We show that system transactions are a similar, broadly applicable abstraction: transactions belong in the system-call API. Without system transactions, important functionality is impossible or difficult to express.

System transactions allow a user to transactionally group a sequence of system calls, for example guaranteeing that two writes to a file are either both seen by a reader or neither are seen. System transactions provide atomicity (they either execute completely or not at all) and isolation (in-progress results are not visible so transactions can be serially ordered). The user can start a system transaction with the `sys_xbegin()` system call, she can end a transaction with `sys_xend()` and abort it with `sys_xabort()`. The kernel makes sure that all system calls between an `sys_xbegin()` and an `sys_xend()` execute transactionally.

This paper introduces TxOS, a variant of Linux 2.6.22 which supports system transactions on commodity hardware. TxOS is the first operating system to support transactions that allow any sequence of system calls to execute atomically and in isolation. It is also the first to apply current software transactional memory (STM) techniques to system transaction implementation, which make transactions more efficient and which allow a flexible contention management policy among transactional and non-transactional operations. This flexibility lets the system balance scheduling and resource allocation between transactional and non-transactional operations.

We use TxOS to solve a variety of systems problems, indicating that system transactions deserve a place in the system API. We demonstrate that system transactions can eliminate time-of-check-to-time-of-use (TOCTTOU) race conditions, they can isolate an application from some misbehaviors in libraries or plugins, and they allow user-level transactions to modify system resources.

An important class of current security vulnerabilities consist of time-of-check-to-time-of-use (TOCTTOU) race conditions. During a TOCTTOU attack, the attacker changes the file system using symbolic links while a victim (such as a setuid program) checks a particular file's credentials and then uses it (e.g., writing the file). Between the credential check and the use, the attacker compromises security by redirecting the victim to another file— perhaps a sensitive system file like the password file. At the time of writing, a search of the U.S. national vulnerability database for the term "symlink attack" yields over 400 hits [30]. System transactions can eliminate TOCTTOU race conditions. If the user starts a system transaction before doing their system calls (e.g., an `access` and `open`), then the OS will guarantee that the interpretation of the path name used in both calls will not change during the transaction.

Having an API for transactions frees the system from supporting complex semantics that have accrued in their absence. For example, text editors [1] and source code control systems [3] use the `rename` system call heavily

because of its strong atomicity and isolation properties—renames either successfully complete or they leave no trace of partial execution. Allowing the user to combine semantically simple system calls, such as `link` and `unlink`, within a transaction more clearly expresses his intent, increases the performance scalability of the system, and reduces the implementation complexity for the operating system.

User-level transactions, such as those provided by a transactional memory system, run into trouble if they need to update system state. Such transactions cannot simply make a system call, because doing so violates their isolation guarantees. System transactions provide a mechanism for the transactional update of system state and in Section 3.4 we show how to coordinate user- and system- level transactions into a seamless whole with full transactional semantics.

In order to support system transactions, the kernel must be able to isolate and undo updates to shared resources. This adds latency to system calls, but we show that it can be acceptably low (13%–327%) within a transaction, and 10% outside of a transaction). However, using system transactions can provide better performance scalability than locks as we show with a web server in Section 5.4, which uses transactions to increase throughput 2× over a server that uses fine-grained locking.

This paper makes the following contributions:

- a new approach to implementating system transactions on commodity hardware, which provides strong atomicity and isolation guarantees with low performance overhead, implemented in Linux 2.6.
- shows that semantically lightweight system calls can be combined within a transaction to provide better performance scalability and ease of implementation than complex system calls. Placing `link` and `unlink` in a transaction outperforms `rename` on Linux by 14% at 8 CPUs.
- demonstration of the use of system transactions to avoid TOCTTOU races whose performance is superior to the current state-of-the-art user-space technique [45].
- showing how system transactions can be used to recover from some faults in software plugins and libraries with minimal performance overhead.
- showing how to maintain transactional semantics for user-level transactions that modify system state.

The paper is organized as follows: Section 2 provides background on each of the example problems that we address with transactions. Section 3 describes the design of operating system transactions within Linux and Section 4 provides implementation details. Section 5 evaluates the system in the context of the target applications. Section 6 provides related work and Section 7 concludes.

2 Overview and motivation

This section defines system level transactions and explains how they can be used by applications. It then describes three case studies with system transactions. The usage scenarios we choose demonstrate the applicability and advantages of system transactions across a wide range of systems problems.

2.1 System transactions

To describe system transactions, we must first make the distinction between *system state* and *application state*. Writes to the file system or forking a thread are actions that update system state, whereas updates to the data structures within an application's address space represent application state. System transactions are a novel and efficient way to provide the atomic and isolated access of transactions to system state.

A complete transactional programming model must provide an interface that is uniform with respect to both system and application state. Several techniques for supporting transactions that manage application state already exist, including transactional memory [19, 21] and recoverable virtual memory [37, 39]. We show in Section 3 that the operating system, by providing system transactions, can coordinate with multiple implementations of application transactions, giving applications the freedom to choose the best fit for their needs. For our target applications, we use both software transactional memory and an implementation of copy-on-write recovery to provide application transactions.

In adding the support for system transactions to Linux, we have focused on system calls related to the file system. Transactions are implemented at the virtual file system layer. This approach has the advantage of providing a transactional interface to non-transactional file systems. At transaction commit, all of the changes are exposed to the file system at once, maintaining the safety guarantees of the underlying file system.

System transactions, as described in this work, commit their results to memory—the effects of a successful system transaction are not necessarily written to stable storage synchronously and may not survive a reboot if they haven't been written to stable storage. Many applications can benefit from transactional semantics of isolation and atomicity without durability. For instance, a file writer might want to make several updates to the contents of a file without concurrent readers seeing any of the individual writes. However, the writing application might be satisfied with the durability semantics of the underlying file system and does not require that the file contents be synced to disk at the conclusion of its update. While durability would be useful for system transactions, it is not necessary and thus deferred to future work.

Victim **Attacker**

```
if(access('foo')){
                    symlink('secret','foo');
  fd=open('foo');
  read(fd,...);
  ...
```

Victim **Attacker**

```
                  symlink('secret','foo');
sys_xbegin();
if(access('foo'))
  fd=open('foo');
sys_xend();
                  symlink('secret','foo');
if(fd >= 0){
   read(fd,...);
   ...
```

Figure 1: An example of a TOCTTOU attack, and eliminating the race with system transactions. The attacker's symlink is serialized (ordered) either before or after the transaction.

Previous research on incorporating transactional semantics into the operating system has focused on committing file system state atomically with a database transaction or cleaning up temporary state used in a computation distributed over a local network [38, 43, 47]. Windows Vista features a transactional file system [31], where file system operations can be transactionally grouped. System transactions, by contrast, allow any group of system calls to be grouped in a transaction and are therefore a more general mechanism. For instance, TxOS implements file-system transactions in the Virtual File system layer, allowing a wide range of file systems to support transactions with minimal change.

2.2 Transactions for security

A classic example of a concurrency vulnerability is the **time-of-check-to-time-of-use (TOCTTOU)** attack. Its most (in)famous instance is the `access`/`open` exploit in the UNIX file system (illustrated in Figure 1) [7], but it also manifests itself in temporary file creation and other accesses to system resources. A TOCTTOU condition occurs whenever there is a change—which may be forced by a concurrent malicious process—to security-critical data between the time it is checked as valid for use and the time it is actually used. This vulnerability occurs even on single-processor machines running commodity operating systems due to the interleaved scheduling of victim and attacker processes, but it is easier to exploit on platforms with multiple processors or cores.

```
int res = sys_xbegin();
if(res == X_OK){
   ...
   invoke plugin
   ...
  sys_xend();
} else {
   plugin faulted, continue execution
       without plugin
}
```

Figure 2: Wrapping a plugin invocation with a transaction for fault tolerance.

The root cause of file system TOCTTOU races is the presence of file names in system calls that check resource permissions (or other attributes like `stat()`). An attacker can make the same path name refer to different underlying files in the file system (e.g., by using symbolic links), thereby causing the victim's check and use system calls to refer to different files. While there are many proposed solutions to this problem, Dean and Hu showed that there is no portable, deterministic solution to TOCTTOU vulnerabilities without changing the system call interface [12].

The current best approaches to eliminating TOCTTOU races either operate at user level and increase the probability of detecting the attack [45] or they force the programmer to perform all checks within a single system call (essentially an atomic action), or on an open file descriptor (whose interpretation will not change like a path name), or they remove the functionality altogether. We propose eliminating TOCTTOU races with system transactions, which provide a deterministic safety guarantee and restore a natural programming model (as seen in Figure 1).

2.3 Isolating applications from libraries and plugins

Applications commonly extend their functionality by allowing third parties to create small extensions consisting of code and data that are loaded into the application's address space. This approach is popular because it simplifies the interfaces between application and extension, is familiar to programmers, and provides good performance.

Unfortunately, loading extensions directly into the application's address space is not necessarily safe. A buggy extension can corrupt application memory, causing the entire program to crash, and possibly corrupting its data on the disk. Transactions are one way to isolate applications from some failures in extensions or libraries with minimal effort on the part of the application programmer. Transactions provide a straightforward mechanism

to checkpoint and roll back application and system state if a fault is detected.

Figure 2 illustrates how calls to a plugin can be wrapped inside of a transaction. For many plugins and libraries, the extension can execute normally, oblivious to the fact that it is being wrapped in a transaction. If an error is detected while the plugin is executing, the application aborts the transaction and the system roll backs to the state checkpointed before the transaction began. The `sys_xbegin()` system call will return an error code, signaling to the application that it should unload the extension.

Recovery from detected faults is well explored in the literature [32, 40, 44, 49]; transactions provide the failure atomicity that is a building block of these techniques. In Section 5.3, we use a web browser to demonstrate how non-critical functionality in a library can be isolated with low overhead and minimal implementation effort.

2.4 System calls within TM

Transactional memory is an alternative to lock based programming that provides a simpler programming model than locking while maintaining good performance scalability [19,21]. It is generally implemented either in hardware (building on cache coherence) [15, 26] or in software (as a library or extension to the JVM or other runtime system) [13, 25].

One of the most troublesome limitations of transactional memory systems is lack of support for system calls within transactions. For example, the code in the top half of Figure 3 will append `buf` to the end of `/foo/bar` an arbitrary number of times when executed on current TM systems, depending on how often the user-level transaction has to abort and retry. Because transactional semantics do not extend to the system call, there is no way to rollback previous appends when the transactions retries.

The frequency and necessity of system calls in user programs make it unlikely that excluding them from all critical sections will ever yield a reasonable programming model. A few TM systems allow the programmer to attempt to undo the results of system calls herself [27, 50]. In earlier work, we argue that the side effects of some system calls are far-flung and very difficult to undo at the user level [20].

System transactions provide a mechanism that transactional memory systems can use to safely allow system calls within a transaction. When a TM application performs an operation that makes a system call, the runtime will begin a system transaction. Figure 3 provides pseudo-code a software transactional memory system might generate to incorporate system transactions into user-level memory transactions and coordinate commit. In this example, the transactional memory system handles buffering and possibly rolling back the user's memory state, and the operating system buffers updates to the file system. The updates to the file system are committed or aborted by the kernel atomically with the commit or abort of the user-level transaction. The programmer is freed from the complexity of implementation and need only reason about transactions.

```
x = 0;
atomic {
   x = 5;
   fd = open("/foo/bar", O_APPEND);
   write(fd, buf, n);
   ...
}
```

```
x = 0;
while(!committed) {
   beginSTM_Tx();
   x = 5;
   sys_xbegin();
   fd = open("/foo/bar", O_APPEND);
   write(fd, buf, n);
   ...
  if prepareSTM_Tx fails {
       abort_STM_Tx;
       sys_xabort();
       continue;
   } else {
       committed = sys_xend();
       if committed
          commit_STM_Tx;
       else
          abort_STM_Tx;
   }
}
```

Figure 3: An example to show how user and system transactions are coordinated. The example code contains system calls within a transactional memory critical region that an application developer might write, followed by pseudocode for how an STM system could expand the code to coordinate the user and system transactions. `sys_xbegin()` and `sys_xend()` are system calls to begin and end system transactions.

System transactions thus expand the transactional programming model, enabling a larger class of applications to use transactional memory. In Section 5, we evaluate the use of system calls within a transactional web server running on a Java STM system [35].

3 TxOS Design

This section outlines the design of system transactions in TxOS, which is inspired by recent advances in software

Function Name	Description
int sys_xbegin (int recover_user)	Begin a transaction. If recover_user is true, OS saves user memory state and automatically rolls back and restarts on conflict.
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 recover_user, setting no_restart rolls the transaction back but does not restart it.

Table 1: TxOS API

transactional memory systems [5, 17, 21, 24]. In particular, we describe the user-level API for system transactions, how isolation is preserved on kernel data structures, the mechanisms for conflict detection and resolution, and the coordination of user-level transactions with system-level transactions.

3.1 Overview

A key design goal of TxOS is to expose system transactions to the user without major modifications to existing code, allowing easy adoption by a variety of applications. TxOS achieves this by adding three simple but powerful system calls that manage transactional state, shown in Table 1. `sys_xbegin()` starts a system transaction and `sys_xend()` commits the transaction. `sys_xabort()` ends the transaction without committing the updates.

All system calls made within a transaction retain their current interface. The only change required to use transactions is enclosing the relevant code region with calls to `sys_xbegin()`, `sys_xabort()`, and `sys_xend()`. Placing system calls within a transaction changes the semantics of when and how their results are published to the rest of the system. Outside of a transaction, actions on system resources are visible as soon as the relevant internal kernel locks are released. Within a transaction, all updates are kept isolated until commit time, when they are atomically published to the rest of the system.

Example. As an example, consider the TOCTTOU race between the `access()` and the `open()` from Figure 1. System transactions eliminate this race if the programmer encloses these system calls in a transaction. By starting a transaction, the user requires the kernel to force conflicting operations to occur either before or after the work within the transaction. The attacker's attempt to inject a symlink modifies a directory entry and inode that the transaction is reading, and is a conflicting operation. The symlink, therefore, must execute either before or after the (former) victim's code.

3.2 Maintaining isolation

System transactions isolate the effects of system calls until the transaction commits, and they undo the effects of a transaction if it cannot complete. Specifically, system transactions isolate the effects of system calls by isolating the effected kernel data structures directly. This isolation is performed by adding a level of indirection in the form of **shadow** or private copies of kernel data structures that are read or written within a transaction. The technique of using shadow objects (called **lazy version management**) is different from the traditional database approach of updating in place and using undo logs to provide transactional semantics (called **eager version management**). For example, the first time a kernel object is encountered within a transaction, a shadow copy is created for the transaction. For the rest of the transaction, this shadow object is used in place of the **stable** object. This ensures that the transaction has a consistent view of system state. When the transaction commits, updates to shadow objects are copied to their stable counterpart.

In the rare case of a conflict, transactions must abort and retry because they cannot be serialized with updates made by other processes. If a transaction aborts, the shadow copies of the kernel objects are simply discarded. This is enough to ensure that the system is in a consistent state. For the user program, TxOS provides the option to roll back the user's memory state, using the page tables to save and restore the user's virtual address space. However, user programs can be more efficient managers of their state. For simple cases like Figure 1, the programmer can make a critical region whose memory state does not need to be restored on a rollback. For multi-threaded programs, a transactional memory system is a more natural fit for managing application state because it works at a finer granularity (objects or memory stripes) than the OS page tables.

Lazy version management is appropriate for the Linux kernel because eager version management has two major problems. First, the kernel must support real-time processes and interrupt handlers that should not be forced to wait on a failed transaction to undo its eager changes to system state (recall that eager version management writes the old data in an undo log). Second, maintaining isolation on eager updates requires holding locks for the duration of the transaction; system locks would need to be held after a return from a system call but before the transaction ends. Holding system locks while a thread executes at user level can deadlock the kernel.

A key limitation of this approach is that all buffered updates to kernel data structures must fit in memory. In our prototype implementation, transactions that perform very large file writes, for instance, will deterministically fail if they overflow the available buffer space. We believe that this can be ameliorated in most cases by

spilling the data to swap space or unallocated blocks of the file system.

TxOS currently ensures isolation only within a single system. Correctly isolating updates to a shared, distributed filesystem (e.g., NFS) would require extensions to the protocol, which we defer to future work. Currently, TxOS can buffer updates and send them to the server at commit time, but cannot guarantee that they will be committed atomically or arbitrate conflicts with other servers.

3.3 Conflict detection and resolution

Transactions can conflict with other transactions or non-transactional operations. For example, a process executing a transaction, which we call a transactional process or transactional (kernel) thread, could be reading a file that another, non-transactional process is trying to write. If the write succeeds before the reader commits, the reader no longer has a consistent view of system state and hence cannot safely commit. Such conflicting situations must be detected and resolved by the system. As we discuss in the implementation section, TxOS detects conflicts by having transactions register objects that they access in a global hash table that is checked by transactional and non-transactional threads.

Unlike most software transactional memory systems, TxOS guarantees **strong isolation**[1] which means that not only are transactions serialized with each other, they are also serialized with respect to non-transactional operations. Strong isolation adds overhead to non-transactional execution paths but is easier to reason about if data is ever touched by both transactional and non-transactional operations [25, 41]. For instance, strong isolation prevents the roll back of a transaction from overwriting a non-transactional update. The complexity of execution paths within Linux makes strong isolation necessary for a kernel developer to have any hope of reasoning about the system.

3.3.1 Contention Management

Once a conflict is detected between two transactions, TxOS invokes the Contention Management module to resolve the conflict. This module implements a policy to arbitrate conflicts among transactions, dictating which of the conflicting transactions may proceed to commit. All other conflicting transactions must abort.

As a default policy, TxOS adopts the *osprio* policy used in TxLinux [36], though there it was used for hardware transactional memory rather than system transactions. *Osprio* always selects the higher priority process as the winner of a conflict, eliminating priority and policy inversion in transactional conflicts. When processes with the same priority conflict, the older transaction wins [33], guaranteeing liveness within a given priority level.

[1] Also called strong atomicity.

3.3.2 Asymmetric conflicts

A conflict between a transactional and non-transactional thread is called an **asymmetric conflict** [34]. Unlike transactional threads, non-transactional threads cannot be rolled back, so the system has fewer options when dealing with these conflicts. However, it is important for TxOS to have the freedom to resolve an asymmetric conflict in favor of either the transactional or non-transactional thread (most of the time); otherwise asymmetric conflicts will undermine fairness in the system, perhaps starving transactions.

While non-transactional threads cannot be rolled back, they can (often) be preempted, which allows them to lose conflicts with transactional threads. Kernel preemption is a recent feature of Linux that allows processes to be preemptively descheduled while executing system calls inside the kernel, unless they are inside of certain critical regions. In TxOS, non-transactional threads detect conflicts with transactional threads before they actually update state, usually when they grab 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. The kernel places the non-transactional process on a wait queue the next time it makes a system call and only wakes it up after the transaction commits.

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

3.4 Coordinating User and System Transactions

When a user-level TM uses system transactions, some care is required to ensure that the two transactions commit atomically: either they both commit at a single serialization point or they both roll back.

3.4.1 Lock-based STM requirements

For a lock-based STM to coordinate commit with TxOS, we use a simplified variant of the two-phase commit protocol (2PC) [14]. The TxOS commit consists of the following steps (also depicted in Figure 3).

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, and
4. 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). 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 commit, prepare is accomplished by simply holding all of the locks required for 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 is serialized after the user commit.

Alternatively, the kernel could prepare first. TxOS does not do this because it incurs the overhead of additional kernel crossings, and would require the kernel to exclude all other processes from prepared resources until the user releases them. Such exclusion is untenable from a security perspective, as it could lead to buggy or malicious users monopolizing system resources.

3.4.2 HTM and obstruction-free STM requirements

Hardware transactional memory (HTM) and obstruction-free STM systems [18] use a single instruction (`xend` and compare-and-swap, respectively), to perform their commits. For these systems, a prepare stage is unnecessary. A more appropriate commit protocol is for the kernel to issue the commit instruction on behalf of the user once the kernel has validated its workset. Both the system and user level transaction now commit or abort depending upon the result of this specific commit instruction.

For hardware transactional memory support, TxOS also requires that the hardware allow the kernel to suspend user-initialized transactions. Every HTM proposal that supports an OS [26, 36, 50] contains the ability to suspend user-initiated transactions so that user and kernel addresses do not enter the same hardware transaction (doing so would create a security vulnerability in most HTM proposals).

TxOS runs on commodity hardware and does not require any special HTM support. It is plausible that performance could be improved using hardware assistance similar to recent proposals [11, 42]. We leave this question for future work.

Data Structure	Description
Transaction	Pointed to by user area (`task_struct`), it stores transactional metadata and statistics.
ConflictTable	Stores entries for all kernel objects that are in any transaction's working set. Used for conflict detection.

Table 2: Main data structures used by TxOS to manage kernel transactions.

4 Implementation

System transactions in Linux add roughly 2,600 lines of code for transaction management, 4,000 lines for object management, and 500 lines for checkpointing user state. TxOS also requires about 2,500 lines of changes to redirect pointers to shadow objects when executing within a transaction and to insert checks for asymmetric conflicts when executing non-transactionally. The changes were largely in the virtual file system, memory management, and scheduling code.

TxOS modified the following Linux data structures to be able to participate in transactions (described in more detail by Bovet and Cesati [9]): `inode`, `dentry`, `super_block`, `file`, `vfsmount`, `list_head`, and `hlist_head`. These are file system data structures, the kernel linked list and hash table implementations. These data structures are modified during essential file system operations such as `open()`, `read()`, `write()`, `link()`, `unlink()`, and `close()`.

4.1 Managing transaction state

In order to manage transactional state, we added two major data structures, the transaction object and the ConflictTable (see Table 2). The transaction object (shown in Figure 4) is pointed to by the kernel thread's control block (the `task_struct` in Linux). A process can have at most one active transaction, though transactions can flat nest (all nested transactions are rolled into the enclosing transaction).

The fields of the transaction object are summarized in Figure 4. The transaction includes a status word (`tx_status`) that can be atomically updated by another thread that wins a conflict with it. The status word is checked by the transaction when attempting to add a new shadow object to its workset and checked before commit. The workset hashtable tracks the transaction's shadow objects.

The transaction stores a start timestamp (`tx_start_time`) that is used to arbitrate conflicts in favor of the older transaction. Non-transactional system calls also acquire a timestamp for fair contention with transactions. The `retry_count` field

```
struct transaction {
   // live, aborted, inactive
  atomic_t tx_status;
   // timestamp for contention management
  uint64 tx_start_time;
  uint32 retry_count;
   //register state at beginning of tx
  struct pt_regs *checkpointed_registers;
   // If recover_user is true, make addr.
   // space copy-on-write and save
   // writeable page table entries.
  pte_list *checkpointed_ptes;
   // Used for conflict detection
  workset_hlist *workset_hashtable;
   // operations deferred until commit
  deferred_ops;
   // ops that must be undone at abort
  undo_ops;
}
```

Figure 4: Data contained in a system transaction object, which is pointed to by the user area (`task_struct`).

stores the number of times the transaction has aborted. The `checkpointed_registers` field stores the register state on the stack at the beginning of the `sys_xbegin()` (if `recover_user` is selected), or the beginning of the current system call otherwise.

The `checkpointed_ptes` field stores a checkpoint of the user's memory state, if requested. If the `recover_user` option is selected, the user's address space is set to copy-on-write and all page faults place a copy of the page in an undo log. These fields allow the kernel to restore user state after a transaction abort. In order to minimize page faults, we pre-copy the user's stack when the transaction starts, and restore the write bits where appropriate upon commit.

There are certain operations that a transaction must defer until it commits, such as freeing memory and delivering `dnotify` events. The `deferred_ops` field stores these events in a representation optimized for the small number of these events that are common in our workloads. Similarly, there are operations that must be undone if a transaction is aborted, such as releasing locks it holds and freeing memory it allocates. These are stored in the `undo_ops` field.

The workset of a transaction is a private hashtable that stores references to all of the objects for which the transaction has private copies. Each entry in the workset contains a pointer to the stable object, the shadow copy, whether the object is read-only or read-write, and a set of type-specific methods (commit, abort, lock, unlock, release). When a transaction adds an object to its workset, it increments the reference count on the stable copy. This prevents the object from being unexpectedly freed while the transaction still has an active reference to it.

4.2 Conflict detection

The second major data structure in TxOS is the ConflictTable, a global hash table of object references, which is used to detect conflicts. Each entry in the table stores a reference to a stable object as well as a list of non-aborted transactions that are using the object.

TxOS adds code to every system call processing path to check the ConflictTable. Both transactional and non-transaction threads must check the table to detect conflicts and asymmetric conflicts, respectively. Sometimes TxOS adds checks of the ConflictTable to calls that already exist (like those that lock objects for update), and some calls were added manually (like those in the list manipulation routines). Buckets in the ConflictTable must be locked by both conflict detection and transaction commit to maintain data structure consistency, so the ConflictTable must have enough hash buckets and a good hash function to avoid becoming a central bottleneck. For our workloads, we found 512 buckets to be a reasonable balance between space efficiency and concurrency.

When a process wishes to access a kernel object for the first time, it hashes the pointer, locks the appropriate ConflictTable bucket and checks for a corresponding entry. If an entry is found with a conflicting mode (i.e. it is being written by the transaction or the current thread wishes to write it), there is a conflict and the thread calls the contention manager to arbitrate the conflict. The contention manager updates the `tx_status` field of the losing transaction to ABORTED.

4.3 Leveraging semantics

Most transaction implementations (including most transactional memory proposals) serialize transactions on the basis of simple read/write conflicts. A datum cannot be accessed by multiple transactions if at least one of the accesses is a write. However, many kernel data structures have lax semantics where multiple writes are not conflicts.

TxOS contains special cases where writes are not considered conflicts or they are deferred to increase concurrency on kernel objects. Modifying a reference count does not cause a transaction to be considered a writer of an object. Similarly, there are fields in common file system objects used by threads in the kernel memory management system (`kswapd`) to reclaim memory used to cache file system data. We allow `kswapd` to freely modify these fields when they do not interfere with a transaction (e.g., moving a dentry to the back of an LRU list). Finally, the access time on an inode is updated at commit time instead of when the access occurs.

4.4 Lock ordering

Transactional systems that update in place (eager version management), like most databases, implement transactional isolation by simply retaining all acquired locks until the end of the transaction. Deadlock avoidance is difficult for these systems because independent operations are composed to form a transaction. Each operation needs its own set of locks. When the first operation acquires a lock, it does not know if a later operation will need a lock that precedes it in the global lock order. Database implementations typically do not order all locks; instead they handle deadlocks by timing out transactions, which are then aborted, randomly backed-off, and retried.

TxOS uses lazy versioning, so it releases locks on objects as soon as it makes a private copy. Because the system retains a consistent, private copy of the object, it elides all subsequent lock acquires on the object, offsetting some synchronization costs. However, all objects must be locked during commit. Because TxOS knows the objects present in the committing transaction's working set, it can lock them in an order consistent with the kernel's locking discipline (i.e., by kernel virtual address).

4.5 Commit protocol

When a system transaction calls `sys_xend()`, it is ready to begin the commit protocol. The first step is to acquire all of the locks protecting objects in its workset and the conflict detection table, performed in the following order:

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

After acquiring all locks, the transaction does a final check of its status word. If it has not been set to ABORTED by any transaction at this point, then the transaction can successfully commit. It holds all relevant locks in the ConflictTable, thereby excluding any transactional or non-transactional threads that would compete for the same objects. TxOS only locks written objects, as it does not need to update state in most read objects (except for states with loose consistency requirements, discussed in Section 4.3).

The flow of the commit protocol is shown in Figure 5. After acquiring all necessary locks, the transaction copies its updates to the stable objects. The locks are then released in the opposite order they were acquired. Between releasing spinlocks and mutexes, the transaction performs deferred operations.

Note that TxOS is careful to acquire blocking locks before spinlocks. This is because acquiring (or releasing) a mutex or semaphore can cause a process to sleep, and sleeping with a held spinlock can deadlock the system.

4.6 Abort Protocol

If a transaction detects that it loses a conflict, either by checking its status word while trying to commit or add an object to its workset, or by losing a conflict that it detected, it must abort. The abort protocol is similar to the commit protocol, but simpler because it does not update stable objects; it only acquires the appropriate bucket locks on the ConflictTable.

If the transaction is holding any locks, it first releases them to avoid stalling other processes. The transaction then removes its remaining entries from the conflict table, allowing other transactions to access those objects. It then frees its shadow objects and decrements the reference count on their stable counterparts. The transaction then walks its undo log to release any other resources, such as memory allocated within the transaction. If the transaction is responsible for recovering user memory, it restores the user-level checkpoint.

If the transaction is aborted midway through a system call, it restores the register state and jumps back to the top of the stack (like the C library function `longjmp`). Initially we attempted to unwind the stack by returning from each frame and checking return codes, but this was difficult to program. There are simply too many places to check for invalid objects or error conditions, and missing any of them compromises the correctness of the system. Because a transaction can be holding a lock or other resources 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.

5 Evaluation

This section evaluates the performance and behavior of TxOS for our case studies: eliminating TOCTTOU races, scalable atomic operations, tolerating library crashes, and integration with software transactional memory.

All of our experiments were performed on a Dell PowerEdge 2900 server with 2 quadcore Intel X5355 processors (total of 8 cores) running at 2.66 GHz. The machine has 4 GB of RAM and a 300 GB SAS drive running at 10,000 RPM. TxOS is compared against an unmodified Linux kernel, version 2.6.22.6—the same version extended to create TxOS .

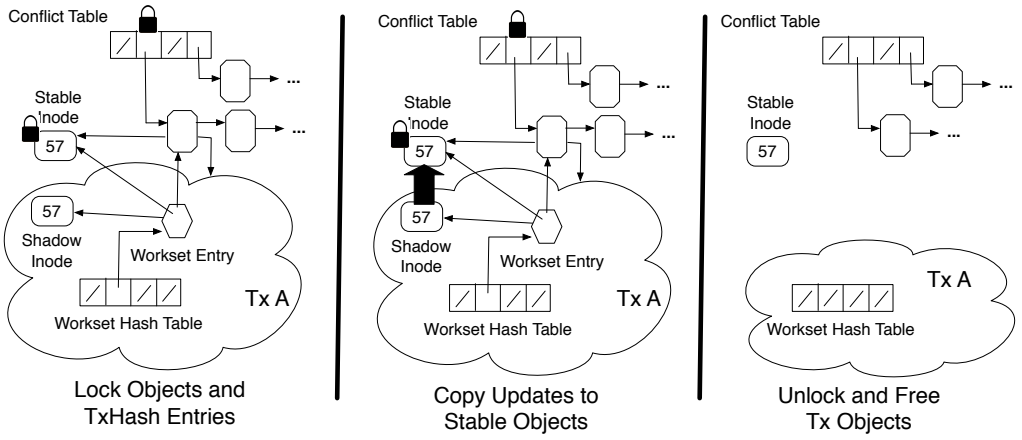

Figure 5: The major steps involved in committing Transaction A with inode 57 in its workset. The commit code first locks the inode and the ConflictTable bucket that the inode's pointer hashes to. It then copies the updates from the shadow inode to the stable inode. Finally, Transaction A frees the resources used for the transactional bookkeeping and unlocks the inode and ConflictTable bucket.

5.1 Withstanding TOCTTOU attacks

Tsafrir et al. provide the current best solution for withstanding TOCTTOU attacks by resolving pathnames in userspace and `stat`-ing each component of the path k times [45]. This technique increases the probability of safe execution, whereas transactions provide a deterministic safety guarantee.

Figure 6 shows the time required to perform an `access`/`open` check as the number of directories in the path name increase. Because of the extra work involved in checking each portion of the path in Tsafrir's technique, performance does not scale well with path length. TxOS has better absolute performance than the Tsafrir technique, and it has better scaling behavior. Its performance is statistically identical to unmodified Linux. In this case, the user pays nothing to guarantee the elimination of TOCTTOU races.

To simulate an attack, we downloaded the attacker used by Borisov et al. [8] to defeat Dean and Hu's probabilistic countermeasure [12]. 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, which allows the attacker to win nearly every time. The attacker uses the access time (atime) on a long maze of symbolic links to infer the progress of the victim.

Both TxOS and Tsafrir's technique successfully resist the attacker. Tsafrir's technique detects an inconsistent `stat` before it completes the check and exits early. The attack is foiled, but the victim is unable to proceed. TxOS reads a consistent view of the directory structure and opens the correct file. Because the attacker's attempt to interpose a symbolic link creates a conflicting update that occurs after the transactional `access` check starts, TxOS puts the attacker to sleep on the asymmetric conflict. The execution time under attack for the Tsafrir countermeasure is 3.00 seconds, whereas the transactional `access`/`open` takes 4.33 seconds. These numbers are skewed in Tsafrir's favor because Tsafrir's technique simply stops once it detects an attack, whereas TxOS correctly completes the operation and thus performs more work.

Transactions provide deterministic safety guarantees and better performance in the common, non-attack case than the current best solution for withstanding TOCTTOU attacks.

5.2 Scalable System Calls

Over the years, system calls like `rename` and `open` have been used as *ad hoc* solutions for the lack of a general-purpose atomic actions. As a result of these system calls becoming semantically heavy, the implementations become complex and don't scale. In particular, `rename` has to serialize all cross-directory renames on a file system mutex because finer-grained locking would be complex and would risk deadlock.

Transactions allow simpler, semantically lighter system calls to be combined to perform heavier weight operations. Figure 7 compares the unmodified Linux implementation of `rename` to calling `sys_xbegin()`, `link`, `unlink`, and `sys_xend()` in TxOS. In this microbenchmark, we divide 500,000 cross-directory renames across a number of threads. Although our transactional `link`/`unlink` has a higher single-thread over-

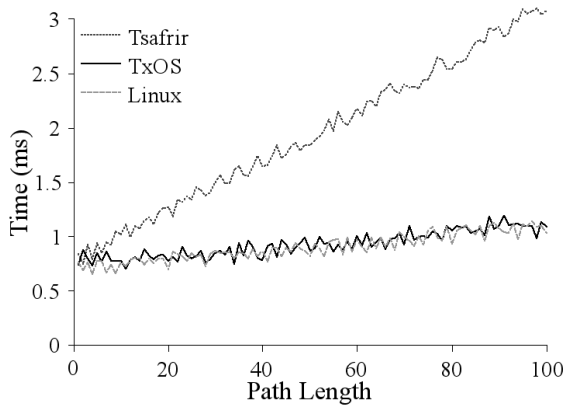

Figure 6: Time to run a simple program that performs an access/open check for increasing path length (lower is better). We present unmodified Linux as a baseline (despite not withstanding a TOCTTOU attack), which overlaps with the line for TxOS . TxOS provides deterministic safety against TOCTTOU, whereas Tsafrir's technique provides only increased probability of success.

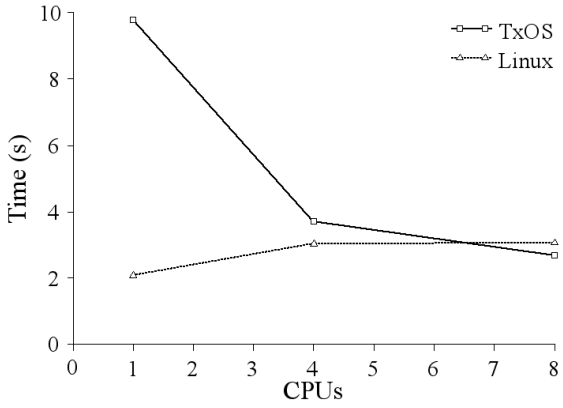

Figure 7: Time to perform 500,000 renames divided across a number of threads (lower is better). TxOS implements its renames as calls to `sys_xbegin()`, `link`, `unlink`, and `sys_xend()`, using 4 system calls for every Linux `rename` call. Despite higher single-threaded overhead, due in large part to an immature implementation, TxOS provides better scalability, outperforming Linux by 14% at 8 processors.

head than `rename` (due largely to an immature implementation), we quickly recover the performance at higher CPU counts, out-performing `rename` by 14%.

The performance of `link` and `unlink` in a system transaction indicates that combining semantically light system calls within a transaction can yield better performance scalability and a simpler implementation.

5.3 Tolerating Faults

In order to demonstrate how an application can use system transactions to detect a fault and recover, we added transactions to Lynx 2.8.6 [2], a console-based web browser. Like most other browsers, Lynx allows websites to send compressed html to save bandwidth, calling a third party library, zlib [4], to unzip the html. But libraries can have vulnerabilities, and zlib has a double free corruption bug in version 1.1.3 [10]. If Lynx is linked with zlib 1.1.3, a malicious web server can send a gzipped html file exploiting the zlib bug, causing Lynx to crash.

Fortunately, by simply wrapping calls to zlib inside a transaction (see Figure 2), Lynx can tolerate faults in zlib. Lynx writes gzipped web pages to a temporary file and then unzips the file and renders it. Before the unzip call, we add code to start a user transaction and also install a SIGABRT signal handler. The libc malloc implementation raises a SIGABRT when it detects memory corruption. If the user visits a compressed webpage that triggers the zlib bug, the signal handler simply aborts the transaction which uses TxOS' recover_user copy-on-write mechanism to roll back the application state to before the unzip began. The browser can then display an error message rather than crashing. The user can then continue to run the same instance of Lynx, and since all transactional state is rolled back, does not need to worry about corruptions to the underlying file system or other system state.

System transactions are necessary for Lynx because the zlib library makes the system calls to read the file being unzipped. The zlib library changes both application and system state. In this case TxOS manages the rollback for both.

In our test case, the overall cost of isolation is quite small. Table 3 shows that when reading small files (2KB), wrapping the decompression code in a transaction incurs a 7% performance overhead compared with the unprotected version. This difference is approximately equal to executing an empty transaction (an `sys_xbegin()` followed immediately by an `sys_xend()`, which takes 243K cycles). When opening larger files (190KB), the transactional overhead of the unzip shrinks to 1%. As Lynx performs many operations other than decompression to display a web page, the end-to-end latency increase for displaying both web pages is less than 1%.

This level of performance overhead demonstrates that isolating a library in transactions is a simple and practical way to tolerate common faults.

Operation	Non-TX	TX	Over-head	End to end
unzip 2KB	3.80M	4.07M	7.37%	<1%
unzip 190KB	138M	140M	1.03%	<1%

Table 3: Cycle cost of wrapping an unzip operation in a transaction for a file containing html text. End to end is the increase vs the cost of fetching, unzipping, and loading a webpage. All values are the average of 8 runs.

5.4 STM with system calls

In this section we evaluate the integration of a Java based software transactional memory system (DATM-J [35]), with system transactions. We extended DATM-J to use the system call API provided by TxOS. The only modification to the STM is to follow the commit protocol as outlined in Section 3.4 when committing a user level transaction that invokes a system call.

We use Tornado, a multi-threaded web server publicly available on sourceforge, to show the benefits of using system transactions in conjunction with an STM. In Tornado, clients connect to the server and make requests to read or write data to files hosted on the server. Tornado has front end threads that listen to ports and puts incoming connections in a work list. The work list is serviced by backend threads from a thread pool. Accesses to this list can be made thread safe by either using an STM or by using locks.

Since multiple clients may be reading or writing to the same file, file access must be serialized. Serialization prevents a read that is concurrent with a write from seeing garbled data in the file, and prevents multiple writers from corrupting a file. A coarse-grained locking strategy would acquire a read-write lock on the directory that contains these files (a read-write lock allows concurrent readers). A single lock is simple but may restrict concurrency and hinder performance. Fine-grained locking may perform better by using a read-write lock for each file. Multiple locks is more complex and requires lock ordering to avoid deadlock, but should improve performance.

System transactions on the other hand are able to provide good performance without the subtleties of fine-grained locking. Current STMs cannot be used in this scenario as it requires making system calls within a user level transaction to read and write file data.

In our experiments we compare a version of Tornado that uses locks to the one that uses DATM-J and system transactions. The coarse-grained locking case uses a single reader-writer lock on the directory serving the files, while the fine-grained case uses per-file locks. Both these variants run on unmodified Linux. Requests are random, with 80% of them reads and the remaining 20% writes. Each client makes a request for one of the four files in a single directory. Figure 8 shows that the STM version outperforms locks as the number of server threads increase. While fine-grained locking outperforms coarse-grained locking, neither scale performance with more processors.

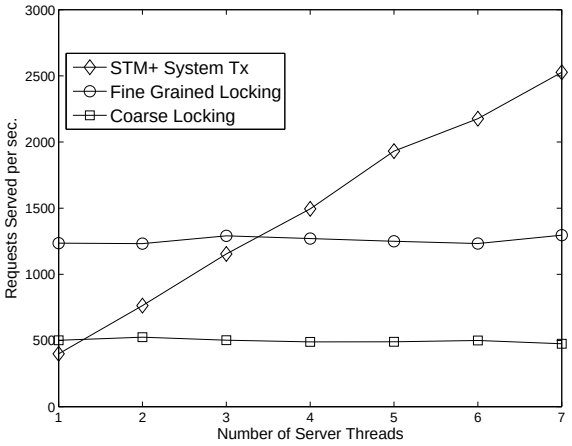

Figure 8: Requests served by the Tornado server when using DATM-J with system transactions versus locks.

System Call	Baseline	Transaction	Overhead
access	2,005	8,572	327%
open	665	781	17%
read	373	427	14%
write	375	431	17%
link	3,270	12,964	296%

Table 4: Execution time in processor cycles of common system calls on unmodified Linux and on TxOS within a transaction.

In terms of absolute performance, transactions scale better than locks because they allow more threads to concurrently access the file system than locks safely can. The STM version performs up to $4.5\times$ better than coarse grained locking and up to $2\times$ better than fine-grained locking. As the number of server threads increase, the STM version incurs more aborts. For example, the system transaction aborts rise from 5 to 49 as the number of threads is increased from 2 to 7. These aborts represent points where concurrent accesses must be serialized by the runtime system.

5.5 Transaction Overhead

Table 4 shows the average execution times of common file system operations, both alone and within a transaction. For these experiments, we disabled user-level re-

covery to focus on the cost overhead for a single system call. Overheads are variable, but quite acceptable on the lower end (13% for `read` and 16% for `write`). The higher overhead on calls such as `link` reflects the fact that less tuning effort has been applied to these system calls, there is nothing inherent that prevents the overheads for all of the system call from being less than ∼25%.

A key performance concern for TxOS is the performance overhead of detecting asymmetric conflicts that is imposed upon non-conflicting, non-transactional applications. In our experience, this has not been a problem because ConflictTable lookups can be elided if the object has no transactional references. We measured the performance overhead of these checks on a microbenchmark which searches `/etc` (containing 1,887 files and 8.9MB data) for a string which is not found. On unmodified Linux, this takes 2.200s, whereas on TxOS this takes 2.429s, an overhead of 10%.

6 Related Work

We distinguish system transactions from previous research in OS transactions, journaling file systems, transactional file systems, Speculator, and transactional memory.

OS transactions. Locus [47] and QuickSilver [16, 38] are historical systems that provide some system support for transactions. The primary goal of these systems is committing file writes atomically with distributed transactions. To get good performance, however, they compromise their isolation semantics. Neither system retains locks on directories, allowing directory contents to change during a transaction. This introduces the possibility of a time-of-check-to-time-of-use (TOCTTOU) race condition. Coordination with user-level transactions (Section 3.4) is another TxOS feature that requires full isolation, which is not provided by either historical system.

These systems also use two phase locking and eager version management. Locking kernel data structures for the duration of a user transaction can deadlock: two transactions simply acquire the same resources in opposite order. Locus does not detect deadlock (but does allow pluggable detection mechanisms), and Quicksilver times out long-running transactions. Timeouts can starve long-running transactions. TxOS does not have to resort to timing out because it uses lazy version management, thus it does not hold locks across system calls. It only holds locks long enough to copy objects and always acquires them in an ordered fashion.

Finally, Quicksilver does not support strong isolation (Section 3.3), and hence does not necessarily serialize non-transactional operations with transactional operations. Locus allows for transactional and non-transactional applications to access the same data, but requires an explicit commit by the non-transactional thread. Uncommitted, non-transactional records are committed by the next transaction to access the data. It is unclear what happens to such a record if the subsequent transaction aborts. The current STM literature shows a number of situations that lead to data structure corruption when strong isolation is not provided [25, 41]. Because TxOS allows transactional and non-transactional updates to kernel data structures, it must provide strong isolation lest the kernel data structures become corrupted.

Journaling file systems. Journaling file systems like ext3 [46] ensure that individual file system actions, like rename, are atomic. A rename failure on ext2 can leave evidence of the new file name. The journal ensures that there are no partial results for individual operations. System transactions allow multiple operations on system resources (not just files).

Transactional file systems. Microsoft Windows Vista has TxF [31], a transactional file system, and transactional file systems have been suggested for Linux, like Amino [48]. These systems allow the grouping of multiple operations on files into transactions, unlike journaling file systems, and both should be able to eliminate TOCTTOU races. However, Amino presupposes a conventional file system that lies outside of the transaction system. The OS boots on a traditional file system which also stores the database which provides Amino's transactional file system. Updates made by system daemons to system directories are not seen by Amino. TxOS provides transactions at the VFS layer, which enables our implementation to work for ext2, proc and tmpfs. TxF is part of NTFS, and would not enable transactions on e.g., a FAT file system. System transactions allow the grouping of non-file system system calls in a transaction so a program could e.g., write a log record and send a signal atomically. Such a facility is not possible with a transactional file system.

Distributed transactions. A number of systems, including TABS [43], Argus [22, 23], and Sinfonia [6] provide support for distributed transactional applications at the language or library level. These papers make important contributions to developing the transactional programming model. Because transactions are implemented at user-level, however, they cannot isolate system resources, whereas TxOS can.

Speculator. Speculator applies to the operating system an isolation and rollback mechanism very similar to transactions, allowing the system to speculate past high-latency remote file system operations [28, 29]. Speculator only buffers speculative state in the kernel, whereas TxOS provides transactional semantics to users. TxOS arbitrates among transactional and non-

transactional threads and integrates with user-level transactions, both of which are not part of Speculator.

Transactional memory. System transactions borrow implementation techniques from software transactional memory systems. TxOS is orthogonal to our previous work on TxLinux, which used hardware transactional memory (HTM) as a synchronization technique within the Linux kernel [34, 36]. TxOS could use HTM for synchronization, but the point of the system is to expose transactions in the system call API. TxOS runs on currently available hardware, though future work might improve performance by using hybrid transactional memory mechanisms [11, 42].

6.1 Future work

This paper focuses on transactional support for system calls that operate on the file system, as this is one of the more natural fits for the transaction programming model. There are portions of the system such as the networking and graphical display utilities where interaction with an external entity (e.g., another server or the desktop user) may be required within a transaction. The correct semantics for system transactions is unclear. Deferring all output until commit is a classic approach to this problem, yet some systems have allowed these round-trip communications within transactions [38, 43], with the requirement that the other end be able to tolerate inconsistencies introduced by a transaction restarting. We plan to investigate this issue in future work as we gain more experience with transactional programming in TxOS.

We also plan to explore several additional target applications in future work, including the interaction of transactions and scheduling to support multi-process transactions for applications such as shell scripts. We also believe the latency hiding techniques from Speculator [28] and xsyncfs [29] will directly apply to TxOS, allowing TxOS to mask the cost of the synchronous writes required to support durability. Finally, transactions can be used to enforce system security policies by inspecting the working set of a transaction at commit time to insure that it has not inadvertently accessed a protected resource.

The object versioning, rollback, and isolation mechanisms used to implement system transactions also provide a useful platform for OS research. At a high level, many good ideas from recent systems conferences require custom variants of these mechanisms, including Speculator [28] and Rx [32]. We believe that generalizing TxOS to reimplement some of these ideas will make it a strong platform for future systems research.

7 Conclusion

This paper demonstrates that system transactions provide a powerful abstraction by efficiently addressing a wide range of programming challenges. Adding efficient transactions to the Linux system call API makes programming easier, more secure, and increases performance.

References

[1] Gnu emacs. `http://www.gnu.org/software/emacs`.

[2] Lynx web browser. `http://lynx.isc.org`.

[3] Subversion. `http://subversion.tigris.org`.

[4] zlib compression library. `http://www.zlib.net`.

[5] A.-R. Adl-Tabatabai, B. T. Lewis, V. Menon, B. R. Murphy, B. Saha, and T. Shpeisman. Compiler and runtime support for efficient software transactional memory. In *PLDI*, pages 26–37, Jun 2006.

[6] M. K. Aguilera, A. Merchant, M. Shah, A. Veitch, and C. Karamanolis. Sinfonia: a new paradigm for building scalable distributed systems. In *SOSP '07: Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles*, pages 159–174, New York, NY, USA, 2007. ACM.

[7] M. Bishop and M. Dilger. Checking for race conditions in file accesses. *Computing Systems*, 9(2):131–152, Spring 1996.

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

[9] D. Bovet and M. Cesati. *Understanding the Linux Kernel*. O'Reilly Media, Inc., 3rd edition, 2005.

[10] CERT Vulnerability Database. Double free bug in zlib compression library. 2002. http://www.cert.org/advisories/CA-2002-07.html.

[11] P. Damron, A. Fedorova, Y. Lev, V. Luchangco, M. Moir, and D. Nussbaum. Hybrid transactional memory. In *ASPLOS*, pages 336–346, 2006.

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

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

[14] J. Gray. Notes on data base operating systems. In *Operating Systems, An Advanced Course*, pages 393–481, London, UK, 1978. Springer-Verlag.

[15] 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*, page 102, June 2004.

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

[17] M. Herlihy and E. Koskinen. Transactional boosting: A methodology for highly-concurrent transactional objects. In *PPoPP*, pages 207–216. ACM, 2008.

[18] M. Herlihy, V. Luchangco, M. Moir, and I. William N. Scherer. Software transactional memory for dynamic-sized data structures. In *PODC*, pages 92–101. ACM, 2003.

[19] M. Herlihy and J. E. Moss. Transactional memory: Architectural support for lock-free data structures. In *ISCA*, May 1993.

[20] O. S. Hofmann, D. E. Porter, C. J. Rossbach, H. E. Ramadan, and E. Witchel. Solving difficult HTM problems without difficult hardware. In *ACM TRANSACT Workshop*, 2007.

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

[22] B. Liskov. Distributed programming in Argus. *Commun. ACM*, 31(3):300–312, 1988.

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

[24] V. Marathe, M. Spear, C. Heriot, A. Acharya, D. Eisenstat, W. S. III, and M. Scott. Lowering the overhead of nonblocking software transactional memory. In *TRANSACT*, 2006.

[25] V. Menon, S. Balensiefer, T. Shpeisma, A. Tabatabai, R. Hudson, B. Saha, and A. Welc. Single global lock semantics in a weakly atomic STM. In *TRANSACT*, 2008.

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

[27] 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.

[28] E. B. Nightingale, P. M. Chen, and J. Flinn. Speculative execution in a distributed file system. In *SOSP*, pages 191–205, 2005.

[29] E. B. Nightingale, K. Veeraraghavan, P. M. Chen, and J. Flinn. Rethink the sync. In *OSDI*. USENIX Association, 2006.

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

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

[32] F. Qin, J. Tucek, J. Sundaresan, and Y. Zhou. Rx: Treating bugs as allergies—A safe method to survive software failures. In *SOSP*, Oct 2005.

[33] R. Rajwar and J. R. Goodman. Transactional lock-free execution of lock-based programs. *SIGARCH Comput. Archit. News*, 30(5):5–17, 2002.

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

[35] H. Ramadan, I. Roy, M. Herlihy, and E. Witchel. Dependence-aware transactional memory. Technical Report TR-08-21, University of Texas at Austin, Computer Sciences Department, 2008.

[36] 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.

[37] M. Satyanarayanan, H. H. Mashburn, P. Kumar, D. C. Steere, and J. J. Kistler. Lightweight recoverable virtual memory. *ACM Transactions on Computer Systems*, 12(1):33–57, 1994.

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

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

[40] M. I. Seltzer, Y. Endo, C. Small, and K. A. Smith. Dealing with disaster: Surviving misbehaved kernel extensions. In *OSDI*, 1996.

[41] T. Shpeisman, V. Menon, A.-R. Adl-Tabatabai, S. Balensiefer, D. Grossman, R. L. Hudson, K. F. Moore, and B. Saha. Enforcing isolation and ordering in STM. *PLDI*, 42(6):78–88, 2007.

[42] A. Shriram, V. J. Marathe, S. Dwarkadas, M. L. Scott, D. Eisenstat, C. Heriot, W. N. Scherer III, and M. F. Spear. Hardware acceleration of software transactional memory. In *TRANSACT*, 2006.

[43] A. Z. Spector, D. Daniels, D. Duchamp, J. L. Eppinger, and R. Pausch. Distributed transactions for reliable systems. In *SOSP*, pages 127–146, 1985.

[44] M. Swift, M. Annamalai, B. Bershad, and H. Levy. Recovering device drivers. In *OSDI*, 2004.

[45] D. Tsafrir, T. Hertz, D. Wagner, and D. D. Silva. Portably solving file TOCTTOU races with hardness amplification. In *FAST*, pages 189–206, 2008. Best paper award winner.

[46] S. Tweedie. Ext3, journaling filesystem. `http://olstrans.sourceforge.net/release/OLS2000-ext3/OLS2000-ext3.html`.

[47] M. J. Weinstein, J. Thomas W. Page, B. K. Livezey, and G. J. Popek. Transactions and synchronization in a distributed operating system. In *SOSP*, pages 115–126, 1985.

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

[49] F. Zhou, J. Condit, Z. Anderson, I. Bagrak, R. Ennals, M. Harren, G. Necula, and E. Brewer. SafeDrive: Safe and recoverable extensions using language-based techniques. In *OSDI*, 2006.

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