Operating System 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-08-50, December 16, 2008

Abstract

Operating systems should provide *system transactions* to user applications, in which user-level processes execute a series of system calls atomically and in isolation from other processes on the system. System transactions provide a simple tool for programmers to express safety conditions during concurrent execution. 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 strong isolation and fairness between transactional and non-transactional system calls.

System transactions provide a simple and expressive interface for user programs to avoid race conditions on system resources. For instance, system transactions eliminate time-of-check-to-time-of-use (TOCTTOU) race conditions in the file system which are a class of security vulnerability that are difficult to eliminate with other techniques. System transactions also provide transactional semantics for user-level transactions that require system resources, allowing applications using hardware or software transactional memory system to safely make system calls. While system transactions may reduce single-thread performance, they can yield more scalable performance. For example, enclosing `link` and `unlink` within a system transaction outperforms `rename` on Linux by 14% at 8 CPUs.

1 Introduction

The prevalence of concurrency due to the proliferation of multicore processors has created a problem for the system call API; current operating systems provide insufficient mechanisms for applications to synchronize these operations. 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. In this paper, we show that system transactions are a similar, broadly applicable abstraction and that transactions belong in the system-call API. Without system transactions, important functionality is impossible or difficult to express.

System transactions provide a simple tool for programmers to express safety conditions during concurrent execution. During a system transaction, the kernel insures that from the user's perspective, no other transaction or non-transactional system call occurs. The user experiences serial execution of the transactional code, eliminating race condtions. Within a system transaction, a series of system calls either execute completely or not at all (atomicity), and in-progress results are not visible (isolation). For example, if one program executes a system transaction that performs two writes to a file, then any program reading that file either sees both writes or neither. System transactions have a simple interface: the user starts a system transaction with the `sys_xbegin()` system call, ends a transaction with the `sys_xend()` system call, and aborts the current transaction with the `sys_xabort()` system call.

This paper introduces TxOS, a variant of Linux 2.6.22 that supports system transactions on commodity hardware. TxOS is the first operating system to support transactions in which any sequence of system calls can execute atomically and in isolation. Unlike historical attempts to add transactions to the OS, transactions in TxOS have stronger semantics, are more efficient, and support a flexible contention management policy between transactional and non-transactional operations. TxOS is unique in its ability to enforce transactional isolation even for non-transactional threads, which is key for making system transactions practical, allowing the OS to balance scheduling and resource allocation fairly between transactional and non-transactional operations. TxOS achieves these goals by using modern, software transactional memory (STM) techniques.

Current operating systems address race conditions in the system API by adding complicated, *ad hoc* extensions to the API, whereas system transactions provide a simple, expressive interface for synchronizing access to system resources. For instance, time-of-check-to-time-of-use (TOCTTOU) race conditions are easier to exploit on multicore platforms [41]. In the past few years, Linux has added file system APIs that take file descriptor arguments to address TOCTTOU races (`openat`, `renameat`, `faccessat`, and ten others), and it has redesigned

its signal-handling API (`sigaction`, `sigprocmask`, `pselect`, `epoll_pwait` and others). Such APIs are intended to solve specific races in a concurrent execution environment, but they have complex semantics and are difficult to learn and master. System transactions provide a single, easily understood, general mechanism that can express safe operations using simpler APIs, such as `open` or `signal`. Instead of fixing particular race conditions with new system calls, system transactions provide a general mechanism to eliminate race conditions completely.

TxOS adds transactions to its system call API, while TxLinux [36] uses hardware transactions to implement the same API as unmodified Linux. TxOS does not use hardware transactions at all, and by themselves hardware transactions are not sufficient to implement a transactional system call API. This paper describes the challenges to providing such an API.

TxOS lets user-level transactions make system calls with full transactional semantics. It provides a simple and semantically complete way for user-level transactions, such as those provided by hardware or software transactional memory systems, to access system resources. User-level transactions cannot make most system calls without violating isolation because system call results become visible to the rest of the system. Attempts to address this limitation are discussed in Section 2.2.1, but they either compromise transactional semantics or greatly increase the complexity and decrease the performance of the transactional memory implementation. In Section 3.4, we show how to coordinate user and system level transactions into a seamless whole while maintaining full transactional semantics.

To support system transactions, the kernel must isolate and undo updates to shared resources. This process 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, system transactions can provide better performance scalability than locks as we show with a web server in Section 5.5, which uses transactions to increase throughput 47% over a server that uses fine-grained locking.

This paper makes the following contributions:

- Describes a new approach to implementing system transactions on commodity hardware, which provides strong atomicity and isolation guarantees, while maintaining a low performance overhead.
- Demonstrates that system transactions can express useful safety conditions by eliminating race conditions while maintaining scalable performance. The performance of TxOS TOCTTOU elimination is superior to the current state-of-the-art user-space technique [40]. Placing `link` and `unlink` in a transaction can outperform `rename` on Linux by 14% at 8 CPUs.
- Shows how to maintain transactional semantics for user-level transactions that modify system state, and measures performance for integrating a software and a hardware transactional memory system with TxOS. We show that TxOS resolves a memory leak in genome (a STAMP benchmark [9]) without modification of the program or libc.

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');
   read(fd,...);
   ...
} else { sys_xend(); }
```

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

The paper is organized as follows: Section 2 motivates system transactions with examples of the problems they solve. 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 summarizes our findings.

2 Overview and motivation

This section motivates the need for system transactions by describing how they eliminate race conditions and how they complete the programming model of user-level transactions.

2.1 Eliminating races for security

Time-of-check-to-time-of-use (TOCTTOU) race conditions are a current source of serious security vulnerabilities, and a good example of the kinds of race conditions that system transactions can eliminate. Its most (in)famous instance is the `access`/`open` exploit in the UNIX file system, illustrated in Figure 1 [6]. The TOCTTOU race also arises in temporary file creation and other accesses to system resources. 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., writes to 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 [31].

System transactions eliminate race conditions. If the user starts a system transaction before doing their system calls (e.g., an `access` and `open`), then the operating system guarantees that the interpretation of the path name used in both calls will not change during the transaction (shown in Figure 1(B)). The attacker is serialized either before or after the transaction.

Eliminating race conditions has motivated the Linux developers to add thirteen new system calls in 2006 (the `openat` family of system calls, also supported by Solaris). This is not surprising, as Dean and Hu showed that there is no portable, deterministic solution to TOCTTOU vulnerabilities without changing the system call interface [11]. Developers continue to plug race condition vulnerabilities with plans to add the close-on-exec flag to fifteen system calls for the 2.6.27 version of Linux [13]. The flag eliminates a race condition between calls to `open` and `fcntl`. This ad-hoc strategy to fixing race conditions is not likely to be effective. Complicating the API to provide security is a risky approach as code complexity is often the enemy of code security [5].

To summarize, the current best approaches to eliminating race conditions on file system names are either:

1. Operate at user level and settle for increasing the probability of detecting the race [40]
2. Add system calls that operate on open file descriptors, whose interpretation will not change like a path name (e.g., `openat`)
3. Add flags to individual system calls, because each system call executes atomically (e.g., the `O_CREAT|O_EXCL` flags for `open` to replace a `stat`, `open` pair)
4. Remove the functionality altogether (usually unacceptable)

System transactions provide a general way to eliminate race conditions that is easy to use, provides deterministic safety guarantees, and a natural programming model.

2.2 Completing user-level transaction model

Transactional memory is an alternative to lock-based programming. It provides a simpler programming model than locking while maintaining good performance scalability [19, 22]. Transactional memory is generally implemented either in hardware, building on cache coherence [15, 19, 27], or in software, as a library or as an extension to the JVM or other runtime system [12, 26].

```
process_records(int f1, int f2) {
// Read records from files in lock step
  atomic {
    read(f1, buf, 16);//read record
    process_record(buf, 16);
    read(f2, buf, 8);//read record
    ...
  }
}
```

```
process_records(int fd, char* buf) {
  s = xbegin;
  if s == ABORT &&
     made_syscall(xsw) {
      sys_xabort();
  }
  read(fd, buf, 16);//implicit  sys_xbegin
  process_record(buf, 16);
  read(f2, buf, 8);
  ...
  if(made_syscall(xsw))
      sys_xend();
  else
     xend; //hardware instruction
```

Figure 2: An example to show how user and system transactions coordinate. The top portion is application code with a transactional memory critical region that contains system calls. The lower portion is pseudo-code for how an HTM system could expand the code to coordinate the user and system transactions. The `sys_xbegin()` system call is implicit. The xsw is the transactional status word.

Transactional memory is a technique to manage *application state* (or user state). The data structures within an application's address space are the application state. System transactions manage *system state*. Writes to the file system or forking a thread are actions that update the system state. This section explains the benefits of having OS support for transactional updates to system state.

2.2.1 Problems with system calls in transactions

Support for arbitrary system calls within a hardware or software transaction is difficult. For system calls in transactions, anything less than system transactions either fails to provide transactional semantics or becomes complicated, requiring non-transactional synchronization. Figure 2(top) shows an example of a transaction that includes two system calls. Simply making the system calls as part of the transaction is not acceptable, because a roll back will cause the system calls to be reexecuted, causing records to be lost.

Open nesting [28, 29] and escape actions [28, 45] have been proposed as partial solutions for system calls within transactions. However, open nesting and escape actions

have complicated and fragile correctness requirements to avoid semantic anomalies [28]. Even accepting complications, there is debate about how many system calls can be made safe with these techniques [21]. Finally, no one claims that these techniques are sufficient to handle system calls like `write` or `rename`, which update system state, because such calls clearly violate isolation. However, there are claims that these techniques support idempotent calls like `pread` [44] (`pread` is a variant of `read` that does not update the current file position).

Consider the problems implementing Figure 2(top) with escape actions [28, 45]. Assume that the transaction is paused before each `read` system call, and any undo actions are enqueued during the pause. Each `read` is undone by a `seek` that rewinds the file position the length of the read. The problem is that during a roll back both files cannot be seeked atomically. Assume that a thread is executing the transaction in the Figure, and the transaction restarts. As an undo action, the system seeks -8 bytes on file descriptor `f2`. Then another thread executes the same transaction, reading the same files as the transaction that is rolling back. However, the files are out of sync because `f2` has had its file position reset, but the undo action has not reset `f1`. It might be possible to fix this example with additional, non-transactional synchronization, like locking, but the complexity of the TM system increases sharply, and its performance is likely to suffer.

Even if all programs were updated to use `pread` instead of `read`, and issues relating to `seek` are ignored, problems persist. One of the file descriptors in the example might refer to a pipe. Reads from a pipe are destructive. In order to preserve the ability to roll back a transaction, the runtime system would have to buffer the data from the pipe reads. The runtime would then need to synchronize the contents of the buffer with other threads by non-transactional means. Again, if it could be made to work, the complexity of the TM system increases sharply and maintaining scalable performance would be difficult. By encapsulating this complexity in the OS, the performance can be optimized and the effort shared among all TM systems.

Transactional semantics for the example can be provided by escape actions and a transactional file system, like TxF [32] in Microsoft Windows Vista, assuming that both file descriptors refer to files stored in the transactional file system. The runtime could start a file system transaction before the first read and commit the file system transaction after the user transaction commits. The user-level transaction and the file system transaction would need coordination for commit, using a protocol similar to TxOS that is described in Section 3.4). However, if one of the file descriptors refers to a pipe, then file system transactions will not help. Of course, a transactional file system would also be unable to roll back any non-file system system call that happens during the transaction.

At the root of this problem is the fact that the operating system does not expose a synchronization mechanism for its resources to applications, and applications cannot know all of the side effects of a given set of system calls. TxOS solves these problems by providing a mechanism to synchronize accesses to system resources and undo all side effects of uncommitted operations.

2.2.2 Transactional system calls

System transactions provide a mechanism that transactional memory systems can use to safely allow almost any system call within a transaction. When a TM application performs an operation that makes a system call, the runtime will begin a system transaction. Figure 2(bottom) provides pseudo-code that a hardware 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 rolling back of 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.

In our HTM model, there is a transaction status word register (`xsw`) that stores the current transaction identifier (or zero if there is no transaction) and has a bit for the OS to set if a system call is made inside a transaction. This bit remains set across transactional restarts, and it is cleared by `sys_xabort()` and by a successful `sys_xend()`. The kernel can detect an active transaction during the first `read` system call, by reading the current transaction identifier. It starts a kernel transaction and sets the bit in the transaction status word indicating a system transaction. The user application can then check this bit when committing and make an `sys_xend()` system call if there are transactional system calls pending, or issue the `xend` instruction directly otherwise. Alternatively the `xend` instruction could automatically trap to the OS, but we elected to minimize the complexity of the hardware support. The example would be similar for an STM, though the runtime would generate an explicit call to `sys_xbegin()`. The programmer is freed from the complexity of writing and synchronizing undo actions and need only reason about transactions.

System transactions thus expand the transactional programming model, bringing user-level transactions to a larger class of applications. The limitations of system transactions are discussed in Section 3.5. In Section 5, we evaluate the use of system calls within a transactional web server running on a Java STM system, and a STAMP benchmark calling `mmap` in an HTM.

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

Table 1: TxOS API

3 TxOS Design

This section outlines the design of system transactions in TxOS, which is inspired by recent advances in software transactional memory systems [1, 17, 22, 25]. In particular, we describe the user-level API for system transactions, the preservation of isolation 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. The `sys_xbegin()` system call starts a system transaction and the `sys_xend()` system call commits the transaction. The `sys_xabort()` system call 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 to enclose 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.

As shown in Figure 2(bottom), calls to `sys_xbegin()` can be implicit when system transactions are used as part of a hardware transactional memory system. By using the transactional status word for communication, the user transaction minimizes the number of kernel crossings.

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 affected kernel data structures directly. This isolation is performed by adding a level of indirection in the form of **shadow** objects 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 object 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 the system state. When the transaction commits, updates to shadow objects are copied to their stable counterpart.

In the case of a conflict, the system can abort and retry a transaction. 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. Application state is maintained by a transactional memory system, or 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.

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, both of which 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 to 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 (at user level) but before the transaction ends. Holding system locks while a thread executes at user level can deadlock the kernel. This issue is discussed further in Section 6.

TxOS uses an object-based STM, not a word-based STM. The object-based system requires less bookkeeping than a word-based system, and works with the current locking strategy of the kernel. Often, kernel-specific conflict resolution obtains the benefits of finer granularity. For instance, updates to the reference count or the access time on an inode do not create conflicts (see Section 4.3 for a full discussion).

3.3 Conflict detection and resolution

Transactions can conflict with other transactions or non-transactional operations. We call a process executing a transaction a transactional process or a transactional (kernel) thread. A transactional process could read 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 de-

tected and resolved by the system. As we discuss in the implementation section, TxOS detects conflicts by having stable objects point to active transactions. Modifying stable objects allows TxOS to detect conflicts between transactions and between transactional non-transactional threads.

Unlike most software transactional memory systems, TxOS guarantees **strong isolation**[1]. 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 [26,38]. For instance, strong isolation prevents the inadvertent loss of a non-transactional update by a transaction that is rolling back. 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 TxLinux uses hardware transactional memory for synchronization within the OS 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 (i.e., timestamp [34]), guaranteeing liveness within a given priority level.

3.3.2 Asymmetric conflicts

A conflict between a transactional and non-transactional thread is called an **asymmetric conflict** [35]. Unlike transactional threads, non-transactional threads cannot be rolled back, so the system has fewer options when dealing with these conflicts. However, TxOS must have the freedom to resolve an asymmetric conflict in favor of either the transactional or non-transactional thread. Otherwise, asymmetric conflicts will undermine fairness in the system, possibly starving transactions.

While non-transactional threads cannot be rolled back, they can often be preempted, which allows them to lose conflicts with transactional threads. Kernel preemption is a recent feature of Linux that allows processes to be preemptively descheduled while executing system calls inside the kernel, unless they are inside of certain critical regions. In TxOS, non-transactional threads detect conflicts with transactional threads before they actually update state, usually when they acquire a lock for a kernel data structure. A non-transactional thread can simply deschedule itself if it loses a conflict and is in a preemptible state. If a non-transactional, non-preemptible process aborts a transaction too many times, the kernel can still prevent it from starving the transaction. 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). Otherwise, TxOS risks a deadlock with the committing transaction, which might need that lock to commit. By using kernel preemption and lazy version management, TxOS has more flexibility to coordinate transactional and non-transactional threads than was possible in previous transactional operating systems.

3.4 Coordinating User and System Transactions

When a user-level TM creates a system transaction, 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 commits with TxOS, we use a simplified variant of the two-phase commit protocol (2PC) [14]. The details of the protocol are discussed in Section 4.1. The TxOS commit consists of the following steps.

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

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

Alternatively, the kernel could prepare first. TxOS does

[1] Also called strong atomicity.

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 HTM support, TxOS requires that the hardware allow the kernel to suspend user-initialized transactions on entry to the kernel. Every HTM proposal that supports an OS [27,36,45] contains the ability to suspend user-initiated transactions so that user and kernel addresses do not enter the same hardware transaction. Doing so would create a security vulnerability in most HTM proposals. Also, the kernel needs to be able to commit a user-level transaction.

Figure 2 shows how an HTM can use system transactions. The `xbegin` instruction writes a non-zero transaction identifier into the transaction status word (xsw) indicating an active user-level transaction. During the transaction's first system call, the OS reads this register, detects the active user transaction and starts a system transaction (with the restart flag equal to false). The OS sets a bit in the xsw indicating an active system transaction. The user must check this bit before issuing the `xend` instruction and trap into the kernel (via `sys_xend()`) if the bit is set. During the xend system call, the kernel tries to commit the system transaction and the user transaction using 2PC. The `xend` instruction, even when issued in the kernel, commits the user transaction because the xsw indicates an active user transaction but no active kernel transaction. The kernel then clears the active system transaction bit in the xsw. If both transactions commit, the kernel returns to user level. If they do not both commit, the kernel updates the xsw to indicate transaction failure. The hardware will then roll back the user transaction once the kernel returns to userspace.

Though TxOS supports user-level HTM, it runs on commodity hardware and does not require any special HTM support itself. Performance could possibly be accelerated using hardware transactional memory, similar to TxLinux [36]. We leave this question for future work.

3.5 Limitations of system transactions

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 have not been written to stable storage. TxOS uses the semantics of the underlying file system. For example, ext3 would journal updates at commit time, whereas ext2 would write them back at its leisure. As part of future work, TxOS's transactions could integrate with a transactional file system, providing durability for system transactions that access the file system.

As currently designed, 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 future work by spilling the data to swap space or unallocated blocks of the file system.

TxOS ensures isolation only within a single system. Correctly isolating updates to a shared, distributed file system, such as 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.

4 Implementation

System transactions in Linux add roughly 2,600 lines of code for transaction management, and 4,000 lines for object management. 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 are largely in the virtual file system, the memory management code, and the scheduling code.

TxOS modified the following Linux data structures to be able to participate in transactions (described in more detail by Bovet and Cesati [8]): `inode`, `dentry`, `super_block`, `file`, `vfsmount`, `list_head`, `hlist_head`, and `vm_area_struct`. These are file system and process address space data structures, the kernel linked list, and the kernel hashtable implementations. These data structures are modified during essential file system and memory management operations, such as `open()`, `read()`, `write()`, `link()`, `unlink()`, `close()`, and `mmap()`.

4.1 Managing transaction state

To manage transactional state, TxOS adds a transaction object to the kernel which stores transactional metadata and statistics. The transaction object, shown in Figure 3, is pointed to by the kernel thread's control block (the `task_struct` in Linux). A process can have at most one active transaction, though transactions can flat nest, meaning that all nested transactions are rolled into the enclosing transaction.

The fields of the transaction object are summarized

```
struct transaction {
  atomic_t tx_status;
      // live, aborted, inactive
  uint64 tx_start_time;
      // timestamp for contention management
  uint32 retry_count;
  struct pt_regs *checkpointed_registers;
     //register state at beginning of tx
  workset_hlist *workset_hashtable;
     // Used for conflict detection
  deferred_ops;
     // operations deferred until commit
  undo_ops;
     // ops that must be undone at abort
}
```

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

in Figure 3. The transaction includes a status word (`tx_status`), that another thread can update atomically if it wins a conflict with this thread. The kernel checks the status word when attempting to add a new shadow object to its workset and checks it 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, which they store in their `task_struct`, for fair contention resolution with transactions. The `retry_count` field stores the number of times the transaction aborted.

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

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, some operations must be undone if a transaction is aborted, such as releasing the locks it holds and freeing the memory it allocates. These are stored in the `undo_ops` field.

The 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, a pointer to the shadow copy, information about whether the object is read-only or read-write, and a set of type-specific methods (commit, abort, lock, unlock, release). When a transactional thread adds an object to its workset, the thread increments the reference count on the stable copy. This increment prevents the object from being unexpectedly freed while the transaction still has an active reference to it. Kernel objects are not dynamically relocatable, so ensuring a non-zero reference count is sufficient for guaranteeing that memory addresses remain consistent for the duration of the transaction.

4.2 Conflict detection

To detect conflicts, TxOS leverages the current locking practice in Linux and augments stable objects with information about transactional readers and writers. Conflicts may occur when one thread wants to write an object. A locked object indicates a non-transactional writer. For all stable objects, TxOS adds a pointer to a transactional writer, so a non-null pointer value indicates an active transactional writer. An empty reader list similarly indicates there are no readers. By locking and testing the transactional readers and writer fields, TxOS detects transactional and asymmetric conflicts. When a thread detects a conflict, it calls the contention manager to arbitrate. The contention manager updates the `tx_status` field of a 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 lenient semantics where multiple writes do not constitute a conflict.

TxOS contains special cases where writes are not considered conflicts or writes are deferred to increase concurrency on kernel objects. For example, 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 that threads in the kernel memory management system (`kswapd`) use to reclaim memory from the cache file system data. We allow `kswapd` to freely modify these fields when the modifications do not inter-

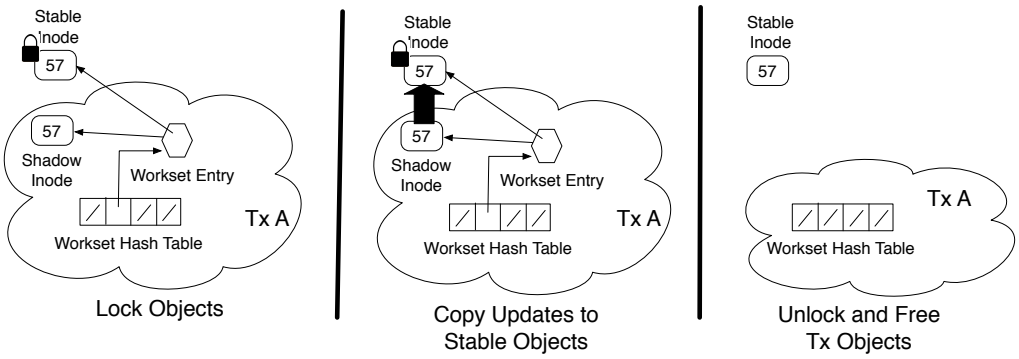

Figure 4: The major steps involved in committing Transaction A with inode 57 in its workset. The commit code first locks the inode. It then copies the updates from the shadow inode to the stable inode. Finally, Transaction A frees the resources used for the transactional bookkeeping and unlocks the inode.

fere with a transaction, such as moving a directory entry 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. Avoiding a deadlock in these systems is difficult because transactions are composed of independent operations. 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 flow of the commit protocol is shown in Figure 4. In the first step, the transaction acquires all of the locks in the conflict table that protect objects in its workset. The transaction collects the locks in the following order:

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

After acquiring all locks, the transaction does a final check of its status word. If it has not been set to ABORTED, then the transaction can successfully commit (this is the transactions' linearization point [20]). If the user has a corresponding hardware transaction, the kernel will attempt to issue the `xend` instruction on its behalf. If the hardware commit is successful, the kernel will proceed to commit its state, aborting otherwise. The committing process holds all relevant object locks during commit, thereby excluding any transactional or non-transactional threads that would compete for the same objects.

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

4.6 Abort Protocol

If a transaction detects that it loses a conflict, it must abort. The abort protocol is similar to the commit protocol, but simpler because it does require all objects to be locked at once. If the transaction is holding any locks, it first releases them to avoid stalling other processes. The transaction then iterates over its working set and locks each object, removes any references to itself from the object's transactional state, and then unlocks the object. This allows other transactions to access the objects in its working set. Next, the transac-

tion frees its shadow objects and decrements the reference count on their stable counterparts. The transaction walks its undo log to release any other resources, such as memory allocated within the transaction.

5 Evaluation

This section evaluates the performance and behavior of TxOS for our case studies: eliminating TOCTTOU races, scalable atomic operations, and integration with hardware and 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 . The HTM experiments were run using MetaTM [35] on Simics version 3.0.27 [24]. The simulated machine has 16 1000 MHz CPUs, each with a 32k L1 and 4 MB L2 cache. An L1 miss costs 24 cycles and an L2 miss costs 350 cycles. We used the timestamp contention management policy and linear backoff on restart.

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 [40]. This technique increases the probability of safe execution, whereas system transactions provide a deterministic safety guarantee.

Figure 5 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 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. [7] to defeat Dean and Hu's probabilistic countermeasure [11]. 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.

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. The attacker's attempt to interpose a symbolic link creates a conflicting update that occurs after the transactional `access` check starts, so, TxOS puts the attacker to sleep on the asymmetric conflict.

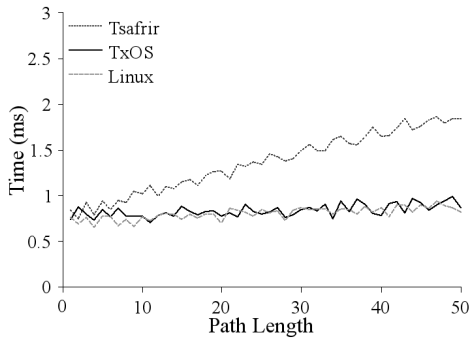

Figure 5: Time to run a simple program that performs an access/open check for increasing path length (lower is better). We present unmodified Linux as a baseline (despite not withstanding a TOCTTOU attack), which overlaps with the line for TxOS . TxOS provides deterministic safety against TOCTTOU, whereas Tsafrir's technique provides only increased probability of success.

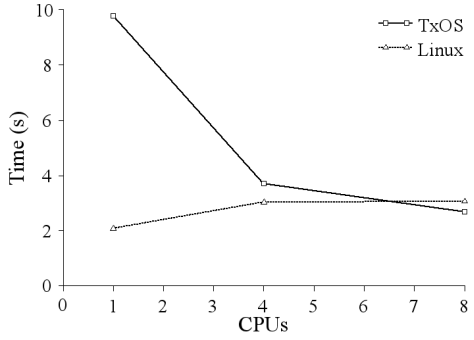

Figure 6: Time to perform 500,000 renames divided across a number of threads (lower is better). TxOS implements its renames as calls to `sys_xbegin()`, `link`, `unlink`, and `sys_xend()`, using 4 system calls for every Linux `rename` call. Despite higher single-threaded overhead, TxOS provides better scalability, outperforming Linux by 14% at 8 processors.

5.2 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 become semantically heavy, resulting in complex implementations whose performance does not scale. As an example in Linux, `rename` has to serialize all cross-directory renames on a single file-system-wide mutex because finer-grained locking would risk deadlock.

Transactions allow simpler, semantically lighter system calls to be combined to perform heavier weight operations yielding better performance scalability and a simpler im-

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 2: 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.

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

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

plementation. Figure 6 compares the unmodified Linux implementation of `rename` to calling `sys_xbegin()`, `link`, `unlink`, and `sys_xend()` in TxOS. TxOS has much worse single-thread performance because it makes four system calls for each Linux system call. But TxOS quickly recovers the performance at higher CPU counts, out-performing `rename` by 14%. The scalability is directly due to TxOS using optimistic synchronization, while Linux uses conservative synchronization and must avoid deadlock.

5.3 Transactional access to system resources

To demonstrate the utility of system transactions without user-level transactions, we added a system transaction to Lynx 2.8.6, a console-based web browser. Like most browsers, Lynx allows websites to send compressed html to save bandwidth, calling a third party library, zlib, to unzip the html. 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. This example does not use a transactional memory system, so application state is rolled back using copy-on-write page table support in the kernel. On an `sys_xbegin()`, the kernel snapshots the user's page table and directs all writes during the transaction to new pages. This technique is only appropriate for single-threaded applications.

System transactions are necessary for Lynx because the zlib library makes the system calls to read the file being unzipped. Lynx writes gzipped web pages to a temporary file and then unzips the file and renders it. The application programmer simply wraps the call to zlib in a transaction, and installs 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. The browser can then display an error message rather than crashing, and the user can continue to run the same instance of Lynx. Since the transactional state is rolled back, he does not need to worry about corruptions to the underlying file system or other system state.

In our test case, the overall cost of isolation is quite small. Table 2 shows that the transactional overhead just for unzipping a file is 7% for a 2KB file and 1% for a 190KB file. This overhead is approximately equal to executing an empty transaction (an `sys_xbegin()` followed immediately by an `sys_xend()`, which takes 243K cycles). Lynx performs many operations other than decompression to display a web page, so the end-to-end latency increase for displaying both web pages is less than 1%.

5.4 HTM with system calls

In this section we evaluate the integration of hardware memory transactions with TxOS. We use the genome benchmark from the STAMP benchmark suite [9], which allocates memory during a transaction. We replace the default bump allocator with the Hoard allocator [4], which is optimized for parallel allocation. Linux experiments were run on version 2.6.16.1.

In this workload, sometimes the allocator makes a system call (`mmap`) to get more memory in a transaction (this happens both with Hoard and with the bump allocator). The transaction that calls `mmap` restarts frequently in this benchmark. Without TxOS, these additional calls to `mmap` leak a large number of memory pages when the user loses a conflict and rolls back. With TxOS, the `mmap` is made part of a system transaction and it is properly rolled back when the user-level transaction aborts.

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

The possibility of an `mmap` leak is a known problem [45], with several solutions including open nesting or a transactional pause instruction. All solutions complicate the programming model, the hardware, or both. System transactions address the the memory leak with the simplest hardware model and user API.

5.5 STM with system calls

In this section, we evaluate the integration of a Java-based software transactional memory system (DATM-J [3]) with system transactions. We extend DATM-J to use the system call API provided by TxOS. The only modification to the STM is to follow the commit protocol when committing a user level transaction that invokes a system call, as outlined

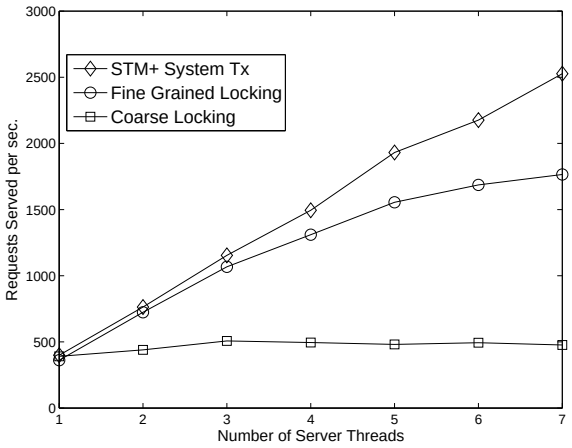

Figure 7: Requests served by the Tornado server when using DATM-J with system transactions versus locks.

in Section 3.4.

We use Tornado, a multi-threaded web server that is 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 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 acquires a read-write lock on the directory that contains these files (a read-write lock allows concurrent readers). A single lock is simple but restricts concurrency and hinders scalability. Fine-grained locking performs better by using a read-write lock for each file. Multiple locks are more complex and require lock ordering to avoid deadlock. System transactions provide good performance without the subtleties of fine-grained locking. Current STMs cannot be used for this critical region because it makes system calls 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. Each file is about 10KB in size. Figure 7 shows

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%

Figure 8: Execution time in processor cycles of common system calls on unmodified Linux and on TxOS within a transaction.

that the STM version scales better than the locks as the number of server threads increase. Coarse grained locking does not scale at all.

Transactions scale better than locks because in this read dominated workload they allow more threads to concurrently access the file system. The STM version performs up to $4.5\times$ better than coarse grained locking and up to 47% better than fine-grained locking. As the number of server threads increase, the STM version incurs more aborts due to contention. For example, the total number of aborts rise from 6 (0.8% of total transactions) to 492 (18.6% of total transactions) as the number of threads increase from 2 to 7. Of these the transaction aborts due to system transaction conflicts are 5 and 49 respectively. All these aborts represent points where concurrent accesses must be serialized by the runtime system.

5.6 Transaction Overhead

Table 8 shows the average execution times of common file system operations, both alone and within a transaction. 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` mostly reflects the fact that the implementation has not been tuned.

A key performance concern for TxOS is the performance overhead of detecting asymmetric conflicts that is imposed upon non-conflicting, non-transactional applications. We measured the performance overhead of these checks on a benchmark that searches `/etc` (containing 1,887 files and 8.9MB data) for a string that it does not find. 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 [42] and QuickSilver [16, 37] are historical systems that provide some system support for transactions. The primary goal of these systems is committing file writes atomically with distributed transactions. However, to get good performance, 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 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, though it allows 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.

Quicksilver does not support strong isolation (Section 3.3), and hence does not 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 transaction aborts. The current STM literature shows a number of situations that lead to data structure corruption when strong isolation is not provided [26, 38]. 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 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.

Transactional file systems. Microsoft Windows Vista contains TxF [32], a transactional file system. Transactional file systems have been suggested for Linux, e.g., Amino [43]. Unlike journaling file systems, these systems allow the grouping of multiple operations on files into transactions. Transactional file systems should be able to eliminate TOCTTOU races. Amino, however, is build on top of a database, which presupposes a conventional file system that lies outside of the transaction system. The OS boots on a traditional file system that also stores the database containing 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 the 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 can, for example, write a log record and send a signal atomically. Such a facility is not possible with a transactional file system. Ultimately, TxOS could integrate transactions in the VFS layer with a transactional file system.

Distributed transactions. A number of systems, including TABS [39], Argus [23], and Sinfonia [2], provide support for distributed transactional applications at the language or library level. These papers make important contributions to developing the transactional programming model. However, because transactions are implemented at user level, they cannot isolate system resources, whereas TxOS can.

Speculator. Speculator applies an isolation and rollback mechanism to the operating system that is very similar to transactions. This mechanism allows their system to speculate past high-latency remote file system operations [30]. 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. TxLinux is a linux kernel that uses hardware transactional memory (HTM) as a synchronization technique within the Linux kernel [35, 36]. TxOS is orthogonal to TxLinux. 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 hardware transactional memory mechanisms.

Fault Recovery. Automated fault recovery systems such as Rx [33] provide a mechanism for check-pointing and rolling back application state after a failure that is similar to transactional memory. These systems are designed to be lightweight in the common case where there are no failures and do not provide a user interface for isolating non-faulting actions as TxOS does.

6.1 Future work

This paper has several examples of 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, such as 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 allow these round-trip communications within transactions [37, 39], requiring that the other end 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.

7 Conclusion

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

References

[1] 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.

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

[3] Anonymized. Scalable software transactional memory. Technical report, http://www.nevercomingdown.com/datmj.pdf, 2008.

[4] E. D. Berger, K. S. McKinley, R. D. Blumofe, and P. R. Wilson. Hoard: a scalable memory allocator for multithreaded applications. In *ASPLOS*, 2000.

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

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

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

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

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

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

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

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

[13] U. Drepper. Secure file descriptor handling. In *LiveJournal*, 2008.

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

[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*, 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*, 2008.

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

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

[20] M. P. Herlihy and J. M. Wing. Linearizability: a correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, 12(3), 1990.

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

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

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

[24] P. Magnusson, M. Christianson, and J. E. et al. Simics: A full system simulation platform. In *IEEE Computer vol.35 no.2*, Feb 2002.

[25] 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.

[26] 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.

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

[28] 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.

[29] J. E. B. Moss. *Nested Transactions: An Approach to Reliable Distributed Computing*. PhD thesis, 1981.

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

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

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

[33] 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.

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

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

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

[38] 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*, 2007.

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

[40] 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.

[41] J. Wei and C. Pu. TOCTTOU vulnerabilities in unix-style file systems: An anatomical study. In *FAST*, 2005.

[42] 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.

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

[44] L. Yen, J. Bobba, M. Marty, K. E. Moore, H. Volos, M. D. Hill, M. M. Swift, and D. A. Wood. Logtm-SE: Decoupling hardware transactional memory from caches. In *HPCA*. Feb 2007.

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