Improving Server Applications with System Transactions

Sangman Kim* Michael Z. Lee* Alan M. Dunn* Owen S. Hofmann*
Xuan Wang† Emmett Witchel* Donald E. Porter†

*The University of Texas at Austin †Stony Brook University

{sangmank,mzlee,adunn,osh,witchel}@cs.utexas.edu {wang9,porter}@cs.stonybrook.edu

Abstract

Server applications must process requests as quickly as possible. Because some requests depend on earlier requests, there is often a tension between increasing throughput and maintaining the proper semantics for dependent requests. Operating system transactions make it easier to write reliable, high-throughput server applications because they allow the application to execute non-interfering requests in parallel, even if the requests operate on OS state, such as file data.

By changing less than 200 lines of application code, we improve performance of a replicated Byzantine Fault Tolerant (BFT) system by up to 88% using server-side speculation, and we improve concurrent performance up to 80% for an IMAP email server by changing only 40 lines. Achieving these results requires substantial enhancements to system transactions, including the ability to pause and resume transactions, and an API to commit transactions in a pre-defined order.

Categories and Subject Descriptors D.1.3 [*Programming Techniques*]: Concurrent Programming; D.4.7 [*Operating Systems*]: Organization and Design

General Terms Design, Performance, Security

Keywords System Transactions

1. Introduction

Server applications must process requests as quickly as possible. Because some requests depend on earlier requests, there is often a tension between increasing throughput and maintaining the proper semantics for dependent requests. For example, if a user is modifying messages in his mailbox, the email server might delay the delivery of a new message; this strategy reduces throughput to avoid race conditions that could result in lost user modifications.

Server applications often use operating system (OS) services, such as the file system and inter-process communication (IPC). But once application data migrates to the OS, the application loses the ability to balance tradeoffs between semantics and concurrent throughput. The POSIX API provides few options for managing concurrency.

Operating system transactions [34], or system transactions, make it easier to write reliable, high-throughput server applications because applications can transactionally execute non-interfering operations in parallel, even if the operations include OS state like file data. Transactions are useful because they detect interfering requests dynamically, based on the actual data access patterns of a request. For most realistic servers, the request dependencies are highly data dependent and can only be determined at runtime.

OS transactions are also valuable to distributed systems because they provide a convenient path to deterministic parallel execution. Some distributed systems, like Byzantine Fault Tolerant (BFT) replication systems, rely on deterministic execution for correctness, and thus must sacrifice performance and simplicity to ensure determinism. By executing requests in transactions with a specified commit order, developers of these replicated servers can easily implement deterministic parallel request execution.

This paper demonstrates that operating system transactions can improve server applications with few modifications to those applications. To demonstrate this principle, we modify the UpRight BFT library [12] and the Dovecot IMAP server [2], changing less than 200 lines of code to adopt system transactions. The applications require few changes because transactions are easy to use. We increase throughput for UpRight by up to 88%, and improve Dovecot throughput by up to 80% while eliminating mail delivery anomalies.

During the execution phase of a BFT system, multiple ordered requests induce state changes on the replicated data. Parallelizing this execution phase has been a long-standing challenge for BFT systems. This challenge arises because of the complexity of detecting dependencies between requests and rolling back server state on misspeculation (see Section 3 for more discussion of previous systems). Most BFT systems execute requests serially to ensure consistent replication. Serial execution throttles performance on increasingly common multi-core systems by preventing parallel execution of independent requests. Even if an application developer understands the application-level semantics enough to manually parallelize the code, reasoning about state that leaks into the OS via system calls (e.g., files, IPC) is intractable without better OS support.

EuroSys'12, April 10–13, 2012, Bern, Switzerland.

The Internet Message Access Protocol (IMAP) is a widely used protocol for accessing email messages [13]. IMAP is supposed to support concurrent clients accessing an inbox, such as a user accessing email on her phone and laptop at the same time. Unfortunately, OS API limitations cause disturbing artifacts in many IMAP implementations, such as temporarily lost messages during concurrent use. IMAP stores the definitive versions of email messages in folders on a server, and an email client acts as a cache of these emails. The `maildir` server storage format for IMAP is designed to be lock-free [6]. Unfortunately, the POSIX API is insufficient for an IMAP server to guarantee repeatable reads of a `maildir` inbox, so IMAP implementations, such as Dovecot, have reintroduced locks on back-end storage [1]. When server threads terminate unexpectedly without releasing locks, for instance due to a bad client interaction, users experience unpredictable email behavior; email may be lost or messages may not be able to be marked as read. System transactions can guarantee repeatable reads without locks, providing strong semantics, more reliable behavior, and higher performance for concurrent clients.

This paper describes the largest workloads for a transactional OS reported to date. Supporting applications of this magnitude requires substantial enhancements to the recent TxOS design [34], including more careful interfaces for composition of synchronization primitives, separation of application transactions from internal JVM or `libc` state, and a number of design and implementation refinements. Fortunately, the transactional interface remains simple and easy to use for the application-level programmer, and system-level software (such as `libc`) requires only a small number of simple modifications, which are encapsulated from the application programmer.

The contributions of this paper follow.

1. The design and implementation of operating system transactions sufficient to support large, distributed applications written in managed languages like Java (§5). This work requires substantial extensions and improvements over the recently published TxOS design [34].
2. The design and implementation of Byzantine fault tolerant replication that allows the server to speculate on the order of requests (§3).
3. The design and implementation of a Dovecot IMAP server that features true lockless operation without behavioral anomalies and high performance for write-intensive workloads. (§4).
4. A careful study of how to compose transactions with other application synchronization primitives, including `futex`-based locks (§6).

2. Operating system transactions

This section reviews operating system transactions, describing the properties most relevant to supporting distributed systems. We refer the reader to previous work for a more complete description of system transactions [33, 34]. Our implementation is derived from the publicly available TxOS code version 1.01[1]. We call our modified system TxOS+.

System transactions group accesses to OS resources via system calls into logical units that execute with atomicity, consistency, isolation, and durability (ACID). System transactions are easy to use: programmers enclose code regions within the `sys_xbegin()` and `sys_xend()` system calls to express consistency constraints to the OS. The user can abort an in-progress transaction with `sys_xabort()`. Placing system calls within a transaction alters the semantics of when and how their results are published to the rest of the system. Outside of a transaction, actions on system resources are visible as soon as the relevant internal kernel locks are released. Within a transaction, all accesses are isolated until the transaction commits, when changes are atomically published to the rest of the system. System transactions provide a simple and powerful way for applications to express consistency requirements for concurrent operations to the OS.

System transactions provide ACID semantics for updates to OS resources, such as files, pipes, and signals. System transactions are serializable and recoverable. Only committed data are read, and reads are repeatable; this corresponds to the highest database isolation level (level 3 [19]). We call a kernel thread executing a system transaction a transactional thread. A transactional system call is a system call made by a transactional thread.

Conflicts. To ensure isolation, a kernel object may only have one writer at a time, excepting containers, which allow multiple writers to disjoint entries. Two concurrent system transactions **conflict** if both access the same kernel object and at least one of them is a write. The kernel detects and arbitrates conflicts. The arbitration logic might abort one of the conflicting transactions or it might put one of the transactions to sleep until the other commits. The latter policy is often called stall-on-conflict, and resembles condition synchronization. where the condition is a transaction commit.

Transactions and non-transactional system calls serialized. TxOS ensures serializable execution among transactions and between transactions and non-transactional system calls (sometimes called strong isolation [7]). Isolating transactions from non-transactional system calls simplifies the programming model. The programmer can think of each individual system call as its own mini-transaction. For example, if one thread is enumerating a directory's contents in a system transaction while another thread does a rename in that directory, the contents listing will contain only the old or the new name, never both. Note that in Linux, the concurrency guarantees for system calls vary: `rename` is atomic, whereas concurrent `read` and `write` calls can return partially interleaved results. When transactions are not

[1] http://code.csres.utexas.edu/projects/txos

involved, TxOS provides the same concurrency guarantees as Linux.

Managing user memory. System transactions primarily provide ACID semantics for system state; the application programmer has several options for managing state in the application's address space. For single-threaded applications, an option to `sys_xbegin()` will make the address space copy-on-write, which allows the OS to revert the application's memory state if the transaction aborts. However, for multi-threaded programs, the OS cannot and does not manage user state. For multi-threaded Java code, pages mapped in response to a transactional allocation request stay mapped on an abort, but any allocated objects are garbage collected by the JVM.

System vs. memory transactions. System transactions are a transactional model for operating system objects. Transactional memory [21, 41] is a transactional model for updating memory objects. Database transactions [19] are a transactional model for a data store. All of these systems are independent realizations of a simple interface for the same ideas: version management and conflict detection. In particular, system transactions do not require transactional memory (though system transactions and transactional memory can interact symbiotically [34]). System transactions are implemented on current, commodity hardware.

Incremental adoption. Concurrency problems are often localized to particular sections of code, and system transactions can be applied to those sections without large-scale redesign. Table 2 shows the magnitude of source code changes (less than 200 lines) to add system transactions to substantial code bases.

3. Replication and Byzantine Fault Tolerance

Distributed systems often replicate services in order to tolerate faults. If one replica behaves incorrectly, others can mask the incorrect behavior until the replica is corrected.

The canonical approach to replicating a service is to design the server as a replicated state machine [38]. The key principle behind replicated state machines is that if the same set of inputs are presented to each replica in the same order, each deterministic replica will produce the same outputs and have the same internal state—yielding identical replicas. Note that inputs in this model include inputs both from a client connected over the network and from the OS, such as a file's contents or the output of a random number generator. However, the OS API is insufficient even to prevent race conditions, including in the file system [10] and signal handlers [49]; concurrent, deterministic execution of arbitrary system calls is impossible.

We examine a Byzantine fault tolerant system, in which a faulty replica can exhibit arbitrary and possibly malicious behavior [26]. The role system transactions play is applicable to any failure model in replicated systems, and is not specific to BFT.

3.1 BFT summary

Byzantine Fault Tolerance (BFT) is a replicated systems framework, which can handle a wide range of faults, including crashes and deliberate misbehavior of a bounded number of faulty servers. When the total number of agreement replicas is n, a typical BFT system based on replicated state machine can tolerate up to $(n-1)/3$ faults, more commonly written as $3f + 1$, where f is the number of faults [8, 25]. Note that this bound is for agreement replicas, and the bound for execution replicas can be as low as $2f + 1$, depending on the protocol [48].

Replicated state machine-based BFT protocols generally process requests in a pipeline consisting of authentication, ordering, and execution phases [12, 48]. When a request arrives from a client, it is first authenticated to ensure that it came from a legitimate client and is well-formed. The ordering phase then imposes a global order on each accepted request. Replicas can execute requests after ordering completes, or speculatively before the ordering [23]. Their responses are generally checked by the client; responses that diverge indicate existence of a faulty node or misspeculation. Each of these phases must be performed by a quorum of replicas (the precise minimum varies by implementation).

3.2 Parallelizing BFT execution with transactions

A key drawback of many BFT systems is their inability to execute requests in parallel. Because multi-threading introduces non-determinism, BFT servers generally cannot leverage the increasingly abundant parallelism of commodity multi-core hardware in the execution phase of the system. This decreases throughput and increases request processing latency. System transactions can provide deterministic, optimistic concurrency to speed up the execution phase. Speeding up execution also speeds up recovery, where one replica must catch up with its peers.

Optimistic parallel request execution for BFT systems has seen limited use because of the complexity of detecting dependencies between requests and rolling back server state on misspeculation. Replicated databases like HRDB [45] or Byzantium [18] rely on transaction support from off-the-shelf database systems to tolerate Byzantine faults inside databases. However, this approach cannot be extended to most other applications, as they do not have a built-in transaction facility. If such applications need BFT, parallelization of replica execution requires very intimate application-specific knowledge about requests and their execution [24], which is infeasible for complex systems. Any attempt to parallelize execution without system-wide concurrency isolation will likely introduce divergence due to non-determinism, which can degrade throughput significantly [42].

Recent research OSes allow deterministic parallel execution [4, 5], but their overheads are substantial. BFT does not actually require a deterministic OS; consistent states on each replica are sufficient, which can be efficiently provided by transactions serialized according to the same schedule.

Parallel execution in replication-based fault tolerant systems also accelerates their recovery. When a failed replica is replaced with a new correct node (or the same node repaired), the new replica's state must be brought up to date with the other replicas. The typical strategy for recovery is for replicas to take periodic checkpoints of their state and to keep a log of subsequent requests. The new replica fetches the most recent checkpoint and replays the log until it catches up. During recovery, requests are processed sequentially, and new requests are buffered. Assuming that some requests can execute safely and concurrently (which is common), optimistic execution can accelerate recovery.

4. IMAP email server

The Internet Message Access Protocol (IMAP) is a widely used protocol for accessing email messages [13]. IMAP stores the definitive versions of email messages in folders on a server, and an email client acts as a cache of these emails. IMAP provides features missing from the previous Post Office Protocol [29], including seamless offline email operations, which are later synchronized with the server, and concurrent email clients (e.g., a laptop, desktop, and smartphone can simultaneously connect to an inbox).

A key feature advertised by IMAP is concurrent access by multiple clients, yet the protocol specifies very little about how the server should behave in the presence of concurrency. There is no protocol-level guarantee about what happens if two clients simultaneously modify a message or folder. If something goes wrong while moving a message to a subfolder, the outcome depends on the client and server implementations: the message can be lost or duplicated.

Allowing a wide range of IMAP client and server implementations to dictate ad hoc concurrency semantics leads to practical challenges. For instance, if a user leaves a mail client running at home that aggressively checks for new email messages, the implementation-specific locking behavior on the server may deny the user's client at work the ability to delete, move, or mark new messages as read. This denial of service can result from either aggressive polling by the client (i.e., lock fairness), or "orphaned" file locks from an improper error handling of a client request. This erratic behavior stems from the limitations of the underlying OS API with respect to concurrency and durability.

4.1 Backend storage formats and concurrency

Although the specific storage formats can vary across IMAP server implementations, there are two widely-used storage formats: mbox and maildir. The storage format dictates much of the concurrent behavior of the server; servers that support both backends will behave differently with each.

mbox. The mbox format stores an entire email folder as a single file. Most mbox implementations also have a single file lock, which serializes all accesses to a particular mail folder. If a server thread fails to release a file lock on a user's inbox, perhaps because it received a malformed client message, *all* clients can be locked out of the mailbox until the lock is manually cleaned up by an administrator.

maildir. The maildir format [6] was created to alleviate the issues with stale locks in mbox. Maildir is designed to be lock-free. Maildir represents a mail folder as a directory on a file system, and each message as a file. Mail flags and other metadata are encoded in the file name.

Although maildir is lock-free, the design does not provide repeatable reads for a user's inbox. Reading a user's inbox is typically implemented by a series of `readdir` system calls, which get the names of each file in the inbox, and a series of `stat` or `open`/`read` system calls, which extract other metadata about the message. If another client is concurrently marking a message as read (by `rename`-ing the file to change its flags), the first client cannot distinguish the change in flags from a deletion, leading to disturbing artifacts such as lost messages. The lack of repeatable reads in maildir led a major IMAP implementation, Dovecot, to reintroduce file locking for its maildir backend [1].

File locking in both storage formats introduces substantial portability issues, as Unix systems have multiple, mutually incompatible file locking regimes, including `flock` and `fcntl` locks. The system administrator must understand the low-level details of the OS and file system in configuring the mail server. Moreover, this proliferation of locking mechanisms diffuses bug fixing effort in the kernel, increasing the likelihood of bugs in specific locking mechanisms. For instance, a Dovecot bug report indicates that using `flock` on Linux triggered a race condition that was eliminated by switching to `fcntl` locks; no bug in the Dovecot source was identified to explain the problem [15].

Because Dovecot's locking scheme is non-standard, a mail client that accesses the backend storage directly, such as `pine`, could still introduce similar anomalies *even if locks are enabled*. This is because lock files are a cooperative primitive—if any application doesn't cooperate, the OS cannot prevent these race conditions.

4.2 Opportunities for system transactions

System transactions should provide a range of benefits to IMAP implementations: eliminating anomalous behavior like freezes and non-repeatable reads, simplifying administration by eliminating lock files, and increasing performance under contention. System transactions give IMAP developers a better interface for managing system-level concurrency.

At a high level, TxOS+ and lock files give the same safety guarantees for Dovecot—the appearance of serial execution.

Dovecot on TxOS+ gives these guarantees with generally higher concurrent performance, as it can execute safe requests concurrently. In contrast, locking always executes requests serially, whether they need to be or not. Strictly speaking, TxOS+ actually gives slightly stronger guarantees than locking: even if a client that does not cooperate in the directory locking protocol tries to access these back-end files non-transactionally, the OS will ensure that Dovecot has repeatable mailbox reads.

Stale lock files left by a server thread that terminates abruptly can prevent other clients from updating their view of a mail folder. System transactions obviate the need for lock files. System transactions allow a consistent read of a directory, even if files in that directory are being added or renamed. System transactions also provide repeatable reads, as the transactions are serializable[2]. Repeatable reads eliminate the disturbing artifacts where messages disappear or spontaneously change state. Finally, because there is no need to lock the mail folder, updates and reads may proceed concurrently, increasing throughput under load.

5. Improving system transactions

This section explains the shortcomings present in the initial TxOS system transaction model and how we addressed them to support large, server applications in TxOS+. The guiding principle for TxOS+ is to keep the application programming interface for transactions as simple as possible.

Many of the challenges that motivated these changes arose from composing transactions with large bodies of middleware code, such as the Java Virtual Machine (JVM) and `libc`. For instance, the JVM issues system calls on behalf of the application, as well as those for its own internal bookkeeping; automatically rolling back part of the JVM's internal state on an application-level transaction abort can corrupt the JVM. Similarly, we faced challenges in composing application-level transactions with synchronization mechanisms encapsulated in middleware code. We extend TxOS+ to address these and other challenges, and demonstrate that the vast majority of this complexity can be sequestered in expert-written, middleware code at a small number of locations, maintaining simplicity at the application-level interface. We emphasize that this complexity arises not from programming with transactions *per se*, but from adapting a large body of already complex code, written with no concept of transactions, to use transactions.

5.1 Managing middleware state

We initially expected user-initiated system transactions to naturally isolate middleware state. For example, if a Java program starts a transaction, then reads from an object bound to a file, it is the JVM that actually issues the `read` system call, which should be isolated as part of the transaction.

[2] Serializability is also known as degree 3 isolation [20], which guarantees repeatable reads.

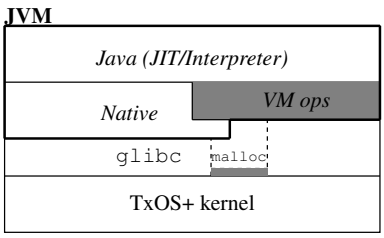

Figure 1. Block diagram of the Hotspot JVM on TxOS+. Shaded regions are where system transactions are paused, including JVM-internal operations and when `malloc` adds pages to a heap.

Some middleware state, however, should not be isolated by a user-initiated system transaction. For instance, if the JVM happens to add pages to a memory allocator while running a transactional thread, portions of this page can be safely allocated to other threads without compromising transactional isolation. Moreover, if the transaction fails, simply unmapping this page is the wrong undo action, as it will cause memory faults in threads uninvolved with the failed transaction. To address this issue, we extended the original TxOS nesting model.

5.2 Nesting model

The **nesting model** for a transaction system defines the semantics of what happens if a transaction executes inside of another. If function A begins a transaction and calls function B, which also begins a transaction, the two transactions must interact in some clearly defined way. The original TxOS nesting model is simple flat nesting. Any inner transaction (e.g., the one started by B) becomes a part of the enclosing transaction (e.g., the one started by A). The inner transaction can see all in-progress updates, and even after it commits, the inner transaction's state (B) is not visible to other threads until the enclosing transaction commits (A). Flat nesting is sufficient to compose many transactions, and its behavior closely matches the naive programmer's intuition.

We extended the nesting model in TxOS+ to allow the JVM to differentiate user-initiated action and middleware-initiated action. System calls issued directly for application operations become part of a system transaction, whereas the JVM or `libc` can *pause* the transaction before issuing system for internal operations. Figure 1 shows a block diagram for a JVM running on TxOS+. A thread can be executing user code, (the Java box), native support code called directly from the Java code, (the Native box), glibc code, (`glibc` box), or kernel code in response to a system call (TxOS+ kernel box). In addition, the JVM sometimes needs to perform operations on behalf of all threads (the VM ops box).

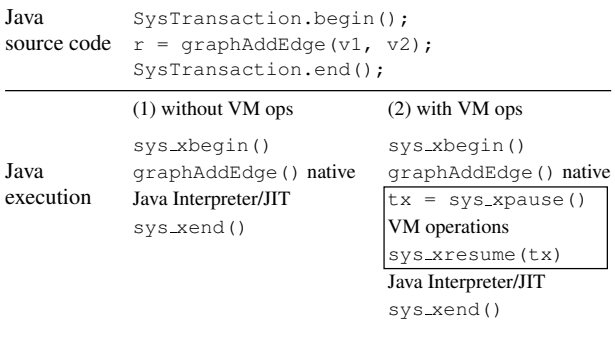

Figure 2. A system transactions code example depicting Java source code, and execution inside the JVM. Any active system transaction is paused during VM operations.

5.3 Designing pause/resume for TxOS+

TxOS+ effectively solves this middleware state problem by providing a transactional **pause** and **resume** primitive. When a transaction is paused, the OS treats the thread as if it were running non-transactionally: any system calls the thread issues will be visible to other threads immediately, the thread can conflict with the parent transaction, and the thread can start a new, independent transaction.

We modified the JVM and `glibc` to pause the running transaction before performing internal bookkeeping, such as adding or removing pages to the heap or updating VM bookkeeping, and to resume the transaction afterwards. We discuss the uses of transactional pause in Section 7.1.

The `sys_xpause` system call returns a unique identifier for a transaction, which is passed as an argument to `sys_xresume`. The kernel tracks the association of a transaction with the participating task(s) for security and garbage collection. A thread may only resume a transaction it previously participated in. When the last task with a reference to a paused, uncommitted transaction exits, the transaction is aborted, rolled back, and freed. Pause and resume are only intended for use in system software, they are not intended for use in applications.

Figure 2 shows an example of how Java source code that creates a system transaction is executed. The JVM adds the system calls to start the transaction, but it also pauses the transaction if necessary to perform internal VM bookkeeping. Many JVMs already save time and space by "borrowing" an already existing thread for JVM bookkeeping, rather than creating a dedicated internal thread; the pause and resume primitives make this safe.

It is a reasonable concern that a more complex nesting model erodes the simplicity benefits of system transactions. Novice programmers may struggle to reason about the interactions of paused and transactional system calls. Our experience is that pause and resume are only required for the JVM and `libc`, which are maintained by experts. All refactored application code, even the 138,000 line IMAP server, only uses the simple transaction calls and flat nesting.

5.4 Ordered transactions

Historically, BFT replicas have serially executed requests that issue system calls. Replicas must be deterministic; in sowing system-level concurrency, one often reaps nondeterminism. Using **ordered transactions**, TxOS can execute requests in parallel, but commit them using a predefined order. For low-contention workloads, ordered transactions should provide performance from parallelism while preserving safety for BFT replicas.

In the original TxOS implementation, OS transactions were serialized according to *some* schedule. The schedule is a result of a configurable, kernel-wide policy that implements performance or fairness heuristics, such as the oldest or highest priority transaction wins a conflict.

We augment these kernel-wide policies with the ability for applications to prescribe the serialized order of their transactions with integer sequence numbers. Transaction ordering is enforced at the granularity of a process group: multi-threaded or multi-process applications can use this feature, but unrelated applications cannot use this mechanism to interfere with each other.

We add an optional `sequence_number` argument and two flags to `sys_xbegin`—one which specifies that the transaction is in an ordered sequence, and one which resets the sequence counter (for loading application checkpoints). The ability to reset the sequence numbers means that sequence numbers are purely a cooperative primitive, they will not constrain arbitrary or malicious code (other work [22] uses system transactions to sandbox potentially malicious code). The sequence counter for a process group is always initialized to zero. Applications that use ordered transactions may need to detach from the current process group to prevent interference from another application using ordered transactions.

Ordered commit is an interface change for transactions. The programmer must decide if *any* commit order is sufficient, or if a specific one is necessary. The nature of the programming problem should make it clear which is needed, for example, deterministic re-execution requires ordered commit, while processing independent requests does not.

6. Composing synchronization mechanisms

Composing synchronization primitives, even of the same type, is a historically fraught problem. In the Linux kernel, for instance, acquiring a blocking lock (e.g., a semaphore or mutex) while holding a spinlock can deadlock the entire system. Transactions are an easy to use synchronization primitive because they guarantee serial behavior and because they compose easily with each other. However, when transactions interact with other synchronization primitives they can inherit the other primitives' complexity.

As we add system transactions to larger code bases, the transactions interact with the application's existing synchronization. Although some mechanisms can be subsumed by

transactions, some serve an independent purpose and must compose with transactions, such as locks on state in `libc`. After puzzling over different interactions, we decided to investigate how to compose all existing kernel-visible Linux synchronization primitives with system transactions.

We identify a set of "best principles" that prevent unintended synchronization errors in conscientiously written applications. These principles are derived from both experience with larger application development on TxOS+, as well as a set of small, carefully-written test cases, summarized in Table 1. In writing these test cases, we set out with the goal of both capturing behavior we had seen in practice, and exploring other common design patterns.

6.1 Futexes

First we describe how system transactions interact with the Linux fast userspace mutex, or `futex` [16]. The `futex` system call provides a wait queue in the kernel, which can be used by blocking application-level primitives, such as a mutex or semaphore. Consider a mutex that is implemented using atomic instructions on a lock variable in the application's address space. Rather than spin and waste CPU cycles, threads that fail to acquire the lock will call `futex` to wait for the lock to be released. When a thread releases the lock, it must also call `futex` with different arguments to wake up one or more of the blocked threads.

In futex-based locks, waiting threads are blocked on a kernel-visible queue, but the kernel cannot infer which thread actually has the user-level lock. The state of the user-level lock is completely contained within the application's address space. When a thread is blocked on a futex, the kernel knows that it does not hold the corresponding user-level lock. That is all the kernel knows for sure. To avoid deadlock, TxOS puts transactional threads that block on a futex in "deferential mode" until they wake up. In deferential mode, a transaction takes the lowest possible priority in all conflict arbitration. Therefore, the waiting thread will lose any transactional conflict with the thread that holds the user-level lock, assuring system progress.

Moreover, upon entering deferential mode, the transaction wakes up any threads that were waiting on it to commit. As explained in Section 2, when a transaction (or non-transactional system call) loses a conflict, it generally waits on the winning transaction to commit before retrying (or restarting). When a transaction is blocked on a futex, it wakes up any waiting transactions and allows them to retry early—just in case one of them has the user-level lock.

6.2 Principles for composing transactions with other synchronization

Avoid circular wait. A number of OS primitives, including pipes, sockets, and futexes, can block waiting on an action taken by a different process. Similarly, lower-priority transactions can stall on a conflict (§2), thus blocking while a higher-priority, conflicting transaction completes. These blocking primitives can deadlock each other. For instance, transaction A can block on pipe input after stalling process B on an unrelated conflict (e.g., a write to another file), yet B is the process responsible for providing the pipe input. The `pipe` test in the table represents this case; the `goodfutex` test exhibits a similar pattern.

TxOS+ avoids deadlocks where the deadlocking primitives are all OS-visible abstractions. Instead of doing costly cycle-detection, the OS simply prevents any tasks from waiting on a transaction to commit if that transaction is itself waiting. This policy prevents circular wait, a necessary precondition for deadlock.

Well-formed nesting and non-blocking transactions. To successfully mix application-level locks and transactions, the lock must be both acquired and released either outside a system transaction, or within it, but not both. Synchronization succeeds when one primitive is consistently nested in the other. If nested outside of a transaction, the application lock simply provides mutual exclusion to the transaction; if nested inside, even a failed transaction will still release the lock. Acquiring a lock and holding it while beginning a transaction requires a special non-blocking flag to `sys_xbegin` to prevent the transaction from blocking while holding a lock.

Moreover, the nesting hierarchy must be strict—one thread should not acquire and release a lock inside of a transaction and another thread nest a transaction inside of a lock acquire and release. If thread A cannot commit its transaction until it acquires a lock, and thread B cannot release the lock until thread A commits, the threads can deadlock. "Deferential" futexes cannot prevent this problem in all cases, and only convert it from a deadlock to a livelock. In the example above, deferential mode will abort thread A's transaction if it conflicts with B while waiting on a futex; however, deferential futexes cannot prevent thread A from restarting its transaction and repeatedly conflicting with thread B on other data. This cycle of repeated conflicts between A and B can prevent thread B from ever committing and releasing the lock. In this case the lack of a well-formed nesting discipline in the presence of conflicting transactions creates a race condition that can livelock. The `badfutex` test experiences this livelock.

Explicit ordering for well-structured, deterministic synchronization. When using well-structured or deterministic synchronization paradigms, such as an internal producer/-consumer, explicit transaction ordering communicates the programmer's intention to the OS, which helps the OS make better scheduling decisions. The `ordered-p/c` test simulates such an application-level producer/consumer paradigm, using ordered transactions combined with a local semaphore to ensure that consumers follow producers for variables stored in a local array.

In general, one should not mix deterministic and non-deterministic primitives, as a non-deterministic primitive

Test	Description	Comments
goodfutex	Use a raw `futex` call to simulate blocking on one futex inside of a transaction and a second outside of a transaction.	Works with transactions, as it follows a well-formed nesting discipline.
badfutex	Combine a `futex` call with conflicting transactions to simulate acquiring the same lock inside of a transaction and outside of a transaction.	Livelocks because it violates a well-formed nesting discipline.
orderedfutex	Combine a `futex` call with ordered transactions to simulate mixing non-deterministic and deterministic synchronization primitives.	Hangs because of a fundamental determinism mismatch.
mutex	Use a `pthread` mutex lock inside and outside of a transaction, similar to goodfutex.	Works with transactions, as it follows a well-formed nesting discipline.
ordered-p/c	Use ordered transactions and `pthread` semaphores to coordinate access to a local array; consumers write outputs to a file in order.	Transaction ordering is sufficient to coordinate the producers and consumers.
pipe-p/c	Use ordered transactions and an OS-level pipe to implement a producer/consumer pattern; consumers write outputs to an file in order.	Transaction ordering is sufficient to coordinate the producers and consumers.
lockfile	Wait on a lockfile to be created inside of a transaction inside of a polling loop.	Can deadlock with file creator; use scheduling priority to defer to the file creator.
pipe	A pipe reader conflicts with the pipe writer on a file access.	Deadlock is prevented by lowering the reader's priority while waiting and waking up any transactions blocked on the reader.

Table 1. Summary of test cases for composing transactions and other synchronization primitives.

permits many schedules that the deterministic primitive will not. The `orderedfutex` test hangs because it mixes deterministic and non-deterministic synchronization.

Scheduling priority for unstructured synchronization. When using ad hoc synchronization, application developers may not be able to specify an ordering. For example, the `lockfile` test waits inside of a transactional polling loop for a new file to be created by another task that does not cooperatively order transactions. If the waiting transaction has enumerated the directory contents, degree 3 isolation should prevent it from seeing newly created entries, which TxOS enforces with read isolation on the directory's child list. When the other task attempts to create the file, it will cause a conflict; if the OS arbitrates in favor of the waiter and blocks the file creator, the application will deadlock.

We address this issue with OS scheduling priorities. The default contention management policy in TxOS uses scheduling priorities to arbitrate conflicts. Before entering a polling loop, the transaction should use `nice` or `setpriority` to defer to the process it is waiting on. In each iteration of the loop, the polling transaction can incrementally lower its priority until it defers to the creating transaction. Similarly, the file creator can set the non-blocking flag and incrementally raise its priority until the transaction completes.

7. Implementation

This section both describes our improvements to system transactions and discusses some non-obvious interactions of the system implementation with system transactions. To create TxOS+, we forked TxOS 1.0.1, which is based on the Linux 2.6.22.6 kernel. Our distribution was Ubuntu 8.10, including glibc 2.8.90, OpenJDK Icedtea6 1.9.7.

7.1 Using pause/resume in TxOS+

Pausing and resuming application-initiated transactions is vital to insulate state in the JVM or in libc. We review how pause/resume is used in TxOS+ starting with examples that have been suggested by other researchers, like bookkeeping state and debugging, then moving on to more subtle cases like dynamic linking and `mprotect` calls.

Bookkeeping state. Pause and resume are useful for managing JVM bookkeeping state (an activity we refer to as VM operations). Bookkeeping state does not need to be rolled back on a failed transaction, because the JVM does not rely on it being completely correct. For example, the JVM maintains file-backed shared memory (`perfdata`) containing the total thread count, JIT compiled methods, and the last JIT compilation failure. Updates to this data by different transactions cause spurious conflicts that would throttle transactional throughput. By pausing any active system transaction before VM operations, the code no longer causes spurious transactional conflicts.

Debugging. Another useful application of pause/resume is debugging messages. While developing an application on TxOS, program writers may want to use consoles or log files for debugging purposes. Since these channels are treated as files in many operating systems, changes to them are not written out until the transaction is committed. This kind of behavior may not be desired by program writers, especially when they want to look at user state changes in multi-threaded applications regardless of transaction commit or abort. Log messages written while a transaction is paused will be written out without delay.

Dynamic linking. C and Java link and load dynamic libraries lazily, deferring the memory mapping overheads until the library is actually used by the application. The application is not aware of when linking and loading happens, and it might occur during a transaction. When a library is loaded the linker modifies the application and library to reflect the library's placement in the application's address space.

TxOS+ should not undo any dynamic linking that occurs during a system transaction, since that would simply be wasted work. Therefore, we pause the current transaction before the dynamic linking done by libc or by the JVM.

mprotect. The JVM and libc use `mprotect` aggressively. `malloc` within `glibc` internally uses `mprotect` for minor changes in heap space size with multiple threads, making calls to `mprotect` on behalf of `malloc` more frequent than calls to `mmap`.

The JVM uses `mprotect` to implement *safepoints* where the JVM will stop threads for garbage collection and other purposes like locking bias adjustment [35], and implementing light-weight memory barriers [14]. It is possible to eliminate some uses of `mprotect`. For example, the JVM can use CPU memory barrier instructions instead of `mprotect` with the -XX:+UseMemBar option. When our JVM can avoid using `mprotect` it does, but completely eliminating the use of `mprotect` in the JVM would be too invasive a change. To avoid spurious conflicts on these frequent `mprotect` calls, we modify libc and the JVM to suspend the current transaction before the call.

brk. The `sys_brk` system call is a relic from the era of segmented virtual memory. It extends the heap portion of the data segment towards the stack. While Linux and other modern systems don't use segmentation any longer, they retain this abstraction. Many memory allocators, including the default `glibc` allocator, still use `brk` to allocate heap space, even though `mmap` could easily serve the same purpose.

Because `brk`-based allocators expect the initial heap region to be contiguous, and the address of the heap end is stored in the kernel, TxOS+ only allows one transaction to modify the process heap boundary at a time. Although extensions of the heap commute, rolling back one modification could leave holes, which would break contiguity.

One minor problem with `brk` is that glibc caches the current `brk` value, and this cache can get out of sync with the kernel's `brk` value. If glibc has an outdated `brk` value, then a subsequent `sys_brk` call could result in a non-contiguous heap. Initially, we attempted to reload the cached `brk` value after a transaction abort; we ultimately found it simpler to pause the current transaction around a `sys_brk`.

7.2 Page locking and transaction abort

If, in the course of servicing a transactional system call, the transaction cannot safely make progress, the TxOS strategy is to use a `longjmp`-like mechanism to immediately exit the system call. This mechanism was adopted because safely unwinding the stack on a transaction abort would require a massive amount of new error-handling code. The longjmp approach requires undo logging of certain operations, such as lock acquisition or temporary memory allocation. Although this approach works in general, we found it did not interact well with disk I/O.

Synchronization for pages locked for disk I/O is complicated. These pages are protected by a per-page lock, but each page also has an "up-to-date" flag that must be set once a disk read completes. Some kernel code paths may detect that this flag is not set, and spin until the disk read completes—without ever acquiring the page lock.

These two independent synchronization mechanisms interact negatively when a transaction aborts holding a page lock for I/O. In the original TxOS prototype, a transaction could begin an I/O on a page, then lose a conflict and jump out of the stack without properly handling the I/O completion. Subsequent requests for the page would hang waiting for this I/O completion, ultimately hanging the system.

We addressed this problem by preventing the transaction from jumping up the stack until it correctly releases all page locks and handles any I/O completions. We also added a debugging mode that asserted that the longjmp code is never executed with a page lock held. This debugging mode allowed us to quickly pinpoint a small number of code paths that needed explicit abort error handling code.

7.3 File descriptor table

The file descriptor table is a complex, contended kernel data structure that we redesigned for TxOS+. The file descriptor table translates file descriptors to opened file objects. The table provides an efficient means for accessing a file without doing repetitive security checks once a file is opened.

In the Linux kernel, the file descriptor table is implemented as a single array that is expanded when entries are filled. One trivial way to manage the file descriptor table in transactions is to consider the whole table as an object. This coarse-grained approach (taken in TxOS) simplifies implementation, but any open or close from two transactions create unnecessary conflicts.

We added a file descriptor object to TxOS+, and restructured the table to be a linked list of segments. The file de-

scriptor objects supports transactional semantics, and can be committed or rolled back individually. To avoid synchronizing a file descriptor table that has changed location (because it changed size), TxOS+ expands the table by linking in a new segment and leaving the old table in place. Accessing an entry can require additional indirections, but each segment allocated is twice the length of the previous segment and the system defines a limit on per-process file descriptors bounding the number of indirections needed to a small number (less than 2 indirections for all of our experiments including system configuration during boot).

7.4 Data structure reorganization

The TxOS prototype design decomposed kernel objects such as the `inode`, `dentry`, and `file` objects into two main components: a stable header for static or kernel-internal bookkeeping, and a data object for fields that could change during the course of the transaction. We found that many of the fields of the `inode` and `super_block`, specifically, were needlessly included in the data object. For instance, the `inode` data object includes several fields that do not change for the life of the inode, such as the inode number and the operations pointer. A more subtle example is the inode `stat` word, which includes some bits that encode the type (which doesn't change) and other bits that include the permissions, which can change in a transaction.

We migrated several of these static fields into the object header, which has both performance and correctness implications. In terms of performance, looking up whether an object is in the transaction's working set is relatively expensive, and adding the object to the transaction's workset is even more expensive. There were a number of cases where these checks can now be elided. Above (§7.2), we describe how to handle transaction aborts during disk I/O safely. Many of the places where a transaction could abort when requesting disk I/O came from accessing needlessly isolated fields, such the type of an inode, or the file system configuration options. This reorganization substantially reduced the amount of code that needed to be rewritten with explicit error handling.

7.5 Releasing isolation on disk reads

When part of the file system directory tree is not in the in-memory cache (`dcache`), it must be read from disk, creating new `inode` and `dentry` data structures to represent these files. If the disk read occurs during a transaction, the original TxOS prototype would add these objects to the transaction's workset. If the transaction aborts, these cached entries are discarded and re-read from disk on the next access.

In TxOS+, we immediately release isolation on these newly created data structures, as they contain only previously committed state. Though the data was read during a transaction, by releasing isolation, the system treats the files as if they were read outside of a transaction. If the transaction modifies any of the files, then TxOS+ creates transactional copies (as it does for all modified kernel objects). If the transaction fails and rolls back, data from clean disk reads are kept in the memory cache. This change eliminates costly and needless disk reads, and reduces the bookkeeping complexity needed to track this special case behavior through the life of the transaction.

Application		LOC	LOC changed	
BFT	UpRight library	22,767	174	(0.7%)
	Graph server	1,006	18	(1.8%)
Dovecot (IMAP)		138,723	40	(0.0003%)
glibc		1,027,399	826	(0.0008%)
IcedTea JVM		496,305	384	(0.0008%)

Table 2. Lines of code changed to add system transactions.

8. Evaluation

We present measurements of our prototype in this section. Table 2 shows the number of lines we changed to add system transactions to a set of large applications and libraries. The number of changed lines is very small both in absolute terms and relative to the code size of these projects. We also modified around 17,000 lines of TxOS source code to create TxOS+. The number of modified kernel lines is high because we modified structure definitons to avoid spurious conflicts.

8.1 BFT and System Transactions

Previous BFT systems were generally evaluated using services where dependencies between requests could be easily reasoned about *a priori*, such as a distributed file system (running a simplified workload) or a directory service. Many realistic workloads are not so simple. Thus, we created a new challenge workload for BFT that has opportunities for concurrency but does not lend itself to trivial dependency analysis.

Our challenge workload for fault-tolerant replication stores backend data for a network router, where nodes are machines and edges are network links. We tested this workload with two kinds of graph datasets, which have different graph density. The denser dataset contains a randomly generated graph with 5,000 vertices and 3,174,615 edges (density is 0.254), and the sparser graph has 10,900 vertices and 31,180 edges (density is 0.000525). The sparse graph is sampled from graph data from the Oregon-2 dataset [27] and is comprised of network routes between autonomous systems inferred from RouteViews [3] and other routing information. Both graphs are undirected. Graph data is stored in a file, which allows it to be larger than memory. The application accesses the data using `pread` and `pwrite` calls, and allows the OS to cache the file data.

The server can handle four types of queries from clients: edge addition, edge deletion, edge existence, and shortest

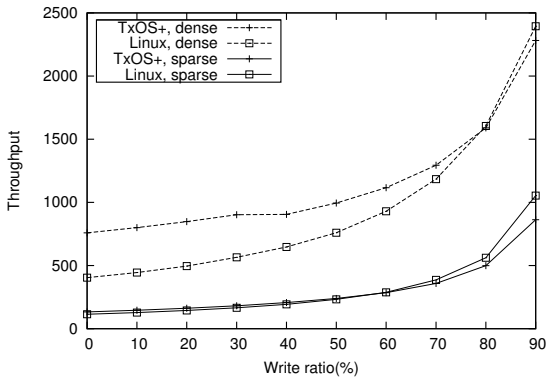

Figure 3. Throughput (higher is better) for UpRight graph server application on TxOS+ and Linux. The x-axis shows different ratios of write requests to shortest path (read) requests.

Dataset			Tput (ops/s)	Latency (ms)	Aborts
Dense graph	0% write	Linux	404.4	96.4	-
		TxOS+	759.1	50.7	0.0 %
	50% write	Linux	760.0	50.4	-
		TxOS+	994.9	40.3	0.43 %
	100% write	Linux	3601	9.49	-
		TxOS+	3360	10.2	0.19 %
Sparse graph	0% write	Linux	114.1	334	-
		TxOS+	132.9	298	0.0 %
	50% write	Linux	232.5	162	-
		TxOS+	239.8	164	3.6 %
	100% write	Linux	3638	9.41	-
		TxOS+	3340	10.0	0.34 %

Table 3. Throughput (higher is better) and latency (lower is better) for the BFT graph service on TxOS+ and Linux. TxOS+ uses server-side speculation to parallelize request processing. The percentage of aborted requests is given for TxOS+ and is highest with 50% write mix.

path between two vertices. The first two operations can modify the graph, while the other two only read it. The shortest path operation takes significantly longer than the others and includes much more data in its transaction, increasing the probability and penalty for a transaction abort (misspeculation). Each modify request first tests for the existence of the edge it seeks to modify, and does no more work if it is trying to add an existent edge or delete a non-existent edge. Dependencies between requests are data dependent and cannot be predicted before execution.

This application runs on the Upright BFT library with TxOS+, our modified JVM and our modified glibc. In our BFT application evaluation, we use 3 execution replicas with 4 threads for each. All the execution replicas run on Dell Optiplex 780 machines with Intel Core2 Q9400 quad-core 2.66 GHz CPU, 3 GB memory. Because this experiment focuses is on concurrent isolation, we use the `ext2` file system, which does not guarantee atomic disk updates, but does simplify kernel debugging. The experiment configuration also has 4 ordering replicas, 4 request quorum replicas, and 40 clients connected over a gigabit network. The clients saturate our execution servers due to the large amount of work in finding a shortest path. All experiments first warm-up the JVM at the replicas [9]. We only consider throughput for requests after the initial 20,000 requests to eliminate transient effects from system startup.

Throughput and Latency Fig 3 shows the throughput of the BFT graph server on TxOS+ with different mixes of read/write requests, compared with the throughput of the server with serial execution on Linux. With read-only requests, the parallelized BFT graph server on TxOS+ achieves 88.3% (dense) or 10.9% (sparse) higher throughput than execution on Linux. TxOS+'s throughput improvements from increased parallelism are offset by bookkeeping that TxOS+ must do to track file I/O. These overheads are lower for denser graphs, which have shorter shortest paths, and hence fewer file reads and subsequent bookkeeping.

Table 3 provides a cross section view of Figure 3, illustrating noteworthy characteristics of the workload. First, write operations are significantly faster than read operations (10× with dense graphs, 35.5× with sparse graphs), as shown by comparing the latencies of the 0% write and 100% write rows. This is because a read operation actually computes a shortest path—reading many edges and performing a substantial computation. Second, shortest path operations on sparse graphs have a larger read set than those on dense graphs. This larger read set causes an order of magnitude more aborts in the sparse 50% write workload (0.4% to 3.6%). Even with substantially more aborts, TxOS+ shows comparable performance to Linux. Finally, the 100% write test has similar latency in both graphs because each write operation simply updates a given vertex. The 100% write workload on TxOS+ has a 7–9% reduction in throughput compared to Linux, which is attributable to the very short write requests (9.4–10.2 ms) not amortizing thread coordination costs. Recall that the Linux baseline is single-threaded, while TxOS+ uses transactions to execute the writes concurently.

Figure 4 compares time to recover a failed graph server node on TxOS+ and Linux. In this experiment, a server is stopped at 36 seconds and restarted at 53 seconds (17 seconds later). The recovering server re-executes all intervening requests until it has caught up. By executing recovery requests in parallel, TxOS+ recovers 29.9% faster than Linux.

8.2 IMAP

We modified the Dovecot email server, version 1.0.10, to use system transactions. Dovecot is a production-quality email server, shipped as a standard feature in many mainstream Linux distributions, supporting a range of protocols, includ-

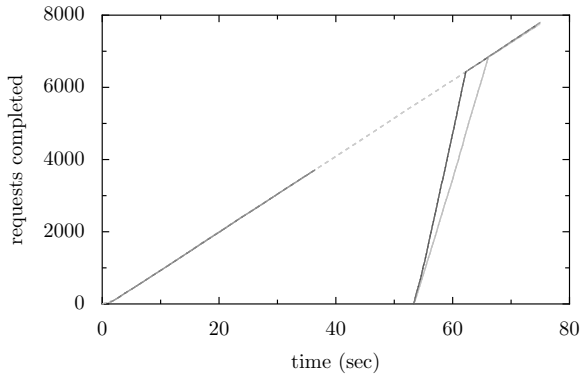

Figure 4. Comparison of time to recover a BFT server in TxOS+ (black) vs. Linux (grey). The x-axis shows time and the y-axis is in 1000's of requests completed. The dotted line tracks the correct execution nodes. In both cases, a node stopped at 36 seconds and restarted at 53 seconds. In this experiment, TxOS+ recovers 29.9% faster than Linux.

ing IMAP, and a range of back-end mail storage formats. Specifically, we changed its `maildir` backend storage format [1, 6], to use transactions instead of directory lock files. We use unmodified `maildir` as our baseline, and replace lock files with system transactions for safety. We test with the IMAP client protocol, which touts concurrent client support as an explicit, but often problematic, feature.

We note that `maildir` was originally designed to be lock free. Due to race conditions between a sequence of `readdir` system calls and `rename` calls, Dovecot has reintroduced lock files at the directory granularity [15]. We performed an experiment where we disabled this lock file and confirmed qualitatively that we could reproduce disturbing anomalies attributable to unrepeatable mailbox reads, including temporarily lost email messages when another client marks a message as read.

To evaluate the impact of adding transactions to Dovecot, we also wrote a client test application in Python 2.6.5, using the `imaplib` IMAP client library. This script uses a single test account, and launches a configurable number of concurrent clients (1–4 in our experiments), accessing a shared mailbox with an initial size of 1500 messages. Each client performs 100 operations on randomly selected messages; a configurable fraction are message reads and the rest are randomly chosen between creating a new message and deleting a random message.

We run our IMAP server on a Dell PowerEdge T300 server, with a quad-core Intel Xeon X3363 CPU at 2.83 GHz and 4 GB of RAM. We ran the client script on a SuperMicro SuperServer, with 4 quad-core Intel Xeon X7350 CPUs (16 cores total), at 2.93 GHz with 48 GB of RAM. As with the BFT experiments, our focus is on concurrent isolation, so we use the `ext2` file system for debugging simplicity.

Table 4 reports a sample of measured performance data at a range of write fractions and concurrent clients. Each value is the average of at least three runs. On single-client or read-only workloads, Dovecot on TxOS+ and Linux perform comparably, indicating that transactions do not harm baseline performance. With two clients, performance improves on TxOS+ by 7–47%, with four clients, the improvement increases from 8–80%. We attribute these performance gains to both the elimination of work creating and deleting lock files, as well as improved block allocation and write scheduling (commensurate with previous results [34]).

Clients	OS	% Writes				
		0	**10**	**25**	**50**	**100**
1	Linux	0.15	0.29	0.63	1.20	2.30
	TxOS+	0.14	0.28	0.62	0.96	2.08
2	Linux	0.12	0.42	0.95	1.47	2.83
	TxOS+	0.10	0.34	0.74	1.38	1.92
4	Linux	0.14	0.60	1.37	2.45	4.80
	TxOS+	0.13	0.38	0.80	1.40	2.66

Table 4. Execution time in seconds of an IMAP client microbenchmark comparing transactional Dovecot on TxOS+ with Unmodified Dovecot on Linux, with varying write fractions and concurrent clients. Lower is better. Work scales with the number of clients, so the same value for a write fraction across more clients indicates perfect scaling.

9. Related Work

System Transactions. A number of historical research operating systems have provided OS-level transactions, either as a general-purpose programming abstraction [37, 46], or to help isolate untrusted OS extensions [40]. These systems essentially applied database implementation techniques to the OS; TxOS [34] innovated by selecting concurrency management techniques more appropriate for an OS kernel. This work innovates by substantially refining these mechanisms and evaluates them on the largest transactional OS workloads to date (to the best of our knowledge).

Transactional file systems are deployed as part of Windows 7 [36], and they exist in several research prototypes [17, 39, 43]. It is possible that a transactional file system is sufficient to run the workloads described in this paper; but to our knowledge, integrating such large applications with file system transactions has not been explored. Based on our experience, there are a number of complications that arise when OS-level primitives outside the file system are not rolled back on a transaction abort. For instance, the IMAP implementation was simplified by automatic rollback of a file descriptor, which would require more cumbersome application code on a transactional file system.

Speculative Execution in the OS. Speculator [30] extends the operating system with an application isolation and rollback mechanism that is similar to transactions in many respects, improving system performance by speculating past

high-latency events. The primary difference is that speculations are not isolated from each other; when two speculations touch the same OS data, they are automatically merged. The initial motivation for Speculator was to hide the latency of common NFS server requests. Speculator has also been extended to hide latency of synchronous writes to a local file system [32], security checks [31], and to debug system configuration [44].

Most relevant to this work, Speculator has been used to hide BFT access latency on the **client**, speculating that it knows the likely answer all replicas will give [47]. Speculator is insufficient to parallelize execution of BFT **servers**, as speculations are not isolated. Concurrent speculations will be merged, doing nothing to prevent non-determinism. Transactions can isolate independent requests on the server and serialize them according to the prescribed order. Thus these similar systems provide related, but distinct guarantees; the differences in their isolation properties dictate what problems they can solve.

High-throughput BFT Systems. BFT systems achieve high availability through replication. This comes at a throughput cost, so improving throughput is a focus of BFT research.

Batching requests increases throughput and is used in PBFT [11] and Zyzzyva [23]. Although batching reduces communication overhead, replicas still must execute requests in a serial order. To parallelize request execution, Kotla and Dahlin [24] suggested using application-specific knowledge to find independent sets of requests in an ordered batch that can be parallelized. Checking dependencies among requests requires complete foreknowledge of their execution behavior—difficult to obtain in practice, especially when the OS is involved.

HRDB [45] and Byzantium [18] are BFT replicated databases that use that use use database transactions to parallelize request processing. These systems use weaker concurrency isolation guarantees: HRDB ensures one-copy serializability while Byzantium relies on snapshot isolation.

These systems, as well as TxOS+, use transactions as a tool for maximizing BFT concurrency. TxOS+ extends the concurrency from structured tables to unstructured OS operations, supporting a wider range of applications. Unlike TxOS+, which uses ordered transactions, these systems order commits by deferring the database commit, harming performance. Ordered system transactions provide a simpler route for applications to use transactional semantics.

Pausing transactional memory. User-level hardware transactional memory systems have proposed similar pausing [50] or open nesting [28] mechanisms to keep libc and OS state out of user-level transactions. In these systems, a program would pause its transaction before calling `malloc` to avoid hardware rollback of libc bookkeeping; in our libc, `malloc` might pause the transaction to keep its state out of the OS. To the best of our knowledge, these pausing mechanisms have not been evaluated on substantial code bases. Because software transactional memory is usually implemented *above* the JVM or libc, pausing is not necessary; pausing is needed only for lower-level transactional mechanisms.

10. Conclusion

This paper takes a promising idea, system transactions, and applies them to server applications that have struggled to balance high throughput with strong safety and concurrency guarantees due to a crippled OS interface. This paper contributes new ideas about the right OS interfaces for these applications and supporting middleware, validated with substantial workloads. In the process of supporting applications that are larger than previously studied, we improved a number of aspects in the design and implementation of system transactions. This work is an important step in a line of research towards OSes that efficiently provide strong guarantees, leading to better applications.

11. Acknowledgments

We thank the anonymous reviewers, Prince Mahajan, and our shepherd, Rodrigo Rodrigues, for valuable feedback and suggestions. We also thank Allen Clement, Manos Kapritsos, and Yang Wang for help with the UpRight code.

This research was supported by the Office of the Vice President for Research at Stony Brook University, NSF Career Award 0644205, NSF CNS-0905602, Samsung Scholarship, and an Intel equipment grant.

References

[1] Dovecot mailbox format - maildir. `http://wiki.dovecot.org/MailboxFormat/Maildir`. §1, §4.1, §8.2

[2] Welcome to the dovecot wiki. http://wiki2.dovecot.org/. §1

[3] University of oregon route views project. http://www.routeviews.org/, 2000. §8.1

[4] A. Aviram, S.-C. Weng, S. Hu, and B. Ford. Efficient system-enforced deterministic parallelism. In *OSDI*, pages 1–16, 2010. §3.2

[5] T. Bergan, N. Hunt, L. Ceze, and S. D. Gribble. Deterministic process groups in dOS. In *OSDI*, pages 1–16, 2010. §3.2

[6] D. J. Bernstein. Using maildir format (the original specification), 1995. http://cr.yp.to/proto/maildir.html. §1, §4.1, §8.2

[7] C. Blundell, E. C. Lewis, and M. M. K. Martin. Deconstructing transactions: The subtleties of atomicity. In *WDDD*. Jun 2005. §2

[8] G. Bracha and S. Toueg. Asynchronous consensus and broadcast protocols. *JACM*, 32(4):824–840, 1985. §3.1

[9] J. Bull, L. Smith, M. Westhead, D. Henty, and R. Davey. A benchmark suite for high performance java. *Concurrency - Practice and Experience*, 12(6):375–388, 2000. §8.1

[10] X. Cai, Y. Gui, and R. Johnson. Exploiting unix file-system races via algorithmic complexity attacks. In *Oakland*, pages 27–41, 2009. §3

[11] M. Castro and B. Liskov. Practical byzantine fault tolerance and proactive recovery. *ACM Transactions on Computer Systems (TOCS)*, 20(4):398–461, Nov. 2002. §9

[12] A. Clement, M. Kapritsos, S. Lee, Y. Wang, L. Alvisi, M. Dahlin, and T. Riché. Upright cluster services. In *SOSP*, pages 277–290, 2009. §1, §3.1

[13] M. Crispin. INTERNET MESSAGE ACCESS PROTOCOL - VERSION 4rev1. RFC 3501 (Proposed Standard), March 2003. Obsoletes by RFC 2060. §1, §4

[14] D. Dice, H. Huang, and M. Yang. Asymmetric dekker synchronization, July 2001. http://home.comcast.net/ pjbishop/Dave/Asymmetric-Dekker-Synchronization.txt. §7.1

[15] Dovecot maildir and racing. `http://www.dovecot.org/list/dovecot/2006-March/011811.html`. §4.1, §8.2

[16] H. Franke, R. Russel, and M. Kirkwood. Fuss, futexes and furwocks: Fast userlevel locking in Linux. In *Proceedings of the Ottawa Linux Symposium*, 2002. §6.1

[17] E. Gal and S. Toledo. A transactional flash file system for microcontrollers. In *USENIX*, pages 89–104, 2005. §9

[18] R. Garcia, R. Rodrigues, and N. Preguiça. Efficient middleware for byzantine fault tolerant database replication. In *EuroSys*, pages 107–122. ACM, 2011. §3.2, §9

[19] J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann, 1993. §2, §2

[20] J. N. Gray, R. A. Lorie, G. R. Putzolu, and I. L. Traiger. Granularity of locks and degrees of consistency in a shared data base. *Modeling in Data Base Management Systems*, pages 364–394, 1976. §2

[21] M. Herlihy and J. E. Moss. Transactional memory: Architectural support for lock-free data structures. In *ISCA*, pages 289–300, 1993. §2

[22] S. Jana, D. E. Porter, and V. Shmatikov. TxBox: Building secure, efcient sandboxes with system transactions. In *Oakland*, Oakland, CA, May 2011. §5.4

[23] R. Kotla, L. Alvisi, M. Dahlin, A. Clement, and E. Wong. Zyzzyva: Speculative byzantine fault tolerance. *SOSP*, pages 45–58, 2007. §3.1, §9

[24] R. Kotla and M. Dahlin. High throughput Byzantine fault tolerance. In *DSN*, pages 575–584. IEEE, 2003. §3.2, §9

[25] L. Lamport. Lower bounds for asynchronous consensus. *Distributed Computing*, 19(2):104–125, 2006. §3.1

[26] L. Lamport, R. Shostak, and M. Pease. The byzantine generals problem. *TOPLAS*, 4(3):382–401, 1982. §3

[27] J. Leskovec, J. Kleinberg, and C. Faloutsos. Graphs over time: densification laws, shrinking diameters and possible explanations. In *SIGKDD*, pages 177–187. ACM, 2005. §8.1

[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*, pages 359–370, 2006. §9

[29] J. Myers. Post Office Protocol - Version 3. RFC 1939 (Standard), May 1996. Obsoletes by RFC 1725. §4

[30] E. B. Nightingale, P. M. Chen, and J. Flinn. Speculative execution in a distributed file system. In *SOSP*, pages 191–205, 2005. §9

[31] E. B. Nightingale, D. Peek, P. M. Chen, and J. Flinn. Parallelizing security checks on commodity hardware. In *ASPLOS*, pages 308–318, 2008. §9

[32] E. B. Nightingale, K. Veeraraghavan, P. M. Chen, and J. Flinn. Rethink the sync. In *OSDI*, pages 1–14, 2006. §9

[33] D. E. Porter. *Operating System Transactions*. PhD thesis, The University of Texas at Austin, December 2010. §2

[34] D. E. Porter, O. S. Hofmann, C. J. Rossbach, A. Benn, and E. Witchel. Operating systems transactions. In *SOSP*, pages 161–176. ACM, 2009. §1, §1, §2, §2, §8.2, §9

[35] K. Russell and D. Detlefs. Eliminating synchronization-related atomic operations with biased locking and bulk rebiasing. *ACM SIGPLAN Notices*, 41(10):263–272, 2006. §7.1

[36] M. Russinovich and D. Solomon. *Windows Internals*. Microsoft Press, 2009. §9

[37] F. Schmuck and J. Wylie. Experience with transactions in QuickSilver. In *SOSP*, pages 239–253. ACM, 1991. §9

[38] F. B. Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Comput. Surv.*, 22(4):299–319, 1990. §3

[39] M. I. Seltzer. Transaction support in a log-structured file system. In *IDCE*, pages 503–510, 1993. §9

[40] M. I. Seltzer, Y. Endo, C. Small, and K. A. Smith. Dealing with disaster: Surviving misbehaved kernel extensions. In *OSDI*, pages 213–227, 1996. §9

[41] N. Shavit and D. Touitou. Software transactional memory. In *PODC*, pages 204–213, 1995. §2

[42] A. Singh, T. Das, P. Maniatis, P. Druschel, and T. Roscoe. BFT protocols under fire. In *NSDI*, pages 189–204, 2008. §3.2

[43] R. Spillane, S. Gaikwad, M. Chinni, E. Zadok, and C. P. Wright. Enabling transactional file access via lightweight kernel extensions. In *FAST*, pages 29–42, 2009. §9

[44] Y.-Y. Su, M. Attariyan, and J. Flinn. AutoBash: Improving configuration management with operating system causality analysis. In *SOSP*, pages 237–250, 2007. §9

[45] B. Vandiver, H. Balakrishnan, B. Liskov, and S. Madden. Tolerating byzantine faults in transaction processing systems using commit barrier scheduling. In *SOSP*, pages 59–72. ACM, 2007. §3.2, §9

[46] 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. §9

[47] B. Wester, J. Cowling, E. B. Nightingale, P. M. Chen, J. Flinn, and B. Liskov. Tolerating latency in replicated state machines. In *NSDI*, pages 245–260, 2009. §9

[48] J. Yin, J.-P. Martin, A. Venkataramani, L. Alvisi, and M. Dahlin. Separating agreement from execution for byzantine fault tolerant services. In *SOSP*, pages 253–267, 2003. §3.1

[49] M. Zalewski. Delivering signals for fun and profit, 2001. http://lcamtuf.coredump.cx/signals.txt. §3

[50] C. Zilles and L. Baugh. Extending hardware transactional memory to support non-busy waiting and non-transactional actions. In *TRANSACT*. ACM, 2006. §9