The Dissertation Committee for Donald Elliott Porter
certifies that this is the approved version of the following dissertation:

Operating System Transactions

Committee:

Emmett Witchel, Supervisor

Lorenzo Alvisi

Kathryn S. McKinley

Vitaly Shmatikov

Michael Swift

Operating System Transactions

by

Donald Elliott Porter, B.A., M.S.C.S.

Dissertation

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

Doctor of Philosophy

The University of Texas at Austin

December 2010

Dedication

To Lindsay, *sine qua non*.

Acknowledgments

I would like to express my deepest gratitude to my advisor, Emmett Witchel, for help and guidance over my graduate career. There are few people who have invested more time and energy into my success than Emmett. Moreover, he has very generously supported my pursuit of independent and external projects over the last two years. There is much that is hard to summarize adequately, so I simply say thank you.

Proverbs 27:17 says, "As iron sharpens iron, so one [person] sharpens another." This has been my experience with my wonderful collaborators: Chris Rossbach, Owen Hofmann, Indrajit Roy, Hany Ramadan, Mike Bond, Kathryn McKinley, Justin Brickell, Vitaly Shmatikov, Jung Woo Ha, Alex Benn, Aditya Bhandari, Jason Davis, and David Chen.

TxOS was a large implementation effort, and I thank the graduate students who contributed code to the system: Owen Hofmann, Chris Rossbach, Sangman Kim, Alex Benn, Indrajit Roy, Andrew Matsuoka, and Baishakhi Ray. I also thank each of my committee members for helpful comments and suggestions on the project.

I am also thankful for colleagues who offered small acts of kindness during my graduate career, including:

- Sara Standtman, who frequently guided me through the UT beauracracy, as well as helping me write a grant proposal that nearly killed us both.
- Peter Djeu, who taught me to teach better, as well as how to play Super Bomberman.

- Chris Rossbach, Harry Li, Allen Clement, and Owen Hofmann, who, in the course of sharing office space with me, returned kind words and good advice for rather frequent interruptions.
- The three Mikes I was a TA for (Scott, Dahlin, and Walfish), as well as Lorenzo Alvisi, who helped me learn to teach others better.
- Taylor Riché, who early in my graduate career impressed upon me the importance of giving a good talk and backed it up with hours watching my bad talks. He also tried (less successfully) to impress upon me the importance of typesetting minutia—you're welcome for the *accent aigu*.

To my dearest friend, Justin Quarry, who continues to teach me the meaning of dedication to one's craft.

To Wesley Beal, who reminds me to make time for important relationships.

To Terry Talley, who gave me particular help starting graduate school and to whom I owe a disproportionate share of my professional wisdom.

I thank God for the many blessings in my life.

I am especially thankful for my family; without their love, encouragement, and support I wouldn't be where I am today. I thank my Mom for showing me the fun and creative side of math at a young age; and my Dad for honing my critical and strategic thinking skills over years of conversation, advice, and merciless defeat at board games. Finally, I thank my spouse Lindsay for sharing this adventure with me.

Operating System Transactions

Publication No. _____

Donald Elliott Porter, Ph.D.
The University of Texas at Austin, 2010

Supervisor: Emmett Witchel

Applications must be able to synchronize accesses to operating system (OS) resources in order to ensure correctness in the face of concurrency and system failures. This thesis proposes **system transactions**, with which the programmer specifies atomic updates to heterogeneous system resources and the OS guarantees atomicity, consistency, isolation, and durability (ACID).

This thesis provides a model for system transactions as a concurrency control mechanism. System transactions efficiently and cleanly solve long-standing concurrency problems that are difficult to address with other techniques. For example, malicious users can exploit race conditions between distinct system calls in privileged applications, gaining administrative access to a system. Programmers can eliminate these vulnerabilities by eliminating these race conditions with system

transactions. Similarly, failed software installations can leave a system unusable. System transactions can roll back an unsuccessful software installation without disturbing concurrent, independent updates to the file system.

This thesis describes the design and implementation of **TxOS**, a variant of Linux 2.6.22 that implements system transactions. The thesis contributes new implementation techniques that yield fast, serializable transactions with strong isolation and fairness between system transactions and non-transactional activity. Using system transactions, programmers can build applications with better performance or stronger correctness guarantees from simpler code. For instance, wrapping an installation of OpenSSH in a system transaction guarantees that a failed installation will be rolled back completely. These atomicity properties are provided by the OS, requiring no modification to the installer itself and adding only 10% performance overhead. The prototype implementation of system transactions also minimizes non-transactional overheads. For instance, a non-transactional compilation of Linux incurs negligible (less than 2%) overhead on TxOS.

Finally, this thesis describes a new lock-free linked list algorithm, called OLF, for optimistic, lock-free lists. OLF addresses key limitations of prior algorithms, which sacrifice functionality for performance. Prior lock-free list algorithms can safely insert or delete a single item, but cannot atomically compose multiple operations (e.g., atomically move an item between two lists). OLF provides both arbitrary composition of list operations as well as performance scalability close to previous lock-free list designs. OLF also removes previous requirements for dynamic memory allocation and garbage collection of list nodes, making it suitable for low-level system software, such as the Linux kernel. We replace lists in the Linux kernel's directory cache with OLF lists, which currently requires a coarse-grained lock to ensure invariants across multiple lists. OLF lists in the Linux kernel improve performance of a filesystem metadata microbenchmark by 3× over unmodified Linux at 8 CPUs.

The TxOS prototype demonstrates that a mature OS running on commodity hardware can provide system transactions at a reasonable performance cost. As a practical OS abstraction for application developers, system transactions facilitate writing correct application code in the presence of concurrency and system failures. The OLF algorithm demonstrates that application developers can have both the functionality of locks and the performance scalability of a lock-free linked list.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

In current operating systems, applications access OS-managed resources, such as files and signals, through hundreds of system calls. An individual system call can generally be thought of as atomic, but the OS does not provide a general way to combine system calls. As a result, application developers face consistency errors that afford no good solution. Because applications can only access persistent storage and other shared resources through the OS, the OS is the only place to address the need for consistent access to these shared resources. The operating system programming model must be improved so that application developers can express their consistency requirements to the OS.

Just as multi-threaded applications must group accesses to shared data structures into critical regions, applications often need to group accesses to operating system resources into logical units. For example, Linux and similar operating systems store local user and group accounts in three files that need to be mutually consistent: `/etc/passwd`, `/etc/shadow`, and `/etc/group`. The lack of a general mechanism to ensure consistent access to multiple system resources leads to a range of seemingly unrelated application problems. These problems include security vulnerabilities arising from time-of-check-to-time-of-use (TOCTTOU) race conditions

in privileged applications, an unusable system after a failed software upgrade, and lost performance when concurrently accessing shared files.

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

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

In practice, OS maintainers address the lack of concurrency control in the system call API in an *ad hoc* manner: new system calls and complex interfaces are added to solve new problems as they arise. The critical problem of eliminating file system race conditions has motivated Solaris and Linux developers to add over a dozen new system calls, such as `openat`, over the last seven years. Linux maintainers added a close-on-exec flag to fifteen system calls in a recent version of Linux [Dre08] to eliminate a race condition between calls to `open` and `fcntl`. Individual file systems have introduced new operations to address consistency needs: the Google File System supports atomic append operations [GGL03], while Windows recently adopted transactions in NTFS and the Windows registry [RS09]. Users should not be required to lobby OS developers for new system calls and file system features to meet their concurrent programming needs. Why not allow users to solve their own problems by supporting composition of multiple system calls into arbitrary atomic and isolated units?

This thesis proposes *system transactions*, which group accesses to system 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 commit time, when they are atomically published to the rest of the system. System transactions provide a simple and powerful way for applications to express consistency requirements for concurrent operations to the OS.

This thesis describes the design and implementation of system transactions

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

This thesis makes three key contributions. First, it describes a new approach to OS implementation that supports efficient transactions on commodity hardware with strong atomicity and isolation guarantees. These techniques are implemented in the TxOS prototype. Secondly, the thesis describes several case studies in using system transactions to build better applications. It shows that system transactions yield better applications in terms of parallel performance or stronger correctness guarantees with simpler code. Finally, the thesis describes a new lock-free linked list algorithm that is appropriate for use in an OS kernel. This algorithm addresses two key limitations of previous lock-free list algorithms: composability and reliance on dynamic memory management.

1.1 Motivating examples

A range of seemingly unrelated application problems share a root cause—the lack of a general mechanism to ensure consistent access to system resources. This section reviews two common application consistency problems and how system transactions remedy those problems. System transactions greatly simplify recovery from a failed software installation, leaving concurrent, independent updates to the system undisturbed. System transactions also eliminate race conditions inherent in the file system API, which can be exploited to undermine security.

1.1.1 Software installation or upgrade

Installing new software or software patches is an increasingly common system activity as time to market pressures and good network connectivity combine to make software updates frequent for users. Yet software upgrade remains a dangerous activity. For example, Microsoft recalled a prerequisite patch for Vista service pack 1 because it caused an endless cycle of boots and reboots [McD08]. In general, a partial upgrade can leave a system in an unusable state.

Current systems are adopting solutions that mitigate these problems, but each has its own drawbacks. Microsoft Windows and other systems provide a checkpoint-based solution to the software update problem. Users can take a checkpoint of disk state before they install software: if something goes wrong, they roll back to the checkpoint. Windows checkpoints certain key structures, like the registry and some system files [Mic08]. Other systems, like ZFS's `apt-clone`, checkpoint the entire file system. If the software installation fails, the system restores the pre-installation file system image, erasing file system updates that are concurrent but independent from the software installation. Partial checkpointing mitigates this problem, but loses the ability to recover from application installations that corrupt files not checkpointed by the system. Moreover, the user or the system must create and manage the disk-based checkpoints to make sure a valid image is always available. Finally, if a bad installation affects volatile system state, errant programs can corrupt files unrelated to the failed installation. Collectively, these problems severely decrease the usability of checkpoint-based solutions.

System transactions provide a simple interface to address these software installation problems (Section 5.3). A user executes the software installation or update within a transaction, which isolates the rest of the system until the installation successfully completes. If the changes to the system made by the installation or upgrade need to be rolled back, independent updates made concurrently remain undisturbed.

Victim	Attacker
`if(access('foo')){`	
	`symlink('secret','foo');`
`  fd=open('foo');`	
`  write(fd,...);`	
`  ...`	
`}`	

Victim	Attacker
	`symlink('secret','foo');`
`sys_xbegin();`	
`if(access('foo')){`	
`  fd=open('foo');`	
`  write(fd,...);`	
`  ...`	
`}`	
`sys_xend();`	
	`symlink('secret','foo');`

Figure 1.1: An example of a TOCTTOU attack, followed by an example that eliminates the race using system transactions. The attacker's symlink is serialized (ordered) either before or after the transaction, and the attacker cannot see partial updates from the victim's transaction, such as changes to `atime`.

1.1.2 Eliminating races for security

Figure 1.1 depicts a scenario in which an application wants to make a single, consistent update to the file system by checking the access permissions of a file and conditionally writing it. Common in `setuid` programs, this pattern is the source of a major and persistent security problem in modern operating systems. An attacker can change the file system name space using symbolic links between the victim's `access` control check and the file `open`, perhaps tricking a `setuid` program into overwriting a sensitive system file, like the password database. The OS API provides no way for the application to tell the operating system that it needs a consistent view of the file system's name space.

Although system API race conditions, or time-of-check-to-time-of-use (TOCT-

TOU) races, are most common in applications' uses of the file system APIs, they are possible in other OS resources. Local sockets used for IPC are vulnerable to a similar race between creation and connection. Versions of OpenSSH before 1.2.17 suffered from a socket race exploit in which a user steals another's credentials [Ach]; the Plash sandboxing system suffers a similar vulnerability [Pla]. Zalewski describes how to exploit races in signal handlers to compromise mature, widely-deployed applications, including sendmail, screen, and wu-ftpd [Zal01].

While TOCTTOU vulnerabilities are conceptually simple, they pervade deployed software and are difficult to eliminate. At the time of writing, a search of the U.S. national vulnerability database for the term "symlink attack" yields over 600 hits [NIS10]. Further, recent work by Cai et al. [CGJ09] exploits fundamental flaws to defeat two major classes of TOCTTOU countermeasures: dynamic race detectors in the kernel [TY03] and probabilistic user-space race detectors [THWS08b]. This continuous arms race of measure and countermeasure suggests that the only way to eliminate TOCTTOU vulnerabilities is changing the OS API.

In practice, such races are addressed with ad hoc extension of the system API. Linux has added a new close-on-exec flag to fifteen different system calls to eliminate a race condition between calls to `open` and `fcntl`. Tsafrir et al. [THWS08a] demonstrate how programmers can use the *openat()* family of system calls to construct deterministic countermeasures for many races by traversing the directory tree and checking user permissions in the application. However, these techniques cannot protect against all races without even more API extensions. In particular, they are incompatible with the `O_CREAT` flag to `open` that prevents exploits on temporary file creation [CBWK01].

Fixing race conditions as they arise is not an effective long-term strategy. Complicating the API in the name of security is risky: code complexity is the enemy of code security [Ber07]. Because system transactions provide deterministic

safety guarantees and a natural programming model, they are an easy-to-use, general mechanism that eliminates API race conditions (Section 5.6).

1.2 Composing linked list operations without locks

In order to safely share data structures, many programs adopt locking, which limits the performance scalability of the application. For some data structures, the programmer can adopt specialized implementations that permit more concurrency, but these specializations often restrict functionality. This thesis addresses the unsavory dilemma between functionality and scalability in linked list implementations, describing a new linked list design that provides the complete functionality of locked lists with performance scalability closer to that of current lock-free implementations.

Applications with invariants across multiple lists cannot benefit from the performance scalability of lock-free lists, such as the `ConcurrentSkipList` classes of the Java Concurrency package. Current lock-free list designs, including the popular Harris-Michael algorithm [Har01, Mic02], can safely insert or remove individual items from a list, but cannot safely compose multiple operations. For instance, if an application moves an item from one lock-free list to another, the list implementation can atomically remove the item from the first list and atomically add the item to the second list, but there will be a period during which the item is is either on both lists, or on neither list. If an application invariant requires the item to be on exactly one of two lists, the only safe option is to lock the lists. The inability to safely compose complex list operations prevents some applications from enjoying the performance benefits of lock-free lists.

A second barrier to adoption of lock-free lists is their reliance on dynamic allocation and reclamation of list nodes. The Harris-Michael algorithm avoids synchronization with readers of a deleted list node by deferring reclamation until it is sure no threads have a reference to the node. If an item is added to another list, a

new list node is dynamically allocated.

This reliance on dynamic allocation of list nodes is unacceptable for operating system kernels, which manage physical memory and file system caches using lists. For instance, it is difficult to write low-level memory management code so reentrant that, under a low memory condition, the kernel can successfully allocate list nodes in order to add reclaimed memory to a free list. Previous attempts to dynamically allocate and garbage collect list nodes in kernel code have proved intractable [MS05]. For this reason, OS kernels including Linux avoid dynamic memory allocation of list nodes by embedding the list nodes directly in the data structures to be listed. In the Linux kernel, these sets of lists are protected by coarse grained locks, which have proved to be a persistent scalability bottleneck [Cor09]. If OS kernels used more scalable lists, all applications would benefit; this will require a lock-free list design that can provide scalability benefits without such heavy reliance on dynamic memory management.

This thesis presents a new lock-free linked list design, called OLF, or optimistic, lock-free lists. OLF eliminates the functional restrictions of previous designs while retaining performance scalability drastically superior to locking. The OLF design is suitable for use in an OS kernel, and this thesis describes its application to improve Linux kernel scalability.

1.3 Summary

This thesis contributes new, practical designs and implementations concurrent programming abstractions that are simple for application programmers to use, but challenging for lower-level software to provide to applications. Specifically, the TxOS prototype demonstrates that a mature OS running on commodity hardware can provide system transactions at a reasonable performance cost. As a practical OS abstraction for application developers, system transactions facilitate writing correct

application code in the presence of concurrency and system failures. The OLF algorithm demonstrates that application developers can have both the functionality of locks and the performance scalability of a lock-free linked list.

Previous publications. An extended motivation for system transactions in a modern OS and a high-level description of the TxOS design appeared in [PW09]. Porter et al. [PHR+09] provide a thorough description of TxOS, and the later Linux Symposium paper [PW10] provides additional technical details. Chapters 3 and 4 substantially extend previously published descriptions of the TxOS system. Ongoing work with TxOS, described in 6, is previously unpublished. The lock-free list design (OLF) described in Chapter 8 is currently under submission.

1.4 Thesis organization

The rest of this thesis is organized as follows. Chapter 2 provides a technical overview of system transactions and the TxOS kernel. Chapter 3 describes the TxOS kernel. Chapter 4 describes how to extend transactions into various kernel subsystems, including the file system, memory management, and signals. Chapter 5 evaluates the performance of the TxOS kernel on a range of benchmarks and microbenchmarks. Chapter 6 describes how system transactions can serve as a building block for better applications, describing both completed and future work. Chapter 7 reviews related work on transactions in an OS kernel. Chapter 8 describes a new lock-free linked list algorithm that eliminates certain restrictions imposed by previous algorithms, which precluded their use in an OS kernel. Chapter 9 concludes.

Chapter 2

Technical overview

The design of system transactions seeks to give programmers a simple and intuitive abstraction for ensuring consistent access to system resources. This chapter describes the API, semantics, and behavior of system transactions, followed by an overview of TxOS, our prototype implementation of system transactions within Linux.

2.1 System transactions

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

2.1.1 System transaction semantics

System transactions share several properties with database transactions, which developers are likely familiar with. 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 [GR93]). Transactions are atomic (the system can always roll back to a pre-transaction state) and durable (transaction results, once committed, survive system crashes).

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

System transactions make durability optional because durability often increases transaction commit latency and the programmer does not always need it. Durability is only relevant to file system state, as volatile system state does not survive system crashes regardless of transactions. The increased commit latency comes from flushing data to a slow block storage device, like a disk. Eliminating the TOCTTOU race in the file system namespace is an example of a system transaction that does not require durability. Durability for system transactions in TxOS is under the control of the programmer, using a flag to `sys_xbegin()` (Table 2.1).

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

Function Name	**Description**
int sys_xbegin (int flags)	Begin a transaction. The flags specify transactional behavior, as described below. The call returns the current retry attempt on success (i.e., 0 on the first instance, 1 after the first retry, etc.) and a negative value on failure.
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 the NO_AUTO_RETRY flag unset (the default behavior), setting no_restart overrides that flag and does not restart the transaction after it is rolled back.
sys_xbegin Flag	**Description**
0 DEFAULTS	By default, transactions are durable, provide rollback of the application's address space with copy-on-write paging, automatically retry on an abort, and the transaction aborts if it attempts an unrecoverable system call. Any of these features can be disabled with the flags below.
1 NONDURABLE	Transaction updates are not guaranteed to be on stable storage when commit returns. Recommended to avoid synchronous writes when durability is not required.
2 NO_USER_ROLLBACK	Disables copy-on-write paging for the user-level address space. If a transaction aborts, user-mode data is not rolled back by the kernel, only kernel modifications are rolled back. Recommended for use by expert programmers, or with a user-level transactional memory system.
4 NO_AUTO_RETRY	Disables automatic retry of an aborted transaction.
8 ERROR_UNSUPPORTED	If a transaction attempts an unrecoverable system call, have that call return error number (ENOTXSUPPORT—134). The transaction does not abort.
16 LIVE_DANGEROUSLY	If a transaction attempts an unrecoverable system call, proceed and log a warning to the system log.

Table 2.1: TxOS API

We call a kernel thread executing a system transaction a transactional kernel thread.

2.1.2 Interaction of transactional and non-transactional threads

The OS serializes system transactions and non-transactional system calls, providing the strongest guarantees and most intuitive semantics to the programmer [GR93]. The serialization of transactional and non-transactional updates to the same resources is called **strong isolation** [BLM05]. Previous OS transaction designs have left the interaction of transactions with non-transactional activity semantically murky. Intuitive semantics for mixing transactional and non-transactional access to the same resources is crucial to maintaining a simple interface to system resources. Strong isolation prevents unexpected behavior when non-transactional and transactional applications access the same system resources.

The presence of system transactions does not change the behavior of non-transactional activity in the underlying operating system. While most system calls are already isolated and atomic, there are important exceptions. For example, Linux does not serialize `read` with `write`. In TxOS, `read` system calls inside of a system transaction will be properly serialized with a non-transactional `write`, but non-transactional system calls can still exhibit non-serializable behavior with respect to each other. Non-transactional system calls serialize with transactions, but the interactions among non-transactional system calls are generally undisturbed on TxOS.

2.1.3 System transaction progress

The operating system guarantees that system transactions do not livelock with other system transactions. When two transactions, A and B, cannot both commit, the system selects one to restart (let's say B in this example), and ensures its decision remains consistent. If A continues and B restarts and again conflicts with A, the OS will again restart B. See Section 3.2.2 for details.

The TxOS design guarantees progress for transactional threads in the presence of non-transactional threads. Because non-transactional system calls cannot be aborted, a simple approach to serializing transactions with non-transactional system calls would always abort the transaction. This runs the risk of starving long-running transactions. If an OS supports preemption of kernel threads (present in Linux 2.4 and 2.6 since 2004), then it can guarantee progress for long running transactions by preempting non-transactional threads that would impede progress of the transaction (Section 3.2.3).

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

2.1.4 System transactions for system state

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

For most applications, we expect programmers will use a library or runtime system that transparently manages application state as well as system transactions. Most applications have state that operates in tandem with OS-managed state, such as a memory allocator having user-level bookkeeping on top of a kernel-allocated memory region. When OS state is rolled back, the application often needs to roll

back the changes to its own state. In simple cases, such as the TOCTTOU example, the developer could manage application state herself. TxOS provides single-threaded applications with an automatic checkpoint and restore mechanism for the application's address space that marks the pages copy-on-write (similar to Speculator [NCF05]), which can be enabled with a flag to `sys_xbegin()` (Table 2.1). In Section 3.7, we describe how system transactions integrate with hardware and software transactional memory, providing a complete transactional programming model for multi-threaded applications.

2.1.5 Communication model

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

Communication among threads within the same transaction is unrestricted. A thread may only be in one transaction at a time, but beyond that, the mapping of threads to transactions is quite flexible. Threads in the same process may be in different transactions, and multiple threads, including threads in different processes, may be in the same transaction. This thesis only considers system transactions on a single machine, but future work could allow system transactions to span multiple machines.

Supported			
Subsystem	**Tot.**	**Part.**	**Examples**
Credentials	34	1	getuid, getcpu, setrlimit (partial)
Processes	13	3	fork, vfork, clone, exit, exec (partial)
Communication	15	0	rt_sigaction, rt_sigprocmask, pipe
Filesystem	63	4	link, access, stat, chroot, dup, open, close, write, lseek
Other	13	6	time, nanosleep, ioctl (partial), mmap2 (partial)
Totals	138	14	Grand total: 152

Unsupported		
Subsystem	**Tot.**	**Examples**
Processes	33	nice, uselib, iopl, sched_yield, capget
Memory	15	brk, mprotect, mremap, madvise
Filesystem	29	mount, sync, flock, setxattr, io_setup, inotify
File Descriptors	14	splice, tee, sendfile, select, poll
Communication	8	socket, ipc, mq_open, mq_unlink
Timers/Signals	12	alarm, sigaltstack, timer_create
Administration	22	swapon, reboot, init_module, settimeofday
Misc	18	ptrace, futex, times, vm86, newuname
Total	151	

Table 2.2: Summary of system calls that TxOS completely supports (Tot.) and partially supports (Part.) in transactions, followed by system calls with no transaction support. Partial support indicates that some (but not all) execution paths for the system call have full transactional semantics. Linux 2.6.22.6 on the i386 architecture has 303 total system calls.

2.2 TxOS overview

TxOS implements system transactions by isolating data read and written in a transaction using existing kernel memory buffers and data structures. When an application writes data to a file system or device, the updates generally go into an OS buffer first, to optimize OS device accesses. By making these buffers copy-on-write for transactions, TxOS isolates transactional data accesses until commit. In TxOS, transactions must fit into main memory, although this limit could be raised in future work by swapping uncommitted transaction state to disk.

TxOS isolates updates to kernel data structures using recent implementation techniques from object-based software transactional memory systems [FH07, HK08, HLMS03]. These techniques are a departure from the logging and two-phase locking approaches of databases and historic transactional operating systems (Section 3.1). TxOS's isolation mechanisms are optimistic, on the assumption that concurrent transactions with conflicts are rare.

Table 2.2 summarizes the system calls and resources for which TxOS supports transactional semantics, including the file system, process and credential management, signals, and pipes. A partially supported system call means that some processing paths are fully transactional, and some are not. For example, `ioctl` is essentially a large switch statement, and TxOS does not support transactional semantics for every case. When a partially supported call cannot support transactional semantics, or an unsupported call is issued, the system logs a warning or aborts the transaction, depending on the flags passed to `sys_xbegin()`.

Ideal support for system transactions would include every reasonable system call. TxOS supports a subset of Linux system calls as shown in Table 2.2. The count of 152 supported system calls shows the relative maturity of the prototype, but also indicates that it is incomplete. The count of unsupported system calls does not proportionately represent the importance or challenge of the remaining work be-

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

Chapter 3

The TxOS Kernel

The TxOS kernel provides transactional semantics for a series of system calls with strong properties. This chapter describes the design and implementation of the core transaction facilities in the TxOS kernel.

TxOS provides a framework for transactionalizing data structures in kernel subsystems, such as the virtual file system and signals. Chapter 4 describes the transactionalized subsystems in TxOS. Across all of these subsystems, TxOS provides transactional semantics for 152 of 303 system calls in Linux, presented in Table 2.2. The supported system calls include process creation and termination, credential management operations, sending and receiving signals, and file system operations.

System transactions in TxOS add roughly 3,300 lines of code for transaction management, and 5,300 lines for object management. TxOS also requires about 14,000 lines of minor changes to convert kernel code to use the new object type system and to insert checks for asymmetric conflicts when executing non-transactionally.

3.1 Version management of transactional state

Version management is the first key design point of any transactional system, be it a database, transactional memory implementation, or operating system. A transactional system must track both speculatively modified data, so that the transaction can commit, and the original data, so that the transaction can abort. As explained in this section, the selected approach to version management has repercussions for scheduling, including the risk of deadlock and high latency for interrupt handling. A key innovation in this thesis is the design of a version management solution that is suitable for the scheduling constraints of an operating system kernel.

Databases and historical transactional operating systems typically update data in place and maintain an undo log. This approach is called **eager version management** [LR06]. These systems isolate transactions by locking data when it is accessed and holding the lock until commit. This technique is called two-phase locking, and it usually employs locks that distinguish read and write accesses. Because applications generally do not have a globally consistent order for data accesses, these systems can deadlock. For example, one thread might read file A then write file B, while a different thread might read file B, then write file A.

The possibility of deadlock complicates the programming model of eager versioning transactional systems. Deadlock is commonly addressed by exposing a timeout parameter to users. Setting the timeout properly is a challenge. If it is too short, it can starve long-running transactions. If it is too long, it can destroy the performance of the system.

Eager version management degrades responsiveness in ways that are not acceptable for an operating system. If an interrupt handler, high priority thread, or real-time thread aborts a transaction, it must wait for the transaction to process its undo log (to restore the pre-transaction state) before it can safely proceed. This wait jeopardizes the system's ability to meet its timing requirements.

TxOS, in contrast, uses **lazy version management**, where transactions operate on private copies of a data structure. Applications never hold kernel locks across system calls. Lazy versioning requires TxOS to hold locks only long enough to make a private copy of the relevant data structure. By enforcing a global ordering for kernel locks, TxOS avoids deadlock. TxOS can abort transactions instantly—the winner of a conflict does not incur latency for the aborted transaction to process its undo log.

The primary disadvantage of lazy versioning is the commit latency due to copying transactional updates from the speculative version to the stable version of the data structures. As we discuss in Subsection 3.1.2, TxOS minimizes this overhead by splitting objects, turning a `memcpy` of the entire object into a pointer copy.

3.1.1 Versioning kernel objects

TxOS implements transactional system call semantics by maintaining multiple versions of kernel data structures. Kernel data structures private to a thread, such as the current user id, are versioned with a simple checkpoint and restore scheme. By versioning shared kernel data structures with a more sophisticated, lazy versioning system, the kernel has more flexibility in arbitrating conflicts and can recover from a transactional conflict faster.

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

```
struct inode_header {
  atomic_t      i_count; // Reference count
  spinlock_t    i_lock;
  inode_data    *data;   // Data object
  // Other objects
  address_space i_data;  // Cached pages
  tx_data xobj;   // for conflict detection
  list i_sb_list;  // kernel bookkeeping
};

struct inode_data {
  inode_header *header;
  // Common inode data fields
  unsigned long i_ino;
  loff_t        i_size; // etc.
};
```

Figure 3.1: A simplified `inode` structure, decomposed into header and data objects in TxOS. The header contains the reference count, locks, kernel bookkeeping data, and the objects that are managed transactionally. The `inode_data` object contains the fields commonly accessed by system calls, such as `stat`, and can be updated by a transaction by replacing the pointer in the header.

3.1.2 Splitting objects into header and data

To efficiently commit lazy versioned data, TxOS decomposes objects into a stable **header** component and a volatile, transactional **data** component. Figure 3.1 provides an example of this decomposition for an `inode`. The object header contains a pointer to the object's data; transactions commit changes to an object by replacing this pointer in the header to a modified copy of the data object.

Note that the header itself is never replaced by a transaction. The header provides a stable target for incoming pointers, similar to a smart pointer. Pointers to objects always point to the object headers. A naïve system might attempt to leave kernel data structures untouched, and instead modify all incoming references. This approach proves impractical because updating these pointers introduces writes to additional objects; these writes potentially generate conflicting accesses and abort otherwise non-conflicting transactions.

The object header can also contain data that is not modified by transactions.

For instance, the kernel garbage collection thread (kswapd) periodically scans the `inode` and `dentry` (directory entry) caches looking for unused, cached file system data to garbage collect. In the case of inodes, kswapd reads certain kernel bookkeeping fields, including the reference count and the superblock list (`i_sb_list` in Figure 3.1), to determine if the inode is in use. If it is being used by any system call, transactional or not, kswapd simply moves on to the next thread. TxOS avoids needless conflicts between a transaction writing to a field in the inode, such as the permission bits, and kswapd by keeping the kernel bookkeeping in the header. Kswapd scans never access the associated `inode_data` objects of an in-use inode and thereby avoid false conflicts.

Multiple data objects. Several kernel objects, including `inodes`, have evolved such that they contain sets of data that have little semantic relationship to each other. In these cases, TxOS decomposes an object into multiple data payloads. For instance, a process may may often read or write a file without updating the metadata. In TxOS, the `inode_data` contains both file metadata (owner, permissions, etc.) and the header has the mapping of file blocks to cached pages in memory (`i_data`). TxOS versions these objects separately, allowing metadata operations and data operations on the same file to execute concurrently when it is safe.

3.1.3 Read-only objects

A system transaction often reads many kernel objects that it does not modify. For instance, path lookup code can read each of the parent directories while locating a file to modify. To avoid the cost of making shadow copies, kernel code can specify read-only access to an object, which marks the object data as read-only for the length of the transaction. Each data object has a transactional reader reference count. If a writer, transactional or non-transactional, wins a conflict for an object with a non-zero reader count, it must create a new copy of the object and install

it as the new stable version. Once all transactional readers release the read-only version and after all CPUs have quiesced, the OS garbage collects the old copy via read-copy update (RCU) [McK04]. In the Linux kernel, a CPU quiesces when it context switches a process. Linux follows an implementation convention that all references to a freeable object (i.e., with a zero reference count) must be released before a thread is descheduled; this convention is enforced by disabling preemption in the kernel if needed. RCU reclamation of read-only objects ensures that all active references to the old, read-only version have been released before it is freed and all tasks see a consistent view of kernel data.

In the current TxOS prototype, non-transactional threads can hold references to read-only data objects across scheduling events; these data objects can be replaced while the thread is suspended. For instance, a non-transactional thread may read an inode's metadata from a read-only `inode_data` object before issuing a disk read. While the thread is descheduled for the disk read, the read-only `inode_data` could be garbage collected. After the disk read completes, the non-transactional thread resumes with a stale reference to the `inode_data`. In TxOS, if a non-transactional task is descheduled during a blocking call, it must re-acquire references to any data objects by reading the object header. Note that the reference count on the header object will be non-zero, so the header can be safely used to re-acquire the data object reference. As an alternative, the programming model might be simplified by imposing additional constraints on reclamation of read-only objects; the ramifications on memory pressure and consistency of non-transactional system calls would need to be more carefully evaluated.

Marking data objects as read-only in a transaction is a structured way to eliminate substantial overhead for memory allocation and copying. Although this optimization complicates the kernel programming model slightly, the instances where reacquisition is needed are small (in the tens). Special support for read-mostly

transactions is a common optimization in transactional systems, and using RCU gives TxOS efficient, concurrent access to read-mostly data.

3.2 Conflict detection and interoperability

Conflicts occur when a transaction attempts to write an object that has been read or written by another transaction. In order to ensure the safety of system transactions, TxOS must serialize conflicting transactions, typically by having one roll back and retry. TxOS efficiently serializes transactions with other transactions as well as non-transactional system calls. In order to uphold concurrent performance, TxOS must avoid needlessly serializing transactions that can safely execute at the same time. This section explains several key design points related to detecting and resolving conflicts.

3.2.1 Conflict detection

In order to provide serializable transactions, TxOS must detect conflicts between transactions, and detect **asymmetric conflicts** [RRP+07] between transactions and non-transactional threads. In general, a conflict is defined as a write to an object that has been read or written by another transaction. An asymmetric conflict is defined similarly: a non-transactional thread attempts to write an object a transaction has read or written, or vice versa.

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

TxOS embeds a `transactional_object` structure (Figure 3.2) in the header portion of all shared kernel objects that can be accessed within a transaction. The

```
struct transactional_object {
       // Type encoding
       enum tx_object_type type;

       // Writer pointer
       struct transaction *writer;

       // List of readers
       struct list_head readers;

       // Spinlock, for synchronization - acquired after object's non-blocking lock
       spinlock_t lock;
};
```

Figure 3.2: The `transactional_object` structure, or `xobj` for brevity, which is embedded in the header of each object. The type field indicates in which type of kernel object the xobj is embedded. The writer pointer, if set, indicates the current transaction with exclusive (write) access. The reader list, if non-empty, indicates which transactions have shared (read) access. The lock protects the xobj from concurrent access.

`transactional_object` structure (or `xobj`, for brevity) includes a pointer to a transactional writer and a reader list. A non-null writer pointer indicates an active transactional writer, and a non-empty reader list indicates there are transactional readers. Note that the reader list is attached to the stable header object, whereas the reader count (§3.1) is used for garbage collecting obsolete data objects. The writer pointer and reader list included in each object header for conflict detection are referred to as **annotations**.

When a transaction accesses an object for the first time, it first acquires the appropriate object locks. These locks prevent transactions from acquiring an object that is concurrently accessed by a non-transactional thread. When a thread acquires these locks, either to make a shadow copy (transaction) or to directly access the object (non-transaction), it then acquires the lock on the `xobj` and uses the writer pointer and reader list to detect a conflict. When a thread detects a conflict, TxOS uses these fields to determine which transactions are in conflict; the conflict is then arbitrated by the contention manager (§3.2.2).

TxOS efficiently provides strong transaction isolation inside the kernel by

requiring all system calls to follow the same locking discipline, and by requiring that transactions annotate accessed kernel objects. When a thread, transactional or non-transactional, accesses a kernel object for the first time, it must check for a conflicting annotation. The scheduler arbitrates conflicts when they are detected. In many cases, conflicts are detected when a thread first enters a critical region.

3.2.2 Contention Management

When a conflict is detected between two transactions or between a transaction and a non-transactional thread, TxOS invokes the contention manager to resolve the conflict. The contention manager is kernel code that implements a policy to arbitrate conflicts among transactions, dictating which of the conflicting transactions may continue. All other conflicting transactions must abort.

As a default policy, TxOS adopts the *osprio* policy [RHP+07]. Osprio always selects the process with the higher static scheduling priority as the winner of a conflict, eliminating priority and policy inversion in transactional conflicts. When processes with the same priority conflict, the older transaction wins (a policy known as timestamp [RG02]), guaranteeing liveness for transactions within a given priority level.

The current TxOS prototype allows the system-wide contention management policy to be configured by writing to `/proc/tx_res_policy_ctl`. Linux provides a special file system in the `/proc` directory for dynamic kernel configuration. TxOS also provides the timestamp policy and a version of osprio based on dynamic scheduling priority.

3.2.3 Asymmetric conflicts and fairness

A conflict between a transactional and non-transactional thread is called an asymmetric conflict. Transactional threads can always be aborted and rolled back, but

non-transactional threads cannot be rolled back. TxOS must have the freedom to resolve an asymmetric conflict in favor of the transactional thread, otherwise asymmetric conflicts will always win, undermining fairness in the system and possibly starving transactions.

While non-transactional threads cannot be rolled back, they can often be preempted, which allows them to lose conflicts with transactional threads. Kernel preemption is a recent feature of Linux that allows the kernel to preemptively deschedule threads executing system calls inside the kernel, unless they are inside of certain critical regions. In TxOS, non-transactional threads detect conflicts with transactional threads before they actually update state, usually when they acquire a lock for a kernel data structure. A non-transactional thread can simply deschedule itself if it loses a conflict and is in a preemptible state. If a non-transactional, non-preemptible process aborts a transaction too many times, the kernel can still prevent it from starving the transaction by placing the non-transactional process on a wait queue the next time it makes a system call. The kernel reschedules the non-transactional process only after the transaction commits.

Linux can preempt a kernel thread if the thread is not holding a spinlock and is not in an interrupt handler. TxOS has the additional restriction that it will not preempt a conflicting thread that holds any blocking locks (mutexes or semaphores). Otherwise, TxOS risks a deadlock with a transaction that might need that kernel lock to commit. In all cases, the kernel locking discipline is followed and deadlock is not a risk. In the pathological case where a transaction is aborted by non-preemptible non-transactional tasks many times, more extreme measures can be taken to ensure progress, such as suspending non-transactional threads that might abort the transaction. These fail-stop features are unimplemented in the current TxOS prototype.

Because asymmetric conflicts in TxOS are often detected before a non-

State	Description
`exclusive`	Any attempt to access the list is a conflict with the current owner
`write`	Any number of insertions and deletions are allowed, provided they do not access the same entries. Reads (iterations) are not allowed. Writers may be transactions or non-transactional tasks.
`read`	Any number of readers, transactional or non-transactional, are allowed, but insertions and deletions are conflicts.
`notx`	There are no active transactions, and a non-transactional thread may perform any operation. A transaction must first upgrade to `read` or `write` mode.

Table 3.1: The states for a transactional list in TxOS. Having multiple states allows TxOS lists to tolerate access patterns that would be conflicts in previous transactional systems.

transactional thread enters a critical region, the scheduler has the option of suspending the non-transactional thread, enforcing fairness between transactions and non-transactional threads. By using kernel preemption and lazy version management, TxOS has more flexibility to coordinate transactional and non-transactional threads than previous transactional operating systems.

3.2.4 Minimizing conflicts on lists

Linked lists are a key data structure in the Linux kernel, and they present key implementation challenges for system transactions. This subsection addresses false conflicts on linked lists in the TxOS kernel. Chapter 8 describes OLF, a new lock-free linked list algorithm, which addresses a different, but related problem. A helpful distinction is that TxOS uses locking to protect list operations (e.g., adding or removing an element), and adds additional bookkeeping to prevent false conflicts at the granularity of the system transaction (described in this section). OLF eliminates locking on list operations. Adopting OLF in TxOS is future work.

Simple read/write conflict semantics for lists throttle concurrent performance, especially when the lists contain directory entries. For instance, two transactions

should both be allowed to add distinct directory entries to a single list, even though each addition is a list write. TxOS adopts techniques from previous transactional memory systems to avoid conflicts on list updates that do not semantically conflict [HK08]. TxOS isolates list updates with a lock and defines conflicts according to the states described in Table 3.1. For instance, a list in the `write` state allows concurrent transactional and non-transactional writers, so long as they do not access the same entry. Individual entries that are transactionally added or removed are annotated with a transaction pointer that is used to detect conflicts. If a writing transaction also attempts to read the list contents, it must upgrade the list to `exclusive` mode by aborting all other writers. The `read` state behaves similarly. This design allows maximal list concurrency while preserving correctness.

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

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

for list traversal or manipulation. Transactional allocations require an atomic memory allocation; if this allocation fails, the transaction may have to abort in order to refill the allocator. Although unimplemented in the current prototype, the risk of allocation failure can be minimized by ensuring the list node allocator has sufficient memory before entering a non-preemptible code region.

A final issue with lists and transactional scalability is that most lists in the Linux kernel are protected by coarse locks, such as the `dcache_lock`. Ideally, two transactions that touch disjoint data should be able to commit concurrently, yet acquiring a coarse lock will cause needless performance loss. Thus, we implemented fine-grained locking on lists, at the granularity of a list. The TxOS `dcache` replaces the Linux `dcache_lock` with a lock for each of the four types of list a directory entry can participate in: 1) a least-recently used list for memory reclamation, 2) a list of all hard links to a file's inode, 3) a list of all cached files in a given directory, and 4) a hash list, for fast name lookup. Fine-grained list locking improves scalability (§ 5.7), but complicates the locking discipline. Locks in TxOS are ordered by kernel virtual address, except that list locks must be acquired after other object locks. This discipline roughly matches the paradigm in the directory traversal code.

3.3 Managing transaction state

TxOS introduces a transaction object to the kernel to store metadata and statistics for a transaction. The kernel thread's control block (the `task_struct` in Linux) points to the transaction object, shown in Figure 3.3. A thread can have at most one active transaction, though transactions can flat nest, meaning that all nested transactions are subsumed into the enclosing transaction. Transactions are independent of a process's address space, so each thread in a multi-threaded application can operate in a separate transaction. Moreover, multiple threads (even those in different processes) may share a transaction, as we discuss in Section 3.3.1. Cur-

```
struct transaction {
        atomic_t status;  // live/aborted/inactive
        uint64 tx_start_time;// timestamp
        uint32 retry_count;
        int flags; // autoretry, durable, etc.
        wait_queue_head_t losers;  /* Wait queue for losers */

        /* Workset */
        struct skiplist_head  object_list;
        struct skiplist_head  list_list;
        struct list_head data_writer_list;
        spinlock_t workset_lock;

        /* Fields used for multi-process transactions */
        spinlock_t mp_lock;  /* Protects the multi-process fields */
        /* The threads associated with this transaction */
        struct list_head tasks;
        atomic_t task_count;
        /* Wait queue for commit */
        wait_queue_head_t sibling_threads;

        orphaned_deferred_ops; // operations done at commit
        orphaned_undo_ops;   // operations undone at abort
};
```

Figure 3.3: Data contained in a system transaction object, which is pointed to by the thread control block (`task_struct`).

rently, multi-threaded and multi-process transactions are only created by `fork`ing a new thread inside of a system transaction, although adding an interface for threads to join a transaction would be straightforward if needed. In order to avoid needless synchronization in a multiprocess transaction, some thread-local transaction data is also stored in the `task_struct`, shown in Figure 3.4.

Figure 3.3 summarizes the fields of the transaction object. The transaction includes a status word (`status`). If another thread wins a conflict with this thread, it will update this word atomically with a compare-and-swap instruction. The kernel

```
struct task_tx_data {
        struct transaction *transaction;
        /* Node in transaction->tasks list */
        struct list_head transaction_list;
        struct pt_regs *checkpointed_registers;

        orphaned_deferred_ops; // operations done at commit
        orphaned_undo_ops;   // operations undone at abort
};
```

Figure 3.4: Task-local data used by a system transaction object, embedded in the thread control block (`task_struct`).

checks the status word when attempting to add a new shadow object to its workset and checks it before commit.

The `tx_start_time` field of the `transaction` object is used by the contention manager (see Section 3.2.2), while the `retry_count` field stores the number of times the transaction aborted. The `flags` field stores the flags passed to `sys_xbegin()` (Table 2.1).

The `transaction` object includes a `wait_queue` for threads that lose a conflict to this transaction. Rather than immediately restart, threads that lose a conflict use this queue to deschedule themselves until the winning transaction commits. When a transaction commits, it reschedules all of the tasks waiting on this queue.

Transaction workset. The transaction workset is tracked in three lists, the `object_list`, `list_list`, and `data_writer_list`. The `object_list` and `list_list` are skip lists [Pug90] that store references to all of the objects for which the transaction has private copies. Each object in these skip lists must be locked during commit to synchronize the committed updates. To avoid deadlock, these objects are locked according to a kernel locking discipline. The skip lists are sorted by the kernel locking discipline for a faster commit, as the lists are only traversed during commit. Because kernel lists are locked after kernel objects to allow a sensible kernel

locking discipline, they are stored on a separate skip list (Section 3.3.1).

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

The `data_writer_list` tracks the order files were written in the course of the transaction. TxOS uses this bookkeeping to retire writes to the underlying file system in roughly the same order they were issued during the transaction. Ordered write-back implements a simple block allocation heuristic based on the assumption that later reads of newly-allocated blocks are likely to follow the same pattern. Ordered write-back is only a performance optimization; it is not necessary for correctness.

Thread checkpointing. If a transactional system call reaches a point where it cannot complete because of a conflict with another thread, it must immediately abort execution. This abort is required because Linux cannot safely follow pointers if it does not have a consistent view of memory. For instance, if a thread is traversing a list in the kernel and reads a bad `next` pointer, it could overwrite arbitrary memory or get into an infinite loop. While a managed language might better detect and handle bad memory accesses than C, following inconsistent pointers in any language could result in infinite loops or inconsistent results.

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 added to the `task_struct`. If the system aborts the transaction midway through a system call, it restores the register state and jumps back to the top of the kernel stack (like the C library function `longjmp`). Because a transaction can hold a lock or other resource when it aborts, supporting the `longjmp`-style abort involves a small overhead to track certain events within a transaction so that they can be cleaned up on abort.

Deferred and logged operations. Transactions must defer certain operations until commit time, such as freeing memory, delivering signals and file system monitoring events (i.e., `inotify` and `dnotify`). The `deferred_ops` field stores these events in the `task_struct`. Similarly, some operations must be undone if a transaction is aborted, such as releasing the locks it holds and freeing the memory it allocates. These operations are stored in the `undo_ops` field.

3.3.1 Multi-process transactions

A dominant paradigm for UNIX application development is the composition of simple but powerful utility programs into more complex tasks. Following this pattern, applications may wish to transactionally fork a number of child processes to execute utilities and wait for the results to be returned through a pipe.

To support this programming paradigm in a natural way, TxOS allows multiple threads to participate in the same transaction. The threads in a transaction may share an address space, as in a multi-threaded application, or the threads may reside in different address spaces. Threads in the same transaction share and synchronize access to speculative state.

When a process forks a child inside a transaction, the child process executes within the active transaction until it performs a `sys_xend()` or it exits (where an exit is considered an implicit `sys_xend()`). The transaction commits when all tasks in the transaction have issued a `sys_xend()`. There is no transactional isolation be-

tween threads within a transaction—all threads share speculative OS state. In this method of process management, transactional programs call high-level convenience functions, like `system`, and easily create processes using the full complement of shell functionality. Transactionally forked programs run with transactional semantics, though they might not contain any explicitly transactional code. After a child process commits, it is no longer part of the transaction and subsequent `sys_xbegin()` calls will begin transactions that are completely independent from the parent.

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

The TxOS prototype does not provide an API for a thread to join a transaction, but one could be added if needed. Currently, only newly spawned threads can be added to a transaction, as this is sufficient for the applications we have studied. Similarly, TxOS does not isolate nested transactions, as our workloads so far have not required sophisticated nesting. Isolated nesting may be a requirement to safely compose a multi-process transaction from a multi-threaded application that relies on multiple, isolated transactions for safety.

Multi-process transaction state. The `transaction` object includes several fields to coordinate commit of a multi-process transaction. The `task_count` tracks how many threads are still active in the transaction. When a thread `exit`s or issues an `sys_xend()` system call, it does an atomic decrement-and-return on this counter. If the decrement returns one, it commits or aborts the transaction.

If a thread is waiting on a sibling to end the transaction, it wait on the `sibling_threads` wait queue. The thread that ends the transaction iterates over the `tasks` list and issues the deferred and undo operations of its siblings. Tasks can fork and exit within a transaction; if a task exits, its deferred and undo operations

are "orphaned" onto the shared `transaction` structure. Thus, these operations are also processed by the terminating thread.

List locking discipline

The Linux kernel locks objects according to kernel virtual address; TxOS augments this to lock objects by the kernel virtual address of the header, followed by lists, also sorted by kernel virtual address. Lists are locked last to maintain a tractable locking discipline within the directory traversal code.

Recall that TxOS replaces coarse-grained locks on lists with fine-grained list locks. Linux's tangled use of lists complicates any attempt to globally order locks on individual lists. The Linux file system code commonly traverses a list to find and lock an intermediate object, and then follows a reference from the intermediate object to another list, and so on. A straightforward approach to fine-grained locking would be to use hand-over-hand locking on the lists; unfortunately there are multiple traversal paths through these lists that can deadlock with each other. Moreover, one needs a simple enough discipline that transaction commit can avoid deadlock with the list-traversal code. Ordering list and object locks by kernel virtual address is a simple solution for commit, but in some traversal paths it could require a list to be locked before the intermediate object one needs to follow to find the list in the first place. Converting a complex body of list code originally written with coarse-grained locks to fine-grained locking will likely introduce complexity on some code paths.

The list locking discipline in TxOS attempts to minimize the locking complexity by following the structure of the list traversal code as much as possible. The high-level strategy is to sort list locks by kernel virtual address and lock lists after other kernel objects. There are few places where multiple lists need to be simultaneously locked, and in that code it is easy to sort the lists by kernel virtual address. In the traversal code, one locks a list, and increments the reference count of

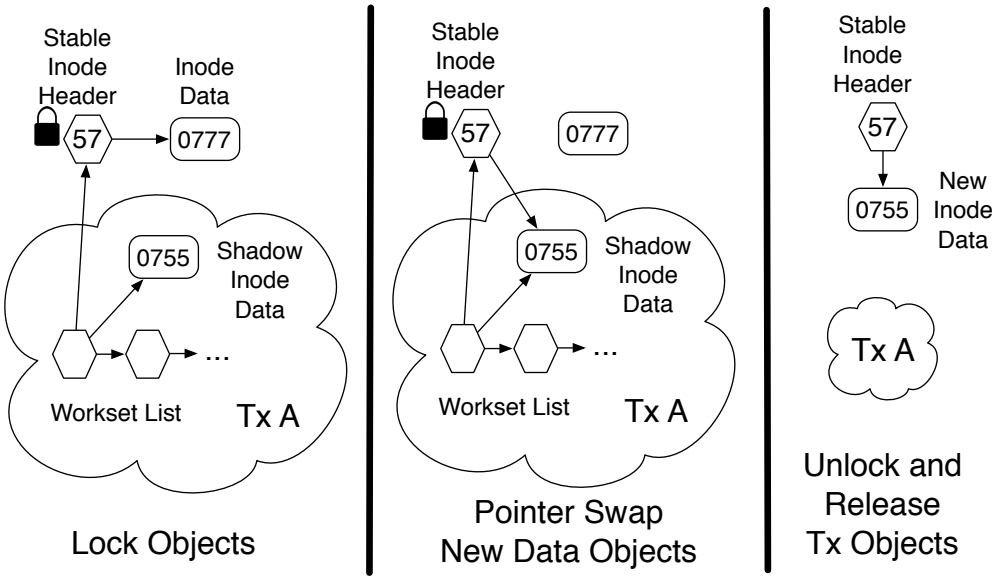

Figure 3.5: The major steps involved in committing Transaction A with inode 57 in its workset, changing the mode from 0777 to 0755. The commit code first locks the inode. It then replaces the inode header's data pointer to the shadow inode. Finally, Transaction A frees the resources used for transactional bookkeeping and unlocks the inode.

the intermediate object to prevent it from being garbage collected. The first list is then unlocked, and the intermediate object is locked, followed by the next list. The intermediate object is then unlocked and its reference count decremented, and the pattern repeats. This modified hand-over-hand locking protocol matches the structure of the Linux traversal code, minimizing the fine-grained locking complexity of both the traversal code and transaction commit.

3.4 Commit protocol

When a system transaction calls `sys_xend()`, it is ready to begin the commit protocol. The flow of the commit protocol is shown in Figure 3.5. In the first step, the transaction acquires locks for all items in its workset. The workset is kept sorted according to the kernel locking discipline to enable fast commit and eliminate the

possibility of deadlock among committing transactions.

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

After acquiring all locks, the transaction does a final check of its status word with an atomic compare-and-swap instruction. If it has not been set to ABORTED, then the transaction can successfully commit. This CAS instruction is the transaction's linearization point [HW90]. The committing process holds all relevant object locks during commit, thereby excluding any transactional or non-transactional threads that would compete for the same objects.

After acquiring all locks, the transaction copies its updates to the stable objects. The transaction's bookkeeping data are removed from the objects, then the locks are released. Between releasing spinlocks and mutexes, the transaction performs deferred operations (like memory allocations/frees and delivering file system monitoring events) and performs any pending writes to stable storage.

During commit, TxOS holds locks that are not otherwise held at the same time in the kernel. As a result, TxOS extends the locking discipline slightly, for instance by requiring that `rename` locks inodes in order of kernel virtual addresses. TxOS also introduces additional fine-grained locking on objects, such as lists, that are not locked in Linux. Although these additional constraints complicate the locking discipline, TxOS uses them to elide coarse-grained locks such as the `dcache_lock`, which protects updates to the hash table of directory entries cached in memory. By eliminating these coarse-grained locks, TxOS improves performance scalability for individual system calls.

3.5 Abort Protocol

If a transaction detects that it lost a conflict, it must abort. The abort protocol is similar to the commit protocol, but simpler because it does not require all objects to be locked at once. If the transaction is holding any kernel locks, it first releases them to avoid stalling other processes. The transaction then iterates over its working set and locks each object, removes any references to itself from the object's transactional state, and then unlocks the object. Next, the transaction frees its shadow objects and decrements the reference count on their stable counterparts. The transaction walks its undo log to release any other resources, such as memory allocated within the transaction.

3.6 Impact of data structure changes

The largest source of lines changed in TxOS comes from splitting objects such as inodes into multiple data structures (Section 3.1.2). After a small amount of careful design work in the headers, most of the code changes needed to split objects were rather mechanical.

A good deal of design effort went into assessing which fields might be modified transactionally and must be placed in the data object, and which can remain in the header, including read-only data, kernel-private bookkeeping, or pointers to other data structures that are independently versioned.

A second design challenge was assessing when a function should accept a header object as an argument and when it should accept a data object. The checks to acquire a data object are relatively expensive and would ideally occur only once per object per system call. Thus, once a system call path has acquired a data object, it would be best to pass the data object directly to all internal functions rather than reacquire it. For example, when the path name resolution code initially

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

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

Figure 3.6: Pseudo-code for the hook used to acquire an inode's data object, and an example of its use in code.

acquires shadow data objects, it then passes these shadow objects directly to helper functions such as `vfs_link` and `vfs_unlink`. In making a data object pointer a function argument, one must carefully evaluate whether the object is used in common case, lest one add objects to the transaction workset needlessly. For instance, the filesystem-specific `ioctl` routine takes a pointer to the `inode_header`, not the `inode_data` object, as many opcodes do not use any fields in the `inode_data`, nor does the parent call (`do_ioctl`).

Once the function signatures and data structure definitions are in place, the remaining work is largely mechanical. The primary change that must be propagated through the code is replacing certain pointer dereferences with hooks (Figure 3.6), so that TxOS can redirect requests for a data object to the transaction's private copy where appropriate. It is in this hook code where TxOS checks for conflicts between transactions. By encapsulating this work in a macro, we hide much of the complexity of managing private copies from the rest of the kernel code, reducing the chances for error.

A benefit of changing the object definitions is that it gives us confidence in the completeness of our hook placement. In order to dereference a field that can be modified in a transaction, the code must acquire a reference to a data object through the hook function. If the hook is not placed properly, the code will not compile. A question for future work is assessing to what degree these changes can be automatically applied during compilation using a tool like CIL [NMRW02]. This "header crawl" technique leads to more lines of code changed, but increases our confidence that the changes were made throughout the large code base that is the Linux kernel.

3.7 Integration with transactional memory

System transactions protect system state, not application state. For multi-threaded programs, the OS has no efficient mechanism to save and restore the memory state of an individual thread. User-level transactional memory (TM) systems, however, are designed to provide efficient transactional semantics to memory modifications by a thread, but cannot isolate or roll back system calls. Integrating user and system transactions creates a simple and complete transactional programming model.

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

On a TM system integrated with TxOS, when a TM application makes a system call, the runtime begins a system transaction. The user-level transactional memory system handles buffering and possibly rolling back the application's memory

state, and the system transaction buffers updates to system state. The updates to system state are committed or aborted by the kernel atomically with the commit or abort of the user-level transaction. The programmer sees the simple abstraction of an atomic block that can contain updates to user data structures and system calls.

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

3.7.1 Lock-based STM requirements

TxOS uses a simplified variant of the two-phase commit protocol (2PC) [Gra78] to coordinate commit of a lock-based user-level software (STM) transaction with a system transaction. The TxOS commit consists of the following steps.

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

This protocol naturally follows the flow of control between the user and kernel, but requires the user transaction system to support the prepared state. We define a prepared transaction as being finished (it will add no more data to its working set), safe to commit (it has not currently lost any conflicts with other threads), and guaranteed to remain able to commit (it will win all future conflicts until the end of the protocol). In other words, once a transaction is prepared,

another thread must stall or rollback if it tries to perform a conflicting operation. In a system that uses locks to protect a commit, prepare is accomplished by simply holding all of the locks required for the commit during the `sys_xend()` call. On a successful commit, the system commits its state before the user, but any competing accesses to the shared state are serialized after the user commit.

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

3.7.2 HTM and obstruction-free STM requirements

Hardware transactional memory (HTM) and obstruction-free software transactional memory systems [HLMS03] use a single instruction (`xend` and compare-and-swap, respectively), to perform their commits. For these systems, a prepare stage is unnecessary. Instead, the commit protocol should have the kernel issue the commit instruction on behalf of the user application after the kernel has locked and validated its workset but before committing any kernel state. Both the system and user-level transaction commit or abort together depending upon the result of this specific commit instruction.

To safely integrate HTM with TxOS, the hardware must suspend a user-level hardware transaction on entry to the kernel, or allow the kernel to suspend it. Every HTM proposal that supports an OS [MBM+06a, RHP+07, ZB06] supports mechanisms that suspend user-initiated transactions, avoiding the mixture of user and kernel addresses in the same hardware transaction. Mixing user and kernel

address in a hardware transaction creates a security vulnerability in most HTM proposals. For HTM integration, transactional pause or escape is sufficient for TxOS as long as the kernel can ensure that the transaction is properly suspended when entering the kernel. The kernel must also be able to issue an `xend` instruction on behalf of the application.

Though TxOS supports user-level HTM, it runs on commodity hardware and does not require any special HTM support itself.

Chapter 4

TxOS Kernel Subsystems

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

In general, the effort required to transactionalize a subsystem is proportional to the number of data structures involved in servicing requests. In many cases, this effort involves a single additional data structure, as most devices drivers, IPC abstractions, etc. store their state in a single opaque pointer attached to the inode for the file which represents the object to users. Even complex systems, such as the virtual file system or the memory management system have fewer than 10 data structures to transactionalize, although their interactions can be complex.

4.1 Virtual file system

In the original UNIX design, files are the primary abstraction for all resources [RT74]; Linux inherits files as a core abstraction. Many kernel subsystems are accessed through the file system interface (e.g., `read`, `write`, `open`). In the discussion of supported system calls in Section 2.2 and listed in Table 2.2, many file system calls will be partially supported essentially forever for this reason, and the effort to add complete transactional support for the tail of system calls is disproportionately high.

Because the virtual file system interface is so pervasive throughout the kernel, many examples in Chapter 3 also use VFS structures, such as `inodes` and `dentries`. This section addresses features of the VFS not previously described, namely how file data is buffered, and the interaction between the VFS and specific file systems, such as `ext3`.

4.1.1 Transactional file data access

Accesses to file data in a transaction are isolated at byte granularity according to a standard single-writer, multiple-reader compatibility matrix. Thus, transactions may concurrently write to disjoint regions of the same file. Written data is buffered in main memory until commit; if the transaction is durable and the file is backed by stable storage, the data is guaranteed to be written to disk before commit returns. If the underlying file system can guarantee atomic updates of stable storage, for instance by using a journal in `ext3` [Twe], then writes will be atomic. Many file systems support atomic updates to stable storage [Bes, Kur, SDH+96], although some do not (e.g., `ext2`).

File data in Linux is stored in the page cache, which serves the larger purpose of tracking the allocation of each page of RAM. In Linux, each file's `inode` includes an `address_space` object, called `i_data`, which is in the `inode_header` in TxOS. The `address_space` structure stores a sparse logical mapping of offsets to pages of

physical memory using a radix tree [Mor68]. In servicing a `read` or `write`, if a page storing a given file offset is not found in the radix tree, the block is fetched from disk. Note that pages can be added and removed from the radix tree independently of any system call based on caching heuristics, including file read-ahead and least-recently-used reclamation.

TxOS adds a second radix tree, called the shadow tree, to the address space object. Data written by transactions is versioned in the shadow tree. When a transaction reads data, it first checks the shadow tree for a page, and if the page is not found, the transaction looks in the main radix tree. All transactions share a single shadow tree per file, which stores speculatively written data.

In order to isolate byte ranges within the shared shadow tree, TxOS uses range locks to track shared and exclusive ownership of file regions. Range locks specify a contiguous file region, access mode, and transaction owner. In each address space, TxOS adds a skip list for shared (read) range locks and exclusive (write) range locks, sorted by file offset. Because skip lists can be searched in time logarithmic to their size, checks of a given file range are efficient.

If a conflict is identified for a region and the owner of an existing range lock loses the conflict, the range lock can be evicted, effectively invalidating its pages in the shadow tree. When the winner allocates a new range lock, the winner is responsible for re-copying committed state from the main radix tree. If the winner needs a smaller range than the evicted range lock such that a page in the shadow tree is no longer needed, the winner is responsible for freeing it. Because disjoint range locks can occupy the same page, pages in the shadow tree contain a reference count. When a transaction aborts, it checks each file it has written for range locks that have not been evicted, and frees the remaining range locks along with any pages in the shadow tree for which it is removing the last reference.

If a transactional write covers an entire page, when the transaction is com-

mitted the page is simply moved to the main radix tree and frees the old contents. For writes smaller than a page, the updated regions are copied to the page in the main radix tree and the reference count on the shadow tree page is lowered. Although `memcpy` is a relatively slow operation, this simple approach obviates the need to ensure that the remaining bytes of the page remain up to date. Once a region of committed data is moved to the pages in the main radix tree, a write is issued to the underlying file system, scheduling the buffers for write back to the disk.

4.1.2 Transactional file systems

The VFS layer in Linux provides a common interface to dozens of different specific file systems, which provide various features including network transparency, journaling, and data layout strategies tuned for specific physical media. In order for transactions to become a ubiquitous feature in this environment, the cost of adoption for a specific filesystem must be relatively small.

TxOS simplifies the task of writing a transactional file system by detecting conflicts and managing versioned data in the virtual filesystem layer. The OS provides the transactional semantics—versioning updates and detecting conflicts. The primary responsibility of a specific file system is to provide the ability to atomically commit updates to stable storage (e.g., via a journal or with versioning). This is a good balance of responsibilities for implementing transactional file systems, as many file systems already provide atomic disk updates.

As a case study, we added transactions to the `ext3` file system. The `ext3` file system uses a journal to ensure that each system call is atomically committed to disk in a journal transaction.[1] Once a journal transaction is committed, updates may be written to the file system. After a failure, the journal can be replayed to

[1]The `ext3` journal typically batches all system calls within a 5 second window, unless there is an explicit `sync()` system call. The journal is typically configured to only record metadata, allowing partial updates to file data on disk in the event of a failure.

Object	Function Name	Description
inode	init_tx	Initialize fs-specific extensions in a private copy.
	lock	Acquire locks for fs-specific fields.
	unlock	Release locks for fs-specific fields.
	commit	Commit fs-specific fields.
	abort	Perform any cleanup of fs-specific fields during rollback.
	validate	Compare fs-specific fields in the shadow and stable object to a checkpoint, validating the transition is valid. Called in a debug build only.
super_block	commit	Commit fs-specific fields.
	abort	Perform any cleanup of fs-specific fields during rollback.
	validate	Compare fs-specific fields in the shadow and stable object to a checkpoint, validating the transition is valid. Called in a debug build only.
address_space	prepare_tx_write	Update fs bookkeeping on a buffered transactional write.

Table 4.1: File system hooks added by TxOS.

ensure that journaled updates are present in the file system. In TxOS, the `ext3` journal interface is slightly augmented to ensure that disk updates from a system transaction are written in a single journal transaction.

In addition to ensuring atomic updates to disk, file system code must be converted to use the new split data types by appropriately placing the hooks described in Section 3.6. This work is quite mechanical. For example, in-memory file systems, including `proc` and `tmpfs`, were made transactional with very minimal code changes.

TxOS provides additional hooks into VFS code to a file system, summarized in Table 4.1. File systems commonly extend the `inode` and `super_block` with additional fields, largely to manage the mapping of the object to disk. The additional methods TxOS provides on these objects give the file system an opportunity to

perform any initialization, commit, or rollback that cannot be accomplished with a simple `memcpy`. Note that file systems approximate sub-classing of objects in C by having each object store a pointer to a table of function pointers, which is initialized by the file system. These pointers can be NULL if they are not required.

The final hook we provide is the `prepare_tx_write` function on the `address_space`, which is called when a transactional write is buffered in the page cache. This hook was introduced for the `ext3` journal, which requires some rather detailed bookkeeping about expected writes at the start of a journal transaction. In general, this hook is also expected to give any file system block pre-allocation algorithms visibility into buffered writes.

Supporting multiple file systems. TxOS does not currently commit updates to multiple, non-volatile file systems atomically. Commits could span multiple file systems in future work by adopting a form of two-phase commit for the disks [Gra78]. Taking a journaling file system as an example, each system transaction would provide a unique identifier for the transaction and a universally unique identifier for the file system[2], that would be recorded in the journal transaction. The file system should not allow subsequent journal transactions until all file systems have committed their journal entry. After a system failure, each file system would only replay this journal transaction only if all other participants have a committed transaction in the journal with the same identifier.

4.1.3 Serializable directory reads

Many applications use lock files to synchronize file system accesses; even on distributed file systems with weak consistency semantics, file creation is usually atomic.

[2]Most modern Linux systems generate a random number-based UUID for each file system when it is created. Distributions, including Ubuntu, use this to identify disk mount points in `/etc/fstab` rather than the conventional device names (e.g., `/dev/sda1`). The traditional names are based on the order the devices are enumerated by the kernel at boot, which can change if a new disk is plugged in, the BIOS or kernel is upgraded, etc.

Transactional Thread

```
sys_xbegin();
if(!exists('lockfile')){

  /* modify data file */

}
sys_xend();
```

Non-Transactional Thread

```
if(create('lockfile')) {
        /* modify data file */
       unlink('lockfile);
}
```

Figure 4.1: An example of a straightforward conversion of application code to use transactions that is only correct if file system directory reads are serialized (i.e., system transactions provide degree 3 consistency, or full serializability). System transactions in TxOS provide full serializability.

Thus, creating a "lock file" is often used as a synchronization primitive. If an application incrementally adopts transactions, transactions may need to coordinate access to a data file through a lock variable.

Because system transactions are serializable, a straightforward conversion to use transactions would simply check whether the lock file exists, and if not, modify the data file, as illustrated in Figure 4.1. This optimization is sensible because the lock file would normally be deleted before the transaction commits, so there is no reason to create it. This code is only correct if the OS detects a conflict on the *non-existence* of the lock file—the transaction requires isolation on the file name without actually creating the file.

Ensuring fully serializable reads on databases is a classic problem. Without serializable reads, a database transaction that enumerates the contents of a table twice may have newly created entries appear on the second read (called "ghosting"), as traditional implementations only acquire read locks on entries that exist at the time they are read. To eliminate ghosting and provide repeatable reads in databases, more expensive predicate locks are used to prevent concurrent updates to a table that is enumerated. Because of this expense, applications that can tolerate ghost updates often run their transactions with degree 2 isolation, which gets most of the benefits of full serializability with lower overheads.

Transactions in TxOS are serializable, also known as degree 3 isolation, ensuring that checks for lock files (as in Figure 4.1) are properly serialized. The reason directory reads are serializable is that when a lookup fails because a file does not exist, the kernel creates a *negative* dentry, which caches the fact that a file does not exist and avoids going back to disk on a second lookup. This negative dentry is added to the workset of a transaction, and attempts to create the file outside of the transaction will create a conflict on the negative dentry.

This feature was experimentally verified by writing a version of the code in Figure 4.1 that single-steps two threads with the `sleep()` function. When run without `sys_xbegin()` and `sys_xend()` on unmodified Linux, the data file is corrupted. When run with transactions on TxOS, the conflict is detected and the threads serialized. In the current prototype, the negative dentry is placed in the transaction's working set in exclusive (write) mode. The actual conflict is detected before the negative dentry is used by the non-transactional thread on the dentry hash list containing the negative dentry. Because the transaction modified the list, and the non-transactional thread must read the list to check whether the file exists in the cache, these operations are conflicting and are serialized. Because the negative dentry is pinned in memory and the dcache's hash table until commit, any attempts to create the file outside of the transaction will be detected as conflicts, providing degree 3 isolation.

4.1.4 Early release of file handles

If a file handle is created and closed in the same transaction, it is never externally visible and can be removed from the workset of the transaction. Note that the file handle only stores a reference to a file and a cursor into the file; updates to the file itself are versioned in the `inode`. In some cases, such as transactional shell scripting, freeing file handles and private resources simplifies kernel bookkeeping. Thus, TxOS

provides an early release function to remove an object from its workset, for use only in these limited circumstances.

4.2 Memory mapping

Defining useful transactional semantics for memory mapping system calls is tricky, as one must distinguish modifications to the mapping from changes to memory contents. A straightforward semantics would roll back all modifications to system state and application state if a system transaction failed. As discussed earlier, the only tool available to the OS for versioning memory contents is copy-on-write (COW) paging, which is very inefficient. Thus, TxOS generally defers management of memory contents to more efficient user-level tools, such as transactional memory. In most cases, this distinction is not important to programmers: changes to memory protection are rolled back in a straightforward manner, unshared, mapped regions are unmapped and deleted, and writes to mapped files are rolled back, as transactional accesses to mapped files operate on shadow pages (Section 4.1.1). If a transactional mapping of a shared memory buffer is rolled back, the transactionally-written contents may persist. Programmers that combine system transactions with shared memory must synchronize access to shared data structures and manually undo changes to its contents.

The approach to anonymous memory mappings in TxOS is to version modifications to the memory mapping for a process, but not to isolate or version the contents of memory. The virtual memory layout of a process is represented in the kernel by a list of `vm_area_struct`'s (vma); each vma represents a contiguous region of virtual addresses and stores the permissions and references to the data structures necessary to populate the address on a page fault (e.g., from swap, creating a new page, reading a file). Modifications to a vma by a system call, such as `mmap`, `munmap`, and `mprotect` are isolated and rolled back by standard modifications to the `vma`.

A similar approach is taken to pages mapping a file inside a transaction, except that writable pages mapping a file must be removed when the transaction begins, forcing a page fault. If a page faults inside of a transaction, a range lock at page granularity is acquired by the transaction (Section 4.1.1), and a shadow page is allocated and mapped into the process. When an update to a mapped file page is committed, the kernel invalidates the page mapping in any other processes that map the file and issues TLB shoot-down for the page.

If multiple threads share a page of memory, either in the same address space or through shared memory, they must synchronize access to data structures in the memory page using application-level techniques.

4.3 Pipes

Reads to and writes from pipes in TxOS have similar semantics to files, with the exception that transactional pipe reads only permit a single reader. This is because reads from a pipe remove data from the pipe, so the readers must also be serialized. TxOS defers destruction of read data until a transaction commits. Unless the reader and writer are part of the same transaction, a reader blocks once it consumes all committed data; data written by another transaction are not available to a reader until the writing transaction commits. As an optimization, when the creator, reader, and writer of a pipe are participants in the same transaction, this buffering can be elided.

Pipe data are stored in a `pipe_inode_info` structure, which is referenced through a pointer in the `inode_header`. In Linux, a pipe may buffer up to 16 pages of data, and a page on x86 is 4 kilobytes by default, for a total of 64 KB of buffered data. Statically limiting the amount of buffered data in Linux pipes is a simple mechanism to avoid resource exhaustion and rate limit producers that are substantially faster than consumers.

The TxOS implementation of pipes adopts a similar static limit of 16 pages of shadow buffers, which are also associated with the pipe's inode. Written data are stored in a shadow buffer, so that the written data can be removed if a transaction aborts. Data read in a transaction are not destroyed, so that the data can be restored to the pipe if the transaction aborts; TxOS instead tracks a read offset into the buffered data until the transaction ends. The static limit of 16 buffered shadow pages is sufficient for our workloads, and easily modifiable to grow dynamically.

The changes needed to implement pipe semantics are entirely localized to the `pipe_inode_info` data structure and `fs/pipe.c`.

4.4 Text console

TxOS provides simple buffering of output to text consoles, which are typically represented by `tty` devices. This feature is implemented using the deferred operations facility in a transaction. This feature can be disabled during kernel compilation to facilitate debugging of the kernel or applications—a very useful feature in our experience.

4.5 Signal delivery

Signal semantics in TxOS provide isolation among threads in different transactions, as well as isolation between non-transactional and transactional threads. Any signal sent to a thread outside of the sender's transaction is deferred until the sender commits by placing it in a queue. Signals in the outgoing queue are delivered in the order sent if the transaction commits, and discarded if the transaction aborts.

A thread in a transaction may specify whether incoming signals should be delivered. When an application begins a transaction, a flag to `sys_xbegin()` specifies whether incoming signals should be delivered speculatively within the transaction

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

When signal delivery is deferred, incoming signals are placed in a queue; when the transaction commits or aborts the signals are delivered to the thread in the order received. Deferring signals allows transactions to ensure that they are atomic with respect to signal handlers [Zal01]. Enclosing signal handling code in a transaction ensures that system calls in the handler are atomic, and forces calls to the same handler to serialize. Transactional handling of signals eliminates race conditions without the need for the additional API complexity of `sigaction`. While the `sigaction` API addresses signal handler atomicity within a single thread by making handlers non-reentrant, the API does not make signal handlers atomic with respect to other threads.

Speculative and deferred delivery apply only to delivery of incoming signals sent from threads outside of the transaction. Signals sent among threads in the same transaction are delivered immediately.

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

4.6 Future work

TxOS does not yet provide transactional semantics for several classes of OS resources. Currently, TxOS either logs a warning or aborts a transaction that attempts

to access an unsupported resource: the programmer specifies the behavior via a flag to `sys_xbegin()`. This section considers some challenges inherent in supporting these resources, which we leave for future work.

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

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

User interfaces Exchanging messages with a user while inside a transaction is unlikely to become a popular paradigm (although TABS implemented a transaction GUI by crossing out text dialogs from aborted transactions [SDD+85]), because the I/O-centric nature of user interfaces is not a natural fit with the transactional programming model. Like other communication channels, however, the OS could naturally support transactions that only read from or write to a user I/O device

by buffering the relevant data. Maintaining a responsive user interface will likely mandate that developers keep transactions involving interfaces short.

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

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

4.7 Summary

A key design goal of TxOS is that transactions should be a core kernel abstraction, with a modular implementation that encourages code reuse across subsystems. This chapter describes the implementation of transactions in several kernel subsystems. In general, each subsystem essentially implements a subclass of a more general transactional object type. This approach minimizes the implementation effort and risk of programmer error. The effort to transactionalize new code paths is generally proportional to the number of data structures involved. Further, the semantics and interfaces of each subsystem can be rather idiosyncratic, as the POSIX API has evolved rather organically. Thus, it is unlikely that one could eliminate the need for subclassing the transactional abstractions for each subsystem without a more uniform system call API.

This same approach to subclassing is also adopted in the division of responsibilities between the VFS and individual file systems. Transactional behavior of file system metadata and page mappings is largely implemented in shared VFS and page cache code, while individual file systems are required to adopt the transactional data types and provide atomic update to disk where appropriate. In designing for an Linux, which includes dozens of file systems, this modular design minimizes the transaction implementation burden on the developer of a particular file system.

Chapter 5

Evaluation

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

All of experiment are performed on a server with 1 or 2 quad-core Intel X5355 processors (total of 4 or 8 cores) running at 2.66 GHz with 4 GB of memory. The 4 core machine has a 7200 RPM SATA drive with a 3.0 GB/s link, and the 8 core machine has a 10,000 RPM SAS drive. All single-threaded experiments use the 4-core machine, and scalability measurements were taken using the 8 core machine. We compare TxOS to an unmodified Linux kernel, version 2.6.22.6—the same version extended to create TxOS.

The hardware transactional memory experiments use MetaTM [RRP+07] on Simics version 3.0.27 [MCE+02]. The simulated machine has 16 1000 MHz CPUs, each with a 32 KB level 1 and 4 MB level 2 cache. An L1 miss costs 24 cycles and an L2 miss costs 350 cycles. The HTM uses the timestamp contention management policy and linear backoff on restart.

Call	Linux	Base		Static		NoTx		Bgnd Tx	
access	2.4	2.4	1.0×	2.6	1.1×	3.2	1.4×	3.2	1.4×
stat	2.6	2.6	1.0×	2.8	1.1×	3.4	1.3×	3.4	1.3×
open	2.9	3.1	1.1×	3.2	1.2×	3.9	1.4×	3.7	1.3×
unlink	6.1	7.2	1.2×	8.1	1.3×	9.4	1.5×	10.8	1.7×
link	7.7	9.1	1.2×	12.3	1.6×	11.0	1.4×	17.0	2.2×
mkdir	64.7	71.4	1.1×	73.6	1.1×	79.7	1.2×	84.1	1.3×
read	2.6	2.8	1.1×	2.8	1.1×	3.6	1.3×	3.6	1.3×
write	12.8	9.9	0.7×	10.0	0.8×	11.7	0.9×	13.8	1.1×
geomean			1.03×		1.14×		1.29×		1.42×

Call	Linux	In Tx		Tx	
access	2.4	11.3	4.7×	18.6	7.8×
stat	2.6	11.5	4.1×	20.3	7.3×
open	2.9	16.5	5.2×	25.7	8.0×
unlink	6.1	18.1	3.0×	31.9	7.3×
link	7.7	57.1	7.4×	82.6	10.7×
mkdir	64.7	297.1	4.6×	315.3	4.9×
read	2.6	11.4	4.3×	18.3	7.0×
write	12.8	16.4	1.3×	39.0	3.0×
geomean			3.93×		6.61×

Table 5.1: Execution time in thousands of processor cycles of common system calls on TxOS and performance relative to Linux. **Base** is the basic overhead introduced by data structure and code modifications moving from Linux to TxOS, without the overhead of transactional lists. **Static** emulates compiling two versions of kernel functions, one for transactional code and one for non-transactional code, and includes transactional list overheads. These overheads are possible with compiler support. **NoTX** indicates the current speed of non-transactional system calls on TxOS. **Bgnd Tx** indicates the speed of non-transactional system calls when another process is running a transaction in the background. **In Tx** is the cost of a system call inside a transaction, excluding `sys_xbegin()`and `sys_xend()`, and **Tx** includes these system calls.

5.1 Single-thread system call overheads

A key goal of TxOS is efficient transactions, taking special care to minimize the overhead incurred by non-transactional applications that are not using transactions. To evaluate performance overheads for substantial applications, we measured the average compilation time across three non-transactional builds of the Linux 2.6.22 kernel on unmodified Linux (3 minutes, 24 seconds), and on TxOS (3 minutes, 28 seconds). Linux compilation is CPU bound by user-space computations, spending well below 10% of its time in the kernel or blocked on disk I/O. Like many applications, system calls in Linux compilation account for a small portion of the execution time and

the application is rather insensitive to system call overheads. This slowdown of less than 2% indicates that for most applications, the non-transactional overheads will be negligible. At the scale of a single system call, however, the average overhead is currently 29%, and could be cut to 14% with improved compiler support.

Table 5.1 shows the performance of common file system system calls on TxOS. We ran each system call 1 million times, discarding the first and last 100,000 measurements and averaging the remaining times. The elapsed cycles were measured using the `rdtsc` instruction. The purpose of the table is to analyze transaction overheads in TxOS, but it is not a realistic use case, as most system calls are already atomic and isolated. Wrapping a single system call in a transaction is the worst case for TxOS performance because there is very little work across which to amortize the cost of creating shadow objects and commit.

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

TxOS replaces simple linked lists with a more complex transactional list (§3.2.4). The transactional list allows more concurrency, both by eliminating transactional conflicts and by introducing fine-grained locking on lists, at the expense of higher single-thread latency. The **Static** column adds the latencies due to transactional lists to the base overheads (roughly 10%, though more for `link`).

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

sive. With compiler support, these overheads (geometric mean of 14%) are achievable. The dynamic checks of the current prototype account for half of the current non-transactional system call overheads (represented by the **NoTx** column, average of 29%).

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

5.2 Applications and micro-benchmarks

Table 5.2 shows the performance of TxOS on a range of applications and micro-benchmarks. Each measurement is the arithmetic mean of three runs. The slowdown relative to Linux is also listed. Postmark is a file system benchmark that simulates the behavior of an email, network news, and e-commerce client. We use version 1.51 with the same transaction boundaries as Amino [WSSZ07]. The LFS small file benchmark operates on 10,000 1024 bytes files, and the large file benchmark reads and writes a 100MB file. The Reimplemented Andrew Benchmark (RAB) is a reimplementation of the Modified Andrew Benchmark, scaled for modern computers. Initially, RAB creates 500 files, each containing 1000 bytes of pseudo-random printable-ASCII content. Next, the benchmark measures execution time of four distinct phases: the `mkdir` phase creates 20,000 directories; the `cp` phase copies the 500 generated files into 500 of these directories, resulting in 250,000 copied files; the `du` phase calculates the disk usage of the files and directories with the `du` command;

and the `grep`/`sum` phase searches the files for a short string that is not found and checksums their contents. The sizes of the `mkdir` and `cp` phases are chosen to take roughly similar amounts of time on our test machines. In the transactional version, each phase is wrapped in a transaction. Make wraps a software compilation in a transaction. Dpkg and Install are software installation benchmarks that wrap the entire installation in a transaction, as discussed in the following section.

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

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

Microbenchmark	**Linux ext2**	**TxOS ACI**		**Linux ext3**	**TxOS ACID**	
postmark	38.0	7.6	0.2×	180.9	154.6	0.9×
lfs small						
create	4.6	0.6	0.1×	10.1	1.4	0.1×
read	1.7	2.2	1.2×	1.7	2.1	1.3×
delete	0.2	0.4	2.0×	0.2	0.5	2.4×
lfs large						
write seq	1.4	0.3	0.2×	3.4	2.0	0.6×
read seq	1.3	1.4	1.1×	1.5	1.6	1.1×
write rnd	77.3	2.6	0.03×	84.3	4.2	0.05×
read rnd	75.8	71.8	0.9×	70.1	70.2	1.0×
RAB						
mkdir	8.7	2.3	0.3×	9.4	2.2	0.2×
cp	14.2	2.5	0.2×	13.8	2.6	0.2×
du	0.3	0.3	1.0×	0.4	0.3	0.8×
grep/sum	2.7	3.9	1.4×	4.2	3.8	0.9×
geomean			.4×			.5×
Application	**Linux ext2**	**TxOS ACI**		**Linux ext3**	**TxOS ACID**	
dpkg	.8	.9	1.1×	.8	.9	1.1×
make	3.2	3.3	1.0×	3.1	3.3	1.1×
install	1.9	2.7	1.4×	1.7	2.9	1.7×
geomean			1.2×			1.3×

Table 5.2: Execution time in seconds for several transactional benchmarks on TxOS and slowdown relative to Linux. ACI represents non-durable transactions, with a baseline of ext2, and ACID represents durable transactions with a baseline of ext3 with full data journaling.

overhead comes primarily from the kernel slab allocator, which allocates 2.4× as much memory. The slab allocator is used for general allocation (via `kmalloc`) and for common kernel objects, like inodes. TxOS's memory use indicates that buffering transactional updates in memory is practical, especially considering the trend in newer systems toward larger DRAM and 64-bit addresses.

5.3 Software installation

By wrapping system commands in a transaction, we extend `make`, `make install`, and `dpkg`, the Debian package manager, to provide ACID properties to software installation. We first test `make` with a build of the text editor nano, version 2.0.6. Nano consists of 82 source files totaling over 25,000 lines of code. Next, we test `make install` with an installation of the Subversion revision control system, version 1.4.4. Finally, we test `dpkg` by installing the package for OpenSSH version 4.6. The OpenSSH package was modified not to restart the daemon, as the script responsible sends a signal and waits for the running daemon to exit, but TxOS defers the signal until commit. This script could be rewritten to match the TxOS signal API in a production system.

As Table 5.2 shows, the overhead for adding transactions is quite reasonable (1.1–1.7×), especially considering the qualitative benefits. For instance, by checking the return code of dpkg, our transactional wrapper was able to automatically roll back a broken Ubuntu build of OpenSSH (4.6p1-5ubuntu0.3), and no concurrent tasks were able to access the invalid package files during the installation.

5.4 Solid state drive measurements

Because disk scheduling has a significant effect on the performance measurements collected above, we measure the overheads of using TxOS on a storage medium with lower latency, namely a solid state drive (SSD). We replaced the root disk in the test machine with an 80GB Intel X25-M drive, which contains MLC NAND flash. We re-ran the microbenchmarks and application benchmarks, excepting compilation of nano, which executed too quickly to differentiate significantly. Attempting to follow best practices for solid state drives, we mounted the root file system with the `noatime` option, selected the deadline scheduler, and batched writes. We also only

Microbenchmark	**Linux ext2**	**TxOS**	**ACI**
postmark	9.8	6.7	0.7×
lfs small			
create	2.3	0.5	0.1×
read	1.2	2.8	2.3×
delete	0.1	0.3	3.0×
lfs large			
write seq	1.3	1.6	1.2×
read seq	0.6	0.5	1.2×
write rnd	6.5	4.8	0.7×
read rnd	3.5	3.6	1.0×
RAB			
mkdir	7.7	1.1	0.1×
cp	3.7	2.5	0.7×
du	0.3	0.3	1.0×
grep/sum	1.9	3.1	1.6×
geomean			.8×
Application	**Linux ext2**	**TxOS**	**ACI**
dpkg	.6	.8	1.3×
install	1.7	2.9	1.7×
geomean			1.5×

Table 5.3: Execution time in seconds for several transactional benchmarks on TxOS and slowdown relative to Linux on a Solid State Drive. ACI represents non-durable transactions, with a baseline of ext2.

ran the `ext2` benchmarks to avoid noise introduced by journaling. Table 5.3 lists the measurements collected from the SSD, again the arithmetic mean of three runs.

Overall, the overheads are comparable and slightly higher—geometric mean overheads of .4 and 1.2 on a SATA drive, compared to .6 and 1.5. This is expected, since we are losing the dramatic effects of improved disk scheduling discussed above. For instance, the speedup of the LFS large random write phase drops from 29.7× on a SATA drive to 35% on the SSD. While better disk scheduling can improve the performance of TxOS on a traditional hard drive, the overheads remain acceptable on a storage medium with different performance characteristics.

5.5 Transactional ext3

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

5.6 Eliminating race attacks

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

We downloaded the symlink TOCTTOU attacker code used by Borisov et al. [BJSW05] to defeat Dean and Hu's probabilistic countermeasure [DH04]. This attack code creates memory pressure on the file system cache to force the victim to deschedule for disk I/O, thereby lengthening the amount of time spent between checking the path name and using it. This additional time allows the attacker to win nearly every time on Linux.

On TxOS, the victim successfully resists the attacker by reading a consistent view of the directory structure and opening the correct file. The attacker's attempt to interpose a symbolic link creates a conflicting update that occurs after the transactional `access` check starts, so TxOS puts the attacker to sleep on the asymmetric conflict. The performance of the safe victim code on TxOS is statistically indistin-

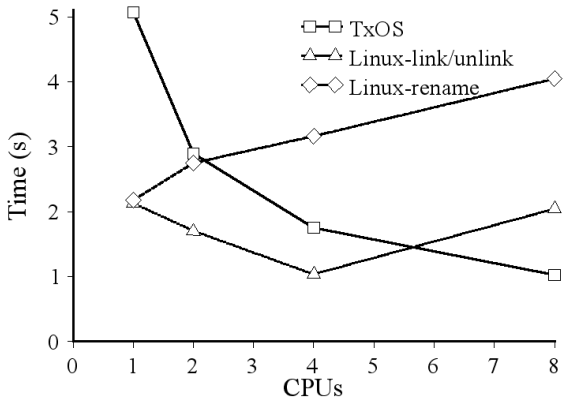

Figure 5.1: Time to perform 500,000 renames divided across a number of threads (lower is better). TxOS implements its renames as calls to `sys_xbegin()`, `link`, `unlink`, and `sys_xend()`, using 4 system calls for every Linux `rename` call. Despite higher single-threaded overhead, TxOS provides better scalability, outperforming Linux by 3.9× at 8 CPUs. At 8 CPUs, TxOS also outperforms a simple, non-atomic `link`/`unlink` combination on Linux by 1.9×.

guishable from the vulnerable victim on Linux.

To demonstrate that TxOS improves robustness while preserving simplicity for signal handlers, we reproduced two of the attacks described by Zalewksi [Zal01]. The first attack is representative of a vulnerability present in `sendmail` up to 8.11.3 and 8.12.0.Beta7, in which an attacker induces a double-free in a signal handler. The second attack, representative of a vulnerability in the `screen` utility, exploits lack of signal handler atomicity. Both attacks lead to root compromise; the first can be fixed by using the `sigaction` API rather than `signal`, while the second cannot. We modified the signal handlers in these attacks by wrapping handler code in a `sys_xbegin, sys_xend` pair, which provides signal handler atomicity without requiring the programmer to change the code to use `sigaction`. In our experiments, TxOS serializes handler code with respect to other system operations, and therefore defeats both attacks.

5.7 Concurrent performance

System calls like `rename` and `open` have been used as *ad hoc* solutions for the lack of general-purpose atomic actions. These system calls have strong semantics (a rename is atomic within a file system), resulting in complex implementations whose performance does not scale. As an example in Linux, `rename` has to serialize all cross-directory renames with a single file-system-wide mutex, as finer-grained locking would risk deadlock. The problem is not that performance tuning `rename` is difficult, but it would substantially increase the implementation complexity of the entire file system, including unrelated system calls.

Transactions allow the programmer to combine simpler system calls to perform more complex operations, yielding better performance scalability and a simpler implementation. Figure 5.1 compares the unmodified Linux implementation of `rename` to calling `sys_xbegin()`, `link`, `unlink`, and `sys_xend()` in TxOS. In this micro-benchmark, we divide 500,000 cross-directory renames across a number of threads.

TxOS has worse single-thread performance because it makes four system calls for each Linux system call. But TxOS quickly recovers the performance, performing within 6% at 2 CPUs and out-performing `rename` by 3.9× at 8 CPUs. The difference in scalability is directly due to TxOS using fine-grained locking to implement transactions, whereas Linux must use coarse-grained locks to maintain the fast path for `rename` and keep its implementation complexity reasonable. While this experiment is not representative of real workloads, it shows that solving consistency problems with modestly complex system calls like `rename` will either harm performance scalability or introduce substantial implementation complexity. Because of Linux's coarse-grained locks, TxOS' atomic `link`/`unlink` pair outperforms the Linux non-atomic `link`/`unlink` pair by a factor of 1.9× at 8 CPUs.

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

Table 5.4: Execution Time, number of system calls, and allocated pages for the genome benchmark on the MetaTM HTM simulator with 16 processors.

5.8 Integration with software TM

We qualitatively verify that system transactions can be integrated with existing transactional memory systems by extending a software and hardware TM implementation to use system transactions. We integrated DATM-J [RRHW09], a Java-based STM, with system transactions. The only modifications to the STM are to follow the commit protocol when committing a user level transaction that invokes a system call and to add return code checks for aborted system transactions, as outlined in Section 3.7.

We tested the integration of DATM-J with TxOS by modifying Tornado, a multi-threaded web server that is publicly available on sourceforge, to use transactions. Tornado protects its data structures with STM transactions, and the STM transparently protects concurrent reads and writes to its data files from interfering with each other. The original code uses file locking. For one synthetic workload, the STM version is 47% faster at 7 threads, attributable to concurrent file system accesses. This result is not necessarily representative of all STM workloads, as many may not benefit from optimistic parallelism in both the application and OS, but rather a proof-of-concept that STM transactions can inter-operate with system transactions.

5.9 Integration with hardware TM

In the genome benchmark from the Stanford STAMP transactional memory benchmark suite [MCKO08], the lack of integration between the hardware TM system and the operating system results in an unavoidable memory leak. Genome allocates memory during a transaction, and the allocation sometimes calls `mmap`. When the transaction restarts, it rolls back the allocator's bookkeeping for the `mmap`, but not the results of the `mmap` system call, thereby leaking memory. When the MetaTM HTM system [RRP+07] is integrated with TxOS, the `mmap` is made part of a system transaction and is properly rolled back when the user-level transaction aborts.

Table 5.4 shows the execution time, number of system calls within a transaction, and the number of allocated pages at the end of the benchmark for both TxOS and unmodified Linux running on MetaTM. TxOS rolls back `mmap` in unsuccessful transactions, allocating 3× less heap memory to the application. Benchmark performance is not affected. No source code or libc changes are required for TxOS to detect that `mmap` is transactional.

The possibility of an `mmap` leak is a known problem [ZB06], with several proposed solutions, including open nesting [MBM+06b] and a transactional pause instruction [ZB06]. All previously proposed solutions complicate the programming model for the application developer, the hardware, or both. Some of the API complexity might be encapsulated in a heap implementation closely integrated with the TM system; to the best of our knowledge, this has not been developed in the literature. System transactions encapsulate this complexity in the OS, addressing the memory leak with the simplest hardware requirements and user API.

5.10 Summary

This chapter demonstrates that system transactions can be a practical abstraction for a modern OS kernel to provide, by evaluating the performance characteristics and the capabilities of the TxOS prototype. The overheads introduced in TxOS can be high in many cases, but performance of write-heavy workloads can also improve dramatically. In general, high overheads for using transactions are more acceptable when an application is also benefitting from enhanced functionality. The more concerning overhead is the overhead of non-transactional operations. This chapter identifies the key sources of non-transactional overhead and proposes techniques to reduce this overhead in future work.

This chapter also shows that application developers can address several challenging problems, including recovery from failed software installations and eliminating TOCTTOU race conditions, with system transactions on TxOS. The chapter also describes how TxOS eliminates several coarse-grained locks that limit kernel scalability, and how system transactions integrate with two different user-level transactional memory implementations. Developers need system transactions to fill the gaps in the current OS API, and system transactions can be provided at an acceptable performance cost.

Chapter 6

Transactions as a Building Block for Distributed Applications

Improving application performance and enabling new features are key goals for system transactions. This chapter describes ongoing and future work to leverage system transactions in distributed applications.

Implementations of distributed systems can be substantially limited by the low-level system services presented to the application developer. For instance, the `maildir` IMAP storage format is designed to be lock free [Ber], yet the POSIX API is insufficient for an IMAP server to guarantee repeatable reads of a `maildir` directory. This insufficiency leads IMAP implementations such as Dovecot to reintroduce locks on backend files [Dovb]. When server threads holding file locks terminate unexpectedly, for instance due to a bad client interaction, users experience unpredictable email behavior, such as lost email or messages that cannot be marked as read. System transactions can make reliable distributed applications easier to build by providing a better abstraction for managing system-level concurrency and

durability,

This chapter describes three target distributed applications for system transactions: a Lightweight Directory Access Protocol (LDAP) server, server-side speculation in replicated, fault tolerant systems, including Byzantine Fault Tolerant (BFT) systems, and an IMAP email server. The work with LDAP is complete and evaluated, whereas the discussion of BFT and IMAP are designs for ongoing work.

6.1 LDAP server

The Lightweight Directory Access Protocol (LDAP) is a standardized protocol for storing and accessing directory information. The LDAP protocol is commonly used to implement user directories (including the one used by the University of Texas to track employees and students), as well as to centralize management of user accounts and password hashes for a network of computers. LDAP forms the basis of Microsoft's Active Directory product, and is often used as a replacement for Sun's Unix Network Information Service (`nis`). LDAP deployments generally manage a relatively small data set with modest storage requirements, that threads in the server should be able to access concurrently for good performance.

The concurrency control and recovery abstractions provided by modern operating systems are insufficient for applications with even modest storage requirements, such as LDAP. An average LDAP data set is sufficiently large that storing the data in a single file and serializing write requests with `rename` system calls will severely harm performance. If an LDAP server splits its data across multiple files, it has to use file locking, which provides no durability guarantees and requires extreme care to avoid deadlock among application and file system locks. LDAP servers, such as the popular OpenLDAP system, often work-around the shortcomings of the OS by replacing OS-managed storage with a database, which provides stronger semantics.

LDAP servers represent a class of concurrent servers with modest storage,

recovery, and concurrency requirements that are forced by their semantic requirements to adopt a heavier-weight database system than is strictly necessary for their stable data storage. Databases are designed to optimize complex queries of large data sets; there are certain trade-offs that introduce overheads that are only offset if the data set is sufficiently large. For these "middle ground" applications, current tools do not strike right balance of strong semantics with low overheads. System transactions can provide a simple, lightweight storage solution for such applications.

6.1.1 Replacing database transactions with system transactions

To demonstrate that system transactions can provide lightweight concurrency control for server applications, we modified OpenLDAP 2.3.35's flat file storage module (called LDIF) to use system transactions. The OpenLDAP server supports a number of storage modules; the default is Berkeley DB (BDB).

In adding system transactions to the LDIF storage module, we opted to manage rollback of application state after a failed system transaction by hand. We initially expected the task to be daunting, but we could not use copy-on-write paging, as the server is multi-threaded, and we did not want to incur the high overheads of software transactional memory. As it turns out, the LDAP server code, like any mature software package, has fairly robust error handling code for any system call it issues. System calls can fail for a number of reasons and mature applications must be prepared to handle these failures. Thus, in most code pathways, the only modifications required were to ensure that a system call which failed because of an aborted transaction exited through the appropriate error handling code.

6.1.2 Evaluation

We used the `SLAMD` distributed load generation engine[1] to exercise the server, running in single-thread mode. We run the server in single-thread mode because the current implementation of TxOS does not provide transactional semantics for the file descriptor table when multiple threads in the same process attempt to open or close file handles concurrently. Note that modifications to individual file descriptors (e.g., `seek`, `read`, etc.) are transactional, the limitation is in the mapping of integer file handles to file descriptor structures in the kernel. We plan to fix this in future work.

Table 6.1 shows throughput for the unmodified Berkeley DB storage module, the LDIF storage module augmented with a simple cache, and LDIF using system transactions. These experiments were performed using the same setup described in Chapter 5. The "Search Single" experiment exercises the server with single item read requests, whereas the "Search Subtree" column submits requests for all entries in a given directory subtree. The "Add" test measures throughput of adding entries, and "Del" measures the throughput of deletions.

The read performance (search single and search subtree) of each storage module is within 3%, as most reads are served from an in-memory cache. LDIF has 5–14× the throughput of BDB for requests that modify the LDAP database (add and delete). However, the LDIF module does not use file locking, synchronous writes or any other mechanism to ensure consistency. LDIF-TxOS provides ACID guarantees for updates. Compared to BDB, the read performance is similar, but workloads that update LDAP records using system transactions outperform BDB by 2–4×. This speedup is commensurate with speedups observed for write-intensive microbenchmarks in Section 5.2, attributable to better I/O scheduling. LDIF-TxOS provides the same guarantees as the BDB storage module with respect to concur-

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

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

Table 6.1: Throughput in queries per second of the OpenLDAP server (higher is better) for a read-only and write-mostly workload. For the **Add** and **Del** workloads, the increase in throughput over BDB is listed in parentheses. The BDB storage module uses Berkeley DB, LDIF uses a flat file with no consistency for updates, and LDIF-TxOS augments the LDIF storage module use system transactions on a flat file. LDIF-TxOS provides the same crash consistency guarantees as BDB with more than double the write throughput.

rency and recoverability after a crash.

For developers of applications with modest storage requirements, this experiment shows that system transactions are a simpler alternative to even lightweight databases like BDB.

6.2 Design: Replication and Byzantine Fault Tolerance

Distributed systems often replicate services in order to tolerate faults. If one replica fails, others can absorb its load while the failed replica recovers.

The canonical approach to designing a replicated service is to design the server as a replicated state machine [Sch90]. 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 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 inputs from the operating system, such as the contents of a file or the output of a random number generator.

Research on replicated systems focuses on one of two fault models. The first is a fail-stop system, in which a failures can only manifest as a replica halting unexpectedly. The second class of systems tolerate a bounded number of Byzantine

faults, in which a faulty replica can exhibit arbitrary behavior, including malicious behavior [LSP82].

6.2.1 BFT in a nutshell

Byzantine Fault Tolerance (BFT) is a framework for building replicated systems that encompasses a wide range of faults. When the total number of replicas is n, the system can tolerate $(n-1)/3$ faults, more commonly written as $3f+1$, where f is the number of faults.

BFT protocols essentially process requests in a pipeline consisting of three stages: authentication, ordering, and execution. Each of these stages must be performed by a quorum of replicas (the precise minimum varies slightly by implementation), and several of the stages must wait for broadcast communication among the replicas. 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 message. Once agreement completes, each replica can then execute the request and return a response to the client. These responses are generally checked by the client for a two-thirds majority consensus on the response; responses that disagree indicate a faulty node.

A key limitation of BFT systems is the lack of parallelism in the implementation of a service (i.e., the execution phase). 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 to increase throughput or decrease request processing latency.

A reasonable approach to the performance constraint of serial execution is to use optimistic parallel execution. To the best of our knowledge, optimistic execution has not been used in a BFT server for any real-world workloads because of the complexity of rolling back server state and especially system calls on a mis-

speculation. The Zyzzva system proposed a protocol for exactly the latter form of speculation [KAD+07], but did not implement this speculation for any non-trivial services. Wester et al. [WCN+09] developed a speculation mechanism for a BFT client, but did not implement server-side speculation.

The key piece of missing infrastructure for existing fault-tolerant systems is simple optimistic execution of server code; system transactions, perhaps combined with user-level TM, provide this mechanism. While this section has focused primarily on BFT, where the request processing pipeline is the most complex, any fault-tolerance mechanism based on replicated state machines will suffer similar pipeline stalls and lack parallelism.

6.2.2 Recovery

A second challenge for replication-based fault tolerant systems is recovering from a node failure. Specifically, when a failed replica is replaced with a new correct node (or the same node repaired), the new replica must be brought up to speed with the rest of the system. The typical strategy for recovery is for correct replicas to take periodic checkpoints of their state and to keep a log of subsequent requests. The new replica loads the checkpoint and replays the log until it catches up. During recovery, requests are processed sequentially, and new requests are buffered. Assuming that a fair number of requests can execute safely and concurrently, optimistic execution during recovery could allow much quicker failure recovery.

6.2.3 Required extensions to system transaction API

System transactions provide the key building block of OS-level recovery from a misspeculation, but our existing API will require two modest extensions to work with BFT.

Ordered commit. In a general-purpose transaction processing environment, *any* serialization is generally acceptable. Because replicated state machines need to be deterministic, it is only acceptable to execute in a serialized order *equivalent* to the order specified by the agreement stage of execution. We propose extending the TxOS API with a mechanism to specify a commit order for a set of transactions, similar to thread-level speculation [SBV95, SCZM00].

Relaxed isolation for BFT protocol messages. In servicing an execution request, a thread may still need to receive messages from the agreement phase in order to determine its execution order. In order to permit communication with the BFT layer, we propose an interface for the developer to selectively specify network or IPC traffic that can escape the transaction.

Both of these extensions are relatively straightforward to implement, and necessary to support BFT server-side speculation. Work to incorporate the BFT UpRight library [CKL+09] with TxOS is ongoing.

6.3 Design: IMAP email server

The Internet Message Access Protocol (IMAP) is a widely used protocol for accessing email messages [Cri03]. 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 is considered an improvement over the previous Post Office Protocol (POP) [Mye96], which simply downloads incoming messages from the server and then deletes them. IMAP provides features missing from POP, 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 heavily on the client and server implementations: you can lose the message or end up with multiple copies of it.

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.

6.3.1 Backend storage formats and concurrency challenges

Although the specific storage formats can vary across IMAP server implementations, there are two widely-used storage formats: mbox and maildir. The selected backend storage format used 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 the 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 To alleviate the issues with stale locks in mbox, the maildir format was created [Ber]. 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 messages randomly disappearing. The lack of repeatable reads in maildir led a major IMAP implementation, dovecot, to reintroduce file locking for its maildir backend [Dovb].

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 specific file system in configuring the mail server. Even under the best of circumstances, the 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 [Dova].

6.3.2 Opportunity for transactions

System transactions address several challenges IMAP implementations face by giving the IMAP developers a better interface for managing system-level concurrency. For instance, file locks introduce problems when a server thread exits without clean-

ing up the lock; failed transactions are automatically cleaned up and obviate the need for file locks. System transactions also provide repeatable reads, as they are serializable[2]. Repeatable reads can eliminate the disturbing artifacts that can arise from concurrent access to a maildir inbox. Finally, transactions can be adopted incrementally in an IMAP server, causing minimal disruption to the common code paths. Work to adopt transactions in an IMAP server is ongoing.

[2]Serializability is also known as degree 3 isolation [GLPT76], which guarantees repeatable reads.

Chapter 7

Related Work

This chapter surveys related work, distinguishing system transactions from previous research in OS transactions, journaling file systems, transactional file systems, Speculator, and transactional memory. This chapter also provides additional background on time-of-check-to-time-of-use (TOCTTOU) race conditions.

7.1 Previous transactional operating systems

Locus [WTWPLP85] and QuickSilver [HMC88, SW91] are historical systems that provide some system support for transactions. The primary goal of these systems is committing file writes atomically with distributed transactions. Both systems use database implementation techniques for transactions, isolating data structures with two-phase locking and rolling back failed transactions from an undo log.

A shortcoming of this approach is that simple locks, and even reader-writer locks, do not capture the semantics of container objects, such as a directory. Multiple transactions can concurrently and safely create files in the same directory so long as none of them use the same file name or read the directory. Unfortunately, creating a file in these systems requires a write lock on the directory, which prevents concurrent

access to the directory, even to unrelated files in the directory. To compensate for the poor performance of reader-writer locks in both systems, directory contents may change during a transaction, which reintroduces the possibility of the TOCTTOU race conditions that system transactions ought to eliminate. In contrast, TxOS implements system transactions with lazy version management, more sophisticated containers, and asymmetric conflict detection, allowing it to provide higher isolation levels while minimizing performance overhead.

Another key challenge when two-phase locking is used inside an OS kernel is the risk of deadlocking the kernel. Holding kernel locks for the duration of a transaction can deadlock the kernel if two transactions simply access the same resources in opposite order. Locus does not detect deadlock (but does allow pluggable detection mechanisms), and Quicksilver times out long-running transactions. Timeouts can starve long-running transactions. TxOS does not have to resort to timing out because it uses lazy version management, thus it does not hold locks across system calls. It only holds locks long enough to copy objects and always acquires them in an ordered fashion.

Finally, Quicksilver does not support strong isolation (Section 3.2), and hence does not necessarily serialize non-transactional operations with transactional operations. Locus allows transactional and nontransactional applications to access the same data, but requires an explicit commit by the non-transactional thread. Uncommitted, non-transactional records are committed by the next transaction to access the data. It is unclear what happens to such a record if the subsequent transaction aborts. The current STM literature shows a number of situations that lead to data structure corruption when strong isolation is not provided [MBS+08, SMAT+07]. Because TxOS performs both transactional and non-transactional updates to kernel data structures, it must provide strong isolation lest the kernel data structures become corrupted.

7.1.1 VINO

The VINO kernel uses transactions in the OS to recover from misbehaved kernel extensions [SESS96, Sma98]. The VINO kernel has an extensible design: applications may load custom code into the kernel that overrides or extends kernel functionality. For instance, an application could load extensions that augment the kernel's TCP/IP stack with proposed extensions for the application's network connections. The key challenge is permitting untrusted kernel extensions without exposing the kernel to compromise from these extensions. A monolithic OS, like Linux, must trust any code loaded into the OS. In the TCP/IP example, either the Linux system administrator must trust the extensions and expose all users to potential bugs or malware in the kernel extension, or the untrusted application must reimplement the entire TCP/IP stack in its address space on top of a lower-level OS abstraction. VINO restricts untrusted modules to an explicit API with careful access control checks using software fault isolation [WLAG93].

If an untrusted module misbehaves and must be removed, VINO uses transactions to recover OS resources held by the untrusted module. When the kernel calls an untrusted module, that call is wrapped in a transaction. All calls from the untrusted module back to the kernel execute in a transactional context. Transactional updates to kernel state are isolated using two-phase locking, and recovery is implemented with a combination of undo logs and deferred actions tracked by a redo log. If misbehavior is detected, such as a failed access check or a timed out request, the transaction is aborted and the kernel uses the undo log to release locks, free memory, etc.

VINO does not export system transactions to applications as part of the system call API; transactions are only used by the kernel to isolate untrusted extensions. User-level applications may not arbitrarily compose system calls to perform an atomic software installation in VINO, as they can in TxOS. TxOS, like Linux,

must trust kernel modules and cannot use system transactions to isolate kernel extensions. The VINO transaction design is similar to other transactional OSes and is subject to the same problems, including deadlock, which the TxOS design addresses.

7.2 Transactional Memory

Transactional Memory (TM) systems provide a mechanism to provide atomic and isolated updates to application data structures. Transactional memory is implemented either with modifications to cache coherence hardware (HTM) [HWC+04, MBM+06a], in software (STM) [DSS06], or as a hybrid of the two [CMTC+07, DFL+06].

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

Integrating user and system transactions creates a simple and complete transactional programming model. On a TM system integrated with TxOS, when a TM application makes a system call, the runtime begins a system transaction. The user-level transactional memory system handles buffering and possibly rolling back the application's memory state, and the system transaction buffers updates to system state. The updates to system state are committed or aborted by the kernel atomically with the commit or abort of the user-level transaction. The programmer sees the simple abstraction of an atomic block that can contain updates to user data structures and system calls.

The remainder of this subsection reviews other approaches to supporting sys-

tem calls inside of a memory transaction, as well as particularly related transactional memory applications and systems.

7.2.1 Open nesting

Several proposals for transactional memory systems, both in hardware [MBM+06b, MCC+06] and software [HSATH06], support open nested transactions [Mos81] to allow system calls inside of a transaction. When an open nested transaction commits, its results are immediately visible, temporarily removing isolation guarantees of the parent transaction. In order for open nesting to be correct, the application must enforce some sort of logical isolation on the operations executed inside the open nested code.

A classic motivating use for open nesting is to optimize page locking in database implementations. Database transactions that update a record typically acquire a record lock and then a lock on the page of memory containing the record. Holding the page lock until commit needlessly prevents other transactions from performing updates to unrelated records on the same page. Performing the page write in an open nested transaction allows the page lock to be released early, increasing concurrency of the system. The write to the page is still logically isolated because the transaction retains the record lock until it commits.

Enforcing logical isolation on operating system managed resources in a user-level transactional memory system, however, is much trickier. One key challenge is that the OS interfaces for isolating updates to shared resources are quite limited. Many shared resources and abstractions, such as signals and pipes, do not provide any isolation mechanism. File locking is the primary isolation mechanism available to users, and support for mandatory file locking is incomplete or difficult to use on modern operating systems.

An even more difficult challenge to supporting system calls with open nesting

is rolling back the system calls. All open nesting proposals require open nested transactions to provide a commit handler to release logical isolation and an abort handler to undo its effects. Some system calls have logical complements, like deleting a created file or freeing allocated memory. The problem is that even these simple calls can have far-flung side effects that are difficult to anticipate and undo. For instance, creating a new file may implicitly delete an existing file with the same name, which must somehow be detected and restored by the application. Mapping a page of memory can also implicitly unmap another page, permanently deleting it from the system. If a page if permanently deleted by the OS, the application loses the ability to recover from a restarted transaction. While open nesting may work for the simple cases and even many common cases, a complete solution will require cooperation from the operating system to identify, isolate, and undo these far-flung side effects.

7.2.2 Transactional pause and escape actions

Zilles and Baugh [ZB06] propose a transactional pause primitive that serves similar purposes to open nesting, including support for memory allocation inside of a transaction. LogTM [MBM+06b] adopts a similar primitive called an *escape action*. The primary distinction between transactional pause and open nesting is whether the nested code executes atomically or not. The transactional pause primitive makes no atomicity guarantees for the paused code or its commit and abort handlers. This tradeoff makes the TM implementation simpler, but requires more expertise to be used correctly. This is arguably a good tradeoff because the marginal benefit to TM designers is much larger than the marginal cost to the programmer, and because both primitives are sufficiently difficult to require expert programmers. This argument is supported by Ni et al. [NMAT+07] who discuss the challenges of writing correct commit and compensating actions, and examine implementation issues

posed by overlapping read-write sets between ancestor and open nested transactions. Moravan et al. [MBM+06b] address some of these challenges by defining a set of formal correctness conditions, under which the TM system can guarantee correct execution of these primitives.

7.2.3 Inevitable transactions

Several transactional memory proposals also include inevitable transactions, which cannot be restarted [BDLM07, SMS08]. Inevitable transactions allow transactions to complete even if they encounter conditions that make rollback difficult, such as system calls or cache overflow in a hardware TM system. As long as the TM system doesn't support explicit transaction abort operations, this approach provides a practical solution to these problems. TM implementations can only support one inevitable transaction at a time. Thus, a key drawback to this approach for system calls is that it can needlessly restrict concurrency when the system calls are logically independent, such as writing to two separate files. System support for transaction rollback, as provided by system transactions, can obviate the need for inevitable transactions in many situations.

7.2.4 xCalls

Volos et al. [VTG+09] extend the Intel STM compiler with xCalls, which support deferral or rollback of common system calls when performed in a memory transaction. Depending on the semantics of a given system call, the xCalls implementation will either defer the operation (e.g., buffering writes to a file in memory), or use application-level locking to isolate the resource (e.g., using application locks to ensure consistent file reads). The xCalls system requires support for a transactional pause primitive or open nesting from the underlying TM system, but encapsulates much of the complexity of writing compensating actions for system calls and rea-

soning about OS semantics.

A key benefit of xCalls over other user-level approaches is that the interface makes explicit when these operations will occur. For instance, the API specifies that data returned from a deferred operation will not be available until the transaction commits. Similarly, errors from deferred operations are returned when the transaction terminates.

xCalls have the advantage of not requiring changes to the OS kernel, but this also limits their utility outside of a single application. Because xCalls are implemented in a single, user-level application, they cannot isolate transaction effects from kernel threads in different processes, ensure durable updates to a file, or support multi-process transactions, all of which are needed to perform a transactional software installation and are supported by TxOS.

7.2.5 TxLinux

The system transactions supported by TxOS solve a fundamentally different problem from those solved by TxLinux [RHP+07]. TxLinux is a variant of Linux that uses hardware transactional memory as a synchronization primitive to protect OS data structures within the kernel, whereas TxOS exports a transactional API to user programs. The techniques used to build TxLinux enforce consistency for kernel memory accesses within short critical regions. However, these techniques are insufficient to implement TxOS, which must guarantee consistency across heterogeneous system resources, and which must support system transactions spanning multiple system calls. TxLinux requires hardware transactional memory support, whereas TxOS runs on currently available commodity hardware.

7.2.6 Precise conflicts

Transactional memory systems, especially hardware implementations, can have high conflict rates on data accesses that are semantically safe. One common example is concurrent insertions into a sorted linked-list. One operation will read a next pointer the other is trying to change, causing a conflict and serializing the operations. Concurrent list insertions, however, can be semantically safe as long as invariants of the list are upheld (e.g., sorted, unique entries) and no transaction attempts to enumerate the list concurrently. The key problem is that the atomicity and consistency guarantees of conflict serializability are often stronger than the programmer really needs. Although conflict serializability is always safe, this generality can come at a needless performance cost.

As discussed above, one solution to this problem is releasing physical isolation on hotly contended resources through the use of open nesting or transactional pause mechanisms. Open nesting and transactional pause are appealing solutions because they are the least restrictive and arguably the most general. These mechanisms require the programmer to ensure logical isolation in the application, and the runtime releases all isolation enforcement on data accessed while the transaction is paused or when an open nested transaction commits. The lack of structure provided by these APIs is a double-edged sword, as it can be difficult for programmers to correctly isolate all side-effects of an operation.

A more structured alternative is *type-specific* locking, where each object type defines a *compatibility matrix* for its operations. A compatibility matrix is essentially a table of which operations conflict [BHG87]. Categorizing all operations as reads and writes is simplest and closely matches current transactional memory implementations.

In general, type-specific locking is implemented in an object by having the object class provide its own locking scheme rather than using the default read-

er/writer locking. Returning to the linked-list example, one might gain additional concurrency by allowing concurrent list operations via more sophisticated locking schemes. For instance, one could use hand-over-hand locking while traversing the incidental list nodes to retain isolation only on the nodes modified, allowing additional concurrency.

The Galois System [KPW+07] and Transactional Boosting [HK08] are similar in spirit to type-specific locking, in that each object can define which operations must be isolated from one another and which can run concurrently (even if the operations are not strictly conflict serializable). Galois and Boosting provide more structure to the programmer by identifying compatible operations through properties such as inverse and commutative functions.

Similarly, database implementations frequently use increment and decrement locks, a type-specific lock of sorts, to protect statistics. These exploit the commutativity of addition to allow concurrent atomic increments and decrements, but no reads or writes of any other type [BHG87]. The DASH SMP prototype [LL98] similarly provides the Fetch&Inc and Fetch&Dec instructions to reduce the synchronization cost of atomic counter operations. This is one of the few cache coherence additions that was considered worthwhile enough to be included in the SGI Origin (as Fetch&Op) [LL97].

TxOS draws inspiration from these systems by supporting type-specific locking to increase concurrency without sacrificing strong consistency guarantees. This approach is particularly valuable for providing transactional semantics to container objects, such as lists of files in a directory. TxOS is able to support concurrent transactional access to disjoint files in the same directory, whereas previous transactional operating systems had to serialize these accesses or allow non-serializable operations to maintain performance.

7.3 Speculator

Speculator [NCF05] 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 initial motivation for Speculator was to hide the latency of common NFS server requests. When applications access files hosted on a network filesystem, the OS generates synchronous requests to the server to ensure that the cached data is coherent. In the common case, the cached data is correct and this latency needlessly harms performance.

Speculator is an extension to the Linux 2.4 kernel that allows applications to speculate past NFS requests. Rather than wait for a server response, system calls can be serviced from cached file data. Before using potentially stale, cached data, Speculator checkpoints the process, and allows the process to continue execution inside of a speculative environment. The speculation ends when the OS receives a response from the NFS server. If the cached data was stale, the application rolls back to the checkpoint and restarts. Otherwise, the application continues execution unencumbered.

The key to safe speculation is preventing speculative processes from publishing speculative data to external I/O devices (in the database literature, this is known as the "output commit" problem). If a speculative process attempts to execute an unsupported action, the process is blocked until the speculation resolves. Speculator also tracks dependences among processes on the system. If speculative process A writes data to a pipe that is read by process B, B's state is checkpointed and it becomes part of A's speculation. Dependence tracking in Speculator allows cascading rollbacks when a speculation fails.

In subsequent work, Speculator was extended to hide the latency of synchronous writes to a local file system [NVCF06], by maintaining the property of *external synchrony*. External synchrony is the property that once a synchronous

write is issued by an application, no other output from the application is externally visible until the write has completed. In this work, applications can continue to speculatively execute after a synchronous write is issued, but subsequent I/O is blocked until the first synchronous write completes. Speculator has been extended to parallelize security checks [NPCF08], to debug system configuration [SAF07], and to hide the latency of requests to replicated network services (e.g., Byzantine fault tolerant services) at the client [WCN+09].

Because Speculator and TxOS both provide certain isolation guarantees and a checkpoint/restore mechanism, the distinctions between the systems can be subtle. The primary difference is that Speculator is designed to isolate a series of operations from external visibility, whereas TxOS transactions are designed to isolate concurrent applications on the same system. Speculations may become dependent and share data, meaning that Speculator does not eliminate TOCTTOU vulnerabilities. If a TOCTTOU attack occurred in Speculator, the attacker and victim would be part of the same speculation, allowing the attack to succeed.

Another useful distinction is that Speculator is primarily transparent to the application, whereas system transactions provide ACID semantics for a user-delimited series of operations. A substantial benefit of the Speculator design is that difficult operations, such as device I/O, need not be supported, as the application can simply block until the speculation resolves. The benefit of providing transactions is that the ACID properties provided by transactions enable more powerful applications such as atomic software installation/update.

7.4 Transactional file systems

TxOS simplifies the task of writing a transactional file system by detecting conflicts and versioning data in the virtual file system layer. Some previous work such as OdeFS [GJR94], Inversion [Ols93], and DBFS [MTV02] provide a file system inter-

Feature	Amino	TxF	Valor	TxOS
Low overhead kernel implementation	No	Yes	Yes	Yes
Can be root fs?	No	Yes	Yes	Yes
Framework for transactionalizing other file systems	No	No [1]	Yes	Yes
Simple programmer interface	Yes	No	No	Yes
Other kernel resources in a transaction	No	Yes [2]	No	Yes

Table 7.1: A summary of features supported by recent transactional file systems.

face to a database, implemented as a user-level NFS server. These systems do not provide atomic, isolated updates to local disk, and cannot address the problem of coordinating access to OS-managed resources. Berkeley DB and Stasis [SB06] are transactional libraries, not file systems. Amino [WSSZ07] supports transactional file operation semantics by interposing on system calls using `ptrace` and relying on a user-level database to store and manage file system data and metadata. Other file systems implement all transactional semantics directly in the file system, as illustrated by Valor [SGC+09], Transactional NTFS (also known as TxF) [RS09], and others [GT05, Sel93, SW91].

Table 7.1 lists several desirable properties for a transactional file system and compares TxOS with recent systems. Because Amino's database must be hosted on a native file system, it cannot be used as the root file system. TxF can be used as the root file system, but the programmer must ensure that the local system is the two-phase commit coordinator if it participates in a distributed transaction.

Like TxOS, Valor provides kernel support in the page cache to simplify the task of adding transactions to new file systems. Valor supports transactions larger than memory, which TxOS currently does not. Valor primarily provides logging and coarse-grained locking for files. Because directory operations require locking the

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

[2] Windows supports a transactional registry.

directory, Valor, like QuickSilver, is more conservative than necessary with respect to concurrent directory updates.

Sun et al. develop a roughly transactional file system based on VFS interposition, with which they sandbox untrusted applications [SLSV05]. This file system does not provide a transaction abstraction to the application directly, but untrusted applications are forced to run inside of a transaction. When the untrusted application completes, the user is presented with a summary of the changes to the file system, which the user can commit or roll back. Like TxOS and Valor, this system benefits from modularity and code reuse by implementing transactions near the VFS layer. A transactional file system built entirely by interposing between the VFS and file system layer, but not changing either layer otherwise, will be limited by the VFS interface. For instance, Sun et al. use POSIX file locks to isolate updates to files and ensure an atomic commit. POSIX locks do not apply to directories, and cannot ensure that updates to the directory namespace are committed atomically. Implementing a "stackable" transactional file system layer is a laudable goal, but the current VFS API is insufficient to implement ACID transactions robustly. TxOS implements transactions in the VFS layer and modifies the VFS API, providing a modular transactional file system design with minimal impact on a specific file system.

In addition to TxF, Windows Vista introduced a transactional registry (TxR) and a kernel transaction manager (KTM) [RS09]. KTM allows applications to create kernel transaction objects that can coordinate transactional updates to TxF, TxR, and user-level applications, such as a database or software updater. KTM coordinates transaction commit, but each individual filesystem or other kernel component must implement its own checkpoint, rollback, conflict detection, etc. In contrast, TxOS minimizes the work required to transactionalize a file system by providing conflict detection and isolation in shared virtual filesystem code. Modularity at the

file system level is less important in Windows, which provides many fewer file systems than Linux, but modularity across different subsystems is useful in both cases. The individual file systems in both Windows and TxOS must use a journal or other mechanism for atomic, durable commits to disk.

TxOS and KTM also represent different points in the design space of transactional application interfaces. KTM requires that all transactional accesses be explicit, whereas TxOS allows unmodified libraries or applications to be wrapped in a transaction. Requiring each system call to be explicitly transactional is a more conservative design because unsupported operations do not compile, whereas TxOS detects these dynamically. A key downside to KTM's low-level interface is that it requires individual application developers to be aware of accesses that can deadlock with completely unrelated applications on the same system (such as accessing two files in opposite order), and implement their own timeout and backoff system. In contrast, transactions in TxOS cannot deadlock and TxOS can arbitrate conflicts according to scheduler policies (Section 3.2.2) without any expert knowledge from the developer.

TxOS provides programmers with a simple, natural interface, augmenting the POSIX API with only three system calls (Table 2.1). Other transactional file systems require application programmers to understand implementation details, such as deadlock detection (TxF) and the logging and locking mechanism (Valor).

7.5 Distributed transactions

A number of systems, including TABS [SDD+85], Argus [LCJS87, Lis88], and Sinfonia [AMS+07], provide support for distributed transactions at the language or library level. These user-level transaction systems are subject to the same challenges in isolating system resources as transactional memory systems. These systems do, however, provide useful insights into the implementation and application

of transactions.

Among other services, TABS [SDD+85] provides transactional support in its window manager. Transactions in the window manager were visible to the end user; text dialogs from aborted transactions were crossed out by the window manager.

TABS also provided developers with a *weak queue*, which relaxes the strict FIFO ordering of enqueues to increase concurrency. Liskov et al. present a similar example of a banking system in Argus where querying the total amount held at a branch can return slightly stale values via a user-defined atomic type, increasing concurrency [Lis88].

Argus is also an early transaction system that allows users to control the safety/performance tradeoff in an application by specifying the level of durability risk they can tolerate [Lis88]. This approach is also taken in the Microsoft's .NET framework by allowing resource managers to be specified as volatile [Mic]. TxOS similarly allows transactions to relax durability guarantees to avoid blocking on synchronous writes during commit.

Sinfonia [AMS+07] increases the concurrency of transactions by restricting their generality. A Sinfonia *minitransaction* consists of a list of memory locations and values to be read, written, and compared. Essentially, a minitransaction is a multi-data compare-and-swap operation. The only control flow inside a minitransaction is that the minitransaction will abort if any of the comparisons fail. This restriction forces the minitransactions to remain short and allows minitransaction execution to overlap completely with the execution of the two-phase commit protocol, minimizing network round trips.

The authors of Sinfonia demonstrate the utility of minitransactions by implementing a network file system and a group communication system. These systems construct larger units of work that appear atomic and consistent to the rest of the system by using minitransactions to read data into a private cache, operating on the

```
    Victim                  Attacker
if(access('foo')){
                        symlink('secret','foo');
  fd=open('foo');
  write(fd,...);
  ...
}
```

```
    Victim                  Attacker
                        symlink('secret','foo');
 sys_xbegin();
if(access('foo')){
  fd=open('foo');
  write(fd,...);
  ...
}
 sys_xend();
                        symlink('secret','foo');
```

Figure 7.1: An example of a TOCTTOU attack, followed by an example that eliminates the race using system transactions. The attacker's symlink is serialized (ordered) either before or after the transaction, and the attacker cannot see partial updates from the victim's transaction, such as changes to `atime`.

private copy with no transactional context, and then using a final minitransaction to atomically revalidate the cache and write the updates to shared memory. In many regards, this is like users manipulating their own lazy versioned cache with lazy conflict detection. Minitransactions have the added advantage that consistency is easily verified by checking that each minitransaction maintains data structure invariants.

7.6 TOCTTOU race conditions

Figure 7.1 depicts a scenario in which an application wants to make a single, consistent update to the file system by checking the access permissions of a file and conditionally writing it. Common in `setuid` programs, this pattern is the source of a major and persistent security problem in modern operating systems. An attacker

can change the file system name space using symbolic links between the victim's `access` control check and the file `open`, perhaps tricking a `setuid` program into overwriting a sensitive system file, like the password database. The POSIX API provides no way for the application to tell the operating system that it needs a consistent view of the file system's name space.

Although most common in the file system, system API races, or time-of-check-to-time-of-use (TOCTTOU) races, can be exploited in other OS resources. Local sockets used for IPC are vulnerable to a similar race between creation and connection. Versions of OpenSSH before 1.2.17 suffered from a socket race exploit that allowed a user to steal another's credentials [Ach]; the Plash sandboxing system suffers a similar vulnerability [Pla]. Zalewski demonstrates how races in signal handlers can be used to crack applications, including sendmail, screen, and wu-ftpd [Zal01].

While TOCTTOU vulnerabilities are conceptually simple, they pervade deployed software and are difficult to eliminate. At the time of writing, a search of the U.S. national vulnerability database for the term "symlink attack" yields over 600 hits [NIS10]. Further, recent work by Cai et al. [CGJ09] exploits fundamental flaws to defeat two major classes of TOCTTOU countermeasures: dynamic race detectors in the kernel [TY03] and probabilistic user-space race detectors [THWS08b]. This continuous arms race of measure and countermeasure suggests that TOCTTOU attacks can be eliminated only by changing the API.

In practice, such races are addressed with ad hoc extensions to the system API. Linux has added a new close-on-exec flag to fifteen different system calls to eliminate a race condition between calls to `open` and `fcntl`. Tsafrir et al. [THWS08a] demonstrate how programmers can use the *openat()* family of system calls to construct deterministic countermeasures for many races by traversing the directory tree and checking user permissions in the application. However, these techniques cannot

protect against all races without even more API extensions. In particular, they are incompatible with the `O_CREAT` flag to `open` that is used to prevent exploits on temporary file creation [CBWK01].

TOCTTOU races are more problematic on UNIX systems than Windows. Windows reduces the risk of privilege escalation from TOCTTOU attacks by only allowing privileged applications to create symbolic links. Windows applications use mandatory locking more heavily than UNIX applications, reducing the likelihood of race conditions in exchange for a higher risk of improperly released locks.

Fixing race conditions in the API as they arise is not an effective long-term strategy. Complicating the API in the name of security is risky: code complexity is often the enemy of code security [Ber07]. Because system transactions provide deterministic safety guarantees and a natural programming model, they are an easy-to-use, general mechanism that eliminates API race conditions.

Chapter 8

Composing Linked List Operations Without Locks

Software developers need better tools to write concurrent programs that effectively leverage new generations of multicore hardware. In order to safely share data structures, many programs adopt locking, which limits the performance scalability of the application. For some data structures, the programmer can adopt specialized implementations that permit more concurrency, but these often heavily restrict functionality. This chapter addresses this unsavory dilemma between functionality and scalability in linked list implementations, describing a new linked list design that provides the complete functionality of locked lists with performance scalability near that of current lock-free implementations.

Applications with invariants across multiple lists cannot benefit from performance scalability of lock-free lists, such as the `ConcurrentSkipList` classes of the Java Concurrency package. Current lock-free list designs, including the popular Harris-Michael algorithm [Har01, Mic02], can safely insert or remove individual items from a list, but cannot safely compose multiple operations. For instance, if an application moves an item from one lock-free list to another, the list implementation

can atomically remove the item from the first list and atomically add the item to the second list, but there will be a period during which the item is is either on both lists, or on neither list. If the application has an invariant that requires the item to be on exactly one of two lists, the only safe option is to use locking. The inability to safely compose complex list operations prevents some applications from enjoying the performance benefits of lock-free lists.

A second barrier to adoption of lock-free lists is their reliance on dynamic allocation and reclamation of list nodes. The Harris-Michael algorithm avoids synchronization with readers of a deleted list node by deferring reclamation until it is sure no threads are referencing the node. If an item is moved to another list, a new list node is dynamically allocated.

This reliance on dynamic allocation of list nodes is unacceptable for operating system kernels, which manage physical memory and file system caches using lists. For instance, it is difficult to write low-level memory management code so reentrant that, under a low memory condition, the kernel allocator can successfully allocate list nodes in order to add reclaimed memory to a free list. Previous attempts to dynamically allocate and garbage collect list nodes in kernel code have proved intractable [MS05]. For this reason, both Linux and Windows avoid dynamic memory allocation of list nodes by embedding the list nodes directly in the data structures to be listed. These sets of lists are protected with coarse grained locks which, at least in the Linux kernel, have proved to be a persistent scalability bottleneck [Cor09]. If OS kernels used more scalable lists, all applications would benefit, but this will require a lock-free design that does not impose such heavy memory management requirements on low-level kernel code.

This chapter presents a new lock-free linked list design, called OLF, or optimistic, lock-free lists. OLF eliminates the functional restrictions of previous designs while retaining performance scalability drastically superior to locking. At a high

Function Name	Description
bool insert (key_t key, val_t value)	Insert a value into the list. Returns true if the value was not already present.
bool delete(key_t key)	Remove an item from a list. Returns true if the item was found and removed, false if the item is not present.
val_t search(key_t key)	Find an entry in the list, returning the associated value.

Table 8.1: The canonical linked-list API.

level, list nodes in OLF are *versioned* according to a shared epoch counter. Readers of an OLF list execute concurrently with writers, and concurrent writers can optimistically create new versions of a list node, retrying incompatible attempts to modify the same list node. The OLF algorithm ensures that threads can safely traverse lists, the system always makes forward progress, and that writing threads can compose arbitrary list modifications safely. The OLF design is suitable for use in an OS kernel, and the chapter describes its application to improve Linux kernel scalability. We leave integration of OLF with TxOS for future work.

8.1 Background on lock-free lists

This section reviews the Harris-Michael lock-free list algorithm [Har01, Mic02], the *de facto* lock-free list algorithm. The Harris-Michael algorithm is used in the Java Concurrency Package's `ConcurrentSkipListSet` and `ConcurrentSkipListMap` classes. This section also reviews the use of read-copy update (RCU) to protect read-mostly lists in the Linux kernel. RCU applies similar concepts to the Harris-Michael algorithm to eliminate read locks, but still requires write locks to compose operations. Write locking in RCU prevents concurrent modifications of lists. Section 8.5 provides more exhaustive coverage of related work on lock-free linked lists.

8.1.1 Harris-Michael algorithm

The Harris-Michael algorithm is restricted to singly-linked lists, and in these examples, we assume the list is sorted by key. The canonical interface for a lock-free list implementation, including this algorithm, consists of three functions: search, insert, and delete, listed in Table 8.1.

The `search` function is implemented in a fairly straightforward manner: reading threads simply follow a series of `next` pointers until they arrive at the desired list node. A key concern of the insert and delete functions, therefore, is to modify the list such that searches do not dereference a bad pointer value.

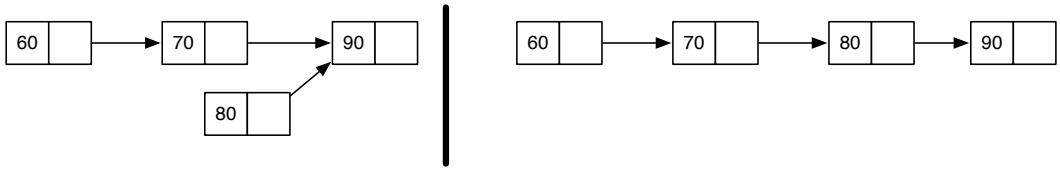

Figure 8.1: Key steps to insert a node in a Harris-Michael list.

The implementation of `insert` is fairly straightforward, as illustrated in Figure 8.1. The list is first traversed to identify where to insert the node. The new list node, for key 80 in the Figure, is allocated and populated with an appropriate `next` pointer. Finally, the `next` pointer of the previous node (70) is set with an atomic compare-and-swap (CAS) instruction. If the CAS fails, the insert must be retried. This CAS ensures that no other node was inserted after node 70 in the list, but there is another race condition that requires additional effort to prevent. As node 80 is being inserted into the list, another thread could delete node 70 from the list, as depicted in Figure 8.2. For this reason, a number of lock-free linked list algorithms require a double-compare-and-swap (DCAS) instruction [GC96, MP92]. For performance reasons, DCAS has not been implemented on a processor since the Motorola 68k.

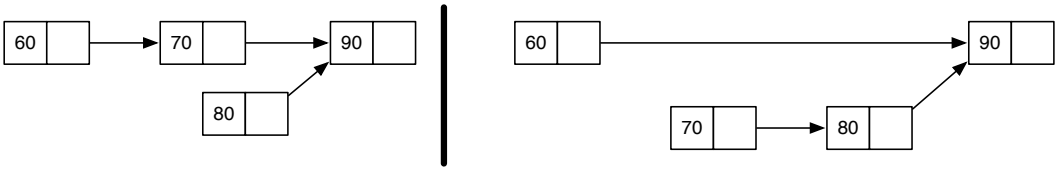

Figure 8.2: Insertion race condition with a single compare-and-swap.

Thus, the challenge for lock-free list algorithms is implementing `delete` such that other, concurrent operations work correctly. The key insight of Harris's original algorithm is separating *logical* deletion from *physical* deletion, as illustrated in Figure 8.3. Logical deletion is performed by setting the next pointer of a deleted node (70 in the example above) to a numerically distinct value that still allows traversal. For example, one could set the low bit of the pointer to indicate logical deletion, assuming that all list nodes are word-aligned. This allows a reader or inserter to continue traversing the list, as well as reliably test for logical deletion. Thus, an inserter would not insert a new node after a logically deleted node, but would back up one step.

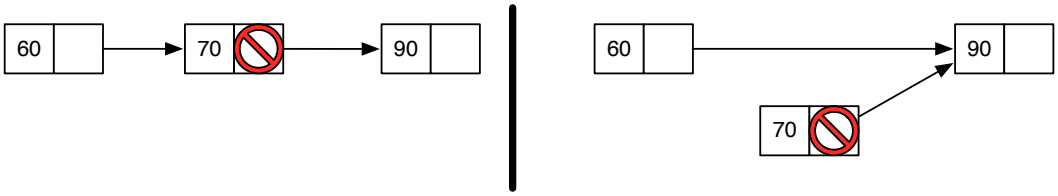

Figure 8.3: Key steps to delete a node from a Harris-Michael list.

After a node is logically deleted, it is then physically removed from the list by changing the `next` pointer of the previous node with a CAS operation. A reader can safely traverse the list and filter out any logically deleted nodes, regardless of concurrent deletions.

The final issue is determining when the memory for a deleted list node can

be reclaimed safely. Note that even after physical deletion, a reading thread may still refer to this node for some time. Harris's original algorithm proposed reference counting garbage collection; Michael later refined the algorithm to incorporate other strategies including hazard pointers [Mic04a], which explicitly track active object references by readers. In order for either memory reclamation approach to work, a deleted list node must persist until all potential readers cannot read the node. This precludes embedding list nodes directly in the object to be listed, because a statically allocated list node must be repurposed immediately if an object is deleted and then inserted in another list.

8.1.2 RCU lists

Read-copy update (RCU) linked lists [McK04] allow limited composition with a Harris-Michael-like design that is lock-free for readers, but serializes writers with a lock. RCU linked lists inherit many of the limitations discussed above, including the requirement of only singly-linked lists and the inability to recycle embedded list nodes until after all potential readers have released a reference.

RCU is primarily used in the Linux kernel to mitigate scalability problems on read-mostly lists, as heavily written lists are protected with a lock and will see no benefits. In this context, garbage collection of list nodes is simplified based on the kernel coding discipline that no references to an RCU-freeable object can be held across a CPU scheduling event. While any reader is traversing an RCU-protected list, it must disable preemption and cannot call any function that can block (e.g., dynamic memory allocation, disk read, etc.). At a high level, freed objects are placed on a pending list; once each CPU has *quiesced*, or scheduled another thread, the objects on the pending list can be freed.

List Property	Why required?	Current Techniques
Compose multiple operations	Atomically move items from one list to another, doubly linked lists, multiple lists with mutual consistency requirements, etc.	Locking
Embedded list nodes	Use lists in code where dynamic memory allocation is not available.	Locking, limited RCU

Table 8.2: Required linked list properties, why they are necessary, and the current options available to developers.

8.1.3 Limitations

The Harris-Michael algorithm is elegant in its simplicity, but it is also functionally limited. Concurrent insertions and removals are handled safely, but it cannot compose these operations into larger operations, such as moving an item from one list to another. Harris-Michael lists are singly-linked; doubly-linked lists require locks. RCU partially addresses these limitations by reverting to locking for writers. Harris-Michael lists also require a lock-free memory allocator and garbage collector; while lock-free allocators exist for user-level programs, they may be impractical to adopt in low-level system software. This chapter addresses these limitations, summarized in Table 8.2, without reverting to locking, thereby retaining the performance scalability benefits.

8.2 Optimistic lock-free list algorithm

This section describes the key contribution of this chapter: a new optimistic, lock-free list (OLF). The OLF algorithm supports atomic composition of multiple list operations as well as embedded list nodes, eliminating reliance on dynamic memory allocation and reclamation of list nodes.

List nodes in OLF are versioned according to a monotonically increasing

epoch. List nodes contain multiple *slots*, which consist of a previous and next pointer associated with a given epoch. Threads manipulate lists in explicit read and write critical regions, which are associated with a given epoch. Writers allocate slots and speculative list entries, which are committed in epoch order. As readers traverse a list, at each list node they follow the pointers in the slot with the greatest epoch less than or equal to their own.

Among a set of lists, epochs are managed using three shared counters: the value of the highest committed epoch (`committed_epoch`), the value of the highest uncommitted epoch (`max_epoch`), and a cache of the oldest reader's epoch (`read_epoch`). These counters can be thought of as replacing a lock variable. All other bookkeeping associated with OLF is in thread-local storage (Figure 8.7); thread-local bookkeeping is only written by one thread, but must be readable by other threads.

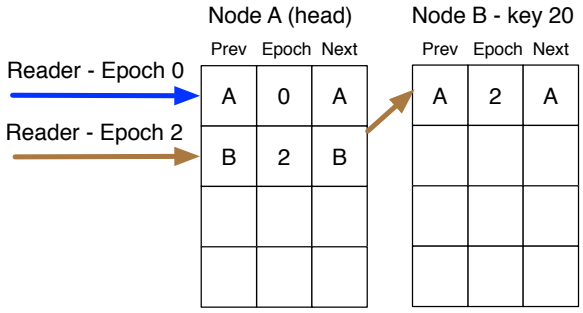

Figure 8.4: Two threads with different epoch reading a list. Each thread sees a view of the list appropriate to the point in logical time in which it is serialized. The reader with epoch value 2 sees Node B with key 20, and the reader with the earlier epoch of 0 does not.

Figure 8.4 illustrates two list readers, one that started at epoch 0 (before a writer added Node B at epoch 2), and one that began after the writer completed (epoch 2). The earlier reader sees a view of the list head that shows it empty (next and previous pointers to itself), while the later reader traverses a different set of slots that lead to Node B. Note that valid epochs increase by 2 because the low bit

of an epoch indicates a failed speculation that should be ignored.

Optimistic writers. The OLF algorithm is optimistic, in that it allows concurrent writers to speculatively update different nodes in the list. If the list nodes are uncontended, multiple write critical sections can execute in parallel and then commit in quick succession. If two writers collide, one may have to retry its modifications. Figure 8.5 illustrates two concurrent write threads. Each thread is inserting a new list node between disjoint pairs of committed nodes. As long as writes are to disjoint list nodes, they can proceed concurrently.

Writers must commit their changes in epoch order. When a writer completes a critical region, it must wait for any writers with an earlier epoch to complete. Writers track an undo log of each modified list node. In order to commit, the writer walks this log and checks for concurrent modifications by a writer with an earlier epoch. If any update was unsafe, all written slots are invalidated by setting the low bit of the epoch and the writer retries. If the commit is safe, the writer simply increments `committed_epoch`. For simplicity and fairness, OLF essentially always arbitrates conflicts in favor of the oldest writer. In the previous example, if the two writing threads attempted to modify the same list node, the younger writer would ultimately invalidate its speculative modifications and retry. Committing and arbitrating conflicts according to a prescribed order ensures that all writers will complete eventually, as no writer can obstruct the oldest writer, and each writer will eventually become the oldest.

Note that there is a tradeoff between space and the amount of speculative parallelism for list writers, as writers can only speculate when there are reclaimable slots available. A list with only two slots per node will behave equivalently to RCU; by allocating additional slots, non-conflicting writers can execute in parallel. For all experiments in this chapter, we allocate four slots per node, as the benefit of additional slots diminishes quickly.

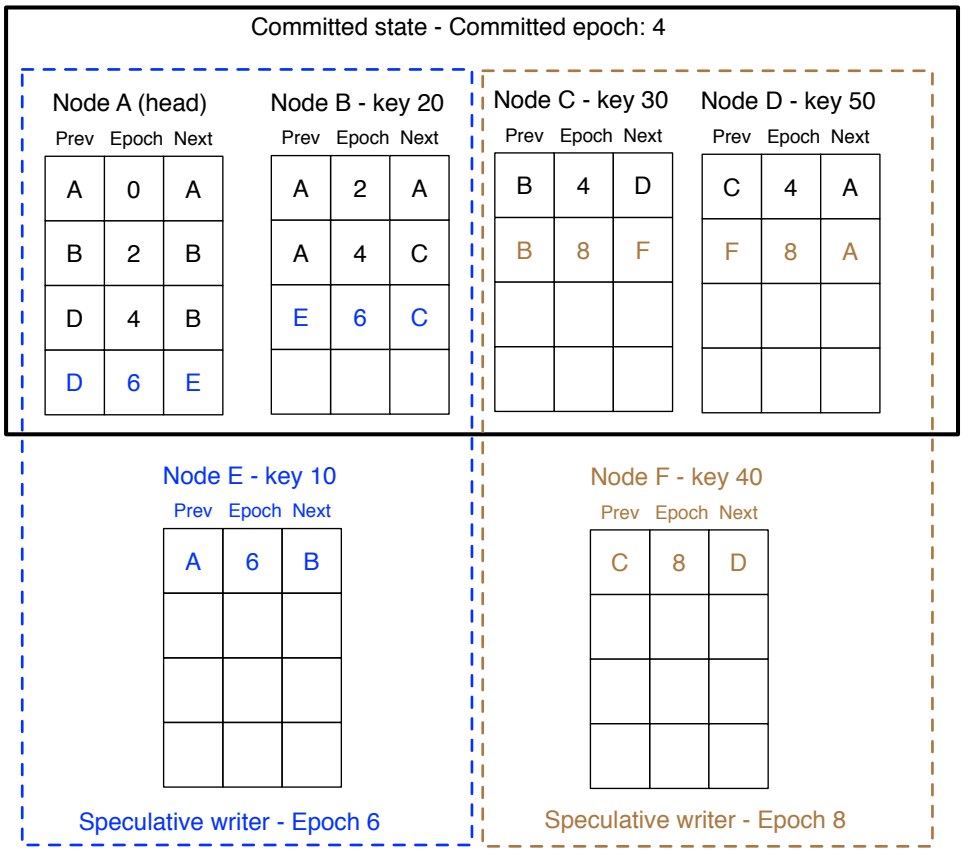

Figure 8.5: Two concurrent, speculative writers in an OLF list. One thread with epoch 6 is speculatively adding a node between nodes A and B, and another is adding a node between C and D. Writes to disjoint nodes are allowed to proceed in parallel, but must be committed in epoch order. Readers will ignore speculative entries until they are committed.

Slot reclamation. The number of slots in the list node is limited to four by default in our implementation. Thus, slots must eventually be reclaimed. Intuitively, a writer can reclaim a slot if it is sure that the oldest reader will not read it. In order to assess this, the algorithm maintains a shared variable, `read_epoch`, a potentially stale cache of the oldest reader's epoch. If a list node has two or more slots with an epoch less than or equal to `read_epoch`, only the most recent of these slots will be read by any threads. Thus, the older slots may be reclaimed, as described further in §8.2.4. If the `read_epoch` is 4 in Figure 8.5's example, the first and second slots

Goal	Invariant
A slot is not reclaimed until all readers will use a later slot (i.e., readers can safely traverse lists without locks).	`read_epoch` $\leq$ $\{e \mid e$ is the epoch of any active reader$\}$ $\leq$ `committed_epoch`
The oldest writer can always make progress.	Each list node has at least one slot the oldest writer can allocate, potentially after waiting for readers with epochs less than the committed epoch complete.

Table 8.3: Key correctness goals for the OLF algorithm, and list invariants that ensure them.

in Node A (epochs 0 and 2) can be reclaimed by a writer.

The rest of this section describes OLF in more detail. §8.2.1 describes the data structure invariants that ensure safe, concurrent access. §8.2.2 presents OLF usage examples, and §8.2.3 describes the steps needed to begin and end a critical region. §8.2.4 describes the `acquire_entry` function, which creates new versions within a list node. §8.2.5 explains where memory barriers are required in the algorithm on the x86 architecture, §8.2.6 explains how the algorithm, simplified for presentation here, can be extended to recover from thread failures, §8.2.7 concludes with discussion of design issues.

8.2.1 Key invariants

Most of the complexity of the OLF algorithm comes from careful management of slots. Table 8.3 shows the two key list node invariants that prevent deadlock as well as prevent unsafe traversal of the list.

A slot must not be reused if another thread might still read its pointers. The first invariant, that `read_epoch` is less than or equal to the epoch of any reader, ensures that slots are not reclaimed while a reader can access them. Recall that `read_epoch` is used by writers to determine when a slot can be reclaimed. As long as this value is less than or equal to the oldest reader's epoch (the left inequality),

writers will not overwrite pointers a reader is trying to follow. Reclamation is discussed in more detail in §8.2.4.

Similarly, speculative writers should not consume available slots to the point that writers deadlock waiting for slots. For example, suppose two writers wish to modify the same two list nodes, and each list node has only one free slot. If each writer allocates one of the two slots, the two can deadlock. The second invariant is that younger writers always reserve a reclaimable slot for the oldest (i.e., next to commit) writer. If a slot is not available, a thread may spin until the `committed_epoch` or `read_epoch` advances. This invariant ensures that the oldest writer can always make progress and eventually commit. By induction, all writing threads will eventually make progress, as critical regions commit in the same order they started.

8.2.2 Code example

This subsection describes several examples taken from the port of the Linux file system directory cache (`dcache`) to use OLF lists, illustrating relatively straightforward changes required to use OLF. Figure 8.6 shows the conversion of the Linux `list_add` function and `list_for_each` iterator to the OLF equivalents. Figure 8.7 provides key structure definitions used in code and pseudo-code examples throughout the rest of the chapter. Note that both the interface and implementation are quite similar. The main difference is that the pointers to be manipulated (`slot_t` structures) must first be allocated and initialized, using the `acquire_entry` routine, described in §8.2.4. Figure 8.6 also shows the conversion of a potentially read-only iterator (`list_for_each`) to OLF equivalent, using the much simpler `read_entry` function.

Programmers using the OLF list must explicitly annotate the beginning and end of critical regions, as illustrated in Figure 8.8. Writers use the functions

```
1  static inline void olf_list_add(struct olf_list_head *new,
2                                  struct olf_list_head *head){
3      struct olf_list_head *next;
4      struct slot_t *prev_e, *new_e, *next_e;
5
6      prev_e = acquire_entry(head);
7      new_e = acquire_entry(new);
8      next = prev_e->next;
9      next_e = acquire_entry(next);
10
11     // Quit early and retry if writer is invalidated
12     if (need_retry) return;
13
14     next_e->prev = new;
15     new_e->next = next;
16     new_e->prev = prev;
17     prev_e->next = new;
18 }
19
20 /* Original list_add implementation and interface
21  * Helper function __list_add inlined for clarity. */
22 static inline void list_add(struct list_head *new,
23                             struct list_head *head) {
24     struct list_head *prev, *next;
25
26     prev = head;
27     next = prev->next;
28
29     next->prev = new;
30     new->next = next;
31     new->prev = prev;
32     prev->next = new;
33 }
34
35 #define list_for_each(pos, head)                \
36         for (pos = (head)->next; pos != (head); \
37              pos = pos->next)
38
39 #define olf_list_for_each(pos, head)            \
40         for (pos = read_entry(head)->next;      \
41              pos != (head);                     \
42              pos = read_entry(pos)->next)
```

Figure 8.6: olf_list_add and olf list iterator implementations in the Linux kernel. Retains similar interface and implementation as original list_add and list_for_each, also depicted for comparison.

`write_cs_enter()` and `write_cs_exit()`, while readers use the `read` variants, described in §8.2.3. In adding OLF lists to the Linux kernel, these generally replaced lock and unlock calls on the coarse-grained lock protecting the cache-related lists.

A second difference introduced by optimistic execution is that a writer can fail and retry if it modifies the same list node as a concurrent writer with an earlier

```
1  struct slot_t {
2       struct olf_list_head *next, *prev;
3       int attempt;
4       atomic_long_t epoch;
5  };
6  struct olf_list_head { /* List node */
7       struct slot_t entries[MAX_SPECULATION]; /* 4 */
8  };
9  struct olf_list_record { /* Log entry */
10      struct olf_list_head *head;
11      int index;
12      unsigned long copy_epoch;
13 };
14 struct olf_list_info { /* Thread-local bookkeeping */
15      volatile unsigned long epoch;
16      int flags; /* writer, need_retry, attempt count*/
17      int olf_record_count; /* undo log length */
18      struct olf_list_record *olf_records; /* undo log */
19 };
```

Figure 8.7: OLF data types. The `olf_list_head` replaces a `list_head` structure in a kernel data structure, such as a `dentry`. Individual readers or writers operate on `slot_t` structures. The `olf_list_info` is a per-thread data structure used to track critical section writes and other bookkeeping.

epoch. In modifying Linux to use OLF lists, write critical regions typically execute in a `do` loop until they can commit. For example, Figure 8.8 shows a modified version of `d_instantiate`, which associates an `inode` with a newly created `dentry`. A `dentry` (directory entry) represents a file name in cached in memory. This function acquires and releases the `dcache_lock` in Linux, but in this code example, places the `d_alias` list modification in a write critical region loop.

8.2.3 Beginning and ending critical sections

When a thread begins a read critical section, the thread sets its thread-local epoch to the currently committed epoch. As the reader encounters a list node, it selects the slot with the greatest epoch less than or equal to the thread's epoch. When a thread exits a read critical section, the thread marks its epoch as inactive by setting the lower bit of its epoch. Because the low bit of an epoch number indicates an inactive speculation, epoch values increase by 2.

Because readers always select the slot with the greatest epoch less than or

```
1  void d_instantiate(dentry *entry, inode * inode)
2  {
3     dentry->d_inode = inode;
4     if (inode) {
5       write_cs_enter();
6       do {
7          olf_list_add(&dentry->d_alias, &inode->i_dentry);
8       } while (write_cs_exit());
9     }
10    fsnotify_d_instantiate(dentry, inode);
11    security_d_instantiate(entry, inode);
12 }
```

Figure 8.8: OLF usage example, from the modified Linux source code. `d_instantiate` is used to associate a new dentry with an inode. Simplified slightly for clarity, including expansion of macros and inlined functions.

equal to their thread-local epoch, writers can assess which slots in a list node can be reused without harming any readers. This assessment is based on the oldest reader's epoch, cached in the shared counter `read_epoch`. The value of `read_epoch` is lazily updated when a writer is under space pressure by scanning each thread's epoch.

A write critical section begins by allocating a new epoch through an atomic increment of `max_epoch`. The end of a write critical section is a bit more complex, as it is the key point at which all speculative operations are serialized. The `write_cs_exit()` function consists of three key steps: 1) It waits for previous writes to complete. 2) It walks its undo log to check that a concurrent writer has not invalidated any list nodes 3) If valid, it increments `committed_epoch`. If invalid, it sets the low bit on each slot it created. Writers must detect conflicting updates to a list node in order to prevent lost updates, which can occur if two concurrent writers initialize a speculative slot from the same committed state (i.e., the older writer's updates would be effectively ignored by the younger writer). When a writer initializes a slot, it records the epoch of the committed slot it copied to initialize the new slot. A writer detects a missed update by an earlier writer when it is committing by scanning the slots in a list node for any epochs between the thread's epoch and its logged copy of the epoch for that slot.

8.2.4 The `acquire_entry` function

At the heart of the OLF algorithm is the `acquire_entry` function, which allocates a slot in a list node. Figure 8.9 provides high-level pseudo-code for this function, which this section walks through, and Figure 8.10 provides pseudo-code for a key helper function. Note that readers call a much simpler function (`read_entry`), that simply selects the slot with the greatest epoch less than or equal to the reader thread's epoch.

The `acquire_entry` function consists of the following key steps: 1) Record the list head in an undo log, 2) Allocate a slot, and 3) Initialize the slot. The undo log for a thread is a straightforward array of pointers in thread-local storage (listed as `olf_records` in Figure 8.7). After logging the entry, the remaining code executes in a loop until it either acquires and initializes a slot or fails and subsequently retries the entire write critical section.

The second step of `acquire_entry` (Figure 8.10) determines the number of reclaimable slots, the best candidate for reclamation (`to_alloc`), and the most recent committed slot to copy into the newly allocated slot (`to_copy`). An entry is safe to reclaim if there is a more recent committed entry in the node and there are no readers that could be reading this entry. A slot is also safe to reclaim if it was invalidated by a failed writer that has subsequently committed without reusing the slot.

A writer only attempts to allocate a slot if it is the oldest writer or it can reserve at least one more slot available for the oldest writer. If more space is required, a writer may update the value of `read_epoch` (Figure 8.9, Line 20) to the oldest reader's epoch by by reading each thread's private epoch value and calculating the minimum. If a writer gets to the point that space is low and it isn't the next to commit, the writer stalls until it is the next to commit (Line 24). This heuristic typically yields marginally higher performance by avoiding bus traffic that is likely

```
1  struct slot_t *acquire_entry(struct olf_list_head *head) {
2     struct slot_t *entry = NULL, *to_copy, *to_alloc;
3     int new = 1;
4     unsigned long cached_max_epoch, alloc_epoch, copy_epoch;
5     /* Record this entry in writer's private log. */
6     thread_log(head);
7     do {
8       to_copy = to_alloc = NULL;
9       int count = -1;
10
11      if (find_slot_to_alloc(&count, &to_copy, &copy_epoch,
12                  &to_alloc, &alloc_epoch, &entry, &new))
13         return NULL;
14
15      /* If space is tight, advance the shared read epoch
16       * to enable garbage collection. */
17      if ((!entry) &&
18          (count < 1 ||
19           (count < 2 && !next_to_commit()))) {
20         advance_read_epoch();
21         if (count < 1 || read_epoch not advanced)
22            continue;
23         /* Throttle unlikely-to-complete writers */
24         while (!next_to_commit()) rep_pause();
25         if (count < 1)
26            continue;
27      }
28
29      if (!entry) {
30         /* Try to allocate an entry. */
31         if (alloc_epoch ==
32             CAS(&to_alloc->epoch, alloc_epoch, my_epoch))
33           entry = to_alloc;
34         else continue;
35      }
36
37      /* Init the entry if it is new. */
38      if (entry && new) {
39         *entry = *to_copy;
40         /* Make sure to_copy didn't change */
41         if (to_copy->epoch != copy_epoch) {
42           continue;
43         log_copy_epoch();
44      }
45    } while (entry == NULL);
46    return entry;
47 }
```

Figure 8.9: High-level pseudo-code for the `acquire_entry` function, which allocates and initializes an `slot_t` from an `olf_list_head`.

to be fruitless, but this heuristic isn't necessary for correctness.

Because younger writers always reserve space for the oldest writer, the only

```
1  int find_slot_to_alloc(int &count, struct slot_t *&to_copy,
2                         unsigned long &copy_epoch, struct slot_t *&to_alloc,
3                         unsigned long &alloc_epoch, struct slot_t *&entry,
4                         int &new) {
5     unsigned long cached_read_epoch = read_epoch;
6     unsigned long cached_committed_epoch = committed_epoch;
7
8     for (i = 0; i < MAX_SPECULATION; i++) {
9        unsigned long entry_epoch = head->entries[i].epoch;
10
11       if (entry_epoch == current_epoch) {
12          /* We can re-access our own written entry */
13          entry = &head->entries[i];
14          if (!alloc) { new = 0; return 0; }
15       }
16
17       if (entry_epoch <= cached_committed_epoch | INACTIVE) {
18          if (((!to_copy) || entry_epoch > to_copy->epoch)
19              && (entry_epoch & INACTIVE) == 0)
20          to_copy = &head->entries[i];
21
22          /* Select the slot to allocate in this preference:
23           * 1 An invalid slot from our previous attempt.
24           * 2 An invalid slot from a committed writer.
25           * 3 The oldest valid slot. */
26           if ( ((entry_epoch & INACTIVE) != 0)
27                || (entry_epoch <= cached_read_epoch)) {
28                 if (alloc_epoch != (current_epoch | INACTIVE)
29                     && ((!to_alloc)
30                       || ((entry_epoch & INACTIVE) && (alloc_epoch & INACTIVE)==0)
31                       || (((entry_epoch & INACTIVE) || (alloc_epoch & INACTIVE)==0)
32                           && entry_epoch < alloc_epoch))){
33                   to_alloc = &head->entries[i];
34                   alloc_epoch = entry_epoch;
35                 }
36                 count++;
37          }
38       } else if (entry_epoch == (current_epoch | INACTIVE)) {
39          if (head->entries[i].attempt != current_attempt) {
40             count++;
41             to_alloc = &head->entries[i];
42             alloc_epoch = entry_epoch;
43          } else {
44             /* This write attempt has been invalidated */
45             set_need_retry(); return -1;
46       } else if (entry_epoch < current_epoch) {
47          /* Stall for previous writer to commit */
48          while (committed_epoch < entry_epoch) spin;
49          i = 0; continue;
50 }} return 0; }
```

Figure 8.10: Pseudo-code for `acquire_entry` helper function that finds a slot to allocate. This function checks whether the writer already has a slot and if not, identifies the best slot to allocate (`to_alloc`) and which slot to copy (`to_copy`). Returns 0 on success, -1 on error.

case where the oldest writer can block is waiting for old readers to complete so that the writer can reclaim a slot. One slot in any list node will contain the most recently committed state. Additional slots may contain older versions that might be accessed by an older reader, which can only be reclaimed when these readers exit. Eventually all readers older than the committed epoch will exit and the slot can be reclaimed.

A writer allocates a slot in the list node by CAS-ing its epoch value to the thread's epoch. If the CAS fails the writer retries the loop, as another writer has acquired this slot and all assumptions about the state of the node may be stale. If the writer successfully allocates the slot, the writer initializes the slot from the contents of the previously committed slot. After initializing entry, the writer logs the epoch of the slot it copied from; this is used to detect intervening updates during commit. On a retry, the previously allocated slot will be used. Once the slot is initialized, the writer will break out of the loop and return a reference to the initialized slot.

8.2.5 Memory barriers

On the x86/64 architecture, the C implementation of the OLF algorithm uses atomic instructions (i.e., instructions with the `lock` prefix) to update shared epoch counters and slot epochs. The OLF algorithm also requires a single write memory barrier (`sfence` instruction). Because the x86 ISA provides total store order (TSO)[AG96] memory consistency, the need for memory barriers is fairly rare, as are examples of when and how to use memory barriers other than Dekker's algorithm [Dij65]. For both completeness, and for a new example, this subsection explains the use of memory barriers in OLF[1].

Recall that `read_epoch` must be less than or equal to the epoch of the oldest reader. Because `read_epoch` constrains which list entries can be reclaimed, violating

[1]The Java implementation of OLF uses `volatile` types for synchronization variables. Volatile types in Java provide a stronger memory model than TSO. To implement this consistency model, the JVM on the x86 architecture automatically issues more memory barriers than strictly necessary for OLF.

```
1  start: Thread1_epoch = ⊥                                                    1
2  read committed_epoch (=2)                                                   2
3                              inc committed_epoch,2 (=4)                      3
4                                                    read committed_epoch (=4) 4
5                                                    read Thread1_epoch (=⊥)   5
6                                                    write read_epoch 4        6
7  write Thread1_epoch, 2                                                      7
```

Thread 1: Beginning read critical section.

Thread 2: Committing write critical section.

Thread 3: Writer calling `advance_read_epoch`.

Figure 8.11: Reads and writes from three code regions that can form a subtle race condition in the OLF list algorithm. Line numbers indicate ordering in time, parentheses indicate the value of a variable.

```
1  do {
2     my_thread_epoch = committed_epoch;
3     sfence();
4  }  while (my_thread_epoch != committed_epoch);
```

Figure 8.12: Modified code for beginning a read critical section that eliminates the race condition illustrated in Figure 8.11. Note that without the `sfence`, this code becomes equivalent to the race-prone code in Figure 8.11.

this invariant can allow a writer to overwrite an entry that an active reader is reading, leading to incorrect behavior.

Recall that a reader begins a critical section by writing the currently committed epoch into its thread-local epoch, which can also be scanned by writers to update `read_epoch`. Because there is no atomic memory-to-memory move instruction on the x86, a reader cannot atomically set its epoch to the `committed_epoch` value, leading to the race condition illustrated in Figure 8.11. Essentially, between Thread 1 reading `committed_epoch` and writing the value (2) back to its thread epoch, Thread 2 can commit a write critical section (incrementing `committed_epoch` to 4). Then another writer (Thread 3) can try to set the `read_epoch` value. In `advance_read_epoch()`, Thread 3 reads the new value of `committed_epoch` (4), and each reader's epoch (Thread 1 is still inactive, represented by $\perp$), writing the minimum (4) to `read_epoch`. Thread 1 then writes 2 as its epoch, violating our invariant (a reader should not have an epoch less than `read_epoch`).

A straightforward solution would put the assignment in a loop that retries the assignment if the value of `committed_epoch` changes during the move, as shown in Figure 8.12. Even in a loop, this code is still prone to the same race without the `sfence` instruction (write barrier). TSO allows loads to be serviced before a pending store is retired from a CPU's store buffer. Thus, the loop condition in Line 4 can be checked and the loop terminated before the update of the thread's epoch is globally visible, making this code isomorphic to the race condition illustrated in Figure 8.11. Adding the write barrier ensures that the update of the thread's epoch is globally visible before checking that `committed_epoch` has not changed.

8.2.6 Failure recovery

The strict definition of a lock-free (or, non-blocking) algorithm is one that guarantees forward progress of *some* thread even in the presence of thread failures. As presented above, and implemented in the prototype, a thread failure can prevent progress. Note that in the context of an OS kernel or other low-level software, one may wish to decouple the scalability benefits of a lock-free algorithm without tolerating thread failures, as OS kernels are designed to immediately stop upon detecting errors to minimize the risk of data corruption. This subsection describes simple extensions that ensure system-wide progress in the presence of failures.

Assuming a mechanism to detect thread failures, one can remove any obstruction from a failed reader simply by setting its low (inactive) bit or by removing it from the list scanned to update `read_epoch`. To remove a writer, one must also traverse its undo log, invalidate any slots it has acquired, and then commit its epoch. Because each of these steps would be implemented with a CAS instruction, any number of blocked threads can perform them concurrently and safely.

8.2.7 Design issues

Shared state The OLF algorithm is presented with three epoch counters and a list of threads as global state. In a more complex application with many lists, one may wish to avoid stalling writers on updates to completely independent lists. One mitigates this by sharing state with a set of lists, introducing similar coding complexity to a lock variable, but with different performance characteristics.

Similarly, in the presentation of updating `read_epoch`, a writer may have to traverse all threads' local epochs. Some applications may have specialized threads and would benefit from restricting the list of threads traversed. Our experience is that lazily updating `read_epoch` is an important performance optimization.

Nested critical sections The OLF implementation permits read critical sections to be nested inside of write critical sections. When a write critical section traverses a list, it allocates slots in each node accessed. Nesting a read critical section allows this overhead to be elided, at the additional complexity of validating that any list nodes found in the nested read critical section have not been changed by an earlier writer. Note that in a garbage collected language or in C with a quiescence-based memory reclamation scheme (as used in the Linux kernel and all C microbenchmarks used to evaluate the list), list nodes will will not be freed until the writer has released its references to the node. In our experience, searching a list in a read critical section nested within a write critical section is a valuable optimization.

Debugging An interesting property of programming with a concurrent, versioned data structure is that concurrency errors are often easier to debug. In debugging the typical concurrency error on a traditional data structure, it is hard to reconstruct the state at the instant something went wrong. In OLF, the needed state is likely to persist, as each list node stores up to the last four modifications. Moreover, a common failure mode of earlier versions of the OLF implementation was for all

threads to hang at commit or trying to allocate a slot. This fail-stop mode combined with the old versions greatly simplified the task of reconstructing error behavior.

8.3 Correctness sketch

This section presents proof sketches for the correctness of the OLF list algorithm. To show correctness, readers must see a consistent view of the data structure and writers must be properly serialized. For liveness, we argue that readers can always make progress and that the oldest writer can always make progress. By induction, all writers will eventually complete and system-wide progress is assured. These properties are upheld by the data structure invariants described in Section 8.2.1.

Readers see a consistent view of lists. In order for readers to see a consistent view of a data structure, 1) readers must ignore slots containing "later" updates to the data structure and 2) writers may not overwrite an earlier slot that a reader still requires. The `read_entry` function gives a fairly clear guarantee that readers will ignore slots marked with later epochs.

Section 8.2.4 discusses the conditions under which a slot may be overwritten. Essentially, the pseudo-code of Figure 8.9 includes logic that tracks the most recent slot older than or equal to `read_epoch`. This slot may not be overwritten, nor can any valid slots with an epoch after or equal to `read_epoch`. Given the invariant discussed in Section 8.2.5 that `read_epoch` $\leq \{e | e$ is the epoch of any active reader$\}$, the most recently written list slot less than or equal to any given reader's epoch will be available at each list node.

Note that epochs in the prototype implementation are represented with 64-bit integers. There is a theoretical possibility, unlikely in practice, that a particularly old slot could persist across two wrap-arounds of the global epoch counter. Assuming an extremely efficient machine were able to process one trillion write critical sections

per second (cf. current CPUs process at most 4 billion instructions per second), a double wrap-around would occur after 5.8×10^{11} years.

Writers are serialized. The design of the OLF list algorithm commits each writer in epoch order. A later writer that completes its critical region will spin until its immediate predecessor completes. This simplifies our reasoning about serializability, in that we only need to show that each allocated list slot is initialized from the version immediately preceding it (i.e., no lost updates). This is ensured by logging the epoch from which a slot was initialized, and verifying during commit that no slots are present with an epoch between the logged and current epoch.

System-wide progress. The OLF list algorithm ensures system-wide progress in the absence of failures. Section 8.2.6 describes simple modifications that allow correct threads to recover from another thread's failure.

The only condition in which a reader can be obstructed by another thread is when setting its epoch initially, as it can spin until `committed_epoch` stabilizes long enough to set its thread-local epoch (Figure 8.11). Starvation is theoretically possible as the algorithm is presented, if enough threads could commit writes fast enough to always be changing `committed_epoch` before a reader can initialize its epoch. This can be prevented by "upgrading" a reader to a writer after a bounded number of retries. In practice, the work required to commit an epoch is always going to be substantially larger than the work required to initialize a reader epoch, so we expect the risk of read starvation without write upgrades to be vanishingly small.

For writers, we show that the oldest writer can always make progress, and, by induction, all writers will eventually commit. This demonstrates both system-wide progress and fairness to each writer. A writer is only obstructed by waiting for older writers to commit before committing itself, or during `acquire_entry()` if it cannot allocate a slot. Because a writer will not allocate a slot unless there is another

available slot for the oldest writer, speculative writers will not block the oldest writer indefinitely in `acquire_entry`. Old readers can prevent the oldest writer from acquiring a slot, but only if there are at least two committed slots that cannot be reclaimed. Since all readers eventually complete and new readers initialize their epoch to the `committed_epoch`, the oldest writer will eventually reclaim one of the committed slots and make progress. Because every writer will eventually become the oldest uncommitted writer, the algorithm assures progress for all writers. Because the algorithm assures that readers can always make progress and that the oldest writer will make progress, the OLF list algorithm ensures system-wide progress.

8.4 Evaluation

We evaluate the performance of C and Java implementations of OLF. We first measure its performance in a microbenchmark developed by Hart et al. [HMBW07] and describe case studies using OLF in Java applications. We then describe the application of OLF to the Linux kernel's `dcache` and measure the kernel's scalability. All data are the average of 3 or more tests.

8.4.1 Microbenchmark performance

We compare the performance of the OLF algorithm to the Harris-Michael algorithm and to locking a list. We use a C microbenchmark developed by Hart et al. [HMBW07] to test the performance of a list implementation under various mixes of read and write operations. For the Harris-Michael algorithm in C, we compare against two reclamation schemes, hazard pointer-based reclamation [Mic04a] and quiescence-based reclamation (similar to RCU's reclamation). We also ported the benchmark to Java, and wrote two locked list classes: one that uses the `synchronized` keyword and one that uses a spinlock. For the locking baseline, we only use a coarse-grained lock that protects the single list. We use the implementation of the Harris-

C	Read-only					10% Write				
Threads	**1**	**2**	**4**	**8**	**16**	**1**	**2**	**4**	**8**	**16**
Spinlock	159	672	2776	15686	37909	163	898	3328	18887	46212
OLF	360	362	362	360	362	694	776	867	1240	3333
HM (hzrd ptrs)	893	897	895	894	895	889	923	959	1050	1381
HM (quiescence)	142	149	146	145	147	156	208	208	494	1000
Java	**Read-only**					**10% Write**				
Threads	**1**	**2**	**4**	**8**	**16**	**1**	**2**	**4**	**8**	**16**
Spinlock	163	569	2526	10108	20452	118	492	3529	13304	34957
`synchronized`	161	780	2072	4088	7372	114	622	2225	2899	7289
OLF	835	1063	1059	970	903	1376	1605	2455	2355	3263
`ConcSkpLstSet`	266	279	339	602	1067	301	389	532	958	1827

C	100% Write				
Threads	**1**	**2**	**4**	**8**	**16**
Spinlock	185	1763	7158	27682	66297
OLF	971	1720	2493	4749	20448
HM (hzrd ptrs)	878	1219	1531	2298	3721
HM (quiescence)	213	538	867	1780	3484
Java	**100% Write**				
Threads	**1**	**2**	**4**	**8**	**16**
Spinlock	133	1191	10510	24850	41937
`synchronized`	129	1306	2854	5397	9192
OLF	2029	3941	5358	7981	21816
`ConcSkpLstSet`	419	1171	2010	4313	10099

Table 8.4: Comparison of locked and lock-free list implementations in C and Java. Workload scales input size with additional threads. Numbers are average execution time of an operation times the number of CPUs in ns (lower is better). Numbers are provided for a read-only workload, a 10% read workload, and a write-only workload. HM is the Harris-Michael algorithm; in C both hazard pointers and quiescence-based reclamation are compared.

Michael algorithm from the Java Concurrency Package's `ConcurrentSkipListSet`. In order to fairly compare the implementations, we reduced the skiplist code to a simple list. These measurements were taken on a 16-core SuperMicro SuperServer, with four 4-core Intel Xeon X7350 chips running at 2.93 GHz and 48 GB of RAM.

Table 8.4 lists the performance of each list implementation traversing a 100-node list. Columns compare read-only, 90% read/10% write, and write-only workloads at thread counts between 1 and 16. Each cell is the average execution time of an operation times the number of CPUs (i.e., the same value across CPU counts indicates perfect scaling).

In general, the Java implementation is less tuned than the C implementation, which manifests primarily in the read-only and 10% write numbers. Another anomaly in the numbers is that the `synchronized` list performed particularly well under high write contention (even outperforming Harris-Michael), whereas the Java and C spinlocks degrade performance similarly. We suspect that this is due to a backoff heuristic that puts threads to sleep under high contention. This microbenchmark only calculates aggregate throughput and does not require all threads to perform equal numbers of requests; this backoff heuristic may yield numbers that are hard to achieve when migration of work between threads is constrained.

OLF dominates the performance of spinlocks and typically performs comparably to or somewhat worse than Harris-Michael. On read-only workloads, the performance of all lock-free algorithms is insensitive to CPU count, and OLF even outperforms Harris-Michael with hazard pointers in C. A key observation of Hart et al. is that the overheads of a lock-free list algorithm are dominated by atomic operations and barriers; because quiescence involves fewer atomic operations and barriers than hazard pointers it will generally have lower overheads. In the read-only case, the quiescence outperforms the OLF algorithm by roughly 200ns and the OLF algorithm outperforms hazard pointers by roughly 500ns.

On write-heavy workloads, OLF generally performs worse than either version of the Harris-Michael algorithm as CPU counts increase, although performance degrades for all algorithms as more threads are added. The OLF algorithm substantially outperforms spinlocks, however. At 16 threads in C, the 100% writer and the 10% writer OLF implementations outperform spinlocks by roughly a factor of 3.2× and 13.7× respectively.

Note that the comparison with Harris-Michael disadvantages the OLF algorithm, which incurs higher overheads in order to provide features that the Harris-Michael algorithm does not. Thus, Figure 8.13 shows an additional microbench-

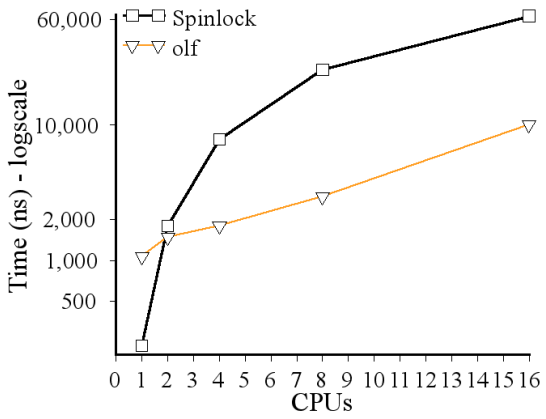

Figure 8.13: List move microbenchmark, which scales input size with threads. Y-axis is execution time in logscale, lower is better. Note that previous lock-free algorithms cannot provide atomic move between lists, so only spinlocks and OLF are compared.

mark that measures the performance of moving elements between two lists, both protected by spinlocks and OLF. Previous lock-free algorithms cannot safely execute this microbenchmark. As in the other benchmarks, the OLF algorithm substantially outperforms spinlocks at high CPU counts, showing a factor of 6.3× improvement at 16 CPUs. In summary, the OLF algorithm allows better scalability than spinlocks in situations where locking is currently the only option programmers have.

8.4.2 Java application studies

In order to assess the macro-level performance and usability of OLF, we ported the xalan and tomcat benchmarks from the DaCapo suite [BGH+06], replacing synchronized lists with OLF. In both cases, the lists were not on the critical path, and the change yielded no discernible performance difference. We highlight, however, that in both applications, we were able to change on the order of ten lines of application code to adopt OLF. The primary changes were to remove `synchronized` keywords and change a few class names. This indicates that OLF can be easily adopted as a drop-in replacement for a standard list-based collection library with no adverse effect on macro-level performance.

8.4.3 Application to Linux

We modified Linux kernel version 2.6.34 for this case study. All experiments in this subsection were performed on a server with two quad-core Intel X5355 processors (total of 8 cores) running at 2.66 GHz with 8 GB of memory. Experiments were performed using an `ext2` file system on a 10,000 RPM SATA disk. We selected `ext2` because synchronous writes to a journaling file system (e.g., `ext3`) tend to make workloads I/O bound and obscure the scalability of the file system data structures. The server is running a 64-bit build of Ubuntu 10.04.

The key goal of this work is to make lock-free lists practical in an OS kernel. As a case study, we replaced the lists in the Linux `dcache` with OLF lists. The `dcache` caches portions of the file system directory tree in memory, using `dentry` structures to represent file names. Directory entries, or `dentries`, in this cache are on four different lists:

1. A least-recently used list for memory reclamation
2. An alias list to track the names of hard links to a file's inode
3. A subdirectory list, associated with the dentry of the parent directory, for traversing the hierarchical directory tree
4. A hash list, for fast lookup of a path name.

Because there are so many traversal paths through these lists, adding fine-grained locking is very difficult; thus the coarse-grained `dcache_lock` has proved difficult to excise [Cor09].

The reliance on a coarse grained lock to protect lists harms Linux scalability. Figure 8.14 shows a simple microbenchmark that creates and deletes 500,000 hard links to files in different directories. This microbenchmark stresses the scalability of the `dcache` specifically, as most file system benchmarks are typically I/O bound. At 4 threads or higher, performance degrades; profiling confirms that this is contention for the `dcache_lock`.

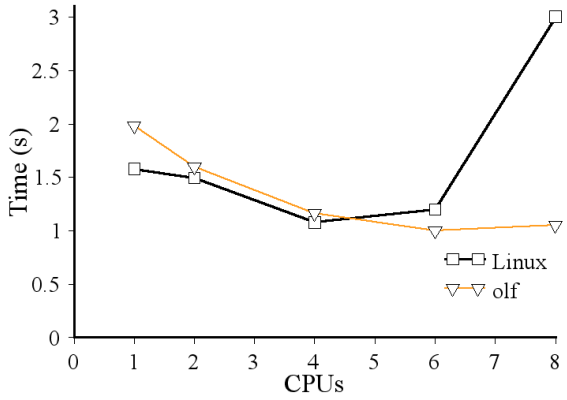

Figure 8.14: Link creation and deletion microbenchmark, showing execution time of 500,000 link/unlink operations divided across a number of threads. Lower is better.

By replacing `dcache_lock`-protected lists with OLF, we improve the scalability of the Linux kernel. OLF lists introduce higher single-threaded overheads, but tracks the performance of unmodified Linux at 2 CPUs and higher within .1 seconds or less. OLF outperforms Linux at 6+ CPUs, by a factor of 3× at 8 CPUs. With additional tuning effort, the single threaded overhead can likely be reduced. This experiment demonstrates that OLF lists provide a more scalable alternative to coarse-grained locking in the Linux kernel—currently coarse grained locking is the only option available to developers.

8.5 Related Work

Valois [Val95] is credited with the first practical lock-free design for a singly-linked list. Valois uses a single compare-and-swap (CAS) instruction to atomically insert or remove a node. This design requires at least one additional *auxiliary node* for each data-carrying list node; auxiliary nodes are used to detect inconsistencies that arise from concurrent modifications.

Harris [Har01] describes a simpler approach that eliminates the need for auxiliary nodes. Harris's key insight is that a deletion in a singly-linked list is safe if

both the previous next pointer and the one to be modified are unchanged. Because two pointers cannot be checked with a single CAS instruction, Harris provides a two phase deletion algorithm that uses two CAS instructions, first logically deleting the node and then physically deleting it.

Michael [Mic02] refined Harris's algorithm to eliminate reliance on reference counting garbage collection, making it compatible with several lock-free memory management methods, including hazard pointers [Mic04a]. The resulting algorithm, commonly called the Harris-Michael algorithm, is used in the Java Concurrency Package [HS08]. It was further refined by Fomitchev and Ruppert [FR04] to provide a better worst-case amortized cost of operations on the list.

Michael [Mic04b] also describes a lock free memory allocator that is compatible with the Harris-Michael algorithm in user-level applications. Like the Hoard allocator [BMBW00] and other user-space memory allocators, it relies on allocation of large regions of memory from the OS via system calls like `mmap`, which themselves generally block and use locks. Petrank et al. [PMS09] provide a formal framework that allows designers of user-level, lock-free algorithms to reason about lock-freedom of applications separately from system services and to demonstrate that their algorithm will be lock-free on the condition that the underlying system is as well

Read-copy update [McK04] (RCU) is a technique used in the Linux kernel that eliminates locking for read-only critical regions by carefully modifying pointers in a manner similar to the Harris-Michael algorithm. RCU lists inherit many of the same limitations as Harris-Michael, including the inability to atomically move nodes from one list to another. In addition, writers are mutually excluded with locks; unlike lock-free algorithms, the only performance scalability benefit is for read-mostly data structures.

Hart et al. [HMBW07] compare the performance of hazard pointers, the quiescence-based memory reclamation used in RCU, and other lockless memory

reclamation techniques. In most cases the frequency of atomic instructions and memory fences are the dominant performance cost. Quiescence and epoch-based schemes require the fewest atomic instructions and generally perform better, provided they occur sufficiently often to avoid memory exhaustion. The algorithm described in this chapter uses epoch-based reclamation [Fra04] for slots in an embedded list node; unlike previous uses, epochs in OLF dictate when a slot in a data structure can be reused, not when the entire data structure can be freed. The algorithm described in this chapter also requires more atomic instructions than RCU in order to permit more write parallelism, which yields higher single-threaded overheads.

Herlihy et al. [HLM02] present an abstract version of the memory management problem for dynamic, lock-free data structures. Similar to Harris's observation about logical versus physical deletion, they present a solution that defers physical deletion until all references to a logically deleted object are released. The OLF algorithm solves a highly constrained variant of the problem; rather than freeing dynamically-allocated nodes, OLF carefully recycles statically-allocated slots within list nodes.

Hohmuth and Härtig [HH01] explore non-blocking synchronization in the context of a microkernel, with a focus on wait-free locking. The only linked lists used in the microkernel are per-CPU, obviating the need for locks or lock-free lists. It is hard to compare the performance scalability of this system with a modern OS kernel like Linux, as their prototype does not support multiprocessing.

Universal constructions. The earliest algorithms for lock-free linked lists were Herlihy's universal constructions [Her91, Her93]; a number of universal constructions for lists and other data structures have followed in the literature. In general, these constructions can turn any sequential algorithm into a parallel algorithm with certain properties, such as wait-freedom [CER10]. These constructions are useful for

reasoning about asymptotic behavior and writing proofs, but the generality often comes at a substantial performance cost when compared to performance-tuned list algorithms. Universal constructions are rarely adopted in practical concurrent code bases.

DCAS-based algorithms. The Motorola 68k architecture provided a double word compare-and-swap (DCAS) instruction, which allowed any two words of memory to be atomically compared and swapped. Several research operating systems in the early 90s used this instruction to implement lock-free data structures, including linked lists [GC96, MP92]. Although quite useful for implementing lock-free algorithms, the DCAS instruction had performance issues and implementation challenges, and thus has not been adopted by any modern instruction set architecture. Best-effort hardware transactional memory might revive these algorithms if it achieves ubiquity in commodity systems [AMD09]. Current, practical systems that wish to reduce coarse-grained locking on lists must use algorithms that require only a single-word CAS instruction, or an equivalent.

8.6 Summary

Developers of concurrent applications currently face an unsavory choice between functionality and performance scalability when selecting list algorithms. The OLF list algorithm eliminates this unsavory choice by providing functionality previously available only with locking (e.g., arbitrary composition of list operations and no reliance on dynamic memory reclamation), but with performance closer to the current state of the art in lock-free lists, except under very high write contention. This chapter demonstrates that this algorithm can be used as a drop-in, and functionally enhanced replacement for list-based classes in Java, as well as adopted in the Linux kernel to improve kernel scalability.

Our hypothesis is that this approach to versioning nodes without locks can be generalized to other data structures, such as trees and arbitrary graphs, which we plan to explore in future work. This work carefully explains and evaluates the application of versioned nodes to linked lists—a simple data structure that nonetheless has frustrated the scalability of real-world applications.

Chapter 9

Conclusion

Developers face an ongoing struggle to write correct, concurrent programs. This struggle has been exacerbated by the increased prevalence of multi-core hardware. Not only must developers adopt threads or other error-prone concurrency techniques to improve application performance on successive generations of hardware, but even single-threaded programs must manage concurrency at the system-level. This thesis makes powerful abstractions for managing concurrency practical in modern systems and software.

First, this thesis shows that system transactions are a practical abstraction for a modern OS to provide to application developers. Application developers need system transactions to manage system-level concurrency and address problems for which current solutions are either *ad hoc* or non-existent. To make system transactions practical, this thesis introduces novel operating system kernel implementation techniques that make system transactions efficient and minimize the effort required to extend transaction support to additional resources, such as adding transactions to a given file system implementation. These implementation techniques avoid many of the issues in previous transactional operating systems, such as deadlock in the OS kernel, while maintaining a very simple and backwards-compatible API.

This thesis also contributes a new lock-free linked list algorithm, OLF, that facilitates writing concurrent applications. Unlike previous lock-free algorithms, OLF can increase the concurrency of an operating system kernel, benefitting all applications. Previous lock-free list algorithms sacrifice functionality for performance; losing, for instance, the ability to atomically move an element from one list to another. OLF lists restore this missing functionality without reintroducing locks. OLF lists also eliminate dependence on dynamic memory reclamation, making them suitable for use in an OS kernel. The OLF algorithm performs substantially better than spinlocks in most cases, and can be used in applications for which spinlocks are currently the only acceptable option.

The thesis demonstrates the practicality of these abstractions by implementing them in mature, highly-deployed software. The TxOS prototype extends the Linux kernel with system transactions and modifies real-world applications, such as OpenLDAP and the Debian package manager, to use system transactions. Similarly, we improve the scalability of the Linux kernel's directory cache with OLF lists. While the implementation techniques described in this thesis are relatively mature, the standards for adoption are quite high in developer communities for software such as the Linux kernel. Future work will focus on addressing the remaining barriers to adoption: lowering the performance overheads and demonstrating the value of these abstractions in additional applications. The abstractions developed in this thesis are important additions to the concurrent programmer's toolbox, better equipping him or her to write the concurrent applications of tomorrow.

Bibliography

[Ach] Steve Acheson. http://www.employees.org/~{}satch/ssh/faq/\TheWholeSSHFAQ.html.

[AG96] Sarita V. Adve and Kourosh Gharachorloo. Shared memory consistency models: A tutorial. *IEEE Computer*, 29(12):66–76, 1996.

[AMD09] AMD. Advanced synchronization facility proposed architectural specification. http://developer.amd.com/assets/45432-ASF_Spec_2.1.pdf, March 2009.

[AMS+07] Marcos K. Aguilera, Arif Merchant, Mehul Shah, Alistair Veitch, and Christos Karamanolis. Sinfonia: a new paradigm for building scalable distributed systems. In *Proceedings of the ACM SIGOPS Symposium on Operating Systems Principles (SOSP)*, pages 159–174, 2007.

[BDLM07] Colin Blundell, Joe Devietti, E. Christopher Lewis, and Milo M. K. Martin. Making the fast case common and the uncommon case simple in unbounded transactional memory. In *Proceedings of the ACM IEEE International Symposium on Computer Architecture (ISCA)*, pages 24–34, 2007.

[Ber] Daniel J. Bernstein. Using maildir format. http://cr.yp.to/proto/maildir.html.

[Ber07] Daniel J. Bernstein. Some thoughts on security after ten years of qmail 1.0. In *Proceedings of the ACM Workshop on Computer Security Architecture (CSAW)*, pages 1–10, 2007.

[Bes] Steve Best. JFS Overview. http://web.archive.org/web/20080129101603/http://www-128.ibm.com/developerworks/library/l-jfs.html.

[BGH+06] S. M. Blackburn, R. Garner, C. Hoffman, A. M. Khan, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, and B. Wiedermann. The DaCapo benchmarks: Java benchmarking development and analysis. In *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 169–190, 2006.

[BHG87] Philip A. Bernstein, Vassos Hadzilacos, and Nathan Goodman. *Concurrency Control and Recovery in Database Systems.* Addison-Wesley, 1987.

[BJSW05] Nikita Borisov, Rob Johnson, Naveen Sastry, and David Wagner. Fixing races for fun and profit: How to abuse atime. In *USENIX Security Symposium*, pages 303–314, 2005.

[BLM05] Colin Blundell, E Christopher Lewis, and Milo M. K. Martin. Deconstructing transactions: The subtleties of atomicity. In *Proceed-*

ings of the Workshop on Duplicating, Deconstructing, and Debunking (WDDD), Jun 2005.

[BMBW00] Emery D. Berger, Kathryn S. McKinley, Robert D. Blumofe, and Paul R. Wilson. Hoard: a scalable memory allocator for multithreaded applications. In *Proceedings of the ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2000.

[CBWK01] Crispin Cowan, Steve Beattie, Chris Wright, and Greg Kroah-Hartman. RaceGuard: Kernel protection from temporary file race vulnerabilities. In *USENIX Security Symposium*, pages 165–176, 2001.

[CER10] Phong Chuong, Faith Ellen, and Vijaya Ramachandran. A universal construction for wait-free transaction friendly data structures. In *Proceedings of the ACM symposium on Parallelism in algorithms and architectures (SPAA)*, pages 335–344, 2010.

[CGJ09] Xiang Cai, Yuwei Gui, and Rob Johnson. Exploiting unix file-system races via algorithmic complexity attacks. In *IEEE Symposium on Security and Privacy*, pages 27–41, 2009.

[CKL+09] Allen Clement, Manos Kapritsos, Sangmin Lee, Yang Wang, Lorenzo Alvisi, Mike Dahlin, and Taylor Riché. Upright cluster services. In *Proceedings of the ACM SIGOPS Symposium on Operating Systems Principles (SOSP)*, pages 277–290, 2009.

[CMTC+07] Chi Cao Minh, Martin Trautmann, JaeWoong Chung, Austen McDonald, Nathan Bronson, Jared Casper, Christos Kozyrakis, and Kunle Olukotun. An effective hybrid transactional memory system

with strong isolation guarantees. In *Proceedings of the ACM IEEE International Symposium on Computer Architecture (ISCA)*, pages 69–80, Jun 2007.

[Cor09] Jonathan Corbet. JLS: Increasing VFS scalability. *LWN*, November 2009. http://lwn.net/Articles/360199/.

[Cri03] M. Crispin. INTERNET MESSAGE ACCESS PROTOCOL - VERSION 4rev1. RFC 3501 (Proposed Standard), March 2003. Obsoletes by RFC 2060.

[DFL+06] Peter Damron, Alexandra Fedorova, Yossi Lev, Victor Luchangco, Mark Moir, and Dan Nussbaum. Hybrid transactional memory. In *Proceedings of the ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 336–346, 2006.

[DH04] Drew Dean and Alan J. Hu. Fixing races for fun and profit: How to use access(2). In *USENIX Security Symposium*, pages 14–26, 2004.

[Dij65] E.W. Dijkstra. Cooperating sequential processes. http://www.cs.utexas.edu/users/EWD/transcriptions/EWD01xx/EWD123.html, 1965.

[Dova] Dovecot maildir and racing. http://www.dovecot.org/list/dovecot/2006-March/011811.html.

[Dovb] Dovecot wiki: Mailbox format/ maildir. http://wiki.dovecot.org/MailboxFormat/Maildir.

[Dre08] Ulrich Drepper. Secure file descriptor handling. In *LiveJournal*, 08.

[DSS06] Dave Dice, Ori Shalev, and Nir Shavit. Transactional locking II. In *Proceedings of the International Conference on Distributed Computing (DISC)*, pages 194–208, 2006.

[FH07] Keir Fraser and Tim Harris. Concurrent programming without locks. *ACM Transactions on Computer Systems (TOCS)*, 25, May 2007.

[FR04] Mikhail Fomitchev and Eric Ruppert. Lock-free linked lists and skip lists. In *Proceedings of the ACM symposium on Principles of distributed computing (PODC)*, pages 50–59, 2004.

[Fra04] Keir Fraser. *Practical lock-freedom.* PhD thesis, University of Cambridge Computer Laboratory, 2004.

[GC96] Michael Greenwald and David Cheriton. The synergy between non-blocking synchronization and operating system structure. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 123–136, 1996.

[GGL03] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The Google file system. In *Proceedings of the ACM SIGOPS Symposium on Operating Systems Principles (SOSP)*, pages 29–43, 2003.

[GJR94] Narain H. Gehani, Hosagrahar V. Jagadish, and William D. Roome. OdeFS: A file system interface to an object-oriented database. In *Proceedings of the International Conference on Very Large Databases (VLDB)*, pages 249–260, 1994.

[GLPT76] Jim N. Gray, Raymond. A. Lorie, G. R. Putzolu, and Irving L. Traiger. Granularity of locks and degrees of consistency in a shared data base. *Modeling in Data Base Management Systems*, pages 364–394, 1976.

[GR93] Jim Gray and Andreas Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann, 1993.

[Gra78] Jim Gray. Notes on data base operating systems. In *Operating Systems, An Advanced Course*, pages 393–481. Springer-Verlag, 1978.

[GT05] Eran Gal and Sivan Toledo. A transactional flash file system for microcontrollers. In *Proceedings of the USENIX Annual Technical Conference*, pages 89–104, 2005.

[Har01] Timothy L. Harris. A pragmatic implementation of non-blocking linked-lists. In *Proceedings of the International Conference on Distributed Computing (DISC)*, pages 300–314, 2001.

[Her91] Maurice Herlihy. Wait-free synchronization. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 13(1):124–149, 1991.

[Her93] Maurice Herlihy. A methodology for implementing highly concurrent data objects. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 15(5):745–770, 1993.

[HH01] Michael Hohmuth and Hermann Härtig. Pragmatic nonblocking synchronization for real-time systems. In *Proceedings of the USENIX Annual Technical Conference*, pages 217–230, 2001.

[HK08] Maurice Herlihy and Eric Koskinen. Transactional boosting: A methodology for highly-concurrent transactional objects. In *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 207–216, 2008.

[HLM02] Maurice Herlihy, Victor Luchangco, and Mark Moir. The repeat offender problem: A mechanism for supporting dynamic-sized, lock-

free data structures. In *Proceedings of the International Conference on Distributed Computing (DISC)*, 2002.

[HLMS03] Maurice Herlihy, Victor Luchangco, Mark Moir, and William N. Scherer, III. Software transactional memory for dynamic-sized data structures. In *Proceedings of the ACM symposium on Principles of distributed computing (PODC)*, pages 92–101, 2003.

[HMBW07] Thomas E. Hart, Paul E. McKenney, Angela Demke Brown, and Jonathan Walpole. Performance of memory reclamation for lockless synchronization. *Journal of Parallel and Distributed Computing*, 67(12):1270–1285, 2007.

[HMC88] Rober Haskin, Yoni Malachi, and Gregory Chan. Recovery management in QuickSilver. *ACM Transactions on Computer Systems (TOCS)*, 6(1):82–108, 1988.

[HS08] Maurice Herlihy and Nir Shavit. *The Art of Multiprocessor Programming.* Morgan Kaufmann, 2008.

[HSATH06] Richard L. Hudson, Bratin Saha, Ali-Reza Adl-Tabatabai, and Benjamin C. Hertzberg. McRT-Malloc: A scalable transactional memory allocator. In *Proceedings of the ACM International Symposium on Memory Management (ISMM)*, pages 74–83, 2006.

[HW90] Maurice P. Herlihy and Jeannette M. Wing. Linearizability: a correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 12(3):463–492, 1990.

[HWC+04] Lance Hammond, Vicky Wong, Mike Chen, Brian D. Carlstrom, John D. Davis, Ben Hertzberg, Manohar K. Prabhu, Honggo Wi-

jaya, Christos Kozyrakis, and Kunle Olukotun. Transactional memory coherence and consistency. In *Proceedings of the ACM IEEE International Symposium on Computer Architecture (ISCA)*, pages 102–113, 2004.

[KAD+07] Ramakrishna Kotla, Lorenzo Alvisi, Mike Dahlin, Allen Clement, and Edmund Wong. Zyzzyva: Speculative byzantine fault tolerance. *Proceedings of the ACM SIGOPS Symposium on Operating Systems Principles (SOSP)*, pages 45–58, 2007.

[KPW+07] Milind Kulkarni, Keshav Pingali, Bruce Walter, Ganesh Ramanarayanan, Kavita Bala, and L. Paul Chew. Optimistic parallelism requires abstractions. In *Proceedings of the ACM SIGPLAN conference on Programming language design and implementation (PLDI)*, pages 211–222, 2007.

[Kur] Gerson Kurz. The ReiserFS Filesystem. http://p-nand-q.com/download/rfstool/reiserfs_docs.html.

[LCJS87] Barbara Liskov, Dorothy Curtis, Paul Johnson, and Robert Scheifer. Implementation of Argus. In *Proceedings of the ACM SIGOPS Symposium on Operating Systems Principles (SOSP)*, pages 111–122, 1987.

[Lis88] Barbara Liskov. Distributed programming in Argus. *Communications of the ACM (CACM)*, 31(3):300–312, 1988.

[LL97] James Laudon and Daniel Lenoski. The SGI Origin: A ccNUMA highly scalable server. In *Proceedings of the ACM IEEE International Symposium on Computer Architecture (ISCA)*, pages 241–251, 1997.

[LL98] Daniel Lenoski and James Laudon. Retrospective: The Dash prototype: Implementation and performance. In *25 years of the International Symposia on Computer Architecture (ISCA) (selected papers)*, pages 80–82, 1998.

[LR06] James Larus and Ravi Rajwar. *Transactional Memory*. Morgan & Claypool, 2006.

[LSP82] Leslie Lamport, Robert Shostak, and Marshall Pease. The byzantine generals problem. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 4(3):382–401, 1982.

[MBM+06a] Kevin E. Moore, Jayaram Bobba, Michelle J. Moravan, Mark D. Hill, and David A. Wood. LogTM: Log-based transactional memory. In *Proceedings of the IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 254–265, Feb 2006.

[MBM+06b] Michelle J. Moravan, Jayaram Bobba, Kevin E. Moore, Luke Yen, Mark D. Hill, Ben Liblit, Michael M. Swift, and David A. Wood. Supporting nested transactional memory in LogTM. In *Proceedings of the ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 359–370, Oct 2006.

[MBS+08] 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 *Proceedings of the ACM SIGPLAN Workshop on Transactional Computing (TRANSACT)*, 2008.

[MCC+06] Austen McDonald, JaeWoong Chung, Brian D. Carlstrom, Chi Cao

Minh, Hassan Chafi, Christos Kozyrakis, and Kunle Olukotun. Architectural semantics for practical transactional memory. In *Proceedings of the ACM IEEE International Symposium on Computer Architecture (ISCA)*, pages 53–65, 2006.

[McD08] Paul McDougall. Microsoft pulls buggy Windows Vista SP1 files. In *Information Week*, 2008. http://www.informationweek.com/story/showArticle.jhtml?articleID=206800819.

[MCE+02] Peter S. Magnusson, Magnus Christensson, Jesper Eskilson, Daniel Forsgren, Gustav Hållberg, Johan Högberg, Fredrik Larsson, Andreas Moestedt, and Bengt Werner. Simics: A full system simulation platform. *IEEE Computer*, Feb 2002.

[McK04] Paul E. McKenney. *Exploiting Deferred Destruction: An Analysis of Read-Copy Update Techniques in Operating System Kernels*. PhD thesis, Oregon Health and Science University, 2004.

[MCKO08] C. Cao Minh, J. Chung, C. Kozyrakis, and K. Olukotun. STAMP: Stanford Transactional Applications for Multi-Processing. In *Proceedings of the IEEE International Symposium on Workload Characterization (IISWC)*, 2008.

[Mic] Microsoft Corp. *.NET Framework Class Library Reference - Transaction Class*. http://msdn2.microsoft.com/en-us/library/system.transactions.transaction(VS.90).aspx.

[Mic02] Maged M. Michael. High performance dynamic lock-free hash tables and list-based sets. In *Proceedings of the ACM symposium on Parallelism in algorithms and architectures (SPAA)*, pages 73–82, 2002.

[Mic04a] Maged M. Michael. Hazard pointers: Safe memory reclamation for lock-free objects. *IEEE Transactions on Parallel and Distributed Systems*, 15(6):491–504, 2004.

[Mic04b] Maged M. Michael. Scalable lock-free dynamic memory allocation. In *Proceedings of the ACM SIGPLAN conference on Programming language design and implementation (PLDI)*, pages 35–46, 2004.

[Mic08] Microsoft. What is system restore? *Microsoft Support KnowledgeBase*, 2008. http://support.microsoft.com/kb/959063.

[Mor68] Donald R. Morrison. Patricia—practical algorithm to retrieve information coded in alphanumeric. *Journal of the ACM*, 15(4):514–534, 1968.

[Mos81] J. Eliot B. Moss. *Nested Transactions: An Approach to Reliable Distributed Computing*. PhD thesis, Massachusetts Institute of Technology, 1981.

[MP92] Henry Massalin and Calton Pu. A lock-free multiprocessor os kernel. *ACM Operating Systems Review*, 26(2):8, 1992.

[MS05] Paul E. McKenney and Dipankar Sarma. Towards hard realtime response from the linux kernel on smp hardware. In *Linux.conf.au*, 2005.

[MTV02] Nick Murphy, Mark Tonkelowitz, and Mike Vernal. The design and implementation of the database file system, 2002.

[Mye96] J. Myers. Post Office Protocol - Version 3. RFC 1939 (Standard), May 1996. Obsoletes by RFC 1725.

[NCF05] Edmund B. Nightingale, Peter M. Chen, and Jason Flinn. Speculative execution in a distributed file system. In *Proceedings of the ACM SIGOPS Symposium on Operating Systems Principles (SOSP)*, pages 191–205, 2005.

[NIS10] NIST. National Vulnerability Database. http://nvd.nist.gov/, 2010.

[NMAT+07] Yang Ni, Vijay S. Menon, Ali-Reza Adl-Tabatabai, Antony L. Hosking, Richard L. Hudson, J. Eliot B. Moss, Bratin Saha, and Tatiana Shpeisman. Open nesting in software transactional memory. In *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 68–78, 2007.

[NMRW02] George C. Necula, Scott Mcpeak, S. P. Rahul, and Westley Weimer. CIL: Intermediate language and tools for analysis and transformation of C programs. In *Proceedings of the International Conference on Compiler Construction (CC)*, pages 213–228, 2002.

[NPCF08] Edmund B. Nightingale, Daniel Peek, Peter M. Chen, and Jason Flinn. Parallelizing security checks on commodity hardware. In *Proceedings of the ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 308–318, 2008.

[NVCF06] Edmund B. Nightingale, Kaushik Veeraraghavan, Peter M. Chen, and Jason Flinn. Rethink the sync. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 1–14, 2006.

[Ols93] Michael A. Olson. The design and implementation of the inversion

file system. In *Proceedings of the USENIX Annual Technical Conference*, 1993.

[PHR+09] Donald E. Porter, Owen S. Hofmann, Christopher J. Rossbach, Alexander Benn, and Emmett Witchel. Operating system transactions. In *Proceedings of the ACM SIGOPS Symposium on Operating Systems Principles (SOSP)*, pages 161–176, 2009.

[Pla] http://plash.beasts.org/wiki/PlashIssues/ConnectRaceCondition.

[PMS09] Erez Petrank, Madanlal Musuvathi, and Bjarne Steesngaard. Progress guarantee for parallel programs via bounded lock-freedom. In *Proceedings of the ACM SIGPLAN conference on Programming language design and implementation (PLDI)*, pages 144–154, 2009.

[Pug90] William Pugh. Skip lists: a probabilistic alternative to balanced trees. *Communications of the ACM (CACM)*, 33:668–676, 1990.

[PW09] Donald E. Porter and Emmett Witchel. Operating systems should provide transactions. In *Proceedings of the USENIX Workshop on Hot Topics in Operating Systems (HotOS)*, 2009.

[PW10] Donald E. Porter and Emmett Witchel. Transactional system calls on linux. In *Linux Symposium*, 2010.

[RG02] Ravi Rajwar and James R. Goodman. Transactional lock-free execution of lock-based programs. *Proceedings of the ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 5–17, 2002.

[RHP+07] Christopher J. Rossbach, Owen S. Hofmann, Donald E. Porter, Hany E. Ramadan, Aditya Bhandari, and Emmett Witchel.

TxLinux: Using and managing transactional memory in an operating system. In *Proceedings of the ACM SIGOPS Symposium on Operating Systems Principles (SOSP)*, pages 87–102, 2007.

[RRHW09] Hany E. Ramadan, Indrajit Roy, Maurice Herlihy, and Emmett Witchel. Committing conflicting transactions in an STM. In *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 163–172, 2009.

[RRP+07] Hany E. Ramadan, Christopher J. Rossbach, Donald E. Porter, Owen S. Hofmann, Aditya Bhandari, and Emmett Witchel. MetaTM/TxLinux: Transactional memory for an operating system. In *Proceedings of the ACM IEEE International Symposium on Computer Architecture (ISCA)*, pages 92–103, 2007.

[RS09] Mark Russinovich and David Solomon. *Windows Internals*. Microsoft Press, 2009.

[RT74] Dennis M. Ritchie and Ken Thompson. The UNIX time-sharing system. *Communications of the ACM (CACM)*, 17(7):365–375, 1974.

[SAF07] Ya-Yunn Su, Mona Attariyan, and Jason Flinn. AutoBash: Improving configuration management with operating system causality analysis. In *Proceedings of the ACM SIGOPS Symposium on Operating Systems Principles (SOSP)*, pages 237–250, 2007.

[SB06] Russell Sears and Eric Brewer. Stasis: Flexible transactional storage. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 29–44, 2006.

[SBV95] Gurindar S. Sohi, Scott E. Breach, and T. N. Vijaykumar. Mul-

tiscalar processors. In *Proceedings of the ACM IEEE International Symposium on Computer Architecture (ISCA)*, pages 414–425, 1995.

[Sch90] Fred B. Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Computing Surveys*, 22(4):299–319, 1990.

[SCZM00] J. Greggory Steffan, Christopher B. Colohan, Antonia Zhai, and Todd C. Mowry. A scalable approach to thread-level speculation. In *Proceedings of the ACM IEEE International Symposium on Computer Architecture (ISCA)*, pages 1–12, 2000.

[SDD+85] Alfred Z. Spector, Dean Daniels, Daniel Duchamp, Jeffrey L. Eppinger, and Randy Pausch. Distributed transactions for reliable systems. In *Proceedings of the ACM SIGOPS Symposium on Operating Systems Principles (SOSP)*, pages 127–146, 1985.

[SDH+96] Adam Sweeney, Doug Doucette, Wei Hu, Curtis Anderson, Mike Nishimoto, and Geoff Peck. Scalability in the XFS file system. In *Proceedings of the USENIX Annual Technical Conference*, pages 1–14, 1996.

[Sel93] Margo I. Seltzer. Transaction support in a log-structured file system. In *Proceedings of the IEEE International Conference on Data Engineering (ICDE)*, pages 503–510, 1993.

[SESS96] Margo Seltzer, Yasuhiro Endo, Christopher Small, and Keith A. Smith. Dealing with disaster: Surviving misbehaved kernel extensions. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 213–227, 1996.

[SGC+09] Richard Spillane, Sachin Gaikwad, Manjunath Chinni, Erez Zadok, and Charles P. Wright. Enabling transactional file access via lightweight kernel extensions. In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST)*, pages 29–42, 2009.

[SLSV05] Weiqing Sun, Zhenkai Liang, R. Sekar, and V. N. Venkatakrishnan. One-way isolation: An effective approach for realizing safe execution environments. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, pages 265–278, 2005.

[Sma98] Christopher Allen Small. *Building an Extensible Operating System*. PhD thesis, Harvard University, 1998.

[SMAT+07] Tatiana Shpeisman, Vijay Menon, Ali-Reza Adl-Tabatabai, Steven Balensiefer, Dan Grossman, Richard L. Hudson, Katherine F. Moore, and Bratin Saha. Enforcing isolation and ordering in STM. *Proceedings of the ACM SIGPLAN conference on Programming language design and implementation (PLDI)*, 2007.

[SMS08] Michael F. Spear, Maged M. Michael, and Michael L. Scott. Inevitability mechanisms for software transactional memory. In *Proceedings of the ACM SIGPLAN Workshop on Transactional Computing (TRANSACT)*, 2008.

[SW91] Frank Schmuck and Jim Wylie. Experience with transactions in QuickSilver. In *Proceedings of the ACM SIGOPS Symposium on Operating Systems Principles (SOSP)*, pages 239–253, 1991.

[THWS08a] Dan Tsafrir, Tomer Hertz, David Wagner, and Dilma Da Silva. Portably preventing file race attacks with user-mode path resolution. Technical report, IBM Research Report, 2008.

[THWS08b] Dan Tsafrir, Tomer Hertz, David Wagner, and Dilma Da Silva. Portably solving file TOCTTOU races with hardness amplification. In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST)*, pages 189–206, 2008.

[Twe] Stephen Tweedie. Ext3, journaling filesystem. http://olstrans.sourceforge.net/release/OLS2000-ext3/OLS2000-ext3.html.

[TY03] Eugene Tsyrklevich and Bennet Yee. Dynamic detection and prevention of race conditions in file accesses. In *USENIX Security Symposium*, pages 243–256, 2003.

[Val95] John D. Valois. Lock-free linked lists using compare-and-swap. In *Proceedings of the ACM symposium on Principles of distributed computing (PODC)*, pages 214–222, 1995.

[VTG+09] Haris Volos, Andres Jaan Tack, Neelam Goyal, Michael M. Swift, and Adam Welc. xCalls: Safe I/O in memory transactions. In *Proceedings of the ACM European Conference on Computer Systems (EuroSys)*, pages 247–260, 2009.

[WCN+09] Benjamin Wester, James Cowling, Edmund B. Nightingale, Peter M. Chen, Jason Flinn, and Barbara Liskov. Tolerating latency in replicated state machines. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 245–260, 2009.

[WLAG93] Robert Wahbe, Steven Lucco, Thomas E. Anderson, and Susan L. Graham. Efficient software-based fault isolation. In *Proceedings of the ACM SIGOPS Symposium on Operating Systems Principles (SOSP)*, pages 203–216, 1993.

[WSSZ07] Charles P. Wright, Richard Spillane, Gopalan Sivathanu, and Erez Zadok. Extending ACID semantics to the file system. *ACM Transactions on Storage*, 3(2):4, 2007.

[WTWPLP85] Matthew J. Weinstein, Jr. Thomas W. Page, Brian K. Livezey, and Gerald J. Popek. Transactions and synchronization in a distributed operating system. In *Proceedings of the ACM SIGOPS Symposium on Operating Systems Principles (SOSP)*, 1985.

[Zal01] Michael Zalewski. Delivering signals for fun and profit, 2001. http://lcamtuf.coredump.cx/signals.txt.

[ZB06] Craig Zilles and Lee Baugh. Extending hardware transactional memory to support non-busy waiting and non-transactional actions. In *Proceedings of the ACM SIGPLAN Workshop on Transactional Computing (TRANSACT)*, Jun 2006.

Vita

Donald Elliott Porter attended Jonesboro High School in Jonesboro, Arkansas. He graduated from Hendrix College in Conway, Arkansas in 2003 with a B.A. in Computer Science and Mathematics. During the following years, he was employed as a software developer at Acxiom in Little Rock, Arkansas. In 2005 he started the doctoral program in the Department of Computer Science at the University of Texas at Austin, and completed an M.S. in Computer Science in 2007.

Permanent Address: porter@cs.stonybrook.edu

This dissertation was typeset with LaTeX 2_ε[1] by the author.

[1] LaTeX 2_ε is an extension of LaTeX. LaTeX is a collection of macros for TeX. TeX is a trademark of the American Mathematical Society. The macros used in formatting this dissertation were written by Dinesh Das, Department of Computer Sciences, The University of Texas at Austin, and extended by Bert Kay, James A. Bednar, and Ayman El-Khashab.