xCalls: Safe I/O in Memory Transactions

Haris Volos, Andres Jaan Tack, Neelam Goyal*, Michael M. Swift, and Adam Welc+
*University of Wisconsin–Madison, *Oracle, +Intel*
{hvolos,tack,neelam,swift}@cs.wisc.edu, adam.welc@intel.com

Abstract

Memory transactions, similar to database transactions, allow a programmer to focus on the logic of their program and let the system ensure that transactions are atomic and isolated. Thus, programs using transactions do not suffer from deadlock. However, when a transaction performs I/O or accesses kernel resources, the atomicity and isolation guarantees from the TM system do not apply to the kernel.

The xCall interface is a new API that provides transactional semantics for system calls. With a combination of deferral and compensation, xCalls enable transactional memory programs to use common OS functionality within transactions.

We implement xCalls for the Intel Software Transactional Memory compiler, and found it straightforward to convert programs to use transactions and xCalls. In tests on a 16-core NUMA machine, we show that xCalls enable concurrent I/O and system calls within transactions. Despite the overhead of implementing transactions in software, transactions with xCalls improved the performance of two applications with poor locking behavior by 16 and 70%.

Categories and Subject Descriptors D.4.1 [*Operating Systems*]: Process Management-Concurrency

General Terms Design, Languages, Performance

Keywords Concurrent programming, Transactional memory, xCalls, System calls, I/O

1. Introduction

As the microprocessor industry transitions to multithreaded and multicore chips, programs must use multiple threads to obtain the full performance of the underlying platform [Sutter 2005]. Transactional memory (TM) [Herlihy 1992] has garnered interest in research and industry as a mechanism to simplify concurrent programming. Transactions allow a programmer to declare a block of code *atomic*, and the TM system ensures that (1) it executes to completion or not at all, and (2) intermediate states of memory are not visible to other transactions. Programmers are given the illusion that transactions execute in a serial order, while the TM system executes them concurrently. As a result, a transaction may abort when it accesses the same data as another transaction, because the two transactions cannot be serialized. This prevents deadlock, which after decades of research remains a problem for many applications [Jula 2008, Wang 2008]. Most major processor and OS vendors have expressed interest in transactional memory [Microsoft Corp. 2008, Saha 2006b, Schlaeger 2008, Tremblay 2008].

While memory within a process is under the TM system's control, memory in the kernel may not be. As a result the atomicity and isolation properties are not automatically enforced for changes to kernel data structures. For example, file data changed by one transaction that subsequently aborts may be read by another transaction. In addition, most I/O operations, such as sending a packet, cannot be reversed on abort. Analyses of multithreaded programs written with locks show that system calls are a regular occurrence in critical sections [Baugh 2007, Swift 2008]. Forbidding system calls in transactions reduces the utility of TM and threatens its validity as a solution to real concurrency problems [Cantrill 2008, Lu 2006].

Prior work on transactions has identified three mechanisms for handling irreversible actions and system calls with side-effects: (1) defer existing system calls and I/O until commit [Baugh 2007, McDonald 2006, Rossbach 2007]; (2) execute existing system calls during the transaction and reverse side effects on abort [Baugh 2007, Moravan 2006], or (3) ensure that transactions with system calls always commit (called *irrevocable transactions*) [Blundell 2007, Olszewski 2007, Spear 2008, Welc 2008]. However, each approach is itself insufficient. When two operations are deferred, the OS may not be able to guarantee that both will succeed, leading to an inconsistent state. When system calls must be reversed on abort, actions to reverse side effects may fail. To guarantee they will commit, irrevocable transactions cannot execute concurrently, limiting performance.

This paper presents a new programming interface for transactional memory programs called *xCalls*. The xCall interface provides transactional access to common OS services, such as file handling, communication, and threading. For example, rather than calling the `write()` system call,

EuroSys'09, April 1–3, 2009, Nuremberg, Germany.

code in a transaction calls `x_write()`. Data written by this call is not visible until the transaction commits.

This interface is guided by two design principles. First, system calls should be executed as early as possible, but no earlier. This ensures that errors from the OS are available early, to allow application recovery, but that irreversible actions are deferred until the transaction commits. Second, xCalls must expose all failures to the application, as do system calls. This bypasses the intractable problem of handling all low-level failures *within* the xCall API and ensures that transactional programs can be as reliable as lock-based ones.

We implement xCalls purely at user-mode implementation to provide portability across systems and to avoid costly kernel modifications. We find that the majority of system calls can be accessed within transactions without support from the operating system. Furthermore, our implementation makes only general demands on the supporting transactional memory system. While implemented for a single software TM system, it could easily be implemented or ported to other software TMs as well as proposed hardware TM systems.

Rather than making every system call transactional, the xCall API handles the common cases of file access, communication, and threading. xCalls provide isolation for kernel resources with *sentinels*, which are revocable user-level locks. A transaction acquires a sentinel when it accesses a kernel resource, such as a file, through an xCall. Competing threads must block until the transaction completes and releases the sentinel. xCalls provide atomicity for system calls through a combination of *deferral*, delaying execution until the transaction commits, and *compensation*, calling back into the kernel to undo the side effects of a previous call. Rather than concealing the execution model, the xCall interface specifies *when* every call executes, so programmers are aware when the side effects of an xCall become visible. Finally, xCalls return errors after the transaction completes to notify programs when a deferred system call or compensation fails.

We implement xCalls for prototype Intel Software Transactional Memory (STM) compiler [Intel 2008] and apply it to three applications: the Berkeley DB embedded database, the BIND DNS server, and the XMMS media player. We find that using transaction in place of locks is straightforward, and that adapting existing code to use xCalls is simple.

In line with recent analysis of STM systems [Cascaval 2008], we find that software TM has non-trivial performance overheads: the Intel STM can slow critical sections by up to 1100%. Thus, we find that TM is best suited to improve (1) the programmability of rarely executed critical sections, where the overhead is small, and (2) the performance of heavily contended critical sections where additional concurrency is possible. Programs with high transaction rates and conflicting critical sections experience performance degradation.

In tests on a 16-core NUMA machine, transactions with xCalls improved performed better than the native transactions provided by the Intel STM. For one workload, performance decreased due to the overhead of the transactional memory system. For another two workloads with heavy lock contention, performance increased by 16 and 70%. With hardware support to remove the overhead of transactions, performance could be even better.

In the next section, we present a primer on transactional memory. We follow with the design of xCalls in Section 3 and the interface in Section 4. We present experimental evaluation of the system in Section 5. We end the paper with related work and conclusions.

2. Transactional Memory Overview

Transactional memory (TM) seeks to simplify multithreaded programming by removing the need for explicit locks. Instead, a programmer can declare a section of code *atomic*, and the TM system will enforce isolation (*i.e.*, no access to uncommitted data) and atomicity (*i.e.*, all or nothing) for the code, and resolve any *conflicts* that occur. Conflicts arise when two concurrent transactions access the same memory items and one transaction performs a write. Transactions can execute concurrently if they do not conflict. Thus, they can improve performance if critical sections rarely conflict. If a conflict occurs, a resolution policy may stall or abort one of the transactions to clear up the conflict. TM systems enforce isolation by detecting when two transactions conflict, and provide atomicity by buffering either old or new values to allow the transaction to abort.

Transactional memory has been implemented in software (an STM) [Dice 2006, Harris 2003, Saha 2006a] and can be implemented in hardware (an HTM) [Hammond 2004, Moore 2006], or with a combination [Baugh 2008, Damron 2006, Minh 2007]. Because a software TM system must perform version management to store both old and new values written and conflict detection on loads and stores, performance may drop by 65% or more [Harris 2006, Saha 2006a]. Proposed hardware transactional memory systems would perform these operations in hardware, so transactions that do not conflict are executed with almost no overhead. However, such hardware is not yet available.

Some TM systems implement *irrevocable* transactions (also called inevitable transactions) that cannot abort. These allow system calls to execute within transactions by removing the need to reverse the system call's effects [Blundell 2007, Olszewski 2007, Spear 2008, Welc 2008]. However, this approach only allows a single irrevocable transaction at a time to prevent conflicts and thus limit concurrency. In addition, irrevocable transactions may not abort themselves, which prevents the use of transactions for error handling [Fetzer 2007] or for conditional blocking [Harris 1991].

The xCall interface depend on three additional features provided by many TM systems:

1. Transactions must be able to *abort themselves* to resolve deadlocks around I/O.
2. Transactions must be able to *escape into non-transactional code* without terminating the transaction. This enables xCalls to invoke system calls and non-transactional library code.
3. Transactions must be able to specify *commit actions* that execute at commit and *compensating actions* that execute on abort.

The Intel STM supports these requirements [Ni 2008], as do many proposed HTM designs [McDonald 2006, Moravan 2006, Moss 2006].

3. xCall Design

The xCall interface enables applications to invoke system calls within transactions. Rather than transparently make existing system calls transactional, xCalls expose to the programmer how atomicity is provided for each function. This informs programmers about which failures may occur during the transaction and which occur on commit or abort.

3.1 Design Overview

The xCall APIs rely on four components to provide transactional semantics for I/O and system calls:

1. *xCall* functions provide transactional semantics for commonly used kernel functionality.
2. *Deferral, compensation, and buffering* provide atomicity for kernel data and I/O.
3. *Sentinels* provide isolation for kernel data.
4. *Error handlers* inform the program asynchronously when deferral or compensation fails.

The xCall interface is similar to the POSIX interface, so porting existing code requires mostly syntactic changes, such as changing the name of a function and passing an additional result parameter. We defer discussion of the xCall API to Section 4.

3.2 Atomic Execution with Deferral, Compensation and Buffering

Work in transaction processing has found two fundamental techniques for atomic state changes: (1) *compensation:* buffer old values, execute the change during the transaction, and revert or undo on abort, or (2) *deferral:* buffer new data and defer the change until commit. The xCall implementation provides atomicity with both deferral and compensation, depending on the semantics of the call.

The xCall APIs classify system calls into six categories based on their behavior. Table 1 lists the categories and the actions they perform during transaction execution, commit, and abort. An xCall executes a system call in place when (1) it has no side effects; (2) it can be reversed; or (3) its output is required for the program to proceed. This ensures that errors are visible to the program as early as possible. An xCall defers a system call until commit when its side effects cannot be reversed at user level, such as sending a packet.

Consumer and reader system calls differ by whether they destroy data in the kernel. For example, reading data from a pipe deletes the kernel's copy of that data. Thus, the xCall must buffer the data to make it available to other threads if the transaction aborts. Similarly, xCalls make a distinction between writer system calls, which have reversible side effects, and producers, which do not. Producer calls must be deferred until commit. Finally, *renegade* system calls are ones that are not reversible and cannot be deferred. These calls produce side effects and either have ambiguous/variable semantics or require two-way communication with a non-transactional device or service. Renegade calls can only be supported by irrevocable transactions, because their effects cannot be either deferred until commit or reversed on abort by library code.

Our study of the Linux system calls, indicates as shown in Table 2, that most system calls can be handled through compensation or deferral. Of 284 Linux system calls, 84 do not modify visible kernel state and can be executed without xCalls. Of the remainder, 163 can be made atomic with deferral and compensation. Only 33 require irrevocable transactions, and none of the programs we experiment with make these calls within transactions.

Strategy	Count	Example
Read-only	88	`getpid, mincore, pread`
Deferral	66	`exit, settimeofday, umount`
Compensation	97	`chdir, flock, symlink`
Irrevocable	33	`fcntl, ioctl, poll, select`

Table 2. Categorization of Linux system calls into those made atomic through deferral, compensation, and those that require irrevocable transactions.

3.3 Sentinels for Isolation

Most transactional memory systems operate only at user level and do not isolate changes to kernel data made within a transaction. For example, a thread may view speculative data written to a file by a transaction that later aborts. Such transient effects do not occur with locks and may cause the program to behave incorrectly.

The xCall interface provides isolation for kernel objects with *sentinels*. A sentinel is a lightweight, revocable user-level lock for a kernel object. The purpose of a sentinel is to isolate the effects of system calls from other threads in the same process. Similar to a database lock, sentinels are centrally managed to detect deadlocks and can be revoked to recover from deadlocks.

Category	Examples	Execution	Commit	Abort
Side-effect free	`fstat()`	Execute	–	–
Readers	`read()` file	Execute	–	Reset kernel pointers
Consumers	`recv()` socket	Execute, buffer results	–	Share buffer
Writers	`write()` file	Execute, buffer old contents	–	Rewrite from buffer
Producers	`send()` socket	Buffer	Execute	–
Renegades	`ioctl()`	Execute, make transaction irrevocable	–	–

Table 1. Categories of system calls and the xCall operations at execution, on commit, and on abort.

The xCall APIs associate sentinels with distinct kernel objects, such as sockets and file descriptors. Sentinels are released only when the transaction commits or aborts to implement two-phase locking. However, sentinels do not protect programs with inherent race conditions, such as when a non-transactional thread reads data from a file written by a transaction.

Before invoking a system call, an xCall acquires the sentinel that isolates the underlying kernel object accessed by the call. Sentinels only lock the logical state of the kernel that is visible through system calls. Internally, kernel queues and buffers may have different contents as long as system calls do not observe a change.

The sentinel implementation must detect deadlock when two transactions acquire the same set of sentinels concurrently in different orders. On deadlock, one transaction aborts and releases its sentinels.

3.4 Error Handlers

Robust programs must check for the failure of a system call and clean up their state when an error occurs. In addition, they may handle errors that arise while cleaning up. Database transactions provide *failure atomicity*, so that all operations either succeed or the transaction aborts. Memory transactions, though, cannot provide this automatic error handling for system calls, so programs must still include this code. However, xCalls change *when* this code executes; some error results are not available until the transaction commits or aborts.

For xCalls that execute system calls in-place within transactions, errors may be handled in place as well. For xCalls that defer system calls until commit, the programmer must move error-handling code to execute after commit. Because a transaction may execute multiple xCalls, it is possible that some deferred operations succeed while others fail.

The xCall API relies on *result parameters* to return errors asynchronously. As illustrated below, every xCall takes an extra parameter that returns errors after the transaction commits or aborts.

```
ssize_t x_write_pipe(int fd, void *buf,
                     size_t nbytes, int * result);
```

The result value is set on failure. If a single variable is passed to a set of calls, it will be set if *any* deferred xCall operation fails, allowing a program to check for failure with a single test.

The xCall interface also reports when a compensating action fails to compensate. A compensating action could fail for either transient reasons, such as low memory, or for persistent reasons, if the state of the system changed between the xCall and its compensating action. For example, an xCall may be unable to rewrite data to a network file server during abort if the network is partitioned. To handle these failures, xCalls allow a program to request that (1) the transaction not be retried when a compensating action fails and (2) a single error result be set. The result parameters from individual xCalls indicate which compensating actions failed. Thus, programs can test with a transaction failed during commit or abort by testing this result.

4. xCall Functions

We implemented xCalls for file handling, communication, and threading system calls and rely on the Intel STM's internal support for memory management. All the system calls invoked within critical sections, both in our workloads and other applications we investigated, fall in these categories [Swift 2008].

4.1 File Handling

The xCall file APIs provide support for common file operations within a transaction. We rely on the TM system for transactional access to memory-mapped files and do not enforce transaction semantics if a file is simultaneously accessed through memory and a system call. We defer discussion of pipes to the following section.

As shown in Table 3, most file operations are handled in place, which is possible because many file modifications can be reversed from user mode. Operations without well-defined semantics, such as `fcntl()`, are not supported by xCalls and must execute in an irrevocable transaction.

There are two xCalls for writing data to the file, allowing the xCall implementation to optimize its atomicity mechanism. The `x_write_seq()` call appends to the file and truncates the file on abort. The `x_write_ovr()` call supports random access. It reads the old content being overwritten into a buffer before writing new data. On abort, it restores the original data. While `x_write_ovr()` is correct for sequential access, it adds unnecessary overhead when appending.

xCall	System Call	Execution	Compensation	Failures
x_open	open	in-place	close	abort
x_create	open	in-place	unlink, rename	abort, commit
x_dup	dup	in-place	close	abort
x_rename	rename	in-place	rename	abort
x_seek	lseek	in-place	lseek	abort
x_read	read	in-place	lseek	abort
x_write_ovr	write	in-place	lseek, write	abort
x_write_seq	write	in-place	truncate	abort
x_unlink	unlink	deferred	–	commit
x_close	close	deferred	–	commit
x_fsync	fsync	deferred	–	commit
x_fstat	fstat	in-place	–	–

Table 3. The xCalls for accessing files. For each xCall, this table shows the underlying system call, when the operation executes, the compensation, and which asynchronous actions can fail.

As illustrated in Figure 1, the x_read() call is also executed in place. Its compensation is to reset the file pointer by seeking backward. To achieve better performance, x_read() places restrictions on the buffer that receives data. The buffer must be (1) thread private and (2) empty at the start of the transaction, meaning that it does not contain useful data. A buffer may be used for multiple x_read() calls, though. With these requirements, the xCall can pass the buffer to the kernel without first saving its original contents. In addition, the TM system need not detect conflicting accesses to this buffer. Programs requiring transactional semantics can memset() the buffer in advance of use.

The read and write xCalls acquire sentinels on file descriptors to prevent other threads from viewing uncommitted changes to a file. We do this for efficiency, as locking the file itself requires an additional system call to find the file's inode number. The library detects when descriptors are duplicated with x_dup(), but will allow concurrent access if the same file is opened twice.

Providing atomicity for directory operations requires special care. When creating a file, x_create() tests whether the file name is in use. If so, it renames the old file before creating the new file. When the transaction commits, the old file is removed. This preserves the contents of the file if the transaction aborts. Similar to x_create(), x_rename first tests if the new file name is in use, and renames it. Then, it creates a new hard link with the new name, which points to the file to be renamed. When the transaction commits, the old file is removed. x_unlink() function is deferred until commit, as its actions cannot easily be reversed.

Race conditions may arise between transactions that perform directory operations. For example, because create operations are executed in place, a transaction could open the newly created file, only to have it disappear if the creating transaction aborts. The x_open(), x_create(), x_unlink() and x_rename() calls prevent this race by acquiring a sentinel indexed by the file inode number. These xCalls invoke the sentinel manager to acquire a lock on a table of open files before opening, creating, deleting or renaming a file. The lock is released after the xCall discovers the file's inode number and acquires the sentinel. Furthermore, both x_create() and x_rename() overwrite existing files, they acquire an extra sentinel indexed by the inode number of the overwritten file.

```
ssize_t x_read(int fd, void *buf,
               size_t nbytes, int *result) {
  ssize_t bytes;
  get_sentinel(fd);
  bytes = read(fd, buf, nbytes);
  if (bytes != -1)
    compensate(x_undo_read, fd, bytes, result);
  return(bytes)
}

int x_undo_read(int fd, ssize_t nbytes, int *result) {
  off_t ret;
  ret = lseek(fd, -nbytes, SEEK_CUR);
  if (ret == -1)
    *err = errno;
  return(ret == -1);
}
```

Figure 1. The xCall code for reading from a file and its compensating action for abort. Code in the xCall for isolation and atomicity is marked with boldface. The compensating action x_undo_read() returns a flag indicating whether the compensation failed.

4.2 Communication

The xCall API supports network communication with sockets and intra- and inter-process communication with pipes. To distinguish between pipe operation and file operations, there are separate xCalls, x_read_pipe() and x_write_pipe(), for pipe access. These calls are also suitable for other forms of streaming communication, such as standard input and output. We have also written xCall functions for standard socket calls, such as x_send() and x_recv(). As

the xCalls for pipe and socket communication behave similarly, we describe their implementations together. As with the file xCalls, communication xCalls acquire sentinels on file descriptors or sockets for send/receive operations.

Sockets and pipes support two fundamental operations, send and receive, which may be accessed through file system APIs or socket APIs. The send xCalls buffer messages at user level until the transaction commits, when messages are delivered to the kernel. Deferring sends ensures that the message recipients only receive committed send operations, but prevents two-way communication.

The receive xCalls execute in place, but store results in a shared buffer. On abort, the data cannot be pushed back into the kernel, so it remains within shared buffers until eventually consumed or the program calls `x_close()` on the socket or pipe. All receives first check whether buffered data exist before requesting new data from the kernel. Thus, receiving data requires two copy operations: one copy from the kernel into a shared buffer, and one copy from the shared buffer into the application buffer.

4.3 Threading

The xCall interface supports thread creation, locking, and condition variables. xCalls defer thread creation until commit, so memory changed by the new thread does not need to be rolled back on abort. As a result, transactions cannot interact with threads they create. No sentinels are used, because the thread is not visible until commit.

Locking and condition variables require special APIs because acquiring a lock within a transaction may lead to deadlock, livelock, or loss of mutual exclusion [Rossbach 2007, Volos 2008]. We implement transaction-safe locks (TxLocks) [Volos 2008] to allow transactions to acquire mutex locks. Unlike other xCalls, the locking functions replace existing Pthread locking functions to allow legacy code using locks to interact with transactional code. TxLocks provide atomicity with compensating and commit actions to release locks: locks acquired within a transaction are released with a compensating action on abort, and locks released within a transaction are held until the transaction commits.

In addition, xCalls provide a transactional version of condition variables [Dudnik 2009]. Like regular condition variables, transactional condition variables commit the transaction before waiting, and start a new transaction after resuming. Unlike regular condition variables, the signal and broadcast functions wake up waiters directly, rather then enqueuing them on the lock.

4.4 xCall Implementation

We implemented the xCall API using the Intel Prototype STM compiler version 2.0 for Linux [Intel 2008]. The xCall APIs execute as escape actions, outside of transaction, using the `__tm_waiver` annotation. We rely on the Intel STM to provide transactional memory management. The STM defers freeing memory until commit and compensates for memory allocation by freeing allocated buffers on abort [Wang 2007, Zilles 2006].

The total implementation is 7990 lines of code, comprising the implementation of 27 xCalls for common system calls and three subsystems: buffer management, sentinel management, and failure management.

Sentinel Manager. This subsystem allocates and maps sentinels to logical kernel objects in user mode. We implement process-wide sentinels with POSIX reader/writer locks to allow multiple readers to access a kernel object simultaneously. The sentinel manager maintains lists of the sentinels acquired by each transaction and releases them at commit or abort.

To prevent deadlocks, the sentinel manager enforces a canonical global order over all sentinels based on the sentinel's index in the global table of allocated sentinels. When a transaction cannot acquire a sentinel, the sentinel manager aborts the transaction and releases the acquired sentinels. The transaction re-acquires all the sentinels it encountered in canonical order before restarting. The transaction holds on to the sentinels until commit, even if it does not require the same sentinels when re-executed.

Buffer Manager. This subsystem provides shared and private buffers to store data for deferred system calls and for inplace system calls that require compensation. Shared buffers store data when the kernel cannot undo a data-producing action, such as reading from a pipe. These buffers persist across transactions and may be accessed by any thread. Private buffers store data for deferred calls for compensating actions. For example, undoing a random file write replaces the data in the file with its original contents, which are stored in a private buffer. As private buffers are discarded when a transaction aborts, they are organized as a per-thread log to reduce fragmentation and bookkeeping overheads.

Failure manager. This subsystem informs programs when a compensating or deferred action fails. It exposes an interface to applications to specify an action to take or a variable to set when an asynchronous failure (one during commit or abort) occurs. Figure 2 illustrates the use of this interface. For example, the program may specify that the transaction should not be retried if a compensating action fails by calling the manager with the `X_NO_RETRY` flag.

4.5 Semantics

The Intel STM provides single lock atomicity (SLA) [Menon 2008]. Under SLA, a program behaves as if a single global lock guards each atomic block. Thus, programs that are race free under a single global lock will execute correctly under transactional execution. xCalls maintain the single lock atomicity (SLA) semantics.

The SLA semantics places two requirements on transactional programs and the xCall implementation. First, the STM requires that program be race free using only transac-

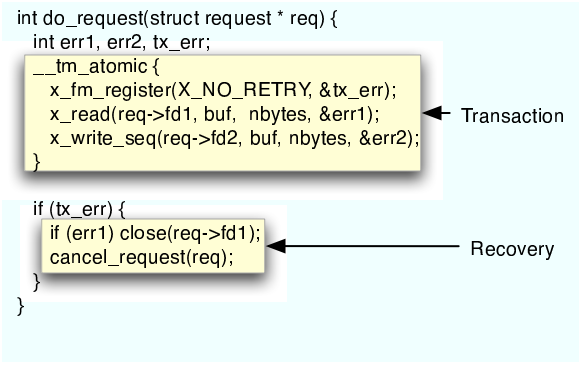

Figure 2. Example code of error handling. The `X_NO_RETRY` indicates that the transaction should not be retried if an asynchronous failure occurs during abort, and the result parameters allow the program to determine exactly what failed.

tions for concurrency control [1]. Consequently, accesses to a system resource (e.g., file, network socket, etc.) from both outside and inside a transaction are considered racy and behavior of a program executing such accesses is undefined (much like behavior of a racy program in Intel's STM). As long as xCalls are strictly used inside transactions, accesses to system resources through xCalls are not racy, and therefore they do not compromise SLA.

Second, SLA requires that schedules be serializable. The Intel STM guarantees serializability of atomic blocks with respect to memory accesses through its internal implementation of concurrency control. xCalls maintain serializability of transactions with two-phase locking of sentinels. Thus, xCalls maintain the SLA semantics of the underlying STM.

These semantics apply only within a single process. For example, while file writes within a transaction are not visible to other threads in the same process until that transaction commits, they are immediately visible to threads running in other processes. This is a result of our choice of user-mode sentinels for isolation.

4.6 Kernel support for xCalls

The current xCall implementation is entirely user-mode code. This implementation therefore demonstrates that most system calls do not require support from the kernel at all. However, we have identified three situations where kernel support could simplify the implementation and use of xCalls. First, the xCalls that access directories hold a user-mode lock while calling into the kernel and must call into the kernel a second time to obtain inode numbers used to identify the sentinel required. A single kernel API that opens a file and acquires a kernel lock on the file would improve performance of directory operations.

[1] There is a growing consensus that it is difficult to guarantee SLA for programs whose race-freedom is guaranteed by traditional synchronization mechanisms, such as mutual exclusion locks - Intel's STM does not provide such guarantee.

Second, a modified kernel could guarantee that some deferred calls or compensating actions cannot fail. This would relieve the programmer of writing error-handling code for these cases. However, we believe that it is a bad practice for the kernel to promise that operations succeed, as it precludes future kernel modifications that introduce new failure modes. Otherwise, kernel support could reduce the frequency of failed compensating and commit actions by reserving resources in advance. Nevertheless, since errors can still occur, error handling within programs is still required for reliability.

Third, kernel-mode sentinels could even provide interprocess isolation guarantees of kernel resources between transactional programs that explicitly use the xCall API and non-transactional programs that use the existing system call interface.

4.7 Summary

The xCall interface provides two concrete benefits to transactional memory programmers. First, it exposes transactional semantics to programmers. Unlike proposals to execute existing system calls transactionally [Baugh 2007], programmers are aware of how the call is made transactional. For example, xCalls do not speculate that deferred calls will succeed; instead they return an error code at commit, after the underlying system call executes. Second, xCalls enable performance optimizations. While system calls make no assumptions about their arguments, an xCall can require that a buffer be empty and need not be restored to its original contents on abort.

5. Evaluation

The goal of xCalls is to enable concurrent system calls and I/O within memory transactions. In this section, we evaluate two aspects of xCalls:

1. *Ease of use.* Is it straightforward to use xCalls and transactions instead of system calls and locks?
2. *Performance.* What is the performance cost of the xCall mechanisms as compared to using locks or irrevocable transactions?

As most transactional memory systems do not support system calls, existing TM workloads have no system calls in their transactions [Minh 2007, Yen 2007]. We therefore converted three large multithreaded programs to use transactions for some of their critical sections. While new programs written from scratch to use transactions could reflect a different programming style, they would not accurately capture the size and complexity of large existing applications. In addition, we wrote microbenchmarks to measure the performance of xCalls for comparison against native system calls and the Intel STM's irrevocable transactions.

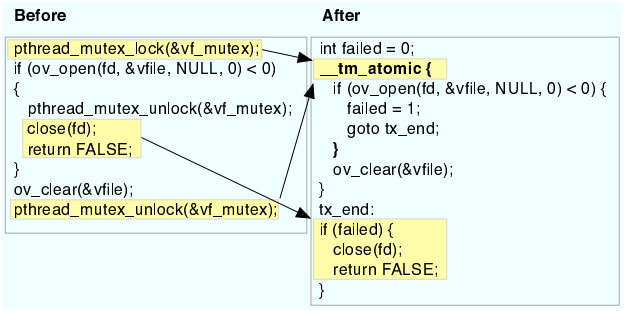

Figure 3. Converted critical section code from XMMS. Arrows indicate corresponding code blocks.

5.1 Ease of use

We converted three large programs to use transactions, as listed in Table 4: Berkeley DB, BIND, and XMMS. We use these programs both for performance testing and to gauge the experience of programming with transactions, to learn whether xCalls burden programmers.

Converting these applications to use memory transactions was a straightforward process of converting all uses of a lock variable into blocks of code denoted with `__tm_atomic`. In addition, we annotated all functions called from transactions with `tm_callable` (required by the STM to produce transactional versions of the code).

A common problem across all three workloads was critical section code with multiple exits. For example, the before code in Figure 3 drops the lock in the middle of a critical section to handle an error. With transactions, shown in the after code, the error handling code must be moved out of the atomic block. This problem arises because the Intel STM enforces a block structure on transactions, and does not arise in TM systems that provide explicit begin and commit transaction statements. A similar problem occurs when a lock is released and re-acquired in the body of a loop. We restructure these loops into a prologue that executes once and a loop containing the termination condition.

In many cases, we found that error handling with transactions was simpler than with locks, because every transaction with xCalls must test for failure at the end of a transaction. Thus, repetitive cleanup code that ordinarily follows each successive invocation of a system call can be consolidated in one place.

Berkeley DB is an embedded database that provides storage, locking, and transactions [Oracle Corporation]. We have converted 20 critical sections in the locking subsystem and 11 critical sections in the logging subsystem to use memory transactions. This subsystem uses xCalls to overwrite a file. The logging routines in Berkeley DB conscientiously check errors from all system calls in the style shown in Figure 3. We moved this code to the end of transactions to recover from failed compensating and commit actions.

The STM provides helpful statistics to indicate how often every transaction aborts, and these helped us quickly identify hotspots in our code. Unlike lock profiling, which identifies which lock is contended but not which critical section is causing the contention, the Intel STM identifies the problematic code. As others have found [Damron 2006], we discovered that direct conversion of the locking subsystem's critical sections into memory transactions resulted in excessive contention on list heads, leading transactions to conflict and abort. We resolved these hotspots by maintaining multiple lists indexed by CPU ID. We also found excessive conflicts in the database deadlock detector, and explicitly serialized this transaction by making it irrevocable.

BIND is a commonly used DNS server [Internet Systems Consortium]. Past versions of BIND had severe scalability problems [Jinmei 2006]. We selected a non-scalable version (9.3.5) and sought to improve its scalability with transactions. We converted the logging and memory subsystems to transactions. In both cases, we converted `write()` and `stat()` system calls to xCalls. We configured the memory subsystem to use BIND's internal memory allocator, which uses `malloc()` occasionally to allocate large blocks of memory and thereafter manages that memory. The logging subsystem records BIND's activity and uses of xCalls to append entries to the log file. BIND handles the failure of a compensating action by ignoring the failure, which is similar to the original source code's practice of ignoring I/O failures during logging.

XMMS is a media player that uses multiple threads to process UI events and sound decoding concurrently [xmms.org]. We converted the playlist interface module and the Ogg Vorbis codec [xiph.com] to use transactions. The playlist module performs memory allocation with transactions but no I/O. The codec uses threads to concurrently read audio data from a file and decode it. We transactionalized critical sections containing file I/O, including calls to `open()`, `read()`, and `lseek()`. We modified XMMS to handle the failure of a compensating action by first, retrying the action to ensure it is not transient, and then terminating playback of the current song.

Summary. Our experience converting these three programs to transactions and xCalls is encouraging, as we found that xCalls could often replace existing system calls with few other changes. With fewer locks, future modifications to these programs may be less likely to cause deadlock. Handling the asynchronous failure of an xCall proved simple in these programs, as they already had mechanisms for recovering from system call failures.

5.2 Performance

We measure the performance impact of using xCalls with transactionalized applications and microbenchmarks. We measure all programs except XMMS on a NUMA machine with 4 quad-core 2 GHz AMD Barcelona processors, 16 GB

Name	Description	Code Size	Transactions	xCalls
Berkeley DB 4.4.20	Database	77,591 lines	31	File read/write
BIND 9.3.5	DNS server	223,755 lines	87	File open, close, write, stat, fsync
XMMS 1.2.11	Media player	59,357 lines	22	File open, read, seek

Table 4. The programs used to evaluate xCalls. The table shows the program name, purpose, size of code base, number of transactions in the code, and the system call operations used in transactions.

RAM, a SATA hard disk, and a gigabit network running Fedora Core 9 in 64-bit mode. We measure the performance of XMMS on a 1.8 GHz Intel Core2-Duo with 2 GB RAM. We replace the C compiler in our workload makefiles with the Intel STM Compiler.

We perform tests in three configuration: *native* uses unmodified applications, locking for synchronization and system calls for I/O; *STM* uses transactions and relies on the Intel STM's irrevocable transactions to execute system calls; *xCalls* uses transactions with xCalls for I/O.

5.2.1 Microbenchmark Performance

We use microbenchmarks to measure the performance overhead of using a software TM system and the potential benefit of xCalls.

The *memtest* program measures the overhead of software transactions by reading or writing 100 integers in an array. We compare the performance of the STM with the performance with locks, both on a single thread. The results below show that the STM causes a 2-5x slowdown for reads and a 5-11x slowdown for writes.

	Read	Write
Stack	2.3x	5.3x
Heap	8.3x	11.6x

In all cases the STM adds substantial overhead to store old values for atomicity. Stack access is cheaper, though, because the compiler is able to determine statically that data has not been shared, whereas heap access requires more expensive runtime conflict detection. These overheads would not be present with proposed hardware supported TM systems [Hammond 2004, Moore 2006]. These results indicate that an STM must substantially improve scalability to overcome the cost of isolation and atomicity.

The *iotest* microbenchmark measures the performance and scalability of system calls. The test issues sets of four identical operations, either in a transaction (for the STM and xCalls configurations) or with no locks held (for the native configuration). Read and write tests access 64KB of data in 16KB chunks from a single file. All data fits within the buffer cache, so the disk is not accessed. Tests of the `open()` call open four files in a transaction and close them after the transaction commits. We separately test `x_write_ovr`, `x_write_seq`, `x_write_pipe`, `x_read` and `x_read_pipe` against a file to measure the overhead of the different atomicity mechanisms.

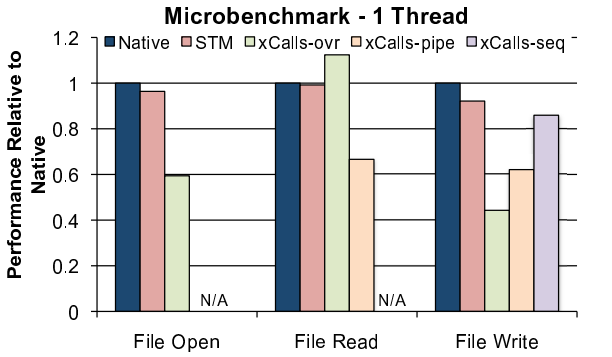

Figure 4. Relative performance of the Intel STM and xCalls compared to native system calls with a single thread.

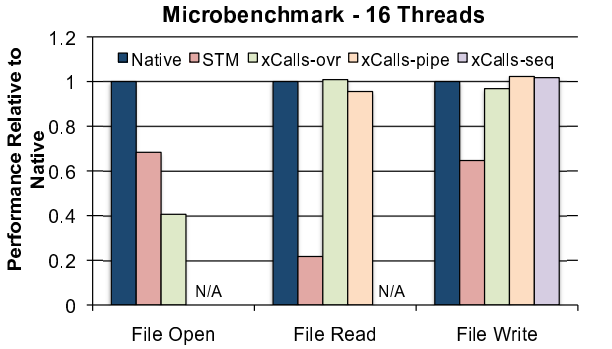

Figure 5. Relative performance of the Intel STM and xCalls compared to native system calls with 16 concurrent threads.

Figure 4 shows the throughput of the STM and xCalls relative to locks for a single thread, and Figure 5 shows results for 16 threads. With a single thread, the Intel STM performs similarly to locks, because irrevocable transactions avoid the overhead of conflict detection and atomicity; instead, the transaction acquires a global lock. xCalls are slower because they buffer data and acquire sentinels. Overwriting a file (labeled xCall-ovr) is the most expensive because it must read in existing data before writing new data. Pipe-style writing is cheaper, because old data is not read, but writes must still be buffered until commit. Sequential writing is cheapest because atomicity is provided by truncating and little extra work is required. Similarly, regular file reads are cheaper than pipe reads, which must buffer data until commit.

With 16 threads, the relative performance of the STM drops because it achieves no concurrency. The concurrency benefit of read and write xCalls raises their performance to a

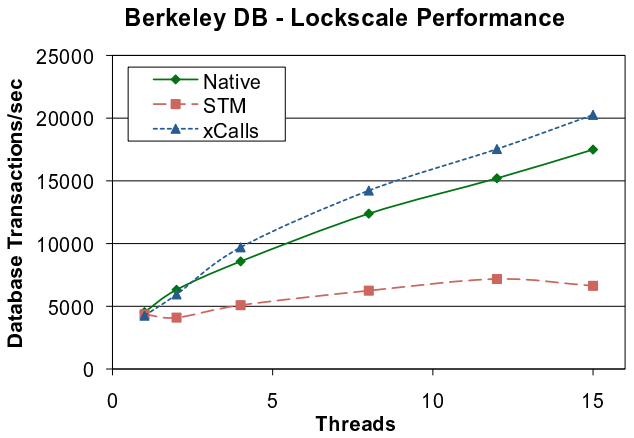

Figure 6. Scalability of Berkeley DB with the Lockscale workload.

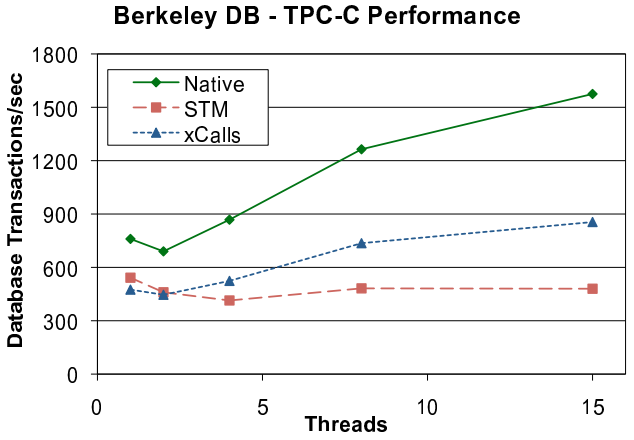

Figure 7. Scalability of Berkeley DB with the TPC-C workload.

point where other system bottlenecks dominate the overhead of buffering and sentinels. However, the open xCall always performs worse than the STM. As noted in Section 4, this function acquires a global lock to prevent races with calls to create a file. As a result, it achieves no scalability with more processors. To verify this cause, we removed the lock from the open xCall and found that its performance rose to near native speed.

These tests demonstrate that with highly concurrent workloads, xCalls can provide scalable I/O performance within transactions despite the added buffering and sentinel costs.

5.2.2 Application Performance

We measure the performance of the test applications with common workloads. The workloads and test results are summarized in Table 5.

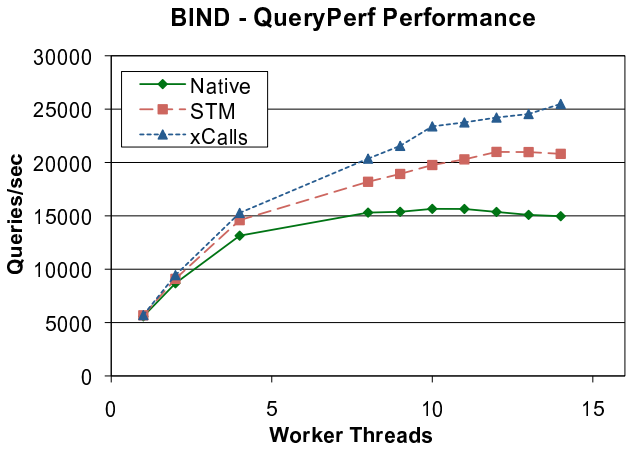

Figure 8. Scalability of BIND.

Berkeley DB. We use two workloads to evaluate Berkeley DB. The *Lockscale* driver, derived from code distributed with the database source, stresses the transaction, logging, and locking subsystems. The driver spawns threads that: begin a database transaction, acquire a write lock for an object picked randomly from a set of 1000 objects, writes a log record, and commit the database transaction.

Second, we use a single-process implementation of TPC-C [Fedorova 2007]. While a full-blown TPC-C requires a multi-tier client server set-up, this version simulates only the load on the server resulting from executing database transactions. The benchmark spawns several threads that perform transactions over a previously generated database. We configured the benchmark to run with 8 warehouses and a 2.5GB buffer cache to prevent serialization around database locks and disk I/O due to buffer page replacements. With a smaller buffer cache the benchmark becomes disk I/O bound, and the performance across all three versions is identical. Despite this configuration, the workload still performs disk I/O for logging. The results for both workloads using all cores are shown in Table 5.

For the Lockscale workload, xCalls performs 16% better than native locking code, while the STM performs 62% worse. Figure 6 shows performance results for different numbers of worker threads. This figure demonstrates that transactions can improve the scalability of lock-based workloads, even with expensive software transactions. While irrevocable transactions limited scalability by serializing around I/O calls, xCalls improved concurrency.

For TPC-C, as shown in Figure 7, the transactional version of Berkeley DB using xCalls achieves 54% of native performance, while the STM version achieves only 30%. However, in testing with fewer threads we find that performance with xCalls scales with the number of threads, while the STM version does not. This illustrates that executing system calls concurrently is critical to the scalability of this application.

Application Name	Workload	# of Threads	xCalls % of locks	Intel STM % of locks	Transaction frequency
Berkeley DB	Lockscale	15	116 %	38 %	58,260 /sec
	TPC-C	15	54 %	30 %	23,661 /sec
BIND	QueryPerf	14	170 %	139 %	228,940 /sec
XMMS	Play .ogg	2	96 %	99 %	182 /sec

Table 5. Performance results for transactional workloads using the Intel STM's irrevocable transactions and xCalls, as a percentage of the performance of lock-based code.

To understand the low performance on TPC-C, we profiled the xCall and native versions of the code to identify where performance is lost. We found that for this workload, Berkeley DB spends an average of 33 ms per database transaction in the STM runtime, while the lock-based version spends a total of 23 ms per database transaction in *all* user-mode code. Transactionalized code invokes the transaction runtime to perform conflict detection and to store old values for abort. In addition, the transactional functions (those called from within transactions) execute 2-3x slower than the non-transactional version, due to the added calls into the runtime. These costs are due to executing transactions in software, and would be greatly diminished with a hardware TM system, or even hardware support for a software TM [Baugh 2008, Minh 2007, Saha 2006b].

BIND. We use the QueryPerf tool to measure the performance of BIND. We loaded an imaginary local zone for BIND with 2,900 domain names scraped from Internet web sites paired with imaginary IP addresses. Against this database, we ran a fixed set of 100,000 queries, repeated as long as each experiment lasted (30 seconds). All three configurations were run at the first debug level (`-d 1`). At this debug level, we configure BIND to print one line of logging output per query.

Figure 8 shows the scalability of the two transactional versions of BIND and the original lock-based version. While the performance of the native version flattens out after 6 threads, with xCalls performance continues to scale up to 14 threads, achieving a 70% performance improvement. The Intel STM also improves performance by up to 39%, but not as much because of serializing around I/O.

XMMS. For XMMS, we play an ogg file and measure the CPU idle time (CPU utilization is so low that it varies by a factor of 5 on short time scales). This application uses only two threads and has a light workload, and neither the STM nor xCalls have a noticeable impact on performance. These results demonstrate xCalls are unnecessary because there is little concurrent I/O. However, xCalls retain a benefit even here, because they allow program-initiated aborts of transactions with I/O, which irrevocable transactions do not.

5.3 Performance Summary

While transactions are primarily intended to simplify multi-threaded programs, the Berkeley DB and BIND results show that supporting concurrent system calls in transactions can improve scalability. In contrast, irrevocable transactions prevent performance from scaling as cores are added. In addition, transactional memory, even a software implementation, can improve the performance of applications with excess synchronization.

6. Related Work

Our work is motivated by studies indicating the prevalence of system calls in the critical sections of lock-based code [Baugh 2007, Blundell 2007, Swift 2008]. The problem of performing non-transactional operations within transactions has long been addressed within the transaction processing and fault-tolerant systems communities. More recently, transactional memory research has proposed mechanisms for addressing the problem.

Transaction processing. Transaction processing systems commonly defer "real actions" that are not under transaction control until the transaction commits [Gray 1993]. Similar to xCalls, failure of a real action must be reported to the application. Compensating actions are common in databases to revert changes made by long-running transactions [Gray 1993]. However, the failure of a compensating action is often handled by (1) retrying the action, or (2) notifying an administrator [Strandenæs 2002]. Both approaches would lead to unreliable software when applied to system calls.

Fault tolerant systems. QuickSilver built transaction support into all system facilities [Haskin 1987]. This support would simplify file I/O and communication in memory transactions. However, QuickSilver made no provision for irreversible I/O operations. Vistagrams provide kernel support for transactional message passing [Lowell 1998] by deferring message send and receive until commit. However, Vistagrams buffering is per-process, not per thread.

Speculator provides isolation for speculative execution of system calls [Nightingale 2005]. Unlike xCalls, which use sentinels to lock kernel objects, Speculator creates a copy of kernel data structures and updates the copy. If the speculation is correct, the copies are made permanent. While speculation could be used to implement transactional I/O and system calls, the implementation requires substantial kernel modifications and supports only single-threaded processes.

Distributed simulation systems have relied on optimistic concurrency of atomic sections, similar to transactions, to

improve performance [Fujimoto 1989, Jefferson 1985]. Like transactional memory, they rely on conflict detection and rollback to correctly order parallel execution. These systems rely on an irrevocability mechanism to execute actions with irreversible side effects non-speculatively.

Transactional memory. TM systems either prohibit system calls in transactions or support them through deferral, compensation, or irrevocable transactions. Baugh *et al.* show that all three mechanisms are required to support every system call made in the critical sections of two programs [2007].

Several systems commit transactions before making a system call, similar to waiting on a condition variable [Smaragdakis 2007, Birrell 2007]. This approach avoids the problem by forcing programmers to remove I/O from critical sections. However, studies of existing multithreaded programs show that I/O and system calls occur frequently within critical sections [Baugh 2007, Lu 2006].

Many TM systems use a combination of deferral and compensation to execute system calls within transactions [Harris 2004, McDonald 2006, Moravan 2006, Zilles 2006]. These systems conceal the failure of deferred system calls or compensating actions from applications, leading to unreliable programs. TxLinux executes the state modifications made by system calls in place, but defers I/O until the transaction commits. This decreases the likelihood of failure, but does not support I/O that returns data or that can fail. Nested LogTM [Moravan 2006] proposed but did not implement a sentinel mechanism for isolation.

Irrevocable transactions provide a simple mechanism for executing system calls in transactions [Blundell 2007, Hammond 2004, Olszewski 2007, Spear 2008, Welc 2008]. Unlike xCalls, they support system calls that are neither deferrable nor reversible. However, they do not provide concurrency between system calls, which may become more important as systems grow to more processors. Irrevocable transactions also prevent program-initiated aborts, which could assist in error handling [Fetzer 2007].

7. Conclusions

Transactional memory must support access to system resources to become a viable method of concurrent programming. Support for these resources will expose the benefits of transactional memory, such as freedom from deadlock and concurrent execution of non-conflicting critical sections, to a larger set of programs. xCalls are a practical approach to executing system calls in transactions that defer calls that can be delayed and compensate for calls that can be reversed. xCalls specify exactly when a system call will take place, so programmers can understand how their code will execute, and provide error-handling mechanisms for when deferral or compensation fail. Thus, the design of xCalls ensures that application reliability is not compromised by the use of transactions.

All is not roses and lilies in the world of xCalls. Our experiments suggest there is a significant and inherent cost to software transactional memory, which is employed by this implementation. Furthermore, there remain system calls whose semantics preclude the use of xCalls.

In general, though, we find with xCalls that much can be done to make transactional memory available to programmers *in commodity operating systems* with few modifications to the systems themselves. Future work includes additional error-handling techniques, such as exception handling, and the prudent application of kernel support where the rewards are large. In addition, a performance study of xCalls on a proposed hardware TM system would clarify the performance implications of transaction-enabled system calls.

Acknowledgments

This work is supported in part by the National Science Foundation (NSF) grants CNS-0205286, CNS-0720565, CNS-0834473. Thanks to Mark Hill for valuable feedback on early versions of this paper, and the Hewlett Packard Corporation for equipment donations. Swift has a significant financial interest in Microsoft.

References

[Baugh 2008] Lee Baugh, Naveen Neelakantam, and Craig Zilles. Using hardware memory protection to build a high-performance, strongly-atomic hybrid transactional memory. In *ISCA 35*, June 2008.

[Baugh 2007] Lee Baugh and Craig Zilles. An analysis of I/O and syscalls in critical sections and their implications for transactional memory. In *TRANSACT 2*, August 2007.

[Birrell 2007] Andrew D. Birrell and Michael Isard. Automatic mutual exclusion. In *HotOS 11*, May 2007.

[Blundell 2007] 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 *ISCA 34*, June 2007.

[Cantrill 2008] Bryan Cantrill. Concurrency's shysters. `http://blogs.sun.com/bmc/entry/concurrency_s_shysters`, November 2008.

[Cascaval 2008] Calin Cascaval, Colin Blundell, Maged Michael, Harold W. Cain, Peng Wu, Stefanie Chiras, and Siddhartha Chatterjee. Software transactional memory: why is it only a research toy? *Commun. ACM*, 51(11):40–46, 2008.

[Damron 2006] Peter Damron, Alexandra Fedorova, Yossi Lev, Victor Luchango, Mark Moir, and Daniel Nussbaum. Hybrid transactional memory. In *ASPLOS 12*, October 2006.

[Dice 2006] Dave Dice, Ori Shalev, and Nir Shavit. Transactional locking ii. In *DISC 20*, September 2006.

[Dudnik 2009] Polina Dudnik and Michael M. Swift. Condition variables and transactional memory: Problem or opportunity? In *TRANSACT 4*, February 2009.

[Fedorova 2007] Alexandra Fedorova, Margo Seltzer, and Michael D. Smith. Improving performance isolation on chip multiprocessors via an operating system scheduler. In *PACT 16*, pages 25–38, 2007.

[Fetzer 2007] Christof Fetzer and Pascal Felber. Improving program correctness with atomic exception handling. *Journal of Universal Computer Science*, 13(8):1047–1072, 2007.

[Fujimoto 1989] Richard M. Fujimoto. The virtual time machine. In *Proceedings of the First ACM Symposium on Parallel Algorithms and Architectures*, June 1989.

[Gray 1993] Jim Gray and Andreas Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann, 1993. ISBN 1-55860-190-2.

[Hammond 2004] Lance Hammond, Vicky Wong, Mike Chen, Brian D. Carlstrom, John D. Davis, Ben Hertzberg, Manohar K. Prabhu, Honggo Wijaya, Christos Kozyrakis, and Kunle Olukotun. Transactional memory coherence and consistency. In *ISCA 31*, June 2004.

[Harris 2004] Tim Harris. Exceptions and side-effects in atomic blocks. In *PODC Workshop on Concurrency and Synchronization in Java Programs*, Jul 2004.

[Harris 2003] Tim Harris and Keir Fraser. Language support for lightweight transactions. In *OOPSLA 18*, October 2003.

[Harris 1991] Tim Harris, Simon Marlow, Simon Peyton Jones, and Maurice Herlihy. Composable memory transactions. In *PPOPP 12*, June 1991.

[Harris 2006] Tim Harris, Mark Plesko, Avraham Shinnar, and David Tarditi. Optimizing memory transactions. In *PLDI 2006*, June 2006.

[Haskin 1987] Roger Haskin, Yoni Malachi, Wayne Sawdon, and Gregory Chan. Recovery management in quicksilver. In *SOSP 11*, pages 107–108, November 1987.

[Herlihy 1992] Maurice Herlihy and J. Eliot B. Moss. Transactional memory: Architectural support for lock-free data structures. Technical Report Technical Report 92/07, Digital Cambridge Research Lab, 1992.

[Intel 2008] Intel. Intel c++ stm compiler prototype edition 2.0 language extensions and user's guide. Technical Report 318253-001US, Intel Corp., April 2008.

[Internet Systems Consortium] Internet Systems Consortium. Berkeley internet name domain (BIND). http://www.isc.org/index.pl?/sw/bind/.

[Jefferson 1985] David R. Jefferson. Virtual time. *ACM Transactions on Programming Languages and Systems*, 7(3):404–425, 1985.

[Jinmei 2006] Tatuya Jinmei and Paul Vixie. Implementation and evaluation of moderate parallelism in the BIND9 DNS server. In *Usenix ATC 2006*, June 2006.

[Jula 2008] Horatiu Jula, Daniel Tralamazza, Cristian Zamfir, and George Candea. Deadlock immunity: Enabling systems to defend against deadlocks. In *OSDI 8*, November 2008.

[Lowell 1998] David E. Lowell and Peter M. Chen. Persistent messages in local transactions. In *PODC 17*, pages 219–226, 1998.

[Lu 2006] Shan Lu, Soyeon Park, Eunsoo Seo, and Yuanyuan Zhou. Learning from mistakes: A comprehensive study on real world concurrency bug characteristics. In *ASPLOS 13*, October 2006.

[McDonald 2006] Austen McDonald, JaeWoong Chung, Brian Carlstrom, Chi Cao Minh, Hassan Chafi, Christos Kozyrakis, and Kunle Olukotun. Architectural semantics for practical transactional memory. In *ISCA 33*, June 2006.

[Menon 2008] Vijay Menon, Steven Balensiefer, Tatiana Shpeisman, Ali-Reza Adl-Tabatabai, Richard L. Hudson, Bratin Saha, and Adam Welc. Single global lock semantics in a weakly atomic STM. In *TRANSACT 3*, February 2008.

[Microsoft Corp. 2008] Microsoft Corp. Transactional memory team blog. `http://blogs.msdn.com/stmteam/default.aspx`, October 2008.

[Minh 2007] 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 *ISCA 34*, June 2007.

[Moore 2006] Kevin E. Moore, Jayaram Bobba, Michelle J. Moravan, Mark D. Hill, and David A. Wood. Logtm: Log-based transactional memory. In *HPCA 12*, pages 258–269, February 2006.

[Moravan 2006] 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 *ASPLOS 12*, pages 359–370, October 2006.

[Moss 2006] J. Eliot B. Moss. Open nested transactions: Semantics and support. In *Workshop on Memory Performance Issues*, February 2006.

[Ni 2008] Yang Ni, Adam Welc, Ali-Reza Adl-Tabatabai, Moshe Bach, Sion Berkowits, James Cownie, Robert Geva, Sergey Kozhukow, Ravi Narayanaswamy, Jeffrey Olivier, Serguei Preis, Bratin Saha, Ady Tal, and Xinmin Tian. Design and implementation of transactional constructs for c/c++. In *OOPSLA 23*, June 2008.

[Nightingale 2005] Edmund B. Nightingale, Peter M. Chen, and Jason Flinn. Speculative execution in a distributed file system. In *SOSP 20*, pages 191–205, October 2005.

[Olszewski 2007] Marek Olszewski, Jeremy Cutler, and J. Gregory Steffan. Judostm: A dynamic binary-rewriting approach to software transactional memory. In *PACT 2007*, September 2007.

[Oracle Corporation] Oracle Corporation. Oracle Berkeley Database. `http://www.oracle.com/database/berkeley-db/index.html`.

[Rossbach 2007] Christopher J. Rossbach, Owen S. Hofmann, Donald E. Porter, Hany E. Ramadan, Aditya Bhandari, and Emmett Witchel. TxLinux: Using and managing hardware transactional memory in an operating system. In *SOSP 21*, October 2007.

[Saha 2006a] Bratin Saha, Ali-Reza Adl-Tabatabai, Richard L. Hudson, Chi Cao Minh, and Benjamin Hertzberg. Mcrt-stm: a high performance software transactional memory system for a multi-core runtime. In *PPOPP 13*, March 2006.

[Saha 2006b] Bratin Saha, Ali-Reza Adl-Tabatabai, and Quinn Jacobson. Architectural support for software transactional memory. In *MICRO 39*, December 2006.

[Schlaeger 2008] Chris Schlaeger. The impact of operating systems on modern CPU designs (and vice versa). `http://arcs08.inf.tu-dresden.de/docs/arcs08_schlaeger-sld.pdf`, February 2008.

[Smaragdakis 2007] Y. Smaragdakis, A. Kay, R. Behrends, and M. Young. Transactions with isolation and cooperation. In *OOPSLA 22*, October 2007.

[Spear 2008] Michael F. Spear, Maged M. Michael, and Michael L. Scott. Inevitability mechanisms for software transactional memory. In *TRANSACT 3*, February 2008.

[Strandenæs 2002] Thomas Strandenæs and Randi Karlsen. Transaction compensation in web services. In *Norsk Informatikkonferanse*, June 2002.

[Sutter 2005] Herb Sutter and James Larus. Software and the concurrency revolution. *ACM Queue*, 3(7), September 2005.

[Swift 2008] Michael M. Swift, Haris Volos, Neelam Goyal, Luke Yen, Mark D. Hill, and David A. Wood. OS support for virtualizing hardware transactional memory. In *TRANSACT 3*, February 2008.

[Tremblay 2008] Marc Tremblay and Shailender Chaudhry. A third-generation 65nm 16-core 32-thread plus 32-scout-thread cmt sparc processor. In *ISSCC 2008 Conference Proceedings*, February 2008.

[Volos 2008] Haris Volos, Neelam Goyal, and Michael M. Swift. Pathological interaction of locks with transactional memory. In *TRANSACT 3*, February 2008.

[Wang 2007] Cheng Wang, Wei-Yu Chen, Youfeng Wu, Bratin Saha, and Ali-Reza Adl-Tabatabai. Code generation and optimization for transactional memory constructs in an unmanaged language. In *CGO 2007*, March 2007.

[Wang 2008] Yin Wang, Terence Kelly, Manjunath Kudlur, Stephane Lafortune, and Scott Mahlke. Gadara: Dynamic deadlock avoidance for multithreaded programs. In *OSDI 8*, November 2008.

[Welc 2008] Adam Welc, Bratin Saha, and Ali-Reza Adl-Tabatabai. Irrevocable transactions and their applications. In *SPAA 2008*, pages 285–296, October 2008.

[xiph.com] xiph.com. Ogg Vorbis documentation. `http://www.xiph.org/vorbis/doc/`.

[xmms.org] xmms.org. X Multimedia System. `www.xmms.org`.

[Yen 2007] Luke Yen, Jayaram Bobba, Michael R. Marty, Kevin E. Moore, Haris Volos, Mark D. Hill, Michael M. Swift, and David A. Wood. LogTM-SE: Decoupling hardware transactional memory from caches. In *HPCA 13*, pages 261–272, February 2007.

[Zilles 2006] Craig Zilles and Lee Baugh. Extending hardware transactional memory to support non-busy waiting and non-transactional actions. In *TRANSACT 1*, June 2006.