# Extending Transactional Memory with Atomic Deferral

**Tingzhe Zhou**[1], **Victor Luchangco**[2], **and Michael Spear**[3]

1 **Lehigh University, Bethlehem, USA**
`tiz214@lehigh.edu`
2 **Oracle Labs, Burlington, USA**
`victor.luchangco@oracle.com`
3 **Lehigh University, Bethlehem, USA**
`spear@lehigh.edu`

**Abstract**

This paper introduces *atomic deferral*, an extension to TM that allows programmers to move long-running or irrevocable operations out of a transaction while maintaining serializability: the transaction and its deferred operation appear to execute atomically from the perspective of other transactions. Thus, programmers can adapt lock-based programs to exploit TM with relatively little effort and without sacrificing scalability by atomically deferring the problematic operations. We demonstrate this with several use cases for atomic deferral, as well as an in-depth analysis of its use on the PARSEC dedup benchmark, where we show that atomic deferral enables TM to be competitive with well-designed lock-based code.

**1998 ACM Subject Classification** D.1.3 Concurrent Programming

**Keywords and phrases** Transactional Memory, Concurrency, Synchronization, I/O

**Digital Object Identifier** 10.4230/LIPIcs.OPODIS.2017.7

## 1 Introduction

Transactional memory (TM) [9], originally proposed as a hardware extension to facilitate the creation of scalable nonblocking data structures, is beginning to be widely available. It is supported by major hardware vendors [14], and a Technical Specification for C++ Extensions for Transactional Memory (henceforth, TMTS) has been proposed [10], and is (partially) implemented by the GCC compiler [5], with support for both hardware (HTM) and software (STM) implementations. Thus, programmers are at last able to use TM in production.

The appeal of TM is its simplicity: a programmer need only wrap an operation inside a language-level "transaction"; a run-time system executes the transaction using custom hardware and/or compiler-generated software instrumentation. The run-time system monitors the low-level memory accesses of transactions, and allows concurrent transactions to execute simultaneously as long as their memory accesses do not conflict. TM is particularly appealing for data structures and applications with irregular or hard-to-predict memory accesses (e.g., the rebalancing operations of a red-black tree mutation), making them difficult to implement efficiently using locks. As long as conflicts are rare, so that transactions can run concurrently with few rollbacks, and the implementation of TM itself does not introduce too much overhead, a TM-based implementation should be comparable in performance to lock-based code when running single-threaded, and scale much better with multiple threads.

Unfortunately, there are only a few examples of TM being used in "real" software [11,18,25]. Why is TM not more widely adopted? One reason is that TM implementations often do introduce significant overhead, especially when transactions are large and there is no hardware support. Another is that adapting a program to use TM is not always a simple matter of replacing lock-based critical sections with transactions because transactions cannot execute some kinds of operations (e.g., I/O and certain system calls), and most TMs do not support condition synchronization and other important synchronization patterns. Achieving good performance with TM often requires significant changes to the code both to reduce the size and number of large transactions and to move "unsafe" operations within critical sections out of transactions while still ensuring correct synchronization.

In this paper, we introduce language and run-time support for the *atomic deferral* of operations in transactions: deferred operations do not execute until after the transaction commits. Unlike prior work on deferred operations, atomic deferral does not violate serializability: concurrent transactions cannot observe an intermediate state in which the transaction's updates are complete but its deferred operation's updates are not. This property is particularly important for operations that perform output: if the output fails, compensating or retrying operations can be performed as part of the deferred operation so that it appears to be atomic with the deferring transaction.

We implement atomic deferral by introducing *transaction-friendly locks*, that is, locks that can be acquired and released within transactions, and to which transactions can "subscribe". With these locks, programmers can "mix and match" lock-based and transaction-based synchronization, using whichever is appropriate to the need. We use these locks to protect shared data accessed by deferred operations. The atomicity of the transaction and its deferred operation is preserved by acquiring the appropriate locks before committing the transaction.

The contributions of this paper are as follows:

- We introduce a transaction-friendly implementation of mutual exclusion locks.
- We present a mechanism for atomically deferring complex operations in transactions.
- We describe a compiler extension that allows the compiler to coordinate deferred operations with concurrent transactions without additional programmer effort.
- We describe use cases for atomic deferral in benchmarks and real-world programs.
- We show how atomic deferral can eliminate scalability bottlenecks in microbenchmarks and the PARSEC dedup workload.

The remainder of this paper is organized as follows: In Section 2, we review salient features of modern TM implementations. We provide an example of atomic deferral behavior in Section 3. Section 4 describe atomic deferral, and how it can be implemented using transaction-friendly locks. We discuss use cases in Section 5, and evaluate our implementation in Section 6. We describe related work and future research directions in Sections 7 and 8.

## 2 Background: Transactional Memory

In this section, we describe salient features of TM, particularly as it is specified in the C++ TMTS [10] and implemented by GCC. The TMTS specifies two lexical blocks for using TM: *atomic blocks* and *synchronized blocks*. Code within atomic blocks is restricted to ensure that the effects of an aborted transaction can be efficiently undone. To enforce this restriction statically, functions called while executing an atomic block must be declared *transaction-safe*. Synchronized blocks lift this restriction at the cost of possibly serializing all transactions (from both atomic and synchronized blocks) when executing an irrevocable operation (i.e., an operation that cannot be undone).

Although TM does not support condition synchronization, the *retry* operation proposed by Harris et al. [6] can produce a similar effect: A transaction that finds that some required

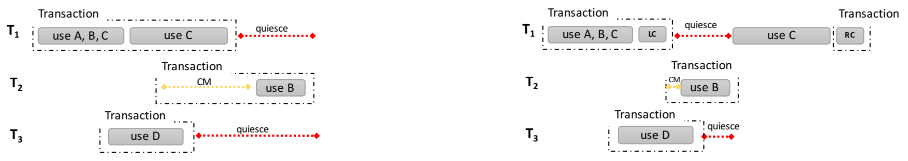


**Figure 1** Motivation for atomic defer. On the left, $T_1$'s transaction includes a long running operation using $C$. On the right, $C$ is locked, and then the operation on $C$ is deferred until after the transaction commits. The use of locking and deferral of the operation on $C$ enables the operations by threads $T_2$ and $T_3$ to progress more quickly, without violating serializability.

condition does not hold can invoke retry, which aborts the transaction and does not reschedule it until some location in its read set has changed. Thus, the transaction appears to execute only from a state in which the desired condition holds. (This differs from traditional condition synchronization in that the transaction aborts, and so its effects before discovering that the condition does not hold are undone.) The TMTS does not support retry directly, but it does provide the ability for atomic blocks to abort themselves, which we can use to approximate retry (albeit inefficiently).

The TMTS does not separate transactional and nontransactional memory: any location can be accessed both transactionally and nontransactionally. This presents a challenge to STM implementations known as the *privatization problem* [19]: Although semantically concurrent transactional and nontransactional access to the same location is a data race (which has undefined semantics in C++), a thread that uses a transaction to remove an object from a shared data structure and accesses it nontransactionally afterwards may conflict with another transaction that accessed the same object concurrently but is still "cleaning up". To avoid this problem, a thread must not access a privatized object nontransactionally until every transaction that may have accessed that object has completed entirely (i.e., committed or aborted, including clean up). Since an STM cannot, in general, determine when an object is privatized, implementations typically wait after committing any writing transaction until every concurrent transaction has completed entirely; this waiting is called *quiescing*.

When transactions repeatedly fail to commit due to repeated conflicts with other threads, a TM implementation may manage contention by delaying some transactions to increase the likelihood that others complete. Although contention management policies vary, most TM implementations employ serialization as a last resort: a transaction that fails too many times will request that all other transactions abort, and no new transactions commence, until it completes. Unless the workload exhibits pathological conflicts, serialization should be rare. In GCC, the default is for software transactions to serialize after 100 attempts, and hardware transactions to serialize after 2. Dynamically tuning this parameter has been shown to have a significant impact on some workloads [4]. Any nontrivial amount of serialization, however, has a terrible effect on performance, particularly because serialization delays all active transactions, even those from completely unrelated parts of the program (unlike locked critical sections, which are partitioned by the locks that they acquire).

## 3 A Motivating Example

To motivate atomic deferral, consider the execution depicted on the left side of Figure 1, which captures behavior we observed when transactionalizing the PARSEC dedup kernel [1]. $T_1$ executes a transaction that first accesses locations $A$, $B$ and $C$, and then does a lengthy operation that accesses only $C$. Concurrently, $T_2$ executes a transaction that accesses $B$.

Because these transactions conflict, either one of them must abort, or $T_2$ must wait until $T_1$ commits before it can proceed with the part of its transaction that accesses $B$, which is what happens in this case. Because $T_1$ may privatize some memory, after it commits, it must quiesce, waiting for $T_2$ to finish (either commit or abort).

The situation is even worse for $T_3$, which accesses a completely different location, and so does not conflict with either $T_1$'s or $T_2$'s transaction. Nonetheless, $T_3$ might privatize some memory, and thus it must quiesce until all concurrent transactions complete, so it must wait for $T_1$'s lengthy operation to complete, and then for $T_2$'s transaction to complete afterwards , before it can proceed.

Note, however, that $T_1$ is only accessing $C$ in the lengthy operation at the end of its transaction. If it could defer that operation until after it commits, then $T_2$ could start the section of its code that accesses $B$ earlier, and likely commit before $T_1$ completes its lengthy operation on $C$. $T_3$ can also stop quiescing earlier (i.e., when $T_2$ commits). This case is depicted one the right side of Figure 1.

One problem with doing this, however, is that a thread accessing $C$ after $T_1$ commits the initial part of its transaction but before $T_1$ finishes its final lengthy operation on $C$ will see an intermediate state of $T_1$'s transaction, violating atomicity. To avoid that, we should prevent other threads from accessing $C$ in that interval. This is represented by the small "LC" and "RC" operations (for "lock $C$" and "release $C$" respectively). Achieving this is the core conceptual contribution of this paper, and we show how to do it in the next section.

Prior studies of concurrent applications [12, 18, 21, 23, 24] found that output operations and long-running operations occur often while locks are held. The consequences of such operations are less severe in lock-based code than in programs with TM, primarily because the lock-based programs use many locks: a long-running operation protected by lock $L_1$ does not impede a thread executing a critical section protected by $L_2$. However, long-running operations tend to hold as few locks as necessary.

## 4 Extending TM with Atomic Deferral

We support atomic deferral using two new keywords: the `deferrable` annotation on classes, and the `atomic_defer` function, which takes as arguments a function and a list of objects, each of which must be an instance of a `deferrable` class. To defer an operation, a programmer calls `atomic_defer` with a function implementing the deferred operation and a list of all the shared objects that this function may access. Fields of `deferrable` objects must not be accessed directly, but only through getters and setters (a recommended software engineering practice in any case). Thus, if $o$ is an object with a `deferrable` class type and an *expensive* method, then we can defer the execution of that method within a transaction by writing:

```
λ ← () { o.expensive() }
atomic_defer(λ, o)
```

The deferred operation will be executed immediately after the enclosing transaction commits, and in such a way that no other transactions can see a state that reflects the effects of the transaction but not those of its deferred operation. A deferred operation will see any effects of the transaction that occur after the call to `atomic_defer`. If `atomic_defer` is called multiple times within a single transaction, the deferred operations will be executed in the order of their respective calls to `atomic_defer`, and the effects of earlier deferred operations will be visible to later ones.

**Listing 1:** Implementation of atomic deferral.

```
// Extensions to classes annotated as deferrable
deferrable class T
  lock : TxLock // implicit per-instance lock
  ...  // programmer-defined fields
function  transaction_safe Method(...)
    // subscribe to the implicit lock
    TxLock.Subscribe(lock)
    // programmer-defined logic
    ...

function atomic_defer(l : λ, objs: Deferrable ...)
    // Use transaction to acquire locks without deadlock
    transaction
        for o : objs do
            TxLock.Acquire(o)
    deferred_ops.append(⟨l, objs⟩)

Additional Per-Thread TM Metadata:
  // all deferred operations for current transaction
  deferred_ops   : list⟨λ, list⟨Deferrable⟩⟩

function TxEnd()
    // Standard STM Commit; HTM uses a special instruction
    ValidateReadsFinalizeWrites()
    // STM-only: ensure transaction finishes before λs run
    Quiesce()
    // Reset thread's TM metadata
    move(tm_free_list, local_frees)
    move(deferred_ops, local_defers)
    ResetLists()
    // Execute deferred operations
    for ⟨l, objs⟩ ∈ local_defers do
        l.execute()
        for o ∈ objs do TxLock.Release(o)
    // Reclaim memory, reset lists
    for ptr ∈ local_frees do free(ptr)
```

## 4.1 Implementing Atomic Deferral

We implement atomic deferral by using locks to protect accesses to `deferrable` objects. To provide atomicity, we acquire the locks required by the deferred operation before the transaction commits. We also need a way to notify transactions that access a `deferrable` object (directly as part of the transaction, not deferred) when the lock protecting it has been acquired (by a transaction that calls `atomic_defer` with the object): such transactions must abort and retry after the deferred operation has completed (and the corresponding locks released).

To this end, we designed *transaction-friendly locks*, which can be acquired and released within a transaction, and which provide a *subscribe* method. The subscribe method must be called from within a transaction, which blocks (or aborts) until the lock is either free or held by the subscribing thread. Multiple threads can subscribe to a lock if it is free. We describe how we implement transaction-friendly locks in Section 4.2.

Pseudocode for implementing atomic deferral appears in Listing 1. In addition to providing implementations for `atomic_defer` and `deferrable`, we modify the commit operation `TxEnd`. This code assumes that `TxLock` is a class of transaction-friendly locks, and that `Deferrable` is a base class for all `deferrable` classes.

For `deferrable` classes, we add a field that maintains a transaction-friendly lock that protects the class, and we inject a call to `TxLock.Subscribe` for this lock as the first instruction of the transaction-safe version of every member function.[1]

The `atomic_defer` function first acquires the locks of all the `deferrable` objects passed to it, and then appends all of its arguments (the function representing the deferred operation and the list of `deferrable` objects it may access) to *deferred_ops*, a thread-local list of deferred operations. This list will be used when the transaction commits, as described below.

When committing the transaction, we first proceed as usual, validating the read set, finalizing the writes, and then quiescing to avoid privatization problems. Remember that any object that might be accessed by deferred operations has already been locked (i.e., when `atomic_defer` was called with the object), so no other transaction can see any writes to it. We then execute the deferred operations in order, releasing the locks on the `deferrable` objects associated with each deferred operation after that operation is complete. (If an object is accessed by multiple deferred operations, each of them would have acquired the corresponding reentrant lock, and so it is not actually released until the last such operation completes.)

The enclosing transaction may have freed memory, which is normally deferred by the TM after the transaction has quiesced. Because deferred operations may refer to memory that was subsequently freed by the transaction, we delay the freeing of that memory a bit more, until all the deferred operations have completed.

Because deferred operations may use transactions internally, we need to make *deferred_ops* and *tm_free_list* available for their use. Thus, we copy them into local variables before executing any of the deferred operations.

To argue that this implementation is correct, that is, that a transaction and its deferred operations appear atomic, we draw an analogy with two-phase locking, a well-understood technique known to guarantee atomicity. Specifically, a transaction can be thought of as acquiring and holding a single global lock until the transaction commits. Because the lock for every object accessed by deferred operations is acquired before the transaction commits, there is an initial phase in which locks are only acquired (i.e., up to the point that the transaction commits), and a concluding phase in which locks are only released (including the implicit global lock released by the transaction on commit). So all locks are held between the time that the commit operation is invoked and the time that the commit actually occurs. We must also ensure that every access is protected by the appropriate lock, which is why the programmer must provide, when calling `atomic_defer`, all the objects that the deferred operation may access. If a deferred operation accesses some object not passed to `atomic_defer`, then a data race may occur.

## 4.2 Transaction-Friendly Mutex Locks

The heart of our atomic deferral mechanism is a transaction-friendly mutual exclusion lock, whose pseudocode appears in Listing 2. The `TxLock` is reentrant, storing an owner and a count of the locking depth. In this manner, a thread that holds the lock may re-acquire it by incrementing the count. Before any other thread can acquire the lock, the current owner must release the lock as often as needed to ensure $depth = 0$. Since the implementation uses transactions, the *owner* and *depth* fields need not be packed into a single machine word:

[1] STM implementations typically maintain two versions of each transaction-safe function, one that is called within transactions and one that is called when not in a transaction.

**Listing 2:** A transaction-friendly, reentrant mutex lock.

```
Fields of TxLock Object:
  owner   : transaction_id    // Lock holder ID
  depth   : Integer           // For reentrancy

function TxLock.Acquire(l)
    atomic
        // Common case: lock is unheld
        if l.owner = nil then
            l.owner ← me
            l.depth ← 1
            return
        // Handle reentrancy/nesting
        else if l.owner = me then
            l.depth ← l.depth + 1
            return
        // Else wait (spin and/or yield CPU) until lock is released
        spin()
        retry

function TxLock.Subscribe()
    // Must be in transaction to call
    if owner ≠ nil ∧ owner ≠ me then
        retry

function TxLock.Release(l)
    atomic
        // [Optional] Forbid handoff of held lock
        if l.owner ≠ me then
            fatal error
        // Handle reentrancy/nesting
        else if l.depth > 1 then
            l.depth ← l.depth − 1
            return
        // Else no reentrancy/nesting
        l.depth ← 0
        l.owner ← nil
```

they are only accessed within transactions. A thread that is currently in a transaction may acquire and/or release `TxLock`s, because it is correct in C++ to nest transactions. Among other things, this means that a thread can acquire multiple locks in a deadlock-free fashion, even without a global locking order: it need only issue all acquisitions inside of a transaction.

`TxLock`s are elidable within transactions, via the `Subscribe` method: a transaction that subscribes to a `TxLock` blocks until the lock is either unheld, or held by the calling thread. Subscription only reads the *owner* field, which allows concurrent subscription by multiple threads. When any thread acquires the `TxLock`, all subscribing transactions will conflict with the new lock owner, and will abort. When the `TxLock` is acquired, the C++ TMTS ensures correct fence semantics: since the transaction accesses shared memory, the TM implementation is required to guarantee that memory accesses preceding the transaction order before it, and memory accesses following the transaction order after it.

If the C++ TMTS adds efficient support for `retry`, transactions could yield the CPU if they attempt to acquire or subscribe to a lock that is held by another thread, and would be woken automatically when the lock is released. In the meantime, we implement `retry` by placing an `atomic` transaction inside a while loop, replacing the `retry` instruction with an exception throw, and adding a `break` as the last statement in the transaction.

**Listing 3:** Diagnostic logging from a critical section.

```
// Version with irrevocable transactions
synchronized
    // x is a mutable string
    // i is a mutable integer
    ...
    fprintf(stderr, str, x, i)
    free(x) // Optional
    ...
```

```
// Deferrable for the log file's descriptor
class defer_fprintf: public Deferrable
    fd : file    // output file descriptor
    ...

// global instance of log file descriptor
df : defer_fprintf(fd ← stderr)
```

```
// Version with atomic_defer
atomic
    ...
    tmp ← sprintf(str, x, i)
    λ ←()
        fprintf(df.fd, tmp)
        free(tmp)
        free(x) // Optional
    atomic_defer(λ, df)
    ...
```

## 4.3 Practical Concerns

Deferring operations creates a nonlinear control flow within a program. This nonlinearity is not observable to concurrent threads: the transaction and its deferred operations appear to be a single, serializable operation. However, within the transaction, the programmer must be mindful of a few challenges. First, the state of the object and thread-private data at the time when the `atomic_defer` keyword appears is not immutable, and may change in the suffix of the transaction that executes before the deferred operation. In addition, the deferred operation does not execute transactionally, and thus races can occur if the deferred operation accesses shared data not protected by the associated `TxLocks`. Second, the programmer must encapsulate shared objects carefully. Consider a deferred operation that performs a `write` of byte stream $B$ to file descriptor $F$. If $F$ is shared, then it should be a field of a `Deferrable` object. If $B$ is shared, then it, too, should be a field of a `Deferrable` object. Programmers must decide if $B$ and $F$ should be fields of the same `deferrable` object, or of multiple objects. Third, since system calls made within a deferred operation happen immediately, some possibility for performance bottlenecks remains. For example, an `fsync` within a deferred operation is often necessary. With `atomic_defer`, the `fsync` and any associated error recovery can be atomic with the transaction, and will not block *all* transactions. However, lengthy deferred operations will still block concurrent transactions that call a method of the associated `deferrable` objects.

## 5 Programming With Atomic Defer

We now present examples of `atomic_defer` in real applications. The examples depict common use cases, and show the deferral of increasingly complex operations without sacrificing atomicity or resorting to serialization.

### 5.1 Basic Logging

In programs such as memcached [18] and Atomic Quake [27], critical sections occasionally perform logging operations, such as error messages and diagnostic writes to per-thread logs. The program does not require any ordering among logging operations: they are timestamped, and the order can be determined post-mortem. The return values of the output operations are typically ignored. An example appears in Listing 3.

When the values to be logged (`x` and `i`) are mutable shared data, existing programs resort to irrevocability or they skip the logging operation. When the values can be encapsulated in a `Deferrable` object, `atomic_defer` is a straightforward transformation: the output string is prepared within the transaction, and the output is deferred until the end of the transaction.

**Listing 4:** Durable output with guaranteed order.

```
     // Deferrable wrapper for file descriptors
     class defer_fd: public Deferrable
         fd : file      // output file descriptor
         ...

     // Deferrable objects
     $fd_{D1}$ : defer_fd(fd ← ...)
     $buff_{D1}$ : defer_buffer(buf ← ..., flag ← false)
     // Durable output to $fd_{D1}$
     atomic
 1       ...
 2       λ ←()
 3           write($fd_{D1}$.fd, $buff_{D1}$.buffer)
 4           fsync($fd_{D1}$.fd)
 5           $buff_{D1}$.flag ← true
 6       atomic_defer(λ, $fd_{D1}$, $buf_{D1}$)

     // Deferrable wrapper for output buffer
     class defer_buffer: public Deferrable
         buf : buffer     // buffer data
         flag: boolean    // is buffer written?

     // Deferrable objects
     $fd_{D2}$ : defer_fd(fd ← ...)
     $buff_{D2}$ : defer_buffer(buf ← ..., flag ← false)
     // Conditional durable output to $fd_{D2}$
     atomic
 7       Subscribe($buff_{D1}$)
 8       if $buff_{D1}$.flag then
 9           λ ←()
10               write($fd_{D2}$.fd, $buff_{D2}$.buffer)
11               fsync($fd_{D2}$.fd)
12               $buff_{D2}$.flag ← true
13           atomic_defer(λ, $fd_{D2}$, $buf_{D2}$)
```

Note that this approach ensures ordering of all logging operations on the encapsulated file descriptor. A simpler approach, when ordering is not needed, is to pass **nil** as the second argument on line 8. This approach causes serialization only among transactions that use $df$.

Because transactional versions of existing programs tend to omit this instrumentation in order to avoid serialization, we did not observe a performance impact when applying `atomic_defer` to memcached. However, atomic deferral keeps the code robust and complete without adding too much burden on programmer, and it makes it easier to debug programs during development, by enabling non-serializing `printf` debugging.

## 5.2 Durable Output

Programs often rely on the `fsync` system call to persistent output. In some cases (e.g., durable database operations), it is necessary to order outputs based on the timing of `fsync` calls, such that when there are two files: $F_1$ and $F_2$, $F_2$is not updated until after $F_1$'s updates have reached the disk. Simply deferring an `fsync` operation in this case is insufficient. With `atomic_defer`, we can encapsulate the completion status of the `fsync` in a `Deferred` object that is associated with the deferred `fsync` operation.

In Listing 4, one thread executes the transaction ($T_1$) on lines 1 to 6, and another executes the transaction ($T_2$) on lines 7 to 13. We wish to ensure that $T_2$ does not write $buff_{D2}$ to file $fd_{D2}$ unless $T_1$'s write of $buff_{D1}$ to file $fd_{D1}$ has been persisted to disk. Since the flag indicating the completion of $T_1$'s `fsync` is encapsulated in a `Deferrable` object, and $T_1$ sets that flag in an operation that has been deferred, we know that $buff_{D1}$'s implicit lock will be held during the time that the flag is set, and will not be released until after the `fsync` returns. Consequently, when $T_2$ executes line 7, three cases are possible: (1) $T_1$ has not yet executed line 6, in which case line 7 returns, and then line 8 evaluates to false; (2) $T_1$ has executed line 6 but has not completed lines 3 to 5, in which case $T_2$ will call `retry` and ultimately land in the third case; or (3) $T_1$ has completed its deferred execution of lines 3 to 5, in which case $T_2$ can subscribe to $buff_{D1}$, and then observe a **true** value on line 8. Note that $T_2$ can only perform its write in the third case, which orders lines 3–5 before lines 10–12, and thus the deferred outputs occur and reach the disk in the required order.

**Listing 5:** MySQL critical sections in file pool management that are used in asynchronous I/O.

```
// atomic_defer: types and variables
class file_system_t: public Deferrable
  ...
  space_list : file_space_t

// wrap the file system as a deferrable object
file_system : file_system_t

mySQL_initialize (...)
    // open tablespace data files
    atomic
        ...
        $\lambda \leftarrow ()$
            for space $\in$ space_list
                for node $\in$ space
                    node $\leftarrow$ open(...)
        atomic_defer($\lambda$, file_system)
        ...
mySQL_destroy (...)
    // close tablespace data files
    atomic
        ...
        $\lambda \leftarrow ()$
            for space $\in$ space_list
                for node $\in$ space
                    close(node)
        atomic_defer($\lambda$, file_system)
        ...

mySQL_io_prepare (...)
    close_more :
    atomic
        // check system states and select files
        ...
        $\lambda \leftarrow ()$
            if close(file) $= -1$
                error
            n_open $\leftarrow$ n_open $- 1$
            if (n_open $\geq$ max_n_open)
                need_close $\leftarrow$ true
                goto end
            // check the node to do I/O
            if $\neg$node.open
                // get file size, do an open and close
                // save metadata for future I/O
                if node.size $= 0$
                    node $\leftarrow$ open()
                    offset $\leftarrow$ lseek(file, 0, SEEK_END)
                    success $\leftarrow$ pread(two pages)
                    close(node)
                node $\leftarrow$ open()
            end :
        atomic_defer($\lambda$, file_system)
        ...
    if (need_close)
        goto close_more
```

## 5.3 Opening Files as Output

Our final example comes from the MySQL InnoDB storage engine. InnoDB maintains a pool of file descriptors, which is protected by a lock. Metadata is associated with each file

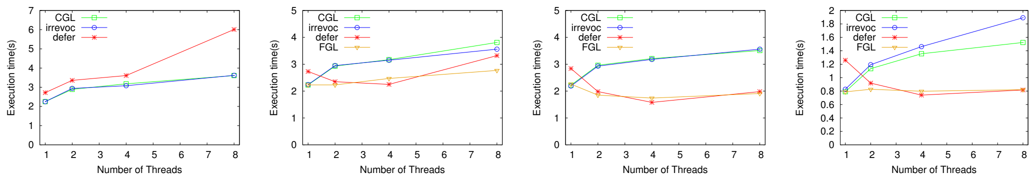


**Figure 2** Atomic_defer performance on an I/O microbenchmark (a) - (d).

descriptor, and allows updates to files to be performed via asynchronous I/O. For example, to append new records to the end of a file, a thread locks the pool, updates the size of the file, and then unlocks the pool. It then issues an asynchronous write. Subsequent appends will follow the same protocol, and hence will appear at a later point in the file, even if their writes reach the disk earlier.

While reads and writes do not occur in critical sections, and hence would not serialize a transactional version of InnoDB, the management of the pool depends on the ability to open and close files dynamically, in order to stay below a pre-set maximum number of open files. If a file must be opened when the pool is at capacity, then a thread will lock the pool, close some other files that do not have outstanding accesses in-flight, and then open the new file. In transactional InnoDB[2], this operation requires irrevocability, and serializes all memory transactions, to include those performing read-only queries of data within the database.

With `atomic_defer`, the pool becomes a `Deferrable` object. On any modification to file descriptor metadata, a thread uses a transaction that subscribes to the pool. Thus, file operations can proceed fully in parallel, since they use asynchronous I/O to perform their file accesses, and transactions to operate on disjoint file metadata regions. In the uncommon cases where files are opened and closed, the system calls are deferred from a transaction. While the system calls are in-flight, concurrent accesses to the pool stall (via `retry`). Once the pool is returned to a usable state, any suspended threads resume.

## 6 Performance Evaluation

We now present experiments that demonstrate the benefit of `atomic_defer`. We conduct tests on two platforms. In charts depicting scalability up to 8 threads, the platform is a 4-core/8-thread Intel Core i7-4770 CPU running at 3.40GHz. This CPU supports Intel's TSX extensions for HTM, includes 8 GB of RAM, and runs a Linux 4.3 kernel. Experiments with larger thread counts were conducted on a machine with two 18-core/36-thread Intel E5-2699 V3 CPUs running at 2.30GHz. This CPU also supports TSX, includes 128 GB of RAM, and runs a Linux 4.8 kernel. Our extensions were implemented in GCC 5.3.1. Results are the average of 5 trials.

### 6.1 Performance of atomic_defer on a Transactional I/O Microbenchmark

One motivation for `atomic_defer` is to avoid the serialization of `synchronized` transactions, while allowing output that is atomic with respect to the transaction. We begin with a microbenchmark study to observe the behavior of transactions that perform irrevocable operations on files. Our microbenchmarks are patterned after work by Demsky and Tehrany [3].

[2] Unfortunately, adding TM to InnoDB revealed a bug in GCC, which produces an internal compiler error.

**Listing 6:** An example of deferring I/O and system calls.

```
  // Encapsulate streams in a Deferrable object
  class defer_file: public Deferrable
      input     // input stream
      output    // output stream

  // An array of files
  $dfs$: $defer\_file[]$

  // Operation to be deferred
1 $\lambda \leftarrow$(id, content)
      // Read File
2     if $\neg dfs[id].input.open()$ then error
      // Get the length of the file
3     $dfs[id].input.seekg(0, end)$
4     $len \leftarrow dfs[id].input.tellg()$
5     $dfs[id].input.close()$
      // Write to the file and close
6     $tmp \leftarrow format(content, len)$
7     $dfs[id].output.write(tmp)$
8     $dfs[id].output.close()$

  // Irrevocable version of benchmark
1 synchronized
2     $content \leftarrow \ldots$
3     $id \leftarrow \ldots$
4     $\lambda(id, content)$

  // atomic_defer version of benchmark
1 atomic
2     $content \leftarrow \ldots$
3     $id \leftarrow \ldots$
4     $atomic\_defer(\lambda(id, content),\ dfs[id])$
```

Whereas they required custom instrumentation of system calls in order to make them transaction safe, we run I/O operations without instrumentation, using either irrevocability or `atomic_defer`. Listing 6 presents the general behavior of our microbenchmarks: a transaction produces content and identifies a file to update. It then performs I/O, which includes opening a file, reading the file length, and appending data to the file. The I/O can be deferred or executed irrevocably. To use `atomic_defer`, we encapsulate the I/O streams in `deferrable` objects, and then use `atomic_defer` to delay the operation on line 5. Figure 2 presents experiments with four configurations of the microbenchmark. In each case, threads cooperate to complete a total of 1M operations. The figure presents results for STM, but trends for HTM are the same.

Figure 2a explores the overhead of atomic deferral when there is only one file, and hence no concurrency, by comparing performance when transactions are replaced with a coarse-grained lock (CGL), and when transactions use irrevocability (irrevoc), or atomic deferral (defer). We see that the baseline GCC TM implementation (irrevoc) is well-tuned to handle irrevocability: it serializes transactions early, avoids instrumentation, and achieves performance comparable to CGL. In contrast, `atomic_defer` pays a constant overhead per transaction to support rollback, even though no rollbacks occur. As the thread count increases, overheads due to `retry` cause additional slowdown. This is partly a result of our `retry` implementation aborting and immediately retrying, instead of de-scheduling the transaction until it can make progress. Until the C++ TMTS includes efficient `retry`, this cost is unavoidable.

Figures 2b and 2c expand the number of files to 2 and then 4, and threads update files with equal probability. We include another non-transactional baseline, with one fine-grained lock (FGL) per file. We again see that single-threaded code has higher overhead when using `atomic_defer`, due to instrumentation and the management of lambdas. While the behavior

**Listing 7:** Deferring reliable output in PARSEC dedup.

```
function pipeline_out(buf, len, fd)
    // fd may be unreliable, so monitor progress of writes
    $(p, nsent, rv) \leftarrow (buf, 0, 0)$
    while $nsent < len$
        $rv \leftarrow$ write$(fd, p, len - nsent)$
        if $transient\_error(rv)$ then
            continue
        if $fatal\_error(rv)$ then error
        $nsent \leftarrow nsent + rv$
        $p \leftarrow p + rv$
    $fsync(fd)$
    $free(buf)$
```

```
// Version with irrevocable transactions
synchronized
    ...
    $pipeline\_out(packet.buf, len)$
    ...

// Version with atomic_defer: packet is now deferrable
deferrable class packet
atomic
    ...
    $\lambda \leftarrow ()$
        $pipeline\_out(packet.buf, len)$
    ...
    $atomic\_defer(\lambda, packet)$
```

of CGL and irrevoc is unchanged, deferral now shows scaling on par with fine-grained locks, achieving indistinguishable performance at 2 and 4 threads. When the thread count greatly exceeds the potential concurrency (e.g., 8 threads and 2 files, Figure 2b), we still see extra overheads from `retry`. However, when there is enough concurrency in the workload (e.g., 4 files), `atomic_defer` scales well.

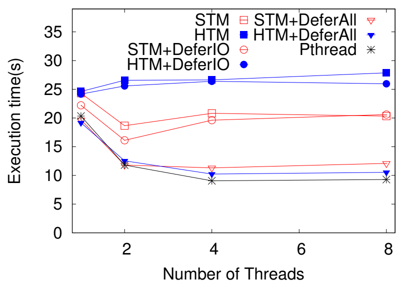
Finally, in Figure 2d transactions append to files that are kept open throughout the experiment. There are still 4 files, but the smaller critical sections reveal an overhead in the irrevocability mechanism: when one transaction becomes irrevocable, the others block, possibly yielding the CPU. When the irrevocable transaction is brief, the overhead of yielding becomes visible, and irrevoc degrades worse than CGL. Meanwhile, FGL has flat performance, and defer overcomes latency at 1 thread to be competitive with FGL.

## 6.2 Performance of atomic_defer on PARSEC Dedup

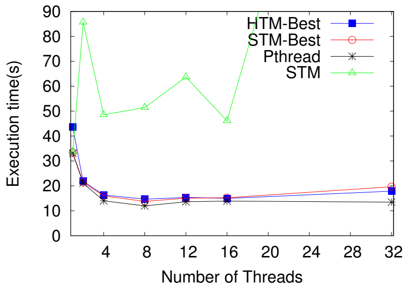
Wang et al. reported [21] that PARSEC's dedup kernel [1] ceased to scale when transactions replaced locks. Dedup is a pipeline application, and the original file output stage performs output while holding a lock; Wang's version replaces that lock with an irrevocable transaction. When the irrevocable transaction executes, it must serialize all concurrent transactions.

When we rewrote dedup's output operation to use `atomic_defer`, irrevocability ceased to cause performance degradation, but the benchmark still scaled poorly. A sketch of the code transformation appears in Listing 7. Since the buffer to be output was already encapsulated in a struct ("packet"), we made that struct `deferrable` and ensured that its fields were accessed through getters and setters. Deferring the operation was then a one-line change, which preserved the ordering of `fsync` operations and error handling with respect to output and subsequent concurrent accesses. The performance of dedup with this change appears in Figure 3 (a), as "+DeferIO".

We discovered that the `Compress` function was marked as `pure`, because it does not access any shared memory. Marking the function `pure` indicates to the compiler that the function can be run without instrumentation, lacks side effects, and can be run from a non-irrevocable context even when the compiler cannot prove that irrevocability is not needed. `Compress` is a long-running function, and in HTM, it accesses more memory than can be tracked by the HTM; the HTM execution serializes whenever a call to `Compress` exceeds the capacity of the hardware. In STM, the call to `Compress` represents a period of time during which other transactions can commit, but will delay in their quiesce operations. While the run-time behaviors are different, the consequence is the same: when one transaction calls `Compress`, other transactions cannot make progress.



(a) Effects of `atomic_defer` on I/O and pure functions



(b) Overall performance

**Figure 3** Performance of PARSEC dedup with `atomic_defer`.

Since `Compress` is pure, it can be deferred. We encapsulated the compressed buffer as a `deferrable` object, so that the run-time system can suspend transactions when they attempt to access a buffer that is locked for deferred compression. This has a profound impact on both STM and HTM. In HTM, the transaction ceases to overflow hardware capacity, and serialization is avoided. In STM, compression ceases to impede quiescence, and concurrent threads can make forward progress. In Figure 3(a), we see that the "+DeferAll" curves for both HTM and STM now compete with pthread locks, representing a 1.7x speedup for STM and 2.7x speedup for HTM.

Lastly, we measure the impact of `atomic_defer` on the 36-core system, to see how performance scales across chips and in the face of significantly more hardware parallelism. In Figure 3 (b), the performance of the baseline HTM is not shown: the 32-thread STM performance exceeds 270 seconds, and HTM never scales. When we employ `atomic_defer` to move output and pure functions out of transactions, both STM and HTM improve by roughly 10x compared to their respective TM baselines, reaching the same performance as pthread locks. While these optimizations require more careful reasoning about the program, we contend that these optimizations are still easier than reasoning about fine-grained locking.

## 7 Related work

Atomic deferral is a general mechanism for moving code out of transactions, but retaining serializability, through the composition of TM with two-phase locking. As we have seen in our examples and evaluation, atomic deferral offers an implementation-agnostic technique for performing output operations and other system calls within transactions. It also enables the movement of costly operations outside of the constrained environment presented by a general-purpose TM. Our work bears resemblance to, and is inspired by, several prior works in the areas of deferral, transactional system calls, irrevocability, and escape actions. Below, we summarize the most significant relationships.

**Support for Deferred Operations.** Language-level support for deferred system calls was first proposed by Carlstrom et al. [2], and expanded by Ni et al. [16]. These proposals took a broad approach to deferral, and considered deferring operations for both committed and aborted transactions. However, these deferred operations do not run atomically with the parent transaction, nor can they access transactional data without either copying or relying on ad-hoc synchronization in the deferred operation. Relative to these works, our contributions are the addition of locks and the introduction of a two-phase locking pattern that ensures serializability.

Rossbach et al. [17] were first to use locks to coordinate transactions and non-transactional code. Volos et al. [20] followed with a comprehensive approach to deferral focused on enabling transactional system calls (including I/O). Volos extended the OS with "sentinel" locks, which allowed software transactions to exclusively access file descriptors and other resources. Using these sentinel locks, output operations could appear to execute in the context of a transaction, but actually run after the transaction committed. On the one hand, Volos's work is more comprehensive than ours, as it deals with a wide array of system calls, and requires less programmer effort to perform transactional I/O. On the other hand, our work is more practical: it does not require a deadlock detection algorithm within the OS, or any OS modifications; it is free of system calls, and hence compatible with HTM (today's HTM systems abort on any change of privilege); and it allows the programmer to control the granularity at which operations are serialized (e.g., in the case of MySQL's file descriptor pool, where one lock abstractly covers an unbounded set of file descriptors). Our approach also makes it easier to handle timing and errors in deferred operations; we are not aware of a way to use Volos's work to achieve the behaviors in Listings 4 and 5.

**Irrevocable Operations.** Prior work has encouraged the broad use of irrevocability [16] for transactional I/O, contention management, and even as a performance optimization for to reduce logging overheads in long-running STM transactions. Clearly atomic deferral does not obviate irrevocability. In particular, our work assumed that the continuation of a transaction does not depend on the result of the deferred operation. For output to unreliable media (e.g., Listing 7), atomic deferral is sufficient. However, if the error on line 6 ought to influence operations after line 4 of the transaction on the right side of the listing, then irrevocability is the only known solution.

**Escape Actions.** Another approach to reducing the costs targeted by atomic deferral is the use of escape actions. These may be ad-hoc [26], or formalized as open nesting [15] or transactional boosting [8]. Like atomic deferral, these mechanisms provide a way to avoid logging overhead in complex operations, and also to perform I/O operations within transactions. However, these techniques are rarely compatible with HTM [13] (an exception is the IBM POWER TM [14]). Additionally, in STM, these techniques reduce the transactional footprint, but still run in the context of an active transaction; consequently, they retain the quiescence-associated delays shown in Figure 1. These techniques also require ad-hoc compensating actions and error handlers. On the other hand, boosting techniques offer great promise in areas where atomic defer is not useful, such as the creation of highly concurrent data structures [7, 22].

## 8 Conclusions and Future Work

In this paper, we presented a technique for atomically deferring operations in memory transactions. The key feature of our work is that concurrent transactions cannot detect that an operation was deferred: the operation appears atomic with the corresponding transaction, which retains serializability. The fundamental technique to enable atomic deferral is composing transactions with locks and `retry`-based condition synchronization, to facilitate a form of two-phase locking. With the deferred operation, transactions may perform complex operations and access a subset of shared memory. Using atomic deferral allows transactions to perform output without serializing, and was the foundation for a dramatic improvement in the performance of the PARSEC dedup benchmark.

Atomic deferral requires more complex reasoning by programmers than irrevocability, and is less general. However, when applicable, it eliminates serialization overheads, and shortens the time that transactions spend quiescing. In our view, the additional programmer overhead to use atomic deferral is small, and more than justified by the benefits. For example, we presented a scenario where atomic deferral can avoid serialization when managing file descriptor pools in MySQL, and another where files can be updated in order, while obeying strong persistence requirements.

As future work, we are interested in tools for automatically transforming output operations into deferred operations, and studying the relationship between atomic deferral and nested transactions. We are also interested in crafting a more formal correctness argument, which may influence the use of transaction-friendly locks in a greater range of workloads.

**Acknowledgements.** We thank Dave Dice and Tim Harris for their advice and guidance during the conduct of this research. At Lehigh University, this work was supported in part by the National Science Foundation under Grant CAREER-1253362, and through financial support from Oracle. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.

## References

1 Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. The PARSEC Benchmark Suite: Characterization and Architectural Implications. In *Proceedings of the 17th PACT*, Toronto, ON, Canada, oct 2008.

2 Brian D. Carlstrom, Austen McDonald, Hassan Chafi, JaeWoong Chung, Chi Cao Minh, Christos Kozyrakis, and Kunle Olukotun. The Atomos Transactional Programming Language. In *Proceedings of the 27th PLDI*, jun 2006.

3 Brian Demsky and Navid Tehrany. Integrating File Operations into Transactional Memory. *Journal of Parallel and Distributed Computing*, 71(10):1293–1304, 2011.

4 Nuno Diegues, Paolo Romano, and Luis Rodrigues. Virtues and Limitations of Commodity Hardware Transactional Memory. In *Proceedings of the 23rd PACT*, Edmonton, AB, Canada, aug 2014.

5 Free Software Foundation. Transactional Memory in GCC [online]. 2016. URL: `http://gcc.gnu.org/wiki/TransactionalMemory`.

6 Tim Harris, Simon Marlow, Simon Peyton Jones, and Maurice Herlihy. Composable Memory Transactions. In *Proceedings of the 10th PPoPP*, Chicago, IL, jun 2005.

7 Ahmed Hassan, Roberto Palmieri, and Binoy Ravindran. Optimistic Transactional Boosting. In *Proceedings of the 19th PPoPP*, Orlando, FL, feb 2014.

8 Maurice Herlihy and Eric Koskinen. Transactional boosting: A methodology for highly-concurrent transactional objects. In *Proceedings of the 13th PPoPP*, Salt Lake City, UT, feb 2008.

9 Maurice P. Herlihy and J. Eliot B. Moss. Transactional Memory: Architectural Support for Lock-Free Data Structures. In *Proceedings of the 20th ISCA*, San Diego, CA, 1993.

10 ISO/IEC JTC 1/SC 22/WG 21. Technical Specification for C++ Extensions for Transactional Memory, May 2015.

11 Tomas Karnagel, Roman Dementiev, Ravi Rajwar, Konrad Lai, Thomas Legler, Benjamin Schlegel, and Wolfgang Lehner. Improving In-Memory Database Index Performance with Intel Transactional Synchronization Extensions. In *Proceedings of the 20th HPCA*, Orlando, FL, February 2014.

**12** Matthew Kilgore, Stephen Louie, Chao Wang, Tingzhe Zhou, Wenjia Ruan, Yujie Liu, , and Michael Spear. Transactional Tools for the Third Decade. In *Proceedings of the 10th TRANSACT*, Portland, OR, jun 2015.

**13** Yujie Liu, Stephan Diestelhorst, and Michael Spear. Delegation and Nesting in Best Effort Hardware Transactional Memory. In *Proceedings of the 24th SPAA*, Pittsburgh, PA, jun 2012.

**14** Takuya Nakaike, Rei Odaira, Matthew Gaudet, Maged M. Michael, and Hisanobu Tomari. Quantitative Comparison of Hardware Transactional Memory for Blue Gene/Q, zEnterprise EC12, Intel Core, and POWER8. In *Proceedings of the 42nd ISCA*, Portland, OR, June 2015.

**15** Yang Ni, Vijay Menon, Ali-Reza Adl-Tabatabai, Antony Hosking, Rick Hudson, Eliot Moss, Bratin Saha, and Tatiana Shpeisman. Open Nesting in Software Transactional Memory. In *Proceedings of the 12th PPoPP*, San Jose, CA, mar 2007.

**16** 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 *Proceedings of the 23rd OOPSLA*, Nashville, TN, USA, 2008.

**17** 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 21st SOSP*, Stevenson, WA, oct 2007.

**18** Wenjia Ruan, Trilok Vyas, Yujie Liu, and Michael Spear. Transactionalizing Legacy Code: An Experience Report Using GCC and Memcached. In *Proceedings of the 19th ASPLOS*, Salt Lake City, UT, mar 2014.

**19** Michael Spear, Virendra Marathe, Luke Dalessandro, and Michael Scott. Privatization Techniques for Software Transactional Memory (POSTER). In *Proceedings of the 26th PODC*, Portland, OR, 2007.

**20** Haris Volos, Andres Jaan Tack, Neelam Goyal, Michael Swift, and Adam Welc. xCalls: Safe I/O in Memory Transactions. In *Proceedings of the EuroSys2009 Conference*, Nuremberg, Germany, March 2009.

**21** Chao Wang, Yujie Liu, and Michael Spear. Transaction-Friendly Condition Variables. In *Proceedings of the 26th SPAA*, Prague, Czech Republic, jun 2014.

**22** Lingxiang Xiang and Michael Scott. Software Partitioning of Hardware Transactions. In *Proceedings of the 20th PPoPP*, San Francisco, CA, feb 2015.

**23** Tingzhe Zhou, Victor Luchangco, and Michael Spear. Brief Announcement: Extending Transactional Memory with Atomic Deferral. In *Proceedings of the 29th SPAA*, Washington DC, July 2017.

**24** Tingzhe Zhou and Michael Spear. The Mimir Approach to Transactional Output. In *Proceedings of the 11th TRANSACT*, Barcelona, Spain, mar 2016.

**25** Tingzhe Zhou, PanteA Zardoshti, and Michael Spear. Practical Experience with Transactional Lock Elision. In *Proceedings of the 46th ICPP*, Bristol, UK, August 2017.

**26** Craig Zilles and Lee Baugh. Extending Hardware Transactional Memory to Support Non-Busy Waiting and Non-Transactional Actions. In *Proceedings of the 1st TRANSACT*, Ottawa, ON, Canada, jun 2006.

**27** Ferad Zyulkyarov, Vladimir Gajinov, Osman Unsal, Adrian Cristal, Eduard Ayguade, Tim Harris, and Mateo Valero. Atomic Quake: Using Transactional Memory in an Interactive Multiplayer Game Server. In *Proceedings of the 14th PPoPP*, Raleigh, NC, feb 2009.