The Mimir Approach to Transactional Output

Tingzhe Zhou and Michael Spear
Lehigh University
{tiz214, spear}@cse.lehigh.edu

Abstract

In order to use transactional memory (TM) in place of locks, it is necessary to design linguistic mechanisms that enable transactions to achieve the same outcomes as lock-based code. The mechanisms need not match their lock-based equivalents exactly, but must provide the same abilities "in spirit," so that programmers can (perhaps with nontrivial code rewriting) achieve the same behaviors and guarantees from transactions as with locks.

In this paper, we focus on the question of transactional output, and introduce the Mimir methodology. Mimir employs an observation about two-phase locking and language-level transactional semantics to enable *deferred* output operations that appear to execute in *isolation* with respect to all concurrent transactions, but without serializing those other transactions. The technique employs *ephemeral privatization* and *retry*-based condition synchronization in a manner that is invisible to concurrent transactions. Most significantly, Mimir avoids buffering of data in order to defer output, and is compatible with both software and hardware TM.

Categories and Subject Descriptors D.1.3 [*Programming Techniques*]: Concurrent Programming—Parallel Programming

Keywords Transactional Memory, I/O, Privatization, Object Orientated Programming, Condition Synchronization

1. Introduction

The effort to standardize Transactional Memory (TM) [12] support in C++ [2] has, to date, taken a pragmatic approach with respect to I/O. Clearly, if transactions are to replace lock-based code, then it must be possible to perform I/O operations on shared data, despite the possibility of concurrent attempts to access that same shared data. However, I/O performance has not seen much attention. In particular, it has been assumed that an I/O transaction can be statically identified by the programmer, and that it is acceptable to serialize all transactions at the time when I/O is attempted, so as to prevent concurrent accesses to the data during the I/O operation. This mechanism, broadly, is known as "irrevocability" [14, 18, 19, 23].

Irrevocability is a coarse-grained mechanism, which allows the execution of arbitrary operations within a transaction, even if those operations cannot be undone. Examples include accessing device registers, arbitrary system calls, communication with other threads via `volatile` and `atomic` variables, and I/O. Any student of Amdahl's law will immediately recognize global serialization of transactions as a potentially significant bottleneck. Indeed, Wang et al. inadvertently discovered as much in their exploration of transactional condition synchronization for the PARSEC benchmark suite [22]: in the "dedup" application, output by one pipeline stage eliminates all concurrency and scaling from an application whose lock-based equivalent scales well.

The one-size-fits-all nature of irrevocability is costly, but its simplicity is appealing: difficult tasks are no harder with irrevocable transactions than with locks. For example, irrevocability ensures low-level atomicity and durability of output: in applications with durability constraints, it is essential that programmers control the timing of calls to `fsync`, and the atomicity of an `fsync` call with respect to preceding `write` operations, and irrevocability affords this level of control.

Apart from irrevocability, the only other promising approaches to transactional output rely on deferred operations. Conventional wisdom suggests that many output operations, such as logging and error messages, can be achieved via deferred operations [6, 14, 15]. More formally, Volos et al. presented a general mechanism for deferring I/O in software transactions, via buffering and "shadow" file descriptors [20].

Unlike irrevocability, deferred output operations do not constrain concurrency. However, they suffer from two problems of their own. First, to ensure that deferral is correct, it is necessary to create an explicit copy of the data to output. This copy is in addition to any copying that occurs as part of a system call, and while it can be optimized in certain limited cases, it nonetheless introduces latency concerns. Furthermore, for hardware transactions, buffering may result in the working set of the transaction exceeding cache capacity. That, in turn, would lead to transactions serializing. The second problem with deferred output is that real programs often care about the return value of a `write` system call. When the `write` is delayed, it seems that the continuation of the transaction must either (a) ignore the return value, or (b) be scheduled after the output, as a second transaction that is not atomic with the first. We have identified situations, both in dedup and in MySQL, where deferral clouds reasoning about program correctness.

In this paper, we introduce Mimir, a methodology for delaying output that (a) remains atomic with respect to the calling transaction, (b) does not require copying, and (c) does not ignore return values. Mimir does not address *input* operations by transactions, for the simple reason that we have not been able to find any nontrivial example of transactional input in real-world code. Such a negative finding is unsurprising: blocking on input while holding a lock is a recipe for poor performance, and it is rare that programmers use a lock to make multiple input operations appear as one atomic operation. As a result, existing techniques (such as a dedicated input thread) appear to solve the input problem well.

The remainder of this paper is organized as follows. Section 2 briefly reviews key features of modern TM systems. Section 3 then introduces the key components of the Mimir methodology. The programming model for Mimir is discussed in Section 4. Section 5 discusses how Mimir can be used to implement the output patterns we have seen in dedup and MySQL. Section 6 concludes.

2. Background: Advanced TM Features

Before introducing the Mimir mechanism, it is useful to review some more advanced concepts from TM literature. We first discuss "privatization" and "publication", and then the state of the art in transactional condition synchronization.

```
1  transaction {
2    ...
3    data_is_private = true
4  }
5
6  use(data)
7
8  transaction {
9    data_is_private = false
10 }
```

Figure 1: Flag-based privatization in a C++ TM program.

2.1 Privatization and Publication

The privatization (and later publication) idioms were originally envisioned as a means of allowing safe non-transactional access to transactional data. The key issues relate to ordering: the programmer is responsible for making data "public" or "private" (e.g., by creating or removing all shared references to that data), and also for ensuring that threads agree on the state of a datum (transactional or non-transactional). In the TM implementation, the main challenge is ensuring that speculation and buffering do not result in races between transactional and non-transactional code [16, 17].

Early examples of publication and privatization include constructing an object and then creating a shared reference to it, and freeing an object after all shared references to it have been erased. In the context of software TM, these idioms avoid instrumentation, and in software and hardware TM systems, they also allow interaction with non-transactional parts of the system, such as custom allocators that make system calls. In early papers, it was assumed that an object's state (public or private; transactional or non-transactional) could be determined by the presence of references to the object in shared memory. Subsequent efforts [13] considered "flag-based" privatization and publication, in which shared references to an object are not created/destroyed, but the state of some auxiliary variable indicates whether transactions are allowed to use those references.

Consider the example in Figure 1. A transaction performs some operations on shared data, potentially including operations on `data` itself, on line 2. It then sets the `data_is_private` flag and commits. Henceforth, while other transactions can see references to `data`, they must obey the convention established by `data_is_private`, and not use those references. The privatizing transaction may use the data (line 6), and may even share `data` with other non-transactional threads (e.g., by using locks). Finally, when the non-transactional operations on `data` are complete, some thread executes the transaction on lines 8–10 to re-enable transactional access to `data`.

To support privatization and publication, a TM implementation must provide subtle ordering guarantees. When line 5 is reached, the calling thread must be certain that there are no pending cleanup operations by concurrent committed or aborted software transactions. Before the publishing transaction runs, all modifications to `data` must be complete, so that the publication of `data` does not precede `use` [13, 16, 17].

2.2 Transactional Condition Synchronization

Despite efforts to make condition variables safe for use within transactions [9, 22, 24], the most promising approaches to transactional condition synchronization appears to be based on scheduling. As Dalessandro et al. established [8], mechanisms like `retry` [11] and X10-style CCRs [7] are essentially scheduling mechanisms: they establish, dynamically, the conditions upon which a transaction that is scheduled to run will, in fact, be able to execute to completion.

The first and most general version scheduling-based technique for transactional condition synchronization is the `retry` mechanism. In this mechanism, a transaction T that determines that it cannot continue, due to some precondition on shared memory not holding, uses `retry` to undo its effects, and request that it not be scheduled again until some subsequent transaction performs a write to a location that had been read by T's unsuccessful attempt.

The `await` mechanism [6] simplifies `retry` by taking the parameter of a memory location. Like `retry`, a call to `await` undoes the transaction's effects. Re-execution of the transaction does not occur until some subsequent transaction performs a write to the location indicated by the parameter to `await`. This simplification avoids some spurious wake-ups. By bounding the number of locations upon which a transaction can be delayed, it also affords a more realistic path to implementation in hardware TM.

The `reschedule` mechanism [21] generalizes `await` by allowing the programmer to specify a predicate over shared memory that must be established before the transaction should be reattempted. Whereas such a predicate is implicit in `retry`, it must be provided by the programmer with `reschedule`. The programming model for `reschedule` is more precise than `retry` with regard to spurious wake-ups, but more error-prone than `retry` in the case where composition of transactions leads to complex predicates. More importantly, `reschedule` is built upon a mechanism that supports `retry`, `await`, and `reschedule` in hardware, software, and hybrid TM systems.

3. The Mimir Mechanism

The key requirement of Mimir is that programmers must embed an output operation, and any operations that depend on the return value of that operation, in a method of the object that encapsulates the data being output. In Section 5, we will give an example of how this requirement can be met in practice.

For programs that can be rewritten accordingly, Mimir will seamlessly privatize the object, execute the method non-transactionally, and then publish the object again. Since the object is privatized, and the output is not transactional, there is no need for global serialization/irrevocability. For transactions that require access to the privatized object, Mimir can employ any of the aforementioned condition synchronization mechanisms to delay execution until the object is published. In this section, we explain how the behavior is achieved, and we argue its correctness.

3.1 Privatization Can Be Two-Phase Locking

For all intents and purposes, the `data_is_private` flag in Figure 1 is a lock. When it is true, all threads other than the caller are forbidden from accessing `data`. When it is false, accesses to `data` are not constrained. Furthermore, if we ignore the discussion about unrolling effects when an exception escapes the boundaries of a lexically scoped transaction [1], the history created by successful transactions in C++ will result in an execution equivalent to one in which all transactions are protected by a single global lock.

Based on these observations, the code in Figure 1 is *almost* two-phase locking. On line 1, the implicit global transactional lock is acquired. On line 3, the lock protecting `data` is acquired. On line 4, the global lock is released. Thus lines 1–4 comprise a correct sequence where all locks are acquired before any are released. The complication is that line 8 appears to re-acquire the global lock before the lock protecting `data` is released on line 9. Such a re-acquisition violates the requirement that no locks are acquired after any locks are released.

There are two conditions that allow us to claim that Figure 1 nonetheless adheres to strict two-phase locking. The first is that

```
1  class class_name _Mimir_
2    // next field and method are injected
3    // by the _Mimir_ annotation
4    mimir_lock _Mimir_LOCK_
5
6    void _Mimir_CHECK_()
7      transaction
8        if _Mimir_LOCK_ != UNHELD
9          && _Mimir_LOCK_.owner != ME
10         // select a condition sync technique
11         reschedule(_Mimir_LOCK_ == UNHELD)
12         await(_Mimir_LOCK_)
13         retry()
14
15   // other fields of the class are unchanged
16   ...
17
18   // _Mimir_CHECK_ is injected into every
19   // method, as in this example:
20   ret_type method_name(...)
21     _Mimir_CHECK_()
22     // original method body follows
```

Figure 2: Mimir instrumentation of a C++ class. The `_Mimir_` annotation on the class body leads to the addition of one fields and one method, and a modification to every method in the original class. Lines 11–13 illustrate three options for condition synchronization.

the transaction on lines 8–10 does not access any program data. The second is that `data_is_private` serves only to synchronize access to other data. Thus it can be thought of as a synchronization variable [3]. We argue that some transactions can serve as an implementation technique for implementing synchronization mechanisms, and that other transactions are, themselves, a synchronization mechanism. We also argue that, under both flat and closed nesting, the addition of arbitrary transactions covering contiguous subsets of the operations within a larger transaction has no effect on the program. Thus lines 1–4 are equivalent to an implementation in which line 3 is wrapped by a nested transaction. In such a setting, transactions protecting `data_is_private` are an implementation technique, and need not be thought of as acquiring and releasing the implicit global lock.

3.2 Mimir Objects

The first aspect of Mimir is the `_Mimir_` annotation on classes. This annotation indicates that every instance of the annotated class has an implicit reentrant lock. We discuss the implementation and use of this lock below.

Figure 2 presents the instrumentation to a `_Mimir_` object. There are three modifications. First, a reentrant lock is added to the object. The lock is implemented using transactions, and it consists of an owner field and a counter. The `_Mimir_CHECK_` method uses a transaction to ensure that the lock is not held by some other thread, and it is called as the first operation of every method of the class. Thus whenever a method is called from a non-transactional context, `_Mimir_CHECK_` will suspend the caller at the point of the method call if the lock is held by a thread other than the caller. When any method is called from within a transaction, the same check is performed. However, if the lock is held by a thread other than the caller, `_Mimir_CHECK_` aborts the calling transaction and does not attempt it again until the lock is released. The mechanism for doing so (`retry`, `await`, `reschedule` does not affect correctness, but may result in spurious wake-ups.

```
ret_type method_name(...)
  ...

  // defer some output and recovery
  _Mimir_(this, [](){
    int res = write(this.field, ...)
    if (res == OK)
      fsync()
    else
      this.recover(...)
  })

  ...
```

Figure 3: Deferring an operation via `__MIMIR__`.

In this manner, a Mimir object renders itself unusable by any thread other than the `_Mimir_LOCK_`-holder, in a manner that is compatible with both transactional and non-transactional code. When the `_Mimir_LOCK_` is not held, the object may be used concurrently by multiple transactions, and even by transactional and non-transactional threads.

3.3 The Mimir Keyword

Thus far, we have not presented a mechanism for acquiring the `_Mimir_LOCK_`. As a lock that is not programmer-visible, acquisition and release are performed by the TM runtime system, in response to uses of the `_Mimir_` keyword. Figure 3 presents an example of how a programmer can defer an operation.

The `_Mimir_` keyword takes two parameters: an object and a function. It acquires the object's `_Mimir_LOCK_`, and schedules the function for execution after the outermost transactional scope commits. When the function completes, the `_Mimir_LOCK_` is automatically released.[1] Note that when `_Mimir_` is encountered in a non-transactional context, the lock will be acquired and the function executed immediately. Figure 4 presents the modifications to the TM library that are required in order to implement the `_Mimir_` keyword.

In the implementation, we must ensure that Mimir operations occur after the transaction's writes have been finalized (lines 26-28). This is necessary, as (a) in hardware TM, the Mimir operations may perform I/O, and must occur outside of a hardware TM context, and (b) in software TM, we must be sure that any writes to the Mimir object have been finalized before the un-instrumented function is executed. This includes those that could lead to privatization errors, necessitating that quiescence [23] precedes Mimir operations. Note that all Mimir locks are acquired before the transaction commits, and released after the transaction commits, ensuring that two-phase locking is obeyed. Since the locks are acquired via a transaction, they can be acquired in any order, without introducing the possibility of deadlock. Note that the release of the Mimir lock (line 28) must use the TM subsystem, so that the lock release is both (a) free of races with concurrent transactions reading the state of the Mimir lock, and (b) visible to the TM system, so that waiting transactions can be rescheduled.

3.4 Compatibility with TM Implementations

There are two requirements for using Mimir in a TM implementation, both of which are easy to satisfy. The first is that the TM must

[1] Note that in the case of reentrant locks, multiple uses of `_Mimir_` may mean that the lock is not released immediately after a specific function completes.

```
// new per-thread metadata
set<object, function> mimir_tasks

// implementation of the Mimir keyword
void _Mimir_(object, function)
  // acquire the _Mimir_LOCK_
  transaction
    _Mimir_CHECK_()
    object._Mimir_LOCK_.acquire()
  // in software TM, next line is
  // *NOT* instrumented
  mimir_tasks.insert(object, function)

// modifications to commit
void TM_COMMIT()
  ...
  // transaction is validated
  // and all writes are finalized
  // TM-related locks are released


  // ensure privatization-safety
  quiesce()

  // execute Mimir tasks
  for o, f in mimir_tasks
    f()
    o.release()

  // deferred allocator operations
  for f in delayed_frees
    free(f)
```

Figure 4: Deferring an operation via `__MIMIR__`.

support deferred operations. In the case of software TM, rudimentary support for deferred operations is available in all known implementations, in order to handle memory management. The GCC TM implementation [10] provides more robust support for deferred operations in software TM, and extending this support to hardware TM is straightforward (see, for example, [22]).

The second requirement is that the TM must be able to abort and delay re-execution of transactions. The three mechanisms we have highlighted in this paper (`retry`, `await`, and `reschedule`) are all compatible with both hardware and software TM, via lightweight runtime support [21]. One could imagine a streamlined hardware TM implementation that did not support any of these mechanisms; in such a case, it would suffice to abort the transaction, wait briefly, and try again.

4. Programming with Mimir

Mimir simplifies the task of delaying output operations, by ensuring that a deferred operation is atomic, via two-phase locking. Since the deferred operation is not executed within a transactional context, it is guaranteed not to unwind and retry, and thus can safely perform output. The operation holds a lock over the object, and thus can access any and all fields of the object, without risk of racy accesses by other threads. Most importantly, the deferred operation can entail arbitrary system calls that happen immediately. This simplifies error handling.

On the other hand, when an operation is deferred via Mimir, it is not statically checked, and thus it is possible to violate two-phase locking, e.g., by executing a transaction (on program data, not on synchronization objects) within the deferred operation. In this section, we describe the constraints on deferred operations, and the consequences of violating those constraints.

4.1 Memory Accesses

Clearly, it is safe for a deferred operation to access thread-private values, as well as the fields of the object whose Mimir lock is held. However, the programmer must keep in mind that the state of the object and thread-private data at the time when the `_Mimir_` keyword appears is not immutable, and may change in the suffix of the transaction that executes before the deferred operation.

The deferred operation does not execute transactionally (though it is part of the transaction), and thus accesses to shared data are not protected by any synchronization mechanism. The programmer must take care! To acquire locks or use transactions in order to access shared data will violate the two-phase locking guarantees of the deferred operation. While it is possible to maintain isolation and atomicity while adding additional synchronization, doing so will require whole-program analysis and should be avoided.

One exception is when a deferred operation on object O_1 accesses the state of O_2, where there is also a deferred operation on O_2 by the same transaction, which has not been executed yet. In that case, it is safe to access O_2, because the transaction still holds the Mimir lock on O_2. Programmers may assume that deferred operations are performed in the order they are registered through the `_Mimir_` keyword. Note that the use of a recursive lock ensures that multiple deferred operations on the same object will appear as a single indivisible operation.

4.2 System Calls

Deferred operations do not execute within a transactional context, and are thus able to perform system calls. However, it is the responsibility of the programmer to ensure that the system call does not cause a race. 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 the `_Mimir_` object. If B is shared, then it, too, should be a field of the `_Mimir_` object. Otherwise, accesses to B by the call to `write` could race with concurrent modifications to B. If the `_Mimir_` object cannot encapsulate both B and F, and both are shared, then buffering of B may be necessary.

System calls made within a deferred operation happen immediately. This enables the programmer to perform a write, verify its correctness (via the return value), and perform an `fsync` to ensure the write reaches its destination. Ensuring durability in this manner is essential in real-world programs, such as databases, and we see it as a critical feature of Mimir. The main constraint on system calls is that the utility of asynchronous output is not clear: since a lock is held, it does not make sense to perform an asynchronous output and then wait on the result; however, if an asynchronous output is requested and the result (which may arrive via signal) is not checked until the operation returns, then any recovery will not be atomic with the transaction.

5. Examples

In this section, we present examples to show how Mimir can be used to perform output from within transactions. The examples cover a set of common and interesting 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 [15] and MySQL, we observed that critical sections occasionally perform logging operations, such as error messages and diagnostic writes to per-thread logs. The

```
// x is a mutable string
// i is a mutable integer
const char *f = "...%s...[%d]..."
fprintf(stderr, f, x, i)
free(x) // optional

// Mimir variant
class defer_fprintf _Mimir_
  void defer_out(...)
    char * out = sprintf(f, x, i)
    _Mimir_([](){
            fprintf(stderr, out)
            free(out)
            free(this)
    })
```

Figure 5: Example of simple logging from within a critical section.

```
1  // for each fd, create one of these
2  // objects and make it global
3  class global_defer_fd _Mimir_
4    // the file descriptor is
5    // encapsulated as a field
6    // of this object
7    int fd
8
9    // constructor
10   global_defer_fd(...)
11     fd = open(...)
12
13   // output to a file that may
14   // not be reliable
15   void defer_out(buf, len)
16     _Mimir_([](){
17       char *p = buf
18       size_t nsent = 0
19       ssize_t rv
20       while (nsent < len)
21         rv = write(sd, p,
22                    len - nsent);
23         if (0 > rv &&
24             (errno == TRANSIENT_ERR)
25           continue
26         if (0 > rv)
27           error()
28         nsent += rv
29         p += rv
30       fsync(fd)
31       free(buf)
32     })
```

Figure 6: Example of ordered, reliable output to a file.

program does not expect any ordering among logging operations: they are diagnostic, and any ordering can be determined post-mortem. The return values of the output operations are typically not used. An example appears in Figure 5

In lock-based programs, we can assume that the critical section surrounding the program guarantees the immutability of both `x` and `i`. If `fprintf` were safe to call from within transactions, we would not have a concern, either: its implementation typically calls `sprintf` to construct a private string containing the final output stream, and then passes a pointer to that string to the `write` system call. However, if we wished to use simple deferral, we would have to (a) manually create the buffered string, (b) defer the output of that string, and (c) reclaim the memory for the string after the transaction commits. With Mimir, we can encapsulate all of this behavior in an object that constructs the string before invoking a Mimir operation, and then frees the string (and destroys itself) at the end of the deferred function. Note, too, that in the case of fixed output messages (such as those in assertions), the `_Mimir_` object can be stateless.

5.2 Ordered Output to a File

While the example above could easily be encapsulated in a one-off transactional `fprintf` function, it provides the basis for our argument that Mimir generalizes. To demonstrate the claim, we first provide a simple extension to the above example, and then describe how that extension manifests in the dedup kernel from the PARSEC benchmark suite [5].

Consider the case where critical sections perform writes to the same file, and the order in which the writes occur must be the same as the order in which the critical sections execute. Performing synchronous writes within the critical sections suffices to achieve the needed guarantee, but results in serialization if we replace the lock with transactions. This approach is particularly valuable when durability requirements necessitate the use of `fsync` within the critical section. Regardless of the presence of `fsync`, our simple logging example from above will fail: since each deferred output can use a distinct `_Mimir_` object, writes to the file can reorder after transaction commit.

To use Mimir in this case, we require only a small change to Figure 5, wherein we cease to create a new `_Mimir_` object each time we wish to perform output, and instead we wrap each file descriptor in a `_Mimir_` object. To make the example more concrete, Figure 6 presents code patterned after PARSEC dedup, which handles "short counts" due to transient errors (i.e., `EINTR` and `EAGAIN`) during the output. We assume that the original code freed the buffer after output, and thus have included line 31.

Since all invocations of the Mimir keyword that use the same object serialize on the object's `_Mimir_LOCK_`, concurrent writes to the file will be ordered identically to the order of their respective transactions. Thus we achieve the desired ordering. Introducing an `fsync` call is trivial (line 30), and is left up to the programmer. More significantly, the example moves beyond deferring a simple library call. The uses of `write` in real programs are varied, and any program wishing to perform low-level output (especially output over a network socket) must have some mechanism for dealing with transient write errors. Rather than create new high-level abstractions for reliable writes over unreliable streams, Mimir allows fine-grained use of return values after each system call. Doing so is straightforward, because the entire encapsulated object is private to the thread executing the `defer_out`.

In the case where the buffer is (a) not freed after the output, and (b) mutable during the I/O, it is desirable to avoid copying the buffer prior to line 16. One mechanism for doing so is to make the buffer object a `_Mimir_` object, and defer an empty operation on the buffer immediately after the transaction calls `defer_out`. Doing so would lock the buffer, so that it is immutable during the output operation. This optimization follows naturally from the requirement to perform Mimir operations in program order.

5.3 Pipelines with Output Stages

The example in Figure 6 is, surprisingly, more complex than required by PARSEC dedup. The dedup benchmark implements a

```
1  ...
2  if (chunk->header.state == COMPRESSED)
3    // write the data
4    write_file(fd, TYPE_COMPRESS, chunk->size,
5               chunk->dataptr)
6    mbuffer_free(&chunk->compressed_data)
7    chunk->header.state = CHUNK_STATE_FLUSHED
8  else
9    // just write SHA1
10   write_file(fd, TYPE_FINGERPRINT, SHA1_LEN,
11              chunk->sha1)
12 ...
```

Figure 7: The output pipeline stage in Dedup.

pipeline, in which output is performed in a serial stage. Transactional versions of dedup serialize all transactions whenever the thread executing the output stage attempts to write to the output file [22]. The output operation itself is similar to lines 17–29 of Figure 6, and in Figure 7, we refer to that sequence as `write_file`.

In this code sequence, program logic naturally encapsulates the file descriptor, since only the thread assigned to the output stage is allowed to perform I/O. However, there is program logic that must execute after the I/O, but isolated with respect to subsequent pipeline stages (lines 6–7). By encapsulating `chunk` in a `_Mimir_` object, we can move lines 4–7 into a deferred operation, thereby avoiding copying and allowing the buffer to be freed before any other thread can observe that the object's output has completed.

5.4 Opening Files as Output

Our final example of how Mimir can simplify transactional output comes from the MySQL InnoDB storage engine, and appears in Figure 8. All file output in InnoDB is achieved via asynchronous I/O, and is constructed carefully, so that the offset and length of each output operation is known in advance. Consequently, writes to files are not performed while any locks are held.

To compute offsets and lengths, InnoDB maintains a pool of open file descriptors, and tracks metadata on a per-descriptor basis. All accesses and modifications to the pool are performed while a lock is held. Most significantly, while the lock is held, files can be opened and closed.

Opening and closing file descriptors is a form of output: upon close, data may be flushed to disk; when opening, a file may be created or truncated. In InnoDB, multiple `open` and `close` operations can be composed into a single atomic operation. How many operations are attempted can depend on the result of each attempt (e.g., if a file cannot be closed due to pending writes, then another file may be closed instead).

Clearly it is not acceptable to delay opening a file, if some other thread may attempt to write to that file. We must ensure that while calls to `open` are completing, no concurrent threads can access the corresponding portions of the descriptor pool. At the same time, when there are no pending `open` or `close` operations, we would like to enable concurrent accesses to different descriptors within the pool.

Wrapping the entire pool as a `_Mimir_` object achieves our requirements, in a manner that is completely invisible to the programmer. The methods in Figure 8 each defer their entire body via the `_Mimir_` keyword. All other methods (not shown) do not use the keyword, and they run immediately when called from a transaction (this is safe, since they only modify the memory state associated with file descriptors). When any `open` is in flight, all concurrent accesses to the pool will be de-scheduled (via a `retry`-like mechanism). During that period, the pool itself will be privatized, such that non-transactional code may modify the pool (e.g., by opening and closing files, and updating their state) while remaining atomic and isolated with respect to all concurrent threads. Upon completion, concurrent operations on the pool will automatically wake and retry.

```
mySQL_initialize(...)
    // open logs & tablespace data files
    ...
    for (space in space_list)
        for (node in space->chain)
            node->handle = open(...);
    ...

mySQL_destroy(...)
    // close logs & tablespace data files
    ...
    for (space in space_list)
        for(node in space->chain)
            close(node->handle);
    ...

mySQL_io_prepare(...)
    // check system states
    ...

    // many open files... close some?
  close_more:

    // select qualified file
    ...
    if (close(file) == -1)
        exit_transaction;
    n_open--;
    if (n_open >= max_n_open)
        goto close_more;

    // check the node to do i/o
    ...
    if (node->open == FALSE)
        // to get file size, do an
        // open and close
        // save metadata for future i/o
        if (node->size == 0)
            node->handle = open();
            offset = lseek(file,0,SEEK_END);
            success = pread(two pages);
            close(node->handle);
        // already know the metadata
        node->handle = open();
    // change system states
    ...
```

Figure 8: MySQL critical sections related to managing a pool of open file descriptors that are used in asynchronous I/O.

6. Conclusions and Future Work

In this paper, we introduced the Mimir methodology for enabling programmers to perform output operations in a manner that remains atomic with respect to memory transactions. Mimir makes heavy use of implicit deferred privatization and `retry`-based condition synchronization, and may be the first concrete and comprehensive motivation for adding `retry` to the C++ TM specification.

Mimir requires the programmer to employ object-oriented design in order to encapsulate the file descriptors and data required for the I/O at a granularity that allows for Mimir to then seamlessly implement a two-phase locking protocol to privatize the object. The output is then performed after the transaction completes, but without risking interleavings by concurrent transactions accessing the object. In our example usage scenarios, we show how Mimir can be used for increasingly complex tasks, such as logging, ordered output, pipelined output, and management of pools of file descriptors. These examples were taken from memcached, PARSEC dedup, and MySQL's InnoDB engine, and support our assertion that Mimir is a practical approach for output in real-world programs.

As immediate future work, we plan to implement Mimir in the GCC-TM system. We are particularly interested in performance differences between eager software TM, lazy software TM, and hardware TM, especially as they relate to transactional acquisition of multiple Mimir locks: While any TM system should ensure this acquisition is deadlock free, regardless of order, we worry that livelock may occur in practice, for hardware and eager software TM. We are also interested in the impact on program complexity that arises when refactoring code to encapsulate related fields in a single Mimir object.

Longer-term, we plan to explore the performance of Mimir, and the degree to which it can be generalized beyond output. In the former category, we will require a broad application study, particularly since some of the examples that motivated this work (e.g., memcached) do not perform output on any critical path. On the other hand, any serialization-free mechanism to perform output in dedup will dramatically improve performance, and we must be cautious not to equate the removal of a pathology with a generalizable success. In the latter category, our analysis of MySQL shows an exciting road forward: `write` system calls are not the only unsafe operations that can be cast as output, and delayed via Mimir. Other possibilities include arbitrary system calls, and we plan to use the taxonomies proposed by Volos et al. [20] and Baugh and Zilles [4] to identify the types of operating system interactions most amenable to deferral via Mimir.

Acknowledgments

We thank Victor Luchangco and Jason Flinn for their advice and guidance during the conduct of this research. This material is based upon work supported by the National Science Foundation under Grants CAREER-1253362 and CCF-1218530. 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] A.-R. Adl-Tabatabai, V. Luchangco, V. J. Marathe, M. Moir, R. Narayanaswamy, Y. Ni, D. Nussbaum, X. Tian, A. Welc, and P. Wu. Exceptions and Transactions in C++. In *Proceedings of the First USENIX Workshop on Hot Topics in Parallelism*, Berkeley, CA, Mar. 2009.

[2] A.-R. Adl-Tabatabai, T. Shpeisman, and J. Gottschlich. Draft Specification of Transactional Language Constructs for C++, Feb. 2012. Version 1.1, http://justingottschlich.com/tm-specification-for-c-v-1-1/.

[3] S. V. Adve and K. Gharachorloo. Shared memory consistency models: A tutorial. *Computer*, 29(12):66–76, 1996.

[4] L. Baugh and C. Zilles. An Analysis of I/O and Syscalls in Critical Sections and Their Implications for Transactional Memory. In *Proceedings of the 2nd ACM SIGPLAN Workshop on Transactional Computing*, Portland, OR, Aug. 2007.

[5] C. Bienia, S. Kumar, J. P. Singh, and K. Li. The PARSEC Benchmark Suite: Characterization and Architectural Implications. In *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*, Toronto, ON, Canada, Oct. 2008.

[6] B. D. Carlstrom, A. McDonald, H. Chafi, J. Chung, C. C. Minh, C. Kozyrakis, and K. Olukotun. The Atomos Transactional Programming Language. In *Proceedings of the 27th ACM Conference on Programming Language Design and Implementation*, June 2006.

[7] P. Charles, C. Donawa, K. Ebcioglu, C. Grothoff, A. Kielstra, C. von Praun, V. Saraswat, and V. Sarkar. X10: An Object-Oriented Approach to Non-Uniform Cluster Computing. In *Proceedings of the 20th ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications*, San Diego, CA, Oct. 2005.

[8] L. Dalessandro, M. Scott, and M. Spear. Transactions as the Foundation of a Memory Consistency Model. In *Proceedings of the 24th International Symposium on Distributed Computing*, Cambridge, MA, Sept. 2010.

[9] P. Dudnik and M. M. Swift. Condition Variables and Transactional Memory: Problem or Opportunity? In *Proceedings of the 4th ACM SIGPLAN Workshop on Transactional Computing*, Raleigh, NC, Feb. 2009.

[10] Free Software Foundation. Transactional Memory in GCC, 2012. http://gcc.gnu.org/wiki/TransactionalMemory.

[11] T. Harris, S. Marlow, S. Peyton Jones, and M. Herlihy. Composable Memory Transactions. In *Proceedings of the 10th ACM Symposium on Principles and Practice of Parallel Programming*, Chicago, IL, June 2005.

[12] M. P. Herlihy and J. E. B. Moss. Transactional Memory: Architectural Support for Lock-Free Data Structures. In *Proceedings of the 20th International Symposium on Computer Architecture*, San Diego, CA, May 1993.

[13] V. Menon, S. Balensiefer, T. Shpeisman, A.-R. Adl-Tabatabai, R. Hudson, B. Saha, and A. Welc. Practical Weak-Atomicity Semantics for Java STM. In *Proceedings of the 20th ACM Symposium on Parallelism in Algorithms and Architectures*, Munich, Germany, June 2008.

[14] Y. Ni, A. Welc, A.-R. Adl-Tabatabai, M. Bach, S. Berkowits, J. Cownie, R. Geva, S. Kozhukow, R. Narayanaswamy, J. Olivier, S. Preis, B. Saha, A. Tal, and X. Tian. Design and Implementation of Transactional Constructs for C/C++. In *Proceedings of the 23rd ACM Conference on Object Oriented Programming, Systems, Languages, and Applications*, Nashville, TN, USA, Oct. 2008.

[15] W. Ruan, T. Vyas, Y. Liu, and M. Spear. Transactionalizing Legacy Code: An Experience Report Using GCC and Memcached. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, Salt Lake City, UT, Mar. 2014.

[16] M. Spear, L. Dalessandro, V. J. Marathe, and M. L. Scott. Ordering-Based Semantics for Software Transactional Memory. In *Proceedings of the 12th International Conference On Principles Of DIstributed Systems*, Luxor, Egypt, Dec. 2008.

[17] M. Spear, V. Marathe, L. Dalessandro, and M. Scott. Privatization Techniques for Software Transactional Memory (POSTER). In *Proceedings of the 26th ACM Symposium on Principles of Distributed Computing*, Portland, OR, Aug. 2007.

[18] M. Spear, M. M. Michael, and M. L. Scott. Inevitability Mechanisms for Software Transactional Memory. In *Proceedings of the 3rd ACM SIGPLAN Workshop on Transactional Computing*, Salt Lake City, UT, Feb. 2008.

[19] M. Spear, M. Silverman, L. Dalessandro, M. M. Michael, and M. L. Scott. Implementing and Exploiting Inevitability in Software Transactional Memory. In *Proceedings of the 37th International Conference on Parallel Processing*, Portland, OR, Sept. 2008.

[20] H. Volos, A. J. Tack, N. Goyal, M. Swift, and A. Welc. xCalls: Safe I/O in Memory Transactions. In *Proceedings of the EuroSys2009 Conference*, Nuremberg, Germany, Mar. 2009.

[21] C. Wang, Y. Liu, and M. Spear. A New API for Transactional Condition Synchronization. In *Proceedings of the 6th Workshop on the Theory of Transactional Memory*, Paris, France, July 2014.

[22] C. Wang, Y. Liu, and M. Spear. Transaction-Friendly Condition Variables. In *Proceedings of the 26th ACM Symposium on Parallelism in Algorithms and Architectures*, Prague, Czech Republic, June 2014.

[23] A. Welc, B. Saha, and A.-R. Adl-Tabatabai. Irrevocable Transactions and their Applications. In *Proceedings of the 20th ACM Symposium on Parallelism in Algorithms and Architectures*, Munich, Germany, June 2008.

[24] R. Yoo, C. Hughes, K. Lai, and R. Rajwar. Performance Evaluation of Intel Transactional Synchronization Extensions for High Performance Computing. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, Denver, CO, Nov. 2013.