Shifting the Parallel Programming Paradigm

Justin E. Gottschlich†, Dwight Y. Winkler§, Mark W. Holmes‡, Jeremy G. Siek†, Manish Vachharajani†

†Department of Electrical and Computer Engineering, University of Colorado at Boulder
§Nodeka, LLC.
‡Raytheon Company

†{gottschl, jeremy.siek, manishv}@colorado.edu, §dwight@nodeka.com,
‡mark_w_holmes@raytheon.com

Abstract

Multicore computer architectures are now the standard for desktop computers, high-end servers and personal laptops. Due to the multicore shift in computer architecture, software engineers must write multithreaded programs to harness the resources of these parallel machines. Unfortunately, today's parallel programming techniques are difficult to reason about, highly error-prone and challenging to maintain for large-scale software.

This paper presents an emerging parallel programming abstraction called transactional memory (TM) and its benefits over mutual exclusion, the most commonly used parallel programming abstraction. TM simplifies the parallel programming model by moving the complexity of shared memory correctness away from the programmer's view. TM also shortens the software lifecycle by simplifying software design, abbreviating software test and reducing software development. We demonstrate the importance of TM by comparing it against mutual exclusion in terms of (1) simplicity, (2) scalability and (3) modularity.

1. Introduction

Software programs are divided into two fundamental categories regarding instruction execution. Programs can either execute instructions sequentially or parallelly. Sequential programs execute one instruction at a time, while parallel programs execute multiple instructions simultaneously. [1] Sequential programs tend to be less complex than parallel programs, but cannot fully harness the resources of machines that support multiple, simultaneous process execution. Parallel programs are generally more complex, but can more fully utilize machines that support the simultaneous execution of more than one process.

Parallel programs are written for convenience or performance [5, 23, 31]. Convenience-driven parallel programs are created because certain problems are naturally stand-alone and, as such, are easier to implement as a separate unit. By separating these stand-alone units away from the main application, they become easier to reason about and result in reduced program logic [5,23,31]. Examples of such convenience problems are those that naturally block, such as network socket processing and user-interface interaction, which, if solved sequentially, would require a complex polling model to be built within the main program. Solving such blocking problems in a separate, parallel thread reduces overall program logic as the programmer is no longer required to create a complex polling loop nor is he or she required to make polling interval estimates which can be a source of errors.

The other type of parallel program, created for performance, can be developed using various methods. One way to develop performance-driven parallel programs is to use multiple threads to execute instructions simultaneously. These parallel programs, called *multithreaded* programs, exploit task parallelism [22], also known as thread-level parallelism (TLP) [16]. Unlike instruction-level parallelism (ILP), where compilers and processors automatically identify and execute parallel instructions, task parallelism requires programmers to manually specify workloads that can be run concurrently. [2]

For this paper, we are concerned with parallel programs written for both performance and convenience. As we will demonstrate, these goals are not mutually exclusive and can be simultaneously achieved with transactional memory (TM). We are particularly concerned with shifting the parallel programming paradigm from the mutual exclusion paradigm, which is difficult to use, does not scale, and fails to support software modularity, to the TM paradigm, which is easier to use, is scalable and supports software modularity through transactional composition. This paper presents the following technical contributions:

[1] In fact, instruction level parallelism exploited by out-of-order microarchitectures can execute multiple instructions of a sequential program simultaneously [16].

[2] Speculative Multithreading, also known as Thread-Level Speculation, achieves TLP without requiring the programmer to specify parallel workloads [30].

- We demonstrate that TM's simplicity removes errors in concurrent interleavings, removing the potential for race conditions and, therefore, reducing software test.
- We show that TM's truly optimistic critical sections result in wider concurrent critical section throughput than is achievable in any possible fine-grained locking solution.
- We illustrate that TM supports software modularity through transactional nesting. The software modularity supported by TM ensures newly derived transactional modules, composed of existing transactional modules, are atomic, consistent, and isolated.

2. Preliminary Definitions of Concurrency

Multithreaded programs achieve task parallelism by using threads to simultaneously perform programmer-divided portions of the program's workload. At programmer-specified points, threads transmit the results of their work through data that are visible to all threads of the program. Such data are referred to as *concurrent objects* or *shared data* because all threads within the multithreaded program have read and write access to these objects [17, 20]. We use the terms concurrent objects and shared data interchangeably.

For multithreaded programs to behave correctly, shared data must be accessed in a manner that is consistent. The process of ensuring consistent access to shared data is called *concurrency control* [3]. There are various types of concurrency control, such as mutual exclusion and transactional memory. All forms of concurrency control use *synchronization mechanisms* to serialize access to concurrent objects in order to ensure data consistency [4, 19, 20].

The region of code that synchronization mechanisms protect are called *critical sections*. The manner in which critical sections are executed determines the maximum concurrent throughput possible for a multithreaded program (details to follow in Section 4). An ideal concurrency control type (1) provides easy-to-use synchronization mechanisms, (2) performs competitively when compared to other types of concurrency control and (3) supports software modularity. [3]

2.1 Mutual Exclusion

The most common type of concurrency control, called mutual exclusion, creates controlled code regions generally guarded by synchronization mechanisms called locks. Locks can only be held by one thread at a time, which guarantees a mutual exclusion's guarded code region, or *critical section*, is limited to a single thread of execution. Mutual exclusion locking is divided into two primary categories: (1) fine-grained locking and (2) coarse-grained locking. When fine-grained locking is used, programmers try to wrap the smallest possible unit of shared data with a single lock. By doing this, locks are only acquired when data they wrap is accessed, ultimately improving concurrent performance. Fine-grained locking performs well when compared to other synchronization mechanisms, but is notoriously difficult to develop, verify, and maintain. In contrast, coarse-grained locking works by using a single lock to guard multiple data. Coarse-grained locking is easy to implement and verify, yet performs substantially slower than its fine-grained locking counterpart. Mutual exclusion is therefore less than ideal since it can deliver performance or simplicity, but not both.

[3] This is a broad generalization, but is used to demonstrate that simplicity, performance, and modularity are important aspects to all synchronization mechanisms.

```
1  // coarse-grained          // fine-grained
2  lock(globalLock);          lock(mutexX);
3                             lock(mutexY);
4  ++x;                       ++x;
5  --y;                       --y;
6                             unlock(mutexX);
7  unlock(globalLock);        unlock(mutexY);
```

Figure 1. Coarse- and Fine-Grained Locking.

Figure 1 shows two mutual exclusion examples writing to the concurrent objects `x` and `y`. One of the examples uses coarse-grained locking (e.g., `globalLock`) and the other uses fine-grained locking (e.g., locks `mutexX` and `mutexY`). As shown in Figure 1, in order for a programmer to properly access `x` and `y`, the appropriate locks must be (sequentially) acquired to ensure program consistency. Unfortunately, these locks expose implementation details about the shared data `x` and `y` and therefore reduce software modularity.

2.2 Transactional Memory

Transactional memory is a modern type of concurrency control that uses *transactions* as its synchronization mechanism [22]. A transaction is a finite sequence of operations that are executed in an atomic, isolated and consistent manner [19, 20, 22]. The atomicity, isolation and consistency (ACI) of transaction's are derived from the ACID principle in the database community. TM does not exhibit the D (durability) of ACID because unlike database transactions, TM transactions are not saved to permanent storage (e.g., hard drives).

Transactions are executed speculatively (optimistically) and are checked for consistency at various points in the transaction's lifetime. Programmers specify the starting and ending points of a transaction. All of the operations between those points make up the transaction's execution body. Transactions are commonly represented using the `atomic` structure shown in Figure 2 [14].

```
1  atomic
2  {
3    ++x;
4    --y;
5  }
```

Figure 2. Simple Transaction Using the atomic Keyword.

Once a transaction has started it either commits or aborts [33]. A transaction's operations are only seen once the transaction

has committed, providing the illusion that all of the operations occurred at a single instance in time. The instructions of a committed transaction are viewed as if they occurred as an indivisible event, not as a set of operations executed serially. The operations of an aborted transaction are never seen by other threads, even if such operations were executed within a transaction and then rolled back [2].

In the case of Figure 2, if the transaction was committed both operations `++x` and `--y` would be made visible to other threads at the same instance in time. If the transaction in Figure 2 was aborted, neither operation (`++x` and `--y`) would appear to have occurred even if the local transaction executed one or both operations.

TM offers three distinct advantages over other parallel programming abstractions. First, TM is simple; transactions, the synchronization mechanism of TM, are easier to program than other synchronization mechanisms because they move shared memory management into the underlying TM subsystem, removing its complexity from the programmer's view. Moreover, TM exposes a simple programmer interface, reducing (or in some cases, removing) the potential for deadlock, livelock and priority inversion. Second, TM is scalable; it achieves increased computational throughput when compared to other parallel programming abstractions by allowing multiple threads to speculatively execute the same critical section. When concurrently executing threads do not exhibit shared data conflicts, they are guaranteed to make forward progress. Third, TM is modular; transactions can be nested to any depth and function as a single unit. This behavior allows application programmers to extend atomic library algorithms into atomic domain-specific algorithms without requiring the application programmers to understand the implementation details of the library algorithm. For these reasons, transactions are considered an important synchronization mechanism and TM is viewed as an important type of concurrency control. The remainder of this paper presents TM from a viewpoint of (1) simplicity, (2) scalability and (3) modularity.

3. Simplicity

Synchronization problems, such as deadlocks, livelocks and priority inversion are common in software systems using mutual exclusion. TM avoids many of these problems by providing a synchronization mechanism that does not expose any of its implementation details to the programmer. The only interfaces the programmer needs to use for TM is as follows:

- `begin_tx()` – the signaled start of the transaction.
- `read(loc)` – reads the specified memory location, storing its location in the transaction's read set and returning its current value.
- `write(loc, val)` - writes the specified memory location to the supplied `val`, storing its location in the transaction's write set.
- `end_tx()` – the signaled end of the transaction. `end_tx()` returns true if the transaction commits, otherwise it returns false.

The above interfaces allow the programmer to create a transaction (using `begin_tx()`), specify its memory operations (using `read()` and `write()`) and terminate (using `end_tx()()`). Moreover, none of the interfaces specify details of the TM subsystem's implementation. This leaves the TM system's implementation disjoint from the interfaces it supplies, a key characteristic for TM's simplicity.

All TM implementations use some combination of the above interfaces. TMs implemented within compilers tend to implicitly annotate transactional `read()` and `write()` operations, whereas those implemented within software libraries tend to require the programmer explicitly state which operations are transactional reads and writes [7, 25, 29]. An example of a transaction using the above interfaces alongside an actual STM library implementation (DracoSTM [7, 10]) is shown in Figure 3.

```
1  // TM interfaces              // DracoSTM
2  do {                         atomic(t)
3    begin_tx();                {
4    write(x, read(x)+1);         ++t.write(x);
5    write(y, read(y)-1);         --t.write(y);
6  while (end_tx());            } before_retry {}
```

Figure 3. Transaction Using (1) Explicit TM Interfaces and (2) DracosTM.

Figure 3 implements the same transaction as shown in Figure 2, except all transactional memory accesses, including the transaction's retry behavior (e.g., its loop), are demonstrated from a simple TM interface perspective and an actual library implementation (DracoSTM). While most TM systems handle some portion of these interface calls implicitly, as is shown in the DracoSTM transaction, it is important to note that even when all operations are made visible to the end programmer, transactions are still devoid of many concurrency problems, such as data races and deadlocks (explained below), that plague other types of concurrency control.

For example, as long as the programmer properly annotates the access to the shared variables `x` and `y` as shown in Figure 3, it is impossible for race conditions or deadlocks to occur. Furthermore, the programmer does not need any program-specific knowledge to use shared data; he or she simply uses the TM interfaces supplied by the system and the resulting behavior is guaranteed to be consistent. This is explained in greater detail in Section 3.1.

Other types of concurrency control, such as mutual exclusion, cannot achieve the same interface simplicity, because part of their implementation is associated with, or exposed through, their interface. To demonstrate this, consider the fine-grained locking example of Figure 1 as shown below.

```
1  // fine-grained locking
2  lock(mutexX);
```

```
3   lock(mutexY);
4   ++x;
5   --y;
6   unlock(mutexX);
7   unlock(mutexY);
```

There is no universal interface that can be used to properly access the shared data protected by the mutual exclusion in the above fine-grained locking example. Instead, the programmer must be aware that `mutexX` and `mutexY` protect shared data `x` and `y` and, therefore, the locks must be obtained before accessing the shared data. In short, the programmer is responsible for knowing not only that mutual exclusion is used, but also *how* it is used (e.g., which locks protect which shared variables). In this case, `mutexX` must be obtained before `mutexY`. If another section of code implements the following, a deadlock scenario will eventually occur.

```
1   // fine-grained locking
2   lock(mutexY);
3   lock(mutexX); // deadlock here
4   --y;
5   ++x;
6   unlock(mutexY);
7   unlock(mutexX);
```

3.1 Understanding Concurrency Hazards

Informally, a *concurrency hazard* is a condition existing in a set of operations that, if executed with a specific concurrent interleaving, results in one or many unwanted side-effects [31]. Most errors in parallel systems, such as deadlocks and priority inversion, are the specific execution of concurrency hazards resulting in the unwanted side-effect(s) they contain. If the concurrency hazards are eliminated, the parallel system errors contained within the concurrency hazards are also eliminated. Unfortunately, detecting existing concurrency hazards is non-trivial and therefore eliminating them is also non-trivial.

Mutual exclusion exhibits more concurrency hazards than TM because its implementation details (i.e., its locks) must be exposed and used by the end programmer. While the locks used to enforce mutual exclusion by themselves are not concurrency hazards, their use can lead to a number of hazards. As such, using locks leads to concurrency hazards.

Because the mutual exclusion locking details are exposed to the programmer and because the programmer must maintain a universal and informal contract to use these locks, concurrency hazards can arise due to the number of possible misuses that can be introduced by the programmer. In particular, if the programmer accidentally deviates from the informal locking contract, he or she may inadvertently introduce a concurrency hazard that can cause the program to deadlock, invert priority or lead to inconsistent data.

In contrast, TM has no universal or informal contract between shared data that the end programmer needs to understand and follow as is required in mutual exclusion. Due to this, TM can hide its implementation details which results in reduced concurrency hazards. In particular, each transaction tracks the memory it uses in its read and write sets. When a transaction begins its commit phase, it verifies its state is consistent and commits its changes. If a transaction finds its state is inconsistent, it discards its changes and restarts. All of this can be achieved using the basic TM interfaces shown in Section 3 without exposing any implementation details. In order to use TM, the end programmer only needs to know how to correctly create a transaction. Once the transaction is executed, regardless of how it is executed, it results in a program state that is guaranteed to be consistent.

Fundamentally, TM exhibits less concurrency hazards than mutual exclusion because its implementation details are divorced from its interface and can therefore be hidden within its subsystem. Any number of implementations can be used in a TM subsystem using only the basic TM interfaces shown in Section 3. The same is not true for mutual exclusion. Mutual exclusion, regardless of how it is implemented, exposes details of its implementation to the programmer. As demonstrated in Section 5, mutual exclusion does not provide software modularity specifically because extending an existing module requires an understanding and extension of that module's implementation. When such locking implementations are hidden inside of software libraries, extending these modules can range from difficult to impossible.

3.2 Testing: Race Conditions and Interleavings

A *race condition* is a common concurrency hazard that exists in parallel or distributed software. As with all concurrency hazards, race conditions rely on a specific interleaving of concurrent execution to cause their unwanted side-effect. In this section we demonstrate that race conditions do not exist in TM and therefore, software testing is greatly simplified because all possible interleavings do not need to be tested to ensure correct system behavior. In order to demonstrate that race conditions are absent from TM, we must first show that they are present in other types of concurrency control.

```
1   // Thread T1            // Thread T2
2                           lock(L2);
3                           lock(L1);
4                           ...
5                           unlock(L1);
6                           unlock(L2);
7   lock(L1);
8   lock(L2);
9   ...
```

Figure 4. Mutual Exclusion Race Condition.

Consider the race condition present in the mutual exclusion example shown in Figure 4. The race condition present in the example results in a deadlock if thread T_1 executes line 7 followed by thread T_2 executing line 2. However, if the program executes the lines in order (e.g., line 1, then line 2, then line 3, etc.), the system will execute properly. The fundamental problem in Figure 4 is that it contains a concurrency hazard; in particular, it contains a race condition. To further complicate matters, the race condition can only

be observed in two of many possible concurrent executions. Those two executions are: T_1 executes line 7 followed by T_2 executing line 2 or T_2 executes line 2 followed by T_1 executing line 7. All other possible concurrent interleavings of threads T_1 and T_2 avoid the deadlock race condition. More specifically, as long as T_1 executes lines 7-8 atomically or T_2 executes line 2-3 atomically, all remaining concurrent interleavings are free of the deadlock race condition.

Because it is unlikely that the deadlock race condition will occur, the programmer may never observe it, no matter how many times the program is tested. Only exhaustive testing, which tests all possible concurrent interleavings, is guaranteed to identify the presence of the deadlock. Regrettably, exhaustive testing is an unrealistic solution for most programs due to the time it would take to execute all possible concurrent interleavings of the program.

An alternative to exhaustive testing is for programmers to use types of concurrency control that are devoid of certain concurrency hazards. For example, if mutual exclusion did not emit the race condition concurrency hazard, it would be impossible for a program using it to deadlock. Therefore, exhaustive testing would not be necessary. While this scenario is hypothetical, it illustrates our larger argument: in order to avoid common parallel problems in a *practical* fashion, programmers may need to only use types of concurrency control that are devoid of certain concurrency hazards. By doing this, the program using the specific type of concurrency control will be guaranteed to be free of certain common parallel problems.

TMs are required to be devoid of race conditions within their implementations because they must enforce the ACI (atomic, consistent and isolated) principles. Transactions must execute as atomic and isolated and, therefore, TMs are not capable of supporting concurrent interleavings between multiple transactions as that would violate the atomic and isolated principles of ACI. Due to this, programs only using TM are *guaranteed* to be free of deadlocks (i.e., deadlock-freedom). Moreover, because TM implementations can guarantee freedom of race condition concurrency hazards, programmers only need to verify their transactional code is correct in a sequential (non-parallel) manner. Once the sequential execution of the transactional code has been verified, no more testing is required as the TM system is required to behave in a consistent manner for all serial orders.

3.3 Development: Mutual Exclusion and TM

The development of fine-grained locking is notoriously difficult [1, 19, 24]. Designing such software is equally as hard. The difficulty in developing and designing fine-grained locking systems is rooted in conflicting heuristics. A primary goal of software design is to identify the most simplistic software solution that exists for a particular problem. A primary goal of fine-grained locking is the most efficient concurrent implementation of a software system. The goals of software design and fine-grained locking are conflicting because the most efficient fine-grained locking solution usually requires some of the most complex software design implementations to achieve such performance [13].

TM achieves scalability by using optimistic concurrency that is implemented within its subsystem (see Section 4). Since the TM subsystem is the efficiency throttle for TM, which is unexposed to the programmer, the software architecture and design never needs to be complicated (nor can it be) in order to achieve increased parallelism when using transactions. As will be demonstrated in the following section, transactions run efficiently using the interfaces shown in this section and are never complicated in order to achieve improved performance, as is commonly found in fine-grained mutual exclusion implementations.

4. Scalability

In this section we analyze the scalability of TM compared to mutual exclusion. We measure scalability by two metrics: consistency and performance. A concurrency control type has *consistent scalability* if it guarantees correct behavior for an arbitrarily large number of concurrently executing processes.[4] *Performance scalability* is measured by the maximum number of consistent processes supported by a concurrency control type while executing concurrently.

4.1 Pessimistic and Optimistic Critical Sections

Critical sections can be pessimistic or optimistic. Pessimistic critical sections limit their critical section execution to a single thread. Locks are an example of a synchronization mechanism that use pessimistic critical sections. Optimistic critical sections allow unlimited concurrent threaded execution. Transactions are an example of a synchronization mechanism that use optimistic critical sections.

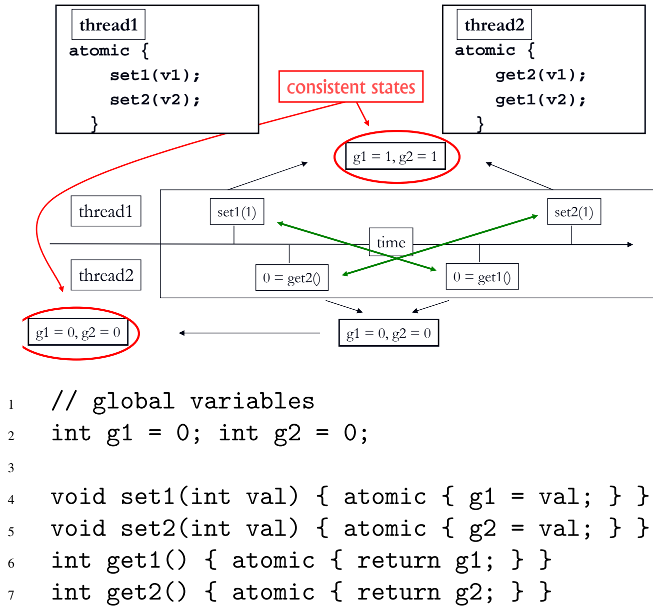
4.1.1 Truly Optimistic Critical Sections

Truly optimistic critical sections are those critical sections which allow multiple *conflicting* threads to simultaneously execute the same critical section. A deferred update (or lazy acquire) TM system supports truly optimistic critical section. A direct update (or eager acquire) TM system does not support truly optimistic critical sections. More details on deferred and direct update TM systems are presented in the subsequent sections.

Truly optimistic critical sections are important because they allow simultaneous conflicting critical section execution, as opposed to disallowing such behavior. It is important to allow conflicting critical section execution because prematurely preventing concurrently executing threads pessimistically degrades performance. To demonstrate this, consider two transactions, called T_1 and T_2, executing the same critical section. Transaction T_1 starts first and tentatively writes to memory location M. Transaction T_2 then starts and tries to write to memory location M. In a truly optimistic TM system, T_2 would be allowed to tentatively write to location M while T_1 is also writing to M. This behavior then allows T_2 to commit before T_1 in the event T_2 completes before T_1.

[4] Non-scalable synchronization methods include algorithms, such as Peterson's locking algorithm, that support the concurrent execution of a limited number of processes. Peterson's algorithm is non-scalable because it can only support two concurrently executing processes.

In comparison, if the TM system is not truly optimistic, once T_1 writes to M, T_2 must stall until T_1 completes. This pessimistically degrades the performance of the system by prematurely deciding that T_1's transactional execution should have higher priority than T_2's.




```
1  // global variables
2  int g1 = 0; int g2 = 0;
3
4  void set1(int val) { atomic { g1 = val; } }
5  void set2(int val) { atomic { g2 = val; } }
6  int get1() { atomic { return g1; } }
7  int get2() { atomic { return g2; } }
```

Figure 5. Truly Optimistic Concurrency Diagram.

Furthermore, and perhaps more importantly, truly optimistic critical sections allow readers and writers of the same memory location to execute concurrently. This behavior is important because in many cases, both the readers and writers of the same memory can commit with consistent views of memory.

An example of this is shown in Figure 5. As demonstrated in Figure 5 thread 1 and thread 2, which we'll refer to as T_1 and T_2 respectively, operate on the same memory locations (g1 and g2). Because the TM system supports optimistic concurrency, T_2 is allowed to execute concurrently alongside T_1 even though their memory accesses conflict. However, in this scenario, because T_2 completes its workload before T_1, both transactions are allowed to commit. T_2 captures the state of g1=0,g2=0 while T_1 sets the state of g1=1,g2=1. As the example addresses, both g1=0,g2=0 and g1=1,g2=1 are legal states.

4.2 Direct and Deferred Update

Updating is the process of committing transactional writes to global memory and is performed in either a *direct* or *deferred* manner [22]. Figure 6 presents a step-by-step analysis of direct and deferred updating.

Deferred update creates a local copy of global memory, performs modifications to the local copy, and then writes those changes to global memory if the transaction commits. If the transaction aborts, no additional work is done. Direct update makes an original backup copy of global memory and then writes directly to global memory. If the transaction commits, the transaction does nothing. If the transaction aborts, the transaction restores global memory with its backup copy. Some TM systems favor direct update due to its natural optimization of commits (BSTM [15], McRT-STM [32] and LogTM [26]). However, other TM systems favor deferred update due to its support for truly optimistic critical sections (DracoSTM and RingSTM) [6, 34].

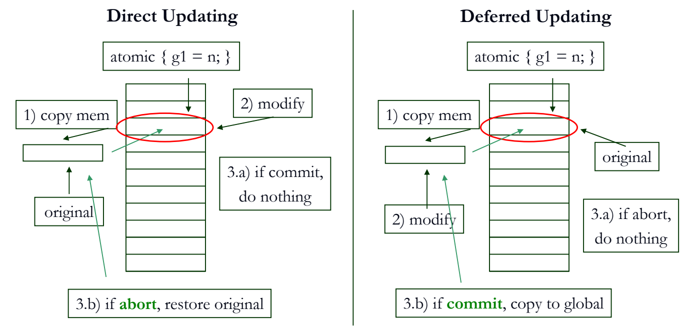


Figure 6. Direct and Deferred Updating.

Direct update enables greater TM throughput when aborts are relatively low because it optimizes the common commit case. Deferred update enables greater TM throughput when (1) aborts are relatively high or (2) short running transactions (e.g., those that complete quickly) are executed alongside long running transactions (e.g., those that do not complete quickly) because long running transactions do not stall shorter running ones as they would in direct update systems, and therefore the fastest transactions can commit first. It is important to note that deferred update supports truly optimistic critical sections without special effort, while direct update does not. Truly optimistic critical sections enable the speculative execution of transactions that arrive after a memory location has already been tentatively written to by another transaction. This allows the first transaction, of potentially many competing transactions, to complete its commit, whether it be the later arriving transaction or the earlier arriving writer. This scenario is not possible with direct update without special effort. [5]

4.3 Scalability: Mutual Exclusion and Transactional Memory

The scalability of mutual exclusion is limited to pessimistic concurrency. By definition, mutual exclusion's critical sections must be pessimistic, otherwise they would not be isolated to a single thread (i.e., they would not be mutually exclusive). TM, however, is generally implemented using optimistic concurrency, but it can enforce pessimistic concurrency amongst transactions if that behavior is required for certain conditions [9, 36, 37]. In certain cases, TMs become more strict and execute pessimistically to enable inevitable or irrevocable transactions. Such transactions have signifi-

[5] Direct update cannot support truly optimistic concurrency, but it can support single-writer, multiple-reader optimistic concurrency. However, enabling such behavior usually requires additional overhead since the original unaltered memory must be found in the in-flight transaction's memory.

cant importance for handling operations that, once started, must complete (e.g., I/O operations).

Since TM can execute optimistically and pessimistically, it is clear that whatever benefits pessimistic concurrency has can be acquired by TM. However, since mutual exclusion can only execute pessimistically, the advantages found in optimistic concurrency can never be obtained by mutual exclusion.

When one first analyzes pessimistic and optimistic concurrency, it may seem that the only benefit optimistic concurrency has over pessimistic concurrency is that multiple critical sections, which conflict on the memory they access, can execute concurrently. The simultaneous execution of such conflicting critical sections allows the execution speed of such critical sections to guide the system in deciding which execution should be allowed to commit and which should be aborted. In particular, the first process to complete the critical section can be allowed to abort the other process of the system. The same scenario cannot be achieved by pessimistic critical sections and is demonstrated in Section 4.1.1.

A counterargument to this scenario is that such optimistic concurrency only allows one critical section to commit, while one must be aborted. Because mutual exclusion only allows one conflicting critical section execution at a time, and because mutual exclusion does not support failure atomicity (i.e., rollbacking of the critical section), mutual exclusion's pessimistic behavior is superior in terms of energy and efficiency. Mutual exclusion, unlike TM, suffers no wasted work because conflicting critical sections are limited to a single thread of execution, reducing the energy it uses. Furthermore, because mutual exclusion does not require original data to copied, as needed for TM's direct or deferred update, it executes faster.

While there is merit to this counterargument, an important scenario is not captured by it: *truly* optimistic critical sections can support multiple reader / single write executions which, if executed so the readers commit before the writer, all critical sections will succeed. This scenario is impossible to achieve using pessimistic critical sections. Although mutual exclusion can use read/write locking, as soon as a writer thread begins execution on a conflicting critical section, all readers must be stalled. TM's truly optimistic concurrency does not suffer from this overly pessimistic limitation of throughput and is therefore capable of producing an immeasurable amount of concurrent throughput under such conditions.

From a theoretical perspective, given L memory locations and P processes, mutual exclusion can support the consistent concurrent execution of $P * L$ number of readers or L writers. TM can support the consistent concurrent execution of $P * L$ number of readers and L writers. Using the above variables, the mathematical expression of the performance scalability of mutual exclusion ($S(ME)$) is:

$$S(ME) = (P * L) \oplus L$$

Using the same variables, the mathematical expression of the performance scalability of transactional memory is:

$$S(TM) = (P * L) + L$$

As should be clear from the above equations, mutual exclusion cannot achieve the same performance scalability of TM. This is because TM supports truly optimistic concurrency and mutual exclusion is confined to pessimistic concurrency. While other examples exist that demonstrate optimistic concurrency can increase throughput via contention management [8, 12, 18, 21, 34, 35], the above equations capture the indisputable mathematical limitations in mutual exclusion's performance scalability.

5. Modularity

Software modularity is an important aspect of software that is necessary for its reuse. Formally, software is *modular* if it can be composed in a new system without altering its internal implementation. Informally, software is modular if it can be used, in its entirety, through its interface.

By making software modular, it can be freely used in an unlimited number of software systems. Without software modularity, software can only be used in the original system where it was written. Clearly, without software modularity, software cannot be reused. Because most software developments are based on extensive library use, software reuse is an integral part of software development. As such, limiting software reuse, would result in severely hampered development capabilities and overall development time. For these reasons, software modularity is vital for any software paradigm to be practical. Software paradigms that do not support software modularity are, in short, impractical.

5.1 Mutual Exclusion and Software Modularity

In this section, we show that mutual exclusion, regardless of its implementation, fails to deliver software modularity. We demonstrate this through a running example started in Figure 7 which implements `inc()`, `mult()` and `get()`; these functions use lock `G` to respectively implement an increment, multiply and get operations for the shared data `g`.

```
1  void inc(int v) {
2    lock(G); g += v; unlock(G);
3  }
4
5  void mult(int v) {
6    lock(G); g *= v; unlock(G);
7  }
8
9  int get() {
10   lock(G); int v = g; unlock(G);
11   return v;
12 }
```

Figure 7. Mutual Exclusion for Increment, Multiply and Get of Shared Variable.

Now suppose a programmer wants to increment and multiply g by some values within the same atomic operation. The initial implementation may look like the following.

```
1  inc(a);
2  mult(-b);
```

An unwanted side-effect of such an implementation is the exposure of the intermediate state of g between `inc()` and `mult()`. A second thread performing a `get()` may read an inconsistent value of g; the value of g between `inc()` and `mult()`. This is demonstrated in the timing diagram of Figure 8 .

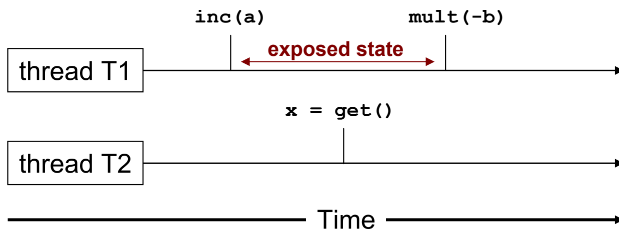


Figure 8. Example of Exposed State of Mutual Exclusion.

If the programmer needs the `inc()` and `mult()` operations to be executed together, without an intermediate state being revealed, he or she could make lock `G` reentrant. Reentrant locks are locks that can be obtained multiple times by a single thread without deadlocking. If `G` is made reentrant, the following code could be used to make `inc(a)` and `mult(-b)` atomic. A basic implementation of a reentrant lock is to associate a counter with its lock and increment the counter each time the `lock()` interface is called and to decrement the counter each time the `unlock()` interface is called. The reentrant lock is only truly locked when a call to `lock()` is made when its associated counter is 0. Likewise, the reentrant lock is only truly unlocked when a call to `unlock()` is made when its associated counter is 1.

```
1  lock(G);
2  inc(a);
3  mult(-b);
4  unlock(G);
```

If the above code uses reentrant locks, it will achieve the programmer's intended atomicity for `inc()` and `mult()`, isolating the state between `inc()` and `mult()`, which disallows the unwanted side-effect shown in Figure 8. While the atomicity of the operations is achieved, it is only achieved by exposing the implementation details of `inc()` and `mult()`. In particular, if the programmer had not known that lock `G` was used within `inc()` and `mult()`, making an atomic operation of `inc()` and `mult()` would be impossible.

An external atomic grouping of operations is impossible using embedded mutual exclusion without exposing the implementation details because the heart of mutual exclusion is based on *named* variables which the programmer specifies to guard their critical sections. Because these variables are named, they cannot be abstracted away and any programmer wishing to reuse the mutually exclusive code must be able to access and extend the implementation details.

```
1  void inc(int v)  { atomic { g += v; } }
2
3  void mult(int v) { atomic { g *= v; } }
4
5  int get()        { atomic { return g; } }
```

Figure 9. TM of Increment, Multiply and Get of Shared Variable.

5.1.1 Summary of Mutual Exclusion Modularity

As we presented at the beginning of this section, software modularity can be informally understood as a component's ability to be used entirely from its interface. Therefore, components that cannot be used entirely from their interface, components that must expose their implementation details to be extended, are not modular. As such, the paradigm of mutual exclusion does not support software modularity.

5.2 Transactional Memory and Software Modularity

Transactional memory works in a fundamentally different manner than mutual exclusion, with regard to its interface and implementation. To begin, as demonstrated in Section 3, TMs do not generally expose any of their implementation details to client code. In fact, in many TMs, client code is more versatile if it knows and assumes nothing about the active implementation of the TM. By abstracting away details of TM implementation, a TM subsystem can adapt its behavior to the most efficient configuration for the program's current workload, much like the algorithms used for efficient operation of processes controlled by operating systems. TM uses such abstractions to optimize the performance of concurrent programs using various consistency checking methods, conflict detection times, updating policies, and contention management schemes [6, 8, 12, 18, 34].

5.2.1 Achieving TM Software Modularity

TM achieves software modularity by allowing transactions to nest. With transactional nesting, individual transactions can be wrapped inside of other transactions which call the methods where they reside, resulting in a transaction composed of both the parent and child transaction. Furthermore, this is achieved without altering or understanding the child transaction's implementation. To best demonstrate transactional nesting, we reuse the prior mutual exclusion example shown in Figure 7 and implement it using transactions as shown in Figure 9.

As before, the programmer's goal is to implement a combination of `inc()` and `mult()` executed in an atomic fashion. The basic, and incorrect implementation is demonstrated below:

```
1  inc(a);
2  mult(-b);
```

Even with transactions, this approach fails because the transactions within inc() and mult() begin and end inside their respective functions. However, to make the above operations atomic, the programmer need only make the following modification shown in Figure 10.

```
1  atomic {           // atomic {
2    inc(a);          //   atomic { g += a;  }
3    mult(-b);        //   atomic { g *= -b; }
4  }                  // }
```

Figure 10. Modularity: Transaction of Increment and Multiply.

In effect, the TM system subsumes the transactions that are nested inside of the inc() and mult() operations. [6] The left side of Figure 10 shows the actual code of the transaction, while the right side shows the child transactions that are subsumed by the parent transaction.

Because transactions are isolated and atomic, the resulting state of g, from operations inc() and mult(), is invisible to outside observers until the transaction is committed. As such, outside threads cannot view any intermediate state constructed by partial transaction execution. The result of such isolated behavior is the guaranteed consistent concurrent execution of interleaved accesses to shared memory from in-flight transactions. This is demonstrated in Figure 11; let g=0 and assume deferred update is the active updating policy, as explained in Section 4.2.

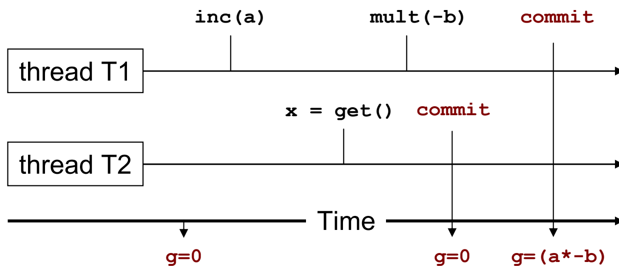


Figure 11. Example of Isolated and Consistent State of TM.

As shown in Figure 11, multiple concurrently executing threads can read and write to the same shared memory in a consistent and isolated fashion when using TM. In the example, thread T_2 performs x = get() after T_1 has already executed inc(a). However, since T_1 has not yet committed its transaction, T_2's view of shared data g is consistent (g=0). When T_2 begins the commit phase of its transaction, the TM subsystem verifies that shared data g has not been updated since it initially read it. Since no other transaction has updated shared data g, T_2's transaction is permitted to commit. Thread T_1 then continues with its mult() operation and then enters its commit phase. The TM subsystem also verifies the consistency of T_1's transaction before it is allowed to commit. Again, since no one transaction has updated shared data g between its reads and writes to it, T_1's transaction is permitted to commit.

The above analysis demonstrates that software modularity can be achieved in TM through transactional nesting (Figure 10). In this case, the specific software modularity achieved is extension to an existing critical section. Critical section extension was also possible with mutual exclusion, as demonstrated in Section 5.1, but only through exposing the details behind the mutual exclusion implementation. Due to this, mutual exclusion fails to deliver a practical level of software modularity.

5.2.2 Summary of Transactional Memory Modularity

TM supports software modularity by allowing transactions to nest, to any depth, while logically grouping the shared data accesses within the transactions into an atomic, consistent and isolated (ACI) operation. Transactional nesting is natural to the programmer because nested transactions behave in the same manner as unnested transactions. TM's ACI support ensures transactions will behave in a correct manner regardless of if the transaction is used by itself or subsumed into a larger transaction.

6. Conclusion

This paper presented an overview of TM, an emerging type of concurrency control, that uses transactions as its synchronization mechanism and supports truly optimistic concurrency. We demonstrated that TM is (1) simple, (2) scalable and (3) modular. TM simplifies parallel programming by moving the complexity of shared memory management into the underlying TM subsystem. TM achieves scalability by using optimistic critical sections that support wider concurrent critical section execution than other methods. TM supports software modularity by allowing transactions to nest to any depth, allowing transactions to be reused within software without knowledge of any of their implementation details.

7. Acknowledgements

We thank Nir Shavit of Sun Microsystems and Tel-Aviv University for his expert direction on the writing of this work. We also thank Ken Prager of Raytheon Company for his editorial corrections on the organization of this paper.

References

[1] A.-R. Adl-Tabatabai, C. Kozyrakis, and B. Saha. Unlocking concurrency. *ACM Queue*, 4(10):24–33, 2006.

[2] C. S. Ananian, K. Asanovic, B. C. Kuszmaul, C. E. Leiserson, and S. Lie. Unbounded transactional memory. volume 26, pages 59–69. IEEE Computer Society Press, Los Alamitos, CA, USA, 2006.

[3] P. A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, 1987.

[4] D. Culler, J. P. Singh, and A. Gupta. *Parallel Computer Architecture: A Hardware/Software Approach*. Morgan Kaufmann Publishers, 1999.

[6] We assume flattened, closed nesting for simplicity, although we admit the execution for such nesting would be different if an open nested TM system were used [11, 27, 28].

[5] R. Ennals. Software transactional memory should not be obstruction-free. Technical Report IRC-TR-06-052, Intel Research Cambridge Tech Report, Jan 2006.

[6] J. Gottschlich. Exploration of lock-based software transactional memory. Master's thesis, University of Colorado at Boulder, Oct. 2007.

[7] J. E. Gottschlich and D. A. Connors. DracoSTM: A practical C++ approach to software transactional memory. In *ACM SIGPLAN Library-Centric Software Design (LCSD)*, Oct. 2007.

[8] J. E. Gottschlich and D. A. Connors. Extending contention managers for user-defined priority-based transactions. In *Proceedings of the 2008 Workshop on Exploiting Parallelism with Transactional Memory and other Hardware Assisted Methods*. Apr 2008.

[9] J. E. Gottschlich, D. A. Connors, D. Y. Winkler, J. G. Siek, and M. Vachharajani. An intentional library approach to lock-aware transactional memory. Technical Report CU-CS 1048-08, University of Colorado at Boulder, Oct 2008.

[10] J. E. Gottschlich, J. G. Siek, P. J. Rogers, and M. Vachharajani. Toward simplified parallel support in C++. In *Proceedings of the 2009 Conference on Boost Libraries*. May 2009.

[11] J. Gray and A. Reuter. *Transaction Processing : Concepts and Techniques (Morgan Kaufmann Series in Data Management Systems)*. Morgan Kaufmann, October 1992.

[12] Guerraoui, Herlihy, and Pochon. Polymorphic contention management. In *DISC: International Symposium on Distributed Computing*. LNCS, 2005.

[13] R. Guerraoui, M. Kapałka, and J. Vitek. STMBench7: A benchmark for software transactional memory. In *Proceedings of the Second European Systems Conference EuroSys 2007*, pages 315–324. ACM, Mar. 2007.

[14] T. Harris and K. Fraser. Language support for lightweight transactions. In *OOPSLA '03: Proceedings of the 18th annual ACM SIGPLAN conference on Object-oriented programing, systems, languages, and applications*, pages 388–402, New York, NY, USA, 2003. ACM.

[15] T. Harris, M. Plesko, A. Shinnar, and D. Tarditi. Optimizing memory transactions. *ACM SIGPLAN Notices*, 41(6):14–25, June 2006.

[16] J. Hennessy and D. Patterson. *Computer Architecture - A Quantitative Approach.* Morgan Kaufmann, 2003.

[17] M. Herlihy. Wait-free synchronization. *ACM Trans. Program. Lang. Syst.*, 13(1):124–149, 1991.

[18] M. Herlihy, V. Luchangco, M. Moir, and I. William N. Scherer. Software transactional memory for dynamic-sized data structures. In *Proceedings of the symposium on principles of distributed computing*, pages 92–101, New York, NY, USA, 2003. ACM.

[19] M. Herlihy and J. E. B. Moss. Transactional memory: Architectural support for lock-free data structures. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*, pages 289–300. May 1993.

[20] M. Herlihy and N. Shavit. *The Art of Multiprocessor Programming*. Elsevier, Inc., 2008.

[21] W. N. S. III and M. L. Scott. Advanced contention management for dynamic software transactional memory. In M. K. Aguilera and J. Aspnes, editors, *PODC*, pages 240–248. ACM, 2005.

[22] J. R. Larus and R. Rajwar. *Transactional Memory*. Morgan and Claypool, 2006.

[23] D. Lea. *Concurrent Programming in Java: Design Principles and Patterns*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1996.

[24] V. J. Marathe, W. Scherer, and M. Scott. Adaptive software transactional memory. In *DISC: International Symposium on Distributed Computing*. LNCS, 2005.

[25] V. J. Marathe, M. F. Spear, C. Heriot, A. Acharya, D. Eisenstat, W. N. Scherer, III, and M. L. Scott. Lowering the overhead of nonblocking software transactional memory. revised, University of Rochester, Computer Science Department, May 2006.

[26] K. E. Moore, J. Bobba, M. J. Moravan, M. D. Hill, and D. A. Wood. Logtm: Log-based transactional memory. In *Proceedings of the 12th International Symposium on High-Performance Computer Architecture*, pages 254–265. IEEE Computer Society, Feb. 2006.

[27] E. Moss. Nesting transactions: Why and what do we need? *TRANSACT*, Jun 2006.

[28] J. E. B. Moss. Open nested transactions: Semantics and support. In *WMPI*, Austin, TX, February 2006.

[29] 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 Object oriented programming systems languages and applications*, pages 195–212, New York, NY, USA, 2008. ACM.

[30] K. Olukotun, L. Hammond, and J. Laudon. *Chip Multiprocessor Architecture: Techniques to Improve Throughput and Latency*. Morgan and Claypool, 2007.

[31] T. Peierls, B. Goetz, J. Bloch, J. Bowbeer, D. Lea, and D. Holmes. *Java Concurrency in Practice*. Addison-Wesley Professional, 2005.

[32] B. Saha, A.-R. Adl-Tabatabai, R. L. Hudson, C. C. Minh, and B. Hertzberg. McRT-STM: a high performance software transactional memory system for a multi-core runtime. In *PPOPP*. ACM SIGPLAN 2006, Mar. 2006.

[33] N. Shavit and D. Touitou. Software transactional memory. In *Proceedings of the 14th ACM Symposium on Principles of Distributed Computing*, pages 204–213. Aug 1995.

[34] M. F. Spear, L. Dalessandro, V. Marathe, and M. L. Scott. A comprehensive strategy for contention management in software transactional memory. In *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming 2009*, Feb. 2009.

[35] M. F. Spear, V. J. Marathe, W. N. Scherer III, and M. L. Scott. Conflict detection and validation strategies for software transactional memory. In *Proceedings of the 20th International Symposium on Distributed Computing*, Sep 2006.

[36] M. F. 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*. Feb 2008.

[37] A. Welc, B. Saha, and A.-R. Adl-Tabatabai. Irrevocable transactions and their applications. In *SPAA*, 2008.