An Intentional Library Approach to Lock-Aware Transactional Memory

Justin E. Gottschlich[1], Daniel A. Connors[1], Dwight Y. Winkler[2], Jeremy G. Siek[1] and Manish Vachharajani[1]

[1]*University of Colorado at Boulder,* [2]*Nodeka, LLC.*

University of Colorado at Boulder
Technical Report CU-CS 1048-08

Dept. of Computer Science
Campus Box 430
University of Colorado
Boulder, Colorado 80309-0430

Dept. of Electrical and Computer Engineering
Campus Box 425
University of Colorado
Boulder, Colorado 80309-0425

Abstract

Transactional memory (TM) has garnered significant interest as an alternative to existing concurrency methods due to its simplified programming model, natural composition characteristics and native support of optimistic concurrency. Yet, mutual exclusion locks are ubiquitous in existing parallel software due to their fast execution, inherent property for irreversible operations and long-standing underlying support in modern instruction set architectures (ISAs). For transactions to become practical, lock-based and TM-based concurrency methods must be unified into a single model.

This work presents the first lock-aware transactional memory (LATM) solution with a deliberate library-based approach. Our LATM system supports locks inside of transactions (LiT) and locks outside of transactions (LoT) while preserving the original programmer-intended locking structure. Our solution provides three policies with varying degrees of performance and required programming. The most basic LATM policy enables transaction-lock correctness without any programming overhead, while improved performance can be achieved by programmer-specified conflicts between locks and transactions. The differences in LATM policy performance are presented through a number of experimental benchmarks.

1. Introduction

Transactional memory (TM), as proposed by Herlihy and Moss [10], is a concurrency control mechanism aimed at simplifying parallel programming. TM simplifies parallel programming by moving shared memory complexity out of the programmer's view and into the TM system [2, 8]. Yet, transactions have other benefits outside of reducing parallel programming complexity. Particular synchronization problems, such as deadlocks and starvation, are eliminated by TM [3, 4, 7, 12]. Transactions also support composition, a characteristic that enables subtransactions to be nested within an outer parent transaction, forming a singularly visible operation that behaves correctly (atomically, isolated and consistently) [9, 13]. Furthermore, TM is an optimistic concurrency mechanism that widens the performance benefits found in highly concurrent algorithms [15, 16].

Yet, for all of the benefits of transactional programming, its native inoperability with mutual exclusion locks is a major obstacle to its adoption as a practical solution [1]. Herlihy and Shavit note that mutual exclusion (implemented through locks) is arguably the most predominant form of thread synchronization used in parallel software [11], yet its native incompatibility with transactions places a considerable restriction on the practical use of TM in real-world software.

A TM system that is cooperative with locks can extend the lifetime and improve the behavior of previously generated parallel programs. We extend the DracoSTM C++ STM library [5, 6] to be a lock-aware transactional memory (LATM) system supporting the simultaneous execution of transactions and locks. Of critical importance is that the extended DracoSTM LATM system naturally supports lock-based composition for locks placed inside of transactions (LiT), a characteristic previously unavailable in locks.

The LATM policies presented in this paper are implemented with software library limitations as a central concern. While novel operating system (OS)-level and language-based transaction-lock cooperative models have been previously found [16, 20], these implementations use constructs not available in library-based STM systems. While useful, the previously identified OS-level and language-based solutions do not address the critical need for a transaction-lock unified model expressed entirely within the limitations of a software library. In languages that are unlikely to be extended, such as C++ [18], and development environments bound to specific constraints, such as a particular compiler or OS, library-based solutions are paramount as they present practical solutions within industry-based constraints. Our approach presents a novel library-based LATM solution aimed at addressing these concerns.

This paper makes the following contributions:

1. Extension of the DracoSTM library for support of locks outside of transactions (LoT) and locks inside of transactions (LiT) [available at http://rogue.colorado.edu/draco/].
2. Proof that an LATM LiT system naturally enables lock composition. Analytical examples are examined that show LiT composable locks demonstrating atomicity, isolation and consistency as well as deadlock avoidance.
3. Introduction of three novel LATM policies: full lock protection, TM-lock protection, and TX-lock protection. Each LATM policy provides different programming / performance trade offs. Experimental results are provided which highlight the performance and programming differences of the three LATM policies.

2. Background

Transaction-lock interaction does not natively exhibit shared memory consistency and correctness due to differences in underlying critical section semantics. Strongly and weakly isolated transactional memory systems are susceptible to incorrect execution when used with pessimistic critical sections.

Mutual exclusion locks use pessimistic critical section semantics; execution of a lock-controlled critical section is limited to one thread and guarantees the executing thread has mutually exclusive (isolated) access to the critical section code region [21]. Transactions use optimistic critical section semantics; transaction-controlled critical sections support unlimited concurrent thread execution. Shared memory conflicts arising from simultaneous transaction execution are handled by the TM system during the commit phase of the transaction [6, 14]. Optimistic critical sections and pessimistic critical sections are natively inoperable due to their contradictive semantics as demonstrated in Figures 1, 2 and 3.

```
1  native_trans<int> x;
2
3  int lock_dec() {
4    lock(L1);
5    int val = --x;
6    unlock(L1);
7    return val;
8  }
9
10 void tx_inc() {
11   for (transaction t;;t.restart())
12     try {
13       ++t.write(x);
14       t.end(); break;
15     } catch (aborted_tx &) {}
16 }
```

Figure 1. Lock and transaction violation (code).

Without an intervening system, thread T_1 executing `lock_dec()` and thread T_2 executing `tx_inc()` (Figure 1) are not guaranteed to operate consistently, as they would if each function were run in a lock-only or transaction-only system, respectively. The following example demonstrates how the correctness of Figure 1 is violated from a deferred update and direct update standpoint.

Deferred Update Lock/Transaction Violation. A deferred update TM system stores transactional writes off to the side, updating global memory once the transaction commits. A deferred update system can exhibit an inconsistency in Figure 1 in the following way (as demonstrated by Figure 2's timing diagram). Given: `x = 0`, thread T_1 executes `lock_dec()` and thread T_2 executes `tx_inc()`. T_2 executes line 10 - the first half of line 13 (storing a transactional value for `x = 0`, but not performing the increment). T_1 executes lines 3-5 (global `x = -1`). T_2 executes the remainder of line 13 and line 14 (`++x` on its stored reference of 0, setting its local $x = 1$) and then commits. T_1 executes lines 6-7. The resulting global state of `x = 1` is incorrect; `x = 0` is correct as one thread has incremented it and one thread has decremented it.

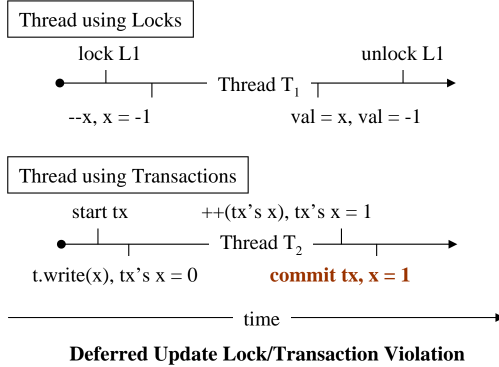


Figure 2. Deferred Update Lock/Transaction Violation.

Direct Update Lock/Transaction Violation. A direct update TM system stores transactional writes directly in global memory, storing an original backup copy off to the side in the event the transaction must be unwound. In a direct update system, threads T_1 and T_2 can exhibit inconsistencies using the code shown in Figure 1 in a variety of ways; Figure 3's timing diagram shows one such case. T_1 executes line 3 - the first half of line 5 (decrementing `x`, but not setting `val`). T_2 executes lines 10-13 (incrementing the global `x` and creating a restore point of `x = -1`). T_1 executes the remainder of line 5 (`val = 0`). T_2 executes line 14, but is required to abort, restoring `x` to -1. T_1 completes and returns `val = 0` which is incorrect. T_2 never committed its `x = 0` and has not successfully committed, therefore, `x` has never correctly been set to 0.

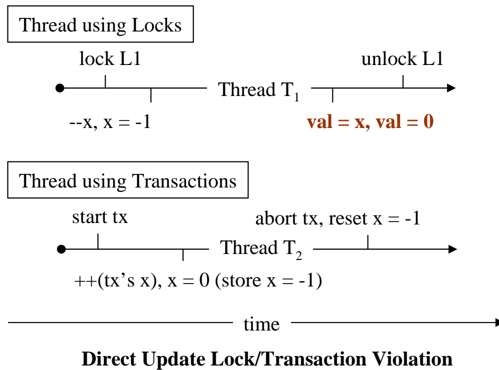


Figure 3. Direct Update Lock/Transaction Violation.

2.1 Overcoming Transaction-Lock Inoperability

In order for transactions and locks to cooperate, transactions must adhere to a single mutual exclusion rule described in Lemma 1.

Mutual Exclusion Lemma 1. *Mutual exclusion semantics require that instructions within a mutex guarded critical section be limited to ≤ 1 simultaneous thread of execution.*

Proof. (Contradiction) Consider the following order of operations given a mutual exclusion critical section spanning two functions: (1) `inc()` which increments two global integers x and y and (2) `get()` which returns the current values of x and y, where $x = 0$ and $y = 0$ as shown in Figure 4. T_1 executes `inc()` lines 3-5 and halts. T_2 then executes `get()` in its entirety (lines 10-15), violating the Mutual Exclusion Lemma 1. T_1 then finishes execution of `inc()` (lines 6-8). The resulting state seen by T_2 ($x = 1$, $y = 0$) is incorrect. The correct program state is either $x = 0$, $y = 0$ or $x = 1$, $y = 1$. □

```
1  int x = 0, y = 0;
2
3  void inc() {
4    lock(L1); // do not limit L1 to 1 thread
5    ++x;
6    ++y;
7    unlock(L1);
8  }
9
10 void get(int &retX, int &retY) {
11   lock(L1); // do not limit L1 to 1 thread
12   retY = y;
13   retX = x;
14   unlock(L1);
15 }
```

Figure 4. Violating Mutual Exclusion Lemma 1.

Our LATM implementation adheres to Lemma 1 and is discussed in two high-level views: (1) locks outside of transactions (LoT) and (2) locks inside of transactions (LiT). We extend DracoSTM (C++ STM library [5, 6]) to support lock-aware transactions by supplying a pass-through interface that is used in place of prior locking calls. We currently only provide C++ pthreads support, as it is the most common threading library for C++ [11]. An example of the required interface change in client code is shown below with commented out pthreads interfaces, followed by their replaced DracoSTM calls.

```
1  pthread_mutex_t L1; // pthread lock
2  // pthread_mutex_lock(&L1);
3  transaction::lock(&L1);
4  // pthread_mutex_trylock(&L1);
5  transaction::trylock(&L1);
6  // pthread_mutex_unlock(&L1);
7  transaction::unlock(&L1);
```

The DracoSTM locking API is used to perform the additional transaction-lock communication before or after the pthreads interface is called. In all cases, the DracoSTM pass-through interface results in at least one corresponding call to the appropriate pthreads interface to lock, try to lock or unlock the mutual exclusion lock. Further details are provided in later sections.

3. Locks Outside of Transactions (LoT)

Locks outside of transactions (LoT) are scenarios where a pessimistic critical section of a lock is executed in a thread T_1 while an optimistic critical section of a transaction is executed in a thread T_2, simultaneously. Thread T_1's lock-based pessimistic critical section is entirely outside of thread T_2's transaction, thus the

term *locks outside of transactions* or LoT. Figure 5 sets up a running LoT example, used throughout this section, by constructing six functions which are simultaneously executed by six threads. Three of the functions in Figure 5, `tx1()`, `tx2()` and `tx3()`, are transaction-based, while the other three functions, `lock1()`, `lock2()` and `lock3()`, are lock-based. The functions `tx1()`, `tx2()` and `lock3()` do not have any memory conflict with any other transaction-based or lock-based function and should therefore be able to run concurrently with any of the other functions. However, certain LoT policies inhibit the execution of these non-conflicting functions; details of such inhibited behavior is explained in the following subsections. Figure 5 is used throughout this section to illustrate the differences in the LoT policies.

```
1  native_trans<int> arr1[99], arr2[99];
2
3  void tx1() { /* no conflict */ }
4  void tx2() { /* no conflict */ }
5  void tx3() {
6    for (transaction t;;t.restart())
7      try {
8        for (int i = 0; i < 99; ++i)
9        {
10         ++t.w(arr1[i]).value();
11         ++t.w(arr2[i]).value();
12       }
13       t.end(); break;
14     } catch (aborted_tx&) {}
15 }
16
17 int lock1() {
18   transaction::lock(L1); int sum = 0;
19   for (int i = 0; i < 99; ++i) sum += arr1[i];
20   transaction::unlock(L1); return sum;
21 }
22 int lock2() {
23   transaction::lock(L2); int sum = 0;
24   for (int i = 0; i < 99; ++i) sum += arr2[i];
25   transaction::unlock(L2); return sum;
26 }
27 int lock3() { /* no conflict */ }
```

Figure 5. 3 Transaction Threads, 3 Locking Threads.

3.1 LoT Full Lock Protection

The most basic implementation of transaction-lock cooperation, what we call full lock protection, is to enforce all transactions to commit or abort before a lock's critical section is executed. All locks outside of transactions are protected from transactions violating their critical section execution by disallowing transactions to run in conjunction with locks. Transactions are stalled until all LoT critical sections are complete and their corresponding locks are released.

An example of full lock protection is shown in Figure 6 using the previously described six threaded model. Full lock protection has no programmer requirements; no new code is required, aside from alteration of existing locking code to use the LATM pass-through interfaces. Additionally, an understanding of how the locks behave within the system to enable transaction-lock cooperation is also not needed. However, full lock protection suffers some performance penalties. As seen in Figure 6, $T_1 - T_3$ are blocked for the entire duration of the critical sections obtained in $T_4 - T_6$, since full lock protection prevents any transactions from running while LoTs are obtained. Although T_3 does conflict with $T_4 - T_5$,

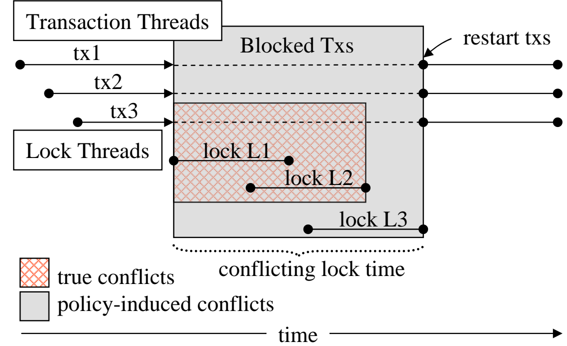


Figure 6. LoT Example of Full Lock Protection.

T_6's critical section does not interfere with any of the transactions and should therefore not prevent any transactions from running concurrently; TM-lock protection, the next level of lock protection, is able to avoid such unnecessary stalling.

3.2 LoT TM-Lock Protection

TM-lock protection is slightly more complex than full lock protection, yet it can yield better overall system performance. TM-lock protection works in the following way: locks which can conflict with transactions are identified by the programmer at startup. Once a conflicting LoT is acquired, all in-flight transactions are either committed or aborted. Transactions are then blocked until the conflicting lock-based critical sections are completed and released. Locks that do not conflict with transactions do not cause any transactions to stall.

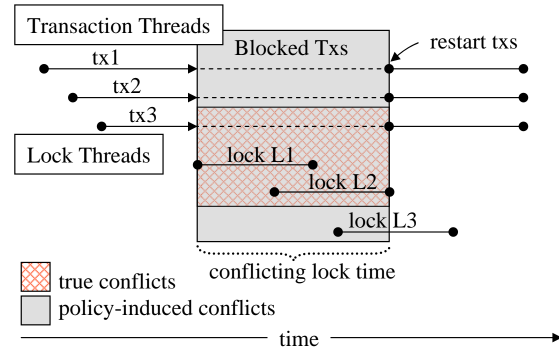


Figure 7. LoT Example of TM-Lock Protection.

The identification of locks that can conflict with transactions requires the programmer to (1) write new code and (2) have a basic understanding of the software system. Due to this, the requirements of TM-lock protection are greater on the end programmer. The trade-off for higher programmer requirements is greater overall system performance.

TM-lock protection addresses the problem of transactions unnecessarily stalled when T_6 is executing. When using TM-lock protection, the end programmer must explicitly express which locks can conflict with the TM system. In this example, locks `L1` and `L2` from threads T_4 and T_5 conflict with `tx3` in thread T_3. The end

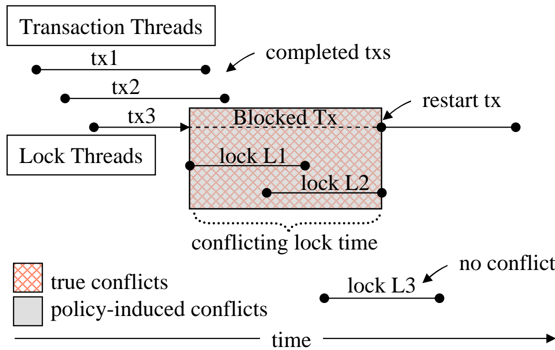
programmer would explicitly label these locks as conflicting in the following way:

```
transaction::do_tm_conflicting_lock_protection();
transaction::add_tm_conflicting_lock(L1);
transaction::add_tm_conflicting_lock(L2);
```

As shown in the six threaded example, TM-lock protection shortens the overall TM run-time by allowing $T_1 - T_3$ to restart their transactions as soon as L2's critical section is completed. Yet, there still exists unnecessary stalls in threads T_1 and T_2 as their associated transactions do not conflict with any of the lock-based critical sections of $T_4 - T_6$. The remaining unnecessary stalls are resolved by using TX-lock protection, the third lock protection policy.

3.3 LoT TX-Lock Protection

TX-lock protection enables maximum performance throughput by identifying only true conflicts as they exist per transaction. TX-lock protection is similar to TM-lock protection except rather than requiring conflicting locks be identified at a general TM-level, conflicting locks are identified at a transaction-level. While this level of protection yields the highest level of performance, it also requires the greatest level of familiarity of the locks within the system and the most hand-crafted code.



LATM LoT: TX-Lock Protection
(tx3 conflicts with *L1 & L2*)

Figure 8. LoT Example of TX-Lock Protection.

An example of TX-lock protection is in Figure 8. By using TX-lock protection and explicitly identifying conflicting locks per transaction, the system only stalls for true conflicts, increasing overall system performance. The code required for correct TX-lock protection in the prior six threaded example is shown below by extending the original tx3() implementation:

```
void tx3() {
  for (transaction t;;t.restart())
    try {
      // identify conflicting locks
      t.add_tx_conflicting_lock(L1);
      t.add_tx_conflicting_lock(L2);
      for (int i = 0; i < 99; ++i)
      {
        ++t.w(arr1[i]).value();
        ++t.w(arr2[i]).value();
      }
      t.end(); break;
    } catch (aborted_tx&) {}
}
```

Using TX-lock protection, threads T_1 and T_2 are no longer stalled when threads T_4 and T_5 lock their associated locks, L1 and L2. In fact, only thread T_3 (the only true conflict) is stalled while the critical sections created by L1 and L2 are executing, resulting in the highest transaction-lock cooperative performance while still adhering to Lemma 1.

3.4 LoT Policy Performance

Figures 9, 10 and 11 display the execution time (*lower is faster*) of the various LoT policies based on the original six threaded example.

Three benchmarks were run, which aimed to demonstrate how the policies behaved, under the following varying workload conditions: balanced, transaction-intensive and lock-intensive. Results shown in Figure 9 represent balanced conditions; the locks and transactions ran at roughly equal times. Figure 10 shows transaction-intensive workloads, the transaction workload was magnified from the balanced conditions. Figure 11 shows lock-intensive workloads which magnified the locking workload from the balanced workload baseline.

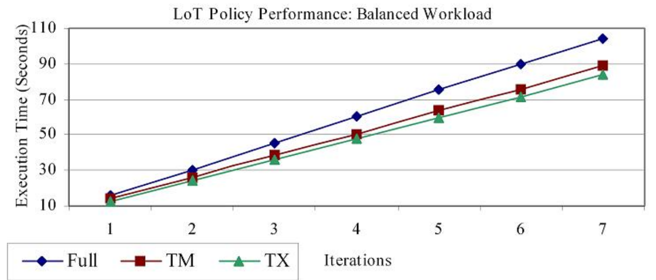


Figure 9. LoT Balanced Workload.

The performance metrics were gathered on a 1.0 GHz Sun Fire T2000, 32 hardware threads and 32 GB RAM. The x-axis shows the number of execution iterations, the y-axis shows the total execution time of each iteration in seconds. Each data point per iteration is the execution time of the LoT policy averaged over three runs.

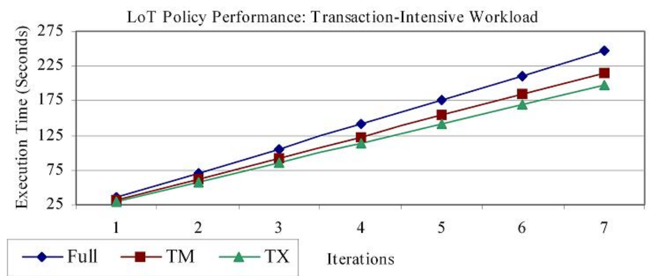


Figure 10. LoT Transaction-Intensive Workload.

From Figure 9, full-lock protection performs at an average of 15 seconds per iteration. TM-lock protection executes each iteration $\approx$ 12.5 seconds, a 20% performance improvement over full lock protection. TX-lock protection executes under 12 seconds, a performance improvement over full lock protection of $> 25\%$ and $\approx 4\%$ faster than TM-lock protection. Figure 10 shows full-lock protection performing at an average of 35 seconds, while TM-lock protection executes at an average of 30.5 seconds, $\approx 15\%$ performance improvement over full lock protection. TX-lock protection performs each iteration at roughly 28.15 seconds resulting in a

$\approx 24\%$ performance improvement over full lock protection and a $\approx 8\%$ improvement in performance over TM-lock protection.

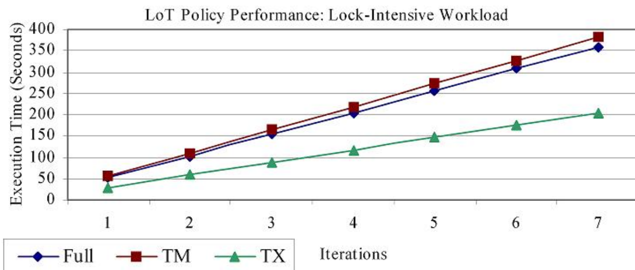


Figure 11. LoT Lock-Intensive Workload.

Figure 11 reveals interesting results for the LoT policies when the workload becomes lock-intensive. To begin, TM-lock protection performs worse than full lock protection. We believe this is due to the overhead in the TM-lock protection algorithm. Usually this benefit is invisible due to the benefit gained by false conflicts. However, in this particular case, the multiple executed locks were almost always conflicting, yielding no improved performance from the optimized TM-lock protection algorithm. Secondly, TX-lock protection performed astoundingly better than the other two policies. We believe this is due to TX-lock protection being able to identify transactional conflicts with only one of the three TX-lock protected, transaction-executing threads, whereas both full lock protection and TM-lock protection could not. This results in two of the three TX-lock protected, transaction threads executing without conflicting with the increased locking, allowing the TM system's invalidation to optimize its consistency checking for single in-flight transaction to 0 operations [6]. This yields TX-lock performance of 86% beyond TM-lock protection and 75% beyond full lock protection.

4. Locks Inside of Transactions (LiT)

Locks inside of transactions (LiT) are scenarios where a lock-based pessimistic critical section is executed partially or completely inside a transaction. Only two of the three possible LiT scenarios are supported by our work: (1) pessimistic critical sections are encapsulated entirely within a transaction or (2) pessimistic critical sections start inside a transaction but end after the transaction has terminated. We do not support LiT scenarios where pessimistic critical sections start before a transaction begins, having the front-end of the transaction encapsulated by the pessimistic critical section. The reason to disallow such behavior is to avoid deadlocks.

Consider the following scenario. Thread T_1 has an in-flight irrevocable transaction Tx_1 and thread T_2, after obtaining lock L_2, starts a transaction Tx_2. Tx_2 is not allowed to make forward progress until it is made irrevocable (details to follow). Tx_1 is already in-flight and irrevocable. Since two irrevocable transactions cannot run simultaneously as they are not guaranteed to be devoid of conflicts with one another, Tx_2 must stall until Tx_1 completes. If irrevocable transaction Tx_1 requires lock L_2 to complete its work the system will deadlock. Tx_1 cannot make forward progress due to its dependency upon L_2 (currently held by Tx_2) and Tx_2 cannot make forward progress as it requires Tx_1 to complete before it can run. As such, LiT scenarios where locks encapsulate the front-end of a transaction are disallowed; our implementation immediately throws an exception when this behavior is detected.

4.1 Irrevocable and Isolated Transactions

The LiT algorithms use the same policies as the LoT algorithms: full lock protection, TM-lock protection and TX-lock protection. Locks inside of transactions have the same characteristics as normal mutual exclusion locks, and Lemma 1 must be followed in order to ensure correctness. Since the LiT algorithms use locks acquired inside of transactions and these locks are not guaranteed to have failure atomicity as transactions do, the containing transactions must become irrevocable (see Lemma 2). Irrevocable transactions, characterized by $T_{(irrevocable)} = true$, are transactions that cannot be aborted. The concept of irrevocable (or inevitable) transactions is not new; Welc et al. and Spear et al. have shown these types of transactions to be of significant importance as well as having a variety of practical uses [17, 19]. We extend the prior work of Welc et al. and Spear et al. by using irrevocable transactions to enable pessimistic critical sections within transactions, as well as to create composable locks within transactions.

In addition to irrevocable transactions, the LiT full lock protection and TM-lock protection require a new type of transaction, one that we term as an *isolated transaction*. Isolated transactions, characterized by $T_{(isolated)} = true$, are transactions that cannot be aborted and require that no other type of transaction run along side it simultaneously. Isolated transactions can be viewed as a superset of irrevocable transactions; isolated transactions have the properties of irrevocable transactions and must be run in isolation.

To demonstrate how the LiT algorithms work, consider the six threaded example shown in Figure 12.

```
1  native_trans<int> g1, g2, g3, g4;
2
3  void tx1() { /* no conflict */ }
4  void tx2() {
5    transaction::add_tm_conflicting_lock(L2);
6    for (transaction t;;t.restart())
7      try {
8        t.add_tx_conflicting_lock(L2);
9        inc2();
10       t.end(); break;
11     } catch (aborted_tx&) {}
12 }
13 void tx3() {
14   transaction::add_tm_conflicting_lock(L3);
15   for (transaction t;;t.restart())
16     try {
17       t.add_tx_conflicting_lock(L3);
18       inc3();
19       t.end(); break;
20     } catch (aborted_tx&) {}
21 }
22
23 void inc2() {
24   lock(L2); ++g2.value(); unlock(L2);
25 }
26 void inc3() {
27   lock(L3); ++g3.value(); unlock(L3);
28 }
29 void inc4() { /* no conflict */ }
```

Figure 12. 3 LiT Transaction Threads, 3 Locking Threads.

In Figure 12 thread T_1 executes `tx1()`, T_2 executes `tx2()`, T_3 executes `tx3()`, T_4 executes `inc2()`, T_5 executes `inc3()` and T_6 executes `inc4()`. Threads T_1 (`tx1()`) and T_6 (`inc4()`) do not conflict with any other thread, yet their execution can be inhibited by other threads based on the LiT policy employed. Thread T_2 has

a true conflict with thread T_4 (both threads call `inc2()`) and thread T_3 has a true conflict with thread T_5 (both threads call `inc3()`). A staggered start time is used in the coming diagrams: T_1 starts, followed by T_2, T_3, T_4, T_5 and finally T_6. We label the LiT threads based on the locks they acquire: $tx1$ acquires lock $L1$, thread $tx2$ acquires lock $L2$ and thread $tx3$ acquires lock $L3$. The same taxonomy is used for locking threads: thread $lockL2$ acquires lock $L2$, thread $lockL3$ acquires lock $L3$ and thread $lockL4$ acquires lock $L4$.

In Figure 12, both `add_tm_conflicting_lock()` and `add_tx_conflicting_lock()` are present. If the system is using TM-lock protection only the `add_tm_conflicting_lock()` is necessary, whereas if the system is using TX-lock protection only the `add_tx_conflicting_lock()` is necessary. If neither TM-lock or TX-lock protection is in use, neither call is needed. These interfaces are supplied for the completeness of the example.

4.2 LiT Full-Lock Protection

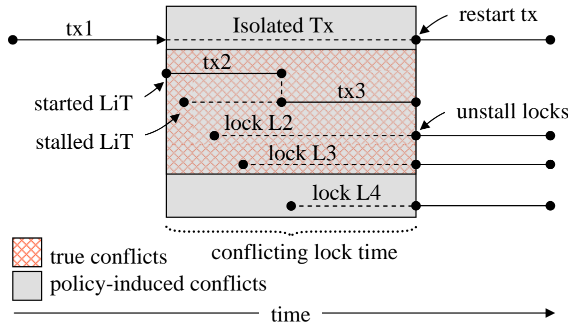


Figure 13. LiT Example of Full Lock Protection.

Figure 13 demonstrates how the six threads interact using the full lock protection policy, as well as showing policy conflicts in comparison to true conflicts. The LiT full lock protection algorithm requires that any transaction that has a lock inside of it be run as an isolated transaction. Prior to a lock inside the transaction being obtained, all other in-flight transactions are aborted or committed and all currently held locks must execute through release. Future attempts to obtain locks outside of the isolated transaction are prevented until the transaction commits. This behavior is required as the system must assume (1) all external locks can conflict with the isolated transaction, so no external locks can be obtained; and (2) all external transactions can conflict with the LiT transaction, and therefore no external transactions can execute.

For Figure 13, once $tx2$ begins, $tx1$ is stalled as $tx2$ must run as an isolated transaction. Due to $tx2$'s isolation, $tx3$ is also stalled. Both $lockL2$ and $lockL3$ are also stalled because full lock protection disallows transactions from running while LoT locks are obtained; as $tx2$ is an isolated transaction, the threads attempting to lock $L2$ and $L3$ are stalled until $tx2$ completes. When $tx2$ completes, $tx3$ is started as it has stalled for the longest amount of time. The thread executing $lockL4$ is stalled until $tx3$ completes. When $tx3$ completes, $tx1$, $lockL2$, $lockL3$ and $lockL4$ are all allowed to resume.

4.3 LiT TM-Lock Protection

Like full lock protection, LiT TM-lock protection runs transactions encapsulating locks in isolation. However, LiT TM-lock protection also requires the end programmer to identify locks obtained within any transaction prior to transaction execution (just as LoT TM-lock protection). Unlike LiT full lock protection, TM-lock protection allows non-conflicting LoT locks to execute along side LiT locks, increasing overall system throughput.

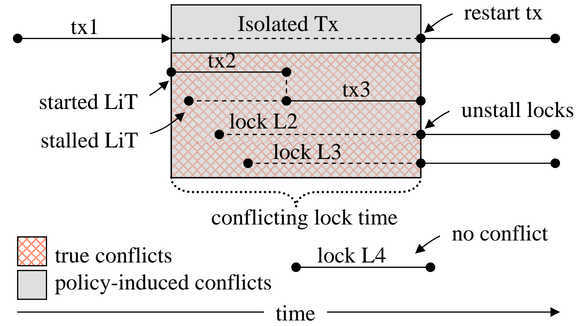


Figure 14. LiT Example of TM-Lock Protection.

As shown in Figure 14 TM-lock protection reduces the overall policy-induced conflicting time to a range closer to the true conflicting time. Since $tx2$ and $tx3$ are true conflicts with $lockL2$ and $lockL3$, $lockL2$ and $lockL3$ must stall while $tx2$ and $tx3$ are executing. However, $lockL4$ does not conflict with either $tx2$ or $tx3$ and as such, should not be stalled while the LiT transactions are in-flight. TM-lock protection correctly identifies this conflict as false, allowing $lockL4$'s execution to be unimpeded by $tx2$ and $tx3$'s execution.

Three problems still exist in the LiT example: (1) $tx1$ is stalled by $tx2$ and $tx3$, (2) $lockL2$ is stalled by $tx3$ and (3) $lockL3$ is stalled by $tx2$. Ideally, none of these stalls should occur as none represent true conflicts. All three false conflicts are avoided by using the following LiT protection policy, LiT TX-lock protection.

4.4 LiT TX-Lock Protection

Like LoT TX-lock protection, the Lit TX-lock protection algorithm allows for the highest throughput of both transactions and locks while requiring the highest level of programmer involvement and system understanding. Unlike both prior LiT algorithms, TX-lock protection allows LiT transactions to run as irrevocable transactions, rather than isolated transactions. This optimization can increase overall system throughput substantially if other revocable transactions can be run along side the LiT transaction.

With LiT TX-lock protection, the programmer specifies locking conflicts for each transaction. In Figure 12's case, the programmer would specify that $tx2$ conflicts with $L2$ and $tx3$ conflicts with $L3$. By specifying true transactional conflicts with locks, the TM system can relax the requirement of running LiT transactions in isolation and instead run them as irrevocable. While no two irrevocable transactions can be run simultaneously, as they may conflict with each other (resulting in a violation of their irrevocable characteristic), other non-irrevocable transactions can be run along side them, improving overall system throughput.

The run-time result of using LiT TX-lock protection is shown in Figure 15. Transaction $tx1$ is able to run without being stalled as it has no conflicts with other transactions or locks. Transaction $tx2$ is run as an irrevocable transaction, rather than as an isolated transaction, allowing $tx1$ to run along side it. Irrevocable transaction $tx3$ is prevented from starting as irrevocable transaction $tx2$ is already in-flight. Likewise, $lockL2$ cannot lock $L2$ since $tx2$ conflicts with $L2$ and allowing $lockL2$ to proceed would require $tx2$ to abort. Since $tx2$ is irrevocable (e.g. unabortable), $lockL2$ is

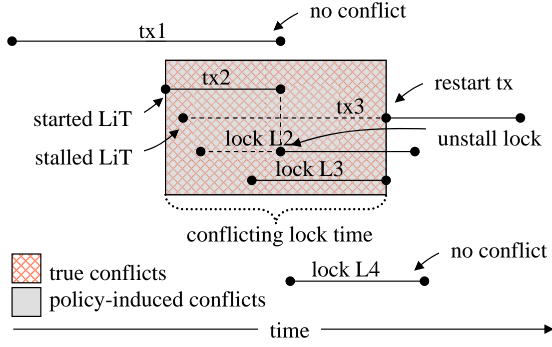


Figure 15. LiT Example of TX-Lock Protection.

stalled. However, $lockL3$ and $lockL4$ start immediately, since neither conflict with any in-flight transaction. When $tx2$ completes, both $tx3$ and $lockL2$ can try to proceed. Transaction $tx3$ is stalled by $lockL3$, but $lockL2$ executes immediately as its conflict with $tx2$ has passed. When $lockL3$ completes $tx3$ begins and runs through to completion.

4.5 LiT Policy Performance

The three LiT policy performances are shown in Figures 16, 17 and 18 (*lower is faster*) demonstrating their execution on the original six threaded example. The tests were run under balanced conditions (Figure 16), transaction-intensive conditions (Figure 17) and lock-intensive conditions (Figure 18).

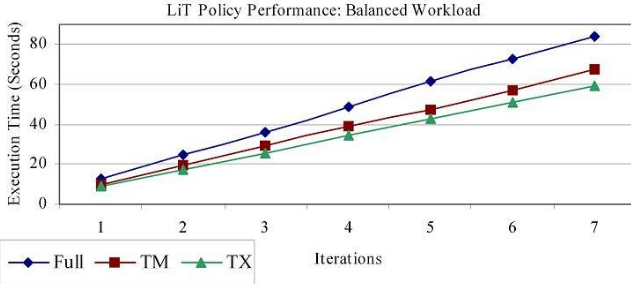


Figure 16. LiT Balanced Workload.

For the balanced workload shown in Figure 16, TM-lock protection outperformed full lock protection by $\approx 25\%$, while TX-lock protection outperformed full lock protection by $\approx 41\% - 43\%$ and TM-lock protection by $\approx 12\% - 15\%$.

The transaction-intensive workloads shown in Figure 17 demonstrate the most even performance distribution between the three policies of the three LiT benchmarks. Full lock protection is outperformed by TM-lock protection by $\approx 12\%$ while TM-lock protection is outperformed by TX-lock protection by $\approx 19\%$. Figure 17 illustrates the gradual performance improvements of the LATM policies from full lock protection (slowest) to TM-lock protection (mid-ranged) to TX-lock protection (fastest).

The LiT lock-intensive workloads shown in Figure 18 report a wide variation of TM-lock protection performance as the number of iterations increase. In addition, not shown by Figure 18, the TM-lock protection policy had a wide variation in performance throughout the iterations. For example, in iteration 3 the TM-lock protection policy completed execution in 200 seconds, 199 seconds and 236 seconds: a wide margin of performance difference. An important open question, not addressed in this work, is whether the current contention manager policy negatively affects the TM-lock protection algorithm under certain non-deterministic situations. Conversely, it is possible that the non-deterministic execution causes more variability in the TM-lock protection policy than the other policies, due to the policy's random thread locking conflicts. Regardless of the answer, as shown in Figure 18, TM-lock protection outperformed full lock protection by $\approx 1\% - 13\%$. TX-lock proteciton outperformed TM-lock protection by $\approx 4\% - 16\%$, while it consistently outperformed full lock protection by $\approx 16\%$.

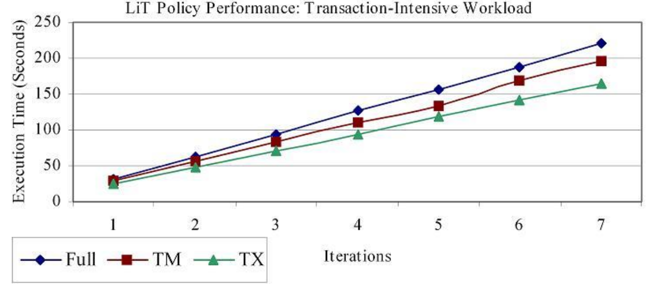


Figure 17. LiT Transaction-Intensive Workload.

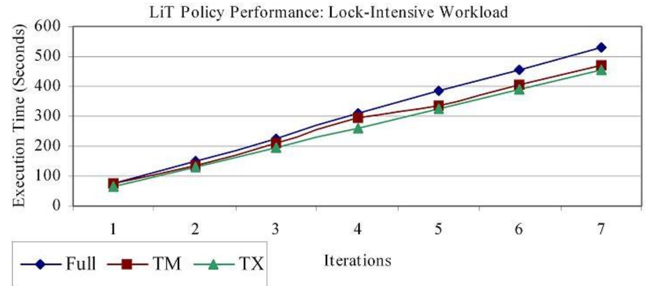


Figure 18. LiT Lock-Intensive Workload.

5. Lock Composition

Any TM system that supports locks inside of transactions must ensure the pessimistic critical sections of the locks inside of transactions are not violated. This is achieved by making the containing transactions either isolated or irrevocable once the lock inside the transaction is obtained. Lemma 2 proves the necessity of the irrevocability characteristic for LiT transactions.

Locks Inside of Transactions Lemma 2. *Any lock L obtained during an in-flight transaction T_{if} requires T_{if} be immediately and permanently promoted to an irrevocable transaction, characterized by $T_{if(irrevocable)} = true$, which cannot be aborted.*

Proof. (Contradiction) Given: threads T_1 and T_2 execute `inc()` and `get()` from Figure 19, respectively and variables $x = 0$, $y = 0$. `++x` and `++y` operations within `inc()` are unguarded direct access variable operations that perform no transactional undo or redo logging operation; these operations are irreversible (e.g. normal pessimistic critical section operations). The atomic property of any transaction T requires all memory operations of T are committed or none are committed.

Execution: T_1 starts transaction Tx_1 ($Tx_{1(irrevocable)} = false$) and completely executes lines 3-7, setting $x = 1$. T_2 executes 13-14, obtains lock $L1$, released by Tx_1 and flags Tx_1 to abort due to its identified lock conflict. T_2 reads $x = 1$ and $y = 0$, unlocks $L1$ and returns. Tx_1 tries to lock $L1$, but instead is required to abort, causing the atomic transactional property of Tx_1 to be violated since ++x has been performed and will not be undone when Tx_1 is aborted.

□

```
1  int x = 0, y = 0;
2
3  void inc() {
4    for (transaction t;;t.restart())
5      try {
6       t.add_conflicting_lock(L1);
7       lock(L1); ++x; unlock(L1);
8       lock(L1); ++y; unlock(L1);
9       t.end(); break;
10     } catch (aborted_tx&) {}
11 }
12
13 void get(int &retX, int &retY) {
14   lock(L1);
15   retX = x; retY = y;
16   unlock(L1);
17 }
```

Figure 19. Violating LiT Lemma 2.

Referring again to Figure 19, now consider the correct execution (following Lemma 2) of threads T_1 and T_2 of inc() and get(), respectively, with variables $x = 0, y = 0$. T_1 starts transaction Tx_1 and executes lines 3-7, setting $x = 1$ and $Tx_{1(irrevocable)} = true$. T_2 executes 13-14, but fails to obtain lock $L1$ even though it is released by Tx_1. T_2 fails to obtain lock $L1$ because in order to do so would require Tx_1 be aborted as it has flagged itself as conflicting with lock $L1$ via t.add_conflicting_lock(L1). Yet, Tx_1 cannot be aborted since $Tx_{1(irrevocable)} = true$. T_2 stalls until Tx_1 completes, setting $x = 1, y = 1$. T_2 then executes, obtaining the necessary locks and returning $x = 1$ and $y = 1$ the atomically correct values for x and y (equivalent values) given transaction Tx_1.

By incorporating both Lemma 1 and 2, locks inside of transactions naturally compose and emit database ACI characteristics; atomic, consistent and isolated. They are atomic (all operations commit): once a lock is obtained inside a transaction, the transaction becomes irrevocable. This ensures that all operations will commit, irrespective of how many locks are obtained within the LiT transaction. They are consistent: no conflicting locks or transactions can run along side the irrevocable LiT transaction, ensuring memory correctness. They are isolated: conflicting locks or transactions are disallowed from running in conjunction with an LiT transaction, preventing its state from being visible before it commits. Even when a lock is released inside an LiT transaction, other threads remain unable to obtain the lock until the transaction commits.

5.1 Criticality of LiT Lock Composition

The composable nature of LiT locks is critical to the incremental adoption of transactions into pre-existing, lock-based parallel software. To understand this, consider a multi-threaded linked list implemented using mutual exclusion locks. The linked list software is mature, thoroughly tested and contains lookup(), insert() and remove() methods. A software designer wishes to extend the linked list's behavior to include a move() operation. The move() operation behaves in the following way: if element A exists in the list and element B does not exist in the list, element A is removed from the list and element B is inserted into the list. With LiT lock composition, the move() operation could be implemented entirely from the previously built locking implementation. Figure 20 demonstrates how this is achieved within the DracoSTM LATM extension.

```
1  bool move(node const &A, node const &B) {
2   // compose locking operations: lookup(), remove()
3   // & insert() in an irrevocable & indivisible tx
4   // to make a new transaction-based move() method
5   for (transaction t;;t.restart())
6     try {
7       t.add_tx_conflicting_lock(list_lock_);
8       // lookup() uses lock list_lock_
9       if (lookup(A) && !lookup(B)) {
10        remove(A); // uses lock list_lock_
11        insert(B); // uses lock list_lock_
12      }
13      else return false;
14
15      t.end(); return true;
16    } catch (aborted_tx&) {}
17 }
```

Figure 20. Move Implemented with LiT Lock Composition.

The move() method in Figure 20 is viewed by other threads as an indivisible operation whose intermediate state is isolated from other threads, even though it contains four disjoint pessimistic critical section invocations (e.g. two lookups, a remove and an insert). Even when list_lock_ is released intermittently during the transaction execution, other threads are prevented from obtaining it until the transaction commits. This behavior ensures each pessimistic critical section within the transaction is prevented from being viewed independently. Once the transaction commits other threads can obtain list_lock_ and view the cumulative affects of the move() operation.

5.2 Understanding LiT Lock Composition

Figure 21 shows a transfer function implemented using LiT lock composition. This example was chosen because the code sample is small enough to be presented in full (not possible with the prior linked list move() example). Figure 21 uses LiT TX-lock protection and contains two locks inside of the same transaction that are subsumed by the transaction. The transaction composes the two separate lock-based critical sections into an atomically viewed and isolated operation.

Consider threads T_1 executing lit_transfer(1), T_2 executing get1() and T_3 executing get2() with $x1 = 0$ and $x2 = 0$ in the time line shown in Figure 22. The dotted vertical lines in the conflicting lock time in Figure 22 demonstrate when T_1's transaction $tx1$ obtains and releases locks $L1$ and $L2$ with regard to T_2 and T_3. Thread T_1 starts transaction $tx1$ and adds $L1$ and $L2$ to $tx1$'s conflicting lock list. Next, $tx1$ locks $L2$ and becomes irrevocable (Lemma 2). Thread T_2 then attempts to lock $L1$, however, since $L1$ conflicts with $tx1$ and $tx1$ is irrevocable, thread T_2 is disallowed from aborting $tx1$ and is therefore prevented from obtaining $L1$, stalling instead. After $tx1$ sets $x2 = -1$ and unlocks $L2$, thread T_3 tries to lock $L1$, however, it is disallowed because $L1$ is on $tx1$'s conflicting lock list and $tx1$ is irrevocable. Thus, thread T_3 is stalled. Transaction $tx1$ then locks $L1$, sets $x1 = 1$ and unlocks

```
int x1 = 0, x2 = 0;

void lit_transfer(int transfer) {
  for (transaction t;;t.restart())
    try {
      t.add_tx_conflicting_lock(L1);
      t.add_tx_conflicting_lock(L2);

      lock(L2); x2 -= transfer; unlock(L2);
      lock(L1); x1 += transfer; unlock(L1);

      t.end(); break;
    } catch (aborted_tx&) {}
}

void get1and2(int& val1, int& val2) {
  lock(L1); lock(L2);
  val1 = x1; val2 = x2;
  unlock(L2); unlock(L1);
}
void get1(int& val) {
  lock(L1); val = x1; unlock(L1);
}
void get2(int& val) {
  lock(L2); val = x2; unlock(L2);
}
```

Figure 21. LiT Transaction and Two Locking Getters.

$L1$. When $tx1$ commits, threads T_2 and T_3 are unstalled and read $x1 = 1$ and $x2 = -1$, respectively, the correct atomic values for the `lit_transfer(1)` operation with lock composition.

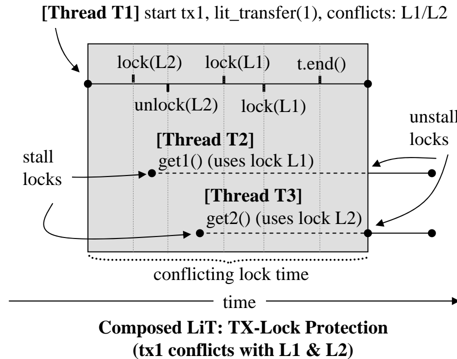


Figure 22. Composed LiT Example using TX-Lock Protection.

In the above scenario, both T_2 and T_3 tried to acquire the unlocked locks $L1$ and $L2$ but failed due to $tx1$'s irrevocability even though the locks themselves were available. The characteristic of disallowing lock acquisition even when locks are available is a primary attribute of LiT transactions. This characteristic is not present in systems which use locks alone, and is a key attribute to enabling lock composition. As demonstrated earlier with LoT transactions, a lock and a transaction conflicting with the same lock cannot both execute simultaneously. LiT transactions use this same behavior and extend it to favor transactions over locks once a lock is obtained inside of a transaction, making it irrevocable. By restricting access of locks to in-flight LiT transactions which have acquired (and then released) them, the system ensures any remaining behavior not yet performed by the transaction will occur prior to other threads obtaining such released locks. This ensures all the pessimistic critical sections within an LiT transaction are executed in isolation and consistently without memory interference from outside locking threads. Note that this was the case when T_3 tried to acquire lock $L2$ even after transaction $tx1$ had completed its operations on its shared memory $x2$.

5.2.1 LiT Lock Identification

LiT TX-lock protection requires that all transaction conflicting locks, those locks used directly inside of transactions, be identified prior to any lock being obtained within the transaction; failure to do so can lead to deadlocks. A simple example of how adding conflicting locks as they are obtained inside of transactions can lead to deadlocks can be derived from Figure 21. If `lit_transfer()` is executed without adding conflicting locks (e.g. no calls to `add_conflicting_lock()`), the following deadlock is possible. Thread T_1 executes lines 3-5, skips lines 6-7 (the conflict calls) and executes line 9 locking $L2$ and adding it to its conflicting lock list. Thread T_2 then executes lines 16 and part of line 17 of `get1and2()`, locking $L1$. Without the transactional code identifying $L1$ as a conflict, the TM system would not disallow T_2 from locking $L1$. The system is now deadlocked. T_2 cannot proceed with locking $L2$ as $L2$ has been added to T_1 conflicting lock list, yet T_1 cannot proceed as lock $L1$ is held by T_2.

These deadlocks scenarios are overcome by requiring the LiT TM-lock or TX-lock protection to list all conflicting locks prior to obtaining any locks within transactions. Attempting to use locks inside of transactions without first identifying them as conflicts causes DracoSTM to throw an exception informing the programmer of the missed conflicting lock and how to correct the error. Recycling the same scenario as above with conflicting locks identified at the start of the transaction avoids deadlocks. For example, thread T_1 adds locks $L1$ and $L2$ to its conflicting lock list. It then executes lines 3-8 and part of line 9, locking $L2$ (but not unlocking it). Thread T_2 executes lines 16 and part of 17, trying to lock $L1$, and sees T_1's transaction as conflicting. Thread T_2 tries to abort T_1's transaction, but is disallowed as the transaction is irrevocable (see Lemma 2). T_2 is therefore stalled prior to obtaining lock $L1$. T_1 continues and executes the remainder of line 9-12, obtaining and releasing lock $L1$ and finally committing. Once T_1's transaction has committed, T_2 is allowed to resume and runs to completion.

6. Conclusion

We presented a library-based lock-aware transactional memory (LATM) system using three different policies for handling transaction-lock interaction. The three policies were found to be useful for different development environments. Full lock protection guarantees correctness at the price of performance, is beneficial when system knowledge is minimal, and requires no code changes. TM-lock protection is useful when locks that conflict with transactions are known, but not to the granularity of the transaction, and benefits the system by improving performance while requiring some additional code. TX-lock protection offers the best performance, but requires the most additional code, and a complete understanding of the system's underlying locking structure to identify locking conflicts on a per transaction basis. We presented experimental results for both LoT and LiT scenarios for the three types of LATM policies.

We showed lock composition is a natural side-effect of an LATM system that implements LiT using irrevocable (inevitable) or isolated transactions. Two LiT examples were given: a high-level example to demonstrate the critical need for LiT composable locks and a more simplified example with complete code for a deeper exploration of LiT transactions. Finally, we discussed how lock acquisition outside of an LiT transaction behaves once a lock within the transaction is released and why conflicting locks must be iden-

tified prior to transactional operations. Both aspects were shown to be critical to the correctness of LiT lock composition.

7. Acknowledgements

We would like to thank David Dice of Sun Microsystems and Nir Shavit of Sun Microsystems and Tel Aviv University for their constant communication and help in clarifying the importance of LATM. We pay special thanks to Michael F. Spear and the rest of the High-Performance Synchronization Group at the University of Rochester for their advice and feedback. Thanks to Mark Holmes and Paul Rogers of Raytheon Company for their help identifying a practical need at Raytheon for the ideas in this paper. Finally, we thank Paul Gottschlich for his numerous edits.

References

[1] A.-R. Adl-Tabatabai, D. Dice, M. Herlihy, N. Shavit, C. Kozyrakis, C. von Praun, and M. Scott. Potential show-stoppers for transactional synchronization. In *PPOPP*, page 55, 2007.

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

[3] A.-R. Adl-Tabatabai, B. T. Lewis, V. Menon, B. R. Murphy, B. Saha, and T. Shpeisman. Compiler and runtime support for efficient software transactional memory. In *PLDI '06: Proceedings of the 2006 ACM SIGPLAN conference on Programming language design and implementation*, pages 26–37, New York, NY, USA, 2006. ACM.

[4] D. Dice, O. Shalev, and N. Shavit. Transactional locking II. In S. Dolev, editor, *Distributed Computing (20th DISC'06)*. Sept. 2006.

[5] 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.

[6] J. E. Gottschlich and D. A. Connors. Brief announcement: Optimizing consistency checking for memory-intensive transactions. In *ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing (PODC) (brief announcement)*, Aug. 2008.

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

[8] L. Hammond, B. D. Carlstrom, V. Wong, M. K. Chen, C. Kozyrakis, and K. Olukotun. Transactional coherence and consistency: Simplifying parallel hardware and software. *IEEE Micro*, 24(6):92–103, 2004.

[9] T. Harris, S. Marlow, S. L. P. Jones, and M. Herlihy. Composable memory transactions. In K. Pingali, K. A. Yelick, and A. S. Grimshaw, editors, *PPOPP*, pages 48–60. ACM, 2005.

[10] 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.

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

[12] 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.

[13] J. E. B. Moss. *Nested Transactions: An Approach to Reliable Distributed Computing*. PhD thesis, 1981.

[14] R. Rajwar and P. A. Bernstein. Atomic transactional execution in hardware: A new high performance abstraction for databases. In *In Position paper for the 10th International Workshop on High Performance Transaction Systems*, 2003.

[15] R. Rajwar and J. R. Goodman. Speculative lock elision: enabling highly concurrent multithreaded execution. In *MICRO*, pages 294–305. ACM/IEEE, 2001.

[16] C. J. Rossbach, O. S. Hofmann, D. E. Porter, H. E. Ramadan, B. Aditya, and E. Witchel. Txlinux: using and managing hardware transactional memory in an operating system. In T. C. Bressoud and M. F. Kaashoek, editors, *SOSP*, pages 87–102. ACM, 2007.

[17] 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.

[18] B. Stroustrup. A brief look at C++0x. *Online: www.artima.com/cppsource/cpp0x.html*.

[19] A. Welc, B. Saha, and A.-R. Adl-Tabatabai. Irrevocable transactions and their applications. In *SPAA*, pages 285–296, 2008.

[20] L. Ziarek, A. Welc, A.-R. Adl-Tabatabai, V. Menon, T. Shpeisman, and S. Jagannathan. A uniform transactional execution environment for java. In *ECOOP*, pages 129–154, 2008.

[21] C. Zilles and D. Flint. Challenges to providing performance isolation in transactional memories. In *Proceedings of the Fourth Workshop on Duplicating, Deconstructing, and Debunking*, pages 48–55, Jun 2005.