An Efficient Lock-Aware Transactional Memory Implementation

Justin E. Gottschlich[†], Jeremy G. Siek[†], Manish Vachharajani[†], Dwight Y. Winkler[‡] and Daniel A. Connors[§]

[†]Department of Electrical and Computer Engineering, University of Colorado at Boulder
[‡]Nodeka, LLC.
[§]Department of Electrical and Computer Engineering, Colorado State University
[†]{gottschl, jeremy.siek, manishv}@colorado.edu, [‡]dwight@nodeka.com, [§]dan.connors@colostate.edu

Abstract

Transactional memory (TM) is an emerging concurrency control mechanism that provides a simple and composable programming model. Unfortunately, transactions violate the semantics of mutual exclusion locks when they execute concurrently. Due to the prevalence of locks, transactions must be made *lock-aware* enabling them to correctly interoperate with locks.

We present a lock-aware transactional memory (LATM) system that employs a unique communication method using local knowledge of locks coupled with granularity-based policies. Our system allows higher concurrent throughput than prior systems because it only prevents *truly* conflicting critical sections from executing concurrently. Furthermore, our system relaxes the prior requirement of transaction isolation when executing conflicting transactional critical sections and instead runs these transactions as irrevocable, improving transaction concurrency. We demonstrate our performance improvements mathematically and empirically.

Our system also advances LATM research in terms of program consistency. This is achieved by detecting potential deadlocks at run-time and aborting the programs that contain them. Prior systems break deadlocks, which reveal partially executed critical sections to other threads, thereby violating mutual exclusion. Because our system disallows deadlocks, it does not suffer from mutual exclusion violations, improving program consistency.

1. Introduction

Transactional memory (TM), as introduced by Herlihy and Moss, is an emerging concurrency control mechanism that provides a simple, efficient, and composable programming model [5, 7, 9, 10, 15, 16]. A shortcoming of transactions, the synchronization mechanism of TM, is that they do not correctly interoperate with locks without special effort. This interoperability failure is magnified by the prevalent use of locks in parallel software [8]. As such, in order for TM to become practical, it must be made *lock-aware* so it can correctly interact with locks [1].

Prior lock-aware transactional memory (LATM) systems require that transactions execute in isolation (1) when they contain locks that cannot be elided away, or (2) when locks are executed concurrently in other threads that could conflict with transactions [18, 20]. In these systems, transactions must execute in isolation because the systems use run-time conflict detection, which is not guaranteed to discover all locking conflicts before transaction execution. The systems must, therefore, overestimate conflicts and require all transactions to execute serially.

Our system uses programmer knowledge, in the form of programmer annotations, to reduce the overestimation of conflicts found in the prior LATMs. The result is improved program performance. The idea of using programmer knowledge to improve performance is not new to transactional memory; transactional boosting and open nesting are based on the same principle [6, 12]. However, to the best of our knowledge, we are the first to use programmer knowledge to improve the efficiency of LATM. The programmer annotations used in our LATM system indicate which locks conflict with (1) all transactions or (2) a specific transaction. Because these annotations identify true conflicts, the resulting program allows many lock-based and transaction-based critical sections to run concurrently, thereby increasing parallel throughput.

Our system also finds potential deadlocks and aborts programs that contain them. Prior systems break deadlocks which can cause inconsistent program behavior as other threads may have visibility into partially executed critical sections. Our solution improves overall program consistency because we avoid these mutual exclusion violations. We make the following technical contributions:

1. For locks outside of transactions we enable greater concurrency between lock-based and transaction-based critical sections than prior work because we only serialize threads with true conflicts between transactions and locks.
2. For locks inside of transactions, we also enable greater concurrency again by only serializing threads with true conflicts, though in this case the transactions that contain locks can be run as irrevocable, rather than isolated as done in prior work, increasing transactional concurrency.
3. Our system disallows deadlocks by detecting them at run-time and aborting the programs containing them. This allows inconsistent program states, found in prior LATMs, to be avoided.

2. Background

When transactions and locks are executed concurrently in non-LATMs, they behave inconsistently due to the differences in their critical section semantics [14, 18, 20]. Mutual exclusion locks use pessimistic critical sections that are limited to one thread of execution [2, 21]. Transactions use optimistic critical sections that support unlimited concurrent thread execution and resolve conflicts in the transaction's commit phase [4, 11, 13]. Optimistic and pes-

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

ICOOOLPS'09 July 6, 2009, Genova, Italy.

Copyright © 2009 ACM [to be supplied]. . . $5.00

simistic critical sections conflict because their semantics differ; an example of this is demonstrated in Figure 1.

```
1     Thread T1                      Thread T2
2
3     lock(L);
4     int tmp = x;                   atomic {
5     x = y;
6                                      int tmp = x;
7                                      x = y;
8     y = tmp;
9                                      y = tmp;
10    unlock(L);                     }
```

Figure 1. Lock and Transaction Swap Violation.

In non-LATMs, threads T_1 and T_2 in Figure 1 behave incorrectly. Both T_1 and T_2 implement a swap function with a lock and a transaction, respectively. Both threads are correct when run in isolation, however, when run together the result is erroneous. Consider an initial state of `x = 1` and `y = 2`. When correctly swapped, `x = 2` and `y = 1`. Thread T_1 starts by setting `tmp = 1` and `x = 2`. Thread T_2 sets `tmp = 2`, `x = 2`. T_1 sets `y = 1`, T_2 then sets `y = 2`. The resulting state (`x = 2, y = 2`) is incorrect.

2.1 Classifying Transaction-Lock Failures

There are five pathological ways transactions and locks can interact, each of which produces a certain type of error. The pathological behaviors identified by Volos et al. are: blocking, livelock, deadlock, early release, and invisible locking [18].

- Blocking occurs when a lock inside a transaction (LiT) cannot be immediately acquired. In certain TMs, transactions must terminate if the thread they are running in is context-switched out by the OS. In such systems, LiTs that do not obtain a lock before being context-switched out are aborted. If these aborts are repeated, forward progress is stalled.
- Livelocks occur when threads try to acquire locks outside of transactions (LoT) via spinning. In some TMs, spinning on locks prevents locks that have been obtained within a transaction from being released, thereby creating a livelock situation when the transaction tries to commit to release the locks.
- Deadlocks occur in a variety of ways through transaction-lock interaction. One example is when a transaction is aborted after a lock within it has been acquired, but before it is released. All non-LATMs are susceptible to transaction-lock deadlock.
- Early release occurs when a transaction releases a lock that was obtained before the transaction began. If the transaction is retried, the lock is released multiple times resulting in an inconsistent program state. Early release also leads to deadlocks.
- Invisible locking occurs in lazy acquire (or deferred update) systems when locks inside of transactions are obtained at commit-time, rather than when the lock operations are executed. This causes locks to become optimistic, altering their (potentially necessary) pessimistic semantics.

2.2 Preventing Transaction-Lock Violations

Locks and transactions conflict because transactions violate the pessimistic critical section semantics of locks. If the mutual exclusion property of locks is not violated by transactions – execution of pessimistic critical sections are limited to a single thread of execution [2] – transactions and locks will behave in a non-pathological manner when run concurrently.

Our system avoids the five pathological behaviors found by Volos et al. in the following way. We prevent deadlock and invisible locking by ensuring transactions do not violate the mutual exclusion property of locks using our granularized LATM policies and local knowledge of these locks (details to follow). Blocking and livelocks are avoided by the implementation as explained in prior work [3]. The final pathological behavior, early release, is deemed illegal and caught at run-time. Further details are provided in Section 5.

3. Related Work

This section briefly presents prior LATM research [18, 20]. The prior systems discover conflicts between locks and transactions at run-time while ours detects conflicts at compile-time. As we demonstrate in Section 5, run-time discovery of *true* locking conflicts in LATMs is insufficient to avoid deadlocks. Because of this, systems that use run-time conflict discovery must be overly pessimistic to ensure program correctness. Since our system uses programmer annotations to identify conflicts at compile-time, we achieve improved system performance as shown in Section 6.

3.1 TxLocks

The work of Volos et al. identified the five transaction-lock pathologies presented in Section 2. To overcome these pathologies, Volos et al. modified OpenSolaris and implemented what they call *TxLocks*. In [18], Volos et al. explain how they prevent the pathological transaction-lock behaviors.

A key difference between TxLocks and our system is that TxLocks does not *disallow* deadlocks as our approach does, instead it breaks deadlocks when they occur at run-time. Unfortunately, breaking deadlocks can have negative side-effects. Deadlocks occur when two or more processes hold a resource the other requires to make forward progress. To break deadlocks, resources are stolen from a process and given to another. However, critical section operations may have already been partially executed by a process that is stolen from and stalled. Once a thread's resources have been stolen, the partial effects of its operations may be seen by other threads resulting in inconsistent program states. As such, breaking deadlocks is unideal.

3.2 P-SLE and Atomic Serialization

Ziarek et al.'s LATM is implemented in Java and introduces two concepts: *pure-software lock elision* (P-SLE) and *atomic serialization*. With the exception of the early release behavior, all pathologies are avoided directly with P-SLE because it converts locks into transactions; since no locks exist, the pathologies cannot occur. The early release behavior is not possible in Ziarek et al.'s implementation because its signature is illegal in Java.

However, as noted by the authors, locks cannot always be converted into transactions. One such example is when a lock-based critical section performs I/O which must be run without transactional interference. P-SLE handles this by reinstating all locks and using a single global lock to serialize transaction execution. This behavior, called *atomic serialization*, guarantees the mutual exclusion property is maintained by requiring that each transaction obtain a shared global lock to execute [20]. Atomic serialization is a suboptimal solution because it temporarily requires that all transactions serialize their execution; an overly pessimistic requirement.

4. Locks Outside of Transactions (LoT)

In this section, we discuss locks outside of transactions or LoTs. LoTs are scenarios where a lock-based critical section is executed in one thread while a transaction-based critical section is concurrently executed in another thread. LoTs require special handling to ensure concurrently executed locks and transactions do not access

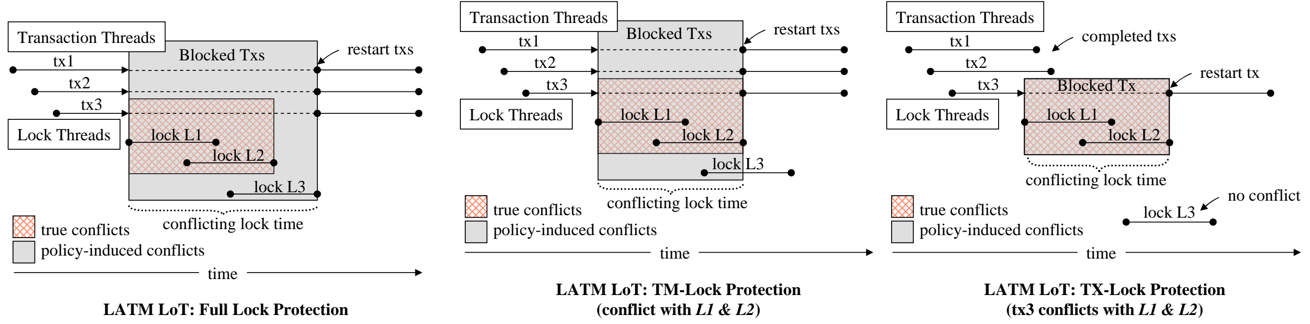

Figure 2. LoT Full (TxLocks), TM-, and TX-Lock Protection.

the same shared memory, since such behavior could result in inconsistent execution as demonstrated in Section 2.

Figure 3 creates a running LoT example used throughout this section. In this example, we create six threads that simultaneously execute six different functions. Three of the functions use transactions: thread T_1 runs `tx1()`, thread T_2 runs `tx2()`, and thread T_3 runs `tx3()`. The other three functions use locks: thread T_4 runs `lock1()`, thread T_5 runs `lock2()`, and thread T_6 runs `lock3()`.

In our example, thread T_3's function `tx3()` conflicts with both thread T_4 and thread T_5's locking functions (`lock1()` and `lock2()`). Threads T_1, T_2 and T_6 do not exhibit any conflicts, thus their calls to `no_conf()`, but are necessary to demonstrate how the different LoT policies behave.

```
1  void tx1() { atomic(t) { no_conf(); } }
2  void tx2() { atomic(t) { no_conf(); } }
3  void tx3() {
4    atomic(t) {
5      for (int i=0; i < N; ++i) {
6        ++t.w(arr1[i]).value();
7        ++t.w(arr2[i]).value();   }
8    } end_atom
9  }
10 int lock1() {
11   lock(L1); int sum = 0;
12   for (int i=0; i < N; ++i) sum += arr1[i];
13   unlock(L1); return sum;
14 }
15 int lock2() {
16   lock(L2); int sum = 0;
17   for (int i=0; i < N; ++i) sum += arr2[i];
18   unlock(L2); return sum;
19 }
20 int lock3() {lock(L3);no_conf();unlock(L3);}
```

Figure 3. Six Threaded LoT Example.

4.1 LoT Full Lock Protection and TxLocks

The largest granularity policy for transaction-lock cooperation, full lock protection, enforces all transactions to commit or abort before a lock's critical section is executed. LoT full lock protection is equivalent to Volos et al.'s TxLocks [18]. Locks are protected from violations from transactions because transactions are not allowed to run alongside them. The system stalls transactions until all lock-based critical sections are complete.

LoT full lock protection is overly pessimistic. Because no information is provided regarding potentially conflicting shared memory access within transactions, each time a lock-based critical section is executed, no transactions may be processed. The maximum concurrent lock and transaction throughput achievable by LoT full lock protection (and TxLocks) at any given point in time is:

$$m(LoT_{fl}) = L_n \oplus T$$

L_n is the maximum number of lock-based critical sections that do not conflict with one another and T is the maximum number of transactions that can be executed. Because of the way LoT full lock protection behaves, only one type of critical section can be executed, locks or transactions, but not both (as denoted by exclusive-or $\oplus$). Figure 2 presents a visual model of LoT full lock protection under the six threaded model. $T_1 - T_3$ are blocked for the duration of the critical sections of $T_4 - T_6$, even though T_6's critical section does not interfere with any of the transactions.

4.2 LoT TM-Lock Protection

TM-lock protection, our mid-level granular policy, requires some program knowledge of lock-transaction conflicts, but results in increased concurrent throughput. TM-lock protection works in the following way. The programmer identifies and specifies locks that can conflict with *any* transaction. These programmer identified conflicts inform the TM system that if critical sections of the specified locks are executed concurrently with a transaction, an inconsistent program state could arise. Therefore, when a conflicting lock is acquired at run-time, the TM system aborts or commits all in-flight transactions and then blocks all transactions until the conflicting lock-based critical section is completed. Locks that do not conflict with any transaction do not cause stalls, an improvement over full lock protection. TM-lock protection can be expressed as follows:

$$m(LoT_{tm}) = L_{nl} \oplus (L_{na} + T)$$

L_{nl} is the total number of locks that do not conflict with one another, but do conflict with transactions. L_{na} is the total number of locks that do not conflict with one another and do not conflict with transactions. T is the maximum number of transactions that can be executed. In the six threaded example, TM-lock protection avoids unnecessary transaction stalling when T_6 is executing. As seen in Figure 2, TM-lock protection shortens the overall TM run-time (compared to full lock protection) by allowing $T_1 - T_3$ to restart their transactions as soon as `L2`'s critical section is completed.

4.3 LoT TX-Lock Protection

TX-lock protection, our smallest granular policy, requires local knowledge of locking conflicts per transaction, but yields the highest potential concurrent throughput. By identifying conflicting locks per transaction, the system only stalls for *true* conflicts as shown in Figure 2. LoT TX-lock protection increases potential concurrency, compared to LoT TM-lock protection, because it only

stalls transaction and lock execution when a true conflict exists between them. The following expression represents the maximum concurrent execution of locks and transactions at any given point in time for LoT TX-lock protection:

$$m(LoT_{tx}) = C_{lt} + L_{na} + T_{na}$$

C_{lt} is the largest system selected set of locks and transactions that can be run concurrently without containing any overlapping conflicting critical sections. L_{na} is the total number of locks that do not conflict with one another and have not been flagged as conflicting with any transaction. T_{na} are the transactions that can be executed which do not conflict with any locks. In the six threaded example, TX-lock protection stalls thread T_3 when the critical sections of `L1` and `L2` are executing. In this example, TX-lock protection's policy-induced conflict time is *equal* to the true conflict time, resulting in the highest concurrent critical section execution possible.

5. Locks Inside of Transactions (LiT)

In this section, we discuss locks inside of transactions or LiTs. LiTs are scenarios where a lock's pessimistic critical section is executed partially or completely inside a transaction. We only support two of three possible LiT scenarios: (1) locks entirely inside a transaction or (2) locks beginning inside a transaction and ending after it. The third scenario, (3) locks beginning before a transaction and ending inside it (known as early release [18]), is disallowed because it can cause deadlocks. This is a fundamental departure of our design from prior work. P-SLE does not deal with early release because it is illegal in Java. TxLocks allows it and attempts to *break* the deadlocks it creates. We believe breaking deadlocks is unacceptable as it can lead to side-effects explained in Section 3.1.

5.1 Early Release Deadlocks in LiTs

To demonstrate how early release can cause deadlocks, consider Figure 4. Thread T_1 obtains lock L_1 while thread T_2 starts transaction Tx_2 which becomes irrevocable, meaning the transaction *must* commit (details to follow). Tx_2 tries and fails to obtain lock L_1 because thread T_1 has it. Thread T_1 then begins transaction Tx_1. This is where the complexity begins.

```
1      Thread T1                      Thread T2
2
3      lock(L1);                      atomic(Tx2) {
4                                       // irrevocable tx
5                                       lock(L1);
6      atomic(Tx1) {
7        ...
8        unlock(L1);
9        ...                            unlock(L1);
10     }                              }
```

Figure 4. Early Release Deadlock.

Since lock L_1 is released inside of Tx_1, Tx_1 must be made irrevocable, otherwise L_1's critical section could be retried, breaking the mutual exclusion property. However, since Tx_2 is already an irrevocable transaction and at most only one irrevocable transaction can execute, Tx_1 cannot be made irrevocable. Two irrevocable transactions cannot execute concurrently because they are not guaranteed to be free of conflicts between each other [17,19]. Therefore, Tx_1 must stall until irrevocable transaction Tx_2 completes. Tx_2 must stall until lock L_1 is released. Lock L_1 cannot be released until Tx_1 completes. The system is now deadlocked.

For brevity, we have omitted the algorithmic details of identifying and prohibiting early release, but they can be found in our open source implementation of TBoost.STM (formerly DracoSTM). Our system is the first that we are aware of to (1) identify the early release deadlock scenario, (2) detect early release at run-time, and (3) disallow the deadlocks early release creates, increasing program correctness.

5.2 Irrevocable and Isolated Transactions

Locks placed inside of transactions must behave like normal locks; the mutual exclusion property must be maintained to ensure correctness. In order to support this property and because locks do not have failure atomicity (i.e., they cannot be retried), the transactions containing locks must become irrevocable. Irrevocable transactions are transactions that cannot be aborted. Irrevocable (also known as inevitable) transactions are thoroughly investigated by Welc et al. and Spear et al [17, 19]. We extend the practical use of irrevocable transactions to enable pessimistic critical sections within transactions, and create *composable locks* within transactions. We also define a new type of transaction, which we call an *isolated transaction*. Isolated transactions cannot be aborted, but they also have the property that no other transaction can concurrently execute alongside them.

5.3 Criticality of LiT Lock Composition

LiT lock composition is critical to the incremental adoption of transactions into existing lock-based software. Consider a multi-threaded linked list implemented with locks. The software is mature, thoroughly tested, and contains `lookup()`, `insert()` and `remove()` methods. A developer wishes to extend the linked list's behavior with 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 and element B is inserted. With LiT lock composition, the `move()` operation is implementable entirely from the previous locking software. Figure 6 demonstrates this using our LATM extension to TBoost.STM.

```
1 atomic(t) {
2   t.lock_conflict(list_lock_);
3   if (lookup(A) && !lookup(B)) { remove(A); insert(B); }
4   else return false;
5 } end_atom
6 return true;
```

Figure 6. Move Implemented with LiT Lock Composition in TBoost.STM.

The move method in Figure 6 is viewed by other threads as an indivisible operation whose intermediate state is isolated, even though it contains four disjoint pessimistic critical sections (e.g. two lookups, a remove and an insert). When `list_lock_` is released during the transaction execution, other threads cannot obtain it until the transaction commits. This behavior, called deferred release, ensures locks obtained inside transactions are not released until the transaction commits.

5.3.1 Understanding LiT Lock Composition

Figure 7 shows the transfer function as implemented using LiT lock composition, uses LiT TX-lock protection, and contains two locks inside of the same transaction that are subsumed by the transaction. The transaction composes 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()`. Let `x1 = 0` and `x2 = 0` as shown in Figure 8. The dotted vertical lines in Figure 8 illustrate when the transactions obtain and release their corresponding locks. Thread T_1 starts transaction $tx1$ and adds $L1$ and $L2$ as conflicting locks. Transaction $tx1$ locks $L2$ and becomes irrevocable. Thread T_2 attempts to acquire $L1$. Even though $L1$ is available,

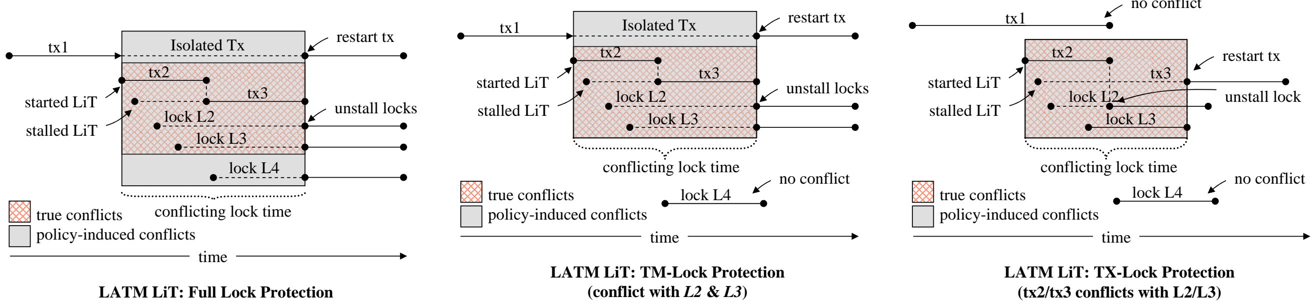

Figure 5. LiT Full (Atomic Serialization), TM-, and TX-Lock Protection.

```
1  void lit_transfer(int transfer) {
2    atomic(t) {
3        t.lock_conflict(L1);
4        t.lock_conflict(L2);
5        lock(L2); x2 -= transfer; unlock(L2);
6        lock(L1); x1 += transfer; unlock(L1);
7    } end_atom
8  }
9  void get1and2(int& val1, int& val2) {
10   lock(L1); lock(L2);
11   val1 = x1; val2 = x2;
12   unlock(L2); unlock(L1);
13 }
14 void get1(int& val) { lock(L1); val = x1; unlock(L1); }
15 void get2(int& val) { lock(L2); val = x2; unlock(L2); }
```

Figure 7. LiT Transaction and Two Locking Getters.

thread T_2 is prevented from acquiring it since it conflicts with irrevocable transaction $tx1$. As such, thread T_2 is stalled. After $tx1$ unlocks $L2$, thread T_3 tries to lock $L1$. However, since $L1$ conflicts with $tx1$ and $tx1$ is irrevocable, thread T_3 is stalled from acquiring lock $L1$. Transaction $tx1$ locks $L1$, sets `x1 = 1`, unlocks $L1$ and completes its transfer function. Threads T_2 and T_3 are then unstalled and read `x1 = 1` and `x2 = -1`, respectively, the correct atomic values for the `lit_transfer(1)` operation.

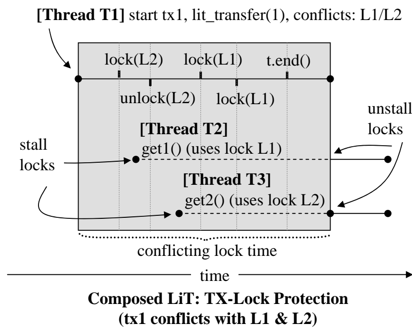

Figure 8. Composed LiT Example using TX-Lock Protection.

5.3.2 LiT Identification

LiT TX-lock protection requires that locks acquired inside of transactions be listed *before* any locks are acquired within the transaction; failure to do so can lead to a deadlock. [1] This requirement is the fundamental reason why TxLocks and Atomic Serialization cannot execute LiTs and locks concurrently, and why our system is able to outperform these solutions: there are no known dynamic analysis techniques which are guaranteed to find all possible locks in an execution path.

The following example illustrates how a deadlock can occur if the complete list of conflicting locks is not known prior to obtaining locks within a transaction. This example is derived from Figure 7. Consider executing `lit_transfer()` without adding the conflicting locks (e.g., no calls to `add_conflicting_lock()`) to the transaction. Thread T_1 executes lines 1-2, skips lines 3-4 (the conflict calls) and executes line 5 locking $L2$ and adding it to its conflicting lock list. [2] Thread T_2 then executes line 10 and part of line 11, 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 $L2$ is held by T_1, yet T_1 cannot proceed as lock $L1$ is held by T_2.

As noted in Section 3, these deadlock 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. This is a unique feature of our implementation. The prior research that attempts to identify these conflicts dynamically, cannot avoid such conflicts because they add locks as conflicts as they are encountered. As demonstrated in the above deadlock scenario, this is insufficient. To overcome this limitation, the prior solutions using run-time conflict discovery only allow a single lock or transaction execute at a time. Our system improves on this by using programmer annotations of lock conflict identification, which results in increased concurrent throughput by allowing transactions and locks to execute simultaneously, even when transactions have locks within them (LiTs).

5.4 LiT Policies

As seen in Figure 10, we use a six threaded example to demonstrate the different LiT policies. 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, indicated by the call to `no_conf()`. Thread T_2 has a true conflict with thread T_4 while thread T_3 has a true conflict with thread T_5. We use the following staggered start time: T_1, T_2, T_3, T_4, T_5 and finally T_6. The LiT threads are labeled based on the locks they acquire: thread `tx1` uses lock L_1, thread `tx2` uses lock L_2 and thread `tx3` uses lock L_3. We do the same for locking threads: thread `lock L2` use lock L_2, thread `lock L3` uses lock L_3 and thread `lock L4` uses lock

[1] LiT TM-lock protection achieves this since locks conflicting with transactions must be listed before running any transaction.

[2] Recall that LiTs are not released until the transaction commits/aborts. So, lock $L2$ is only released when the transaction commits/aborts.

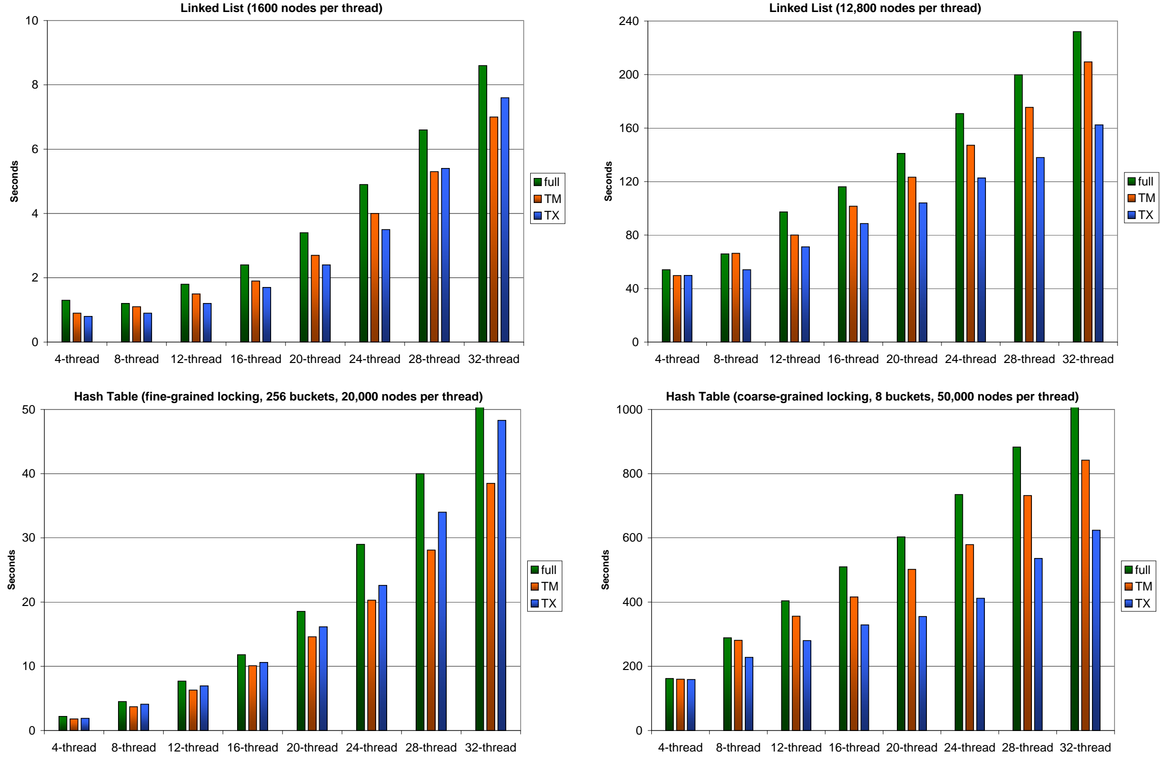

Figure 9. Linked List and Hash Table Benchmarks: LoT Lock Protection Policies.

L_4. This naming convention makes identifying conflicts in prose easier: `tx2` conflicts with `lock L2`, `tx3` conflicts with `lock L3`, while `tx1` and `lock L4` do not have any conflicts (no matching number).

```
void tx1() { atomic(t) { no_conf(); } }
void tx2() {
  atomic(t) {
    t.lock_conflict(L2); inc2();
  } end_atom
}
void tx3() {
  atomic(t) {
    t.lock_conflict(L3); inc3();
  } end_atom
}
void inc2() { lock(L2); ++g2; unlock(L2); }
void inc3() { lock(L3); ++g3; unlock(L3); }
void inc4() {lock(L4);no_conf();unlock(L4);}
```

Figure 10. Six Threaded LiT Example.

5.4.1 LiT Full-Lock Protection and Atomic Serialization

LiT full lock protection does not require any knowledge of locking conflicts to execute correctly. It ensures correctness by disallowing the execution of any lock-based or transaction-based critical section while the LiT transaction is executing. LiT full lock protection assumes that other transactions may acquire locks within them and must therefore be disallowed from executing until the LiT transaction is complete. LiT TM-lock protection behaves like Ziarek et al.'s atomic serialization [20]. LiT full-lock protection's maximum concurrent throughput at any given moment in time, while executing a LiT transaction is expressed as follows:

$$m(LiT_{fl}) = t_l \oplus L_n \oplus T_{nl}$$

t_l is a single transaction that acquires a lock inside of it. L_n is the maximum number of lock-based critical sections that do not conflict with one another. T_{nl} is the maximum number of transactions that can be executed which do not have locks inside of them. LiT full lock protection can only support the execution of one type of critical section: a LiT transaction, non-conflicting locks or non-LiT transactions (as denoted by the exclusive-or $\oplus$).

5.4.2 LiT TM-Lock Protection

LiT TM-lock protection improves on the performance of LiT full lock protection by using knowledge of locks, in the form of programmer annotations, to identify locks that may conflict with *any* transaction. By identifying conflict locks, those locks that do not conflict with the LiT transaction can be run concurrently, increasing overall system throughput. LiT TM-lock protection requires the programmer to specify locks that are obtained inside of transactions. When transactions obtain *any* lock inside of them, all of the locks listed as conflicting are prevented from being obtained until the LiT transaction completes. LiT TM-lock protection requires LiT transactions to be run in isolation, because other transactions may attempt to acquire a lock that would conflict with the LiT. The maximum concurrent throughput of TM-lock protection at a specific instance in time while executing a LiT is:

$$m(LiT_{tm}) = (t_l + L_{nt}) \oplus (L_n + T_{nl})$$

t_l is a single transaction that acquires a lock inside of it. L_{nt} is the maximum number of lock-based critical sections that do not conflict with one another and do not conflict with t_l. L_n is the maximum number of locks that do not conflict with each other, but do conflict with t_l. T_{nl} is the maximum number of transactions

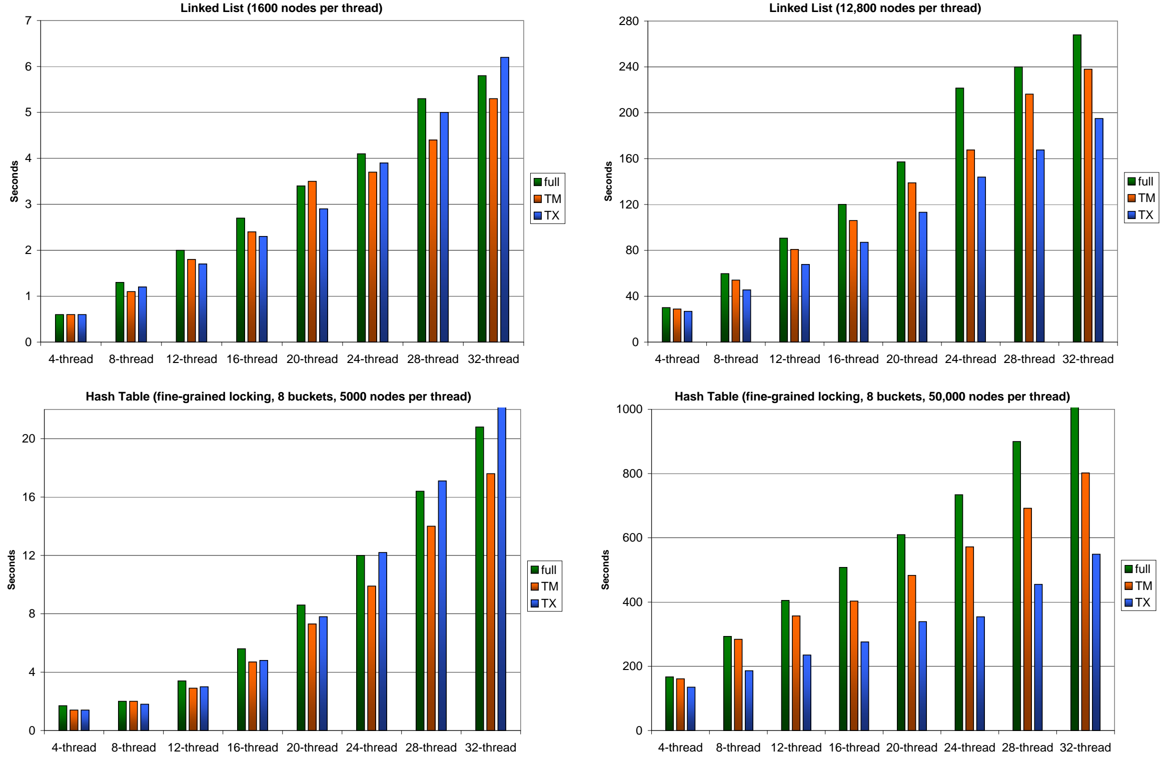

Figure 11. Linked List and Hash Table Benchmarks: LiT Lock Protection Policies.

that can be executed which do not have locks inside of them and do not conflict with L_n. TM-lock protection is an improvement over LiT full lock protection since non-conflicting locks can be executed alongside a LiT transaction or non-conflicting locks can be executed alongside non-conflicting transactions, as illustrated with Figure 5. TM-lock protection correctly identifies the false conflict of `lock L4` with `tx2` and `tx3` as false and allows `lock L4` to execute while `tx2` and `tx3` are in-flight.

5.4.3 LiT TX-Lock Protection

LiT TX-lock protection offers the highest potential concurrent throughput of the LiT policies, but requires explicit local knowledge of locks. TX-lock protection relaxes the requirement of LiT transaction execution, allowing LiT transactions to run as *irrevocable* rather than isolated. This optimization allows other revocable transactions to be run concurrently alongside a LiT transaction, a significant step forward in improving concurrency for transaction execution in the presence of locks.

With LiT TX-lock protection, the programmer specifies which locks are used within each transaction, requiring specific local knowledge of locks to achieve its efficiency. The TM system can then relax the requirement of running LiT transactions as isolated and instead run them as irrevocable. While only one irrevocable transaction can be run at a time, other revocable transactions can be run alongside an irrevocable transaction, improving potential transaction concurrency over Ziarek et al.'s atomic serialization which requires only one transaction to be executed at a time [20]. The maximum concurrent throughput LiT TX-lock protection can achieve at any moment in time while executing a LiT transaction is expressed as follows:

$$m(LiT_{tx}) = (t_l + L_{nt} + T_r) \oplus (L_n + T_{nl})$$

All variables of LiT TX-lock protection are the same as LiT TM-lock protection except TX-lock protection supports T_r. T_r are the maximum number of *revocable* transactions that can be executed concurrently alongside t_l as long as they do not conflict with the set of locks in L_{nt}. As shown in Figure 5, the true conflict time is *equal* to the TX-lock protection policy-induced conflict time. This demonstrates that with the use of local information, in conjunction with small granularity TM policies, allows TX-lock protection to function in a mathematically *optimal* manner.

6. Experimental Results

In this section we present our performance metrics. All benchmarks were gathered on a 1.0 GHz Sun Fire T2000 supporting 32 concurrent hardware threads and 32 GB RAM. For all benchmarks, the x-axis shows the number of active threads and the y-axis shows the total execution time in seconds. Lower total time is faster (e.g., the smaller the bar the better).

LoT Policy Performance Figure 9 displays the execution time of the LoT policies on a 4-threaded through 32-threaded linked list and hash table experimental model. Each benchmark was run using full (left bar), TM- (middle bar), and TX-lock (right bar) protection. The benchmarks use a 4-threaded model that is incremented in multiples up to 32 threads. In the basic 4-threaded model, three containers (linked list or hash tables) are populated. Thread T_1 populates container L_1 with locks. Thread T_2 populates container L_2 with transactions. Container L_3 is concurrently populated by thread T_3 with locks, and thread T_4 with transactions.

LiT Policy Performance Figure 11 displays the execution time of the LiT policies on a 4-threaded through 32-threaded experimental model. In the recurring basic 4-threaded model from the LoT experiments, three containers (linked lists or hash tables) are populated.

The container population is identical to the LoT benchmarks. However, for the LiT benchmarks, a composed LiT transaction is added which first performs a lock-based `lookup()` operation, followed by a lock-based `insert()` operation.

6.1 Performance Summary

For all eight LoT and LiT benchmarks, half of the benchmarks demonstrate TX-lock protection outperform TM-lock protection while the other half demonstrate TM-lock protection outperform TX-lock protection. We intentionally chose these benchmarks to demonstrate that TM-lock protection *can* outperform TX-lock protection under certain circumstances. Two factors contribute to this: (1) critical section execution time compared to algorithmic operational time and (2) the number of active threads and critical sections that must be analyzed compared to the number of existing false conflicts that exist among the critical sections.

In particular, while TX-lock protection supports wider potential concurrent throughput than the other lock protection policies, as demonstrated from the equations in Section 4 and 5, this added concurrency requires TX-lock protection to identify the true conflicts that exist between locks and transactions. The identification of such conflicts incurs algorithmic operational overhead not found when performing a rudimentary pass of conflicts as is done in full or TM-lock protection. Therefore, in cases where workloads have critical sections that execute faster than the time it takes the TX-lock protection algorithm to identify true conflicts, TM-lock protection will outperform TX-lock protection.

As thread and critical section count grows, more work is needed to identify true conflicts. This is demonstrated in our benchmarks with low overall workloads; as the number of threads grows, TX-lock protection's performance gradually degrades. However, for the larger workload benchmarks, the algorithmic time TX-lock protection needed to identify true conflicts is minimal compared to the critical section execution time, resulting in a performance improvement for TX-lock protection as the number of threads increase.

Finally, program performance is based on the existence of false locking conflicts that full and TM-lock protection cannot detect. When no false conflicts exist, full lock protection will outperform TM- and TX-lock protection. In general, we believe TX-lock protection will outperform full and TM-lock protection, but some corner cases do exist, as demonstrated in our benchmarks, that result in inferior TX-lock protection performance when compared to the other policies due to TX-lock protection's algorithmic overhead.

7. Conclusion

This paper presented our unique LATM system using programmer knowledge of locks in conjunction with granularity-based policies. Our LATM system improves the performance of prior research in two ways: (1) we increase the potential for transaction and lock concurrency by serializing only truly conflicting critical sections and (2) since our programmer annotations only identify true conflicts, we can relax the prior requirement of transaction isolation and instead run pessimistic transactions as *irrevocable*, which enables wider transaction concurrency. Our solution also improves program correctness by *disallowing* deadlocks. We find deadlocks at run-time and abort the programs that contain them, circumventing the inconsistent program states that arise from breaking deadlocks. We presented mathematical expressions and empirical benchmarks that illustrate the performance benefits of our system.

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] E. W. Dijkstra. Solution of a problem in concurrent programming control. *Commun. ACM*, 8(9):569, 1965.

[3] 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.

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

[5] 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.

[6] M. Herlihy and E. Koskinen. Transactional boosting: a methodology for highly-concurrent transactional objects. In *Proceedings of the Symposium on Principles and practice of parallel programming*, New York, NY, USA, 2008. ACM.

[7] 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.

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

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

[10] V. J. Marathe and M. Moir. Toward high performance nonblocking software transactional memory. In *PPoPP '08: Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, pages 227–236, New York, NY, USA, 2008. ACM.

[11] 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.

[12] J. E. B. Moss and A. L. Hosking. Nested transactional memory: model and architecture sketches. *Sci. Comput. Program.*, 63(2), 2006.

[13] 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.

[14] 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.

[15] 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.

[16] 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 Symposium on Principles and Practice of Parallel Programming*, Feb. 2009.

[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] H. Volos, N. Goyal, and M. M. Swift. Pathological interaction of locks with transactional memory. In *ACM SIGPLAN Workshop on Transactional Computing*, February 2008.

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