

Optimizing the Concurrent Execution of Locks and Transactions

Justin E. Gottschlich JaeWoong Chung

Programming Systems Lab (PSL), Intel Corporation
{justin.e.gottschlich, jaewoong.chung}@intel.com

Abstract. Transactional memory (TM) is a promising alternative to mutual exclusion. In spite of this, it may be unrealistic for TM programs to be devoid of locks due to their abundant use in legacy software systems. Consequently, for TMs to be practical they may need to manage the interaction of transactions and locks when they access the same shared-memory. This paper presents two algorithms, one coarse-grained and one fine-grained, that improve the state-of-the-art performance for TMs that support the concurrent execution of locks and transactions. We also discuss the programming language constructs that are necessary to implement such algorithms and present analyses that compare and contrast our approach with prior work. Our analyses demonstrate that, *(i)* in general, our proposed coarse- and fine-grained algorithms improve program concurrency but *(ii)* an algorithm’s concurrent throughput potential does not always lead to realized performance gains.

1 Introduction

Transactional memory (TM) [6, 11] is a promising alternative to mutual exclusion because it simplifies parallel programming by moving some of the complexity of shared memory management away from the programmer’s view. While TM shows promise for future software, most TMs have undefined behavior when locks and transactions are used to concurrently synchronize the same shared-memory. This creates a notable void in TM applicability for programmers who wish to use transactions in legacy software that already use locks.

Zyulkarov et al. explored one possible solution to this problem by converting all the locks used in the Quake game server, a large-scale multiplayer software engine [1], to transactions [19]. During this process they encountered significant transition challenges, such as unstructured locking (i.e., non-block-structured critical sections), I/O and system calls, error handling, privatization and thread local memory, and compensating and escape actions necessary to handle up to nine levels of dynamic nesting. They also found that only using transactions resulted in reduced performance when compared to only using locks or using a combination of the two. These results seem to indicate that converting all locks to transactions may be unrealistic for all but expert parallel programmers and, if such a conversion were successful, it might result in a performance degradation.

Another alternative, such as that proposed by Volos et al.’s TxLocks [15] and Ziarek et al.’s P-SLE and atomic serialization [17], is to provide support for transactions and locks so that they can safely access the same shared-memory. In this paper, we focus on this approach and present a system that improves performance over the aforementioned research. A difference in our approach and the prior work of Volos et al. and Ziarek et al. is that our system extends the programming language constructs for locks and transactions so that they contain static (compile-time) information about the conflicts that persist between them, while prior systems deduce conflicts between locks and transactions dynamically (run-time). We demonstrate that our static extensions reduce the number of false conflicts produced at run-time, resulting in improved concurrent throughput.

Throughout this paper we gradually extend and refine the notion of concurrent lock and transaction execution. Our algorithms, one coarse-grained and one fine-grained, manage two general cases of conflicts: (i) non-nested cases, when locks and transactions that have no nesting are executed side-by-side and (ii) nested cases, when locks are nested within transactions and transactions or locks execute along side such transactions.¹

The algorithms require two programming interface enhancements that are similar in structure, but different in purpose, to those proposed by Usui et al. for adaptive locks [14]. We augment the `atomic` transaction block so it takes a single parameter, `locks []`, which is a list of locks that conflict with the transaction. We also introduce a `TmLock` that behaves like a typical mutex and additionally communicates with the TM subsystem before it is acquired and after it is released so the TM system can manage conflicts of, and the forward progress between, `TmLocks` and transactions. We also present an interesting finding, which extends the prior findings of Gottschlich et al. [4], and reflects a counterintuitive result. Our benchmarks show that our coarse-grained policy is *always* faster than the prior systems, while our fine-grained policy is usually faster, but in some cases it can be notably slower (up to $\approx 2\times$).

This paper makes the following technical contributions.

1. We present two algorithms, one coarse-grained and one fine-grained, that allow for the concurrent execution of locks and transactions in the same program and improve concurrent throughput beyond the state-of-the-art.
2. We propose two new TM language constructs: `TmLock` and an extended `atomic` block structure that are used statically (compile-time) to capture the conflicts between locks and transactions.
3. We include a qualitative analysis that precisely captures the potential concurrent throughput of existing systems compared to our algorithms.
4. Our experimental results show that our fine-grained algorithm yields up to $\approx 2.0\times$ performance improvements over prior systems but can sometimes degrade performance. Our coarse-grained algorithm yields up to $\approx 1.5\times$ performance improvements and never performs worse than prior systems for our tested benchmarks.

¹ A third case, when transactions are nested within locks, is not discussed as Volos et al. [15] and Gottschlich et al. [4] show it does not require special effort.

2 Background and Related Work

When transactions and locks are executed concurrently, they can behave inconsistently due to the differences in their critical section execution [10, 15, 17]. Mutual exclusion locks generally use pessimistic critical sections that are limited to one thread of execution [3, 18]. Transactions can use optimistic critical sections that support unlimited concurrent thread execution and resolve conflicts at various points during transaction execution [7, 8]. This difference can incur correctness issues (e.g., transactions being aborted after executing a nested locked region with I/O operations) and cause pathological interferences (e.g., blocking, livelock, and deadlock) [15].

2.1 Conflicts Between Locks and Transactions

For transactions and locks to execute concurrently, the notion of when they conflict must be understood. A lock *conflicts* with a transaction (and vice versa) when both access the same memory location and at least one of those accesses is a write. Although this notion of a conflict is principally the same as that for transactions alone, the conflict resolution [5, 12] and concurrent execution guarantees may be notably different. For this paper, we assume mutually exclusive critical sections are not failure atomic and they execute pessimistically, that is, without write buffering or speculative lock elision [9]. Therefore, conflicts that arise between transactions and a given mutex must be identified before the mutex’s critical section is executed.

TxLocks Volos et al. propose a transaction-aware lock primitive, TxLock, to handle the conflicts between locks and transactions without special hardware support [15]. When used outside of a transaction, TxLocks execute pessimistically and no information is provided from TxLocks to the transactions that might concurrently execute alongside them. This is done to minimize the programmer’s burden of using TxLocks. TxLocks must therefore assume that all transactions can conflict with any TxLock and, likewise, prohibit TxLocks and transactions from executing concurrently. While TxLocks correctly manages conflicts between locks and transactions, and minimizes programming overhead, such an approach can limit concurrent throughput because it conservatively overestimates the conflicts that exist between TxLocks and transactions.

P-SLE and Atomic Serialization Ziarek et al. introduce two key concepts: pure-software lock elision (P-SLE) and atomic serialization [17]. P-SLE eliminates conflicts between locks and transactions by converting locks into transactions. However, as noted by the authors, and as enumerated by Zyulkarov et al. [19], there are numerous reasons why the atomic regions protected by locks cannot seamlessly transition into transaction-based atomic regions (see Section 1). P-SLE handles these cases by reinstating all locks and using a single global lock to serialize transaction execution. This behavior, called *atomic*

serialization, guarantees mutual exclusion between transactions [17]. Although atomic serialization is correct because it serializes all transactions, such a strict serialized ordering may be unnecessary and may adversely effect performance.

Full Lock Protection TxLocks [15] and atomic serialization [17] provide essentially the same guarantee: they prevent a transaction from executing in one thread while a lock-based critical section is active in another. We call this behavior *full lock protection* because the shared-memory accessed within a lock is fully protected from transaction interference.

However, TxLocks and atomic serialization are not identical. Atomic serialization allows irrevocable operations to be used within locks that can then be placed inside of transactions. TxLocks does not allow such behavior. TxLocks' nesting model, therefore, differs from our own and atomic serialization. Therefore, when we discuss locks nested within transactions, we only consider atomic serialization. For non-nested cases, we consider both TxLocks and atomic serialization because their behavior is identical. For the remainder of the paper, we refer to TxLocks and atomic serialization as implementations of full lock protection (under the above restrictions), as it simplifies the discussion.

3 Language Constructs

We propose two interface extensions to enable programmers to efficiently manage conflicts between locks and transactions. Using these extensions, the programmer can choose to provide coarse-grained, fine-grained, or no information about potential conflicts between locks and transactions.

3.1 Coarse-Grained Conflict Management: TmLock

The TmLock data structure is used to allow programmers to provide coarse-grained information about potential conflicts between locks and transactions. The code shown below is a pseudocode example for the TmLock declaration and usage. TmLocks are used for locks that could potentially conflict with *any* transaction. When a TmLock is acquired at run-time, the TM system aborts or commits all in-flight transactions and then prevents any transaction from (re)starting until the the TmLock is released. Contention among TmLocks are handled in the same way as normal locks are handled. By allowing programmers to differentiate between locks that potentially conflict with transactions from the locks that do not, TmLock can improve concurrency because normal locks can run in parallel alongside transactions without conflict.

```
1 class TmLock {                               TmLock tmLock;
2 public:
3     void lock()                               tmLock.lock();
4     { /* Arbitration Algorithm */ }           ...
5     void unlock()                             tmLock.unlock();
6     { /* TM Subsystem Communication */ }
7 };
```

For unmanaged languages, if the programmer is unsure about potential conflicts between a lock and any transaction, a `TmLock` can be used in place of a normal lock as a conservative overestimation. Likewise, this same approach can be used for managed languages where all locks can be automatically converted to `TmLocks` by the compiler to guarantee conservative correctness. In the event of external locking conflicts (e.g., OS-level or library-level conflicts), the programmer can wrap external interfaces with `TmLocks`.

3.2 Fine-Grained Conflict Management: `atomic()`

We extend the `atomic` block to allow programmers to manage conflicts between locks and transactions at a finer granularity than supported by `TmLock` alone. Our extended `atomic` block, `atomic (TmLock [])`, takes an array of `TmLocks` as an argument and is shown in the example below. The array contains the list of `TmLocks` that can conflict with the transaction.

If a programmer is unsure about potential conflicts, she can conservatively overestimate them by using the `atomic` block with no argument, indicating the transaction may conflict with any `TmLock`. If the programmer knows the transaction does not conflict with any `TmLock`, she can pass `NULL` to the `atomic` block, indicating no conflicts. Under this programming model, transactions behave correctly with any number of `TmLocks` without *any* modification to the `atomic` construct. We chose this design over others because it reduces the initial challenge of integrating transactions into lock-based software and because it creates an iterative optimization path for programmers, where `atomic` blocks that have been overestimated to conflict with all `TmLocks` can be optimized at a later date.

```

1 // syntax: atomic (TmLock []) {}
2 TmLock L1, L2; TmLock locks[] = {L1, L2};
3 atomic (locks) {...}; // conflict with L1 and L2
4 atomic (NULL) {...}; // no conflicts
5 atomic {...}; // conflicts with all TmLocks

```

4 Algorithms

The steps to (re)start and end a transaction for both the fine- and coarse-grained algorithms are shown in Algorithm 1. The definition of a *conflicting* `TmLock`, represented by *conflictingLocks*, is context-sensitive. The coarse-grained algorithm defines all `TmLocks` as conflicting, so *conflictingLocks* is equal to all `TmLocks`. For the fine-grained algorithm, *conflictingLocks* is equal to those `TmLocks` specified in the `atomic TmLock` list. The *IsolatedTx()* procedure returns true if an isolated transaction (i.e., a transaction that forbids the concurrent execution of other transactions) is active, otherwise it returns false. The *obtainedMutexes* set collects the `TmLocks` that have been acquired during a transaction’s execution and is used to ensure a transaction’s state remains isolated by preventing other threads from acquiring such locks until the transaction has committed [15].

Due to space limitations, we have omitted the serialization used to access the shared data in all the algorithms shown in this section. In the **Begin** procedure, lines 3-4 are serialized. In the **End** procedure, the entire procedure is serialized.

Algorithm 1 Begin and End Transaction Procedures

```

1: procedure BEGIN(Transaction  $tx$ )
2:   loop
3:     Set  $L = \text{TmLocks.locked}()$     ▷ Returns set of currently locked TmLocks
4:     if  $(L \cap tx.conflictingLocks \equiv \emptyset \wedge !\text{IsolatedTx}())$  then return
5:     sleep(); continue
6: procedure END(Transaction  $tx$ )
7:    $tx.obtainedMutexes = \emptyset$ 
8:   Unblock( $tx.conflictingLocks$ ) ▷ Fine-grained only: unblock  $tx.conflictingLocks$ 

```

4.1 Coarse-Grained Algorithm

Algorithm 2 shows the `TmLock.lock` and `TmLock.unlock` procedures for the coarse-grained algorithm when a `TmLock` is both nested, and not nested, within a transaction. When a `TmLock` is acquired inside a transaction, the transaction requests permission from the contention manager (CM) to become isolated. If successful, this request aborts all active transactions and prevents new transactions from starting. This is done because `TmLocks`, by the coarse-grained definition, can conflict with any transaction. Hence, when a `TmLock` is acquired within a transaction, no other transactions can execute alongside it.

If the `TmLock` is not nested within a transaction, the algorithm calls `AbortTxes()` which requests permission from the CM to abort all transactions if there are any active (`ActiveTxes()`). In both cases, once there are no active transactions, except for the transaction that may nest the `TmLock`, the `TmLock` can be acquired.

When a `TmLock` is nested within a transaction, the `tx.partialCommit()` procedure is called as soon as the `TmLock` is acquired. This procedure commits the transaction's executed operations to the program state so non-transactional reads and writes, performed inside the `TmLock`'s critical section, will access the correctly updated memory. In Algorithm 2, lines 2-6 and lines 8-9 are serialized if the `while` loop's condition is false. If it is true, only the procedures called within the `while` loop's condition are serialized. For `TmLock.unlock`, the entire procedure is serialized.

4.2 Fine-Grained Algorithm

Algorithm 3 shows the fine-grained algorithm for `TmLock.lock` and `TmLock.unlock`. It includes cases where the procedures are nested, and not nested, within a transaction. The serialization used in Algorithm 3 is identical to Algorithm 2.

Algorithm 2 Coarse-Grained Lock and Unlock TmLock Procedures

Require: `threadId` is the global and unique id of thread that called `lock()`

```
1: procedure TmLock.Lock
2:   Transaction* tx = ActiveTx(threadId)           ▷ Pointer to active tx
3:   if (tx ≠ NULL) then                             ▷ TmLock.lock() is nested within tx
4:     if (tx.makeIsolated(CM)) then                 ▷ Request CM permission
5:       Acquire TmLock mutex; tx.obtainedMutexes.insert(this)
6:       tx.partialCommit()
7:   else                                               ▷ TmLock.lock() not nested in tx
8:     while (ActiveTxes() ≠ ∅ ∨ AbortTxes() ≡ false) do {}
9:     Acquire TmLock mutex
10: procedure TmLock.Unlock
11:   Transaction* tx = ActiveTx(threadId)
12:   if (tx ≠ NULL) then                             ▷ TmLock.unlock() is nested within tx
13:     if tx.obtainedMutexes ∩ this ≡ ∅ then         ▷ TmLock.lock() not nested in tx
14:       Throw EarlyReleaseDeadLock exception         ▷ Prevent deadlock [4]
15:   Release TmLock mutex
```

When a `TmLock.lock` call is not nested within a transaction, the `TmLock` requests the CM's permission to abort conflicting transactions, via `AbortConflictTxes()`. Because only those transactions whose extended atomic block has included the `TmLock` are aborted (checked by the `ConflictTxes()` procedure), those transactions that did not include the `TmLock` in their atomic block can continue to execute while a `TmLock` is acquired.

When `TmLock.lock` is called inside of a transaction, the transaction first requests permission to become irrevocable [16]. Irrevocable transactions cannot be aborted, just like isolated transactions, but unlike isolated transactions, they may yield greater concurrent throughput because other non-conflicting, revocable transactions may execute alongside them. Once the transaction is made irrevocable, all transactions that conflict with the `TmLock` are aborted, via `AbortConflictTxes()`. When `AbortConflictTxes()` is called within an irrevocable transaction, it always returns true.

By allowing transactions that acquire `TmLocks` to become irrevocable, rather than isolated, the fine-grained algorithm has the opportunity to produce more concurrent throughput than the coarse-grained algorithm. Transactions that acquire `TmLocks` using the fine-grained algorithm can be made irrevocable, rather than isolated, because the fine-grained algorithm requires that all conflicts between transactions and `TmLocks` be listed within in the extended atomic block. Because of this, non-conflicting revocable transactions may execute alongside an irrevocable transaction that has acquired a `TmLock`.

5 Qualitative Comparison

In this section, we compare the concurrency potential of full lock protection to our coarse- and fine-grained algorithms. We consider two general cases: (i) when

Algorithm 3 Fine-Grained Lock and Unlock TmLock Procedures

Require: threadId is the global and unique id of thread that called lock()

```
1: procedure TmLOCK.LOCK
2:   Transaction*  $tx = ActiveTx(threadId)$  ▷ Pointer to active tx
3:   if ( $tx \neq NULL$ ) then ▷ TmLock.lock() is nested within tx
4:     if ( $tx.makeIrrevocable(CM)$ ) then ▷ Request CM permission
5:        $AbortConflictTxes(this)$ ; Acquire TmLock mutex
6:        $tx.obtainedMutexes.insert(this)$ ;  $tx.partialCommit()$ 
7:   else ▷ TmLock.lock() not nested within tx
8:     while ( $ConflictTxes(this) \neq \emptyset \vee AbortConflictTxes(this) \equiv false$ ) do {}
9:     Acquire TmLock mutex
10: procedure TmLOCK.UNLOCK
11:   Transaction*  $tx = ActiveTx(threadId)$ 
12:   if ( $tx \neq NULL$ ) then ▷ TmLock.unlock() is nested within tx
13:     if  $tx.obtainedMutexes \cap this \equiv \emptyset$  then ▷ TmLock.lock() not nested in tx
14:       Throw EarlyReleaseDeadLock exception ▷ Prevent deadlock [4]
15:   else ▷ TmLock.unlock() is not inside tx
16:      $Unblock(this)$  ▷ Send message to unblock threads waiting for  $this$ 
17:   Release TmLock mutex
```

locks are not nested within transactions and (ii) when locks are nested within transactions.

5.1 No Nesting between Locks and Transactions

We use the six threaded example shown in Figure 1 to contrast the three algorithms when locks are not nested within transactions. Each thread executes a single function, shown in Figure 2, in a staggered fashion. Three of the threads use transactions, thread T_1 runs $Tx1()$, T_2 runs $Tx2()$, and T_3 runs $Tx3()$, while the other three use locks, thread T_4 runs $L1()$, T_5 runs $L2()$, and T_6 runs $L3()$. The example has a single conflict between T_3 's transaction ($Tx3()$) and T_4 and T_5 's locks ($L1$ and $L2$). Threads T_1 , T_2 , and T_6 do not exhibit any conflicts and only exist to draw out the differences in the concurrency potential of the algorithms. The coarse-grained conflict management algorithm only uses the information from TmLocks, while the fine-grained algorithm uses both TmLocks and the extended atomic blocks. Full lock protection uses no information from either construct, which is identical to both TxLocks and atomic serialization.

Full Lock Protection Transactions and locks are not allowed to concurrently execute when using full lock protection. This is because no information is provided about the conflicts that may persist between them. Because of this, the maximum concurrent lock and transaction throughput achievable by full lock protection is: $m(NoNesting_{\bar{fl}}) = \max(L_n, T)$. 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 concurrently executed. Because

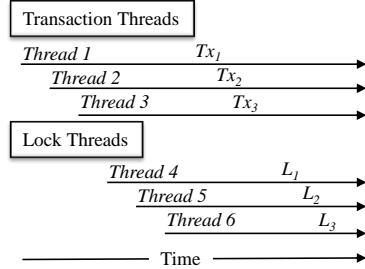


Fig. 1. Threads With No Nesting

```

1 TmLock L1, L2; Lock L3;
2 TmLock list[] = {L1,L2};
3
4 void Tx1() { atomic(NULL) {...} }
5 void Tx2() { atomic(NULL) {...} }
6 void Tx3() { atomic(list) {...} }
7 void L1() { /* lock/unlock L1 */ }
8 void L2() { /* lock/unlock L2 */ }
9 void L3() { /* lock/unlock L3 */ }

```

Fig. 2. Functions With No Nesting

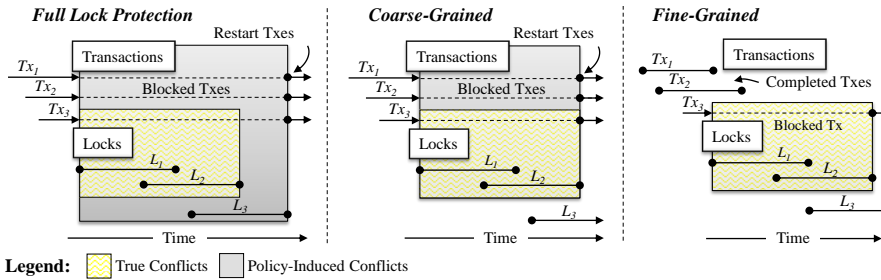


Fig. 3. Non-Nested Example: Full Lock Protection, Coarse-, and Fine-Grained.

no conflict information is used by full lock protection, only one type of critical section can be executed, locks or transactions, at a time. Figure 3 presents a visual model of full lock protection under the six threaded example. The transactions used in threads $T_1 - T_3$ are blocked while the lock-based critical sections of threads $T_4 - T_6$ execute, even though thread T_6 's lock does not conflict with any of the transactions and threads T_1 and T_2 's transactions do not conflict with any of the locks.

Coarse-Grained Conflict Management The coarse-grained algorithm leverages TmLock information that distinguishes locks that might conflict with transactions from those that are guaranteed to be conflict-free. This yields the following concurrency potential: $m(NoNesting_{tm}) = \max(L_{nl}, (L_{na} + T))$. 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, the coarse-grained approach allows the transactions in threads $T_1 - T_3$ to execute while only T_6 is executing, because lock L3 does not conflict with any transaction. As seen in Figure 3, this optimization shortens the overall TM run-time compared to full lock protection by allowing $T_1 - T_3$ to restart their transactions as soon as L1 and L2's critical section execution has completed.

Fine-Grained Conflict Management The fine-grained algorithm uses both `TmLock` and the extended `atomic` block information, enabling it to capture the greatest amount of potential concurrency. This is expressed as: $m(\text{NoNesting}_{tx}) = C_{lt} + L_{na} + T_{na}$. C_{lt} is the largest system selected set of locks and transactions that can be run concurrently without any conflicts. 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} is the total number of transactions that can be executed without conflicting with a lock. In the six threaded example, the fine-grained algorithm only stalls thread T_3 when locks L1 and L2, of threads T_4 and T_5 , are executing. The conflict time introduced by this approach is equal to the actual conflict time between the transactions and locks, resulting in the maximum amount of concurrent execution of locks and transactions.

5.2 Nesting Locks Inside of Transactions

It complicates conflict management for locks to be lexically nested within transactions. Locks must remain mutually exclusive, and, for this paper, we assume that locks do not have *failure atomicity* (i.e., they do not emit the property of side-effect free failure such as those found in transactions [2]). To ensure mutual exclusion among locks, when a lock is acquired inside of a transaction, the transaction becomes irrevocable (i.e., it cannot be aborted) [13, 16] or isolated (i.e., it is irrevocable and executes without any concurrently executing transactions).

Irrevocable and isolated transactions limit concurrency amongst transactions because conflicts between such transactions must be prevented pessimistically. That is, other transactions of the same type must be prevented from running concurrently even though conflicts between them may not exist. For isolated transactions, only one transaction can execute at a time, whereas with irrevocable transactions, only one irrevocable transaction may execute at a time, but any number of revocable transactions can concurrently execute alongside it.

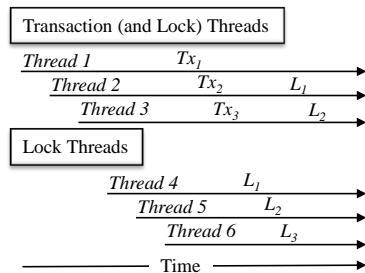


Fig. 4. Threads With Nesting

```

1 TmLock L1, L2; Lock L3;
2 TmLock tm1[] = {L1}, tm2[] = {L2};
3
4 void Tx1() { atomic(NULL) {...} }
5 void Tx2() { atomic(tm1) {L1();} }
6 void Tx3() { atomic(tm2) {L2();} }
7 void L1() { /* lock/unlock L1 */ }
8 void L2() { /* lock/unlock L2 */ }
9 void L3() { /* lock/unlock L3 */ }

```

Fig. 5. Functions With Nesting.

We use a six threaded example shown in Figures 4 and 5 to draw out the differences in potential concurrency of the three algorithms when locks are nested

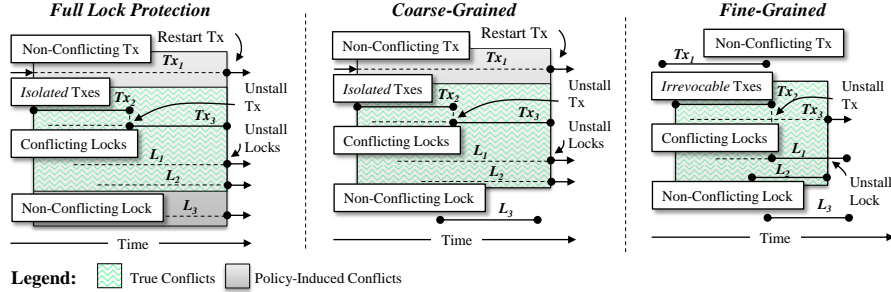


Fig. 6. Nested Example: Full Lock Protection, Coarse-, and Fine-Grained.

within transactions. Three of the threads, $T_1 - T_3$, execute transactions while the other three, $T_4 - T_6$, execute locks. Thread T_1 runs the transaction $Tx1()$, which contains no nested locking. Thread T_2 runs transaction $Tx2()$ which lexically nests lock L1. Thread T_3 runs transaction $Tx3()$ which nests lock L2. Threads $T_4 - T_6$ access locks L1 - L3, respectively.

Only two conflicts exist in the six threaded nested example. Threads T_2 and T_4 conflict on lock L1, while threads T_3 and T_5 conflict on lock L2. Threads T_1 and T_6 have no conflicts and are included only to highlight the differences in potential concurrency of the three algorithms.

Full Lock Protection Full lock protection conservatively assumes that all transactions may contain nested locks. Because of this, it disallows other transactions from executing until a transaction that has acquired a nested lock has committed. In addition, once a lock is acquired within a transaction, full lock protection must assume other locks may also be acquired within the transaction. Because such transactions guarantee isolation, and because locks cannot be released until transaction commit time, as explained in Section 4, full lock protection must prevent all lock-based critical sections from executing until the transaction has committed. This behavior is identical to atomic serialization, but not TxLocks for of the reasons mentioned in Section 2.1.

The maximum concurrent throughput given these restrictions is: $m(Nesting_{FL}) = \max(t_l, L_n, T_{nl})$. t_l is a single transaction that acquires a lock inside of it. L_n is the maximum number of locks that do not conflict with one another. T_{nl} is the maximum number of transactions that can be executed that do not have locks inside of them. Full lock protection only supports the execution of one of the following: an isolated transaction that contains nested locks, non-conflicting locks, or revocable transactions (as denoted by the \max function).

Coarse-Grained Conflict Management When using the coarse-grained approach, if a transaction acquires a $TmLock$, the transaction becomes isolated because the algorithm must assume all transactions can conflict with a $TmLock$. Because isolated transactions must not abort, no $TmLocks$ can be acquired while

the isolated transaction is active. This results in the following maximum concurrent throughput potential: $m(Nesting_{tm}) = \max((t_l + L_{nt}), (L_n + T_{nl}))$. t_l is a single transaction that acquires a **TmLock** 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 that can be executed which do not have **TmLocks** inside of them and do not conflict with L_n .

The coarse-grained approach is an improvement over full lock protection, and likewise, atomic serialization, because non-conflicting locks (i.e., locks that are not **TmLocks**) can be executed alongside an isolated transaction that has acquired a lock. In addition, another improvement over full lock protection is that non-conflicting locks can concurrently execute with other non-conflicting transactions, as illustrated with Figure 6.

Fine-Grained Conflict Management The fine-grained approach offers the greatest potential concurrent throughput by allowing **TmLocks** and revocable transactions to be run in parallel with a transaction that has acquired a **TmLock**. When using the fine-grained algorithm, if a **TmLock** is acquired by a transaction, the transaction becomes irrevocable, not isolated. This is an improvement over the coarse-grained algorithm because irrevocable transactions allow other revocable transactions to execute in parallel. Likewise, because each transaction using the fine-grained algorithm has its conflicting **TmLocks** listed, the **TmLocks** that are not listed as conflicting by a transaction can be acquired while the transaction executes, even if it is irrevocable.

This results in the following concurrency potential: $m(Nesting_{tx}) = \max((t_l + L_{nt} + T_r), (L_n + T_{nl}))$. All variables of the fine-grained algorithm are the same as coarse-grained algorithm except T_r . T_r is the maximum number of *revocable* transactions that can be executed concurrently alongside t_l that do not conflict with the set of locks, L_{nt} . As shown in Figure 6, the conflict time introduced by the fine-grained algorithm is equal to the actual conflict time.

6 Experimental Results

Our benchmark data was gathered on a 1.0 GHz Sun Fire T2000 supporting 32 concurrent hardware threads. For all benchmarks with the exception of the red-black trees in Figure 10, the x-axis shows the number of active threads and the y-axis shows the total execution time in seconds. In Figure 10, the x-axis shows the number of inserts and lookups and the y-axis shows the total execution time in seconds. Smaller total time indicates more efficient execution.

For cases where locks are not nested within transactions, we use an experimental model that ranges from 4 – 32 threads in multiples of four. In the four threaded version, the threads populate three containers of the same type (linked list, hash table, or red-black tree) and then perform sanity checks (i.e., `lookup()`s) on the values inserted. One thread populates a container with locks,

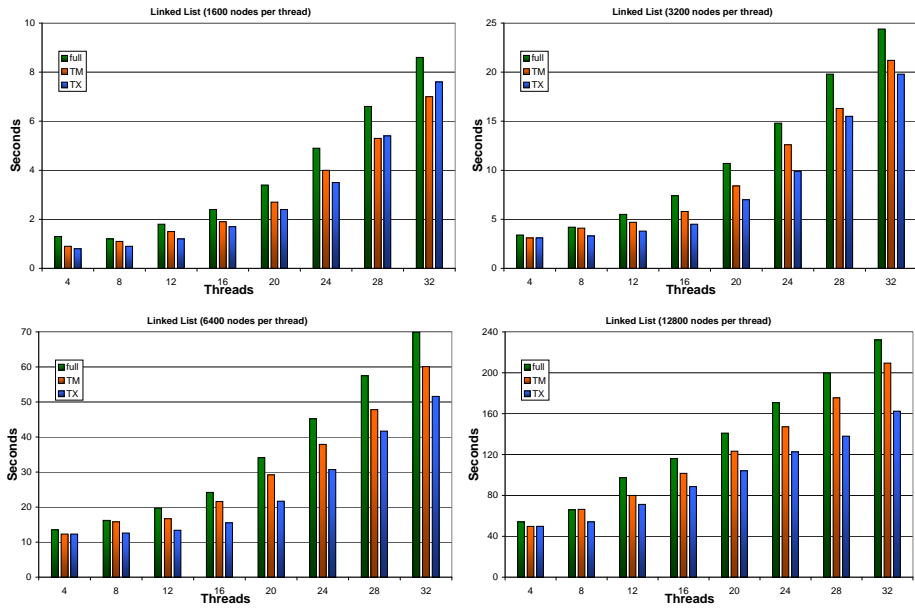


Fig. 7. Linked List Using Locks and Transactions Without Nesting.

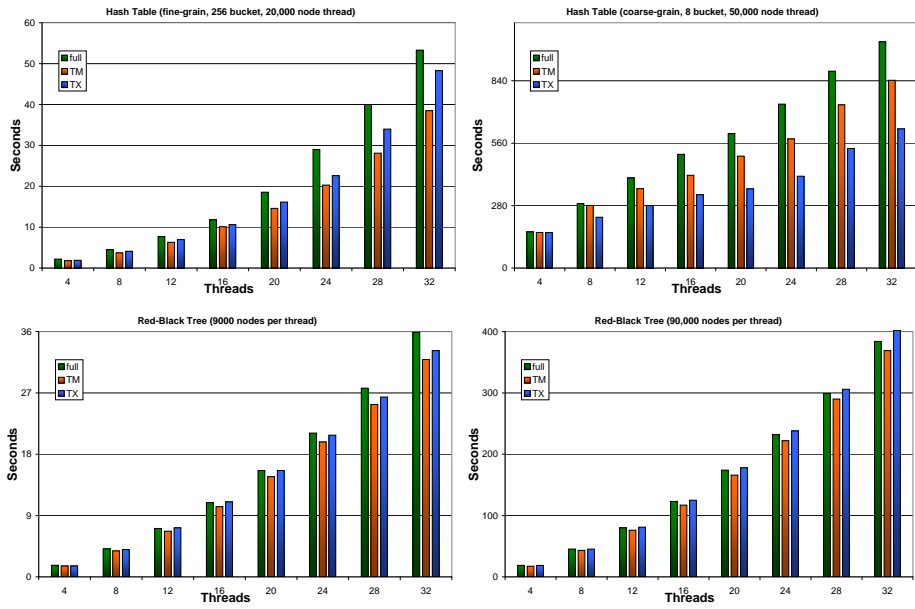


Fig. 8. Hash Table and Red-Black Tree Using Locks and Transactions Without Nesting.

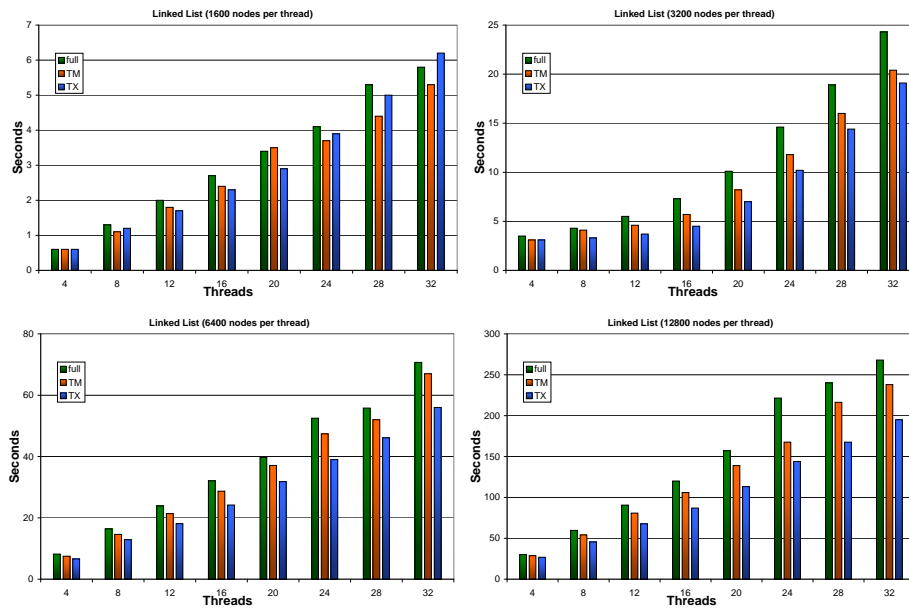


Fig. 9. Linked List With Locks Nested Inside of Transactions.

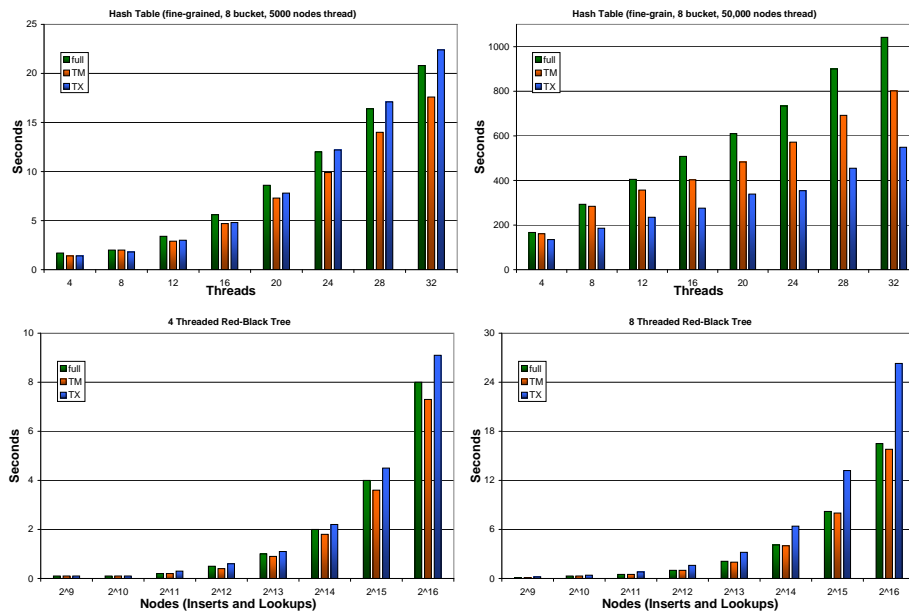


Fig. 10. Hash Table and Red-Black Tree With Locks Nested Inside of Transactions.

another thread populates a different container with transactions, and the third and fourth threads populate a third container with locks and transactions, respectively. Figures 7 and 8 display the execution time of these benchmarks ranging from 4-32 threads. Each benchmark was run using full lock protection (left bar), coarse- (abbreviated TM, middle bar), and fine-grained (abbreviated TX, right bar) algorithms.

For cases where locks are nested within transactions, shown in Figures 9 and 10, we use a similar model to those used in non-nested executions. We use the same basic 4-threaded model as described above, except that each fourth thread’s transactions nest calls to lock-based insert and lookup operations. In Figure 10, we slightly deviate from this model to isolate the red-black tree performance using only 4- and 8-threaded experiments while doubling the tree size each iteration. These benchmarks provide insight into the performance degradation of the fine-grained algorithm.

6.1 Performance Summary

Our experimental results are surprising. The coarse-grained algorithm (abbreviated as TM in the benchmarks) consistently outperforms full lock protection, while the fine-grained algorithm (abbreviated as TX in the benchmarks) ranges from $\approx 2x$ faster to $\approx 2x$ slower than either of the other approaches. Our initial results seem to indicate that the coarse-grained algorithm is a better general candidate than the fine-grained algorithm, in its current form, for software that supports the concurrent execution of locks and transactions. This is because *(i)* the coarse-grained algorithm requires minimal additional code (e.g., each of our benchmarks only required one extra line of code for the coarse-grained algorithm) and *(ii)* its performance efficiency is consistently better than the prior systems for our experimental benchmarks.

These results are not intuitive from the analyses presented in Sections 5.1 and 5.2. Our experimental results capture what was missed from the mathematical analysis in Sections 5.1 and 5.2. That is, the fine-grained algorithm’s computational overhead introduce latencies that can degrade overall program performance, even though it can increase the number of locks and transactions that can execute concurrently. Two general factors contribute to this. First, in order for the fine-grained algorithm to yield a performance benefit over the other algorithms, the cumulative critical section overhead when executed serially must be greater than the overhead incurred by the fine-grained algorithm. Second, the fine-grained algorithm must locate false conflicts that are overlooked by the other algorithms, which result in additional concurrent throughput. If both of these conditions are not satisfied, the fine-grained algorithm may not produce enough extra concurrent throughput to offset its algorithmic overhead and therefore it may perform worse than if it did no (full lock protection) or minimal (coarse-grained) conflict management.

An example of a benchmark that does not satisfy the above conditions can be seen in the nested red-black tree benchmarks of Figure 10. As can be seen in the 4-threaded red-black tree nested benchmark (Figure 10), as the workload

grows, there is a growing divide between the fine-grained algorithm and the other algorithms. This divide demonstrates that the critical section workload of the threads is less than the algorithmic overhead of the fine-grained algorithm, but greater than the other algorithms. Comparing the 4-threaded and 8-threaded red-black tree nested benchmarks to each other (again, Figure 10), one can observe an increased performance degradation of the fine-grained algorithm in the 8-threaded red-black tree compared to the 4-threaded red-black tree. This illustrates that the algorithmic overhead of the fine-grained algorithm is greater than the extra concurrency reclaimed from the false conflicts it finds, because when more threads are added to the benchmark, the fine-grained algorithm performs worse, not better, than the other algorithms.

7 Conclusion

While TM shows promise for future software programs, most TMs have undefined behavior for the concurrent execution of locks and transactions when they are used to synchronize the same shared-memory. This paper presented a performance study between our system and prior works that allow locks and transactions to execute in the same program.

We introduced two new language constructs: the `TmLock` and the extended `atomic` block. We analyzed how these constructs worked with two algorithms at different granularities. Our `TmLock` data structure combined with a coarse-grained algorithm yielded $\approx 1.5x$ improved program performance compared to prior systems and never performed worse than those systems. Our fine-grained algorithm provided up to a $\approx 2.0x$ performance improvement but, in some cases, resulted in a $\approx 2.0x$ performance degradation compared to prior work.

References

1. A. Abdelkhalik and A. Bilas. Parallelization and performance of interactive multiplayer game servers. In *IPDPS*, 2004.
2. J. Bloch. *Effective Java (2nd Edition) (The Java Series)*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2008.
3. E. W. Dijkstra. Solution of a problem in concurrent programming control. *Commun. ACM*, 8(9):569, 1965.
4. J. E. Gottschlich, J. G. Siek, M. Vachharajani, D. Y. Winkler, and D. A. Connors. An efficient lock-aware transactional memory implementation. In *Proceedings of the International ACM Workshop on IC00OLPS*. July 2009.
5. J. E. Gottschlich, M. Vachharajani, and J. G. Siek. An efficient software transactional memory using commit-time invalidation. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO)*, April 2010.
6. M. Herlihy and J. E. B. Moss. Transactional memory: Architectural support for lock-free data structures. In *Proceedings of the International Symposium on Computer Architecture*. May 1993.
7. K. E. Moore, J. Bobba, M. J. Moravan, M. D. Hill, and D. A. Wood. LogTM: Log-based transactional memory. In *HPCA*, pages 254–265. IEEE Computer Society, Feb. 2006.

8. R. Rajwar and P. A. Bernstein. Atomic transactional execution in hardware: A new high performance abstraction for databases. In *Workshop on High Performance Transaction Systems*, 2003.
9. R. Rajwar and J. R. Goodman. Speculative lock elision: enabling highly concurrent multithreaded execution. In *MICRO*, pages 294–305. ACM/IEEE, 2001.
10. 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 *SOSP*, pages 87–102. ACM, 2007.
11. N. Shavit and D. Touitou. Software transactional memory. In *Proceedings of the Principles of Distributed Computing*. Aug 1995.
12. M. F. Spear, V. J. Marathe, W. N. Scherer III, and M. L. Scott. Conflict detection and validation strategies for software transactional memory. In *Proceedings of the 20th International Symposium on Distributed Computing*, Sep 2006.
13. 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.
14. T. Usui, R. Behrends, J. Evans, and Y. Smaragdakis. Adaptive locks: Combining transactions and locks for efficient concurrency. In *Journal of Parallel and Distributed Computing*, pages 1009–1023, 2010.
15. H. Volos, N. Goyal, and M. M. Swift. Pathological interaction of locks with transactional memory. In *TRANSACT*, February 2008.
16. A. Welc, B. Saha, and A.-R. Adl-Tabatabai. Irrevocable transactions and their applications. In *SPAA*, 2008.
17. L. Ziarek, A. Welc, A.-R. Adl-Tabatabai, V. Menon, T. Shpeisman, and S. Jaggannathan. A uniform transactional execution environment for Java. In *ECOOP*, pages 129–154, 2008.
18. 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.
19. F. Zuykyarov, V. Gajinov, O. S. Unsal, A. Cristal, E. Ayguadé, T. Harris, and M. Valero. Atomic quake: using transactional memory in an interactive multiplayer game server. In *PPoPP*, pages 25–34, New York, NY, USA, 2009. ACM.