Maintaining Consistent Transactional States without a Global Clock

Hillel Avni[1,3] and Nir Shavit[1,2]

[1] Tel-Aviv University, Tel-Aviv 69978, Israel
[2] Sun Microsystems Laboratories, 1 Network Drive, Burlington MA 01803-0903
[3] Freescale Semiconductor Israel Ltd., 1 Shenkar Street, Herzliya 46725, Israel
hillel.avni@gmail.com

Abstract. A crucial property required from software transactional memory systems (STMs) is that transactions, even ones that will eventually abort, will operate on consistent states. The only known technique for providing this property is through the introduction of a globally shared version clock whose values are used to tag memory locations. Unfortunately, the need for a shared clock moves STM designs from being completely decentralized back to using centralized global information.

This paper presents *TLC*, the first thread-local clock mechanism for allowing transactions to operate on consistent states. TLC is the proof that one can devise coherent-state STM systems without a global clock.

A set of early benchmarks presented here within the context of the TL2 STM algorithm, shows that TLC's thread-local clocks perform as well as a global clock on small scale machines. Of course, the big promise of the TLC approach is in providing a decentralized solution for future large scale machines, ones with hundreds of cores. On such machines, a globally coherent clock based solution is most likely infeasible, and TLC promises a way for transactions to operate consistently in a distributed fashion.

1 Introduction

The question of the inherent need for global versus local information has been central to distributed computing, and will become central to parallel computing as multicore machines, now in the less than 50 core range, move beyond bus based architectures and into the 1000 core range. This question has recently arisen in the context of designing state-of-the-art software transactional memories (STMs).

Until recently, STM algorithms [1,2,3,4,5] allowed the execution of "zombie" transactions: transactions that have observed an inconsistent read-set but have yet to abort. The reliance on an accumulated read-set that is not a valid snapshot [6] of the shared memory locations accessed can cause unexpected behavior such as infinite loops, illegal memory accesses, and other run-time misbehavior. Overcoming zombie behavior requires specialized compiler and runtime support, and even then cannot fully guarantee transactional termination [7].

In response, Reigel, Felber, and Fetzer [8] and Dice, Shalev, and Shavit [7] introduced a *global clock* mechanism as a means of guaranteeing that transactions operate on consistent states. Transactions in past STM systems [1,2,3,4,5] typically updated a tag in the lock-word or object-record associated with a memory location as a means of providing transactional validation. In the new global clock based STMs [7,9,10] instead of having transactions locally increment the tags of memory locations, they update them with a time stamp from the globally coherent clock. Transactions provide consistency (recently given the name *opacity* [11]) by comparing the tags of memory locations being read to a value read from the global clock at the transaction's start, guaranteeing that the collected read-set remains coherent.

Unfortunately, this globally shared clock requires frequent remote accesses and introduces invalidation traffic, which causes a loss of performance even on small scale machines [7]. This problem will most likely make global clocks infeasible on large scale machines with hundreds of cores, machines that no longer seem fictional [12,13].

To overcome this problem, there have been suggestions of distributing the global clock (breaking the clock up into a collection of shared clocks) [14,15], or of providing globally coherent clock support in hardware [16]. The problem with schemes that aim to distribute the global clock is that the cost of reading a distributed clock grows with the extent to which it is distributed. The problem with globally coherent hardware clocks, even of such hardware modifications were to be introduced, is that they seem to be limited to small scale machines.

This paper presents *TLC*, the first *thread-local* clock mechanism that allows transactions to operate on consistent states. The breakthrough TLC offers is in showing that one can support coherent states without the need for a global notion of time. Rather, one can operate on coherent states by validating memory locations on a *per thread* basis. TLC is a painfully simple mechanism that has the same access patterns as prior STMs that operate on inconsistent states [1,3,2,4,5]: the only shared locations to be read and written are the tags associated with memory locations accessed by a transaction. This makes TLC a highly distributed scheme by its very nature.

1.1 TLC in a Nutshell

Here is how TLC works in a nutshell. As usual, a *tag* containing a time-stamp (and other information such as a lock bit or HyTM coordination bit) is associated with each transactional memory location. In TLC, the time-stamp is appended with the ID of the thread that wrote it. In addition, each thread has a *thread local clock* which is initially 0, and is incremented by 1 at the start of every new transaction. There is also a *thread local array* of entries, each recording a time-stamp for each other thread in the system. We stress that this array is local to each thread and will never be read by others.

Without getting into the details of a particular STM algorithm, we remind the reader that transactions in coherent-state STMs [7,10,9] typically *read* a location by first checking its associated tag. If the tag passes a *check*, the location is consistent with locations read earlier and the transaction continues. If it is not, the transaction is aborted. Transactions *write* a memory location by *updating* its associated tag upon commit.

Here is how TLC's *check* and *update* operations would be used in a transaction by a given thread i:

1. *Update*(location) Write to the locations tag my current transaction's new local clock value together with my ID i.
2. *Check*(location) Read the location's tag, and extract the ID of the thread j that wrote it. If the location's time-stamp is higher than the current time-stamp stored in my local array for the thread j, update entry j and abort my current transaction. If it is less than or equal to the stored value for j, the state is consistent and the current transaction can continue.

This set of operations fits easily into the global-clock-based schemes in many of today's STM frameworks, among them McRT [10], TinySTM [17], or TL2 [7], as well as hardware supported schemes such as HyTM [1] and SigTM [9].

How does the TLC algorithm guarantee that a transaction operates on a consistent read-set? We argue that a TLC transaction will always fail if it attempts to read a location that was written by some other transaction after it started. For any transaction by thread i, if a location is modified by some thread j after the start of i's transaction, the first time the transaction reads the location written by j, it must find the associated time-stamp larger than its last recorded time-stamp for j, causing it to abort.

An interesting property of the TLC scheme is that it provides natural locality. On a large machine, especially NUMA machines, transactions that access a particular region of the machine's memory will only ever affect time-stamps of transactions accessing the same region. In other words, the interconnect traffic generated by any transaction is limited to the region it accessed and goes no further. This compares favorably to global clock schemes where each clock update must be machine-wide.

The advantages of TLC come at a price: it introduces more false-aborts than a global clock scheme. This is because a transaction by a thread j may complete a write of some location completely before a given transaction by i reads it, yet i's transaction may fail because its array recorded only a much older time-stamp for j.

As our initial benchmarks show, on small scale state of the art multicore machines, the benefits of TLC are overshadowed by its higher abort rate. We did not have a 1000 node NUMA machine to test TLC on, and so we show that on an older generation 144 node NUMA machine, in benchmarks with a high level of locality, TLC can significantly outperform a global clock. Though this is by no way an indication that one should use TLC today, it is an indicator of its potential on future architectures, where a global clock will most likely be costly or even impossible to implement.

2 An STM Using TLC

We now describe an STM implementation that operates on consistent states without the need for a global clock by using the TLC algorithm. Our choice STM is the TL2 algorithm of Dice, Shalev, and Shavit [7], though TLC could fit in other STM frameworks such as McRT [10], or TinySTM [17] as well as hardware supported schemes such as HyTM [1] and SigTM [9]. We will call this algorithm *TL2C*.

Recall that with every transacted memory location TL2 associates a special versioned write-lock. The time-stamp used in the TL2C algorithm will reside in this lock. The structure of the TL2C algorithm is surprisingly simple. Each time-stamp written will be tagged with the ID of the thread that wrote it. Each thread has:

– a thread local *TLClock*, initially 0, which is incremented by 1 at the start of every new transaction, and
– a thread local *CArray* of entries, each entry of which records a time-stamp for each other thread in the system.

The clock has no shared components.

In the TL2C algorithm, as in the original TL2, the write-lock is updated by every successful lock-release, and it is at this point that the associated time-stamp is updated. The algorithm maintains thread local read-and write-sets as linked lists. The read-set entries contain the address of the lock. The write-set entries contain the address of the memory location accessed, the value to be written, and the address of the associated lock. The write-set is kept in chronological order to avoid write-after-write hazards. During transactional writing operations the read-set is checked for coherency, then write set is locked, and then the read-set is rechecked. Obviously aborts that happen before locking are preferable.

We now describe how TL2C, executes in commit mode a sequential code fragment that was placed within a transaction. The following sequence of operations is performed by a writing transaction, one that performs writes to the shared memory. We will assume that a transaction is a writing transaction. If it is a read-only transaction this can be annotated by the programmer, determined at compile time, or heuristically inferred at runtime.

1. Run through a speculative execution in a TL2 manner collecting read and write sets. A load instruction sampling the associated lock is inserted before each original load, which is then followed by post-validation code which is different than in the original TL2 algorithm. If the lock is free, a TL2C *check* operation is performed. It reads the location's time-stamp from the lock, and extracts the ID of the thread j that wrote it. If the location's time-stamp is higher than the current time-stamp stored in its *CArray* for the thread j, it updates entry j and aborts the transaction. If it is less than or equal to the stored value for j, the state is consistent and the speculative execution continues.

2. Lock the write set: Acquire the locks in any convenient order using bounded spinning to avoid indefinite deadlock. In case not all of these locks are successfully acquired, the transaction fails.
3. Re-validate the read-set. For each location in the read-set, first check it was not locked by another other thread. It might have been locked by the local thread if it is a part of both the read and write sets. Then complete the TL2C *check* for the location, making sure that its time stamp is less than the associated thread j's entry in the *CArray*. In case the *check* fails, the jth entry of the *CArray* transaction is aborted. By re-validating the read-set, we guarantee that its memory locations have not been modified while steps 1,2 and 3 were being executed.
4. Increment the local *TLClock*.
5. Commit and release the locks. For each location in the write-set, store to the location the new value from the write-set and *update* the time-stamp in the location's lock to the value of the *TLClock* before releasing it. This is done using a simple store.

Note that the updating of the time-stamps in the write-locks requires only a simple store operation. Also, notice that the local *TLClock* is only updated once it has been determined that the transaction will successfully commit.

The key idea of the above algorithm is the maintaining of a consistent read-set by maintaining a local view of each thread's latest time-stamp, and aborting the transaction every time a new time-stamp value is detected for a given thread. This prevents any concurrent modifications of the locations in the read set since a thread's past time-stamp was determined in an older transaction, so if the change occurs within the new transaction the new time stamp will be detected as new. This allows detection to proceed on a completely local basis. It does however introduce false aborts, aborts by threads that completed their transaction long before the current one, but will cause it to fail since the time-stamp recorded for them in the *CArray* was not current enough.

We view the above TL2C as a proof of concept, and are currently testing various schemes to improve its performance even on today's machines by reducing its abort rate.

3 Proof of the TL2C Algorithm

We outline the correctness argument for the TLC algorithm in the context of the TL2C algorithm. Since the TLC scheme is by construction wait-free, we only to argue safety. The proof of safety amounts to a simple argument that transactions always operate on a consistent state.

We will assume correctness of the basic underlaying TL2 algorithm as proven in [7]. In our proof argument, we refer to the TL2C algorithm's steps as they were defined in Section 2. Given the assumption that TL2 operates correctly, we need only prove that in both steps (1) and (3) in which the algorithm collects a read-set using TLC, this set forms a coherent snapshot of the memory, one that can be linearized at the start (first read) of the given collection phase.

We recall that every TL2 transaction that writes to at least one variable, can be serialized at the point in which it acquired all the locks on the locations it is about to write. Consider any collection phase (of read and write sets), including reads and writes by memory by a transaction of thread i in either step (1) or (3). For every location read by i, let the transaction that wrote to it last before it was read by i be one performed by a thread j. If j's transaction was not serialized before the start of the current collection, then we claim the collection will fail and the transaction by i will be aborted. The reason for this is simple. The last value stored in the *CArray* of i for j was read in a prior transaction of i, one that must have completed before i started the current transaction. Thus, since j increments its *TLClock* before starting its new transaction, if j's transaction was not linearized before i, then the value it wrote was at least one greater than the one recorded for j in the *CArray* of i. Thus i will detect an inconsistent view and abort its current transaction.

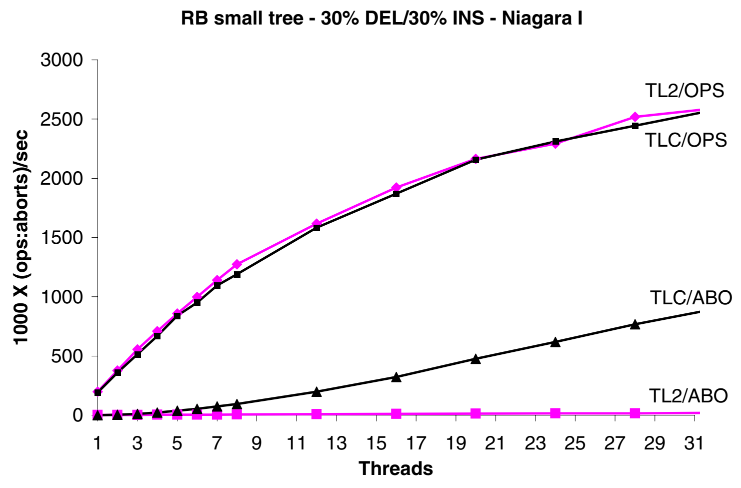
4 Empirical Performance Evaluation

The type of large scale NUMA multicore machine on which we believe one will benefit from the TLC approach is still several years ahead. We will therefore present two sets of benchmarks to allow the reader to gauge the limitations of the TLC approach on today's architectures and its potential benefits on future ones.

- The first benchmark is a performance comparison of the TL2C algorithm to the original TL2 algorithm with a version GV5 global clock (see [7] for details, the key idea of GV5 is to avoid frequent increments of the shared clock by limiting these accesses to aborted transactions.) on a 32-way Sun UltraSPARC T1™ machine. This is a present day single chip multi-core machine based on the Niagara architecture that has 8 cores, each supporting 4 multiplexed hardware threads.

  Our benchmark is the standard concurrent red-black tree algorithm, written by Dave Dice, taken from the official TL2 release. It was in turn derived from the `java.util.TreeMap` implementation found in the Java 6.0 JDK. That implementation was written by Doug Lea and Josh Bloch. In turn, parts of the Java TreeMap were derived from the Cormen et al [18].
- The second benchmark is a performance comparison of the TL2C algorithm to the TL2 algorithm on a Sun E25K™ system, an older generation NUMA multiprocessor machine with 144 nodes arranged in clusters of 2, each of which sits within a cluster of 4 (so there are 8 cores per cluster, and 18 clusters in the machine), all of which are connected via a large and relatively slow switch.

  Our benchmark is a synthetic work-distribution benchmark in the style of Shavit and Touitou [19]. The benchmark picks random locations to modify, in our case 4 per transaction, and has overwhelming fraction of operations within the cluster and a minute fraction outside it. This is intended to mimic



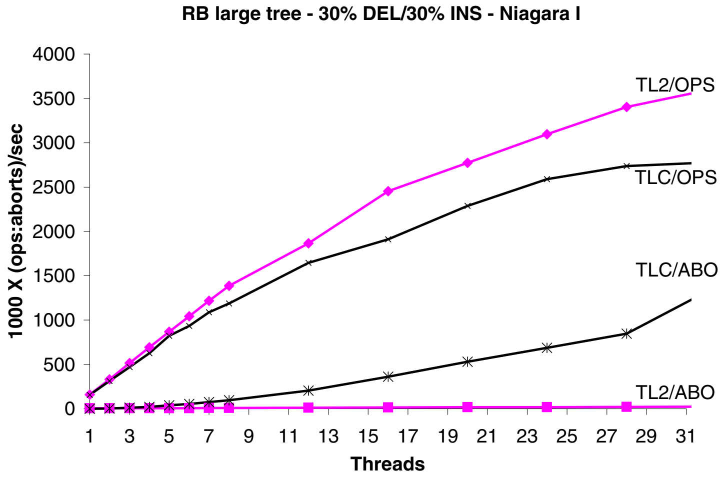


Fig. 1. Throughput of TL2 and TL2C on a Red-Black Tree with 30% puts, 30% deletes. The figure shows the throughput and the abort rate of each algorithm.

the behavior of future NUMA multicore algorithms that will make use of locality but will nevertheless have some small fraction of global coordination.

The graph of an execution of small (1000 nodes) and large red-black trees (1 million nodes) appears in Figure 1. The operation distribution was 30% puts, 30% deletes, and 40% gets. To show that the dominant performance factor in terms of TLC is the abort rate, we plot it on the same graph.

As can be seen, in both cases the smaller overhead of the TLC mechanism in the TL2C algorithm is shadowed by the increased abort rate. On the smaller tree the algorithms perform about the same, yet on the larger one the price of the

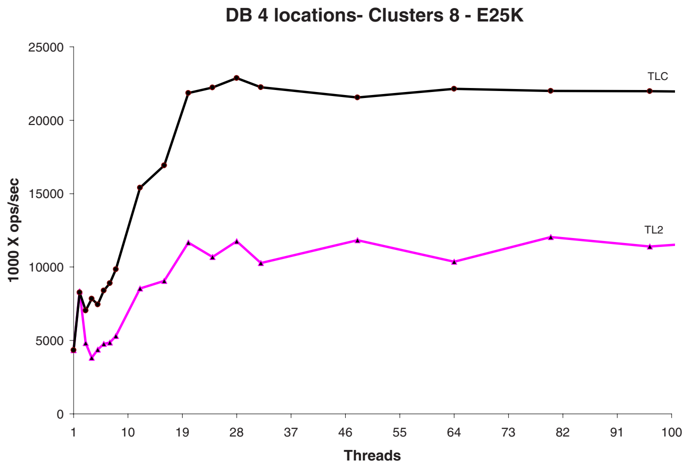


Fig. 2. Throughput of TL2 and TL2C on the work distribution benchmark in which most of the work is local within a cluster of 8 nodes

aborts is larger because the transactions are longer, and so TL2C performs more poorly than the original TL2 with a global clock. This result is not surprising as the overhead of the GV5 clock mechanism is very minimal given the fast uniform memory access rates of the Niagara I architecture.

The graph in Figure 2 shows the performance of the artificial work-distribution benchmark where each thread picks a random subset of memory locations out of 2000 to read and write during a transaction, mimicking a pattern of access that has high locality by having an overwhelming fraction of operations happen within a cluster of 8 nodes and a minute fraction outside it. As can be seen, the TL2C algorithm has about twice the throughput of TL2, despite having a high abort rate (not shown) as in the Niagara I benchmarks. The reason is that the cost of accessing the global clock, even if it is reduced by in relatively infrequent accesses in TL2's GV5 clock scheme, still dominates performance. We expect the phenomena which we created in this benchmark to become prevalent as machine size increases. Algorithms, even if they are distributed across a machine, will have higher locality, and the price of accessing the global clock will become a dominant performance bottleneck.

5 Conclusion

We presented a novel decentralized local clock based implementation of the coherence scheme used in the TL2 STM. The scheme is simple, and can greatly reduce the overheads of accessing a shared location. It did however significantly increase the abort rate in the microbenhmarks we tested. Variations of the algorithm that we tried, for example, having threads give other threads hints, proved too expensive given the simplicity of the basic TLC mechanism: they reduced the abort rate but increased the overhead. The hope is that in the future, on larger distributed machines, the cost of the higher abort rate will be offset by the reduction in the cost that would have been incurred by using a shared global clock.

References

1. Damron, P., Fedorova, A., Lev, Y., Luchangco, V., Moir, M., Nussbaum, D.: Hybrid transactional memory. In: ASPLOS-XII: Proceedings of the 12th international conference on Architectural support for programming languages and operating systems, pp. 336–346. ACM, New York (2006)
2. Dice, D., Shavit, N.: What really makes transactions fast? In: TRANSACT ACM Workshop (to appear, 2006)
3. Ennals, R.: Software transactional memory should not be obstruction-free (2005), `http://www.cambridge.intel-research.net/~rennals/notlockfree.pdf`
4. Harris, T., Fraser, K.: Language support for lightweight transactions. SIGPLAN Not. 38(11), 388–402 (2003)
5. Saha, B., Adl-Tabatabai, A.R., Hudson, R.L., Minh, C.C., Hertzberg, B.: Mcrt-stm: a high performance software transactional memory system for a multi-core runtime. In: PPoPP 2006. Proceedings of the eleventh ACM SIGPLAN symposium on Principles and practice of parallel programming, pp. 187–197. ACM, New York (2006)
6. Afek, Y., Attiya, H., Dolev, D., Gafni, E., Merritt, M., Shavit, N.: Atomic snapshots of shared memory. J. ACM 40(4), 873–890 (1993)
7. Dice, D., Shalev, O., Shavit, N.: Transactional locking ii. In: Dolev, S. (ed.) DISC 2006. LNCS, vol. 4167, pp. 194–208. Springer, Heidelberg (2006)
8. Riegel, T., Fetzer, C., Felber, P.: Snapshot Isolation for Software Transactional Memory. In: Proceedings of the First ACM SIGPLAN Workshop on Languages, Compilers, and Hardware Support for Transactional Computing (2006)
9. Minh, C.C., Trautmann, M., Chung, J., McDonald, A., Bronson, N., Casper, J., Kozyrakis, C., Olukotun, K.: An effective hybrid transactional memory system with strong isolation guarantees. In: ISCA 2007. Proceedings of the 34th annual international symposium on Computer architecture, pp. 69–80. ACM, New York (2007)
10. Wang, C., Chen, W.-Y., Wu, Y., Saha, B., Adl-Tabatabai, A.-R.: Code generation and optimization for transactional memory constructs in an unmanaged language. In: CGO 2007. Proceedings of the International Symposium on Code Generation and Optimization, Washington, DC, USA, pp. 34–48. IEEE Computer Society, Los Alamitos (2007)
11. Guerraoui, R., Kapalka, M.: On the correctness of transactional memory (2007)
12. Sun Microsystems, Advanced Micro Devices: Tokyo institute of technology (tokyo tech) suprecomputer (2005)
13. Systems, A.: Azul 7240 and 7280 systems (2007)
14. Dice, D., Moir, M., Lev, Y.: Personal communication (2007)
15. Felber, P.: Personal communication (2007)

16. Riegel, T., Fetzer, C., Felber, P.: Time-based transactional memory with scalable time bases. In: 19th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA) (2007)
17. Felber, P., Fetzer, C., Riegel, T.: Dynamic Performance Tuning of Word-Based Software Transactional Memory. In: Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP) (2008)
18. Cormen, T.H., Leiserson, C.E., Rivest, R.L.: Introduction to Algorithms. MIT Press, Cambridge (1990); COR th 01:1 1.Ex
19. Shavit, N., Touitou, D.: Software transactional memory. Distributed Computing 10(2), 99–116 (1997)