A New Contention Management Technique for Obstruction Free Transactional Memory

Ammlan Ghosh, Anubhab Sahin, Anirban Silsarma, Rituparna Chaki

A New Contention Management Technique for Obstruction Free Transactional Memory

Ammlan Ghosh, Anubhab Sahin, Anirban Silsarma, Rituparna Chaki

University of Calcutta, Kolkata, India
ammlan.ghosh@gmail.com, anubhab.s@hotmail.com,
anirban.silsarma@gmail.com, rchaki@ieee.org

Abstract. Transactional Memory, one of the most viable alternatives to lock based concurrent systems, was explored by the researchers for practically implementing parallel processing. The goal was that threads will run parallel and improve system performance, but the effect of their execution will be linear. In STM, the non-blocking synchronization can be implemented by Wait-Freedom, Lock-Freedom or Obstruction-Freedom philosophy. Though Obstruction Free Transactional Memory (OFTM) provides the weakest progress guarantee, this paper concentrates upon OFTM because of its design flexibility and algorithmic simplifications. In this paper, the major challenges faced by two state of the art OFTMs viz. Dynamic Software Transactional Memory (DSTM) and Adaptive Software Transactional Memory (ASTM), have been addressed and an alternative arbitration strategy has been proposed that reduces the abort percentage both in case of Read-Write as well as Write-Write conflicts.

Keywords: Software Transactional Memory (STM), Obstruction-free Transactional Memory (OFTM), Contention Management, Concurrency.

1 Introduction

Developing systems with multiple threads that can execute concurrently is no more a notion, but a reality. And in the current era, it is more of a necessity to utilize the full capacity of multi-core processors. Improvement of performance within a single core becomes essential to utilize the computational power provided by chip level multiprocessing. Locking has been an in-vogue technique used by the programmers for writing parallel programs. Lock based synchronization, however, leads to a number of unwanted situations like occurrence of deadlocks, priority inversion of processes and complication of fine-grained locking.

Concept of transactional memory addresses these issues and provides a promising alternative to lock based synchronization. The idea is to allow concurrent execution of transactions maintaining atomicity, consistency and isolation (ACI property) of each, i.e. threads will run parallel and improve system performance, but the effect of their execution will appear linear. Unlike database transactions, transactional memory instructions are meant to be short span transactions that access a relatively smaller

number of memory locations [1]. Transactional Memory systems can be purely hardware based (Hardware based Transactional Memory or HTM) [2], software-only (Software Transactional Memory or STM) [3] or hybrid. Naturally, the level of flexibility in STM over modification and integration is maximum. In STM, the fundamental operations i.e. the processes of acquiring and releasing ownership of concurrent objects (shared memory locations) are done atomically by non-blocking synchronization techniques using design primitives LL/SC (Load Linked Store Conditional) [4] and CAS (Compare and Swap) [4,5,7]. The key advantages are low space complexity and reduced performance overhead. These atomic operations are widely supported by multi-core processors.

The non-blocking implementations of STM systems have been mostly designed on the basis of either Lock-Freedom or Obstruction-Freedom philosophy. An STM system is lock free if some transactions are guaranteed to commit in a finite number of steps [6]. Although Lock-Freedom often delivers exceptional results, there is a question mark over the correctness of semantics in these algorithms. An STM system is obstruction free if every transaction is guaranteed to commit in absence of contention. Obstruction freedom provides the weakest progress guarantee and also admits the possibility of livelocks. Still obstruction freedom has been the preferred choice of many as it substantially reduces the implementation complications i.e. codes are simple, flexible and depending upon the design, can considerably improve parallelism and scalability of a system with many cores.

In 2003, Herlihy et al. constructed one of the earliest obstruction-free STM systems called DSTM [10] to support dynamic sized data structures. Since then several OFTMs have been proposed including ASTM [11], RSTM [12] and NZTM [13], with considerable differences in their respective system designs. The researchers were mainly interested in improving the throughput and minimizing the computational overhead of transaction processing.

In this paper, some major challenges faced by two state of the art OFTMs viz. DSTM and ASTM, have been addressed and an alternative negotiation strategy has been proposed. Unlike the existing OFTM systems, the proposed method allows multiple Read-Only transactions to share data object concurrently along with Write transactions. When a Write transaction reaches its commit point it checks the maturity of the all active read-only transactions and decides which of them are allowed to be committed. The proposed algorithm also presents a new contention management policy to resolve conflicts between Write transactions. Section 2 describes some existing works, followed by section 3 which presents the proposed algorithm; section 4 evaluates the performance of the algorithm; finally we conclude and discuss future scope in section 5.

2 Background

The STM uses primitive atomic operations like and LL/SC (load-link and store-conditional) [4] and CAS (Compare and Swap) [5] for implementing read, write, commit and abort statements. Load-link and store-conditional are a pair of instructions used together in multithreading to achieve synchronization. Load-link returns the current value of a memory location and a subsequent store-conditional will

store a new value if no updates are made in that location meanwhile. CAS is used to read from a particular memory location and to write back the modified value in the same location after ensuring that the location has not been altered in between. Of late a slightly sophisticated version viz. DCAS (Double-word Compare and Swap) [14] has been used in some STMs, which basically executes two CAS operations simultaneously. These primitive atomic operations are used to guarantee that consistency of the system is not hampered during an update. The common performance metrics for the various STM systems have been (i) Conflict Management, (ii) Transaction Granularity and (iii) Number of Basic Operations. In obstruction free environment, when a conflict occurs among two or more transactions (of which at least one is a Write transaction) over a particular resource, the management policy of the concerned system will determine which transaction(s) will progress and which will abort. This conflict management strategy of an OFTM is determined by the contention manager. The performance of an OFTM depends largely upon the efficiency of the contention manager [15]. Granularity is considered as the smallest data store memory unit that can be possessed by a transaction for its Read/Write operations.
The authors have discussed two well known OFTM implementations viz. DSTM [10] and ASTM [11] as the proposed methodology has been influenced by these implementations.

2.1 DSTM

Herlihy et al. proposed one of the earliest obstruction-free STM (OFTM) systems called DSTM [10] to support dynamic sized data structures. The highlight of this system was assurance of progress in practice with the introduction of a modular contention manager, thus removing the single biggest drawback of OFTM. In DSTM, the TM-Object (Figure 1) points to a locator object with three pointers: pointer 1 points to the descriptor of the most recent transaction that held the object; pointers 2 and 3 point to the old and new versions of the data object. When a transaction successfully commits, the new version of the data object is made permanent. On the other hand, when a transaction is aborted by other transaction, the old version of the data object is read by the aborting transaction before its execution. Concepts of early release and visible/invisible reads were also coined by the DSTM developers, which have been applied in various forms in the latter STM designs. The idea of early release is that a transaction may release an opened object before committing. This sometimes proves really beneficial in case of data structures like trees. The read visibility helps to avoid unnecessary contention between Read only transactions. In this scheme, each transaction maintains a separate Read-list of the objects that have been opened by Read only transactions. Before commit, a Write transaction checks the Read-list to resolve the contention. The read visibility yields a large performance benefit, especially in read-dominated work load, due to its easy read-object validation.

Herlihy et al. [10] proposed two basic contention managers viz. Aggressive Manager and Polite Manager. An Aggressive Manager directly aborts the conflicting transaction(s) whereas Polite Manager uses exponential back-off to acquire ownership of the TM-Object.

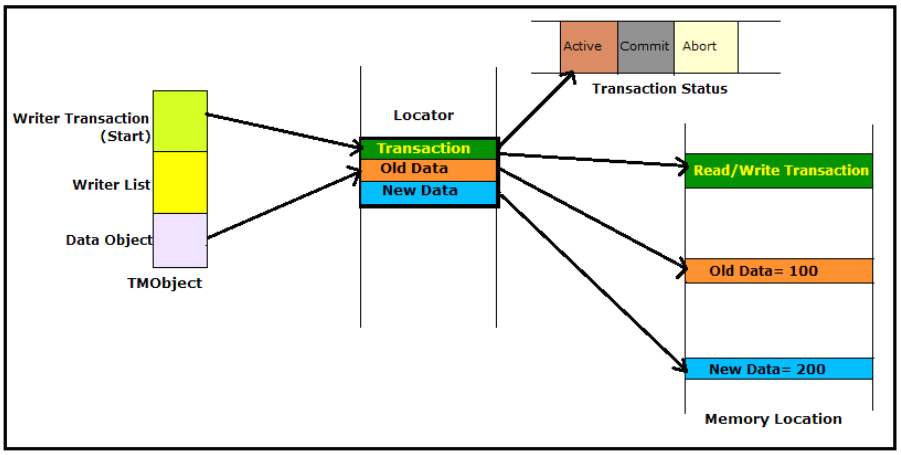


Fig. 1. Transactional Object (TM Object) structure in DSTM

2.2 ASTM

Adaptive Software Transactional Memory (ASTM) [11] used the structures of DSTM along with a modified conflict management scheme to overcome the loopholes of DSTM. ASTM offered adaptive methodology to adjust the workload. A transaction acquires an object in two ways. In eager acquire methodology a transaction acquires the objects at the beginning, where as in lazy acquire scheme a transaction acquires objects at commit time. Eager acquire method allows a transaction to detect contention earlier and helps to ensure consistency. Naturally eager-ASTM works better in write dominated workload while lazy-ASTM works better in read dominated workload. The flip side is that it increases the overhead of the system because of the adaptive nature of acquire methodology.

Initially ASTM did not introduce any new manager, but shortly ASTM-2 [16] was released with Adaptive Contention Management Policies. The Adaptive Contention Manager uses the idea of machine learning motivated by behaviorist psychology, i.e. from previous experience a manager decides how to take action in a particular environment so as to maximize transaction throughput**.** ASTM2 includes a number of contention managers viz. Karma Manager, Eruption Manager and Greedy Manager. Karma Manager gives absolute priority to the transaction that has done more work. Eruption Manager is based on the principle that the more number of transactions a particular transaction is blocking, the higher priority the blocker transaction should have. Finally Greedy Manager does not abort the conflicting transaction unless it has a lower priority or is currently waiting for another transaction; otherwise it aborts the running transaction.

For Write transactions (acquired state), the TM-Object points to a locator with similar structure as that of DSTM. But by default the TM-Object points to the data objects (unacquired state). So when a Read transaction follows a Write transaction, the former suffers indirection overhead, not only for acquiring the ownership of the TM-Object but also for the extra CAS for changing the direction of the pointer from

locator to data. Analysis of this STM system over simulated test cases also verifies this finding; as expected, in workloads dominated by either read or write operations, ASTM gives excellent performance. But in case of a uniformly mixed set of Reads and Writes, the throughput degrades drastically.

The above STM systems use object granularity where there is no need to change the original object structure for converting a normal program to a transactional one. Object granularity also perfectly suits object oriented programming style.

Both in case of ASTM and DSTM, a transaction that opens *n* objects in write mode requires *n* CASes to acquire the objects and an additional CAS to commit which makes a total of *n+1* CASes. But the cost might increase manifold in ASTM as the subsequent readers might perform up to *n* CASes to return the objects to unacquired state [6].

3 Proposed scheme of arbitration over a resource between two or more competing transactions

From the above inductive analysis we observe two major challenges common to both DSTM and ASTM:

(i) Considerably high number of aborts, and

(ii) The complexity involved in implementing modular contention management policy leads to a higher computational overhead

The frequent roll-backs of write transactions hamper the transactional processing greatly as normally write transactions execute longer than read-only transactions. The loss proves much costlier when a lengthy write transaction gets aborted by a much smaller write because of the rigidity of the concerned contention management policy. Also modular contention management schemes discussed above requires imparting intelligence in the software system such that based on the workload pattern the system can decide for itself which contention manager to use in a particular situation. Keeping these two major challenges in mind, this paper proposes a generic conflict management strategy aiming at reducing the abort percentage. The technique is based on the use of a single contention manager for all types of workload patterns. The proposed method uses lazy conflict detection scheme for both read-only as well as write transactions. In a bid to avoid spurious aborts for read-only transactions, a list of 'matured' read transactions (on the basis of their execution time) is maintained. When a write transaction tries to commit, it checks this list and backs-off to give a chance to these read-only transactions to commit. When a transaction detects conflicts, it either backs-off for certain time to give chance to the conflicting transactions or aborts conflicting transactions or aborts itself. The decision is taken after consulting the contention manager, in order to achieve synchronization in a non-blocking manner.

3.1 TM-Object structure in proposed OFTM

The proposed OFTM maintains the TM Object structure (Figure 2) similar to that of DSTM. Additionally, this TMObject has a pointer to the write transaction's Q_RdrLst, a list of qualified read-only transaction, on the basis of which a write transaction decides its back-off policy. Also for log file storage a descriptor is maintained by every transaction that stores the transaction metadata. The descriptor remains in the thread local storage (TLS) and is created right at the beginning, i.e. during thread initialization.

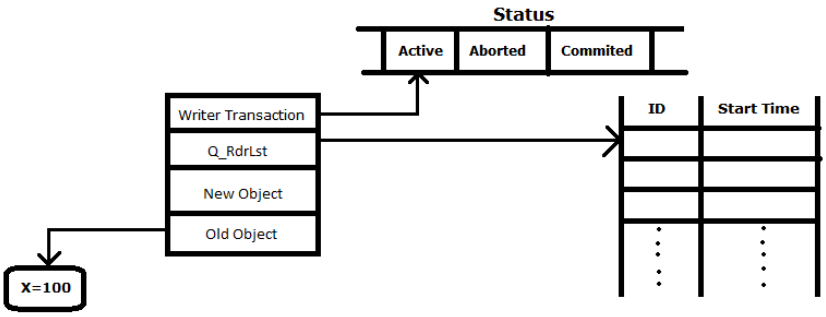


Fig. 2. TM Object structure in proposed OFTM

3.2 Nomenclature

It is presumed that transactions belonging to $\sum^{n}_{i=1} T^{i}_{R/W}$ and obtained from a Pseudo Random Number Generating Algorithm open the object 'O' for either reading or modifying (each write transaction acquires a different version).

The underlying assumption is that work done by a transaction is roughly an increasing linear function of its total execution time. The following is an exhaustive list of term definitions used to describe the functioning of the system.

$\mathbf{t^{\mu}_{W}}$: mean time taken by the CPU to finish a write transaction.

$\mathbf{t^{\mu}_{R}}$: mean commit time of a read-only transaction.

$\mathbf{\#T^{s}_{w}}$: number of committed write transactions whose execution time is less than or equal to t^{μ}_{w}.

$\sum^{n}_{i=1}\mathbf{T^{i}_{R/W}}$: set of all transactions that initiate and commit within time span 't'.

$\mathbf{\#T_{w}}$: total number of committed write transactions.

$\mathbf{t^{i}}$: total time for which a transaction $T^{i}_{R/W}$ has executed, as calculated at that particular time instant. $T^{i}_{R/W}$ may be in active/aborted/committed state.

RdrLst: all active read-only transactions are stored in this list.

$\mathbf{T^{i}_{w}.Q\_RdrLst \subset RdrLst}$: Qualified Reader List is a 2-tuple {j, $Init^{j}_{R}$} list corresponding to write transaction T^{i}_{w}, which is the set of all active read-only transactions that started before T^{i}_{w}'s *try_commit* point.

- **j**: Identity of the read-only transaction
- $\mathbf{Init^{j}_{R}}$: timestamp at the initiation of T^{j}_{R}

Mtr_lvl^i_W: *maturity_level* of the i^{th} write transaction; if any write transaction executes lesser than t^{μ}_w, then *maturity_level* of that transaction is considered as ***LO***; if any write transaction executes longer than t^{w}_{max}, then *maturity_level* of that transaction is considered as ***HI***,

- $t^{w}_{max} = t^{\mu}_w * \#T_w / \#T^s_w$

The values of t^{μ}_W and t^{μ}_R get modified every time a new write transaction or a new read-only transaction commits. *Maturity_level* is considered only for active write transactions.

3.3 Proposed Algorithm

For all transactions T^i_w that modified object O and has reached its *try_commit* point

If Found = TRUE

If T^i_w.Q_RdrLst != Ø
$t_b = t^{\mu}_R$ - Min(t^k) ∀ $T^k_R \in T^i_w$.Q_RdrLst
T^i_w backs off for time t_b
Commit all $T^k_R \in T^i_w$.Q_RdrLst that reach their respective *try_commit* points within this time period; remove them from RdrLst and T^i_w.Q_RdrLst
Abort all $T^m_R \in T^i_w$.Q_RdrLst
End if

If Mtr_lvl^i_W = *LO*
Check if there exists transaction, T^j_w which holds a version 'O_j'
∋ (Mtr_lvl^j_W = *HI*) && (O_i!= O_j)

If Found = TRUE
Complete execution of T^j_w and commit T^j_w
End if
End if

Commit T^i_w
Roll back all other write transactions holding O_k, ∋ O_k != O_i and free the corresponding memory locations
End if

3.4 Case Study

Normally it is observed that within a workload, even the longest read-only transaction executes for a shorter period than the smallest write transaction. But exceptions may happen especially in case of reads involving indirect addressing. In the proposed negotiation strategy, neither absolute free hand has been given to read-only transactions, nor occurrence of a conflict results in indiscriminate aborting of all reads under consideration. An intermediate approach has been adopted where only those read-only transactions that can finish execution within a stipulated time (t_b) are allowed to commit.

We have categorized the active Write transactions in terms of their total time of execution as maturity_level = *LO* and maturity_level = *HI*. Reverse ratio has been assumed while defining t^{w}_{max} because experimental results reveal that the number of committed write transactions which execute for less than 't^{μ}_{w}' time is considerably greater than #Tw/2.

3.4.1 Random generation of threads

We can visualize the system threads that use a particular object (say O) in the form of the following function,

$$i = \{(Rand(x) \text{ Mod } \#T_n) + 1\}$$

This function gives a very good virtualization of random order acquisition of an object by different transactions (For e.g. T_i can be T_5, T_{33}, T_{17}, etc.).

But if all the threads are independent, i.e. they do not share any common object, each will commit on reaching its *try_commit* point without bothering the others. This, however is a trivial case and is very unlikely to happen in practical systems.

3.4.2 Checking before Write commit

Before a write transaction commits, the system basically performs the following: it checks if any read-only transaction(s) is holding an old version of the same object; next it verifies if the *maturity_level* of the *try_commit* write transaction is *LO*; if the outcome of this second check is affirmative, it performs the final check, i.e. if the transaction under consideration is forcing another write transaction with *maturity_level* = *HI* to roll back. From the outcome of this threefold checking, it effectively makes way for most read-only transactions (if not all) to commit and also saves any lengthy write transaction which would have been sacrificed otherwise. The value of back-off time t_b is environment dependent and might vary from one workload set to another. This is shown in the Figure 3a and 3b. Table 1 compares two well-known OFTMs i.e. DSTM and ASTM with the proposed one, by characterizing some important designing parameters.

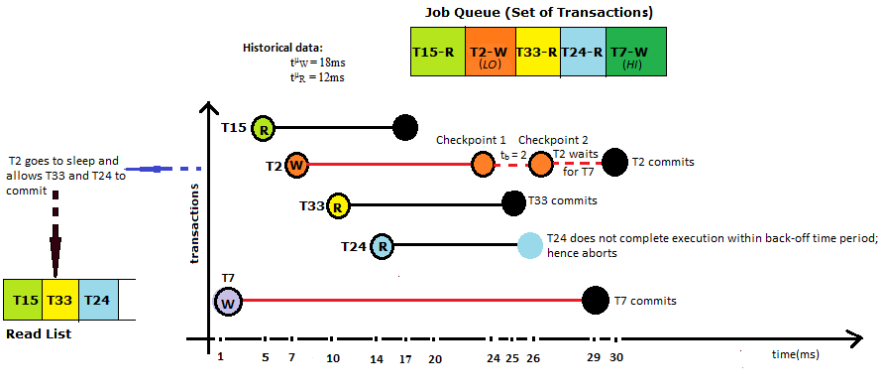


Fig. 3. a. Case 1. T_2 waits for T_7 to commit

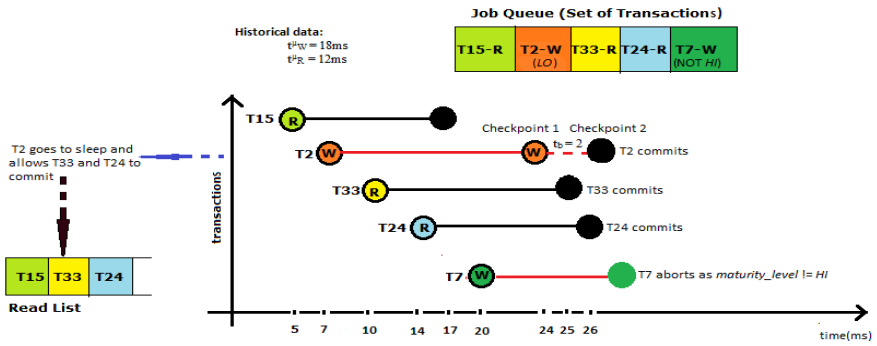


Fig. 3. b. Case 2. T_2 forces T_7 to abort

3.5 Comparison of the proposed OFTM with DSTM and ASTM

Table 1. Analysis of DSTM, ASTM and the proposed OFTM based on some designing parameters.

Designing Parameters	DSTM	ASTM	PROPOSED OFTM
Synchronization	Obstruction Freedom	Obstruction Freedom	Obstruction Freedom
Granularity	Object Based	Object Based	Object Based
Conflict Detection	Eager	Lazy	Lazy
Update Strategy	Direct	Deferred	Deferred
Read Visibility	Invisible	Invisible	Visible
Data Organization	Keeps transactional data and object data in separate memory structures	Keeps transactional data and object data in separate memory structures	Keeps transactional data and object data in the same memory structure
Data Indirection	Two indirections	Two indirections	One indirection

4 Performance Evaluation

The efficiency and performance improvement of the proposed method over the existing OFTMs [10], [11] depends a lot upon the level of domination of reads in the set. This has been analyzed here in three different categories. However, towards the end of this section, it would be established that the proposed approach would guarantee at least equivalent throughput as compared to the existing approaches.

4.1 Metric of evaluation

There are various parameters of evaluating the performance of an STM system. One fundamental approach is by measuring the percentage of committed transactions. To compare the performance of the various Contention Managers, we have created a pool of fifty threads. With variation in the percentage of writes, we have executed two very well-known Contention Managers, viz. Aggressive Manager proposed by Herlihy et. al. in DSTM and Karma Manager of ASTM-2 in the thread pool. On the same test beds we have tested our proposed manager. Here contention over a single resource has been considered. The start time and execution time of the transactions were generated randomly and the following results were achieved.The following figures show the results of evaluation.

Fig. 4.a. 10% Write Set

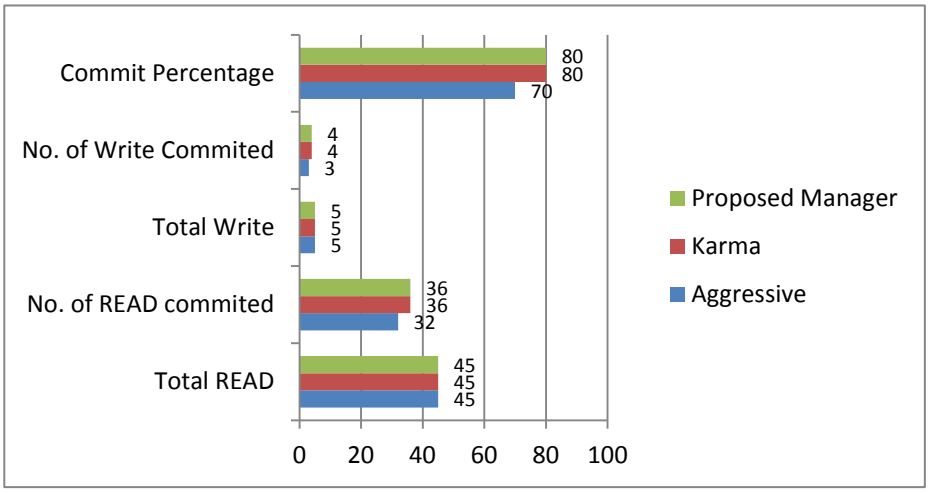


Fig. 4.b. 20% Write Set

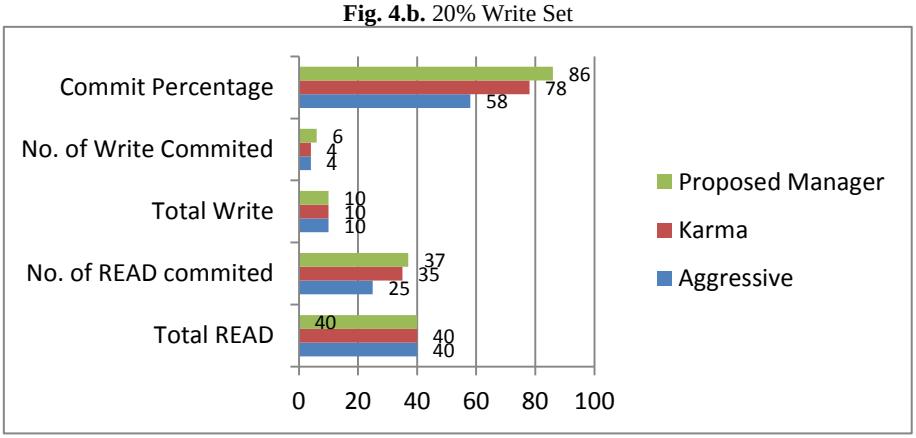
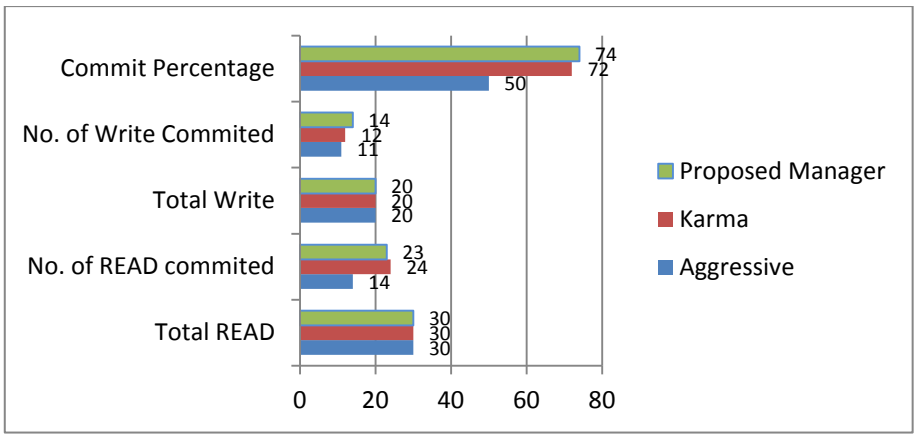


Fig. 4.c. 40% Write Set



4.3 Performance Summary:

The simulation results on the three different workload sets reveal that performance of the proposed algorithm is always better than performance of both these managers in case of read-write conflicts. This is because of the sensible back-off performed by the contention manager whenever there is a contention between a read transaction and a write transaction. Also it is found that performance of the proposed manager with respect to the others improves significantly with decrease in the domination of reads in the workload. From the graphs, it can be concluded that though the proposed manager performs slightly better than Karma manager, but it surpasses the commit percentage of Aggressive Manager by miles over all workloads.

By a non-weighted average across all the four test cases, our manager achieves a flat betterment of 36.85% over the performance of Aggressive Manager and 4.34% over that of Karma Manager.

5 Conclusion and Future Scope

The proposed OFTM aims at reducing the number of aborts. It gives a fair amount of time period to the active read-only transactions so that they can complete execution and commit. In many cases it will not abort any of the read-only transactions. The novelty of this system lies in the fact that in case of write-write conflicts, it saves the matured writes, instead of aborting it. The system also performs clean-ups for all unsuccessful transactions and frees the corresponding memory locations. On the flip side, this STM system is not strictly non-blocking as putting T^{i}_{w} off to sleep is

basically blocking it from completing its execution. In this regard we have made a trade-off between rigidity of non-blocking semantics and system throughput.

The authors plan to test the algorithm upon some more sophisticated benchmarks like Red Black tree and Hash Table. Once done, the performance of the proposed OFTM system shall be compared with the best known existing OFTMs. Considering the degree of reduction of computational overhead, a par performance or even 10% degradation in terms of throughput should be a satisfactory result for proposed OFTM.

6 References

1. Tim Harris, James Larus and Ravi Rajwar.: Transactional Memory 2nd Ed. Morgan & Claypool. 2010.
2. Maurice Herlihy and J. Eliot B. Moss.: Transactional memory: architectural support for lockfree data structures. In: ISCA '93: Proc. 20th Annual International Symposium on Computer Architecture, pages 289–300, May 1993
3. N. Shavit and D. Touitou.: Software transactional memory. In: ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing, pp. 204-213. ACM August 1995.
4. Virendra J. Marathe and Michael L. Scott.: Using LL/SC to simplify word-based software transactional memory (poster). In: PODC'05: Proc. 24th ACM Symposium on Principles of Distributed Computing, July 2005.
5. Rachid Guerraoui and Michal Kapalka.: On Obstruction-Free Transactions. In: SPAA '08: Proc. 20th Annual Symposium on Parallelism in Algorithms and Architectures, pages 304–313, June 2008.
6. Virendra J. Marathe., and Michael L. Scott.: A Qualitative Survey of Modern Software Transactional Memory Systems. Technical Report Nr. TR 839. University of Rochester Computer Science Dept. (2004)
7. Maurice Herlihy.: Wait-free synchronization. In: TOPLAS: ACM Transactions on Programming Languages and Systems, vol. 13, no. 1, pp. 124–149, Jan. 1991.
8. Keir Fraser.: Practical lock freedom, PhD Dissertation, Cambridge University Computer Laboratory, 2003.
9. M. Herlihy, V. Luchangco, and M. Moir.: Obstruction-free synchronization: Double-endedqueues as an example, In: Proceedings of the 23rd International Conference on Distributed Computing Systems, pp. 522-529, 2003.
10. M. Herlihy, V. Luchangco, M. Moir, and W. N. Scherer III.: Software Transactional Memory for Dynamic-sized Data Structures. In: 22nd Annual ACM Symposium on Principles of Distributed Computing, pp. 92-101, July 2003.
11. V. J. Marathe, W. N. Scherer III, and M. L. Scott. Adaptive software transactional memory. In: Proceedings of the 19th International Symposium on Distributed Computing (DISC), pp. 354–368, 2005.
12. V. J. Marathe, M. F. Spear, C. Heriot, A. Acharya, D. Eisenstat, W. N. Scherer III, and M. L. Scott.: The Rochester software transactional memory runtime. http://www.cs.rochester.edu/research/synchronization/rstm, 2006.
13. F. Tabba, C. Wang, J. R. Goodman, and M. Moir. NZTM: non-blocking zero-indirection transactional memory. In: Proceedings of the 21st ACM Annual Symposium on Parallelism in Algorithms and Architectures (SPAA), pp. 204-213, 2009.
14. Hagit Attiya, and Eshcar Hillel.: The power of DCAS: highly-concurrent software transactional memory. In: Proceedings of the twenty-sixth annual ACM symposium on Principles of distributed computing. ACM, 2007.

15. W. N. Scherer, III and M. L. Scott.: Advanced contention management for dynamic software transactional memory. In: PODC ’05: Proceedings of the twenty-fourth annual ACM symposium on Principles of distributed computing, pp. 240–248, NY, USA, 2005.

16. Joel. C. Frank and Robert Chun.: Adaptive Software Transactional Memory: A Dynamic Approach to Contention Management. In: PDPTA’08: Proceedings of 14th International Conference on Parallel and Distributed Processing Techniques and Applications, pp. 40-46, Nevada, USA, 2008.