Starvation Freedom in Transactional Memory Systems *

Ved Prakash Chaudhary[1], Raj Kripal Danday[1], Hima Varsha Dureddy[1], Sandeep Kulkarni[2], Sweta Kumari[1], and Sathya Peri[1]

[1]Department of Computer Science Engineering, IIT Hyderabad
[2]Department of Computer Science, Michigan State University

Abstract

In the recent years *Big Data Analytics* has become a very popular paradigm for solving problems of diverse fields from engineering to education. Big data analytics as the name suggests involves processing large amounts of data requires huge processing power. Multi-core systems which have become prevalent can address the processing needs of Data Analytics.

Multi-core programming typically involves synchronization and communication which can be very expensive. Software Transactional Memory systems (*STMs*) have garnered significant interest as an elegant alternative for addressing synchronization and concurrency issues in multi-core systems.

In order for STMs to be efficient, they must guarantee some progress properties. This work explores the notion of starvation-freedom in Software Transactional Memory Systems (STMs). To the best of our knowledge, starvation-freedom has not yet been explored in the context of STMs.

In this paper, we first present *Single-Version Starvation Free STM* or *SV-SFTM*. As the name suggests, this system maintains a single version for each tobj. SV-SFTM satisfies opacity and starvation-freedom. But SV-SFTM does not take advantage of multiple versions. It can cause abort of many transactions (although it ensures that every transaction commits if it is re-executed sufficient number of times). As a result, the progress of the entire system can be brought down. We can alleviate this situation by using multiple versions.

We propose KSTM, a Multi-Version STM system that maintains K versions for each tobj. KSTM satisfies opacity but not starvation-free. As a part of future work, we plan to develop a Multi-Version Starvation Free STM System, *MV-SFTM* that guarantees starvation-freedom of transactions. Although KSTM does not guarantee starvation-freedom, it is a precursor to MV-SFTM. It provides an insight as to how to achieve starvation-freedom with multi-version STMs and thus help us design MV-SFTM.

1 Introduction

In the past few years *Big Data Analytics* has become a very popular paradigm for solving problems of very diverse fields from engineering to education. It is clear that to solve challenges of big data analytics, huge processing power will be required. Multi-core systems which have become prevalent can address the processing needs of Data Analytics.

Programming multi-core systems is usually performed using multi-threading. But, multi-threading and hence multi-core programming typically involves synchronization and communication which can be very expensive. The cost of synchronization can sometime be high that it can negate the programming power of multi-core systems and thus result in degrading multi-core to single-core systems.

Software Transactional Memory systems (*STMs*) [10, 18] have garnered significant interest as an elegant alternative for addressing synchronization and concurrency issues in multi-core systems. STMs are a convenient programming interface for a programmer to access shared memory without worrying about consistency issues [10, 18]. STM systems uses optimistic approach in which multiple transactions can execute concurrently. On completion, each transaction has to validate and if any inconsistency is found then it is *aborted*. Otherwise it is allowed to *commit*. A transaction that has begun but has not yet

* **Work currently in Progress**

been validated is referred to as *live*.

A typical TM system is a library which exports the methods: begin which begins a transaction, *read* which reads a *transaction-object* (data-item) or *tobj*, *write* which writes to a tobj, *tryC* which tries to commit.

An important requirement of STM systems is to precisely identify the criterion as to when a transaction should be aborted/committed referred to as *correctness criterion*. Several correctness criterion have been proposed for STMs such as opacity [7], virtual worlds consistency [12], local opacity [14], TMS [1, 5] etc. All these correctness criterion require that all the transactions including aborted to appear to execute sequentially in an order that agrees with the order of non-overlapping transactions. Unlike the correctness criterion for traditional databases serializability [16], these correctness criterion ensure that even aborted transactions read consistent values. This is one of the fundamental requirements of STM systems first observed in [7] which differentiates STMs from Databases.

Another important requirement of STM system is to ensure that transactions make *progress* i.e. they do not abort unnecessarily. It would be ideal to abort a transaction only when it does not violate correctness requirement (such as opacity). However it was observed in [2] that many STM systems developed so far spuriously abort transactions even when not required.

Wait-freedom is one of the interesting progress condition for STMs in which every transaction commits regardless of the nature of concurrent processes [9]. But it was shown by Guerraoui and Kapalka [8] that it is not possible to achieve wait-freedom in dynamic TMs in which data sets of transactions are not known in advance. So in this paper, we explore a weaker progress condition *starvation-freedom* [11, chap 2]. Intuitively, it is defined as follows in the context of TM systems: Suppose a transaction T_i on getting aborted by the TM system is re-executed. Then, the STM system is said to be starvation-free if it can ensure that T_i will eventually commit if T_i is retried every time it aborts (and T_i does not invoke tryA). It can be seen that in order to ensure starvation-freedom, the STM system must store some state information for each aborted transaction.

Algorithm1 illustrates starvation-freedom. It shows the overview of *insert* method which inserts an element e into a linked-list LL. Insert method is implemented using transactions to ensure correctness in presence of concurrent threads operating on common data-items. The method has an infinite while loop line 1 to line 15. In this while loop, a new transaction is created to read and write onto the shared memory. This corresponds to creating and inserting a new node into the shared memory. If the transaction succeeds then the control breaks out of the loop. Otherwise, this process continues until a transaction is eventually able to succeed. Thus, it can be seen that insert method can execute forever if transactions created by it never successfully commits. To ensure that insert method eventually completes, the STM system must guarantee starvation-freedom of transactions.

Algorithm 1 Insert(LL, e): Invoked by a thread to insert a value v into a linked-list LL. This method is implemented using transactions.

```
1:  while (true) do
2:      id = tbegin();
3:      ...
4:      ...
5:      v = read(id, x);
6:      ...
7:      ...
8:      write(id, x, v');
9:      ...
10:     ...
11:     ret = tryC(id);
12:     if (ret == success) then
13:         break;
14:     end if
15: end while
```

In this paper, we explore ideas to achieve starvation-freedom for in STMs. To the best of our knowledge, starvation-freedom has not yet been explored in the context of STMs. We first present *Single-Version Starvation Free STM* or *SV-SFTM*. As the name this system maintain a single version for each tobj.

SV-SFTM is based on Forward-Oriented Optimistic Concurrency Control Protocol (FOCC), a commonly used optimistic algorithm in databases [20, Chap 4]. As per this algorithm, when two transactions T_i, T_j conflict, one of them is aborted. The transaction to be aborted, say T_j, is one which has lower priority in terms of how long it has executed. When a transaction T_i begins, it is allotted an *initial-timestamp* or *ITS*. If T_i gets aborted, then it restarts again with a new identity, say T_p, but retains the original ITS. In case of conflict of T_p with T_j, the conflict is resolved based on ITS of T_p (which is same as T_i) and T_j. The transaction with higher ITS is aborted. The details of this algorithm are described in SubSection**??**.

It was observed that more read operations succeed by keeping multiple versions of each object, i.e.multi-version STMs can ensure that more read operations to return successfully [13, 15]. History $H1$ shown in Figure 1 illustrates the idea of multi version. $H1$: $r_1(x,0)w_2(x,15)w_2(y,20)c_2r_1(y,0)c_1$.

In history $H1$ the read on y by T_1 is not reading the previous closest write of 20 by T_2 instead of that it returns 0. This is only possible when it's having multiple versions of y. As a result, this history H is opaque with the correct equivalent execution T_1T_2. If multiple versions will not be available then $r_2(y)$ has to read 10 only because this is only the available version. This value would make the read of $r_2(y)$ to be inconsistent (opaque) and hence abort T_2.

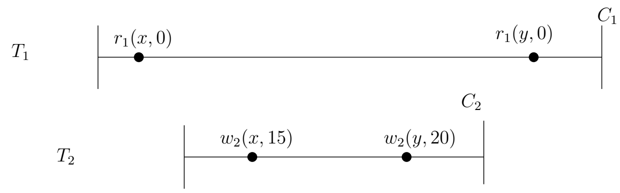


Figure 1: Pictorial representation of a History $H1$

Thus, multi-version STMs (MVSTMs) can achieve greater concurrency and progress. Many STM systems have been proposed using the idea of multiple versions [13, 15, 6, 4, 17]. All these MVSTMs do not place a limit on the number of versions created. They have separate thread routines that perform *garbage-collection* on old and unwanted versions periodically. In fact, it was shown in [13], greater the number of versions, lesser the number of aborts.

It can be seen that SV-SFTM does not take advantage of multiple versions. As a result, SV-SFTM can still cause abort of many transactions (although it ensures that every transaction commits if it is re-executed sufficient number of times). Consider the case that a transaction T_i with has the lowest ITS. Hence, it cannot be aborted as per SV-SFTM. But if it is slow (for some reason), then it can cause several other conflicting transactions to abort. Hence, the progress of the entire system can be brought down. We can alleviate this situation by using multiple versions.

Hence, we plan to develop a Multi-Version Starvation Free STM System, *MV-SFTM* that guarantees starvation-freedom of transactions. In this paper, we propose a K-version Multi-Version STM system that maintains K versions, KSTM. It is a precursor to MV-SFTM as KSTM does not guarantee starvation-freedom, but provides an insight into how to achieve starvation-freedom with multi-version STMs.

KSTM maintains K versions where K can range from between $1 - \infty$. When K is 1 then this algorithm boils down to a single-version STM system. If K is ∞ then it is similar to existing MVSTMs which do not maintain a upper bound on the number of version. We show KSTM satisfies opacity.

To study the efficiency of STMs developed, we consider a useful metric *commit time* defined as the time taken by a transaction to commit which includes the re-execution time caused by aborts. Naturally, this metric depends on the applications with which the STM system is tested. We plan to measure the performance commit time of SV-SFTM, KSTM and MV-SFTM (to be developed in future) using various benchmarks. The advantage of KSTM is that one can tune the value of K to obtain the best commit time for a given application. We want to understand which variant of STM can provide greater commit time: FOCC,SV-SFTM, KSTM, MV-SFTM. For the latter two, we have to experiment with a suitably chosen value for K. We have shown some preliminary results in appendix.

Overview of our Contributions and Roadmap. We describe our system model in Section 2. In Section 3 we describe SV-SFTM and in Section 4, we describe KSTM. We conclude in Section 5. Finally in appendix, we describe about missing code and some preliminary results.

2 System Model and Preliminaries

We assume a system with n processes/threads, $P_1, P_2, ..., P_n$ that access collection of objects through atomic $transactions$. A transaction is the sequence of instructions executing in memory. Each transaction is having the unique transaction identifier. The objects accessed by the transaction during read and write operations are known as transactional objects (tobjs). History H is the interleaving operations of different transactions including commit and abort. There are four possible $transactional$ $operations$ in the processes: the read operation reads the value of x and return it, write operation updates the value of x by v, tryCoperation tries to commit the transaction and returns commit or abort, the *tryA*operation aborts the transaction and return abort. In the read operation, a transaction reads from the shared memory and write operation, writes into its local buffer. The transaction executes *tryC*operation, when it has completed all its read-write operations. In this all the reads/writes are validated to see if they form a consistent view of the memory. Consistent view means, for a given history H there exist an equivalent serial history H'. If so, the transaction is committed and all the local writes are written into the shared memory. Otherwise, the transaction is aborted.

A history is said to be *valid*, if all the successful read is reading from any previously committed transaction. A history is said to be *legal*, if all the successful read is reading from latest committed transaction. So, every legal history is also a valid history.

A history H is said to be *opaque* [8], if there exists an equivalent sequential history S such that S respects real time order and is legal. Unlike serializability [16], it consider all the transaction including aborted one. It omits all the writes and unsuccessful reads of aborted transaction from H. In this, we consider incomplete transaction as aborted.

3 Single Version Starvation Free STM

In this section, we describe SV-SFTM algorithm.

3.1 Main Idea

Forward-oriented optimistic concurrency control protocol (FOCC), is a commonly used optimistic algorithm in databases [20, Chap 4]. In fact, several STM Systems are also based on this idea. In a typical STM system (also in database optimistic concurrency control algorithms), a transaction execution is divided can be two phases - a *read/local-write phase* and *try-Commit phase* (also referred to as validation phase in databases). The various algorithms differ in how the try-Commit phase executes. Let the write-set or WS and read-set or RS of a t_i denotes the set of tobjs written & read by t_i. In FOCC a transaction t_i in its try-Commit phase is validated against all live transactions that are in their read/local-write phase as follows: $\langle WS(t_i) \cap (\forall t_j : RS^n(t_j)) = \Phi \rangle$. This implies that the WS of t_i can not have any conflict with the current RS of any transaction t_j in its read/local-write phase. Here $RS^n(t_j)$ implies the RS of t_j till the point of validation of t_i. If there is a conflict, then either t_i or t_j (all transactions conflicting with t_i) is aborted. A commonly used approach in databases is to abort t_i, the validating transaction.

In SV-SFTM we use *time-stamps* which are monotonically in increasing order. We implement the time-stamps using atomic counters. Each transaction t_i has two time-stamps: (i) *current time-stamp or CTS*: this is a unique time-stamp alloted to t_i when it begins; (ii) *initial time-stamp or ITS*: this is same as CTS when a transaction t_i starts for the first time. When t_i aborts and re-starts later, it gets a new CTS. But it retains its original CTS as ITS. The value of ITS is retained across aborts. For achieving starvation freedom, SV-SFTM uses ITS with a modification to FOCC as follows: a transaction t_i in try-Commit phase is validated against all other conflicting transactions, say t_j which are in their read/local-write phase. The ITS of t_i is compared with the ITS of any such transaction t_j. If ITS of t_i is smaller than ITS of all such t_j, then all such t_j are aborted while t_i is committed. Otherwise, t_i is aborted. Due to lack of space, we have showed an example illustrates the working of SV-SFTM in appendix Section 3.2. We show that SV-SFTM satisfies opacity and starvation-free.

Theorem 1 *Any history generated by SV-SFTM is opaque.*

Theorem 2 *SV-SFTM ensure starvation-freedom.*

We prove the correctness by showing that the conflict graph [20, Chap 3], [14] of any history generated by SV-SFTM is acyclic. We show starvation-freedom by showing that for each transaction t_i there eventually exists a global state in which it has the smallest ITS.

3.2 Illustration of SV-SFTM

Figure 2 shows the a sample execution of SV-SFTM. It compares the execution of FOCC with SV-SFTM. The execution on the left corresponds to FOCC, while the execution one the right is of SV-SFTM for the same input. It can be seen that each transaction has two time-stamps in SV-SFTM. They correspond to CTS, ITS respectively. Thus, transaction $T_{1,1}$ implies that CTS and ITS are 1. In this execution, transaction T_3 executes the read operation $r_3(z)$ and is aborted due to conflict with T_2. The same happens with $T_{3,3}$. Transaction T_5 is re-execution of T_3. With FOCC T_5 again aborts due to conflict with T_4. In case of SV-SFTM, $T_{5,3}$ which is re-execution of $T_{3,3}$ has the same ITS 3. Hence, when $T_{4,4}$ validates in SV-SFTM, it aborts as $T_{5,3}$ has lower ITS. Later $T_{5,3}$ commits.

It can be seen that ITSs prioritizes the transactions under conflict and the transaction with lower ITS is given higher priority.

4 K-version Multi-Version STM

SV-SFTM drawback: We start this section by illustrating the drawback of SV-SFTM.

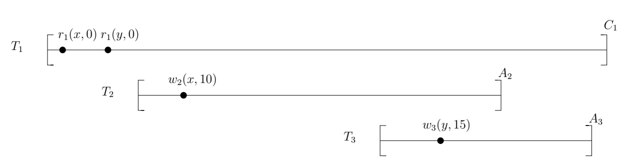


Figure 3: Pictorial representation of execution under SFTM

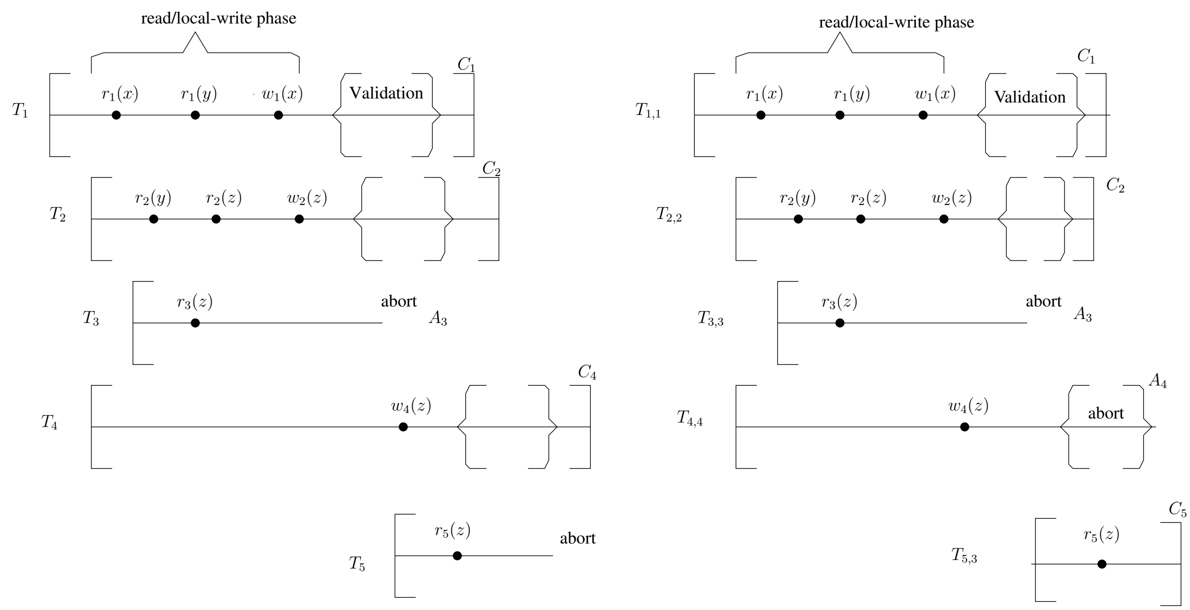


Figure 2: Sample execution of SV-SFTM

Figure 3 is representing history H: $r_1(x, 0)r_1(y, 0)w_2(x, 10)w_3(y, 15)a_2a_3c_1$ It has three transactions T_1, T_2 and T_3. T_1 is having lowest time stamp and after reading it became slow. T_2 and T_3 wants to write to x and y respectively but when it came into validation phase, due to $r_1(x)$, $r_1(y)$ and not committed yet, T_2 and T_3 gets aborted. However, when we are using multiple version T_2 and T_3 both can commit and T_1 can also read from T_0. The equivalent serial history is $T_1T_2T_3$.

4.1 Main Idea

KSTM algorithm is based on *MVTO* algorithm for STMs [13] which again is similar to the MVTO algorithm proposed for databases [3]. The proposed MVTO algorithm does not maintain any limit on the number of versions. As a result it has to execute a separate garbage-collection procedure.

KSTM algorithm as the name suggests maintains k-versions for each tobj and uses time-stamps (like SV-SFTM). Each tobj maintains all its versions as a linked-list. Each version of a tobj has three fields (1) time-stamp which is the CTS of the transaction that wrote to it; (2) the value of the version; (3) a list, called read-list, consisting of transactions CTSs that read from this version.

1. $read(x)$: Transaction t_i reads from a version of x with time-stamp j such that j is the largest time-stamp less than i (among the versions x), i.e. there exists no version k such that $j < k < i$ is true. If no such version exists then t_i is aborted.

2. $write(x, v)$: t_i stores this write to value x locally in its WS.

3. $tryC$: This operation consists of multiple steps:

   (a) t_i validates each tobj x in its WS as follows:

      i. t_i finds a version of x with time-stamp j such that j is the largest time-stamp less than i (like in read).

      ii. Then, among all the transactions that have read from j if there is any transaction t_k such that $j < i < k$ and t_k has already committed then t_i is aborted. Otherwise, if t_k is still live then t_k is aborted. Transaction t_i then proceeds to validate the next tobj in its WS.

      iii. If there exists no version of x with time-stamp less than i then t_i is aborted

   (b) After performing the tests of Step 3(a)i, Step 3(a)ii, Step 3(a)iii over each tobjs x in t_i's WS, if t_i has not yet been aborted, then for each x: among all the versions of x currently present, the oldest version is over-written with i and i's value. Transaction t_i is then committed.

Further details of KSTM algorithm can found in appendix.
We have the following result.

Theorem 3 *Any history generated by KSTM is opaque.*

We prove the correctness of the algorithm by showing that the equivalent serial history, all the transactions are ordered by their time-stamps. But KSTM does not satisfy starvation-freedom which is illustrated in an example.

KSTM illustration: We now illustrate the working of the algorithm with an example. Figure 4 shows an execution where $K = 3$ and the currently considered versions for a tobj x are $5, 15$ & 25. Consider version 15. Its value is 8 and its read-list consists of transactions with time-stamps $17, 22$. The C next to id 22 indicates that t_{22} is already committed. Transactions t_{17} is still live. In this setting suppose transaction t_{23} intends to commit and create a new version. In this case, $15 < 23 < 24$ and t_{24} is still live. Hence, t_{24} is aborted and a new version with time-stamp 23 is allowed to be created. Since 5 is the oldest version, the newly created version 23 overwrites 5. Next, consider the case that transaction t_{26} intends to commit and create a new version. Since t_{29} is already committed, t_{26} is not allowed to create a new version.

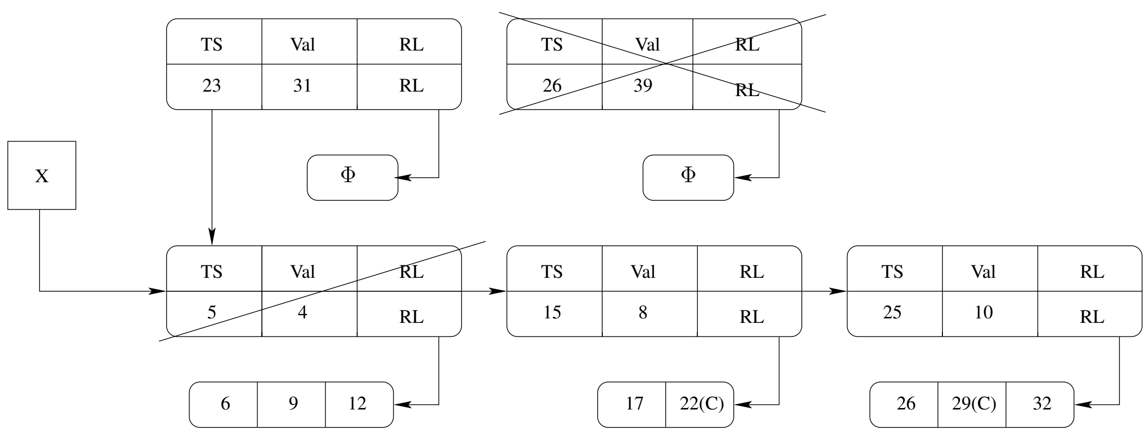


Figure 4: Sample execution of KSTM

In this example suppose t_{26} has the lowest ITS and let t_{29} have a higher ITS. But t_{26} still has to abort due to commit of t_{29}. This shows the drawback of KSTM w.r.t starvation-freedom.

Thus, although t_{26} has lowest ITS, it has to abort due to t_{29} which has higher CTS. Suppose there was no transaction with higher CTS than t_{26}. Then, it can be seen that t_{26} can not abort since it has lowest ITS and highest CTS.

Thus, the key observation here is that a transaction with lowest ITS and highest CTS can not abort. We plan to capitalize on this property to build MV-SFTM.

5 Discussion and Conclusion

Software Transactional Memory systems (*STMs*) have garnered significant interest as an elegant alternative for addressing synchronization and concurrency issues in multi-core systems.

In order to be efficient, STMs must guarantee some progress properties. In this paper, we explored the notion of starvation-freedom [11, chap 2] for TM systems. To the best of our knowledge, starvation-freedom has not yet been explored in the context of STMs.

We presented a starvation-free STM system, SV-SFTM using single versions. It is based on FOCC, a popular algorithm in databases.

It was observed that more read operations succeed by keeping multiple versions of each object [13, 15]. Since SV-SFTM does not consider multiple versions, we observed that it is possible that a slow running old transaction can cause several newer transactions to abort while ensuring starvation-freedom. To address this issue, we proposed KSTM, a MVSTM that maintains fixed number of versions.

But, KSTM does not guarantee starvation-freedom. By understanding the cases where KSTM fails to provide starvation-freedom, we plan to develop MV-SFTM. The key observation in working of KSTM is that a transaction with lowest ITS and highest CTS can not abort. We plan to use this to develop MV-SFTM.

References

[1] Hagit Attiya, Alexey Gotsman, Sandeep Hans, and Noam Rinetzky. Safety of Live Transactions in Transactional Memory: TMS is Necessary and Sufficient. In *Distributed Computing - 28th International Symposium, DISC 2014, Austin, TX, USA, October 12-15, 2014. Proceedings*, pages 376–390, 2014.

[2] Hagit Attiya and Eshcar Hillel. A Single-Version STM that is Multi-Versioned Permissive. *Theory Comput. Syst.*, 51(4):425–446, 2012.

[3] Philip A. Bernstein and Nathan Goodman. Multiversion Concurrency Control: Theory and Algorithms. *ACM Trans. Database Syst.*, 8(4):465–483, December 1983.

[4] Joao Cachopo and Antonio Rito-Silva. Versioned boxes as the basis for memory transactions. In *OOPSLA 2005 Workshop on Synchronization and Concurrency in Object-Oriented Languages (SCOOL)*, oct 2005.

[5] Simon Doherty, Lindsay Groves, Victor Luchangco, and Mark Moir. Towards Formally Specifying and Verifying Transactional Memory. In *REFINE*, 2009.

[6] Sérgio Miguel Fernandes and Joao Cachopo. Lock-free and Scalable Multi-version Software Transactional Memory. In *Proceedings of the 16th ACM symposium on Principles and practice of parallel programming*, PPoPP '11, pages 179–188, New York, NY, USA, 2011. ACM.

[7] Rachid Guerraoui and Michal Kapalka. On the Correctness of Transactional Memory. In *PPoPP '08: Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, pages 175–184, New York, NY, USA, 2008. ACM.

[8] Rachid Guerraoui and Michal Kapalka. *Principles of Transactional Memory, Synthesis Lectures on Distributed Computing Theory*. Morgan and Claypool, 2010.

[9] Maurice Herlihy. Wait-free Synchronization. *ACM Trans. Program. Lang. Syst.*, 13(1):124–149, January 1991.

[10] Maurice Herlihy and J. Eliot B.Moss. Transactional memory: Architectural Support for Lock-Free Data Structures. *SIGARCH Comput. Archit. News*, 21(2):289–300, 1993.

[11] Maurice Herlihy and Nir Shavit. *The Art of Multiprocessor Programming, Revised Reprint*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition, 2012.

[12] Damien Imbs, José Ramon de Mendivil, and Michel Raynal. Brief announcement: virtual world consistency: a new condition for STM systems. In *PODC '09: Proceedings of the 28th ACM symposium on Principles of distributed computing*, pages 280–281, New York, NY, USA, 2009. ACM.

[13] Priyanka Kumar, Sathya Peri, and K. Vidyasankar. A TimeStamp Based Multi-version STM Algorithm. In *ICDCN*, pages 212–226, 2014.

[14] Petr Kuznetsov and Sathya Peri. Non-interference and Local Correctness in Transactional Memory. In *ICDCN*, pages 197–211, 2014.

[15] Li Lu and Michael L. Scott. Generic multiversion STM. In *Distributed Computing - 27th International Symposium, DISC 2013, Jerusalem, Israel, October 14-18, 2013. Proceedings*, pages 134–148, 2013.

[16] Christos H. Papadimitriou. The serializability of concurrent database updates. *J. ACM*, 26(4):631–653, 1979.

[17] Dmitri Perelman, Anton Byshevsky, Oleg Litmanovich, and Idit Keidar. SMV: Selective Multi-Versioning STM. In *DISC*, pages 125–140, 2011.

[18] Nir Shavit and Dan Touitou. Software Transactional Memory. In *PODC '95: Proceedings of the fourteenth annual ACM symposium on Principles of distributed computing*, pages 204–213, New York, NY, USA, 1995. ACM.

[19] Robert H. Thomas. A majority consensus approach to concurrency control for multiple copy databases. *ACM Trans. Database Syst.*, 4(2):180–209, June 1979.

[20] Gerhard Weikum and Gottfried Vossen. *Transactional Information Systems: Theory, Algorithms, and the Practice of Concurrency Control and Recovery*. Morgan Kaufmann, 2002.

Appendices

A PCode of SV-SFTM

Data Structure: We start with data-structures that are local to each transaction. For each transaction T_i:

- RS_i(read-set): It is a list of data tuples (d_tuples) of the form $\langle x, val \rangle$, where x is the t-object and v is the value read by the transaction T_i. We refer to a tuple in T_i's read-set by $RS_i[x]$.

- WS_i(write-set): It is a list of (d_tuples) of the form $\langle x, val \rangle$, where x is the tobj to which transaction T_i writes the value val. Similarly, we refer to a tuple in T_i's write-set by $WS_i[x]$.

In addition to these local structures, the following shared global structures are maintained that are shared across transactions (and hence, threads). We name all the shared variable starting with 'G'.

- G_tCntr (counter): This a numerical valued counter that is incremented when a transaction begins

For each transaction T_i we maintain the following shared time-stamps:

- G_lock_i: A lock for accessing all the shared variables of T_i.

- G_its_i (initial timestamp): It is a time-stamp assigned to T_i when it was invoked for the first time.

- G_cts_i (current timestamp): It is a time-stamp when T_i is invoked again at a later time. When T_i is created for the first time, then its G_cts is same as its ITS.

- G_valid_i: This is a boolean variable which is initially true. If it becomes false then T_i has to be aborted.

- G_state_i: This is a variable which states the current value of T_i. It has three states: `live`, `committed` or `aborted`.

For each data item x in history H, we maintain:

- $x.val$ (value): It is the successful previous closest value written by any transaction.

- `rl` (readList): rl is the read list consists of all the transactions that have read it.

Algorithm 2 STM $init()$: Invoked at the start of the STM system. Initializes all the data items used by the STM System

G_tCntr = 1;
for all data item x used by the STM System **do**
  add $\langle 0, nil \rangle$ to $x.val$;// T_0 is initializing x
end for;

Algorithm 3 STM $tbegin(its)$: Invoked by a thread to start a new transaction T_i. Thread can pass a parameter its which is the initial timestamp when this transaction was invoked for the first time. If this is the first invocation then its is nil. It returns the tuple $\langle id, G\_cts \rangle$

i = unique-id; // An unique id to identify this transaction. It could be same as G_cts
if ($its == nil$) **then**
  $G\_its_i = G\_cts_i = G\_tCntr.get\&Inc()$;
  // $G\_tCntr.get\&Inc()$ returns the current value of G_tCntr and atomically increments it
else
  $G\_its_i = its$;
  $G\_cts_i = G\_tCntr.get\&Inc()$;
end if
$RS_i = WS_i = null$;
G_state_i = `live`;
$G\_valid_i = T$;
return $\langle i, G\_cts_i \rangle$

Algorithm 4 STM $read(i, x)$: Invoked by a transaction T_i to read x. It returns either the value of x or $\mathcal{A}$

if ($x \in RS_i$) **then** // Check if x is in RS_i
    return $RS_i[x].val$;
else if ($x \in WS_i$) **then** // Check if x is in WS_i
    return $WS_i[x].val$;
else// x is not in RS_i and WS_i
    lock x; lock G_lock_i;
    if ($G\_valid_i == F$) **then**
        return abort(i);
    end if
    // Find available value from $x.val$, returns the value
    $curVer = findavilval(G\_cts_i, x)$;
    $val = x[curVer].v$; add $\langle x, val \rangle$ to RS_i;
    add T_i to $x[curVer].rl$;
    unlock G_lock_i;
    unlock x;
    return val;
end if

Algorithm 5 STM $write_i(x, val)$: A Transaction T_i writes into local memory

Append the $d\_tuple\langle x, val \rangle$ to WS_i.
return ok;

Algorithm 6 STM $tryC()$: Returns ok on commit else return Abort

// The following check is an optimization which needs to be performed again later
Set<int> TSet $\leftarrow \phi$ // TSet storing transaction Ids
for all $x \in WS_i$ **do**
    lock x in pre-defined order;
    for <each transaction t_j of [x].rl> **do**
        TSet = [x].rl
    end for
    TSet = TSet $\cup \{t_i\}$
end for// $x \in WS_i$
lock G_lock_i;
if ($G\_valid_i == F$) **then** return abort(i);
else
    Find LTS in TSet // lowest time stamp
    if (TS(t_i) == LTS) **then**
        for <each transaction t_j of [x].rl> **do**
            $G\_valid_j \leftarrow$ false
            unlock G_lock_j;
        end for
    else
        return abort(i);
    end if
end if
// Store the current value of the global counter as commit time and increment it
$comTime = G\_tCntr.get\&Inc()$;

for all $x \in WS_i$ **do**
replace the old value in $x.\mathtt{vl}$ with $newValue$;
end for
G_state_i = `commit`;
unlock all variables;
return $\mathcal{C}$;

Algorithm 7 $abort(i)$: Invoked by various STM methods to abort transaction T_i. It returns $\mathcal{A}$

$G\_valid_i = F$; G_state_i = `abort`;
unlock all variables locked by T_i;
return $\mathcal{A}$;

B Pcode of KSTM

Algorithm 8 STM $init()$: Invoked at the start of the STM system. Initializes all the tobjs used by the STM System

G_tCntr = 1;
for all x in $\mathcal{T}$ **do** // All the tobjs used by the STM System
add $\langle 0, 0, nil \rangle$ to $x.\mathtt{vl}$; // T_0 is initializing x
end for;

Algorithm 9 STM $tbegin(its)$: Invoked by a thread to start a new transaction T_i. Thread can pass a parameter its which is the initial timestamp when this transaction was invoked for the first time. If this is the first invocation then its is nil. It returns the tuple $\langle id, G\_cts \rangle$

i = unique-id; // An unique id to identify this transaction. It could be same as G_cts
// Initialize transaction specific local & global variables
if $(its == nil)$ **then**
// $G\_tCntr.get\&Inc()$ returns the current value of G_tCntr and atomically increments it
$G\_its_i = G\_cts_i = G\_tCntr.get\&Inc()$;
else
$G\_its_i = its$;
$G\_cts_i = G\_tCntr.get\&Inc()$;
end if
$RS_i = WS_i = null$;
G_state_i = `live`;
$G\_valid_i = T$;
return $\langle i, G\_cts_i \rangle$

Algorithm 10 STM $read(i, x)$: Invoked by a transaction T_i to read tobj x. It returns either the value of x or $\mathcal{A}$

if $(x \in RS_i)$ **then** // Check if the tobj x is in RS_i
  return $RS_i[x].val$;
else if $(x \in WS_i)$ **then** // Check if the tobj x is in WS_i
  return $WS_i[x].val$;
else// tobj x is not in RS_i and WS_i
  lock x; lock G_lock_i;
  if $(G\_valid_i == F)$ **then** return abort(i);
  end if
  // findLTS: From $x.\texttt{vl}$, returns the largest time-stampvalue less than G_cts_i. If no such version exists, it returns nil
  $curVer = findLTS(G\_cts_i, x)$;
  if $(curVer == nil)$ **then** return abort(i); // Proceed only if $curVer$ is not nil
  end if
  $val = x[curVer].v$; add $\langle x, val \rangle$ to RS_i;
  add T_i to $x[curVer].rl$;
  unlock G_lock_i; unlock x;
  return val;
end if

Algorithm 11 STM $write_i(x, val)$: A Transaction T_i writes into local memory

Append the $d\_tuple\langle x, val \rangle$ to WS_i.
return ok;

Algorithm 12 STM $tryC()$: Returns ok on commit else return Abort

// The following check is an optimization which needs to be performed again later
lock G_lock_i;
if $(G\_valid_i == F)$ **then**
  return abort(i);
end if
unlock G_lock_i;
$largeRL = allRL = nil$; // Initialize larger read list (largeRL), all read list (allRL) to nil
for all $x \in WS_i$ **do**
  lock x in pre-defined order;
  // findLTS: returns the version with the largest time-stampvalue less than G_cts_i. If no such version exists, it returns nil.
  $prevVer = findLTS(G\_cts_i, x)$; // prevVer: largest version smaller than G_cts_i
  if $(prevVer == nil)$ **then** // There exists no version with time-stampvalue less than G_cts_i
    lock G_lock_i; return abort(i);
  end if
  // **getLar**: obtain the list of reading transactions of $x[prevVer].rl$ whose G_cts is greater than G_cts_i
  $largeRL = largeRL \cup getLar(G\_cts_i, x[prevVer].rl)$;
end for// $x \in WS_i$
$relLL = largeRL \cup T_i$; // Initialize relevant Lock List (relLL)
for all $(T_k \in relLL)$ **do**
  lock G_lock_k in pre-defined order; // Note: Since T_i is also in $relLL$, G_lock_i is also locked
end for
// Verify if G_valid_i is false

if $(G\_valid_i == F)$ **then**
  return abort(i);
end if
$abortRL = nil$ // Initialize abort read list (abortRL)
// Among the transactions in T_k in $largeRL$, either T_k or T_i has to be aborted
for all $(T_k \in largeRL)$ **do**
  if $(isAborted(T_k))$ **then** // Transaction T_k can be ignored since it is already aborted or about to be aborted
    continue;
  end if
  if $(G\_cts_i < G\_cts_k) \wedge (G\_state_k == \texttt{live})$ **then**
    // Transaction T_k has lower priority and is not yet committed. So it needs to be aborted
    $abortRL = abortRL \cup T_k$; // Store T_k in abortRL
  else// Transaction T_i has to be aborted
    return abort(i);
  end if
end for
// Store the current value of the global counter as commit time and increment it
$comTime = G\_tCntr.get\&Inc()$;
for all $T_k \in abortRL$ **do** // Abort all the transactions in abortRL
  $G\_valid_k = F$;
end for
// Having completed all the checks, T_i can be committed
for all $(x \in WS_i)$ **do**
  $newTuple = \langle G\_cts_i, WS_i[x].val, nil \rangle$; // Create new v_tuple: G_cts, val, `rl` for x
  if $(|x.vl| > k)$ **then**
    replace the oldest tuple in x.`vl` with $newTuple$; // x.`vl` is ordered by time stamp
  else
    add a $newTuple$ to $x.vl$ in sorted order;
  end if
end for// $x \in WS_i$
G_state_i = `commit`;
unlock all variables;
return $\mathcal{C}$;

Algorithm 13 $isAborted(T_k)$: Verifies if T_i is already aborted or its G_valid flag is set to false implying that T_i will be aborted soon

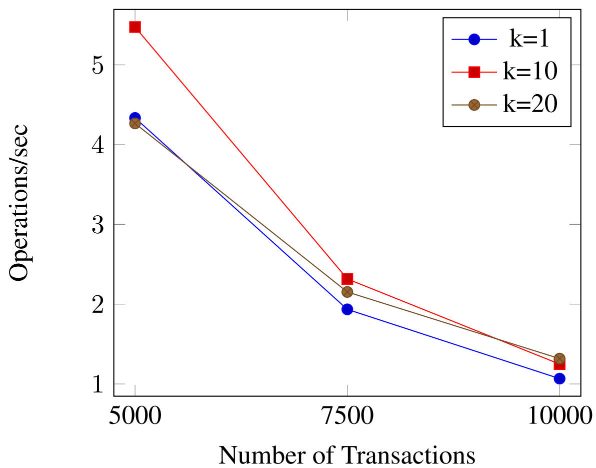
if $(G\_valid_k == F) \vee (G\_state_k == \texttt{abort}) \vee (T_k \in abortRL)$ **then**
  return T;
else
  return F;
end if

Algorithm 14 $abort(i)$: Invoked by various STM methods to abort transaction T_i. It returns $\mathcal{A}$

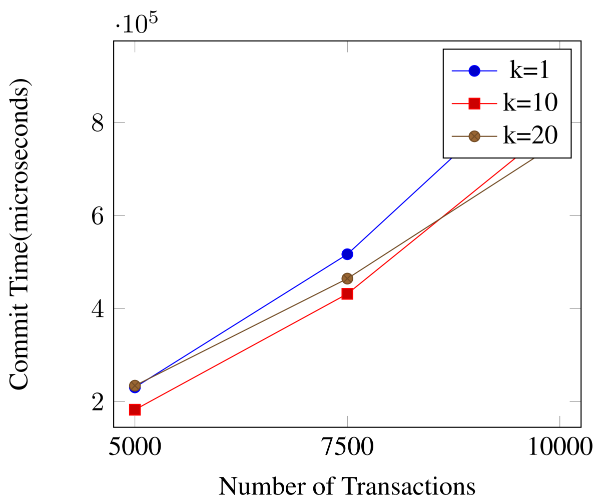
$G\_valid_i = F$; G_state_i = `abort`;
unlock all variables locked by T_i;
return $\mathcal{A}$;

C Some Preliminary Results

The below graphs have been produced by using a linked list application to compare the performance of KSTM with different values of k. In the application chosen below, there were 90% lookups and remaining were 9:1 ratio of inserts and deletes. Varying number of threads were generated and each thread in turn generated 100 transactions.

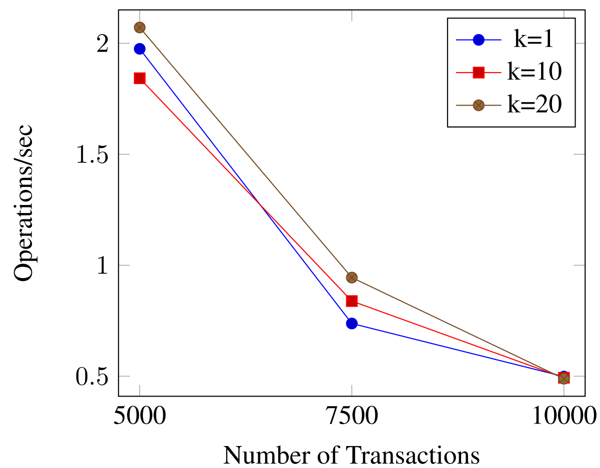


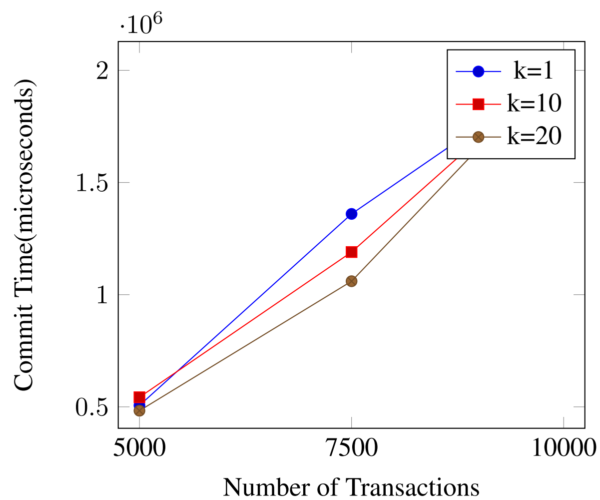
As per the results obtained, multiversion performs better than single version STM. This is because the multiple versions used in KSTM decreases the number of aborts per transaction, thereby effectively increasing the operations/sec performed.



The commit time (time taken per transaction to commit) observed during KSTM (k = 10 here) is the least since is inversely proportional to the operations/sec. As the number of transactions are increasing, they need more versions to read from, to attain higher concurrency leading to lesser abort counts.

In the application chosen below, there were 50% lookups and remaining were 9:1 ratio of inserts and deletes into the linked list. This kind of setup will have more read-write conflicts between the transactions involved when compared to the previous setup.





As per the graph, k = 20 gives the best operations/sec and the least commit time. Hence, having multiple versions(KSTM) performs better than single version STM in this setup too.