1 Abstract

Transactional Memory (TM) is a promising concurrent programming abstraction that has been widely spread and researched in computer engineering. TM is a good substitute for the conventional concurrent programming models, which significantly reduces the complexity of writing code. TM implementations are divided into two different types: Software Transactional Memory (STM) and Hardware Transactional Memory (HTM), in this paper, the STM approach is adopted.

All TM implementations must guarantee transaction atomicity and some kind of progress guarantee. In order to ensure atomicity and progress, transaction collisions must be detected and upon a conflict, the STM must resolve the collision by deciding which of the transitions are to abort. The latter operation is usually done by an integrated Contention Manager Module (CM).

We consider three algorithms that improve the efficiency of STM by serializing conflicted transactions: CAR-STM – Collision Avoidance and Resolution, Bimodal and LO-SER – Low Overhead Serializing. We also present a novel algorithm (TSER – Transaction Serialization) that combines the benefits of the studied implementations while omitting some of their bad properties.

T-SER adopts the serialization technique using condition variables, as done in LO-SER, while alternating between incoming, reading and writing epochs to get the benefit of running RO transactions in parallel as in Bimodal.

We begin with a description of TSER, we then present its pseudo-code and close by providing formal proofs for the STM’s correctness and discuss some of its properties.

2 Introduction

The current concurrent computer engineering trend is adopting transactions to manage access to shared variables or resources. Both software and hardware implementations have been proposed. From the hardware perspective, new computer architectures and new instructions to adopt transactions have been proposed. The software perspective proposes libraries that provide the functionality necessary for transactions by using conventional concurrent programing primitives. We will adopt the latter solution in our paper.

Transactional memory has been widely researched since it is a new abstraction in concurrent programing where locks are not necessary; hence, all lock related bugs will not occur once this abstraction is adopted in concurrent programing. As an example, all bugs related to fine grain locks will not happen since transactions abstract away the name of the locks, and all deadlock and atomicity violation bugs that occur due to unlocking the wrong lock will not happen.

All TM implementations must guarantee transaction atomicity and some kind of progress guarantee. In order to ensure atomicity and progress, transaction collisions must be detected and upon a conflict, the STM must resolve the collision by deciding which of the
transitions are to abort. The latter operation is usually done by an integrated Contention Manager Module (CM). This module will hold the functionality necessary to resolve conflicts while letting all transactions commit eventually.

We consider three STM implementations and present a novel STM that combines their benefits while omitting some of their bad properties. The first is CAR-STM (Collision Avoidance and Resolution) [1]. It contains per-core transaction queues to serialize collided transactions. Instead of letting transactions collide over and over again, upon detecting the first collision, the CAR-STM CM aborts one transaction and moves it to the queue of the winning transaction; this mechanism effectively serializes the execution of the two transactions and ensures that they will not collide again.

The second is Bimodal [2], which is specially tailored for workloads that consist of read-only transactions and early-write transactions (transactions that start to write at the beginning of the transaction). Bimodal contains per-core transaction queues to serialize collided writing transactions (as in CAR-STM) and a central queue, called RO-queue, to enqueue transactions that have not performed a write to a shared variable. The unique architecture of Bimodal, allows it to alternate between read and write epochs thus running only RO transactions in parallel in the read epochs. Bimodal has an advantage over CAR-STM since it does not over-serialize transactions (it does not serialize transactions that are read-only after another read-only transaction which clearly can occur in CAR-STM).

The third is called LO-SER [3] (Low Overhead Serializing Algorithm). LO-SER adopts a new technique for serializing collided transactions. It lets the loser transaction wait on the condition variable of the winner transaction. It is clear that there is no over-serialization; the only serialization that may occur is the result of two actual transactions colliding.

We present a novel algorithm called T-SER that adopts the serialization technique using condition variables as in LO-SER and while alternating between incoming, reading and writing epochs to get the benefit of running RO transactions in parallel as in Bimodal.

We start by describing the STMs that we considered in section 3, we then move to TSER. A description of the STM’s functionality is given in section 4 followed by the pseudo code in section 5. In section 6 proofs for TSER’s correctness and some of its properties are provided. At the end, we discuss possible future work in section 6.

3 Overview

In this section we describe three STMs and state their points of strength and the areas for improvements. We also discuss how they contribute to our algorithm.

3.1 CAR-STM

CAR-STM (a scheduling-based mechanism for STM collision avoidance and resolution) is a scheduling based mechanism that serializes transactions upon collision. Every core has its own transactions queue that contains the transactions for the core to execute one after the other. When two transactions collide and the contention manager (CM) decides that one of them wins and the other loses, the loser transaction is serialized after the winner
transaction. This action is done by enqueueing the loser transaction to the transaction queue of the winner core; this is referred to as *serializing contention managers*. This mechanism is adopted by our algorithm TSER.

CAR-STM also uses a conflict-probability (CP) method to compute the probability of collision with another transaction in order to decide whether or not to serialize and which of the transaction queues to choose; this mechanism is referred to as *proactive collision reduction*.

As described in Bimodal, CAR-STM may over serialize transactions. The main reason is that CAR-STM, unlike Bimodal, does not distinguish between reading and writing transactions, therefore it might serialize reading transactions instead of running them concurrently like in Bimodal. The use of queues also naturally leads to over serialization (see sub-section 1.2).

### 3.2 Bimodal

Bimodal is a scheduler that also resorts to serialization upon a conflict, in order to diminish, and in most cases eliminate, repeated conflicts between the same transactions. Bimodal is able to distinguish between reading and writing transactions, therefore being able to give different priorities depending on such types. The incentive of Bimodal is that usually many workloads have a high number of read-only transactions, and attempting to serialize in every conflict may result in serializing read transactions that shouldn’t wait for each other.

In the model described, each core has its own queue, where transactions are dequeued in order to be executed. As transactions arrive to the system, they are dispatched in a Round Robin fashion to the different cores, placing them in their queue. All cores also share one different queue, meant for read-only transactions, from which each core is able to dequeue. In the initial state of the system, this queue remains empty.

Bimodal alternates between 2 modes, or epochs, that differ in the priority that is given to the two different types of transactions. The first epoch is called writing epoch, when a writing transaction is given priority upon a conflict, over a reading transaction. The second is called reading epoch, and while it takes place, reading transactions are given priority upon a conflict. If two writing transactions conflict, the oldest one prevails.

Whenever a transaction has to be aborted, instead of letting it restart immediately, it is enqueued in one of the different queues. The destination queue depends on the type of transaction: writing transactions are placed in the queue of the core where the winner transaction is being executed, therefore effectively serializing it. Reading transactions are placed in the shared Read-Only queue. Therefore, this last Read-Only queue is populated only with transactions that are read-only (or at least didn’t write yet).

Thus, a core has 2 sources of transactions: its own queue and the general Read-Only queue. In the write epoch, each core reads from its own queue, whereas in the read epoch, the cores dequeue from the shared Read-Only queue. Using this mechanism, Bimodal avoids
serializing read transactions in each core. When the read epoch arrives, those read transactions are dequeued concurrently.

In a write epoch, whenever there are m transactions (m being the number of cores) in the Read-Only queue, or all the queues of the cores are empty, a new read epoch takes place. In a read epoch, whenever m transactions are dequeued from the Read-Only queue, or it becomes empty, a new write epoch takes place. It is easy to see that when Bimodal switches to a new epoch, there might be still transactions running on the cores. Thus, transactions of different epochs can be executed at the same time. This is defined as epoch overlap. When transactions of different epochs overlap, the transaction of the oldest epoch prevails.

During the read epoch, fewer conflicts are expected to occur. However, they are still possible since a read transaction can conflict with a writing transaction of another epoch, or simply because a read transaction of the Read-Only queue might decide to write. Write transactions are spotted only in the moment that they write, so this situation is possible and it is defined as a false positive.

Bimodal serializes write transactions avoiding serializing those that read. This way, read transactions run concurrently in read epochs and don’t wait in line, placed in the queue of a single core. However, those read transactions can be executed again in the system when their conflicted writing transactions are still running, consequently repeating the same conflict. In workloads where transactions have different write transaction durations and very short reads, the same conflicts can be repeated.

Serializing write transactions on a queue also leads to over serialization, since if n write transactions conflict with a transaction, say T, all those n transactions will wait for each other in line, in the order they collided with T.

3.3 LO-SER

LO-SER is a Low Overhead transaction serialization mechanism which, just as CAR-STM and Bimodal, serializes conflicted transactions to avoid as much as possible the occurrence of the same conflict. The work is presented in [3] along with a whole set of adaptive algorithms that apply LO-SER only when enough contention is detected. Those mechanisms vary from partially adaptive, where the LO-SER serialization is applied to a transaction once that transaction collides k times (being k a parameter of the system), to fully adaptive mechanisms, where general statistics of the system are collected to compute a general system contention level. Once that contention level surpasses a given threshold, LO-SER is applied.

The core of the work is precisely the LO-SER algorithm, which is in charge of the serialization of transactions. This approach avoids the use of queues which, as shown above in CAR-STM and Bimodal, still serialize more than might be needed. Instead, the serialization is based on condition variables. Each transaction owns a condition variable, and after a conflict between two transactions, the aborted transaction waits on the condition variable of the winner. Due to the inherent properties of condition variables, it is possible to serialize the set S of transactions that conflicted with a transaction T, after such transaction T. They
will be released altogether when T commits, therefore being able to be executed again, without any transaction in S serialized unnecessarily after another transaction of the same set. LO-SER is mounted over a known STM platform and after releasing a transaction, the algorithm sends it to the underlying STM where it is managed. Thus, how and when this released transaction is restarted is the responsibility of the STM.

This serialization method poses a great advantage to enqueuing, but if it is applied strictly to every aborted transaction, it may lead to waiting cycles where transactions wait indefinitely for each other inside a loop. To prevent this situation, the authors avoid waiting for a transaction that is in turn waiting for another. If a transaction finds itself in that circumstance, it is not serialized at all and continues, being able to be restarted immediately. This can cause in some situations the recurrence of the same conflict if the transaction manages to restart while the conflicted transaction is still running. It is also a conservative solution, since not always waiting for a winner transaction that is in turn waiting will create a loop, but the wait is avoided in all cases where the winner transaction is waiting.

LO-SER loses control over the transactions after they are released, and delivers them to the underlying STM that is not aware of the waiting process and penalty they already have suffered. Thus, the STM is unable to give them a different treatment in case they need it to overcome the penalty they already drag. Combining it with a transaction scheduler aware of this issue might be an interesting analysis.

This mechanism does not separate reading and writing transactions and therefore suffers from the same drawback as CAR-STM, causing read-only transactions not to work well in bimodal and read-dominated workloads.

4 T-SER Description

Based on the work previously described, the T-SER algorithm is presented. The motivation of T-SER is to create transaction serialization mechanism that does not use queues to serialize transactions, avoiding the natural over serialization of those data structures, while being able to separate reading and writing transactions in order to parallelize reads as much as possible. It also adds a transaction scheduler that is aware of the penalty that transactions may suffer due to serialization.

T-SER assumes a basic STM functionality running underneath and taking care of the basic functions of transaction management, basically collision detection and contention management. This STM will be seen as a black box that will send events to T-SER, mainly whenever a collision is detected or a transaction is aborted/committed. A conservative CM is assumed to exist, and upon a collision event, it will decide which transactions to abort. In such event, collision information will be given, such as whether the losing transaction wrote or read.

As mentioned before, T-SER takes care of the transaction scheduling, receiving the arriving transactions in buffers and taking control over the dispatching of transactions to the cores. This scheduler keeps three different buffers:
○ A buffer for the transactions that are arriving (incoming)
○ A buffer for old writing transactions that conflicted
○ A buffer for old reading transactions that conflicted

The different buffers exist in order to separate different kinds of transactions, with the intention of scheduling them differently depending on their nature. Read-conflicted transactions will be put together in one buffer, separated from the writes, in order to maximize the parallelism between reads. The buffer for new transactions is used to separate them from the ones that already conflicted: this gives preference in the system to transactions that arrived before and are trying to commit after a conflict. Using these different buffers, the scheduler is able to give preference to conflicted transactions over new ones, and thus it appeases the penalty of transaction serialization.

It is clear to see that this scheduler mechanism keeps different sources of transactions. In order to schedule them properly, the epoch idea of BIMODAL is borrowed. The scheduler will dispatch transactions from the three different buffers based on three epochs. These epochs are based on the number of cores in the system, namely m.

There will be a read epoch in which m transactions will be dispatched from the read buffer. After m transactions (or empty queue) a write epoch will take place. In this write epoch, m transactions will be dispatched from the write buffer. After m transactions (or empty queue) another read epoch will take place.

Whenever both queues are empty, an incoming epoch will start, in which transactions from the incoming buffer are dispatched. Whenever m transactions are placed in the read or write buffers, another read or write epoch (respectively) will take place. If m incoming transactions are dispatched and either the read or write queues are not empty a read or write epoch will take place, respectively. It is clear to see that in T-SER, just like in BIMODAL, epochs can overlap in the system.

Those different epochs also define the priorities that are given to conflicted transactions. Hence, the Contention Manager will take into account the current epoch to make a decision. Working in a similar way to the Contention Manager in BIMODAL, during the read epoch the priority will be given to read transaction, whereas in write epochs, the write transactions will prevail. In a similar way, new transactions will win the conflict in incoming epochs, in order not to penalize those transactions too much. Whenever there is a conflict between two transactions of the same type and epoch (being this the current epoch or not), the older transaction will prevail. In order for this to be possible, transactions need to be given a timestamp at the beginning, in the moment they arrive to the system. The transaction data structure in T-SER has a timestamp field named origin for this purpose. If the collision takes place between transactions of different epochs, the one of the older epoch will win. Therefore, it will also be necessary to give a timestamp to each epoch.

T-SER is based on a Contention Manager that behaves in the way previously outlined. However, in this document only the description will be given, as providing a detailed implementation of a contention manager falls out of the scope of this work.
Upon a conflict between 2 transactions, at least one transaction is aborted. Once this happens, the transaction is serialized after the winner transaction in order to wait until this last transaction finishes. T-SER uses condition variables so the loser transactions wait on the condition variable of the winner. Condition variables are a means for transaction serializing that enables multiple transactions to wait for the winner, thus being released as soon as the winner finishes, avoiding over serializing.

When a transaction commits, it will release all the transactions that were waiting for it. Those released transactions will enqueue themselves in the scheduler, which will place them in either the write or read buffer according to their type. Using this mechanism, transactions are properly serialized after the colliding transaction and at the same time, read transactions are treated differently after waiting, and are run concurrently to maximize their parallelism.

The scheduling and serializing mechanisms of T-SER are divided into three different components: the events, the transaction runner threads and the scheduler object. In order for them to access and store the information they need, a set of fields are added to the transaction information. These fields include a condition variable for other transactions to wait on it and be serialized, a lock used for synchronization of the condition variable, a status field described later in the section, the epoch to which the transaction belongs and several basic fields such as a timestamp and a set of flags.

### 4.1 T-SER EVENTS

The first part of the code to be described is the part related to events. T-SER receives from the underlying STM three particular events that will be used to manipulate the transactions, serialize and schedule them. These events are the collision between two transactions as well as the abort and commit actions of each transaction.

When a conflict is detected, the Contention Manager is called to decide which transaction to abort. Then the status field of the aborted transaction is changed. This field is composed by four values aggregated in an Integer. Those values are:

1. The current outcome of the transaction (COMMIT or ABORT)
2. Whether the conflict was generated by a read or write action
3. The transaction that won in case the outcome is abort
4. The timestamp of that winning transaction when the conflict occurred

If the result of the conflict is to abort the transaction of the current thread (abort itself), the status is changed accordingly. However, if the loser is the transaction running in another process, a previous check is made to ensure the transaction is still active and if so, a CAS operation is used to change its status, ensuring consistency of the status change and its timestamp if the other process tries to change them at the same time.

When T-SER intercepts the commit event, it advances its timestamp. After that, a new thread is launched with the same context information, which will try to dequeue and run a new transaction. T-SER lets the new and current thread run concurrently on the same core since from that point on, they do not share any variable and therefore there is no chance of collision. After this, the status of the transaction is changed to COMMITTED and the
corresponding timestamp is written, using also a CAS operation to avoid committing if another process aborts that same transaction while the commit phase takes place. If the operation succeeds, the transaction reads its flag named release to check if there is any transaction waiting for it to commit, in which case it broadcasts its condition variable and resets the flag. These last steps are encapsulated within a lock to synchronize the use of the flag and ensure no transaction will wait after the broadcast.

Similarly, when T-SER intercepts the abort event, the timestamp is increased, and then a new thread is launched to try to dequeue a new transaction. If the winner transaction is still active, the aborted transaction will mark its flag waiting and block itself on the condition variable of the winner, marking the flag release of that transaction to indicate it should broadcast its condition variable after committing. All this code is encapsulated within the lock of the winner transaction to synchronize the flags. An additional check is made as well to ensure the timestamp of the winner is the same as it was the last time it was read before the lock, to ensure the winner transaction didn’t commit and broadcast already in the meantime (a transaction increments its timestamp also upon commit). Lastly, before waiting on the condition variable, the aborted transaction ensures the winner has its waiting flag unmarked, to avoid waiting for a transaction that is in turn waiting for another. Avoiding this, T-SER avoids the creation of waiting cycles.

When the waiting process is done and the transaction is released, the code continues, and the aborted transaction is enqueued again in the scheduler object in order to be re-scheduled.

4.2 TRANSACTION RUNNER

The second component of T-SER is the code that each thread will execute in order to run a transaction. Note that it is possible to implement a scheduling in a centralized way, as a thread that gives transactions to the cores. However, it is an interesting analysis to create a distributed implementation where such thread is not used and instead, each core fetches a transaction, advancing the epoch if necessary. This second option has been selected for the implementation and thus, each thread will run code to manage the scheduling before running the transaction. The thread will first execute a busy wait, using the Anderson’s algorithm on the shared variable named busy, which marks the cores that are waiting or have the lock (and therefore trying to dequeue from the scheduler). The busy wait uses an index on the busy variable in order to create a Round Robin order of execution, to let all cores dequeue without underutilizing any of them. If all cores get transactions freely, it could be possible that some always see the queue empty, since the faster processes always dequeue the transactions. Round Robin ensures a fair share.

Once the process exits the busy wait, it will be the only one accessing the next lines of code, which are the lines that manage the current epoch. The method get_epoch of the scheduling object is called to retrieve the current epoch. Using it, the process accesses the corresponding queue. The scheduling object returns an epoch whose queue will not be empty. It will also take care of the epoch advance, returning the next epoch if necessary.
It can be possible that all buffers are empty and the method returns a wait value, indicating the process it must wait until transactions arrive to the system. If that is the case, the process will wait on the condition variable of the scheduling object, and will mark the flag release of such object to inform it that a signal is needed upon the arrival of a transaction. The flag management is done using the lock of the scheduling object, to ensure synchronization. Note that at a given time, only a process is waiting for the condition variable, since the rest are waiting in the busy wait (or running a transaction already). The buffers are checked again after locking, to avoid waiting if a transaction arrived just in the moment the process is trying to lock. After the process is released from the condition variable, there will be transactions in the queues, so a new call to the method is done. This answer will be different than wait (the process is still the only one in this section so nobody else that arrived after it tried to dequeue, therefore no other process "stole" the new incoming transaction).

After that point the code continues, exits the Critical Section advancing the index of this busy shared variable. Then, it dequeues from the corresponding queue. Note that the extraction of transactions from the queues is done concurrently, since the epoch management already takes care of giving each process the correct queue to use, ensuring there will be enough items. This queue uses a concurrent implementation to make it possible for many processes to dequeue at the same time. Once they do so, the epoch obtained is assigned to the transaction, which is immediately executed.

### 4.3 SCHEDULING OBJECT

The scheduling object is the third component of T-SER and takes care of the transaction scheduling as well as the epoch management. The get_epoch implementation in this object takes care of it. Note that this method is always called in a Critical Section using the Round Robin busy wait mentioned above.

This method uses two variables per epoch: one named available to count the transactions that are actually present in the buffer of the epoch, and one named counter to count the transactions left until an epoch change (typically it starts with m). These variables are used to advance the epoch in the way previously defined in the general description. If all buffers are empty the wait value is returned.

The enqueue method will put the transaction in the corresponding buffer depending on the action given as a parameter. It will atomically increment the available variable (done atomically not to collide with the decrement of a dequeue) and signal its condition variable if the flag release is set, releasing the core waiting for transactions to come. This signal is done within the lock of the scheduling object to synchronize the use of the flag.

### 5 T-SER Pseudo-Code

#### Shared Variables

- `Busy[1..#cores] = [1,0..0]` //array marking which cores are busy
- `Last=0`: integer //Round Robin Index. Index to traverse the Busy array

#### Structure of a transaction:
- `C`: cond

---

10
M: mutex
status: integer
id_process: integer
origin: timestamp
ts: timestamp
E: Epoch
release: Boolean
waiting: Boolean

Events:
1 Upon Collision(T, T_Action, T', T'_Action)
2 {
3     result = T.ContentionManager(T')
4     if (result = ABORT_SELF)
5         T.status = <ABORT, T_Action, T', T'.Ts>
6     else if (result = ABORT_OTHER)
7         int s = T'.status
8         if (s.state == ACTIVE)
9             T'.status.CAS(s, <ABORT, T'_Action, T, T.ts>)
10    end if
11 }

12 Upon COMMIT by transaction t
13 {
14     t.ts++
15     Generate new thread for core I of transaction_runner()
16     if (t.status.CAS(<ACTIVE, Action_Any, 0, 0>,
17         <COMMITTED, Action_Any, 0, 0>))
18         {
19             If (t.release)
20             {
21                 t.M.lock()
22                 t.C.broadcast()
23                 t.release = false //Own flag
24                 t.M.unlock()
25             }
26         }
27 }
28
29 Upon ABORT of transaction t
30 {
31     t.ts++
32     Generate new thread for core I of transaction_runner()
33     w = eStat.winner //Get the winner transaction
34     if w.status.state = Active then
35         t.waiting = true
36         w.M.lock()
37         w.release=true
38         if w.ts = t.status.winnerTs and not w.waiting then
39             w.C.wait(w.M)
40         end if
41         w.M.unlock()
42         t.waiting=false
43     end if
44     Sched_obj.enq(t, t_Action)
45 }
transaction_runner()
{
    myPlace = rmw(Last, Last+1 mod n) //Anderson's MUTEX algorithm
    wait until Busy[myPlace] == 1
    Busy[myPlace] = 0

    my_epoch = Sched_obj.get_epoch()
    if my_epoch = wait then
        Sched_obj.L.lock
        Sched_obj.release=true
        if all values in Sched_obj.Available_trans are zero then
            Sched_obj.C.wait(Sched_obj.L)
        else
            Sched_obj.release = false //Avoid fake signaling
        End if
        Sched_obj.L.unlock
        my_epoch = Sched_obj.get_epoch()
        //It won't be wait, since everybody else is waiting
        //and won't dequeue.
    end if
    Busy[myPlace+1 mod n] = 1
    //Safe deq. Epoch variable guarantees there will be values.
    T = Sched_obj.Q[my_epoch].deq()
    T.E = my_epoch
    run T
}

Scheduling Object:

Q = { Incoming = {}, Reading = {}, Writing = {} } //Array with the 3 queues
Available_trans[epoch] = 0 //The number of transactions available in each queue
C //Condition variable to liberate threads waiting for transactions
L //Lock of the condition variable
shared_epoch = <Incoming, m, 0> //The current epoch. Epoch values may be {Incoming, Reading, Writing}
release = false //Boolean. Indicates if there are processes waiting for transactions

get_epoch()
{
    if all values in Available_trans are zero
        return wait
    else if Available_trans[shared_epoch.epoch] = 0 or shared_epoch.counter = 0
        then
            new_epoch.counter = m-1
            new_epoch.epoch = next(shared_epoch.epoch)
            new_epoch.timestamp = new_epoch.timestamp + 1
            dec Available_trans[new_epoch.epoch] //Atomic decrement
            shared_epoch = new_epoch
            return new_epoch
    else if my_epoch.epoch = incoming
        if Available_trans[reading] ≥ m
            shared_epoch.epoch = reading
            shared_epoch.timestamp = shared_epoch.timestamp +1
            shared_epoch. counter = m
        end if
else if $\text{Available}\_\text{tran}[\text{writing}] \geq m$
91 shared\_epoch\_epoch = writing
92 shared\_epoch\_timestamp = shared\_epoch\_timestamp + 1
93 shared\_epoch\_. counter = m
94 end if
95 new\_epoch\_. counter = shared\_epoch\_. counter - 1
96 new\_epoch\_. epoch = shared\_epoch\_. epoch
97 dec $\text{Available}\_\text{tran}[\text{my\_epoch\_epoch}]$  //Atomic decrement
98 return new\_epoch
99 }
100  //Gets the next value of an epoch
101  //Precondition: at least one value in $\text{Available}\_\text{tran}$ is not zero
102 next\(\text{epoch}\) {  103  if epoch = reading
104     set\_epoch = \{writing, incoming, reading\}
105  else if epoch = writing
106     set\_epoch = \{reading, incoming, writing\}
107  else
108     set\_epoch = \{reading, writing, incoming\}
109     end if
110  for each new\_epoch $\in$ set\_epoch do
111     if $\text{Available}\_\text{tran}[\text{new\_epoch}] > 0$
112        return new\_epoch
113     end do
114   //Will eventually return a value due to the precondition
115  }
116  enq\(t, t\_action\) {  117     Q[t\_action].enq\(t\)
118     inc $\text{Available}\_\text{tran}[t\_action]$  //Atomic increment
119     if (release)
120        L.lock
121        //release might have been set back to false while waiting
122        if (release)
123            C.signal()
124        End if
125        release=false
126        L.unlock
127  }
130  }

6 PROOFS

Claim 0: The transactions have a finite duration

Lemma 0: The value of Available[] contains the logical size of each buffer

Proof: With logical size we refer to the size understood by the algorithm, which in some cases may differ from the real physical size of the buffers. More concretely, available is decremented in a different code (get\_epoch) than the code where the item is physically
dequeued from the buffer, and both codes can be run concurrently by different cores. This is made to maximize parallelism and it might cause that get_epoch is called more than once, and hence the value available decremented more than once, before the real dequeue happens. Enqueuing and dequeueing can also be run concurrently, and available is not incremented in the same atomic step where the transaction is actually enqueued in the buffer. What T-SER cares about is the number of transactions that it is able to give to processes, rather than the transactions that are physically in the queue in that instant. The value of Available[] has initial value zero, and incremented in the enqueue (line 121) after placing it in the buffer, being this the only place in the code where transactions are placed in the buffers. The same variable is used as index to access available and Q, so it is ensured that the value in available corresponds to the buffer to enqueue in. Note that the value of t_action in this line will be a valid epoch, since the method is only called in the abort event passing the type of action that caused such abort. This value cannot be null, since the method went through a collision event before the abort event so the value was populated with a valid read or write value, coming from the event information (lines 5 and 9). In case the enqueue is called from the system to place an incoming transaction, it is assumed that the system will give the incoming value. Available[] is decremented in the get_epoch method, before returning the current epoch (line 97) to mark it as taken as soon as the transaction is assigned to the process that invoked the method. This method can only be executed once at a time since it is enclosed in a Critical Section. Available is accessed using the current epoch value, which is straightforward to see in the code that is always assigned a valid epoch. If the described line of code is reached, it is clear to see following the code that the wait value is not returned, so the transaction runner, the only caller of the method, will go through lines 66-69, therefore accessing the buffer with the same index used to access available. This ensures the value of available corresponds to the buffer to dequeue from. Both increment and decrement are done atomically since enqueuing and dequeuing may occur concurrently.

**Theorem 1:** Epoch advancement: If there are non-empty buffers, the transactions of all of them will be dispatched and committed.

**Proof:** Since transactions have a finite duration, eventually each core will finish running each one of them. Their outcome can only be commit or abort, and in both cases the specific event is executed, where a new thread is generated to dequeue a new transaction (lines 15 and 32). Those threads will reach the entry section, and by the non-starvation property of its MUTEX algorithm, they will eventually enter the Critical Section, call get_epoch (line 52) exit the Critical Section and dequeue more transactions.

The current epoch is initialized with an incoming epoch. It is clear to see, following the code of get_epoch and by Lemma 0, that if there are transactions available in at least one buffer, the conditions to return a wait value (line 74) are not met. The variable counter of an epoch holds the transactions that can be still dispatched in the current epoch (it is never incremented, decremented only in line 95, when a transaction is going to be dispatched and assigned in lines 78, 88 and 92 when a new epoch takes place), and by Lemma 0 available holds the transactions available to dispatch in the buffer. As successive calls are made to get_epoch, several configurations may be reached:

- **Case 1:** A configuration C_i where available or counter of current epoch reaches zero. The epoch is advanced to the next one (line 79) calling the method next. Note that the call is always made having at least available transactions in one buffer. Following the code of this method, it can be seen that if the current epoch is reading or
writing, the writing or reading epoch, respectively, is returned if there are 
transactions available. Otherwise, the incoming epoch is attempted to return. If 
there are no transactions still available in incoming, the current epoch will be kept. If 
the current epoch is incoming, the read, or then write epochs are attempted to 
return if they have available transactions, keeping the incoming epoch otherwise.

- Case 2: A configuration C₂ where the current epoch is incoming and the read buffer 
  has m or more transactions. The epoch is advanced to read (line 86) so the read 
  buffer is dispatched. Then line 95 is reached where a transaction of the new epoch is 
  dispatched, decrementing available and the epoch counter.

- Case 3: A configuration C₃ where the current epoch is incoming and the write buffer 
  has m or more transactions. The epoch is advanced to write (line 90) so the write 
  buffer is dispatched. Then line 95 is reached as well as in case 2, decrementing 
  available and counter.

- Case 4: A configuration C₄ where none of the cases above are met. Following the 
  code, line 95 is reached directly, dispatching a transaction of the current epoch, and 
  also decrementing available and the epoch counter. If more items are enqueued, the 
  available counter will be incremented, but the epoch counter is never incremented 
  anywhere in the code. It is only assigned a new value but only if there is an epoch 
  switch (lines 78, 88 and 92). This ensures that eventually at least case one will occur.

As it can be seen in the different case analysis, either available or the epoch counter (or both) will eventually reach zero if there is no epoch switch, thus no epoch will be kept forever. The epochs advance giving priority to read and write until they are empty. And then the incoming epoch is restored, switching back to reading / writing in the way the cases describe. If available of incoming reaches zero, at least there will be transactions available in any of the other two buffers, so an epoch switch takes place as well. The system will be able traverse through all epochs as long as the read and write buffers are able to be emptied. Once a read or write epoch is reached, the system will alternate between these 2 epochs, not dispatching any new ones from incoming. Note that those buffers are populated only with transactions that conflicted, not being enqueued if they commit. Since the Contention Manager ensures that all transactions will eventually commit, all those transactions will reach the commit state, leaving the system. Therefore, these 2 buffers will not be indefinitely non-empty at the same time, making it possible for the incoming epoch to take place again. Thus, all transactions of all buffers will eventually be dispatched and committed.

**Theorem 2:** If a thread \( t_2 \) finds all buffers empty when trying to dequeue a transaction, it waits until transactions arrive to the system. After waiting, the thread continues and finds at least one transaction in one of the buffers.

**Proof:** Let C be a configuration where all buffers are empty, the release field of the scheduled object is false, the value with which it is initialized, and thread \( t_2 \) is in the Critical Section about to try to dequeue a transaction. Then consider an execution \( \sigma \) starting from C, where \( t_2 \) executes get_epoch (line 52) which returns the value wait, since \( t_2 \) found all values in available zero by Lemma 0. Then, \( t_2 \) acquires the lock of the Scheduling Object. No other transactional thread can acquire the lock since it is a Critical Section, and the enqueuing thread of the system will find release=false. Note that if before checking the value of available, some other process P managed to successfully increment it, the value wait wouldn't be returned, so neither \( t_2 \) would wait nor P would broadcast (P would find release to false). In this point of execution several cases may occur:
• $t_1$ is running solo, so nobody else tries to enqueue while it executes. After locking, $t_1$ sets release to true in the scheduling object and checks the value of available, whose value didn't change since nobody could increment it in the enqueuing. Then, $t_1$ waits on the condition variable of the Scheduling Object waiting for transactions to arrive. As soon as transactions arrive to the system, the method $\text{eqq()}$ will be called, placing at least one transaction in the buffers. The first process that finds the flag release=true, will signal the condition variable in line 126, releasing $t_1$. After being released, $t_1$ will find a transaction in the buffers. The rest of enqueuing processes will find release=false either before or after the lock (lines 122 or 125), so they will bypass the signaling.

• There is a set $S$ of processes running along with $t_1$ each of which is enqueuing a transaction. However, no process manages to increment available before $t_1$ sets release to true (line 55). Let’s consider the execution $\beta$ where at least one enqueuing process in $S$ increments available before $t_1$ checks it again in line 56. It means that it already enqueued a transaction, say $T$. The thread $t_1$ would obtain a value different from zero and therefore it will not wait, changing release to false and liberating the lock, so the enqueuing processes continue without signaling. Thread $t_1$ finds at least the transaction $T$ in the buffers. Now let’s consider an execution $\alpha$ where $t_1$ manages to advance before any process in $S$ increases the value of available, so $t_1$ checks available again. The thread $t_1$ will obtain the same value zero. After $\alpha$ and due to the use of locks, regardless of the action of the processes in $S$, $t_1$ will wait and release the lock. The first process $P$ in $S$ in advancing and acquiring the lock will see release=true (which means that $P$ also has enqueued a transaction, say $T'$, in the buffers) and will advance until it signals the condition variable and resets the release flag. $P$ might or might not wait in the lock of line 123, depending on whether it arrived to it before or after $t_1$ releases it. In any case, $t_1$ will be released. After being released, $t_1$ will find at least $T'$ in the buffers. The rest of the processes in $S$ will see release = false after that. Either in line 122 or 125.

• There is a set $S$ of processes running along with $t_1$ each of which is enqueuing a transaction. At least one process in $S$ manages to increment the value of available before $t_1$ sets release to true. Those threads that have managed to increase available have also placed a transaction in the buffers. The thread $t_1$ will advance and encounter some value in available different than zero in line 56, therefore not waiting and resetting the flag release. Note that $t_1$ will find at least one transaction in the buffers. The enqueuing threads never reach the code to signal in this case, since they might advance before $t_1$ sets release to true and terminate, or will wait in the lock, in which case after waiting, they encounter release=false again in line 125.

In any of the cases above, thread $t_1$ will not wait indefinitely and it is guaranteed to be released if it waits, as long as transactions arrive. The thread will not continue the code if the buffers are empty. After waiting, the thread $t_1$ will find at least one transaction in the buffers.

Correct serialization

Lemma1: Let $t_1$ and $t_2$ be two colliding transactions such as the CM decides to abort $t_2$ and continue to run $t_1$. Then $t_2$ decided to wait on the winner condition variable in its abort code. Transaction $t_2$ is serialized after $t_1$.

Proof: If $t_2$ ran the collision code then the result in line 3 will be $\text{abort\_self}$ and the winner transaction set in the status variable will be set to $t_1$ (line 5). On the other hand, if $t_2$ runs the collision code then the result would be $\text{abort\_other}$ and the winner transaction set in the status variable will be set to $t_1$ (line 9). (the CAS is used in order to make sure that the status of $t_2$ is active when it is updated to abort, the CAS is not needed in line 4, since $t_2$ is
updating its status therefore it is already active). Hence, when t2 gets to line 33, the value of w (the winner transaction) will be t1 and therefore when it waits on the winner’s condition variable it will be t1’s.

Lemma 2: A successfully serialized transaction is eventually released

Proof: By Lemma 1 a transaction t1 is serialized when it aborts, waiting for the winner transaction t2. Since it was serialized, it waits on line 37, which means that it also went through line 37 and set the flag release to true. Since it managed to be serialized, t2 was not waiting for any other transaction and did not commit or abort yet. The transaction t1 waited successfully which means it released the lock when waiting on the condition variable. If any more transactions try to request the lock, whether they manage to wait or not, they will eventually release the lock as well. The Contention Manager is assumed to let all transactions commit eventually, so upon commit of t2, it will find the flag release=true. It will be able to request the lock and broadcast the condition variable, liberating t1.

Theorem 3: If a transaction is successfully serialized, the same conflict will not happen again.

Proof: Note that a successful serialization means that upon a conflict, the aborted transaction waits on the condition variable of the winner, by Lemma 1. Then by Lemma 2 it is released after the transaction is committed. After being released it continues the code (line 44) enqueuing itself in the buffers. By Theorem 1 it will be executed again. However, in this execution, the transaction that conflicted with it has already committed. It is possible that the running transaction enters in a new conflict with another transaction, but it is not possible anymore to clash with the original transaction that caused the serialization.

Lemma3: there cannot be n>2 transactions such that for i=1..n t(i+1 mod n) is waiting in the condition variable of ti.

Proof: assume by a way of contradiction that there exists such a loop in some configuration during the execution.

We start the proof by stating the drawback of such phenomena; the main drawback is that all these transactions will stay forever waiting for each other and will never be released, because that every winner condition variable for each of the waiting transactions is also waiting for another condition variable; hence these transactions will cause their threads to starve.

Now we go back to the proof, consider the last transaction to execute line 35 (t.waiting = true), let us call this configuration C1 and let us assume that t1 is the last transaction (w.l.o.g). Since all the transactions are eventually waiting for each other, then all of them must go through line 35 and get to line 39 to wait. Hence, in some later configuration, say C2, t1 must execute line 38 and get a positive answer in order to wait, which means that it saw that w.waiting = false in contradiction to the assumption that t1 is the last transaction to execute line 35.

Theorem 4: In every configuration of every execution, there are no waiting loops, and if t1 and t2 are two colliding transactions such as the CM decides to abort t2, continues to run t1 and t2 decided to wait on the winner condition variable in its abort code, then t2 is serialized after t1.

Proof: Directly from lemmas 1 & 3.
Correct transaction dispatching

Lemma 4: Transaction runner threads are dispatched one after the other in RR manner and every thread is dispatched for some existing transaction in shed_obj.

Proof: Transaction runner algorithm uses Anderson’s mutex algorithm in order to ensure a Round Robin accesses to transactions. Epoch management is done in the critical section of the algorithm (lines 51 – 65). However, other updates to the shared variable Available may be done in parallel to update it once a transaction is added to the shed_obj (only increment updates the decrement updates are done only by the transaction runner algorithm in the CS). From the correctness of available (Lemma 0), every thread is dispatched for some existing transaction already existing in the shed_obj. Since get_epoch is executed in the CS (no parallel accesses except of incrementing Available) it is clear that the code for the method will be safely executed and that every thread, when dispatched, will execute some existing transaction in the shed_obj.

Definition 1: Processor Stagnation – if there is an execution where there is an infinite number of transactions and some processor never executes any

- In TSER there is no Processor Stagnation. In Lemma 4 it is proved that the transactions are dispatched among the processor in RR manner, therefore all processors have a chance of executing a transaction if there are enough transactions in the system.

Theorem 5: Transactions are dispatched in Round Robin and run transactions concurrently on the processors; enqueues and dequeues of transactions are done in parallel correctly (no dequeue from an empty queue) and all transactions will be eventually dispatched to some processor.

Proof: The transactions are dispatched to the processors in RR manner (lemma 4), furthermore, every dispatched processor is responsible for some existing transaction in the sched_obj. After dispatching the transaction to the process, the process dequeues from the queue of the intended epoch (line 68), sets the epoch of the transaction to its respective epoch and runs it. From Lemma 0, available holds the logical size of each buffer, and all the dequeues will succeed since they dequeue from the epoch assigned to them by get_epoch. This method returns wait or an epoch with available > 0. The enqueues and dequeues are atomic (distributed, atomic queues are used). From Anderson’s algorithm, every free processor will be waiting to dispatch a new transaction (line 48) and from Theorem 2, if the result of some process was wait then in some later configuration it is guaranteed that the processor will get a transaction (if there are new available transactions), which means the processor will not get stuck and hold the system if it decides to wait. Then all transactions will eventually be dispatched to some processor.

7 Future Work

As work to be done in the future, it is possible to introduce a serialization parameter in the algorithm, deciding the number transactions to be serialized one after the other instead of serializing all the collided transactions after the winner transaction. For example, if the serialization parameter is 2, then if transaction t1 is running in the system and t2 is the first transaction to collide with it and lose, it will be serialized after t1. The second transaction to collide with t1 and lose (t3) will be serialized after t2. All other transactions that collide with t1 and lose will be serialized after t3. It's easy to see that if the parameter is equal to 0 then
the algorithm is equivalent to the original TSER. If the parameter is infinity, then it’s somewhat like CAR-STM.

The notion behind this idea is that the probability of collision between two transactions (that already collided on the same base transaction) is smaller as those transactions occur after a longer time period (the same parameter can be defined on the number of instructions executed from the beginning on the base transaction). The parameter may be adaptive to adjust to different workload characteristics.

Both probabilistic analysis of the new algorithm and benchmarking on actual workloads may prove or refute the added efficiency of the parameterized algorithm.

8 References
