International Journal of Networking and Computing – `www.ijnc.org`, ISSN 2185-2847
Volume 8, Number 1, pages 93–123, January 2018

Approximate Consistency in Transactional Memory

Basem Assiri

Jazan University
Jazan, Saudi Arabia
bas0911@hotmail.com

Costas Busch

Louisana State University
Baton Rouge, LA 70803, USA
busch@csc.lsu.edu

Received: July 30, 2017
Revised: October 31, 2017
Accepted: December 12, 2017
Communicated by Susumu Matsumae

Abstract

In *Transactional Memory* each shared object can be accessed by concurrent transactions which may cause conflicts and aborts. *Opacity* is a precise consistency property which maps a concurrent execution to a *legal* sequential execution that preserves the *real time order* of events. However, having precise consistency in large scale network and database applications may result in a large rate of aborts, especially in systems that have frequent memory updates. Actually, high rate of aborts causes huge negative performance impact. In real applications, there are systems that do not require precise consistency especially when the data is not sensitive. Thus, we introduce here the notion of approximate consistency in transactional memory. We define K*-opacity* as a relaxed consistency property where transactions are allowed to read one of the K most recent written values to the objects (and not just the latest value only). This increases the throughput and reduces the abort rate by reducing the chance of conflicts. In multi-version transactional memory, the relaxed consistency allows to save a new object version once every K object updates, and hence, reduces space requirements by a factor of K. In fact, we apply the concept of K-opacity to regular read/write, count, and queue objects, which are common objects used in typical concurrent programs. We use the technique of *writer lists* to keep track of the transactions and the data being written to the system, in order to control the error rate and to prevent error propagation. We illustrate with an experimental analysis the positive impact of our approach on performance, where higher opacity relaxation (higher values of K) increases the throughput and decreases the aborts rate significantly.[1]

Keywords: Transactional Memory, K-opacity, Counting Operation, Queue Operation, Approximate Consistency.

[1]This paper combines the preliminary versions of works that appear in SRMPDS'16 [3] and APDCM'17 [4].

1 Introduction

The evolution of parallel computing [2] is confronted by the challenge of concurrent memory accesses. Transactional memory (TM) enables concurrent processing by offering transactional support to concurrent access of shared data [14]. A transaction is a finite sequence of machine operations that access local and shared memory. The operations are reads or writes to shared memory objects. The read operation returns data from memory while the write operation writes data on the memory. *Read-only* transactions consist only of read operations, while *update* transactions have at least one write operation. The transaction is atomic so that after all its operations execute, it either commits allowing the changes to take effect, or it aborts without having any effect [22]. The execution of concurrent transactions must satisfy consistency conditions which affirm the correctness of transactions' execution which will appear as if they execute sequentially (even if they execute concurrently) [12]. For this purpose transactions may abort if they violate the correctness of the execution, and otherwise, they commit. In fact, conflicts between concurrent transactions occur when at least one of them writes to some shared object and another transaction accesses the same shared object for read or write. Software transactional memory (STM) implementations provide flexibility and control over consistency [22], and can be implemented with the use of lock-based or lock-free techniques [15].

In TM the opacity property is used to ensure the correctness of a concurrent execution [12]. Opacity requires *legality* where each read operation in all transactions must read the last written (committed) value on the respective accessed object. A transaction is aborted if its execution affects the validity of read operations in other transactions. However, aborts are generally inefficient as 80% of the execution time might be wasted on them [17]. Thus, in order to cope with the high number of aborts, we relax the strict consistency requirements of opacity. As an alternative we propose to relax the definition of opacity using the notion of K-opacity as a means for implementing an *approximately opaque transactional memory*. K-opacity tolerates some inconsistent read operations, such that it is allowed to read any one of the last K writes, where K is a constant number that is determined according to the data/system sensitivity to stale values. For example, we set K to 1 for sensitive systems in which strict consistency is mandatory (1-opacity is the same with normal opacity), but K can be greater than 1 to tolerate some conflicts.

In real life there are many kinds of systems where precise computations are not required and relaxing strict consistency is acceptable for non-sensitive data and systems [21, 23]. For example, in large distributed systems there are often delays to update all copies of objects which may result in some inconsistent reads and aborts. Also approximated results are acceptable for some database queries such as inventory queries through Online Analytical Processing (OLAP), as it can return approximated results to nested and complicated database queries [8, 23]. Decision Support Systems also work with approximated results such as queries about average income and the percentage of newborns in the country [1]. Moreover, there is no risk for advertising and recommendation systems to have approximated results (suggesting inaccurate restaurant or song would not harm). Furthermore, sensors for temperatures or weather forecasting, usually give approximated reads that satisfy the specification of some systems. Approximation may take place in systems with huge data and frequent changes (such as social network mining systems) [7]. In addition, K-opacity can be used to relax priorities in priority queue applications such as scheduling, bandwidth management, and Dijkstra's algorithm [24]. In such cases, the approximated data would be sufficient and hence allow to relax the precision level. Therefore, we will be able to commit some transactions even if they violate consistency which increases throughput and reduces aborts.

1.1 Contributions

We first apply K-opacity on read-only transactions. A read-only transaction accesses shared objects for read, while update transaction updates the value of any object. Using multi-version TM, it is possible to avoid all aborts of read-only transactions, since a transaction can commit by reading stale stored values. However, using stale values may require a lot of space to store all the saved object versions. By relaxing the opacity precision we can reduce the number of saved versions without aborting any read-only transaction. Particularly, with K-opacity we can reduce the number of saved versions by a factor of K, improving significantly the space requirements of multi-version TM. For read-only transactions we used simple read/write objects. Even though the consistency is relaxed for read-only transactions, for transaction with update operations we require precise consistency (1-opacity).

The promising results of applying the K-opacity concept on read-only transactions encourages us to apply the concept to other kinds of objects as well that involve update operations. Specifically, we consider *count* and *queue* objects. A count object has an initial numerical value and it supports the *add* operation which increments or decrements the object's value, and returns the last object value. A queue object maintains a list of elements and supports *enqueue* and *dequeue* operations, where enqueues occur at the tail of list and dequeues at the head. For these kinds of objects we do not consider the problem of minimizing the number of versions, but rather, we focus on minimizing the total number of aborts by relaxing the consistency requirement.

Maintaining relaxed consistency on update transactions is challenging. Committed update transactions may write some values based on inconsistent reads which would make them produce imprecise results. If new transactions use those imprecise results, then the imprecision could propagate indefinitely. To prevent this scenario, every K update operations a new object version is created that is guaranteed to be precisely consistent. Hence, the read values are always consistent within the last K operations; hence, we achieve K-opacity. To facilitate this, we use *writer lists* that record values of committed write operations which are used to determine the point of time at which the Kth consistent write is to be applied.

Experimental analysis demonstrates the usefulness of our approach. We use synthetic benchmarks where we randomly generate sequences of transactions that access and update the objects. The experiments depict that the throughput increases as the value of K increases.

In summary, our contributions are:

- We introduce the notion of K-opacity as an approximately opaque STM consistency property.
- We propose a multi-version model that applies K-opacity on read-only transactions, while update transactions are precisely opaque. This reduces space requirements by a factor of K.
- We apply K-opacity on STM that uses common objects such as read/write, count and queue objects to reduce the number of total aborts on all kinds of (read-only and update) transactions.
- We also demonstrate the ability of applying K-opacity on single-version and multi-version STM, for some objects including the read/write and count object.

1.2 Outline

The rest of this paper is organized as follows: The system model and definitions are presented in Section 2. Section 3 shows the applying of K-opacity on STM for read-only transactions. In Section 4, we present the design of the algorithm that applies K-opacity on STM using read/write, count and queue objects on all kinds of transactions. The the proof of the correctness is presented in Section 5. Section 6 shows the experimental results and in Section 7, we discuss some related works. Section 8 concludes the paper with some discussion.

2 Notions and Definitions

2.1 Kinds of Objects

A transactional memory system accesses shared memory objects. In this paper, we consider transactions which may perform *read/write operations*, *counting operations* such as addition and subtraction, and *queue operations* such as enqueue and dequeue. We start with some basic definitions of read/write objects, and then we extend these definitions to count and queue objects. The discussion below is focused on the legality of operations which is better described for sequential executions. The notion of legality will be used later in opacity.

2.1.1 Read/Write Objects

Let x be a shared read/write object. The object supports the *read operation* $x.r()$, which returns the stored data in the object, and it also supports the *write operation* $x.w(data)$ which writes the value *data* to x. A sequence of operations on object x is *legal* if every read operation on x returns the most recent written value to x (or the initial value if there is no previous write operation). We relax this definition, and we say that a sequence of operations to object x is *K-approximately legal* (or *K-legal* for brevity) if every read operation returns one of the K most recent written values to x (or also the initial value if the number of previous write operations is less than K). Clearly, for $K = 1$, a 1-legal execution is also a legal execution. In the example below the sequential execution is 1-legal for object x, and 2-legal for object y, since the last $y.r()$ operation returns the value written by the second to the last write operation to y. (Returned values are shown below their respective operations.)

```
x.w(1) y.w(1) x.w(2) y.w(2) x.r() y.r()
                             2     1
```

2.1.2 Count Objects

Let x be a shared count object. The object supports the *add operation* $x.add(val)$, which returns the last value of x and adds *val* to the current value of x. A sequence of operations on object x is *legal* if every add operation returns the last value written to the object by the immediately previous add operation (or the initial value if there is no previous add operation). We relax this definition, and we say that a sequence of operations to object x is *K-legal* if every add operation returns the value written by one of the K most recent add operations applied to the object (or also the initial value if the number of previous add operations is less than K). In the example below assume that count objects x and y are initialized to value 0. The sequential execution is 1-legal for object x, and 2-legal for object y, since the last add operation $y.add(3)$ returns the outcome of the second to the last add operation to y. (Returned values are shown below their respective operations.)

```
x.add(1) y.add(1) x.add(2) y.add(2) x.add(3) y.add(3)
  0        0        1        1        3        1
```

2.1.3 Queue Objects

Let x be a shared queue object. The object supports the *enqueue operation* $x.enq(elem)$ which inserts an element *elem* in the queue, and it also supports the *dequeue operation* $x.deq()$ which removes an element from the queue and returns it. A sequence of operations on object x is *legal* if every dequeue operation returns the oldest inserted element to the queue (if the queue is empty then it returns *nil*). We relax this definition, and we say that a sequence of operations to object x is *K-legal* if every dequeue operation returns one of the K oldest inserted elements to the queue (or also nil if the number of elements in the queue is less than K). In the example below assume that queue objects x and y are initially empty (hold no elements). The sequential execution is 1-legal for object x, and 2-legal for object y, since the last dequeue operation $y.deq()$ returns the second to the oldest inserted element to queue y. (Returned values are shown below their respective operations.)

```
x.enq(a) y.enq(c) x.enq(b) y.enq(d) x.deq() y.deq()
                                      a       d
```

2.2 Approximate Opacity

Each execution thread may execute a sequence of transactions. A transaction may access different kinds of objects and can execute multiple operations on them. All changes made by a transaction

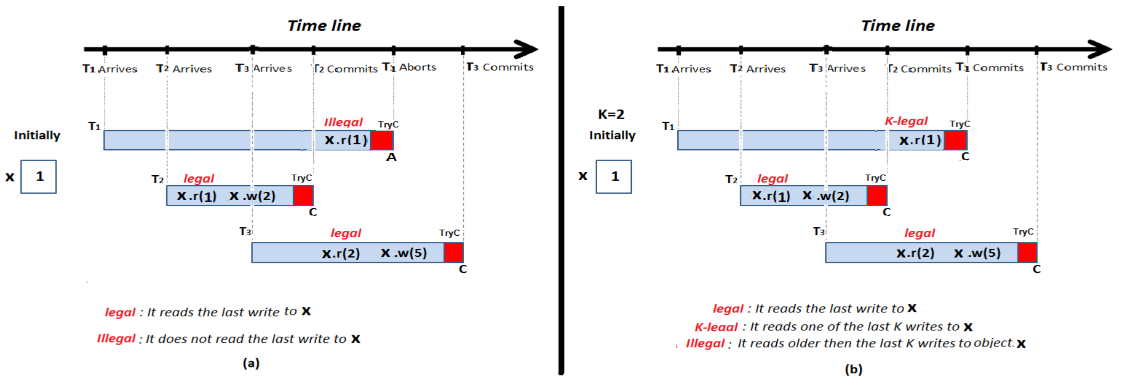

Figure 1: (a) Example of Legal Reads, (b) Example of K-Legal Reads

to shared objects become visible to other transactions only when the transaction commits. If a transaction aborts then all its changes to shared objects are discarded.

Every transaction has a *status* which is initially *live*, and it changes to *committed* or *aborted* at the end of its execution. In our algorithms, every transaction is assigned a unique timestamp i upon its arrival. The timestamp is used as a unique identifier of the transaction [19, 16], so that T_i denotes a transaction that has a timestamp i. Also $i.r_x(data)$ is a read operation belongs to T_i and $i.w_x(data)$ is a write operation that belongs to T_i. Moreover transaction T_i has an access set $T.accSet$ that contains all objects the transaction would access (by applying respective operations) during its execution.

For a transaction, there are two special instantaneous events signifying the begin (TX_begin) and end (TX_end) of a transaction execution. Within a transaction, each operation on a object executes between two instantaneous *events* which are the *invocation* and *response* of the respective object operation (the duration between these two events is the *interval* of the operation).

An *execution history* H is a sequence consisting of all the transactions' events. A history H is *complete* if all transactions in H are either committed or aborted, namely, there are no pending transactions (still in execution) [12, 15].

The partial order $<_H$ is used to express the *real time order* of transactions in history H [12, 15, 16]. For any two transactions T_i and T_j in H, $T_i <_H T_j$ implies that all events of T_i happen before all events of T_j. If the events of T_i and T_j interleave (i.e. there is an overlap between the execution intervals of T_i and T_j), then T_i and T_j are not related according to $<_H$. In a *sequential* history S transactions execute one after the other, namely, $<_S$ is a total order [15, 16]. In addition, we say that any two histories are *equivalent* if they have the same set of events.

Now, in order to define legality we need to be very careful since we have to discard the update operations that belong to aborted transactions. Let S be a sequential history. For any transaction T_i denote by S_i the subsequence of S that includes T_i and all committed transactions that appear in S before T_i. In other words, for any transaction $T_j \in S_i$, either $j = i$ or T_j committed and $T_j <_S T_i$.

In addition, for any transaction T_i there is V_i which is a set of all write, count and queue operations in S_i; in other words, V_i contains all operations that are visible to T_i. Moreover, with respect to any shared object x, we can define $V_i(x)$ as a subset of V_i that considers only the operations on x.

Based on the above, a *legal operation* by T_i on object x is the one that reads the value of the last operation in $V_i(x)$. Furthermore, a K*-legal operation* on the object x is the one that reads the value of one of the K last operations in $V_i(x)$. Based on legality definition, we assume that for a read/write object there is at most one write operation for each transaction which is the same with the last write operation for the object within the transaction. Similarly, for a count object there is at most one representative add operation in a transaction, which aggregates all the individual add operation arguments in the transaction. On the other hand, for a queue object each individual enqueue or dequeue operation is considered in the legality specification of the object.

Thus, transaction T_i is *legal*, if all operations in T_i are *legal*. Then, S is *legal* if all transactions in S are *legal*. Also T_i is K-legal, if all operations in T_i are K-legal [3]. Then, S is K-legal if all transactions in S are K-legal. Consequently, a legal history is a special case of K-legal history by taking $K = 1$.

In Figure 1 we give an example that demonstrates the notion of legal and K-legal executions. In Figure 1(a) transaction T_1, T_2 and T_3 arrive consecutively. Transaction T_2 has a read operation to object x that returns the value 1 which is a legal read because it reads the initial value of x. Then, T_2 writes the value 2 to x and it commits. Transaction T_1 has illegal read as for some reasons, it returns 1 while the last write to x writes the value 2, so it aborts. However, T_3 reads 2 which is the last written value to x and that is legal. It also writes to x and commits. In Figure 1(b) we show how K-legal allows to commit more transactions. Suppose $K = 2$, means it is allowed to read one of the last two writes on x. Now, the three transactions arrive. Transaction T_2 reads the initial value of x which is 1, so it is a legal read. Then, it writes 2 to x and commits. Thus, the last two writes on x are 1 and 2 (we consider the initial value of the object as the first write). Transaction T_1 has a read operation that reads the value 1 which is the second last write to x. As $K = 2$ it is K-legal, and then T_1 commits. Moreover, the read operation in T_3 reads the last write to x which is legal. After that, it writes to x and commits.

Definition 1 (K-approximate opacity) *A history H is K-opaque if it can be converted into complete history H' (by aborting pending transactions) which further has an equivalent sequential history S such that:*

- *H' preserves the real time order of transactions in H, namely, $<_H \subseteq <_{H'}$.*
- *S preserves the real time order of transactions in H', namely, $<_{H'} \subseteq <_S$.*
- *S is K-legal with respect to all of the involved operations.*

Figure 2 shows two example executions (a) and (b) of three transactions sharing two objects x and y. The respective histories H_a and H_b are shown below. It is clear that the histories H_a and H_b of the two executions are complete as there is no live transaction or pending operation. Figure 2(a) shows an opaque execution (H_a) that aborts T_2, and there is an equivalent legal sequential history S, such that $T_2 <_s T_1 <_s T_3$. So, H_a is opaque.

Figure 2(b) illustrates H_b which is not opaque. This is because if T_2 commits then T_1 cannot be ordered before T_2 as T_1 reads $y = 1$, and it cannot be ordered after T_2 as it reads the initial value of x, which means it reads before T_2 writes. However, H_b is K-opaque, as there is an equivalent K-legal sequential history S, such that $T_2 <_s T_1 <_s T_3$. For $K = 2$, T_1 reads one of the last K writes. So, H_b is 2-opaque. [2]

$$H_a = \langle x.r_1(0), x.w_2(1), y.w_2(1), TryC()_2, Abort_2, y.r_1(0), y.r_3(0), TryC()_1, Commit_1, y.w_3(2), TryC()_3, Commit_3\rangle$$

$$H_b = \langle x.r_1(0), x.w_2(1), y.w_2(1), TryC()_2, Commit_2, y.r_1(1), y.r_3(1), TryC()_1, Commit_1, y.w_3(2), TryC()_3, Commit_3\rangle$$

3 Approximately Opaque STM for Read-only Transactions

We consider multi-version TM, where each object stores each new version that is created on updates. Read-only transaction do not perform any updates. In multi-version TM, read-only transactions do not need to abort, while transactions that update objects may abort. Using K-opacity we can reduce the number of saved versions by a factor of K.

Since any new update creates new version and we still save previous ones, read-only transactions avoid aborting by finding the version that preserves the correctness and does not violate consistency. In this work we improve the space complexity of saved versions by reducing the number of saved versions. With reduced number of saved versions, some transactions may not be able to find the suitable version for precise consistency. To cope with this issue, we relax the consistency precision for some read-only transactions such that they are allowed to read some stale values of the memory up to some limit K.

Therefore, we apply K-opacity on read-only transactions while the update transactions are precise. We focus on transactions that perform operations on shared read/write objects.

[2]Each operation in H_a and H_b consists of two events which are *invocation* and *response.*

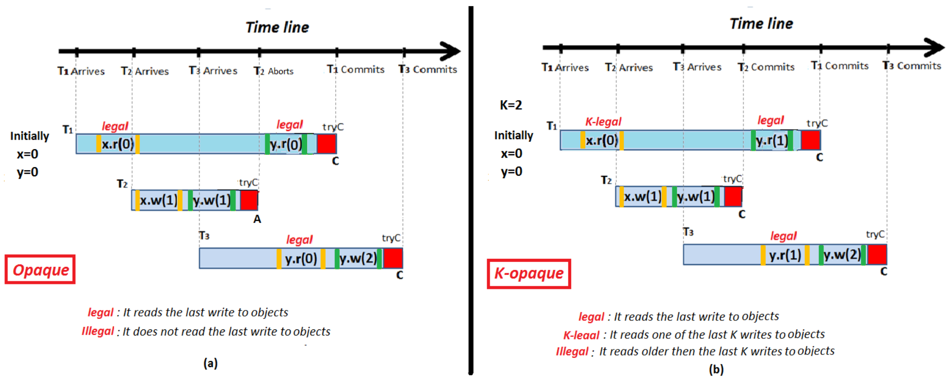

Figure 2: (a) Example of Opaque Execution, (b) Example of K-opaque Execution

3.1 Design of the Algorithm

Our multi-version algorithm (Algorithm 1) is timestamp-based as in previous works that do not consider approximate opacity [16, 9]. As shown in Figure 3, each object x has multiple versions that are stored in a list $x.vl$. We denote a version of object x as $v_i = (ts, data, rl)$, where ts is the timestamp of the transaction that creates (writes) this version, $data$ is the value of x, and rl is a reader-list that includes the timestamps of all transactions that have been reading this version. In our algorithm, we create a new version of x each K commits and we save it in $x.vl$. The last written value is maintained in $x.lastCommit = (ts, data, rl)$ which is an independent version that is overwritten with each commit on x to record the last written value.

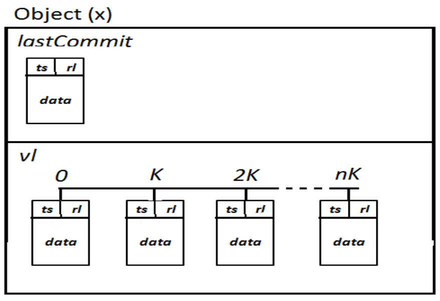

Figure 3: Object data structure.

In many systems, the kinds of transactions (read-only or update) are identified at the beginning such as read balance and bank statements in bank systems, or product quantities in inventory systems. The read operation in read-only transaction T_i tries to read the last written value in $x.lastCommit$ if $x.lastCommit.ts$ is smaller than its own (in Algorithm 2). Otherwise, if $x.lastCommist.ts > i$, then it reads from a suitable saved version in $x.vl$. Also, it adds T_i timestamp to that version's rl. When T_i finishes the execution of all operations, it commits directly.

Figure 4(a) shows an example where read-only transaction can read $x.lastCommit$ without violating the correctness of execution. The execution in Figure 4(a) illustrates the situation where there is no concurrent write on the object. The read-only transaction T_3 finds that the $x.lastCommit.ts$ equals to 2 which is smaller than its own timestamp 3. Therefore, it reads $x.lastCommit$ which shows the last write on the object x. In contrast, Figure 4(b) demonstrates a situation where there is one concurrent write on the object x. When the read-only transaction T_3 executes the read operation $x.r()$ it finds that $x.lastCommit.ts = 4$ (due to the update transaction T_4) which is greater than its own timestamp 3. Thus, it reads a version from $x.vl$. In particular, it reads the latest saved version with the timestamp smaller than its own.

For an update transaction T_i, for any read operation it checks only $x.lastCommit$ and adds i to

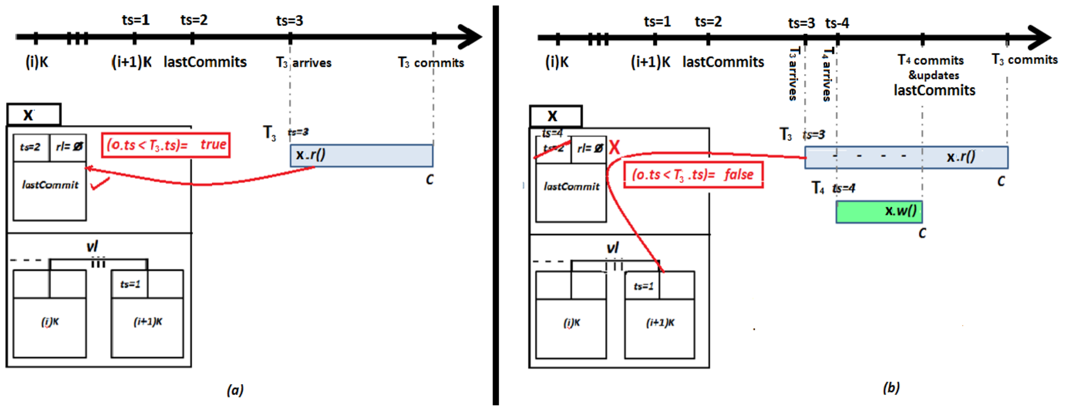

Figure 4: (a) Read-only transaction with no concurrent writes; (b) Read-only transaction with a concurrent write.

$x.lastCommit.rl$ (Algorithm 2). Then, as shown in Algorithm 1, if $x.lastCommit.ts > i$, T_i aborts immediately. Otherwise, it gets $x.lastCommit.ts$ (for validation) and reads the data of x. For the write operations the transaction T_i just writes to its local memory and it maintains its own write set $wSet$ during the execution.

When the update transaction T_i finishes the execution of all operations, it attempts to commit by calling TryC (Algorithm 3). For any object x that was read by T_i, if the $x.lastCommit$ has been overwritten, then T_i aborts. Moreover, T_i aborts if it has a write that invalidates another transaction T_m where $m > i$ (Algorithm 4). In TryC, T_i locks each object x in its $wSet$ and if it commits it overwrites the $x.lastCommit$ version and updates $x.lastCommit.ts = i$. We create a new version in $x.vl$ only every K commits of the object x. We let the new version's ts to be equal to i. After that, we release all locks. In our algorithm read-only transactions never abort, while update transactions may abort.

3.2 Correctness of the Algorithm

In the correctness analysis we prove that our algorithm is opaque for update transactions, and K-opaque for read-only transactions. Let H be an arbitrary execution history, and H' the respective complete history. Consider the sequential execution S which is a serialization of the transactions in H' such that the order of transactions is determined by the timestamps of the transactions, such that if in H' for any two transaction T_i and T_j, $i < j$, then $T_i <_s T_j$.

Lemma 1 *S preserves the real time order of H'.*

Proof According to Algorithm 1, for a transaction T_i the timestamp i is obtained through an atomic operation $i \leftarrow timestamp.\text{getAndInc}()$; If $T_i <_{H'} T_j$ then, it has to be that $i < j$. Since S orders transactions in the timestamp order, then we also have that $T_i <_S T_j$, as needed. □

Lemma 2 *For any object x, the history S is K-legal with respect to read-only transactions accessing x.*

Proof Let T_i be a read-only transaction. Note that in our algorithm read-only transactions do not abort, and hence T_i does not abort. Suppose T_i executes operation $x.r()$. According to function GetLatestVersion(), we have that T_i observes either $x.lastCommit.ts < i$ or $x.lastCommit.ts > i$. We examine these two cases separately.

i. $x.lastCommit.ts < i$:
then GetLatestVersion() returns $x.lastCommit$ and data $y = x.lastCommit.data$, which is the latest version of the object at that moment when x is accessed by T_i. Let T_j be the transaction

Algorithm 1: K-Opaque Multi-Version

```
/* global variable initialization */
$timestamp \leftarrow 0$;
foreach transaction $T_i$ do
    /* $i$ gets a unique timestamp */
    $i \leftarrow timestamp$.getAndInc();
    $T_i.status \leftarrow live$;
    $T_i.wSet \leftarrow \emptyset$;
    while there is an unexecuted operation $x$ do
        /* if operation is read */
        if $x = x.r()$ then
            $v \leftarrow$ GetLatestVersion$(i, x)$;
            if  $v.ts > i$ then
                /* this check only for read operation in update transaction to have immediate abort */
                $T_i.status \leftarrow aborted$;
                return;
            $y \leftarrow v.data$;
            $x_{local}.ts \leftarrow v.ts$;
        else
            /* $x = x.w(data)$; write local copy */
            $x_{local}.data \leftarrow data$;
            $T_i.wSet \leftarrow x \cup T_i.wSet$;
    if TryC$(i)$ then
        $T_i.status \leftarrow committed$;
    else
        $T_i.status \leftarrow aborted$;
    return;
```

Algorithm 2: GetLatestVersion(i, x)

```
$last \leftarrow null$;
Lock $x$;
if $T_i.kind = readonly$ then
    if $x.lastCommit.ts < i$ then
        $last \leftarrow x.lastCommit$;
        Add $i$ to list $x.lastCommit.rl$;
    else
        $v \leftarrow$ the most recent version in $x.vl$ with timestamp smaller than $i$;
        $last \leftarrow v$;
        Add $i$ to list $v.rl$;
else
    /* update transaction */
    $last \leftarrow x.lastCommit$;
    Add $i$ to list $x.lastCommit.rl$;
Unlock $x$;
return $last$;
```

Algorithm 3: TryC(i, x)

```
/* check if $T_i$ is readonly */
if $T_i.kind = readonly$ then
    return $true$;
/* $T_i$ has to be update transaction */
$L \leftarrow \emptyset$;
/* assume a predetermined order for the objects */ forall the $x \in i.wSet$ do
    Lock $x$;
    $L \leftarrow L \cup x$;
    if Validate$(i, x) = false$ then
        unlock all locked objects in $L$;
        return $false$;
forall the $x$ in $i.wSet$ do
    $x.versionCounter$.getAndInc();
    if $x.versionCounter \mod K = 0$ then
        /* add new version to $x.vl$ */
        Add $(i, x_{local}.data, nil)$ to $x.vl$;
    /* overwrite $x.lastCommit$ */
    $x.lastCommit \leftarrow (i, x_{local}.data, nil)$;
Unlock all objects in $L$;
return $true$;
```

Algorithm 4: Validate(i, x)

```
/* Check if $lastCommit$ has been overwritten */
if $x.lastCommit.ts > x_{local}.ts$ then
    return $false$;
/* Check if some other transaction $T_m$ has read the same version read by $T_i$, where $m > x_{local.ts}$ */
if $x.lastCommit.rl$ contains a transaction $T_m$, where $m > i$ then
    return $false$;
return $true$;
```

that committed the value, that is, $j = x.lastCommit.ts < i$. Since S preserves the timestamp order, T_j appears before T_i in S. Suppose that there is another transaction T_k, with $j < k < i$, that commits a value to object x. If T_k commits after $x.r()$ locks x (in GetLatestVersion()), then according to function Validate(), T_k has to abort because T_i is in the reader-list of x when T_k attempts to commit. On the other hand, if T_k commits before $x.r()$ locks x, then T_i must have read the value committed by T_k, or in other words $T_k = T_j$.

ii. $x.lastCommit.ts > i$:
then GetLatestVersion() returns a version v, and data $y = v.data$, where v belongs to version list $x.vl$ and it is the latest version of x with timestamp $v.ts = j < i$. Let T_j be the transaction that created version v. We need to prove that there cannot be more than $K - 1$ other committed transactions for object x between the time that T_j commits and T_i starts in H'. We observe that any transaction T_k that commits a value for x after T_j must have timestamp $k > j$, since otherwise the interval of T_k would contain the interval of T_j in H', and according to function Validate() T_k would abort. We have that in S, T_j appears before T_i, since $j < i$. Let X be the set of transactions which appear in S between T_j and T_i and commit a value for x (for any $T_k \in X$ it holds $j < k < i$). We want to show that $|X| \leq K - 1$. Similar to the reasons explained above in case i, X cannot contain any transaction which commits after $x.r()$ locks x. Moreover, X cannot contain any transaction T_k that commits before T_j, since in H' interval T_j would contain interval T_k and according to function Validate() T_j would abort. Hence, all the transactions in X must have timestamp greater than j and must commit in H' after T_j. If $|X| \geq K$, according to our algorithm, a newer version v' of x must have been saved (in $x.vl$) after T_j commits and before T_i starts, by some transaction $T_\zeta \in X$. However, this is impossible, since T_i would have read v' and not v.

Therefore, we have that in case i execution S is 1-legal, while in case ii the execution S is K-legal. □

Lemma 3 *For any object x, the history S is 1-legal with respect to update transactions accessing x.*

Proof Now consider the update transactions that access object x. Let T_i be an update transaction that invokes operation $x.r()$. According to the algorithm, if $x.lastCommit.ts > i$, then T_i is aborted and operation $x.r()$ never completed. On the other hand, if $x.lastCommit.ts < i$ then $x.r()$ completes with $y = x.lastCommit.data$. Let $j = x.lastCommit.ts$, namely, T_i reads the value written by T_j, with $j < i$. Similar to the proof of Lemma 2, any transaction T_k that commits a value for x after T_j must have timestamp $k > j$.

In S transaction T_j appears before T_i. We only need to show that in S there is no other committed transaction between T_j and T_i for object x. Suppose that there is a transaction T_k, with $j < k < i$, which appears between T_j and T_i in S and commits a value to x. If T_k commits before T_j in H', function Validate() would cause to abort T_j. If T_k commits before $x.r()$ locks object x, then T_i must have used the value committed by T_k. On the other hand, if T_k commits after $x.r()$ locks object x, then according to function Validate() T_k has to abort, since T_i is in the reader-list of x when T_k attempts to commit, and $i > k$. □

Since H' respects the real time order of H, considering all objects used in H, from Lemmas 1, 2 and 3, we obtain the following theorem.

Theorem 1 *Any execution history H of our algorithm is K-opaque with respect to read-only transactions and 1-opaque with respect to update transactions.*

It is easy to check that the proposed algorithm does not deadlock, since function GetLatestVersion() accesses one object at a time, and function TryC() accesses objects in a predetermined order, avoiding racing situations.

Lemma 4 *Our algorithm does not deadlock.*

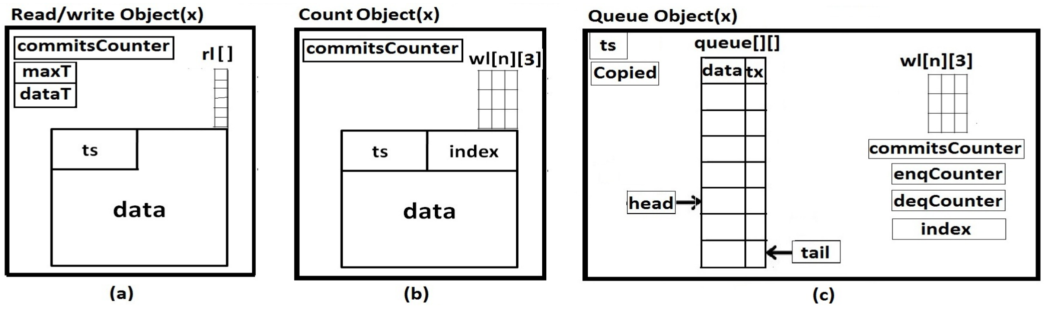

Figure 5: (a) Read/Write object data structure; (b) Count object data structure; (c) Queue object data structure.

Assume that the transactions in our algorithm access the set of objects $X = (x_1, x_2, ..., x_n)$. Let V be the set of all committed versions (updating $x_i.lastCommit$) and V' the set of all saved versions (saved in $x_i.vl$).

Theorem 2 *In any execution of our algorithm, the total number of saved object versions is* $|V'| = \Theta(|V|/K + |O|)$.

Proof Throughout the execution, for any object x_i let the number of committed versions be v_{x_i} (updating $x_i.lastCommit$). The total number of committed versions for all objects is $|V| = \sum_{i=1}^{n} v_{o_i}$. Our algorithm saves a new version for object x_i each K object commits. Thus, the total number of saved versions for object x_i (saved in $x_i.vl$) is v_{x_i}/K, and consequently, the total number of saved versions $|V'|$ for all objects will be $|V|/K$. In addition, each object has a $lastCommit$ version which adds a number of $|O|$ versions to $|V|/K$. □

Now, if we exclude the last committed versions, the total version space of regular (not ours) multi-version TM is $\Theta(|V|)$, since every version is saved at some point of time. On the other hand, from Theorem 2, with our approximately opaque multi-version algorithm we only create a new version each K commits reducing this number to $\Theta(|V|/K)$, a reduction by a factor of K.

4 Approximately Opaque STM for Read/Write, Count and Queue Objects

The promising results of applying K-opacity on read-only transactions (see section 6.1), encourages us to apply the concept of K-opacity on update transactions as well. The update transactions access regular read/write, count and queue objects, which are common objects used in typical concurrent programs. The goal is to reduce the overall number of aborts by taking advantage of the relaxed consistency offered by K-opacity.

4.1 Design of the Algorithm

Let us start with the structure of the STM objects where there are three kinds of objects which are read/write, count and queue objects. For a read/write object x, Figure 5(a) depicts the basic data structure used in our algorithm. This structure has fields $(ts, data, commitsCounter, maxT, dataT, rl[])$ such that: ts shows the maximum timestamp of the transactions that overwrite x, $data$ is the value of x, $commitsCounter$ records the number of commits on x to ensure that we overwrite x every K commits, $maxT$ and $dataT$ record the maximum timestamp of the transactions that commit on x and the value it writes, respectively, and rl is a list that records the timestamps of the readers of the object.

Figure 5(b) illustrates the counter object x which consists of $ts, data, commitsCounter, wl$ and $index$, where ts represents the maximum timestamp of the transactions that overwrite x, $data$ is the

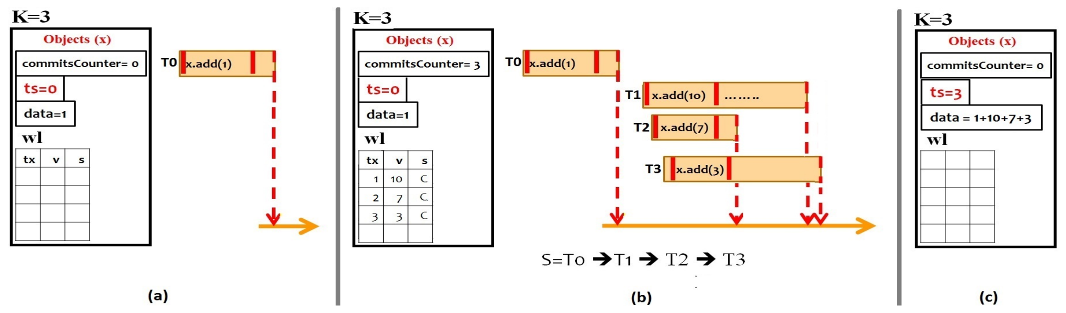

Figure 6: Concurrent transactions execute add operation on the count object x.

value of x, $commitsCounter$ records the number of commits on x and $wl[][]$ is a writer list which is an array of size n, where n is the number of threads in the system. Actually, $wl[][]$ has three columns such that each update transaction that writes to x must post its timestamp (tx) in the first column, the $data$ it writes in the second column, and the transaction status ($live, committed$ or $aborted$) in the third one. Since $wl[][]$ is an integer array, we represent the transaction status with 0, 1 and 2 corresponding respectively to $live, committed$ and $aborted$. In addition, we use $index$ to tell the update transactions in which position to write in $wl[][]$.

Figure 5(c) illustrates the structure of the queue object x that has a field ts which is the maximum timestamp of the transactions that update x. $queue[][]$ is an array of two columns. The first one is for $data$ and the other is for the timestamp of the transaction that enqueues or dequeues that $data$. Also x has $head$ and $tail$. The $commitsCounter$ records the number of commits on x. To show the number of concurrent transactions that perform enqueues and dequeues we use $enqCounter$ and $deqCounter$. Moreover, every queue has $wl[][]$ (writer list) which is an array of size n, where n is the number of threads in the system. In fact $wl[][]$ has three columns such that each update transaction that writes to x must post its timestamp in the first column, the second column illustrates the kinds of the operation which is either enqueue or dequeue; the status of the transaction (which is either $live, committed$ or $aborted$) appears in the third column.

4.2 Example

For simplicity we start the explanation of our algorithm with an example of concurrent transactions that execute *add operations* on a counter object x where $K = 3$. Figure 6(a) shows that the transaction T_0 has committed and updated x that $x.commitsCounter = 0$, $x.ts = 0$ and $x.data = 1$.

Three concurrent transactions, T_1, T_2, and T_3 execute and add their own value but each returns the same value 1, the previous value inserted by last committed transaction T_0, since $K = 3$. Figure 6(b) shows that the transaction T_1 executes the operation $x.add(10)$. It records its information in $x.wl[][]$ such that the timestamp is in the first column, the new value 10 in the second, and status (live) in the third. Then, T_1 reads $x.data = 1$ (the returned value) and at commit increases $x.commitsCounter$ by 1 and changes its status in the writer list to C (committed). Also, T_2 executes the operation $x.add(7)$. So it records its information in $x.wl[][]$ and it reads $x.data = 1$ and adds 7. At commit, T_2 increases $x.commitsCounter$ by 1 and changes its status in the writer list to C (committed). Further, T_3 executes $x.add(3)$ and records its information in $x.wl[][]$. T_3 also reads $x.data = 1$. At commit, T_3 increases $x.commitsCounter$ by 1 so it becomes 3 which is equal to K. T_3 changes its status in the writer list to C (committed), and updates x with the new value aggregating the previous transactions, $x.data = 1 + 10 + 7 + 3 = 21$. The $x.commitsCounter$ is also reset to 0, and the latest committed timestamp is set to $x.ts = 3$, as shown in Figure 6(c).

4.3 The Detailed Design of the Algorithm

To start with our algorithm, the classes RWObject, Count and Queue, show the structure of the objects as described earlier. Basically our algorithm is timestamp-based that executes transactions concurrently but does not update objects with each commit. Indeed, every K commits on the object we overwrite the object with a new value.

The Main (Algorithm 5) shows the structure of the transaction where it has a unique timestamp i using getAndIncerement() method that increments the value of a special object *timestamp* by 1. A transaction T_i has a status $T_i.status$ that is set to a *live*. It also has an access set $T_i.accSet$ that is maintained during the execution. We also have two global variables which are *correct* (which is used to calculate the correct value of update transactions) and *commits* (which counts the number of commits on x). We assume that the timestamp of T_i is i (i.e. its subscript).

Then, transaction T_i executes all its operations on the different objects kinds it accesses which are read, write, add, enqueue or dequeue operations. For all operations we add the object x and $x.ts$ at that moment to $T_i.accSet$ and execute by calling functions ReadWrite(), Add(), Enqueue() or Dequeue(). If the function returns $false$, then T_i aborts immediately. For queue operations, before we execute the operation the enqueue and dequeue require to count the number of concurrent enqueues and dequeues using $x.enqCounter$ and $x.deqCounter$. When T_i executes all operations it calls TryC() (Algorithm 11) and based on that it commits or aborts. [3]

4.3.1 Read/Write Operations

For read/write operations, in ReadWrite() (Algorithm 6), if the transaction's T_i timestamp i is smaller than $x.ts$, then T_i aborts since it violates the correctness property (timestamp-based execution). Otherwise, if $data = Null$ means that the operation is a read, and then we add i to $x.rl[]$ and get $x.data$. At the end of T_i execution, it calls TryC() where the read operations get validated.

On the other hand, if $data \neq Null$ it means that the operation is write, and T_i writes in its own local memory. At the end of T_i's execution, it calls TryC() and it locks x (we lock objects in a predetermined order to avoid deadlock) to check if there is another transaction with a timestamp greater than i that commits on x. Also, it checks if another transaction with greater timestamp has read x. In these two cases T_i aborts. Otherwise, in Commit() (Algorithm 13), we store the maximum timestamp of the transactions that commit on x in $x.maxT$, and the data of that transaction in $x.dataT$. We increase $x.commitsCounter$ by one, and if it is equal to K, then we overwrite x with the timestamp and the data that are stored in $x.maxT$ and $x.dataT$. We then reset $x.commitsCounter, x.maxT$ and $x.dataT$. We release all locks and we change $T_i.status$ to *committed*. However, if $x.commitsCounter < K$, T_i just commits.

4.3.2 Counting Operations

For counting operations, in Add() (Algorithm 7), if $x.ts > i$, then T_i aborts. Otherwise, to simply read the counter (by applying $x.add(0)$), we get the data and write locally.

To add a nonzero value to x, if T_i accesses x for the first time, then T_i adds itself to $x.wl[][]$, such that it increments $x.index$ which tells in which row T_i can write, and then it posts $i, data$ and $status$ to $x.wl[][]$. T_i also writes in its local memory. If T_i already accessed x, then it would already be in $x.wl[][]$. In this case, it just adds the data of the operation to the data that is already stored in $x.wl[][]$.

Then, in TryC() the transaction locks x and calls CheckStatus() (Algorithm 10) to check if T_i is aborted by another transaction or whether x has been overwritten by a transaction T_m where $m > i$ tin which case it returns $false$ and T_i aborts. In Abort() (Algorithm 12), for all objects in $T_i.accSet$, we change $T_i.status$ to *aborted*.

Otherwise, T_i calls Commit() and checks ($x.commitsCounter$) if the number of committed transactions including T_i equals to K, then T_i overwrites x with the correct value that considers the values of all committed transactions in $x.wl[][]$. Therefore, T_i aborts all live transactions in $x.wl[][]$, copies the data of committed transactions from $x.wl[][]$ to temporary array $temp[][]$ and sorts $temp[][]$ based on the transactions' timestamps. After that, it calculates the correct value of $x.data$ and finds the maximum timestamp in $temp[][]$ (max). It overwrites x with $x.ts = max$, $x : data = correct$ and

[3]The operations within a transaction are sequential

resets all other fields of x. We release all locks and change $T_i.status$ to committed. However, if the number of committed transactions is less than K, T_i just commits and changes its status in $x.wl[][]$.

4.3.3 Queue Operations

In Enqueue() (Algorithm 8), a transaction T_i first checks if there is no version of object x in any transaction's local memory, and then T_i accesses the (global) shared object x and T_i aborts if $x.ts > i$. Otherwise, it gets $x.tail$ and uses *enqs* which shows how many concurrent enqueues there are, and that is used to map T_i to the right spot in the $x.queue$. If the queue is full, it returns *nil* (special specification of the queue object), or T_i enqueues the new element. T_i can be mapped in any position between $tail + 1$ and $tail + K$. Also, we write T_i timestamp next to the element in $x.queue$. T_i posts $i, status$ and *op* (enqueue or dequeue) in $wl[][]$, and returns *true*. Now we check if $x.commitsCounter = K - 1$ which means that this operation is the Kth operation on x, then, we change $x.copied$ flag to 1 and we copy x to T_i's local memory.

Second, if x is copied by T_i, we find x in T_i's local memory (which means T_i already accessed x and executed the Kth operation on x) and then T_i accesses the local copy. After that it will do the same procedure that would be done on the global x but on the local copy instead. If T_i executes another Kth operation on x, it just overwrites the local copy and does not touch global x. Third, if x is copied by another transaction then T_i aborts.

In Dequeue() (Algorithm 9) if there is no version of object x in T_i's local memory, then T_i accesses the (global) shared object x and we abort T_i if $x.ts > i$. Otherwise, we get $x.head$ and $x.tail$. If the queue is empty, we return *nil* (special specification of the queue object), or we use *deqs* (which shows the number of concurrent dequeues) to map T_i to a specific position in the queue and copy the element. T_i can be mapped to dequeue any element between *head*, and $head+K$. Also T_i writes its information in $wl[][]$ and returns *true*. Indeed, a dequeue operation does not remove the element from the queue since T_i may abort later. Now we check if $x.commitsCounter = K - 1$ which means that this operation is the Kth operation on x; the rest follows the same procedure as in Enqueue().

In TryC() we call CheckStatus() to check whether T_i is aborted and removed from $wl[][]$ by another transaction, in which case we return $false$ and call Abort(). Otherwise, we check if the number of committed transactions ($x.commitsCounter$) equals to K, Then, we abort all the other *live* transactions in $x.wl[][]$ and consider two possible scenarios:

i. If a copy of x is in T_i local memory, we just copy the local copy of x to global.

ii. If there is no copy of x in T_i local memory, then T_i overwrites shared object x.

We adjust the field $x.queue$ by deleting the elements of committed dequeues and removing the empty spots that result from aborted enqueues. In addition, we update $x.ts$ and reset $x.enqCounter, x.deqCounter$ and $x.wl[][]$. However, if the number of committed transactions does not equal to K, then we do nothing. We release all locks and we change $T_i.status$ to *committed*. In Abort(), we change $T_i.status$ to *aborted*, and we change T_i status to aborted in the writer lists of all objects in $T_i.accSet$. Also, for any x that has a copy in T_i local memory, we reset $x.copied$ to 0.

```
Class RWObject
int $ts \leftarrow 0$;
int $data \leftarrow 0$;
int $commitsCounter \leftarrow 0$;
int $maxT \leftarrow -1$;
int $dataT$;
int $rl[]$;
```

5 Correctness of the Algorithm

In the correctness analysis we prove that our algorithm is K-opaque for all transactions. Let H be an arbitrary history of an execution. Let H' be a complete history that we obtain such that if a

```
Class Count
int $ts \leftarrow 0$;
int $data \leftarrow 0$;
int $commitsCounter \leftarrow 0$;
int $index \leftarrow 0$;//Array's index
//$wl[][]$ is an array to record update transactions
//$n$ is the number of the thread in the system
int $wl[n][3]$;
```

```
Class Queue
int $ts \leftarrow 0$;
int $head \leftarrow 0$;
int $tail \leftarrow 0$;
int $queue[size][]$;
int $commitsCounter \leftarrow 0$;
int $x.copied = 0$;
//$wl[][]$ is an array to record update transactions
//$n$ is the number of the thread in the system
int $wl[n][3]$;
int $index \leftarrow 0$;//$wl[][]$ array's index
$enqCounter \leftarrow -1$;
$deqCounter \leftarrow -1$;
```

pending transaction in H didn't invoke either Commit() or Abort() then its status is aborted, while in any other case the status is either committed or aborted, according to which of the two functions the transaction invoked. Let S be the sequential execution which is a timestamp-based serialization of the transactions in H'. (Due to lack of space the proofs appear in the Appendix.)

Lemma 5 *S preserves the real time order of the transactions in H'.*

Proof According to Main() (Algorithm 5), each transaction T_i obtains a unique timestamp using $i \leftarrow timestamp$.getAndInc() which is an atomic operation. If $T_i <_{H'} T_j$, then $i < j$. Since S orders transactions based on their timestamp, we get $T_i <_S T_j$. In other words, $<_{H'} \subseteq <_S$, as needed. □

We continue to prove that S is K-legal with respect to any object x for any transaction.

Lemma 6 *The history S is K-legal, for any read/write object x.*

Proof Let T_i be a transaction that executes read operation $x.r_i(data)$. According to function ReadWrite(), T_i checks $x.ts$ that shows the timestamp of the last transaction that overwrites x. If $x.ts = j > i$, then T_i aborts. If T_j is the last transaction that overwrites x, then we need to prove that there cannot be more than $K-1$ other committed transactions on x between the time that T_j commits and T_i performs its read in S. Since $j < i$ we have that in S, $T_j <_S T_i$. Let Q be the set of transactions that appear in S between T_j and T_i that have a write operation to x and commit (Q does not contain T_j or T_i). We only need to show that $|Q| \leq K-1$.

We first show that none of the transactions in Q overwrite x. Suppose for the sake of contradiction that there is a transaction $T_m \in Q$ which overwrites x, namely, it sets $x.commitsCounter = 0$ and updates $x.data = x_{local}.data$ and $x.ts = m$. Note that $j < m < i$. We examine three cases with respect to when T_m commits in H':

- T_m commits before T_j commits.
  In this case, when T_j invokes TryC() it observes one of the following two scenarios:
  - $maxT = m > j$: T_j observes that a transaction T_m with higher timestamp ($m > j$) has committed on x (but it does not overwrite x), since $maxT$ records the maximum timestamp of the committed transactions, and hence T_j aborts.

Algorithm 5: Main()

```
$timestamp \leftarrow 0$;
int $correct \leftarrow 0$;//To calculate the correct value
int $commits \leftarrow 0$;
bool $valid = true$;
foreach (transaction $T$) do
    //Get a unique timestamp
    $i \leftarrow Timestamp$.getAndInc();
    $T_i.status \leftarrow live$;
    $T_i.accSet \leftarrow \emptyset$;
    while (there is unexecuted operation on object $x$) do
        switch operation on $x$ do
            case ($x.r()$)
                //Read operation
                $T_i.accSet \leftarrow x \cup T_i.accSet$;
                $valid =$ ReadWrite$(i, x, data)$;
            case ($x.w(data)$)
                //Write operation
                $T_i.accSet \leftarrow x \cup T_i.accSet$;
                $valid =$ ReadWrite$(i, x, data)$;
            case ($x.add(data)$)
                //Add operation
                $T_i.accSet \leftarrow x \cup T_i.accSet$;
                $valid =$ Add$(i, x, data)$;
            case ($x.enqueue(data)$)
                //Enqueue operation
                $enqs \leftarrow x.enqCounter$.getAndInc();//How many concurrent enqueues
                $T_i.accSet \leftarrow x \cup T_i.accSet$;
                $valid =$ enqueue$(i, x, data, enqs)$;
            case ($x.dequeue(data)$)
                //Dequeue operation
                $deqs \leftarrow x.deqCounter$.getAndInc();//How many concurrent dequeues
                $T_i.accSet \leftarrow x \cup T_i.accSet$;
                $valid =$ dequeue$(i, x, deqs)$;
        if ($valid = false$) then
            Abort$(i, T_i.accSet)$;
            return;
    if (TryC$(i, T_i.accSet)$) then
        Commit$(i, T_i.accSet)$;
    else
        Abort$(i, T_i.accSet)$;
    return;
```

Algorithm 6: ReadWrite$(i, x, data)$

```
//If a newer transaction overwrites $x$, then $T_i$ aborts
if ($i < x.ts$) then
    return $false$;
if ($data = Null$) then
    //Read operation
    add $i$ to $x.rl[]$;
    get$(x.data)$;
else
    //Write operation
    //Write the value $data$ to $x$ in local memory
    let $x_{local}.data \leftarrow data$;
return $true$;
```

Algorithm 7: Add$(i, x, data)$

```
int $max \leftarrow -1$;
//If a newer transaction overwrites $x$, then $T_i$ aborts
if ($x.ts > i$) then
    return $false$;
//If the operation is reading the counter
if ($data = 0$) then
    //Get the value of $x$
    int $y = x.data$;
    let $x_{local}.data \leftarrow y$;
else
    //If the operation is writing to the counter
    //If the same transactions execute more than one counting operation on the same object $x$, it maintains
    its data and then behaves like arriving for first time
    for ( from $m \leftarrow 0$ to $m = n$) do
        if ($wl[m].tx = i$) then
            $x.wl[].data = x.wl[].data + data$;
    //If $T_i$ accesses $x$ for first time
    $r \leftarrow x.index$.getAndInc();
    //$wl$ is an array of size $n$
    $x.wl[r].tx \leftarrow i$;
    $x.wl[r].data \leftarrow data$;
    $x.wl[r].status \leftarrow T_i.status$;
    //Write the value $data$ to $x$ in local memory
    int $y = x.data$;
    let $x_{local}.data \leftarrow y$;
return $true$;
```

  - $x.ts \geq j$: T_j observes that x was actually overwritten by T_m or by a more recent transaction, and hence, T_j aborts.

  In either scenario, T_j aborts, which is impossible.

- T_m commits and overwrites x before $x.r_i()$ starts.
  In this case, T_i reads either the value written by T_m or by a more recent transaction (with timestamp $x.ts \geq m$). However, this contradicts the assumption that T_i reads $x.ts = j$.

- T_m commits and overwrites x after $x.r_i()$ ends.
  In this case, in its TryC() transaction T_m will observe that $x.commitsCounter \geq K-1$ (which means T_m is the K^{th} transaction), and also it observes that T_i is in the reader list of x (that is, $i \in x.rl$ with $i > m$), and the combination of these two observations together force T_m to abort, which is a contradiction.

Therefore, no transaction in Q overwrites x. This implies that each transaction in Q increments $x.commitsCounter$. For a transaction $T_k \in Q$, let c_k denote the respective updated value of $x.commitsCounter$. Next we show that for any pair $T_k, T_l \in Q$, where $k < l$, it must hold that $c_k \neq c_l$. Since $x.commitsCounter$ is updated atomically (is locked by each transaction that modifies it), if $c_k = c_l$ then some transaction T_m must commit and overwrite x (reset $x.commitsCounter$) after T_k commits and before T_l commits in H'. We know that transaction T_m cannot be in Q. Therefore, $m < j$, which is impossible since T_m would abort observing a higher timestamp on x than its own ($x.ts \geq j$). Therefore, the transactions in Q assign unique values to $x.commitsCounter$. Since the $x.commitsCounter$ cannot exceed $K-1$ and none of the transactions in Q set $x.commitsCounter = 0$, we get that $|Q| \leq K-1$, as needed.

As a special case, the same properties for Q hold even if T_i reads the initial value of x and T_j is replaced by a special instantaneous event that initializes x. □

Lemma 7 *The history S is K-legal, for any count object x.*

Algorithm 8: Enqueue($i, x, data, enqs$)

```
//If $x$ is not in any transaction local memory
if ($x.copied = 0$) then
    //If a newer transaction overwrites $x$, or there are more than $K$ dequeues, then $T_i$ aborts
    if ($(x.ts > i) \vee (enqs >= K)$) then
        return $false$;
    $t \leftarrow x.tail$;
    if ($t = size$) then
        //The queue is $full$
        return $nil$;
    //Mapping
    $position \leftarrow enqs + t + 1$;
    //Insert element
    $x.queue[position].data \leftarrow data$;
    $x.queue[position].tx \leftarrow i$;
    $x.wl[index].tx \leftarrow i$;
    $x.wl[index].op \leftarrow enqueue$;
    $x.wl[index].status \leftarrow live$;
    //Now we check if it is the $K^{th}$ operation
    if ($x.commitsCounter = K - 1$) then
        $x.copied = i$;
        //We prepare a local version of $x$
        copy $x$ to $T_i$ local memory;
        adjust $x_{local}.queue[][]$;
        $max$ = max($x_{local}.queue[].tx$);//Maximum timestamp in $x_{local}.queue[][]$
        update $x_{local}.head$; update $x_{local}.tail$; $x_{local}.ts \leftarrow max$;
        reset $x_{local}.enqCounter$; $x._{local}deqCounter$; $x_{local}.wl[][]$;
        $x_{local}.commitsCounter = 0$;
else
    //If $x$ is in another transaction local memory
    if ($x.copied \neq i$) then
        return $false$;
    do the same thing (in $if()$ part) on the local copy but we do not copy $x$ to local memory again;
return $true$;
```

Algorithm 9: Dequeue($i, x, deqs$)

```
//If $x$ is not in any transaction local memory
if ($x.copied = 0$) then
    //If a newer transaction overwrites $x$, or there are more than $K$ dequeues, then $T_i$ aborts
    if ($(x.ts > i) \vee (deqs >= K)$) then
        return $false$;
    $h \leftarrow x.head$;
    $t \leftarrow x.tail$;
    //Mapping
    if ($(t = 0) \vee ((t - h) + 1) < deqs)$) then
        //Empty queue
        return $nil$;
    $position \leftarrow deqs + (h + 1)$;
    //Get element
    $data \leftarrow x.queue[position].data$;
    $x.queue[position].tx \leftarrow i$;
    $x.wl[index].tx \leftarrow i$;
    $x.wl[index].op \leftarrow dequeue$;
    $x.wl[index].status \leftarrow live$;
    //Now we check if it is the $K^{th}$ operation
    if ($x.commitsCounter = K - 1$) then
        $x.copied = 1$;
        //We prepare a local version of $x$
        copy $x$ to $T_i$ local memory;
        adjust $x_{local}.queue[][]$;
        $max$ = max($x_{local}.queue[].tx$);//Maximum timestamp in $xlocal.queue[][]$
        update $x_{local}.head$; update $x_{local}.tail$; $x_{local}.ts \leftarrow max$;
        reset $x_{local}.enqCounter$; $x._{local}deqCounter$; $x_{local}.wl[][]$;
        $x_{local}.commitsCounter = 0$;
else
    //If $x$ is in another transaction local memory
    if ($x.copied \neq i$) then
        return $false$;
    do the same thing (in $if()$ part) on the local copy but we do not copy $x$ to local memory again;
return $true$;
```

Algorithm 10: CheckStatus(i, x)

```
int $T_iRemoved \leftarrow 0$;
//Check if $T_i$ is aborted and removed from $wl$ by another transaction
for ( from $m \leftarrow 0$ to $m = n$) do
    if ($x.wl[m].tx = i$) then
        $T_iRemoved \leftarrow 1$;//It is not removed
        break;
//If $T_i$ is removed from $wl$
if ($T_iRemoved = 0$) then
    $commits = -1$;
    return $commits$;
//$T_i$ still in $wl$
$x.wl[m].status = committed$;
//Check how many committed transactions
$commits = x.commitsCounter$;
return $commits$;
```

Algorithm 11: TryC($i, T_i.accSet$)

```
$L \leftarrow \emptyset$;
forall the $(x$ in $T_i.accSet)$ do
    lock();
    $L \leftarrow L \cup x$;
    switch $Type$ of $x$ do
        case $x$ is $RWObject$
            if $(data = Null)$ then
                //Read operations do not validate
            else
                //For write operation
                if $((x.ts > i) \vee (maxT > i))$ then
                    //Aborting because a concurrent write
                    Unlock() all objects in $L$;
                    return $false$;
                if $((x.rl[]$ has a transaction $T_m$ where $m > i) \wedge (x.commitsCounter = K - 1))$ then
                    //Aborting because a concurrent read
                    Unlock() all objects in $L$;
                    return $false$;
        case $(x$ is $Counter)$
            $commits \leftarrow$CheckStatus$(i, x)$;
            if $((commits = -1) \vee (x.wl[][]$ has a transaction $T_m$ where $m > i))$ then
                Unlock() all objects in $L$;
                return $false$;
        case $(x$ is $Queue)$
            $commits \leftarrow$CheckStatus$(i, x)$;
            if $((commits = -1) \vee (x.wl[][]$ has a transaction $T_m$ where $m > i))$ then
                Unlock() all objects in $L$;
                return $false$;
return $true$;
```

Algorithm 12: Abort($i, T_i.accSet$)

```
$T_i.status \leftarrow aborted$;
//Change its status in all counters and queues it accesses
forall the $(x$ in $T_i.accSet)$ do
    if $(x.copied = i)$ then
        $x.copied = 0$;
    $x.wl[x.index].status \leftarrow aborted$;
    remove $i$ from $x.rl[]$;
return;
```

Algorithm 13: Commit($i, T_i.accSet$)

```
int $max \leftarrow -1$;
$T_i.status \leftarrow committed$;
forall the $(x\ in\ T_i.accSet)$ do
    switch $Type\ of\ x$ do
        case $x\ is\ RWObject$
            if $(x.maxT < i)$ then
                //Recording the maximum timestamp and its data
                $x.maxT = i$;
                $x.dataT = data$;
            $x.commitsCounter++$;
            if $(x.commitsCounter = K)$ then
                //Overwrite $x$
                $x.ts = x.maxT$;
                $x.data = x.dataT$;
                reset $x.commitsCounter$, $x.rl[]$, $x.maxT$ and $x.dataT$;
        case $(x\ is\ Counter)$
            $x.commitsCounter++$;
            if $(x.commitsCounter = K)$ then
                //Calculate the correct value considering only the committed transactions
                abort the other $live$ transactions in $x.wl[][]$;
                copy the data of $committed$ transactions from $x.wl[][]$ to $temp[][]$;
                sort($temp[][]$); //Based on the timestamps
                $max$ = max($temp[].tx$); //Maximum timestamp in $temp[][]$
                for $(from\ j \leftarrow 0\ to\ j < K)$ do
                    $correct \leftarrow correct + temp[j].data$;
                $x.data \leftarrow correct$;
                $x.ts \leftarrow max$;
                reset $x.wl[][]$ and $x.index$;
            else
                //Just Commit
                $x.wl[].status = Committed$;
        case $(x\ is\ Queue)$
            $x.commitsCounter++$;
            if $(a\ version\ of\ x \in T_i.accSet)$ then
                abort the other $live$ transactions in $x.wl[][]$;
                let $x = x_local$;
            else if $(x.commitsCounter = K)$ then
                abort the other $live$ transactions in $x.wl[][]$;
                adjust $x.queue[][]$;
                $max$ = max($queue[].tx$); //Maximum timestamp in $queue[][]$
                update $x.head$; update $x.tail$; $x.ts \leftarrow max$;
                reset $x.enqCounter$; $x.deqCounter$; $x.wl[][]$;
                reset $x.copied$; $x.commitsCounter$;
            else
                //Just Commit
                $x.wl[].status = Committed$;
Unlock() all objects in $L$;
return $true$;
```

Proof According to the implementation of Algorithm Add(), within a transaction T_i we can treat the sequence of all add operations as a single $x.add_i(v_i)$ operation which aggregates the added values of the operations to a single operand value v_i, to be added to the current value of x. Assume that in each transaction T_i there is at most one add operation on object x with operand v_i.

Let $S(x) = T_1, T_2, \ldots, T_q$ be the subsequence of transactions in S that invoke an add operation to x. A transaction T_i that overwrites x is a transaction that commits and updates $x.data = x_{local}.data$, and it also sets $x.commitsCounter = 0$ and $x.ts = i$. Let $S'(x)$ denote the subsequence of $S(x)$ consisting of all transactions that commit. Let $S''(x)$ denote the subsequence of $S(x)$ consisting of all transactions that overwrite x.

Let $r_1, \ldots, r_q$ be the respective sequence of returned values of the add operations of transactions in $S(x)$, namely, $r_i = x.add_i(v_i)$ is the value returned by T_i. Let r'_g denote the *precise value* that would have been returned by the add operation of transaction T_g when the consistency is precise, that is, $r'_g = v_0 + \sum_{(1 \leq l < g) \wedge T_l \in S'(x)} v_l$, where v_0 is the initial value of x. Let $o_g = r_g + v_g$ be the result of the add operation of transaction T_g. Similarly, let $o'_g = r'_g + v_g$ be the respective precise result of the add operation of transaction T_g.

For each $T_g \in S(x)$ let P_g denote the set of the last K transactions which precede T_g in $S'(x)$. We only need to prove the following two properties:

(i) For each $T_g \in S''(x)$, $r_g = r'_g$ (the transactions in $S''(x)$ are precise).

(ii) For each $T_g \in S(x) \setminus S''(x)$, either $r_g = o_l$ and $T_l \in P_g$ and $T_l \in S''(x)$, or $r_g = v_0$ and $|P_g| < K$.

We prove these properties by induction on q. For $q = 0$, $S(x)$ is empty and the properties hold trivially. Assume now that the properties hold for any $q < i$; we will show that the properties hold also for $q = i > 0$.

Consider now the last transaction T_i. According to function Add(), T_i checks $x.ts$ that shows the timestamp of the last transaction that overwrites x. Suppose that $x.ts = j > i$. Let Q be the set of transactions that appear in S between T_j and T_i and that have an add operation to x and commit (Q does not contain T_j or T_i). We continue to show that $|Q| \leq K - 1$.

We first show that none of the transactions in Q overwrite x. Suppose for the sake of contradiction that there is a transaction $T_m \in Q$ which overwrites x. Note that $j < m < i$. We examine three cases with respect to when T_m commits in H':

- T_m commits before T_j commits.
  In this case, when T_j invokes TryC() it observes one of the following two scenarios:
  - $m \in x.wl$ and $m > j$: T_j observes that a transaction T_m with higher timestamp ($m > j$) has committed on x but does not overwrite x, and hence T_j aborts.
  - $x.ts \geq j$: Since T_m commits and overwrites x based on the Algorithm Commit(), T_m aborts all live transactions with smaller timestamp than m, and hence T_j aborts.

  In either scenario, T_j aborts, which is impossible.
- T_m commits and overwrites x before $x.add_i()$ starts.
  In this case, T_i reads either the value of x written by T_m or by a more recent transaction (with timestamp $x.ts \geq m$). However, this contradicts the assumption that T_i reads $x.ts = j$.
- T_m commits and overwrites x after $x.add_i()$ ends.
  In this case, in its TryC() transaction T_m will observe that $x.commitsCounter \geq K - 1$ (which means T_m is the K^{th} transaction), and also it observes that T_i is in the writer list of x (that is, $i \in x.wl$ with $i > m$), and the combination of these two observations together force T_m to abort, which is a contradiction.

Therefore, no transaction in Q overwrites x. This implies that each transaction in Q increments $x.commitsCounter$. Therefore, similar to the proof of Lemma 6 the transactions in Q assign unique values to $x.commitsCounter$. Since the $x.commitsCounter$ cannot exceed $K - 1$ and none of the transactions in Q set $x.commitsCounter = 0$, we get that $|Q| \leq K-1$. The same observations for Q hold even if T_i reads the initial value of x and T_j is replaced by a special event that initializes x.

Suppose now that $T_i \in S''$. When T_i invokes Algorithm Commit() the only committed transactions in the writer list of x are the ones in set Q. Therefore, the value returned by the add operation of T_i is equal to $r_i = o_j + \sum_{T_l \in Q} v_l$. By induction hypothesis, $r_j = r'_j$, and hence $r_i = r'_i$; therefore property (i) holds. If $T_i \in S \setminus S''$, then it returns $r_i = r_j = r'_j$. Since, $Q \cup \{T_j\} \subseteq P_i$ property (ii) holds as well. (Note that properties (i) and (ii) hold even if T_i reads the initial value v_0 and T_j is replaced with a special initialization event of x.) □

Lemma 8 *The history S is K-legal, for any queue object x.*

Proof According to the implementation of algorithms Enqueue() and Dequeue(), within a transaction T_i each individual enqueue or dequeue operation is considered in the legality specification of the queue object x. So assume that in transaction T_i, each enqueue and dequeue operation on object x is denoted as qO_i.

Let $S(x) = T_1, T_2, \ldots, T_q$ be the subsequence of transactions in S that invoke enqueue or dequeue operations to x. A transaction T_g that overwrites x is a transaction that commits and updates $x.queue[][]$, $x.head$ and $x.tail$. It also sets $x.copied = 0$, $x.ts = g$ and $x.commitsCounter = 0$; Let $S'(x)$ denote the subsequence of $S(x)$ consisting of all transactions that commit. Let $S''(x)$ denote the subsequence of $S(x)$ consisting of all transactions that overwrite x. Let $V(x)$ denote all individual operations in $S(x)$ and let $V'(x)$ denote all individual operations in $S'(x)$ while $V''(x)$ denote all individual operations in $S''(x)$. For any queue operation $qOp_{g,u}$, g is the timestamp of the transaction that executes $qOp_{g,u}$ and u is the order of $qOp_{g,u}$ in V.

Let qs denote the queue status and $qs_{0,0}, \ldots, qs_{q,d}$ be the respective sequence of returned queue status of the enqueue and dequeue operations of transactions in $S(x)$, namely, $qs_{g,first}, \ldots, qs_{g,last}$ represent the queue statuses returned by T_g. Let $sq'_{g,u}$ denote the *precise queue status* that would have been returned by any queue operation of transaction T_g when the consistency is precise, that is, $qs'_{g,u} = qOp_{0,0} \wedge \sum_{(1 \leq l < j) \wedge qOp_{l,s} \in V'(x)} qOp_{l,s}$, where $qOp_{0,0}$ is the initial value of x. Let $o_{g,u} = qs_{g,a} \wedge qOp_{g,u}$ be the result of any queue operation of transaction T_g. Similarly, let $o'_{g,u} = qs'_{g,u} \wedge qOp_{g,u}$ be the respective *precise* result of the queue operation of transaction T_g.

For each $qOp_{g,u} \in V(x)$ let $P_{g,u}$ denote the set of the last K operations which precede $qOp_{g,u}$ in $V(x)$. We only need to prove the following two properties:

(i) For each $qOp_{g,u} \in V''(x)$, $qs_{g,u} = qs'_{g,u}$ (the operations in $V''(x)$ are precise.

(ii) For each $qOp_{g,u} \in V(x) \setminus V''(x)$, either $qs_{g,u} = o_{r,z}$ and $qOp_{r,z} \in P_{g,u}$ and $qOp_{r,z} \in V''(x)$, or $qs_{g,u} = qs_{0,0}$ and $|P_{g,u}| < K$.

We prove these properties by induction on q. For $q = 0$, $V(x)$ is empty and the properties hold trivially. Assume now that the properties hold for any $q < n$; we will show that the properties hold also for $q = n > 0$.

Consider now the last operation $qOp_{i,n}$. According to function Enqueue() and Dequeue(), $qOp_{i,n}$ checks $x.ts$ that shows the timestamp of the last transaction that overwrites x. Suppose that $x.ts = j \geq i$. Let Q be the set of operations that appear in V between $qOp_{j,a} \in T_j$ and $qOp_{i,n} \in T_i$ and that have queue operations to x and commit (Q does not contain $qOp_{j,a}$ or $qOp_{i,n}$). We continue to show that $|Q| \leq K-1$.

We first show that none of the operations in Q overwrite x. Suppose for the sake of contradiction that there is an operation $qOp_{m,b} \in Q$ which overwrites x. Note that $j, a \leq m, b \leq i$. We examine three cases with respect to when $qOp_{m,b}$ belongs to a committed transaction T_m in H':

- T_m commits before T_j commits.
  In this case, when T_j invokes TryC() it observes one of the following scenarios:
  - $m \in x.wl$ and T_j observes that a transaction T_m with higher timestamp ($m > j$) has committed on x but does not overwrite x, and hence T_j aborts (considering that $j \neq m \neq i$).

- $x.ts \geq j$: Since T_m commits and overwrites x based on the Algorithm Commit(), T_m aborts all live transactions with smaller timestamp than m, and hence T_j aborts (considering that $j \neq m \neq i$).
- Since some of these operations may belong to the same transaction, and since T_m commits before T_j commits, then $qOp_{j,a}$ and $qOp_{m,b}$ belong to different transactions.
  1. Now let us assume that $j = i$ (which means $qOp_{j,a}$ and $qOp_{i,n}$ belong to the same transaction). Based on Enqueue() and Dequeue(), $qOp_{j,a}$ would copy x to T_j local memory and $qOp_{i,n}$ would read from the local copy. Then $qOp_{m,b}$ would find that the object is copied by another concurrent transaction, so T_m aborts and $qOp_{m,b}$ would not execute between $qOp_{j,a}$ and $qOp_{i,n}$ which means it cannot be in Q, contradiction.
  2. If $m = i$ and then $qOp_{m,b}$ would copy x to T_m local memory and $qOp_{i,n}$ would read from the local copy, which contradicts our assumption that $qOp_{i,n}$ reads $qOp_{j,a}$.

In all scenarios, either T_j aborts which is impossible, or $qOp_{m,b}$ is not in Q.

- T_m commits and overwrites x before $qOp_{i,n}$ starts.
  In this case, $qOp_{i,n}$ reads either the value of $x.head$ and/or $x.tail$ that is written by $qOp_{m,b}$ or by a more recent operation However, this contradicts the assumption that $qOp_{i,n}$ reads $qOp_{j,n}$. Actually this also holds if $j = m$, $j = m = i$ or $m = i$.

  On the other hand if $j = i$, then $qOp_{j,a}$ would copy x to T_j local memory and $qOp_{m,b}$ would find that x is copied. So T_m aborts and it cannot execute between $qOp_{j,a}$ and $qOp_{i,n}$. Thus $qOp_{m,b} \notin Q$, contradiction.

- T_m commits and overwrites x after $qOp_{i,n}()$ ends.
  In this case, we have the following scenarios:
  - In case of $j \neq m \neq i$, in its TryC() transaction T_m will observe that $x.commitsCounter \geq K-1$ (which means T_m is the K^{th} transaction), and also it observes that T_i is in the writer list of x (that is, $i \in x.wl$ with $i > m$), and the combination of these two observations together force T_m to abort, which is a contradiction.
  - In case some of these operations belong to the same transaction, we examine the following cases:
    1. Since T_m commits and overwrites x after $qOp_{i,n}()$ ends, then we cannot have that $m = i$ or $j = m = i$, otherwise $qOp_{m_b} \notin Q$.
    2. If $j = i$, then x would be copied by T_j and $qOp_{m,b} \notin Q$.
    3. If $j = m$ then x would be copied by T_j, and T_i would abort and $qOpi, n$ cannot execute.

Therefore, no operation $qOp_{m,b}$ in Q overwrites x. This implies that each operation in Q increments $x.commitsCounter$. Therefore, similar to the proof of Lemma 6 the operations in Q assign unique values to $x.commitsCounter$. Since the $x.commitsCounter$ cannot exceed $K - 1$ and none of the operations in Q set $x.commitsCounter = 0$, we get that $|Q| \leq K - 1$. The same observations for Q hold even if $qOp_{i,n}$ reads the initial value of x and $qOp_{j,a}$ is replaced by a special event that initializes x.

Suppose now that $qOp_{i,n} \in V''$ which implies that $T_i \in S''$. When T_i invokes Algorithm Commit(), the only committed transactions in the writer list of x are the ones in set Q. Therefore, the value returned by the $qOp_{i,n}$ of T_i is equal to $qs_{i,n} = qs_j \wedge \sum_{qOp_l \in Q} qOp_{i_n}$. By induction hypothesis, $qs_{j,a} = qs'_{j,a}$, and hence $qs_{i,n} = qs'_{i,n}$, and therefore, property (i) holds. If $qOp_{i,n} \in V \setminus V''$ (which implies $T_i \in S \setminus S''$), then it returns $qs_{i,n} = qs_{j,a} = qs'_{j,a}$. Since, $Q \cup \{qOp_{j,a}\} \subseteq P_{i,n}$ property (ii) holds as well. (Note that properties (i) and (ii) hold even if $qOp_{i,n}$ reads the initial value $qs_{0,0}$ and $qOp_{j,a}$ is replaced with a special initialization event of x.) □

Based on Lemmas 5, 6, 7 and 8, we obtain the following theorem.

Theorem 3 *Any execution history H of our algorithm is K-opaque.*

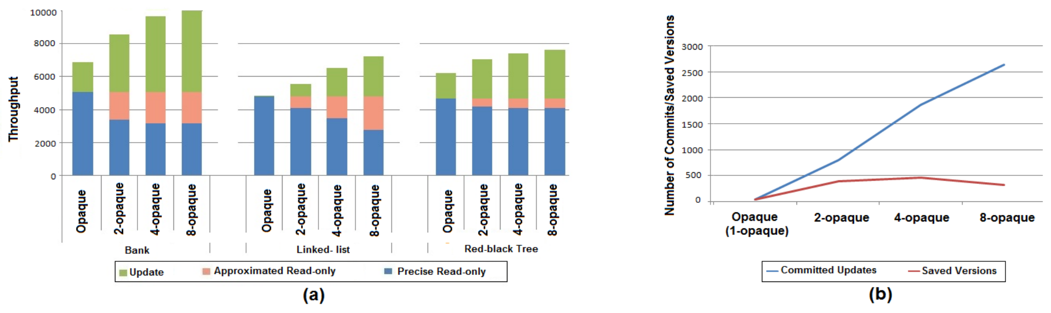

Figure 7: (a) Comparison of the throughputs (committed transactions per time) of opaque, 2-opaque, 4-opaque and 8-opaque using Bank, Linked-list and Red-black tree benchmarks; (b) Comparison of the number of committed updates to the number of the saved versions in opaque, 2-opaque, 4-opaque and 8-opaque using Linked-list benchmark.

6 Experimental Results

In our experimental analysis, we simulate the Bank, Linked-list and Red-black Tree benchmarks from TinySTM-1.0.5 [10]. We run the experiments on a machine with dual Intel(R) Xeon(R) CPU E5-2630 (6 cores total) clocked at 2.30 GHz. Each run of the benchmark takes about 5500 milliseconds using 10 threads. We test the benchmarks to compare the opaque execution (1-opaque) with 2-opaque, 4-opaque and 8-opaque. The Bank benchmark is used to test read, write and counting operations. In Bank benchmark, there are three kinds of operations which are read balance, write amount and transfer. In the Linked-list and Red-black Tree benchmarks, we have search operations, add and delete node. Read balance (in Bank) and search (in Linked-list and Red-black Tree) are read-only, but write, transfer (in Bank) and add/delete node (in Linked-list and Red-black Tree) are update transactions.

6.1 K-opacity for Read-only Transactions Using Read/write Object

First we show the results of applying K-opacity on read-only transactions using multi-version STM. In Figure 7(a), we compare the throughput (commits per time) of an opaque execution (1-opaque), 2-opaque, 4-opaque and 8-opaque using the three benchmarks. Clearly, the relaxed opacity in 2-opaque, 4-opaque and 8-opaque helps to avoid some aborts and to improve the throughput. Furthermore, in 1-opaque all read-only transactions are precise but in 2-opaque, 4-opaque and 8-opaque the percentage of approximated read-only transactions is smaller than the percentage of the precise ones. We note that there is an increase in the number of committed updates since relaxing the opacity of a read-only transaction sometimes allows to avoid many aborts; as 1-opaque read-only transaction may conflict with many update ones.

Figure 7(b) shows a comparison between the number of committed updates and the number of the saved versions using Linked-list benchmark. In 1-opaque the number of committed updates and the number of the saved versions are the same, since we save a new version with each committed update. In 2-opaque and 4-opaque, the number of saved version increases because such relaxations allow to commit very large number of updates. However, in 8-opaque the number of committed updates increases but the number of non-saved versions is very large (as we save 1 version every 8 commits), so the number of saved versions decreases.

6.2 K-opaque STM for Count Object

Now we apply K-opacity concept on all kind of transactions of STM. In this part we focus on the count object and we show the results using single-version and multi-version STM.

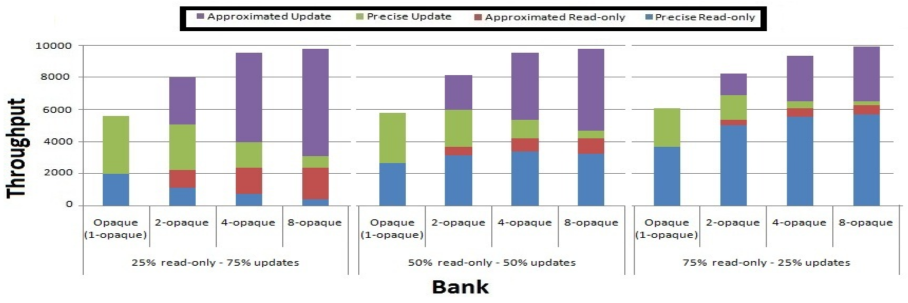

Figure 8: Comparison of the throughput of 1-opaque, 2-opaque, 4-opaque and 8-opaque executions on the Bank benchmark with 25% read-only and 75% updates, 50% read-only and 50% updates, and 75% read-only and 25% update transactions.

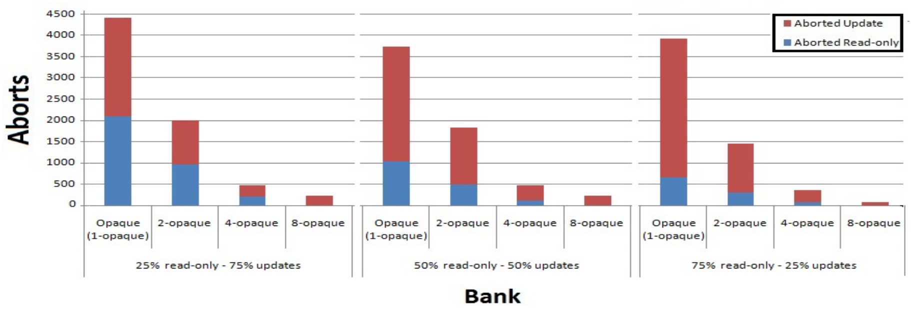

Figure 9: Comparison of the aborts of 1-opaque, 2-opaque, 4-opaque and 8-opaque executions on the Bank benchmark with 25% read-only and 75% updates, 50% read-only and 50% updates, and 75% read-only and 25% update transactions.

6.2.1 K-opaque Single-version STM

We apply K-opacity concept on single-version STM. we modify the objects of Bank benchmark such that each account is a count object. Also there are three kinds of operations which are read balance ($add(0)$), write amount ($add(data)$, where $data$ is a positive or negative number), and transfer (which has two add operations to subtract a number from one account and add it to another account). Read balance represents a read-only transaction, but write and transfer are update transactions. We run our experiments with different ratio of read-only and update transactions.

Figure 8 shows the throughput of 1-opaque, 2-opaque, 4-opaque and 8-opaque executions on the Bank benchmark. The figure shows an execution of 25% read-only and 75% updates, where in opaque execution all committed transactions are precise. In 2-opaque, 4-opaque and 8-opaque executions the throughput increases because the relaxation (which means to have some approximated transactions) of some read-only and update transactions results in the avoidance of some aborts. Also the throughput increases with a large value of K because there is a higher commit rate since update transactions get faster as their commit phase creates new versions with lower frequency. Moreover, the same thing happens with executions of 50% read-only and 50% updates, and 75% read-only and 25% update transactions.

Figure 9 shows the abort rate of 1-opaque, 2-opaque, 4-opaque and 8-opaque executions on the Bank benchmark. The figure shows executions of 25% read-only and 75% updates, 50% read-only and 50% updates, and 75% read-only and 25% update transactions. In all executions the abort rate decreases as we relax the opacity. In 8-opaque, the abort rate drops by about 88% and all read-only transactions are committed.

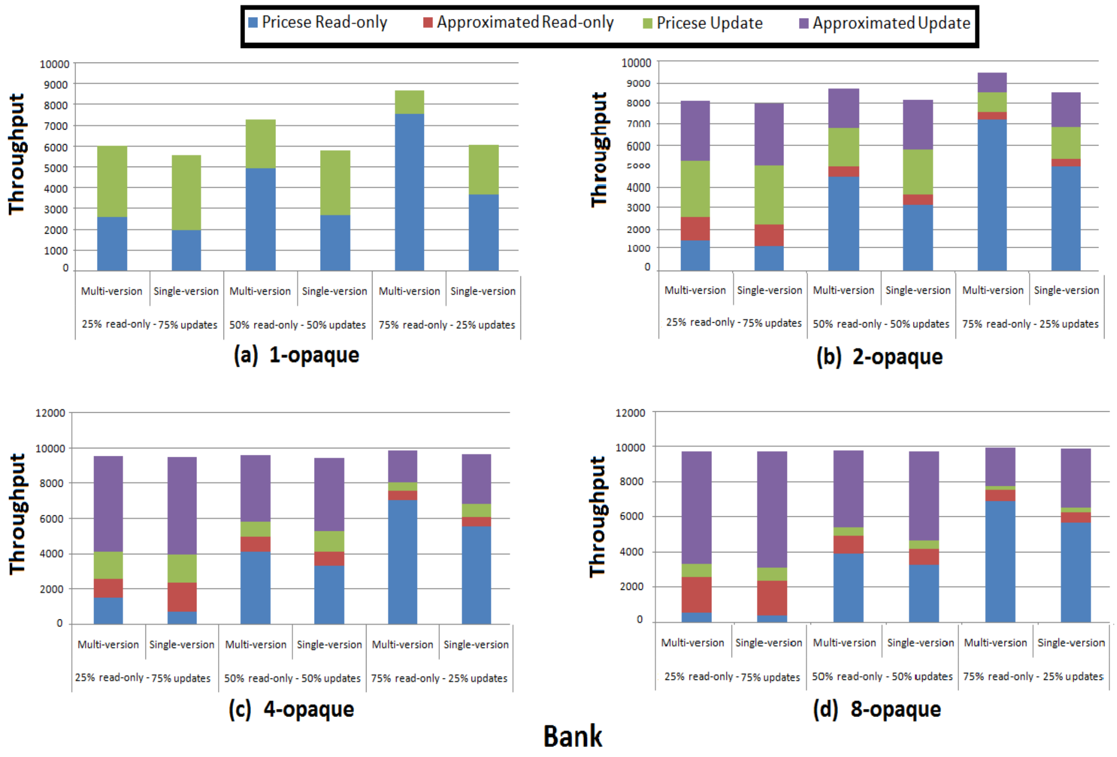

Figure 10: (a) Comparison of the throughput of 1-opaque single-version STM to 1-opaque multi-version STM using the Bank benchmark; (b) Comparison of the throughput of 2-opaque single-version STM to 2-opaque multi-version STM using the Bank benchmark; (c) Comparison of the throughput of 4-opaque single-version STM to 4-opaque multi-version STM using Bank benchmark; (d) Comparison of the throughput of 8-opaque single-version STM to 8-opaque multi-version STM using the Bank benchmark.

6.2.2 K-opaque Multi-version STM

After we applied the concept of approximated opacity on the single-version STM, we applied the same concept on multi-version STM. In multi-version STM each shared object may have multiple versions. In this way, when an update transaction commits, it creates new versions for the objects in its write set. Using multiple memory object versions, read-only transactions can successfully commit (without aborting at all) by preserving the versions in the access sets of the transactions. Therefore, we have slightly modified the structure of the count object and the algorithm to satisfy multi-version specifications.

Now we compare the performance of the single-version STM to the multi-version STM using the Bank benchmark. We use different ratios of read-only and update transactions.

Figure 10(a) Compares the throughput of 1-opaque single-version STM to 1-opaque multi-version STM using the Bank benchmark. With all different ratios of read-only and update transactions, multi-version STM usually shows better performance than the single-version one. More improvement of throughput happens when there is a higher ratio of read-only transactions, since in multi-version STM read-only transactions never abort. The same thing happens with Figure 10(b) which compares the throughput of 2-opaque single-version STM to 1-opaque multi-version STM using the Bank benchmark. However, because of the relaxation of opacity we have some approximated read-only and approximated update transactions. In addition, Figures 10(c) and 10(d) show the 4-opaque and the 8-opaque executions. In both, the throughput of multi-version STM and the throughput of single-version STM are almost the same with all different ratios of read-only and update transactions. This happens because more relaxation of the single-version STM allows to commit more read-only

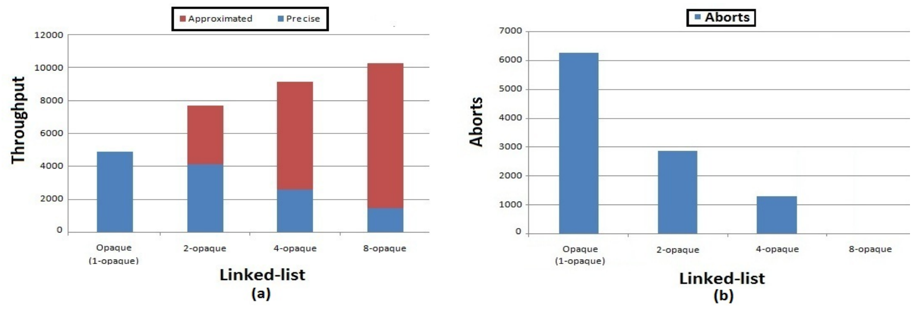

Figure 11: (a) Comparison of the throughput of 1-opaque, 2-opaque, 4-opaque and 8-opaque executions on the Linked-list benchmark; (b) Comparison of the abort rate of 1-opaque, 2-opaque, 4-opaque and 8-opaque executions on the Linked-list benchmark.

transactions without aborting update ones, while the multi-version STM originally has such property. However, in multi-version STM, the ratio of precise read-only transactions is usually higher.

6.3 K-opaque STM for Queue Object

Now, we use the Linked-list benchmark to test the queue operations. We initialize 70 lists (we modify the structure of each list to be a queue object) and we use the *add* and *delete* nodes to simulate enqueue and dequeue operations. Both enqueue and dequeue operations are considered update transactions and for queue specification we consider a single-version STM only.

Figure 11(a) demonstrates the throughput of 1-opaque, 2-opaque, 4-opaque and 8-opaque executions on the Linked-list benchmark. It shows an execution of 50% enqueues and 50% dequeues, where all of them are update transactions. The throughput of the 8-opaque execution is the maximum since the throughput improves as we relax the opacity. Moreover, Figure 11(b) shows the abort rate on the Linked-list benchmark. The figure illustrates the fast drop of the abort rate as we relax opacity.

7 Related Work

STM is a suitable paradigm to support parallel processing as it is able to guarantee progressiveness which aims to reduce aborts [13, 22].

An approach to reduce aborts is the multi-version STMs which keep multiple versions of each memory object [11, 18]. In case of a conflict this kind of STMs prevent aborts by allowing some operations of the conflicted transactions to use old object versions and maintain consistency at the same time. The main quality of using multi-version STMs is to never abort read-only transactions.

In addition, *opacity* is a property which is used to ensure the correctness of concurrent executions, such that the parallel execution of transactions must be matched with a sequential one [12].

A relaxed model that is proposed for database problems, classifies database objects into two classes which are sensitive and non-sensitive objects. Then they use strong consistency with sensitive objects and weak consistency with the non-sensitive ones [23]. However, we use the concept of opacity to verify the correctness of the execution in STM and we relax the opacity concept to K-opacity.

In fact, K-opacity is stronger than *Read Uncommitted* [5] which allows *dirty reads* while K-opacity does not. Also K-opacity is stronger than *Read Committed* [6] since K-opacity does not allow non-repeatable reads such that the transactions are serializable on each object even if any transaction accesses the same object multiple times. Also K-opacity is stronger than *Repeatable Read* [5] such that phantom reads are limited. Indeed phantom reads may happens within no more than K concurrent transactions. Thus K-opacity preserves an approximated serializability up to the limit K.

Snapshot Isolation (SI) [20] allows to read old values, but it still ensures that each transaction sees a consistent snapshot. So, SI is stronger than K-opacity transactions that may see a K-consistent snapshot which preserves a relaxed (approximated) serializability.

A *Lazy Snapshot Algorithm* (LSA) [19] allows older reads, but it still ensures that each transaction sees a consistent snapshot, while in K-opacity we may see a K-consistent snapshot (by proper adaptation of the snapshot definition). On the other hand, in LSA when the transaction arrives (is issued), it gets a timestamp and it reads from the versions that already committed before that point (i.e. it does not read from concurrent transactions). However, in K-opacity the transactions read from concurrent updates only if they have smaller timestamps. Therefore, LSA guarantees serializability, while K-opacity guarantees relaxed (approximated) serializability. In short with K-opacity we can improve the performance comparing to LSA by relaxing the consistency and avoiding some aborts.

8 Conclusion

In conclusion, our algorithm usually allows to commit some of the conflicted transactions, due to relaxed consistency, which improves the performance. The count object could be extended to execute other arithmetic operations as well. For that, we can add another column in the wl array to record the type of operations, i.e. multiplication or division. Also the timestamp order of the sequential history S is used for the correctness of transactions' execution. It may be possible to use another S that does not respect the timestamps' order but it allows more commits (i.e. value based approximation). For future work, the composability of approximated opacity can be applied to other object kinds (such as stacks).

References

[1] Swarup Acharya, Phillip B Gibbons, and Viswanath Poosala. Aqua: A fast decision support systems using approximate query answers. In *Proceedings of the 25th International Conference on Very Large Data Bases*, pages 754–757. Morgan Kaufmann Publishers Inc., 1999.

[2] George S Almasi and Allan Gottlieb. *Highly parallel computing.* Benjamin-Cummings Publishing Co., Inc., 1989.

[3] Basem Assiri and Costas Busch. Approximately opaque multi-version permissive transactional memory. In *Proceedings of SRMPDS'2016, Parallel Processing Workshops (ICPPW), 45th International Conference on Parallel Processing*, pages 393–402. IEEE, 2016.

[4] Basem Assiri and Costas Busch. Approximate count and queue objects in transactional memory. In *Proceedings of 19th Workshop on Advances in Parallel and Distributed Computational Models (APDCM), IPDPS Workshops.* IEEE, 2017.

[5] Hal Berenson, Phil Bernstein, Jim Gray, Jim Melton, Elizabeth O'Neil, and Patrick O'Neil. A critique of ansi sql isolation levels. In *ACM SIGMOD Record*, volume 24, pages 1–10. ACM, 1995.

[6] Josep Maria Bernabé-Gisbert, José Enrique Armendáriz-Inigo, Rubén de Juan-Marın, and Francesc D Munoz-Escoı. Providing read committed isolation level in non-blocking rowa database replication protocols. *XV Jornadas de Concurrencia y Sistemas Distribuidos (JCSD 07), Torremolinos, Spain*, pages 159–171, 2007.

[7] Chen Chen, Xifeng Yan, Feida Zhu, and Jiawei Han. gapprox: Mining frequent approximate patterns from a massive network. In *Data Mining, 2007. ICDM 2007. Seventh IEEE International Conference on*, pages 445–450. IEEE, 2007.

[8] Edgar F Codd, Sharon B Codd, and Clynch T Salley. Providing olap (on-line analytical processing) to user-analysts: An it mandate. *Codd and Date*, 32, 1993.

[9] Pascal Felber, Christof Fetzer, Patrick Marlier, and Torvald Riegel. Time-based software transactional memory. *Parallel and Distributed Systems, IEEE Transactions on*, 21(12):1793–1807, 2010.

[10] Pascal Felber, Christof Fetzer, and Torvald Riegel. Dynamic performance tuning of word-based software transactional memory. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, pages 237–246. ACM, 2008.

[11] Rachid Guerraoui, Thomas A Henzinger, and Vasu Singh. Permissiveness in transactional memories. In *Distributed Computing*, pages 305–319. Springer, 2008.

[12] Rachid Guerraoui and Michal Kapalka. On the correctness of transactional memory. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, pages 175–184. ACM, 2008.

[13] Rachid Guerraoui and Michal Kapalka. The semantics of progress in lock-based transactional memory. *ACM SIGPLAN Notices*, 44(1):404–415, 2009.

[14] Maurice Herlihy and J. Eliot B. Moss. Transactional memory: Architectural support for lock-free data structures. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*, ISCA '93, pages 289–300, New York, NY, USA, 1993. ACM.

[15] Maurice Herlihy and Nir Shavit. *The Art of Multiprocessor Programming, Revised Reprint*. Elsevier, 2012.

[16] Priyanka Kumar, Sathya Peri, and K. Vidyasankar. A timestamp based multi-version STM algorithm. In *Distributed Computing and Networking*, pages 212–226. Springer Berlin Heidelberg, 2014.

[17] Dmitri Perelman, Anton Byshevsky, Oleg Litmanovich, and Idit Keidar. SMV: selective multi-versioning STM. In *Distributed Computing*, pages 125–140. Springer, 2011.

[18] Dmitri Perelman, Rui Fan, and Idit Keidar. On maintaining multiple versions in STM. In *Proceedings of the 29th ACM SIGACT-SIGOPS symposium on Principles of distributed computing*, pages 16–25. ACM, 2010.

[19] Torvald Riegel, Pascal Felber, and Christof Fetzer. A lazy snapshot algorithm with eager validation. In *International Symposium on Distributed Computing*, pages 284–298. Springer, 2006.

[20] Torvald Riegel, Christof Fetzer, and Pascal Felber. Snapshot isolation for software transactional memory. In *First ACM SIGPLAN Workshop on Languages, Compilers, and Hardware Support for Transactional Computing (TRANSACT'06)*, pages 1–10. Association for Computing Machinery (ACM), 2006.

[21] Adrian Sampson, Jacob Nelson, Karin Strauss, and Luis Ceze. Approximate storage in solid-state memories. *ACM Transactions on Computer Systems (TOCS)*, 32(3):9, 2014.

[22] Nir Shavit and Dan Touitou. Software transactional memory. *Distributed Computing*, 10(2):99–116, 1997.

[23] Mukul K Sinha. Nonsensitive data and approximate transactions. *Software Engineering, IEEE Transactions on*, (3):314–322, 1983.

[24] S Skiena. Dijkstras algorithm. *Implementing Discrete Mathematics: Combinatorics and Graph Theory with Mathematica, Reading, MA: Addison-Wesley*, pages 225–227, 1990.