Opacity Proof for CaPR+ Algorithm

Anshu S Anand
Homi Bhabha National Institute, Mumbai
anshusanand2001@gmail.com

R K Shyamasundar
Indian Institute of Technology Mumbai
shyamasundar@gmail.com

Sathya Peri
Indian Institute of Technology Hyderabad
sathya_p@iith.ac.in

ABSTRACT

In this paper, we describe an enhanced Automatic Checkpointing and Partial Rollback algorithm($CaPR^+$) to realize Software Transactional Memory(STM) that is based on continuous conflict detection, lazy versioning with automatic checkpointing, and partial rollback. Further, we provide a proof of correctness of $CaPR^+$ algorithm, in particular, Opacity, a STM correctness criterion, that precisely captures the intuitive correctness guarantees required of transactional memories. The algorithm provides a natural way to realize a hybrid system of pure aborts and partial rollbacks. We have also implemented the algorithm, and shown its effectiveness with reference to the Red-black tree micro-benchmark and STAMP benchmarks. The results obtained demonstrate the effectiveness of the Partial Rollback mechanism over pure abort mechanisms, particularly in applications consisting of large transaction lengths.

Keywords

STM, transaction, opacity, correctness, multi-core

1. INTRODUCTION

The challenges posed by the use of low-level synchronization primitives like locks led to the search of alternative parallel programming models to make the process of writing concurrent programs easier. Transactional Memory is a promising programming memory in this regard.

A Software Transactional Memory(STM)[2] is a concurrency control mechanism that resolves data conflicts in software as compared to in hardware by HTMs.

STM provides the programmers with high-level constructs to delimit transactional operations and with these constructs in hand, the programmer just has to demarcate atomic blocks of code, that identify critical regions that should appear to execute atomically and in isolation from other threads. The underlying transactional memory implementation then implicitly takes care of the correctness of concurrent accesses to the shared data. The STM might internally use fine-grained locking, or some non-blocking mechanism, but this is hidden from the programmer and the application thereby relieving him of the burden of handling concurrency issues.

Several STM implementations have been proposed, which are mainly classified based on the following metrics:
1) shared object update(version management) - decides when does a transaction update its shared objects during its lifetime.
2) conflict detection - decides when does a transaction detect a conflict with other transactions in the system.
3) concurrency control - determines the order in which the events - conflict, its detection and resolution occur in the system.

Each software transaction can perform operations on shared data, and then either commit or abort. When the transaction commits, the effects of all its operations become immediately visible to other transactions; when it aborts, all its operations are rolled back and none of its effects are visible to other transactions. Thus, abort is an important STM mechanism that allows the transactions to be atomic. However, abort comes at a cost, as an abort operation implies additional overhead as the transaction is required to be re-executed after canceling the effects of the local transactional operations. Several solutions have been proposed for this, that are based on partial rollback, where the transaction rolls back to an intermediate consistent state rather than restarting from beginning. [4] was the first work that illustrated the use of checkpoints in boosted transactions and [11] suggested using checkpoints in HTMs. In [5] the partial rollback operation is based only on shared data that does not support local data which requires extra effort from the programmer in ensuring consistency. [7] and [6] is an STM algorithm that supports both shared and local data for partial rollback. [12] is another STM that supports both shared and local data. Our work is based on [7]. We present an improved and simplified algorithm, Automatic Checkpointing and Partial Rollback algorithm($CaPR^+$) and prove its correctness.

Several correctness criteria exist for STMs like linearizability, serializability, rigorous scheduling, etc. However, none of these criteria is sufficient to describe the semantics of TM with its subtleties. Opacity is a criterion that captures precisely the correctness requirements that have been intu-

itively described by many TM designers. We discuss Opacity in section 2 and present the proof of opacity of $CaPR^+$ algorithm in section 4.2.

2. SYSTEM MODEL

The notations defined in this section have been inspired from [3]. We assume a system of n processes (or threads), $p_1, \ldots, p_n$ that access a collection of *objects* via atomic *transactions*. The processes are provided with the following *transactional operations*: *begin_tran*() operation, which invokes a new transaction and returns the *id* of the new transaction; the $write(x, v, i)$ operation that updates object x with value v for a transaction i, the $read(x)$ operation that returns a value read in x, $tryC()$ that tries to commit the transaction and returns *commit* (c for short) or *abort* (a for short), and $tryA()$ that aborts the transaction and returns A. The objects accessed by the read and write operations are called as t-objects. For the sake of presentation simplicity, we assume that the values written by all the transactions are unique.

Operations *write*, *read* and *tryC* may return a, in which case we say that the operations *forcefully abort*. Otherwise, we say that the operation has *successfully* executed. Each operation is equipped with a unique transaction identifier. A transaction T_i starts with the first operation and completes when any of its operations returns a or c. Abort and commit operations are called *terminal operations*. For a transaction T_k, we denote all its read operations as $Rset(T_k)$ and write operations $Wset(T_k)$. Collectively, we denote all the operations of a transaction T_i as $evts(T_k)$.

Histories. A *history* is a sequence of *events*, i.e., a sequence of invocations and responses of transactional operations. The collection of events is denoted as $evts(H)$. For simplicity, we only consider *sequential* histories here: the invocation of each transactional operation is immediately followed by a matching response. Therefore, we treat each transactional operation as one atomic event, and let $<_H$ denote the total order on the transactional operations incurred by H. With this assumption the only relevant events of a transaction T_k are of the types: $r_k(x, v)$, $r_k(x, A)$, $w_k(x, v)$, $w_k(x, v, A)$, $tryC_k(C)$ (or c_k for short), $tryC_k(A)$, $tryA_k(A)$ (or a_k for short). We identify a history H as tuple $\langle evts(H), <_H \rangle$.

Let $H|T$ denote the history consisting of events of T in H, and $H|p_i$ denote the history consisting of events of p_i in H. We only consider *well-formed* histories here, i.e., (1) each $H|T$ consists of a read-only prefix (consisting of read operations only), followed by a write-only part (consisting of write operations only), possibly *completed* with a $tryC$ or $tryA$ operation[a], and (2) each $H|p_i$ consists of a sequence of transactions, where no new transaction begins before the last transaction completes (commits or a aborts).

We assume that every history has an initial committed transaction T_0 that initializes all the data-objects with 0. The set of transactions that appear in H is denoted by $txns(H)$. The set of committed (resp., aborted) transactions in H is denoted by $committed(H)$ (resp., $aborted(H)$). The set of *incomplete* or *live* transactions in H is denoted by $incomplete(H)$ ($incomplete(H) = txns(H) - committed(H) - aborted(H)$).

For a history H, we construct the *completion* of H, denoted $\overline{H}$, by inserting a_k immediately after the last event of every transaction $T_k \in incomplete(H)$.

Transaction orders. For two transactions $T_k, T_m \in txns(H)$, we say that T_k *precedes* T_m in the *real-time order* of H, denote $T_k \prec_H^{RT} T_m$, if T_k is complete in H and the last event of T_k precedes the first event of T_m in H. If neither $T_k \prec_H^{RT} T_m$ nor $T_m \prec_H^{RT} T_k$, then T_k and T_m *overlap* in H. A history H is *t-sequential* if there are no overlapping transactions in H, i.e., every two transactions are related by the real-time order.

For two transactions T_k and T_m in $txns(H)$, we say that T_k *precedes* T_m *in conflict order*, denoted $T_k \prec_H^{CO} T_m$ if: (a) (w-w order) $c_k <_H c_m$ and $Wset(T_k) \cap Wset(T_m) \neq \emptyset$; (b) (w-r order) $c_k <_H r_m(x, v)$, $x \in Wset(T_k)$ and $v \neq A$; (c) (r-w order) $r_k(x, v) <_H c_m$ and $x \in Wset(T_m)$ and $v \neq A$. Thus, it can be seen that the conflict order is defined only on operations that have successfully executed.

Valid and legal histories. Let H be a history and $r_k(x, v)$ be a read operation in H. A successful read $r_k(x, v)$ (i.e $v \neq A$), is said to be *valid* if there is a transaction T_j in H that commits before r_K and $w_j(x, v)$ is in $evts(T_j)$. Formally, $\langle r_k(x, v)$ is valid $\Rightarrow \exists T_j : (c_j <_H r_k(x, v)) \wedge (w_j(x, v) \in evts(T_j)) \wedge (v \neq A)\rangle$. The history H is valid if all its successful read operations are valid.

We define $r_k(x, v)$'s *lastWrite* as the latest commit event c_i such that c_i precedes $r_k(x, v)$ in H and $x \in Wset(T_i)$ (T_i can also be T_0). A successful read operation $r_k(x, v)$ (i.e $v \neq A$), is said to be *legal* if transaction T_i (which contains r_k's lastWrite) also writes v onto x. Formally, $\langle r_k(x, v)$ *is legal* $\Rightarrow (v \neq A) \wedge (H.lastWrite(r_k(x, v)) = c_i) \wedge (w_i(x, v) \in evts(T_i))\rangle$. The history H is legal if all its successful read operations are legal. Thus from the definitions we get that if H is legal then it is also valid.

Opacity. We say that two histories H and H' are *equivalent* if they have the same set of events. Now a history H is said to be *opaque* [9, 15] if H is valid and there exists a t-sequential legal history S such that (1) S is equivalent to $\overline{H}$ and (2) S respects $\prec_H^{RT}$, i.e $\prec_H^{RT} \subset \prec_S^{RT}$. By requiring S being equivalent to $\overline{H}$, opacity treats all the incomplete transactions as aborted.

Implementations and Linearizations. A (STM) implementation is typically a library of functions for implementing: $read_k$, $write_k$, $tryC_k$ and $tryA_k$ for a transaction T_k. We say that an implementation M_p is correct w.r.t to a property P if all the histories generated by M_p are in P. The histories generated by an STM implementations are normally not sequential, i.e., they may have overlapping transactional operations. Since our correctness definitions are proposed for sequential histories, to reason about correctness of an implementation, we order the events in a non-concurrent history in a sequential manner. The ordering must respect the real-time ordering of the operations in the original history. In other words, if the response operation o_i occurs before the

[a]This restriction brings no loss of generality [13].

invocation operation o_j in the original history then o_i occurs before o_j in the sequential history as well. Overlapping events, i.e. events whose invocation and response events do not occur either before or after each other, can be ordered in any way.

We call such an ordering as *linearization* [8]. Now for a (non-sequential) history H generated by an implementation M, multiple such linearizations are possible. An implementation M is considered *correct* (for a given correctness property P) if every its history has a correct linearization (we say that this linearization is exported by M).

We assume that the implementation has enough information to generate an unique linearization for H to reason about its correctness. For instance, implementations that use locks for executing conflicting transactional operations, the order of access to locks by these (overlapping) operations can decide the order in obtaining the sequential history. This is true with STM systems such as [17, 16, 14] which use locks.

3. CAPR+ ALGORITHM

In this section, we present the data structures and the $CaPR^+$ Algorithm. The various data structures used in the $CaPR^+$ Algorithm are categorized into local workspace and global workspace, depending on whether the data structure is visible to the local transaction or every transaction. The data structures used in the local workspace are as follows:

1. Local Data Block(LDB) - Each entry consists of the local object and its current value in the transaction(Table 1).

2. Shared object Store(SOS) - An entry in Table 2 stores the address of the shared object, its value, a read flag and a write flag. Both read and write flags have 0 as the initial value. Value 1 in read/write flag indicates the object has been read/written by the transaction, respectively.

3. Checkpoint Log(Cplog) - Used to partially rollback a transaction as shown in Table 3, where each entry stores, a) the shared object whose read initiated the log entry (this entry is made every time a shared object is read for the first time by the transaction), b) program location from where the transaction should proceed after a rollback, and c) the current snapshot of the transaction's local data block and the shared object store.

Table 1: Local Data Block

Object	Value

Table 2: Shared object Store

Object	Current Value	Read flag	Write flag

The data structures in the global workspace are:

Table 3: Checkpoint Log

Victim Shared object	Program Location	Local Snapshot

Table 4: Global List of Active Transactions

Transaction ID	Status Flag	Conflict Objects

1. Global List of Active Transactions(Actrans) - Each entry in this list contains a) a unique transaction identifier, b) a status flag that indicates the status of the transaction, as to whether the transaction is in conflict with any of the committed transactions, and c) a list of all the objects in conflict with the transaction. This list is updated by the committed transactions.

2. Shared Memory(SM) - Each entry in the shared memory stores a) a shared object, b) its value, and c) an active readers list that stores the transaction IDs of all the transactions reading the shared object.

The $CaPR^+$ algorithm is shown in Algorithm 1.

4. CONFLICT OPACITY

In this section we describe about *Conflict Opacity* (CO), a subclass of Opacity using conflict order (defined in Section 2). This subclass is similar to conflict serializability of databases whose membership can be tested in polynomial time (in fact it is more close to order conflict serializability) [18, Chap 3].

DEFINITION 1. *A history H is said to be* conflict opaque *or* co-opaque *if H is valid and there exists a t-sequential legal history S such that (1) S is equivalent to $\overline{H}$ and (2) S respects $\prec_H^{RT}$ and $\prec_H^{CO}$.*

From this definition, we can see that co-opaque is a subset of opacity.

4.1 Graph characterization of co-opacity

Given a history H, we construct a *conflict graph*, $CG(H) = (V, E)$ as follows: (1) $V = txns(H)$, the set of transactions in H (2) an edge (T_i, T_j) is added to E whenever $T_i \prec_H^{RT} T_j$ or $T_i \prec_H^{CO} T_j$, i.e., whenever T_i precedes T_j in the real-time or conflict order.

Note, since $txns(H) = txns(\overline{H})$ and $(\prec_H^{RT} \cup \prec_H^{CO}) = (\prec_{\overline{H}}^{RT} \cup \prec_{\overline{H}}^{CO})$, we have $CG(H) = CG(\overline{H})$. In the following lemmas, we show that the graph characterization indeed helps us verify the membership in co-opacity.

LEMMA 2. *Consider two histories $H1$ and $H2$ such that $H1$ is equivalent to $H2$ and $H1$ respects conflict order of $H2$, i.e., $\prec_{H1}^{CO} \subseteq \prec_{H2}^{CO}$. Then, $\prec_{H1}^{CO} = \prec_{H2}^{CO}$.*

Algorithm 1 CaPR Algorithm

```
1: procedure ReadTx(t, o, pc)
2:     if o is in t's local data block then
3:         str.val ← o.val from LDB
4:         return l ← 1(Success);
5:     else if o is in t's shared object store then
6:         str.val ← o.val from SOS
7:         return l ← 1(Success);
8:     else if o is in shared memory then
9:         obtain locks on object o, & the entry for transaction t;
10:        if t.status_flag = RED then
11:            Unlock lock on t and o
12:            PL = partially_Rollback(t);
13:            update str.PL = PL
14:            return l ← 0(Rollback);
15:        create checkpoint entry in checkpoint log for o;
16:        str.val ← o.val from Shared Memory
17:        add t to o's readers' list
18:        add o into SOS and set its read flag to 1;
19:        release locks on o and t;
20:        return l ← 1(Success);
21:    else                                  ▷ o not in shared memory
22:        return l ← 2(Error);
23: procedure WriteTx(o, t)
24:    if o is a local object then
25:        update o in local data block;
26:    else if o is a shared object then
27:        if o is in shared object store then
28:            update o in SOS and set its write flag to 1;
29:        else
30:            insert o in SOS and set its write flag to 1;
31: procedure commitTx(t)
32:    Assign t's write-set, t.WS =
       {o|o is in SOS and o's write flag = 1}
33:    Sort all objects in t.WS, and obtain locks on them;
34:    Initialize A = {t};
35:    for each object o in the t.WS
36:        A = A ∪ active readers of o;
37:    Sort all transactions in 'A', and obtain locks on them;
38:    if t.status_flag = RED then
39:        PL = partially_Rollback(t);
40:        release all locks;
41:        return PL;
42:    for each object, wo in t's write set, t.WS
43:        update wo.value in SM from the local copy of wo.
44:        for each transaction rt in wo's active readers' list,
45:            add the objects in t.WS to transaction rt's
               conflict objects' list;
46:            set transaction rt's status flag to RED;
47:    delete t from actrans;
48:    for each object, ro in t's readers-list
49:        delete t from ro's active readers list;
50:    release all locks;
51:    return 0;
52: procedure Partially_Rollback(t)
53:    identify safest checkpoint - earliest conflicting object;
54:    apply selected checkpoint;
55:    delete t from active reader's list of all objects rolled back
56:    reset status flag to GREEN;
57:    return PL(the new program location);
```

Table 5: Shared Memory

Shared object	Value	List of active readers

Proof. Here, we have that $\prec_{H1}^{CO} \subseteq \prec_{H2}^{CO}$. In order to prove $\prec_{H1}^{CO} = \prec_{H2}^{CO}$, we have to show that $\prec_{H2}^{CO} \subseteq \prec_{H1}^{CO}$. We prove this using contradiction. Consider two events p, q belonging to transaction $T1, T2$ respectively in $H2$ such that $(p, q) \in \prec_{H2}^{CO}$ but $(p, q) \notin \prec_{H1}^{CO}$. Since the events of $H2$ and $H1$ are same, these events are also in $H1$. This implies that the events p, q are also related by CO in $H1$. Thus, we have that either $(p, q) \in \prec_{H1}^{CO}$ or $(q, p) \in \prec_{H1}^{CO}$. But from our assumption, we get that the former is not possible. Hence, we get that $(q, p) \in \prec_{H1}^{CO} \Rightarrow (q, p) \in \prec_{H2}^{CO}$. But we already have that $(p, q) \in \prec_{H2}^{CO}$. This is a contradiction. □

Lemma 3. *Let $H1$ and $H2$ be equivalent histories such that $\prec_{H1}^{CO} = \prec_{H2}^{CO}$. Then $H1$ is legal iff $H2$ is legal.*

Proof. It is enough to prove the 'if' case, and the 'only if' case will follow from symmetry of the argument. Suppose that $H1$ is legal. By contradiction, assume that $H2$ is not legal, i.e., there is a read operation $r_j(x, v)$ (of transaction T_j) in $H2$ with lastWrite as c_k (of transaction T_k) and T_k writes $u \neq v$ to x, i.e $w_k(x, u) \in evts(T_k)$. Let $r_j(x, v)$'s lastWrite in $H1$ be c_i of T_i. Since $H1$ is legal, T_i writes v to x, i.e $w_i(x, v) \in evts(T_i)$.

Since $evts(H1) = evts(H2)$, we get that c_i is also in $H2$, and c_k is also in $H1$. As $\prec_{H1}^{CO} = \prec_{H2}^{CO}$, we get $c_i <_{H2} r_j(x, v)$ and $c_k <_{H1} r_j(x, v)$.

Since c_i is the lastWrite of $r_j(x, v)$ in $H1$ we derive that $c_k <_{H1} c_i$ and, thus, $c_k <_{H2} c_i <_{H2} r_j(x, v)$. But this contradicts the assumption that c_k is the lastWrite of $r_j(x, v)$ in $H2$. Hence, $H2$ is legal. □

From the above lemma we get the following interesting corollary.

Corollary 4. *Every co-opaque history H is legal as well.*

Based on the conflict graph construction, we have the following graph characterization for co-opaque.

Theorem 5. *A legal history H is co-opaque iff $CG(H)$ is acyclic.*

Proof. *(Only if)* If H is co-opaque and legal, then $CG(H)$ is acyclic: Since H is co-opaque, there exists a legal t-sequential history S equivalent to $\overline{H}$ and S respects $\prec_H^{RT}$ and $\prec_H^{CO}$. Thus from the conflict graph construction we have that $CG(\overline{H})(= CG(H))$ is a sub graph of $CG(S)$. Since S is sequential, it can be inferred that $CG(S)$ is acyclic. Any sub graph of an acyclic graph is also acyclic. Hence $CG(H)$ is also acyclic.

(if) If H is legal and $CG(H)$ is acyclic then H is co-opaque: Suppose that $CG(H) = CG(\overline{H})$ is acyclic. Thus we can perform a topological sort on the vertices of the graph and obtain a sequential order. Using this order, we can obtain a sequential schedule S that is equivalent to $\overline{H}$. Moreover, by construction, S respects $\prec_H^{RT}=\prec_{\overline{H}}^{RT}$ and $\prec_H^{CO}=\prec_{\overline{H}}^{CO}$.

Since every two events related by the conflict relation (w-w, r-w, or w-r)in S are also related by $\prec_{\overline{H}}^{CO}$, we obtain $\prec_S^{CO}=\prec_{\overline{H}}^{CO}$. Since H is legal, $\overline{H}$ is also legal. Combining this with Lemma 3, we get that S is also legal. This satisfies all the conditions necessary for H to be co-opaque. □

4.2 Proof of Opacity for CaPR+ Algorithm

In this section, we will describe some of the properties of $CaPR^+$ algorithm and then prove that it satisfies opacity. In our implementation, only the read and tryC operations access the memory. Hence, we call these operations as memory operations. The main idea behind our algorithm is as follows: Consider a live transaction T_i which has read a value u for t-object x. Suppose a transaction T_j writes a value v to t-object x and commits. When T_i executes the next memory operation (after the c_j), T_i is rolled back to the step before the read of x. We denote that T_j has *invalidated* the T_i's read of x. Transaction T_i then reads x again.

The following example illustrates this idea. Consider the history $H1 : r_1(x,0)r_1(y,0)r_2(x,0)r_1(z,0)$
$w_2(y,5)c_2w_1(x,5)$. In this history, when T_1 performs any other memory operation such as a read after $C-$, it will then be rolled back to the step $r_1(y)$ causing it to read y again.

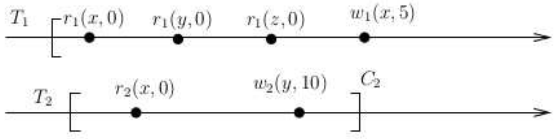

Figure 1: Pictorial representation of a History $H1$

Thus as explained, in our algorithm, when a transaction's read is invalidated, it does not abort but rather gets rolled back. In the worst case, it could get rolled back to the first step of the transaction which is equivalent to the transaction being aborted and restarted. Thus with this algorithm, a history will consist only of incomplete (live) and committed transactions.

To precisely capture happenings of the algorithm and to make it consistent with the model we discussed so far, we modify the representation of the transactions that are rolled back. Consider a transaction T_i which has read x. Suppose another transaction T_j that writes to x and then commits. Thus, when T_i performs its next memory operation, say m_i (which could either be a read or commit operation), it will be rolled back. We capture this rollback operation in the history as two transactions: $T_{i.1}$ and $T_{i.2}$.

Here, $T_{i.1}$ represents all the successful operations of transaction T_i until it executed the memory operation m_i which caused it to roll back (but not including that m_i). Transaction $T_{i.1}$ is then terminated by an abort operation $a_{i.1}$. Then, after transaction T_j has committed transaction $T_{i.2}$ begins. Unlike $T_{i.1}$ it is incomplete. It also consists of all same operations of $T_{i.1}$ until the read on x. $T_{i.2}$ reads the latest value of the t-object x again since it has been invalidated by T_j. It then executes future steps which could depend on the read of x. With this modification, the history consists of committed, incomplete as well as aborted transactions (as discussed in the model).

In reality, the transaction T_i could be rolled back multiple times, say n. Then the history H would contain events from transactions $T_{i.1}, T_{i.2}, T_{i.3}....T_{i.n}$. But it must be noted that all the invocations of T_i are related by real-time order. Thus, we have that $T_{i.1} \prec_H^{RT} T_{i.2} \prec_H^{RT} T_{i.3}.... \prec_H^{RT} T_{i.n}$

With this change in the model, the history $H1$ is represented as follows, $H2 : r_{1.1}(x,0)r_{1.1}(y,0)r_{2.1}(x,0)$
$r_{1.1}(z,0)w_{2.1}(y,5)c_{2.1}w_1(x,5)a_{1.1}r_{1.2}(x,0)r_{1.2}(y,10)$.

For simplicity, from now on in histories, we will denote a transaction with greek letter subscript such as α, β, γ etc regardless of whether it is invoked for the first time or has been rolled back. Thus in our representation, transaction $T_{i.1}, T_{i.2}$ could be denoted as T_α, T_γ respectively.

We will now prove the correctness of this algorithm. We start by describing a property that captures the basic idea behind the working of the algorithm.

PROPERTY 6. *Consider a transaction T_i that reads t-object x. Suppose another transaction T_j writes to x and then commits. In this case, the next memory operation (read or tryC) executed by T_i after c_j returns abort (since the read of x by T_i has been invalidated).*

For a transaction T_i, we define the notion of *successful final memory operation(sfm).* As the name suggests, it is the last successfully executed memory operation of T_i. If T_i is committed, then $sfm_i = c_i$. If T_i is aborted, then sfm_i is the last memory operation, in this case a read operation, that returned *ok* before being aborted.

For proving the correctness, we use the graph characterization of co-opacity described in Section 4.

Consider a history H_{capr} generated by the CaPR algorithm. Let $CG(H_{capr})$ be the conflict graph of H_{capr}. We show that this graph denoted, g_{capr}, is acyclic.

LEMMA 7. *Consider a path p in g_{capr} abstracted as: $T_{\alpha 1} \rightarrow T_{\alpha 2} \rightarrow .... \rightarrow T_{\alpha k}$. Then, $sfm_{\alpha 1} <_{H_{capr}} sfm_{\alpha 2} <_{H_{capr}} .... <_{H_{capr}} sfm_{\alpha k}$.*

PROOF. We prove this using induction on k.

Base Case, $k = 2$. In this case the path consists of only one edge between transactions $T_{\alpha 1}$ and $T_{\alpha 2}$. Let us analyse the various types of edges possible:

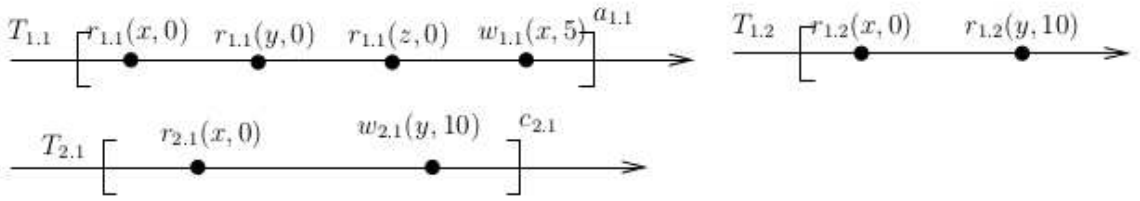

Figure 2: Pictorial representation of the modified History $H2$

- *real-time edge:* This edge represents real-time. In this case $T_{\alpha 1} \prec^{RT}_{H_{capr}} T_{\alpha 2}$. Hence, we have that $sfm_{\alpha 1} <_{H_{capr}} sfm_{\alpha 2}$.
- *w-w edge:* This edge represents w-w order conflict. In this case both transactions $T_{\alpha 1}$ and $T_{\alpha 2}$ are committed and $sfm_{\alpha 1} = c_{\alpha 1}$ and $sfm_{\alpha 2} = c_{\alpha 2}$. Thus, from the definition of this conflict, we get that $sfm_{\alpha 1} <_{H_{capr}} sfm_{\alpha 2}$.
- *w-r edge:* This edge represents w-r order conflict. In this case, $c_{\alpha 1} <_{H_{capr}} r_{\alpha 2}(x,v)$ $(v \neq A)$. For transaction $T_{\alpha 1}$, $sfm_{\alpha 1} = c_{\alpha 1}$. For transaction $T_{\alpha 2}$, either $r_{\alpha 2} <_{H_{capr}} sfm_{\alpha 2}$ or $r_{\alpha 2} = sfm_{\alpha 2}$. Thus in either case, we get that $sfm_{\alpha 1} <_{H_{capr}} sfm_{\alpha 2}$.
- *r-w edge:* This edge represents r-w order conflict. In this case, $r_{\alpha 1}(x,v) <_{H_{capr}} c_{\alpha 2}$ (where $v \neq A$). Thus $sfm_{\alpha 2} = c_{\alpha 2}$. Here, we again have two cases: (a) $T_{\alpha 1}$ terminates before $T_{\alpha 2}$. In this case, it is clear that $sfm_{\alpha 1} <_{H_{capr}} sfm_{\alpha 2}$. (b) $T_{\alpha 1}$ terminates after $T_{\alpha 2}$ commits. The working of the algorithm is such that, as observed in Property 6, the next memory operation executed by $T_{\alpha 1}$ after the commit operation $c_{\alpha 2}$ returns abort. From this, we get that the last successful memory operation executed by $T_{\alpha 1}$ must have executed before $c_{\alpha 2}$. Hence, we get that $sfm_{\alpha 1} <_{H_{capr}} sfm_{\alpha 2}$.

Thus in all the cases, the base case holds.

Induction Case, $k = n > 2$. In this case the path consists of series of edges starting from transactions $T_{\alpha 1}$ and ending at $T_{\alpha n}$. From our induction hypothesis, we know that it is true for $k = n-1$. Thus, we have that $sfm_{\alpha 1} <_{H_{capr}} sfm_{\alpha(n-1)}$. Now consider the transactions $T_{\alpha(n-1)}, T_{\alpha n}$ which have an edge between them. Using the arguments similar to the base case, we can prove that $sfm_{\alpha(n-1)} <_{H_{capr}} sfm_{\alpha n}$. Thus, we have that $sfm_{\alpha 1} <_{H_{capr}} sfm_{\alpha n}$.

In all the cases, we have that $sfm_{\alpha 1} <_{H_{capr}} sfm_{\alpha n}$. Hence, proved. □

Using Lemma 7, we show that g_{capr} is acyclic.

LEMMA 8. *Graph, g_{capr} is acyclic.*

PROOF. We prove this by contradiction. Suppose that g_{capr} is cyclic. Then there is a cycle going from $T_{\alpha 1} \rightarrow T_{\alpha 2} \rightarrow .... \rightarrow T_{\alpha k} \rightarrow T_{\alpha 1}$.

From Lemma 7, we get that $sfm_{\alpha 1} \rightarrow sfm_{\alpha 2} \rightarrow .... \rightarrow sfm_{\alpha k} \rightarrow sfm_{\alpha 1}$ which implies that $sfm_{\alpha 1} \rightarrow sfm_{\alpha 1}$. Hence, the contradiction. □

THEOREM 9. *All histories generated by $CaPR^+$ are co-opaque and hence, $Capr^+$ satisfies the property of opacity.*

PROOF. Proof follows from Theorem 5 and Lemma 8. □

5. CONCLUSION

In this paper, we have described $CaPR^+$, an enhanced $CaPR$ algorithm and proved its opacity. We have also implemented the same and tested its performance. While it shows good performance for transactions that take time, its performance for small transactions shows overhead which is obvious. A thorough comparison with STAMP benchmarks with varying transactions has been done and shows good results. This will be reported elsewhere. Further, we have been working on several optimizations like integrating both partial rollback and abort mechanisms in the same implementation to exploit the benefits of both mechanisms, and also integrate with it the contention management.

6. REFERENCES

[1] M. Herlihy and J. E. B. Moss. âĂIJTransactional memory: architectural support for lock-free data structuresâĂİ. In Proc. of the *Twentieth Annual International Symposium on Computer Architecture*, pp. 289âĂŞ300, San Diego, California, 1993, ACM Press.

[2] N. Shavit and D. Touitou. "Software transactional memory". *In Distributed Computing*, Special Issue (10) : 99116, 1997.

[3] Petr Kuznetsov and Sathya Peri. On non-interference of transactions. CoRR, abs/1211.6315, 2012

[4] Koskinen E and Herlihy M, "Checkpoints and continuations instead of nested transactions". In Proceedings of the *Twentieth annual symposium on Parallelism in algorithms and architectures (SPAA* 08*)* (New York, NY, USA, 2008), ACM, pp. 160-168

[5] Lupei, D.: "A study of conflict detection in software transactional memory". Masters thesis, University of Toronto, the Netherlands (2009)

[6] Gupta, M., Shyamasundar, R.K., Agarwal, S.: "Clustered checkpointing and partial rollbacks for reducing conflict costs in stms", IBM IRL Report, also in International Journal of Computer Applications 1(22) (February 2010) 80-85

[7] Gupta, M., Shyamasundar, R.K., Agarwal, S. "Automatic checkpointing and partial rollback in software transaction memory", (January 2012) IBM patent Publication 20110029490.

[8] M. Herlihy and J. M. Wing. "Linearizability: a correctness condition for concurrent objects", *ACM Transactions on Programming Languages and Systems*, 12(3):463âĂŞ492, June 1990.

[9] R. Guerraoui and M. Kapalka, "On the correctness of transactional memory", In Proceedings of the *13th ACM SIGPLAN Symposium on Principles and practice of parallel programming, PPoPP 08*, pages 175-184. ACM, 2008

[10] Chi Cao Minh, JaeWoong Chung, Christos Kozyrakis, and Kunle Olukotun, "STAMP: Stanford transactional applications for multi-processing", In *IISWC 08: Proceedings of The IEEE International Symposium on Workload Characterization*, September 2008.

[11] M. M. Waliullah and P. Stenstrom. "Intermediate checkpointing with conflicting access prediction in transactional memory systems". In IPDPS, pages 1-11.IEEE Computer Society, 2008

[12] Porfirio Alice et. al. "Transparent Support for Partial Rollback in Software Transactional Memories", Euro-Par 2013 Parallel Processing, 2013

[13] Petr Kuznetsov and Srivatsan Ravi. "On the cost of concurrency in transactional memory". In OPODIS, pages 112-127, 2011.

[14] Hagit Attiya and Eshcar Hillel. "A single-version stm that is multi-versioned permissive". Theory Comput. Syst., 51(4):425-446, 2012.

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

[16] Tyler Crain, Damien Imbs, and Michel Raynal. "Read invisibility, virtual world consistency and probabilistic permissiveness are compatible". In ICA3PP (1), pages 244-257, 2011.

[17] Damien Imbs and Michel Raynal. "A lock-based stm protocol that satisfies opacity and progressiveness". In OPODIS 08: Proceedings of the 12th International Conference on Principles of Distributed Systems, pages 226-245, Berlin, Heidelberg, 2008. Springer-Verlag.

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