Opacity Proof for CaPR+ Algorithm

Anshu S. Anand, R.K. Shyamasundar* and Sathya Peri†
Homi Bhabha National Institute, Mumbai 400084

ABSTRACT

Software Transactional Memory(STM) is a new programming paradigm that can be an effective alternative to the conventional parallel programming models and languages. It absolves the programmer of having to synchronize and coordinate parallel computations, and instead delegates these to the compiler and run-time systems. In this paper, we describe an enhanced Automatic Checkpointing and Partial Rollback algorithm($CaPR^+$) to realize STMs 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 refernce to the Red-black tree microbenchmark and STAMP benchmarks. The results obtained demonstrate the effectiveness of the Partial Rollback mechanism over abort mechanism, 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)[1] is a concurrency control mechanism that resolves data conflicts in software as compared to in hardware by HTMs.

*IIT Bombay, Mumbai 400076
†IIT Hyderabad, 502205

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 reexecuted after cancelling 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. [3] was the first work that illustrated the use of checkpoints in boosted transactions and [13] suggested using checkpoints in HTMs. In [4] 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. [6] and [5] is an STM algorithm that supports both shared and local data for partial rollback. [14] is another STM that supports both shared and local data. Our work is based on [6]. 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 intuitively 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 [2]. 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* [8, 17] 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

[a] This restriction brings no loss of generality [15].

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 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* [7]. 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 [19, 18, 16] which use locks.

3. CAPR+ ALGORITHM

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

1. Local Data Block - Each entry in the local data block consists of the local object and its current value in the transaction.
2. Shared object Store - The shared object store contains a) the shared object, b) its initial value, c) an info-flag that gives information about the shared object. Value 1 of this flag indicates the shared object has been read from the shared memory, value 2 indicates it has been modified locally by the transaction, and value 3 indicates the object has been created locally.
3. Checkpoint Log - It is used to partially rollback a transaction. Each entry in the checkpoint log consists of 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) the 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

The data structures in the global workspace are as follows:

Table 2: Shared object Store

Object	Current Value	Info flag

Table 3: Checkpoint Log

Victim Shared object	Program Location	Local Snapshot

1. Global List of Active Transactions - 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 - 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.

Table 4: Global List of Active Transactions

Transaction ID	Status Flag	Conflict Objects

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) [20, 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. We now give an algorithm to show that the membership of co-opacity can be tested in polynomial time. This algorithm is based on graph characterisation.

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 lem-

Algorithm 1 CaPR Algorithm

```
procedure ReadTx(t, o, pc)
    if o is in t's local data block then
        read value of o from LDB into str-⟩val;
        update l = 1(Success);
    else if o is in t's shared object store then
        read value of o from SOS into str-⟩val;
        update l = 1(Success);
    else if o is in shared memory then
        obtain locks on shared object o, & transaction, t;
        if color of t's status flag = RED then
            PL = partially_Rollback(t);
            update str-⟩PL = PL, l = 0(Rollback);
        create checkpoint entry in checkpoint log for o;
        read o from shared memory into str-⟩val;
        update l = 1(Success);
        add t to o's readers' list
        insert o into shared object store;
        release locks on o and t;
    else                                  ▷ o not in shared memory
        update l = 2(Error);
    release lock on transaction t;
    return l;
procedure WriteTx(o, t)
    if o is a local object then
        update o in local data block;
    else if o is a shared object then
        update o in shared object store;
procedure commitTx(t)
    find the write set of t;
    obtain locks on all objects in its write set(after sorting);
    for each object o in the write set
        collect its active readers to 'A'
    Add t to 'A'
    obtain locks on all transactions in 'A' in a pre-defined order;
    if color of t's status flag = RED then
        PL = partially_Rollback(t);
        release all locks;
        return PL;
    if all locks obtained then            ▷ ready to commit
        write desired shared memory objects
        for each reader transaction,
            add the objects to their conflict objects' list;
            set status flag to RED;
            drop t from actrans;
        drop t from acreader's lists;
        release all locks;
    return 0;
procedure Partially_Rollback(t)
    identify safest checkpoint - (victim);
    apply selected checkpoint;
    reset status flag to GREEN;
    proceed with the new program location;
```

Table 5: Shared Memory

Shared object	Value	List of active readers

mas, 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, $\prec_{H1}^{CO} = \prec_{H2}^{CO}$.*

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. $\square$

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.

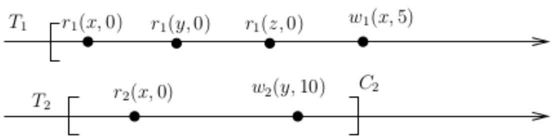

Figure 1: Pictorial representation of a History $H1$

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 $g_{capr} = CG(H_{capr})$ be the corresponding conflict graph. We will show that g_{capr} is acyclic. We start with the following lemma on the path of the graph.

LEMMA 7. *Consider the conflict graph $g_{capr} = CG(H_{capr})$, the conflict graph of H_{capr}. Consider a path p in g_{capr} as*

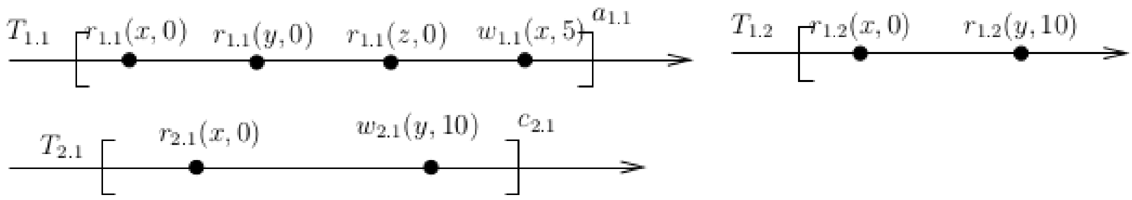

Figure 2: Pictorial representation of the modified History $H2$

follows: $T_{\alpha 1} \rightarrow T_{\alpha 2} \rightarrow .... \rightarrow T_{\alpha k}$. *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:

- *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. *Consider the conflict graph $g_{capr} = CG(H_{capr})$ the conflict graph of H_{capr}. The 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. □

This lemma shows that the history generated by H_{capr} is in co-opacity and hence in opacity.

5. PERFORMANCE EVALUATION OF CAPR ALGORITHM

Existing benchmarks may be categorised into microbenchmarks and individual(or set of) applications. Microbenchmarks are composed of transactions that execute a few operations on a data structure. These are typically easy to develop, parameterize, and port across systems. They may be useful in cases when a particular aspect of the implementation is to be selectively evaluated. However, they do not allow us to evaluate the whole implementation exhaustively. Whereas full applications have transactions that consist of many operations over many data structures, and may include a significant amount of non-transactional code as well.

Red-black tree and hash-tabes are examples of microbenchmarks used for evaluating TM systems. Short and simple transactions of microbenchmarks are good for testing mechanics of STM itself and comparing low-level details of various implementations.

STMBench7 [9] is another candidate benchmark for evaluating STM implementations. However STMBench7 targets only a specific class of applications i.e. CAD/CAM. Other benchmarks include SPLASH-2 [10], SPEComp [11], BioParallel, MineBench etc, but almost all of them lack in dealing with a wide range of transactional behaviours - contention, length of transactions, and sizes of their read and write sets. Also they are only partially portable. This leads us to another benchmark called STAMP [12] (Stanford Transactional Applications for Multi-Processing).

It is a comprehensive benchmark suite that includes eight applications spanning different classes of applications, and thirty variants of input parameters and data sets that exercise a wide range of transactional behaviours. It is for this reason that we target STAMP benchmarks for performance analysis. The main features of benchmarks thrusted upon by the authors are:

1. Breadth - the benchmark must target a variety of algorithms and application domains.

2. Depth - it must cover a wide range of transactional characteristics - transaction lengths, contention and size of read and write sets.

3. The amount of time spent in executing the transactions should be varied.

4. Portability- it must be compatible with a large variety of TM systems.

Table 6 depicts the different applications and their brief description.

Experimental setup

The experiments were performed on the following platforms:

1. Intel Xeon X5650 - comprising 6 cores, operating at 2.7GHz.

2. TILEPro64 Tile processor - comprising 64 cores, operating at 700MHz.

The TILEPro64 processor consists of 64 general-purpose, three-wide issue Very Long Instruction Word (VLIW) CPUs that are are interconnected using an 8 x 8 mesh-based network-on-chip.

To analyse the performance gain obtained by the partial rollback mechansim, we compare the CaPR implementation with a light-weight version of the CaPR implementation that implements abort mechanism. This is done by eliminating the book-keeping required by the CaPR algorithm like checkpointing. The comparison is done by using the kmeans, genome and ssca2 applications of the STAMP benchmarks. The experiments involve analysing the execution time of a benchmark application with the abort and partial-rollback implementations for different number of threads.

Observations: From Figures 3 and 4, its evident that abort mechanism performs better for the kmeans application. This can be explained by the small length of transactions in kmeans which renders rollback less feasible. However, an interesting point to note is that in Figure 4, there is only a marginal difference in performance of both mechanism. This is because kmeans does not use transactions frequently, it spends very less of its execution time in transactions(which is as low as 3 percent for low contention), where as for Figure 3, the overhead for rollback is exposed completely, without

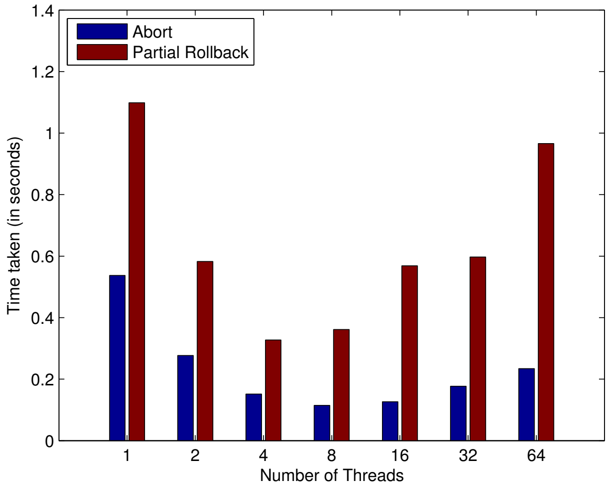

Figure 3: Results for kmeans, input 1

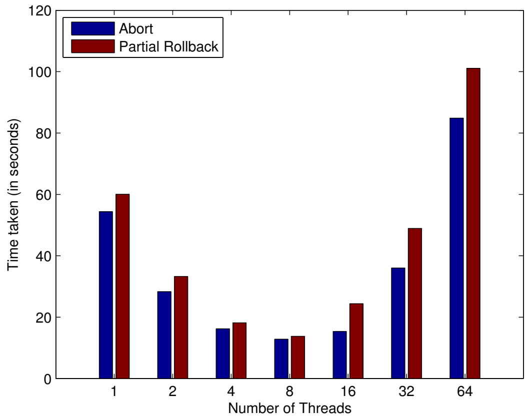

Figure 4: Results for kmeans, input 3(high contention)

contributing any performance gain. Similarly, for ssca2 application(Figures 5,6) too, the abort mechanism fares well. Here, the amount of contention is relatively low due to infrequent concurrent updates of the same adjacency list. This is because of the large number of graph nodes resulting in lesser number of conflicts, and hence lesser number of aborts/rollbacks.

This application uses transactions quite frequently, as it operates continuously on shared data structures and a significant amount of execution time is spent in transactions. Also, the mean number of instructions per transaction is higher in genome as compared to kmeans and ssca2, which makes genome an ideal candidate for studying the impact of the partial rollback mechanism. From Figures 7,8, an interesting trend is observed. For a single thread, the abort performs significantly well, and then as the number of threads increases, the performance of abort mechanicm gradually decreases, while that of rollback mechanism increases. And as is evident from Figure 8, the overall best performance is

Table 6: STAMP applications

Application	Domain	Description
bayes	machine learning	Learns structure of a Bayesian network
genome	bioinformatics	Performs gene sequencing
intruder	security	Detects network intrusions
kmeans	data mining	Implements K-means clustering
labyrinth	engineering	Routes paths in maze
ssca2	scientific	Creates efficient graph representation
vacation	online transaction processing	Emulates travel reservation system
yada	scientific	Refines a Delaunay mesh

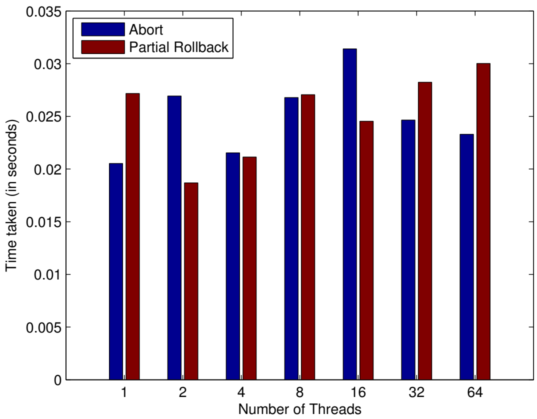

Figure 5: Results for ssca2, input 1

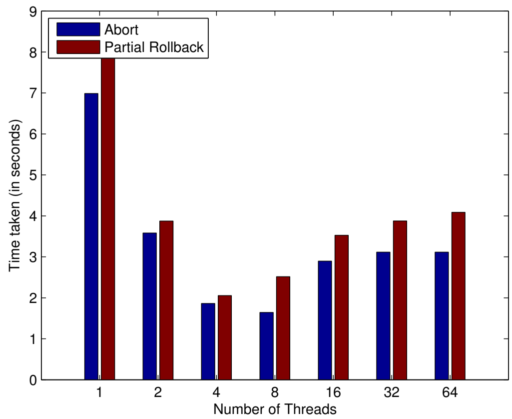

Figure 6: Results for ssca2, input 2

achieved for 16 threads for the rollback mechanism, resulting in around 10% performance gain.

Another important point to note is that with greater number of threads, there is higher contention, leading to greater number of rollbacks/aborts. This is apparent from the results, and justifies why the performance takes a hit beyond 8/16 threads. This is also because the Xeon processor consists of only 6 cores. For further substantiation, we conducted the same set of experiments on TilePro64, the results for which also exhibit similar trends. However, due to availability of greater parallelism in the form of 64 cores, we now get the best results for 16/32 threads, beyond which the performance starts degrading again. Similarly, for Red-Black tree microbenchmarks too, we found partial rollback to be better as compared to abort mechanism. The results have been shown in the Table 7 for 100 transactions.

6. AN INTEGRATED ABORT-PARTIAL ROLLBACK FRAMEWORK FOR STMS

The partial rollback mechanism is expected to perform well with applications in which transaction lengths(determined by number of reads) are relatively longer. With smaller transactions, the overhead incurred in book-keeping exceeds the performance improvement achieved using rollback. In these circumstances, it makes sense for us to run the transactions using abort mechanism.

Table 7: Results for Red-Black tree microbenchmark for 100 transactions

	Abort		Partial Rollback	
No. Threads	Time(in ms)	Abort	Time(in ms)	Rollbacks
1	474	0	460	0
2	456	81	443	75
4	642	117	626	101
8	942	134	747	160
16	1209	196	1291	184

Towards this, we give an integrated abort-partial rollback framework, that provides support for the user to designate some of the transactions(typically with small transactional length), to run with partial rollback mechanism, and others to run with abort mechanism, which entails minimum book-keeping. This allows us to extract maximum benefit by exploiting both the mechanisms simultaneously.

Apart from transactional length, which can be easily determined by the user, another factor that affects the utility of partial rollback is contention. Contention in the application, however, may not be determined/known to the user apriori. In order to address this, we need to provide an additional instrumentation mechanism which measures the

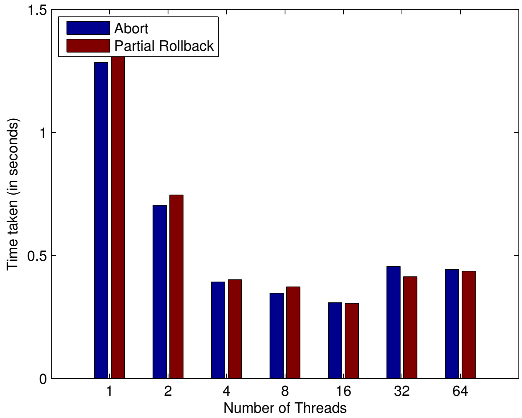

Figure 7: Results for genome, input 1

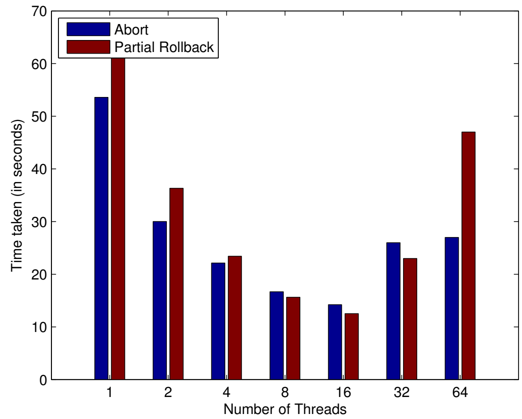

Figure 8: Results for genome, input 2

contention. One way to achieve this is to keep track of the average number of conflicts occurring in the execution per unit time. This information could be used by the transactions to determine dynamically the mechanism to which it should subscribe to - abort or partial rollback.

7. CONCLUSION

We have modified the CaPR algorithm for performance and presented the proof for opacity for CaPR algorithm, which guarantees the correctness required of transactional memories. STAMP benchmark suite is used to evaluate the performance of the partial rollback mechanism, implemented by CaPR algorithm. The results show that the partial rollback mechanism is found to be effective for applications where the transaction length is more, while for shorter transactions, abort mechanism performs better. We exploit this observation by suggesting an integrated Abort-Partial Rollback framework that provides support for utilizing both the abort and partial rollback mechanism simultaneously. The current implementation has been realized without much optimizations, so we are trying to optimize it(rollback mechanism in particular) resulting in better performance, so that our implementation can be effectively compared with other existing STM implementations like RSTM, and then extend it further to the hybrid model, incorporating both the partial rollback and abort mechanisms.

8. REFERENCES

[1] N. Shavit and D. Touitou. ‘Software transactional memory‘. *In Distributed Computing*, Special Issue (10):99âĂŞ116, 1997.

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

[3] 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 âĂŹ08)* (New York, NY, USA, 2008), ACM, pp. 160âĂŞ168

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

[5] Gupta, M., Shyamasundar, R.K., Agarwal, S.: Article: Clustered checkpointing and partial rollbacks for reducing conflict costs in stms. International Journal of Computer Applications 1(22) (February 2010) 80âĂŞ85

[6] Gupta, M., Shyamasundar, R.K., Agarwal, S. Automatic checkpointing and partial rollback in software transaction memory (January 2012) US Patent 20110029490.

[7] 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.

[8] R. Guerraoui and M. Kapalka, “On the correctness of transac- tional memory”, In Proceedings of the *13th ACM SIGPLAN Symposium on Principles and practice of parallel program- ming, PPoPP âĂŹ08*, pages 175-184. ACM, 2008

[9] Rachid Guerraoui, Michal Kapalka, and Jan Vitek, “Stmbench7: a benchmark for software transactional memory”, *SIGOPS Oper. Syst. Rev.*, 41(3):315-324, 2007.

[10] S. C. Woo, M. Ohara, et al, “The SPLASH2 Programs: Characterization and Methodological Considerations”, In Proceedings of the *22nd Interna- tional Symposium on Computer Architecture*, pp. 24-36. June 1995.

[11] Standard Performance Evaluation Corporation, SPEC OpenMP Bench- mark Suite. http://www.spec.org/omp.

[12] Chi Cao Minh, JaeWoong Chung, Christos Kozyrakis, and Kunle Olukotun, “STAMP: Stanford transactional applications for multi-processing”, In *IISWC âĂŹ08: Proceedings of The IEEE International Symposium on Workload Characterization*, September 2008.

[13] 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

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

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

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

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

[18] Tyler Crain, Damien Imbs, and Michel Raynal. Read invisibility, virtual world consistency and probabilistic permissiveness are compatible. In ICA3PP (1), pages 244âĂŞ257, 2011.

[19] Damien Imbs and Michel Raynal. A lock-based stm protocol that satisfies opacity and progressiveness. In OPODIS âĂŹ08: Proceedings of the 12th International Conference on Principles of Distributed Systems, pages 226âĂŞ245, Berlin, Heidelberg, 2008. Springer-Verlag.

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