Runtime Checking of Serializability in Software Transactional Memory

Arnab Sinha, Sharad Malik
Dept. of Electrical Engineering, Princeton University
{*sinha,sharad*}*@princeton.edu*

***Abstract*—Ensuring the correctness of complex implementations of software transactional memory (STM) is a daunting task. Attempts have been made to formally verify STMs, but these are limited in the scale of systems they can handle [1], [2], [3] and generally verify only a model of the system, and not the actual system. In this paper we present an alternate attack on checking the correctness of an STM implementation by verifying the execution runs of an STM using a checker that runs in parallel with the transaction memory system. With future many-core systems predicted to have hundreds and even thousands of cores [4], it is reasonable to utilize some of these cores for ensuring the correctness of the rest of the system. This will be needed anyway given the increasing likelihood of dynamic errors due to particle hits (soft errors) and increasing fragility of nanoscale devices. These errors can only be detected at runtime. An important correctness criterion that is the subject of verification is the serializability of transactions. While checking transaction serializability is NP-complete, practically useful subclasses such as interchange-serializability (DSR) are efficiently computable [5]. Checking DSR reduces to checking for cycles in a transaction ordering graph which captures the access order of objects shared between transaction instances. Doing this concurrent to the main transaction execution requires minimizing the overhead of capturing object accesses, and managing the size of the graph, which can be as large as the total number of dynamic transactions and object accesses. We discuss techniques for minimizing the overhead of access logging which includes time-stamping, and present techniques for on-the-fly graph compaction that drastically reduce the size of the graph that needs to be maintained, to be no larger than the number of threads. We have implemented concurrent serializability checking in the Rochester Software Transactional Memory (RSTM) system [6]. We present our practical experiences with this including results for the RSTM, STAMP [7] and synthetic benchmarks. The overhead of concurrent checking is a strong function of the transaction length. For long transactions this is negligible. Thus the use of the proposed method for continuous runtime checking is acceptable. For very short transactions this can be significant. In this case we see the applicability of the proposed method for debugging.**

***Keywords*-Serializability; Software Transactional Memory;**

I. Introduction

Software transactional memory implementations can be quite complex, with several tradeoffs made to reduce the performance overhead. These tradeoffs result in choices made along several orthogonal axes such as strong/weak isolation, direct/deferred update, object granularity, concurrency control, conflict detection/resolution etc. and are reflected in the prominent STM implementations such as DSTM [8], WSTM [9], ASTM [10], RSTM [6] and McRT [11].

Attempts have been made to formally verify STMs, but these are limited in the scale of systems they can handle due to the state space explosion problem and generally verify only a model of the system, and not the actual system [1], [2], [3]. Thus, they, at best, provide limited assurance on the correctness of the STM implementation.

In this work we explore an alternate direction in checking the serializability property of an STM implementation. We propose the use of an independent serializability checker than runs in parallel to the main STM system. We envisage the use of such a checker for (i) debugging an STM implementation (ii) using it for continuous runtime checking. In the latter application, we overcome the large state space challenges of formal verification by limiting the checking to the actual traces encountered during the run. There is strong practical motivation for this: (i) the growing complexity of concurrent software systems will make it even harder to apply formal verification techniques and (ii) with future processing platforms likely to have hundreds and even thousands of cores, it is reasonable to use some of these cores for checking the correctness of the rest of the system. This will be needed anyway given the increasing likelihood of dynamic errors due to particle hits (soft errors) and increasing fragility of nanoscale devices. These errors can only be detected at runtime.

The rest of the paper is organized as follows. In Section II, we review the various notions of transaction serializability. Section III gives an overview of the proposed methodology. Following this, Section IV explains the mechanism used for logging critical events needed for the parallel checking. The parallel checker is the subject of Section V. The design choices are discussed in Section VI. In Section VII, we present the experimental setup and the results. This is followed by a survey of related work in Section VIII, and conclusions and future work in Section IX.

II. Serializability

A parallel execution of transactions is said to be serializable if the values of the variables on the completion of the transactions are the same as would have been produced by *some* sequential ordering of these transactions [12].

A. An Illustrative Example

Consider the examples depicted in Figures 1 and 2. The shared variables a and b are initialized to 0. The actual time order of execution of the instructions is referred to as the *history*. We refer to the threads executing the transactions as *transaction threads*. In this example, we assume that transactions $T1$ and $T2$ are being executed by two different transaction threads.

In both Figures 1 and 2, $T1$ begins execution before $T2$ and $T1$ commits after $T2$ commits. In the execution history of Figure 1, the value of the tuple (a,b) after $T2$ commits is (0,1) and $T1$ reads (0,1) as the value of the tuple. Effectively, $T1$ executes after the completion of execution of $T2$, although $T1$ starts execution before $T2$. Thus, $T1$ can be *serialized* after $T2$.

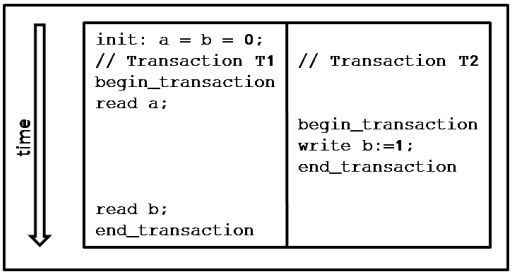


Figure 1. The value of the tuple (a,b) read by $T1$ (which is modifying none of the variables) is (0,1), and the value written by $T2$ is (0,1). Therefore, $T1$ can be serialized after $T2$. Hence, it is a serializable execution.

Next, examine the execution history of Figure 2. $T1$ reads (0,1) and $T2$ writes (1,1). For this history to be serial, either $T2$ follows $T1$ or vice versa. In the first case, if $T2$ follows $T1$, then $T1$ should have read (0,0) i.e. the unchanged initial value of the tuple. Further, in the second case, if $T1$ follows $T2$, then $T1$ should have read (1,1) i.e. the values written by $T2$. As there is no valid serial execution equivalent to this history, this execution is non-serializable.

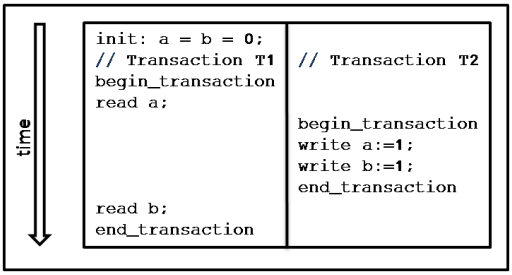


Figure 2. The value of the tuple (a,b) read by $T1$ (which is modifying none of the variables) is (0,1), whereas, the values written by $T2$ is (1,1). $T1$ should have read (0,0) or (1,1). Therefore, $T1$ reads an inconsistent value of (a,b). Hence, it is a non-serializable execution.

This illustration of serializability is generalized in the following graph-theoretic formulation.

B. Theoretical Background

Let $G(V, E)$ be a graph where V is set of vertices and E is the set of edges. A vertex u_i represents the execution of a transaction T_i and a directed edge (u_i, u_j) represents the conflicting access (read-after-write (RAW), write-after-read (WAR) and write-after-write (WAW)) of a shared variable by T_i before T_j. *These accesses are said to be conflicting because at least one of them is a write. Note that read-after-read (RAR) is not a conflicting access.* This graph captures the history of the transaction execution. The general problem of serializability is often referred to as SR. It has been shown by Papadimitriou that testing whether a history h is serializable is NP-complete [5]. He also showed that there exist certain efficiently computable subclasses of SR such as DSR (interchange serializable [5], [13]). We now give an intuition for the this NP-completeness result and how this leads to the notion of DSR.

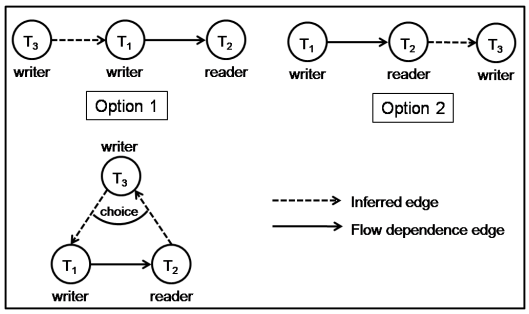


Figure 3. Serializability order and NP-completeness

Consider the example in Figure 3. Let transactions T_1 and T_3 write to a shared object o and transaction T_2 read from o. T_2 is reading the value which is written by T_1. An edge (u_i, u_j) is a *flow dependence* edge if T_j reads the value written by T_i (the bold arrows in Figure 3). Hence, (u_1, u_2) is a flow dependence edge. In other words, no other transaction that writes to o can be inserted between T_1 and T_2 in the serializability order. Therefore, $T_1 \rightarrow T_3 \rightarrow T_2$ is not a valid serializability order. Edge (u_i, u_j) is said to be *inferred* if it is required to impose a possible total order of the vertices in the graph (the broken arrows in Figure 3) consistent with the flow dependence edges. Hence, there are two options for ordering T_3, as shown in Figure 3. The order should be either $T_3 \rightarrow T_1 \rightarrow T_2$ or $T_1 \rightarrow T_2 \rightarrow T_3$. These options lead to a choice of retaining one of the two inferred edges (u_3, u_1) and (u_2, u_3). There may be several such choice-points in the graph leading to an exponential number of possibilities for the total serial order which is the source of the NP-completeness. However, if we take the chronological order of write accesses into account, then we can break the choice. In that case, either T_3 follows T_1 or vice versa. Therefore, the order is either $T_1 \rightarrow T_2 \rightarrow T_3$ (if T_1 writes before T_3), or $T_3 \rightarrow T_1 \rightarrow T_2$ (if T_3 writes before T_1). Breaking this choice results in a DSR (interchange

serializable) graph. Hence the serializability order in this graph can be computed in polynomial time as a topological order of the DSR graph. As a corollary, *a DSR graph is serializable iff it is acyclic.*

However, it should be noted that using DSR graphs may produce false positives, i.e. the DSR graph may be cyclic even though the history is serializable. Intuitively this is less likely to happen since the DSR selects the preferred order for each choice as one that is consistent with the chronological order. A false positive would require the existence of a serializability order that is inconsistent with the actual chronological order which, while possible, is less likely. Thus, there is practical value in checking DSR, and this coupled with its efficient computation make this an attractive candidate as a correctness criterion for TM systems [14].

C. Handling Nested Transactions

The above discussion of serializability is in the context of flat transactions, i.e. a transaction cannot include another transaction. However modern STMs allow for nested transactions [15]. We briefly discuss the applicability of the DSR graph for checking the serializability of different possible choices of nested transactions.

- **Single-Thread Nesting:** In this case the transaction is limited to a single thread which in practice is scheduled on a single processor. This is the case for all known STMs. The sequential execution semantics of the processor guarantee the sequential execution of a transaction and thus serializability of any nested transactions among themselves. The serializability of transactions across threads depends on the semantics of the nesting.
  - **Flattened:** In this case, aborting the inner transaction aborts the outer transaction, and committing an inner transaction is not visible to other threads till the outer transaction commits. Thus, serializability checking across threads reduces to checking the serializability of the outermost transactions in a nest.
  - **Closed:** In this case, the inner transaction aborts without aborting the outer transaction, and committing an inner transaction is not visible to other threads till the outer transaction commits. As in the flattened case, serializability checking across threads reduces to checking the serializability of the outermost transactions in a nest.
  - **Open:** In this case, the inner transaction aborts without aborting the outer transaction, and committing an inner transaction is visible to all the threads, even if the outer transaction eventually aborts. In this case, serializability checking across threads reduces to checking the serializability of the outermost *committed* transaction in a nest.
- **Multiple-Thread Nesting:** In this case a single transaction can be split across multiple threads. While this is not seen in STMs thus far, it is present in database systems. In this case the serializability condition for correctness can get quite complex (e.g., [16]). Since this is not present in STMs, we do not consider this further.

In summary, the DSR graph has direct application for the practical single-threaded nesting case. For the rest of the paper, we will assume only flat transactions, though as explained above, this method can be applied for the flattened, closed and open nesting cases also. However, for these cases, careful bookkeeping is needed during the dynamic construction of the DSR graph to track the nested transactions.

III. Concurrent Checking of Serializability

Concurrent checking of serializability requires the construction and subsequent checking of cycles in the DSR graph. There are two main challenges associated with doing this in practice:

- *Minimizing performance overhead*: The computation overhead necessary for capturing shared accesses and graph checking should minimally affect the transaction throughput performance.
- *Bounding the DSR graph size*: The DSR graph size is $O(N^2)$, where N is the number of dynamic transactions. For a transaction system that runs indefinitely, the number of dynamic transitions is unbounded. Thus, we need to find techniques for bounding the size of the graph and that result in efficient graph construction and compaction in practice.

We address the first challenge by minimizing the additional work done in transaction threads. We limit this to an efficient logging of the following *critical events*: commencement of a transaction, object accesses made by transactions, and termination of a transaction (either commit or abort). The graph construction and checking uses this log in parallel with the transaction threads.

We address the second challenge through an on-the-fly graph compaction algorithm for which the bound on the dynamic graph size is the number of transaction threads. This is critical in enabling this methodology. This algorithm is inspired by ideas used in database schedulers [12].

Figure 4 presents an overview of the components of the concurrent checking. These are:

1) **Access logging**: Logging the critical events in chronological order while the transactions are active. This includes relative timestamping of these events. Section IV discusses this in detail.
2) **Graph construction and compaction**: The compaction of the graph is integrated with the construction and is in parallel with the logging. The thread which modifies the graph is referred to as the *graph thread*.

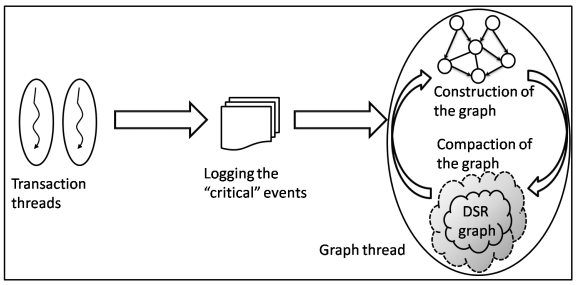


Figure 4. Concurrent Checking of the DSR Graph: An Overview

Section V discusses the algorithm in detail. For now we use a single graph thread and discuss the possibility of using multiple graph threads as part of future work.

In addressing the challenges indicated above this paper makes the following primary contributions:

- A technique for checking serializability in parallel with the transaction system.
- An on-the-fly graph compaction algorithm for which the upper bound on the size of DSR graph is the number of transaction threads.
- Experimental validation of these ideas on a practical STM system (RSTM).
- A characterization of the applicability of this technique (continuous runtime checking/debugging) based on a characterization of the transactions.

IV. Logging Critical Events

This logging takes place within the STM library which supports the transaction book-keeping and is transparent to the application code. Any critical event is logged along with a timestamp since we are interested in the *happens before* relationship between the events. A naive solution is to use a central log shared by all the transaction threads and the graph thread. However, experiments reveal that this suffers from heavy contention between the transaction threads (producers) and the graph thread (consumer). A better match for this producer-consumer relationship is the use of private buffer pairs as described below.

A. Private Buffer Pair

Each transaction thread has a pair of private buffers (queues) where it logs the critical events. Having buffers private to a thread eliminates the contention between the transaction threads which is unnecessary since the critical events are not shared between threads. Further, having a pair of buffers per thread is a good match for the producer-consumer relationship between the transaction thread and the graph thread. The transaction thread fills up one buffer while the graph thread empties the other buffer. When both threads are done, the buffers are swapped between them. This minimizes the contention between the producer and consumer threads. Rather than locking every access to a single shared buffer, the locks are limited to when the buffers need to be swapped. This idea is very straightforward and we are not claiming this as a contribution of this paper.

B. Timestamping Events

As the transaction threads are writing to private buffers, the order of the events across threads needs to be determined for obtaining the *happens before* relationship between the accesses of shared objects. For this purpose, we define a $timestamp(e)$ for the event e with the following properties:

P1: $timestamp(e)$ has a unique value for each critical event happening in the same thread.

P2: An event e_a in thread A accesses an object before another event e_b in thread B iff $timestamp(e_a) < timestamp(e_b)$.

Next, we discuss two different timestamping mechanisms; experimental results of using them are deferred to Section VII.

1) RDTSC: Read Time-Stamp Counter (RDTSC) is an Intel Pentium instruction available in Intel multicore processor used in our experiments [17], with the following properties:

1) It is guaranteed to return a unique value across all cores (except for 64 bit wrap-around which is not likely in 10 years).
2) The returned value is monotonically increasing.

RDTSC can give the 'wall-clock' time for its execution, which can then be used to determine the time for an event. Our experiments show that this instruction is very cheap with the same execution cost as a 'nop' instruction, even when executed in parallel across multiple threads.

However, this instruction timestamps *its own execution time*. We are interested in timestamping the execution time of a critical event. To ensure that the timestamps for critical events obey property **P2** above, we need to make sure that when the critical event is paired with an RDTSC instruction, it is guarded by a lock per object.

2) Lamport's Logical Clock: The other timestamping choice that we explored is Lamport's Logical Clock [18]. This is briefly reviewed here. For each shared object o there is global counter, $g(o)$. For each thread there is a local event counter, l. Consider two threads A and B trying to access object o. Then the timestamp $l(A)$ for thread A accessing o is computed as following.

$$\begin{aligned} l(A) &\leftarrow max(l(A), g(o)) + 1 \\ g(o) &\leftarrow l(A) \end{aligned}$$

The computation of timestamp involves $g(o)$ which is shared. Hence this computation is guarded by a lock per object. For any other critical event not involving access to a shared object simply increment the local counter for the thread.

$$l(A) \leftarrow l(A) + 1$$

This operation is entirely local and hence no locks are required.

V. Graph Compaction

In this section, we discuss the construction and on-the-fly compaction of the DSR graph G. Before presenting the detailed pseudo-code for the algorithm, we informally present the main ideas.

A *committed (aborted) vertex* is a vertex representing a transaction which has already committed (aborted). A cycle of committed vertices in the DSR graph implies non-serializability of the corresponding history. We refer to this cycle as a *committed cycle*.

The algorithm uses the following two observations:

- Consider a vertex u, with zero indegree, which commits in G. Assuming correct operation, after u commits, it will not access any shared variable. Hence, u cannot have an incident edge in the future. Therefore, u cannot be part of a committed cycle in the future. Thus we can safely delete u. If u accesses a shared variable after it commits, then we have detected an error in the implementation.
- Consider a vertex u, with non-zero indegree, that commits in G. Let the parents and children of u in G be $p_1, p_2, \ldots p_k$ and $c_1, c_2, \ldots c_l$ respectively, as illustrated in Figure 5. As before, we know that u cannot have any additional incoming edges in the future. Thus, the set of incoming edges is limited to the current set of incoming edges from $p_1, p_2, \ldots p_k$. Any future cycle that contains u must also contain one of these parents. Thus, u can be deleted from the graph if the connectivity between the parents and children of u is maintained by introducing edges (p_i, c_j), $\forall i \in (1, k)$ and $\forall j \in (1, l)$ in the compacted graph G'. However, some additional bookkeeping is needed. Since u can have additional outgoing edges in the future, these edges need to be captured. This is done as follows. The unique object-id's of the shared objects accessed by Transaction u is denoted as the *access-set* of node u. The access-sets of u are *inherited* by all parents. Thus, in future if some vertex v makes a conflicting access to a variable in the access-set of u, this will result in edges from the parents of u to v to represent the connected paths which would have been there had we not deleted u.

There are two kinds of access-sets associated with a vertex. $reg(w)$ denotes the regular accesses made by the transaction corresponding to vertex w. Let u be a child of w in the DSR graph. When u commits, the access-sets of u are *inherited* by w. The expression $inh(w)$ denotes the set of such *inherited* access-sets of already committed successors of w in the uncompacted graph. An object may be part of $reg(w)$ as well as $inh(w)$. Whenever an active vertex inherits the access-sets of its committing child, the vertex is referred to as a *meta-vertex*.

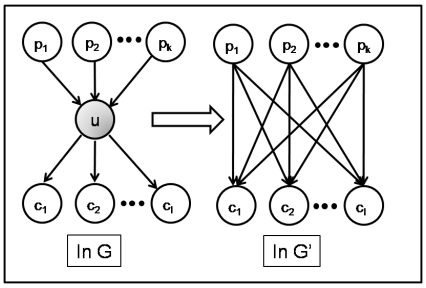


Figure 5. An example of graph compaction. u is a committed vertex.

The DSR graph is altered in response to the following events.

A. Possible Events

Case 1: A new transaction begins. A new vertex corresponding to the transaction is added to G.

Case 2: The transaction corresponding to vertex u has a conflicting access (read/write) to an object o.

for all vertices $p \in G$ ($p \neq u$) such that
$o \in inh(p) \cup reg(p)$ do
add edge (p, u) in G;
if ($o \in inh(u)$) // *self-loop case*
add edge (u, u) in G;

Case 3: The transaction corresponding to vertex u terminates. A transaction can either commit or abort.

Case 3.1: The transaction corresponding to u commits.

Case 3.1.1: In-degree(u) = 0
We follow the *null indegree compaction* part of the **compactGraph** algorithm (Figure 6).

Case 3.1.2: In-degree(u) > 0
We follow the *non-null indegree compaction* part of the **compactGraph** algorithm (Figure 6).

Case 3.2: The transaction corresponding to u aborts.
We follow the *compaction on abort* part of the **compactGraph** algorithm (Figure 6).

B. An Illustrative Example

Let us consider the DSR graph shown in Figure 7. The subscripts of the vertices in the graph are ordered according to their commit order. Each vertex has a pair of access-sets *reg* and *inh*. The access-sets constitute the unique object-id's for the objects accessed by the transaction. The shaded vertices represent the meta-vertices.

When u_4 commits the self-loop is detected and hence the execution is non-serializable. We need to wait until the last member of the cycle commits for reporting the cycle, since if any member of the cycle aborts the cycle disappears.

There are a few interesting observations about the dynamic DSR graph.

1) **Graph size:** When a vertex aborts, it is deleted. When a vertex commits, its object access-sets might get

```
compactGraph(vertex u ∈ G)
 1: {
    // null indegree compaction
 2: If (u is committed and indegree(u) = 0)
 3:     deleteVertex(u); // also deletes out edges

    // non-null indegree compaction
 4: If (u is committed and indegree(u) > 0){
 5:     for each parent p of u{

            // inherit the object access-sets
 6:         inh(p) ← inh(p) ∪ reg(u) ∪ inh(u);
 7:         label p as a meta-vertex

            // connect with the children
 8:         for each child c of u
 9:             addEdge (p, c);
10:     }

        // delete the vertex
11:     deleteVertex (u); // also deletes both in- & out-edges
12: }
    // compaction on abort
13: if(u is aborted)
14:     deleteVertex(u); // also deletes both in- & out-edges
15: }
```

Figure 6. Algorithm for graph compaction

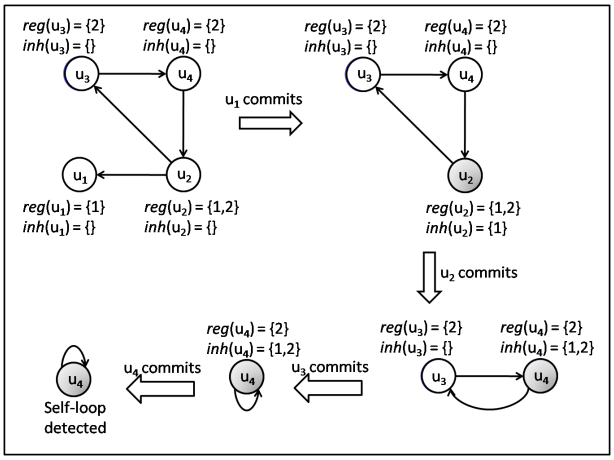


Figure 7. An example of graph compaction

inherited (depending on the in-degree) and the vertex is deleted. Hence, the vertices which are present in the graph are all active. Moreover, at any time at most one transaction can be active in a thread. *Therefore, at any given time, the number of vertices in the DSR graph is bounded by the number of active threads.*

2) **Cycle detection:** As there are only active vertices in the DSR graph, a cycle of committed vertices in the original uncompacted graph will be represented by a *self-loop* of the vertex (which is a member of the cycle in the uncompacted graph too) committing last. *Therefore, in order to detect a cycle, we just need to check every committing vertex for a self-loop.*

C. Cycle Preservation Theorem

We now prove the correctness of the graph compaction algorithm. Specifically we show that the resulting compacted graph is cycle-equivalent to the original graph.

We start by defining the following *Transaction Events* and the corresponding *Graph Events*.

Transaction Events (TE):
- TE_1: A new transaction begins.
- TE_2: A transaction makes a conflicting access to a shared object.
- TE_3: A transaction commits/aborts.

Graph Events (GE):
- GE_1: A new node is added to the graph when a transaction begins.
- GE_2: A new edge is added to the graph when a transaction makes a conflicting access to an object.
- GE_3: $deleteVertex(u)$ when u commits and $indegree(u) = 0$.
- GE_4: $deleteVertex(u)$ when u commits and $indegree(u) > 0$.
- GE_5: $deleteVertex(u)$ when u aborts.

The impact of the TE's on the DSR graph is a combination of one or more of the GE's. TE_1 corresponds to GE_1, TE_2 corresponds to (GE_2)*, while TE_3 corresponds to $(GE_3 + GE_4 + GE_5)$. Here "*" indicates "one or more instances of", and "+" indicates "or". Further, graph events GE_3, GE_4 and GE_5 are triggered by the procedure **compactGraph** only. If the procedure **compactGraph** is never called, nodes or edges are not deleted from the DSR graph (GE_1 and GE_2 never delete any part of the graph).

$$\begin{array}{ccc} G & \xrightarrow{(GE_1+GE_2)*} & G_f \\ \downarrow (GE_3+GE_4+GE_5) & & \\ G' & \xrightarrow{(GE_1+GE_2)*} & G'_f \end{array}$$

Figure 8. At any step, the transformation $G \xrightarrow{(GE_3+GE_4+GE_5)} G'$ is triggered by procedure **compactGraph** and leads to compaction of the graph. The other transformations lead to expansion of the graph.

At any point in time let G be the DSR graph. If the procedure **compactGraph** is not invoked from this point on, G_f is the final DSR graph. The transition $G \rightarrow G_f$ is a result of combinations of the graph events GE_1 and GE_2. Any one of the following graph events: GE_3, GE_4 or GE_5 enables the transition: $G \rightarrow G'$. If the procedure **compactGraph** is not invoked from this point on, G'_f is the final DSR graph. The transition $G' \rightarrow G'_f$ is a result of combinations of the graph events GE_1 and GE_2 (see Figure 8).

The following theorem states that any one application of GE_i's (i = 3,4,5), preserves the cycles in the DSR graph. Observe that each call of procedure **compactGraph** (which is recursive by nature) triggers a combination of graph events: GE_i's (i = 3,4,5). Thus, the transitive application of this theorem proves that there is a cycle in the uncompacted DSR graph iff there is a cycle in the compacted DSR graph.

Cycle Preservation Theorem: There is a committed cycle in G_f iff there is a committed cycle in G'_f.

Proof: Observe that (i) $V(G) \subseteq V(G_f)$ & $E(G) \subseteq E(G_f)$, (ii) $V(G') \subseteq V(G'_f)$ & $E(G') \subseteq E(G'_f)$ (iii) $G \rightarrow G'$ involves deletion of nodes and edges.

We prove the theorem for each of the possible transition choices. We start with the easiest case.

Case 1: $G \xrightarrow{GE_5} G'$, i.e. u gets deleted on its abort.
If u aborts it can never be a vertex in a committed cycle in G_f. Also, $u \notin G'$ as u is deleted by GE_5 and hence it is not part of a committed cycle in G'_f. Clearly, GE_5 never affects any other vertices besides u and any other edges besides those that are incoming or outgoing from u. Thus, it never affects the committed cycles of either G_f or G'_f. Therefore, for the case of the transition, $G \xrightarrow{GE_5} G'$ there exists a cycle in G_f iff there exists a cycle in G'_f.

Case 2: $G \xrightarrow{GE_3} G'$, i.e. u gets deleted when u commits and $indegree(u)$ = 0. If u is committed with null indegree, in correct operation the committed u cannot have any incident edge in the future. (If there is an edge in the future, then we have caught an error and we can stop.) Thus, u cannot be part of a committed cycle in G_f. Moreover, u gets deleted in G', therefore, it cannot be part of a committed cycle in G'_f. Further, deleting u (and any outgoing edges), does not delete any other edges and vertices in the graph. Thus, GE_3 does not affect the committed cycles of either G_f or G'_f. Therefore, for the case of the transition: $G \xrightarrow{GE_3} G'$, there exists a cycle in G_f iff there exists a cycle in G'_f .

Case 3: $G \xrightarrow{GE_4} G'$, i.e. u gets deleted when u commits and indegree(u) > 0.

If part: We consider the following cases.

Case 3.1: u is part of a cycle in G. u must have some parent p and some child c which are also member vertices of the cycle. By the graph event GE_4, edges (p, u) and (u, c) are replaced by the edge $(p, c) \in G'$ (line 14 in procedure **compactGraph**). Therefore, if edges (p, u) and (u, c) complete a cycle in G, (p, c) will complete a cycle in G' too. As neither GE_1 nor GE_2 deletes any edge, if there exists a cycle in G_f, then there also exists a cycle in G'_f.

Case 3.2 (Future inclusion of u in the cycle): u is not a part of a cycle in G but a part of a cycle in G_f. There are two cases here.

Sub-case A: Some conflicting access creates an outward edge from u (say (u, v)) in the future (incident edge is impossible since u is already committed) which makes it part of the cycle in G_f. Also let p (some parent of u in G) and v be part of this cycle in G_f. By GE_4, the object accesses of u ($reg(u) \cup inh(u)$) will be inherited by the all the parents of u in G' (line 6 in procedure **compactGraph**). Therefore, the conflicting edge (p, v) will be present in G'_f and hence completes the cycle.

Sub-case B: Some edge e ($e \notin G$ & $e \in G_f$) not adjacent to u includes u in the cycle. This implies that some parent (p, say) and some child (c, say) of u are also part of the cycle in G_f. We argue similar to Case 3.1, edge (p, c) should be in G'. Therefore, (p, c) is also in G'_f and hence it completes the cycle.

Only-if part: Let p and c be a parent and child of u in G respectively.

Case 3.3: Edge (p, c) is part of the cycle in G'_f. p and c have a conflicting access of object o such that either $o \in reg(p)$ or $o \in inh(p)$. In the former case, the edge (p, c) is also present in G_f thereby completing the cycle. In the later case, object $o \in inh(p)$ is inherited from some committed vertex (say u). This implies that edges (p, u) and (u, c) are present in G_f. If (p, c) completes the cycle in G'_f, edges (p, u) and (u, c) will also complete a cycle in G_f. Therefore, if there is a cycle in G'_f, there is a cycle in G_f too.

Case 3.4: p is part of the cycle but none of the children of u (in G') is part of the cycle in G'_f. Let edge (p, v) ($v \notin children(u)$) be part of the cycle. Therefore, p and v have a conflicting access of object o such that either $o \in reg(p)$ or $o \in inh(p)$. In the former case, the edge (p, v) is also present in G_f. In the later case, object $o \in inh(p)$ is inherited from some committed vertex (say w). Hence, $o \in reg(w)$ or $o \in inh(w)$. Therefore, in G_f, both the edges (p, w) and (w, v) are present completing the cycle.

VI. Implementation Issues

We used the Rochester Software Transactional Memory (RSTM) system for our experiments. We present an overview of the RSTM benchmark code and the backend library followed by a discussion of the engineering design choices made in the implementation of concurrent logging, graph construction and compaction.

A. Overview of RSTM code

We discuss the structure of the RSTM code briefly with an example. `Counter` is a simple RSTM benchmark. A transaction of `Counter` increments the value of a shared counter. Figure 9 gives the transaction code.

```
1: BEGIN_TRANSACTION
2: stm::wr_ptr<Counter> wr(m_counter);
3: wr->set_value(wr->get_value(wr)+1, wr);
4: END_TRANSACTION
```

Figure 9. Transaction code for the RSTM benchmark `Counter`

The transaction boundaries are demarcated by `BEGIN_TRANSACTION` and `END_TRANSACTION`. These are macros which get expanded to lines of code which handle the commit mechanism (or rollback mechanism in case of abort) and other book-keeping tasks. The benchmark requests a pointer to the shared variable `m_counter` in the 'write' mode (line 2). This request is served by the function `OpenReadWrite` in the backend library (function `OpenReadOnly` serves the request for read-only pointers). Function `OpenReadWrite` returns a pointer after certain condition checks and book-keeping operations are done. In our implementation, we instrument these library functions such that just before returning the pointer, the access information is recorded in the log. Access information includes thread-id, transaction-id, object-id, mode of access (read/write) and the timestamp. Moreover, the shared objects are guarded using light-weight locks (`bool_cas` available in the RSTM library).

B. Graph Construction and Compaction

The graph thread reads access information from the private buffer pairs. For the first object access from a transaction, it creates a new vertex and connects edges between the new vertex and other vertices which accessed the object before in a conflicting manner. If a transaction wants to access a shared variable in 'read' mode, the event is logged right before the pointer is returned from the function `OpenReadOnly`. However, in case of 'write' access, we record the event only when the transaction 'commits' and the old version is replaced by the 'cloned' copy (since RSTM follows a deferred update policy). Moreover, the actual 'commit' event is recorded, only after recording all the write accesses of a committing transaction. We do so in order to guarantee that no incident edge can appear once a vertex commits or aborts. For the error free case, the final graph should contain no vertices, i.e. if all the transactions terminate and the execution is serializable. There are three possible error scenarios.

1) The transaction accesses an object after it commits.
2) There is an uncommitted vertex at the end.
3) A committing vertex has a self-loop indicating a serializability violation.

VII. Results

We conducted experiments with the following benchmarks.

- *RSTM benchmark suite* - The benchmark suite of RSTM (release 4) uses the object-based library [6].
- *Synthetic benchmark* - Experiments conducted with a self-constructed synthetic benchmark (using the object-based library) to show how the overhead varies with different factors, such as the number of shared objects and length of activity inside a single transaction. Experiments with RSTM benchmarks and synthetic benchmark were performed with an Intel i7 processor (4 physical and 8 virtual cores, 2.67 GHz, 3 GB memory) running 64-bit Windows Vista Home Premium.
- *STAMP benchmarks* - STAMP benchmarks [7] use one of the word-based libraries (e.g. Extended Timestamp (ET)) in RSTM (release 5). Experiments were performed with the same Intel i7 processor running Ubuntu 2.6.31-14-generic.

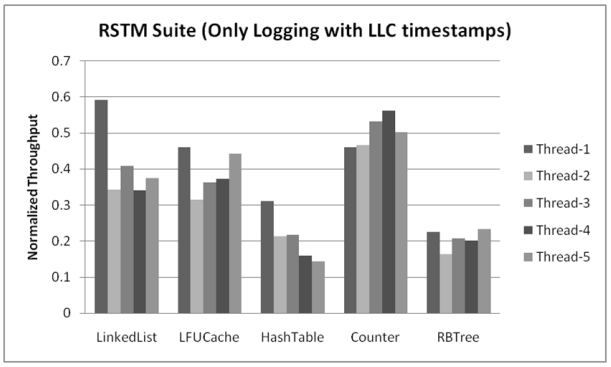
The experiments with the RSTM (object-based) suite and the synthetic benchmark are conducted for 30-60 seconds with 1-5 threads executing transactions. Each of the threads runs on a dedicated virtual core. There is a *single* graph-thread which also runs on a separate virtual core. The experiments were conducted with both timestamping methods discussed in Section IV, *viz.*, Read Timestamp Counter (RDTSC) and Lamport's Logical Clock (LLC).

For the RSTM suite and synthetic benchmark, the user specifies the length of the execution of transactions. Once the execution is over, the software reports the average committed transaction throughput per thread. The following three sets of experiments were run with the RSTM suite and the synthetic benchmark, for 1-5 transaction threads and timestamping mechanisms (RDTSC or LLC).

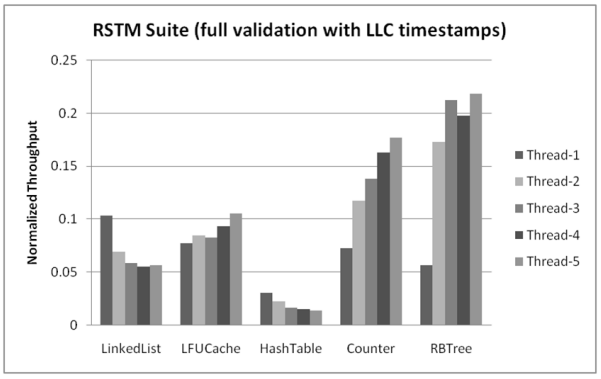
1) **Baseline Experiments:** Experiments with the validation part turned off i.e. normal execution of RSTM. The transaction throughputs in these experiments is referred to as *original throughput*.
2) **Only Logging:** In these experiments, the critical events in the transaction threads are logged but no graph is constructed. In this case, the consumer thread simply drains the filled buffer. We refer to this throughput as *logging-only throughput*. These experiments are performed to see the effect of logging (without graph-checking) on the transaction throughput.
3) **Graph-checking:** The logging of events and the graph-checking take place concurrently in this set of experiments. This throughput is referred to as *full-validation throughput*.

A. Experiments with the RSTM suite

The benchmarks in this suite consist of kernels with very little computation inside a transaction as exemplified by the `Counter` benchmark shown in Figure 9. Figure 10(a) shows the logging-only throughput normalized with respect to the original throughput using LLC timestamping. The average normalized throughput is approximately 0.3 which implies a 3X slowdown of the benchmarks. With concurrent

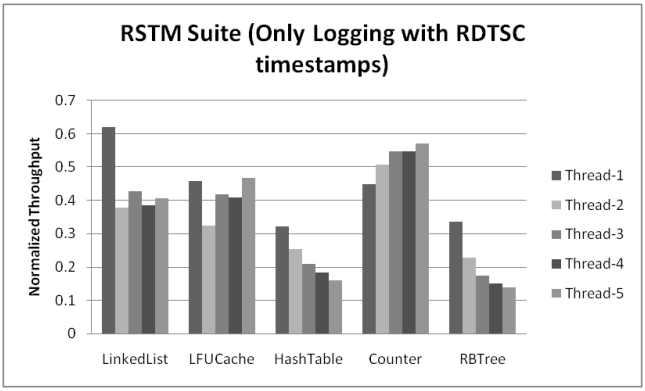


(a) Ratio of logging-only throughput to original throughput

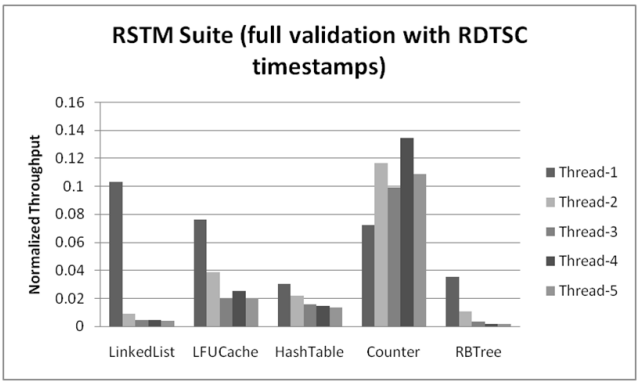


(b) Ratio of full-validation throughput to original throughput

Figure 10. Results of RSTM suite with LLC timestamping



(a) Ratio of logging-only throughput and original throughput



(b) Ratio of full-validation throughput and original throughput

Figure 11. Results of RSTM suite with RDTSC timestamping

graph-checking the average normalized throughput drops to 0.1 (see Figure 10(b)). In comparison, normalized logging-only throughput with RDTSC timestamps (Figure 11(a)) performs slightly better. However, the normalized throughput with concurrent graph-checking suffers a significant degradation with RDTSC timestamps (Figure 11(b)). Thus, LLC is the overall preferred mechanism for timestamping. This was also the observation with the synthetic benchmark.

These graphs omit results for one benchmark - `RandomGraph`. In this case, the overhead is negative, i.e. the validation results in an increase in throughput as a result of a change in the conflict pattern. This anomalous off-scale case is dropped to provide better resolution of the throughput axis.

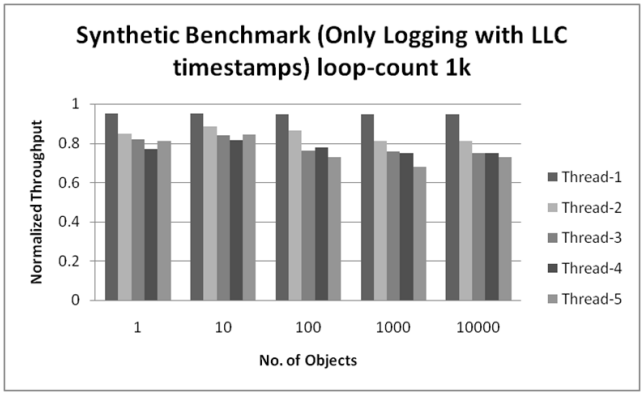
B. Experiments with a Synthetic Benchmark

The synthetic benchmark parameterizes the `Counter` benchmark (in the RSTM suite) in two ways. The number of the variables is a parameter. It randomly chooses a variable out of the pool and increments its value at each step inside a `do-while` loop for a user specified *loop-count* which is the second parameter. Figure 14 presents the pseudocode for the benchmark.

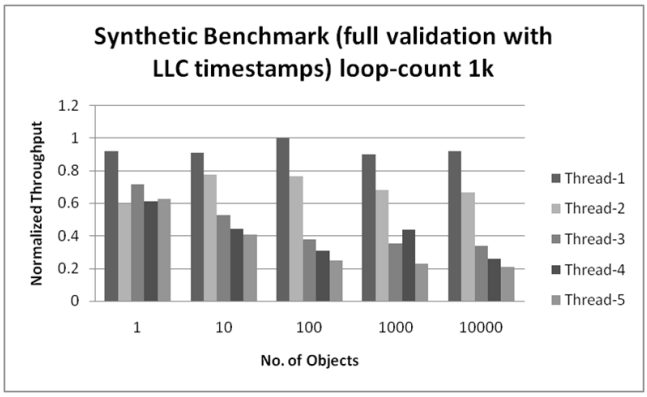
Two sets of experiments were performed with the synthetic benchmark.

Experiment 1: In the first experiment, we were interested in the variation of the normalized throughput with varying number of threads and number of shared objects while the loop-count is fixed at 10^3. Similar experiments (as described before) were performed with the synthetic benchmark employing both timestamping mechanisms. For brevity, the results obtained with LLC timestamping are presented as this had the lower overhead. We observe that the logging overhead increases slightly with increasing number of variables (Figure 12(a)). However, the full validation overhead increases significantly with increasing number of variables (Figure 12(b)). While contention drops when the number of shared objects increases, the access-sets (defined in Section V) of the vertices in the graph are larger which implies more computation for graph-checking.

Experiment 2: Next, we were interested in studying the effect of transaction length on the normalized throughput. In this experiment the number of shared objects varied, but the number of transaction threads was fixed at 5 (Figure 13). We observe that with fewer shared objects (< 10) there is a significant increase in throughput when the loop-count increases from 10^2 to 10^3. However, at the other extreme (when the no. of objects=10^4) this inflection point is at loop-count=$10^3 \rightarrow 10^4$. Therefore, concurrent graph-checking



(a) Ratio of logging-only throughput to original throughput



(b) Ratio of full-validation throughput to original throughput

Figure 12. Results of synthetic benchmark with loop-count=10^3 and LLC timestamping

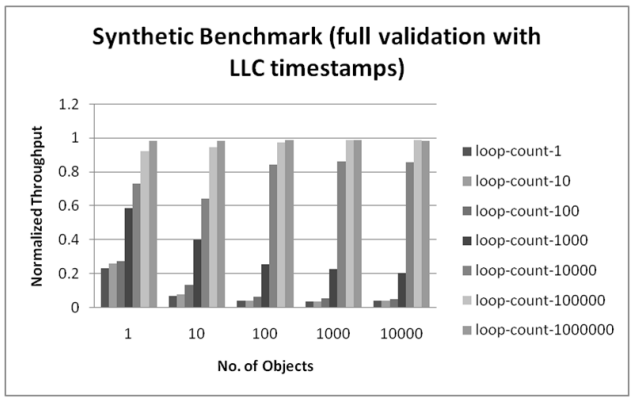


Figure 13. Results of synthetic benchmark with varying loop-count(1-10^6) with 5 transaction threads using LLC timestamps. This chart shows the ratio of full-validation throughput and original throughput.

overhead is low when diminished contention for objects is coupled with prolonged activity within the transaction. In particular, when the transaction length is significant (loop-count=10^6), the logging overhead is negligible compared to transaction activity and the graph thread has no difficulty keeping up with the transaction threads - resulting in negligible overall throughput degradation.

```
1: BEGIN_TRANSACTION
2: obj_id := rand() % no_of_objects;
3: stm::wr_ptr<Synthetic> wr(ObjList[obj_id]);
4: do{
5:  i++;
6:  wr->set_value((wr->get_value(wr)+1), wr);
7: }while(i<loop_count);
8: END_TRANSACTION
```

Figure 14. Pseudocode for the Synthetic Benchmark. `no_of_objects` and `loop_count` are parameters.

C. Experiments with the STAMP benchmark suite

STAMP benchmarks [7] emulate real-world applications with varying degrees of contention and lengths of transaction. Moreover, each benchmark is provided with various input sets, which can be broadly classified into two classes. The input sets which are good for execution either in simulation environment or in low-contention scenarios are denoted as lightweight input sets, while the ones which are better suited for execution either in actual systems or in high-contention scenarios are denoted as heavy-duty input sets. We have experimented with and compared both the cases. In Figure 15, each benchmark is labeled first with its length of transaction (short (S), moderate (M), long (L), very long (VL)); and then with the level of contention (low (L), low to moderate (LM), moderate to high (MH) and high (H)). For instance, benchmark `vacation` (M-LM) has moderately (M) long transactions with low to moderate (LM) contention. Moreover, unlike the RSTM infrastructure which reports average throughput, in this case, the infrastructure reports the validation time. Thus, the columns 2 & 3 in Figure 15 provide the normalized validation time (validation time compared to the baseline with no validation) with LLC timestamping. The results are shown in tabular instead of graphical form due to the greater variation.

Benchmarks	low-contention /simulator	high-contention /non-simulator
STAMP benchmarks (high-overhead)		
ssca2 (S-L)	2.018	62.067
kmeans (S-L)	4.255	317.462
genome (M-L)	25.161	181.804
intruder (M-MH)	10.043	322.019
vacation (M-LM)	93.537	263.336
yada (L-M)	1.79	67.907
STAMP benchmarks (low-overhead)		
bayes (L-H)	1.427	1.477
labyrinth (VL-VH)	0.331	2.505

Figure 15. Normalized validation times for STAMP benchmarks

The benchmarks have been grouped according to their transaction lengths and level of contention. The results with STAMP benchmark reinforce the previous result obtained with synthetic benchmark - the higher the length and contention, the lower the validation time.

D. Inferences

We draw the following inferences from these results.

1) LLC is to be preferred over RDTSC for timestamping.
2) For short transactions the logging and graph-checking overhead is significant. This rules out the application of the current implementation for continuous runtime checking. However, we see an application in debugging. This of course comes with the usual concern related to Heisenbugs [19], bugs which disappear when they are tried and observed, in this case due to change in the concurrent execution due to the checking computation.
3) For long transactions both the logging and graph checking overheads are negligible compared to the transaction execution. This makes it acceptable to use the current implementation for continuous checking.

It is reassuring to report that no cycles, i.e. violations of serializability were detected in any of the examples.

VIII. Related Work

Serializability is a well studied problem in the context of database systems. Bernstein presents a summary of concurrency control and recovery in database systems in his book [12]. Traditionally, database schedulers are entrusted with the responsibility of maintaining the serializability of transactions. These schedulers can be broadly classified into three categories - Two Phase Locking (2PL), Timestamp Ordering (TO) and Serialization Graph Testing (SGT). Our graph compaction technique is inspired by SGT.

Papadimitriou defined and investigated the complexity of verifying whether a given history is serializable [5]. He also showed that the general problem of recognizing a serializable transaction history is NP-complete and that subclasses such as DSR and SSR are efficiently recognizable. Another correctness criterion, *linearizability* was defined by Herlihy & Wing [20]. Linearizability, a special case of strict serializability, assumes that the effect of a method invocation occurs at an instantaneous point somewhere between the invocation and completion of the method.

The rise in popularity of STMs has also attracted interest from the formal verification community. Researchers have attempted to verify serializability and other correctness criteria using well-studied formal techniques such as model-checking [1], [2], [3]. Henzinger et al. showed that an STM that enjoys certain symmetry properties either violates a safety or liveness requirement on some program with 2 threads and 2 shared variables, or satisfies the requirement on all programs [1]. Cohen et al. have developed a formal specification of correctness of TM and a methodology for verifying the TM implementations [2]. In this paper, they have developed specifications for three different kinds of TM. However, they did not publish any experimental results to show the efficacy of their proposed methodology. Researchers at Intel attempted to model-check an industrial transactional memory system (McRT STM from Intel) for checking serializability [3]. The authors were able to verify that McRT STM guarantees the serializable execution of two threads, where each thread is running a transaction consisting of three reads or writes using the Spin model-checker in their experiments. This shows that existing formal tools are unable to handle the large state-space of TM systems. Tasiran proposes formal verification of an implementation of TM [21]. This is an indirect proof that uses a set of sequential assertions to verify the serializability property.

The intractability of the large state-space of STMs have led researchers to explore other alternatives such as runtime approaches. In this direction, Chen et al. [14] proposed a promising solution to verify serializability of hardware transactional memory by piggy-backing the underlying cache-coherence protocol. However, the absence of such an in-built mechanism to resolve conflicting accesses make it a more difficult problem in software.

Our graph construction and compaction relies on the faithful recording of the critical events in chronological order. Replay architectures also use recording for debugging in a multiprocessor environment (due to the inherent non-repeatability of the bugs in this framework), intrusion analysis and fault tolerance e.g. [22], [23], [24]. All of them utilize hardware support for: 1. recording at-speed of data production, 2. minimizing the logging overhead, and 3. replay at a speed similar to that of the initial execution [24]. In our work, recording critical events is done purely in software.

IX. Conclusions and Future Work

In this paper we explore the concurrent checking of STM implementations. We show how concurrent checking can be accomplished through a logging of critical events in the transaction threads accompanied by graph checking of the DSR graph in a parallel thread. A key contribution here is the use of a dynamic graph compaction algorithm that reduces an otherwise unbounded graph to one no larger than the number of threads. As part of the checking we also explore two alternate timestamping mechanisms for logging critical events: the use of a hardware timestamp counter (the RDTSC Intel Pentium instruction), and Lamport's Logical Clock.

We demonstrate a practical implementation of these ideas as part of the RSTM system and study its application to the RSTM benchmarks as well as a parameterized synthetic benchmark. Here we observe that the overhead of the concurrent checking is a strong function of the transaction length. For transactions with significant computation, the logging and graph checking have negligible overhead, making the use of this implementation very practical for continuous runtime checking. For transactions with short computation, the overhead is significant, making this implementation impractical for continuous checking. However, it may be

reasonable to utilize this implementation for debugging. Further, we see potential for making the implementation more efficient and thus increasing the range of instances for which it can be using for continuous checking. In particular, the current implementation uses a single graph thread. We will explore various alternatives for parallelizing this thread in our future work.

ACKNOWLEDGMENT

The authors acknowledge the support of the Gigascale Systems Research Center, one of six research centers funded under the Focus Center Research Program (FCRP), a Semiconductor Research Corporation program. The authors also like to thank the support extended by Luke Dalessandro and Michael L. Scott of the RSTM group at the University of Rochester, Christos Kozyrakis of the STAMP project at the Stanford University and Daniel Schwartz-Narbonne at the Princeton University.

REFERENCES

[1] R. Guerraoui, T. A. Henzinger, B. Jobstmann, and V. Singh, "Model checking transactional memories," in *PLDI '08: Proceedings of the 2008 ACM SIGPLAN conference on Programming language design and implementation*. New York, NY, USA: ACM, 2008, pp. 372–382.

[2] A. Cohen, J. W. O'Leary, A. Pnueli, M. R. Tuttle, and L. D. Zuck, "Verifying correctness of transactional memories," in *FMCAD '07: Proceedings of the 7th International Conference on Formal Methods in Computer-Aided Design (FMCAD)*, November 2007, pp. 37–44.

[3] J. O'Leary, B. Saha, and M. R. Tuttle, "Model checking transactional memory with spin," in *PODC '08: Proceedings of the twenty-seventh ACM symposium on Principles of distributed computing*. New York, NY, USA: ACM, 2008, pp. 424–424.

[4] *International Technology Roadmap for Semiconductors (ITRS) 2007 Edition*, ITRS, 2007.

[5] C. H. Papadimitriou, "The serializability of concurrent database updates," *J. ACM*, vol. 26, no. 4, pp. 631–653, 1979.

[6] V. J. Marathe, M. F. Spear, C. Heriot, A. Acharya, D. Eisenstat, W. N. Scherer III, and M. L. Scott, "Lowering the overhead of software transactional memory," in *ACM SIGPLAN Workshop on Transactional Computing*, Jun 2006.

[7] C. Cao Minh, J. Chung, C. Kozyrakis, and K. Olukotun, "STAMP: Stanford transactional applications for multiprocessing," in *IISWC '08: Proceedings of The IEEE International Symposium on Workload Characterization*, September 2008.

[8] M. Herlihy, V. Luchangco, M. Moir, and W. III, "Software transactional memory for dynamic-sized data structures," in *PODC '03: Principles of Distributed Computing*, Jul 2003, pp. 92–101.

[9] T. Harris and K. Fraser, "Language support for lightweight transactions," in *Object-Oriented Programming, Systems, Languages, and Applications*, Oct 2003, pp. 388–402.

[10] V. J. Marathe, W. N. Scherer III, and M. L. Scott, "Adaptive software transactional memory," in *Proceedings of the 19th International Symposium on Distributed Computing*, Cracow, Poland, Sep 2005.

[11] B. Saha, A.-R. Adl-Tabatabai, R. L. Hudson, C. Cao Minh, and B. Hertzberg, "McRT-STM: a high performance software transactional memory system for a multi-core runtime," in *Proc. 11th ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming (PPoPP '06)*, Mar 2006, pp. 187–197.

[12] P. A. Bernstein, V. Hadzilacos, and N. Goodman, *Concurrency control and recovery in database systems*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1986.

[13] R. Sethi, "Useless actions make a difference: Strict serializability of database updates," *J. ACM*, vol. 29, no. 2, pp. 394–403, 1982.

[14] K. Chen, S. Malik, and P. Patra, "Runtime validation of transactional memory systems," in *International Symposium on Quality Electronic Design*, March 2008.

[15] J. R. Larus and R. Rajwar, *Transactional Memory*. Morgan & Claypool, 2006.

[16] R. F. Resende and A. El Abbadi, "On the serializability theorem for nested transactions," Amsterdam, The Netherlands, The Netherlands, Tech. Rep. 4, 1994.

[17] *Intel 64 and IA-32 Architectures Software Developer's Manual Vol 2B*, Intel(R), http://developer.intel.com/design/processor/manuals/253667.pdf.

[18] L. Lamport, "Time, clocks, and the ordering of events in a distributed system," *Commun. ACM*, vol. 21, no. 7, pp. 558–565, 1978.

[19] G. Neville-Neil, "Kode vicious bugs out," *Queue*, vol. 4, no. 3, pp. 10–12, 2006.

[20] M. P. Herlihy and J. M. Wing, "Linearizability: a correctness condition for concurrent objects," *ACM Trans. Program. Lang. Syst.*, vol. 12, no. 3, pp. 463–492, 1990.

[21] S. Tasiran, "A compositional method for verifying software transactional memory implementations," Microsoft Research, Tech. Rep. MSR-TR-2008-56, Apr 2008, technical Report.

[22] M. Xu, R. Bodik, and M. D. Hill, "A "flight data recorder" for enabling full-system multiprocessor deterministic replay," in *ISCA '03: Proceedings of the 30th annual international symposium on Computer architecture*. New York, NY, USA: ACM, 2003, pp. 122–135.

[23] S. Narayanasamy, C. Pereira, and B. Calder, "Recording shared memory dependencies using strata," *SIGPLAN Not.*, vol. 41, no. 11, pp. 229–240, 2006.

[24] P. Montesinos, L. Ceze, and J. Torrellas, "Delorean: Recording and deterministically replaying shared-memory multiprocessor execution efficiently," in *ISCA '08: Proceedings of the 35th International Symposium on Computer Architecture*. Washington, DC, USA: IEEE Computer Society, 2008, pp. 289–300.