Using Concurrency to Check Concurrency: Checking Serializability in Software Transactional Memory

Arnab Sinha, Sharad Malik, Princeton University

Resilient System Design, Task 1.2.2.3

GSRC Annual Symposium

Sep 3-4, 2009

Abstract

Software transactional memory systems (STM) are very complex. Attempts have been made to formally verify STMs, but these are limited in the scale of systems they can handle and generally verify only a model of the system, and not the actual system.

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. This will be needed anyway given the increasing likelihood of dynamic errors due to particle hits (soft errors) and increasing fragility of nano-scale devices. These errors can only be detected at runtime.

We have implemented concurrent serializability checking in the Rochester Software Transactional Memory (RSTM) system. The overhead of concurrent checking is a strong function of the transaction length. For long (short) transactions this is negligible (significant).


```
init:
a = b = 0;
// Trans T1                 // Trans T2
begin_trans
read a;
                            begin_trans
                            write b:=1;
                            end_trans
read b;
end_trans
```

T1 reads (a, b) = (0, 1).
T2 writes (a, b) = (0, 1).

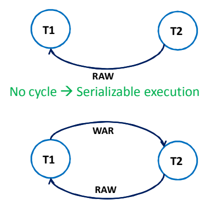
Hence, in effect, T1 follows T2.
Therefore, this execution trace is serializable!

```
init:
a = b = 0;                  // Trans T2
// Trans T1
begin_trans
read a;
                            begin_trans
                            write a:=1;
                            write b:=1;
                            end_trans
read b;
end_trans
```

T1 reads (a, b) = (0, 1).
T2 writes (a, b) = (1, 1).
Case 1: T1 < T2
T1 should have read (0, 0).
Case 2: T2 < T1
T1 should have read (1, 1).
Therefore, this execution trace is not serializable!

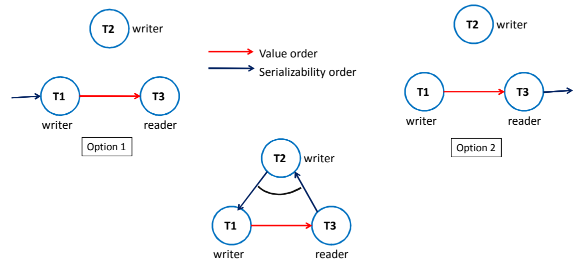
Modeling Serializability as Graph Problem

- Vertices are the transactions.
- Edges represent the conflicting shared data accesses (e.g. RAW, WAR and WAW).
- Edge A → B, if trans. A accesses before trans B.



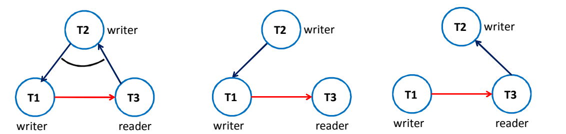
Cycle exists → Therefore, this execution trace is not serializable!

Serializability and DSR (Interchange Serializable [Sethi '82])



"Testing whether a history h is serializable (SR) is NP-complete" – '*Serializability of Concurrent Database Updates*', Christos H. Papadimitriou.

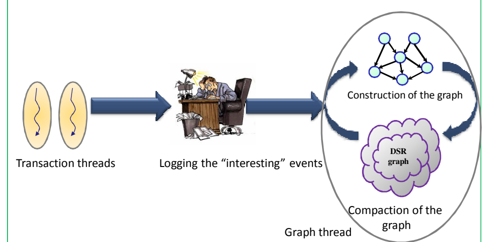
Taking actual time-order, we get DSR.



DSR is efficiently computable!

Challenges and Overview of the work

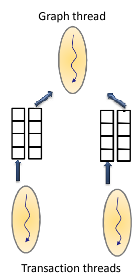
Challenges:
- *Minimizing performance overhead:* Need for efficient validation computation .
- *Bounding the DSR graph size:* The DSR graph size is $O(N^2)$, (N = no. of transactions).



Overview:
- *Access Logging:* Chronological logging of the critical events.
- *Graph construction and compaction:* Logging, construction and compaction of the graph – concurrent operations.

Logging

- Multiple producer (transaction-threads) consumer (graph-thread) relationship.
- Employment of private pair of buffers → reduction of producers' contention.
- The transaction thread fills up one buffer while graph-thread empties the other. When done, buffers are swapped. Thus every access does not need locking.



Timestamping

Requirements:
- Unique value.
- Monotonically increasing value.

Options:
- Read Timestamp Counter (RDTSC).
- Lamport's Logical Clock (LLC).

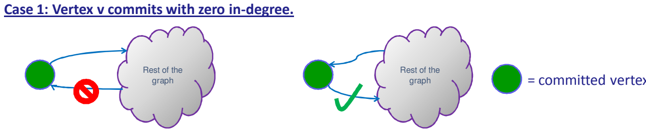
Properties of RDTSC (an Intel Pentium instruction):
- Guaranteed to return a unique value across all cores.
- Monotonically increasing value.

Computation of LLC:
- For each shared object o, there is a global counter $g(o)$. For each thread there is a local event counter l.
- Timestamp $l(A)$ is computed for shared access as follows:

$$l(A) \leftarrow max(l(A), g(o)) + 1$$
$$g(o) \leftarrow l(A)$$

Graph Compaction

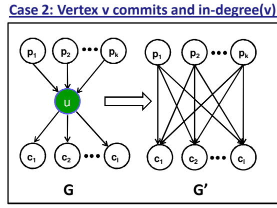
Case 1: Vertex v commits with zero in-degree.



Compaction Rule: Committed vertex with no incident edge can be deleted as it cannot participate in a cycle, in the future.

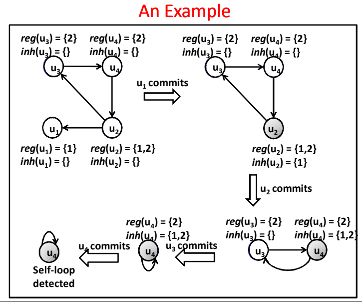
Compaction Strategy:
On vertex commit, if criterion satisfied, delete vertex and its out-edges. Strategy is employed recursively on committed children.

Case 2: Vertex v commits and in-degree(v) > 0.



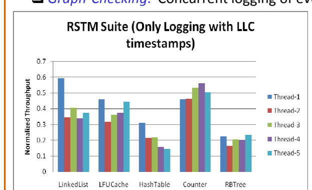
Access-set – Set of unique object-id's accessed.
Access-sets are of two kinds: *reg(u)*, regular accesses and *inh(u)*, inherited accesses

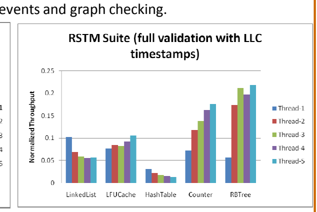
An Example



Results

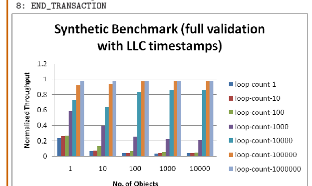
3 sets of experiments:
- *Baseline experiments:* Validation part turned off.
- *Only Logging:* Critical events are logged but no graph.
- *Graph-Checking:* Concurrent logging of events and graph checking.





Code for Synthetic Benchmark:

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



Inferences:
- LLC is better than RDTSC for timestamping.
- For short transactions the overhead is significant. Application is in debugging.
- For long transactions the overhead is minimum. Application is in continuous checking.