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).

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.

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

- Testing whether a history h is serializable (IR) 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.
- Rounding 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.

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.

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 with in-degree(v) > 0.

An Example

Synthetic Benchmark (full validation with LLC timestamps)

Code for Synthetic Benchmark:

- n  = number of vertices
- m  = number of edges
- L   = max size of the graph
- file  = name of the input file
- 4 = 4 processes
- 1 = write process, 2 = read process

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.

Logging

Graph thread

Transaction threads

Timestamping

Requirements:
- Unique value.
- Read Timestamp Counter (RDTSC).
- Guaranteed to return a unique value across all cores.
- Monotonically increasing value.

Computation of LLC

- For each shared object o, there is a local counter gc(o). For each thread there is a local event counter fe(c).
- Timestamp (tA) is computed for shared access as follows:
  \[ t(A) = \max(t(A), g(c)) + 1 \]

Access-set – Set of unique object-id’s accessed.
Access-sets are of two kinds: reg(o), regular accesses and inf(o), inherited accesses.