Verifying Correctness of Transactional Memories

Ariel Cohen[1] John W. O'Leary[2] Amir Pnueli[1]
Mark R. Tuttle[2] Lenore D. Zuck[3]

[1]New York University

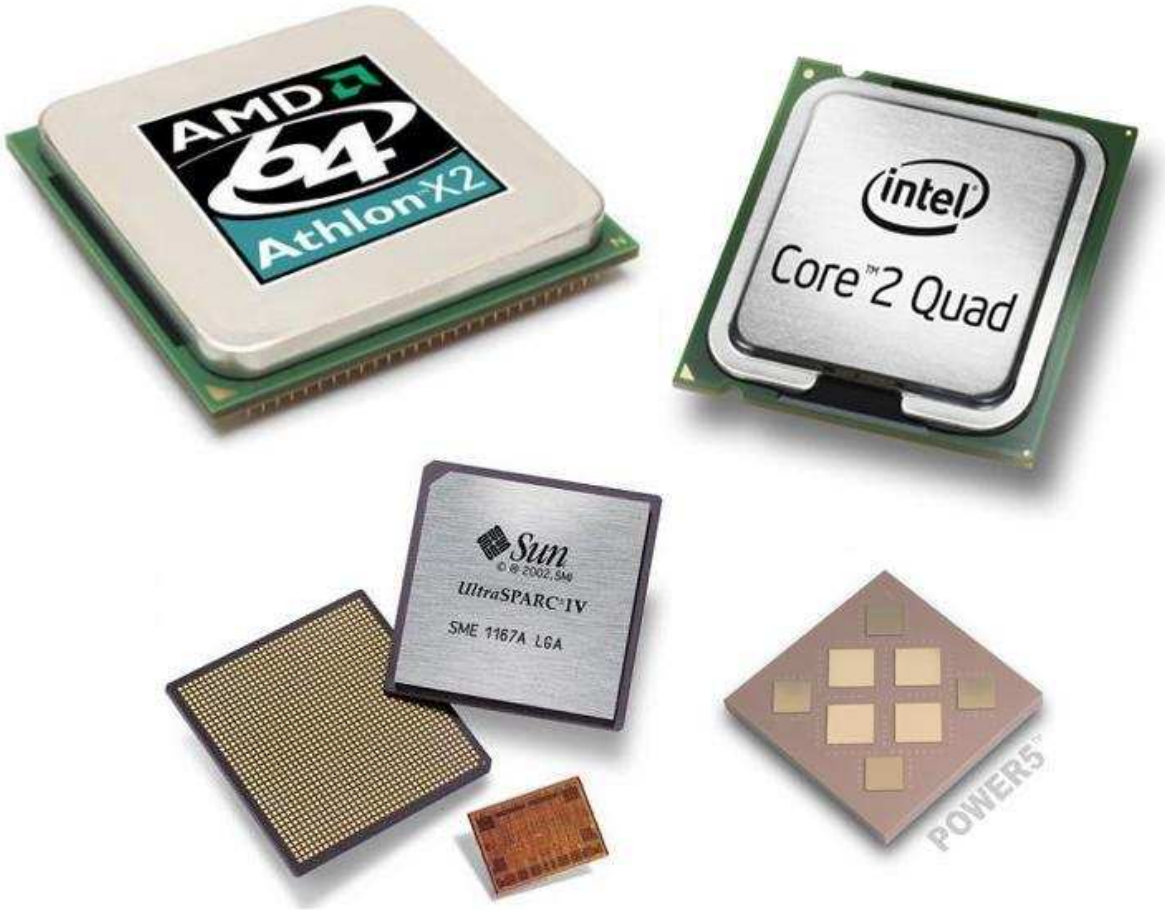
[2]Intel

[3]University of Illinois at Chicago

FMCAD – November 2007

Transactional Memory, why Now?

- Multicore is now a mainstream architecture;



Transactional Memory, why Now?

- Multicore is now a mainstream architecture;
- Concurrent programs are hard to write:
  - locks, semaphores, etc, are difficult to compose;
- TM is a simple(r) solution for coordination and synchronization of threads, that
  - transfers the burden of the concurrency management from the programmers to the system designers;
  - enables programmers to compose scalable applications safely;
- Many processors are now constructed with the goal of offering TM.

Objectives of Research

- What: Define a methodology, supported by tools, to determine when does a TM satisfy its specification;

Objectives of Research

- What: Define a methodology, supported by tools, to determine when does a TM satisfy its specification;
- How: Propose a general model for abstract TM, based on the model of fair discrete systems, and proof rules, based on abstraction mapping, to verify that an implementation of a TM correctly refines its abstract specification;

Objectives of Research

- What: Define a methodology, supported by tools, to determine when does a TM satisfy its specification;
- How: Propose a general model for abstract TM, based on the model of fair discrete systems, and proof rules, based on abstraction mapping, to verify that an implementation of a TM correctly refines its abstract specification;
- Verify implementations using TLA$^+$/TLC;

Transactional Sequences (TS)

A TS (Transaction Sequence) is a sequence of events, each one of the form:

- $\blacktriangleleft_i$ – open a transaction;
- $R_i(x, w)$ – read value w from address x;
- $W_i(x, v)$ – write value v to address x;
- $\blacktriangleright_i$ – commit the transaction;
- $\not\blacktriangleright_i$ – abort the transaction;

where

1. i is a client ID;
2. Each event abbreviates invocation of a request and a non-error response. For example, $R_i(x, w)$ abbreviates $R_i(x)$ request responded by w.

Well-Formed TSs

- Transactions of each client do not intersect: for every i, the projection of the TS on i is a sequence of transactions, each of the form $\blacktriangleleft_i (R_i + W_i)^* \ (\blacktriangleright_i + \not\blacktriangleright_i)$.
- Each transaction satisfies local R/W consistency: if in a given transaction a $W_i(x, v)$ occurs, then every later $R_i(x, w)$ in the *same transaction* is such that $w = v$, unless another $W_i(x, u)$ occurs first.

Atomic and Serializable TSs

A TS is atomic if

- Transactions don't overlap (even for different clients);
- Any $R_i(x, v)$ has the value of the most recent $W_j(x, v)$ in a committed transaction (i.e. in a transaction that ends with $\blacktriangleright$).

Atomic and Serializable TSs

A TS is atomic if

- Transactions don't overlap (even for different clients);
- Any $R_i(x, v)$ has the value of the most recent $W_j(x, v)$ in a committed transaction (i.e. in a transaction that ends with $\blacktriangleright$).

> A TS is serializable if it can be "transformed" into an atomic TS.

- Such transformation is effected by exchanging contiguous events according to specified rules.

Interchanging Events

- Restricting which events in TS may be exchanged, defines
  - correctness conditions;
  - conflicts to be avoided;
- When defining whether two contiguous events e_i and e_j $(j \neq i)$ may be interchanged,
  - consider only events that belong to transactions i and j;
  - consider no future events;
  - require restrictions to be independent of data values;
- Let $\mathcal{A}$ denote the interchange set – pairs of events allowed to be interchanged.

Transforming TS's

- A TS is serializable wrt to $\mathcal{A}$ if, after removing all aborted transactions (transactions ending in $\not\blacktriangleright_i$) it can be transformed into an atomic TS using only interchanges allowed in $\mathcal{A}$.
- Strict Searializability: do not allow $(\blacktriangleright_i, \blacktriangleright_j)$ in the interchange set.

Capturing Conflicts

The interchange set $\mathcal{A}$ can characterize conflicts that should be avoided in a correct behavior.

- Overlap conflict: a conflict arising when one transaction begins before another pending transaction ends. In $\mathcal{A}$ we do not allow $(\blacktriangleleft_i, \blacktriangleright_j)$ or $(\blacktriangleright_i, \blacktriangleleft_j)$.
- Writer Overlap conflict: a conflict arising when two transactions overlap and one writes before the other ends. In $\mathcal{A}$ we do not allow $(W_i, \blacktriangleright_j)$, and also not $(\blacktriangleleft_i, \blacktriangleright_j)$ if there exists W_j.
- Other conflicts of [Scott06] can be similarly defined; however, not all of them.

TMs

An implementation TM consists of two functions:

- A read function that, given a prefix η of a TS, a client id i, and a memory address x, determines which value for $\mathit{read}(\eta, i, x)$ is returned;
- A commit function that, given a prefix η and a client i, determines if $\mathit{commit}(\eta, i)$ may be accepted;

A TS is compatible with a TM if for every event sequence η,

- If $\eta R_i(x, u)$ is a prefix of TS, then $\mathit{read}(\eta, i, x) = u$;
- If $\eta \blacktriangleright_i$ is a prefix of TS, then $\mathit{commit}(\eta, i) = \mathit{True}$;

TMs

An implementation TM consists of two functions:

- A read function that, given a prefix η of a TS, a client id i, and a memory address x, determines which value for $\mathit{read}(\eta, i, x)$ is returned;
- A commit function that, given a prefix η and a client i, determines if $\mathit{commit}(\eta, i)$ may be accepted;

A TS is compatible with a TM if for every event sequence η,

- If $\eta R_i(x, u)$ is a prefix of TS, then $\mathit{read}(\eta, i, x) = u$;
- If $\eta \blacktriangleright_i$ is a prefix of TS, then $\mathit{commit}(\eta, i) = \mathit{True}$;

A TM correctly implements a transactional memory (with respect to $\mathcal{A}$) if every TS that is compatible with it (once aborted transactions are removed) is serializable.

Formal Specification

A Specification Module consists of the following:

- $spec\_mem : \mathbb{N} \longrightarrow \mathbb{N}$ – a persistent memory, init all 0;
- q – a queue of pending events;
- $spec\_out$ – most recent event added to q;
- An interchange set $\mathcal{A}$

Formal Specification

A Specification Module consists of the following:

- *spec_mem*: $\mathbb{N} \longrightarrow \mathbb{N}$ – a persistent memory, init all 0;
- q – a queue of pending events;
- *spec_out* – most recent event added to q;
- An interchange set $\mathcal{A}$

The module can:

- Issue an event and add it to the end of q;
- Remove an aborted transaction from q;
- Interchange consecutive events in q, if $\mathcal{A}$ allows;
- Remove from the front of q *spec_mem-consistent* committed transaction and update *spec_mem* accordingly;

Verification

Given a specification $\mathcal{D}_A$ and an implementation $\mathcal{D}_C$, how to verify that $\mathcal{D}_C$ implements $\mathcal{D}_A$?

Verification

Given a specification $\mathcal{D}_A$ and an implementation $\mathcal{D}_C$, how to verify that $\mathcal{D}_C$ implements $\mathcal{D}_A$?
Find an abstraction relation R between $\mathcal{D}_C$'s and $\mathcal{D}_A$'s states, such that the following all hold:

- Every initial concrete state has an R-related initial abstract state;
- Every concrete transition can be emulated by an abstract transition;
- Every pair of R-related states agree on their observables;
- Abstract fairness requirements hold in any abstract state sequence that is R-related to a concrete computation;

Verification Using TLC

TLC is an explicit state model checker for TLA^+. It requires TLA^+ descriptions of:

- A specification module;
- An implementation module;
- A refinement mapping from the implementation to the specification;

TLC runs the implementation module while using the refinement mapping to map concrete steps into abstract steps, and checks if they are compatible with the specification module.

Example: Lazy Invalidation

- Scott: a conflict occurs when the commitment of one transaction may invalidate a read of the other;
- More formally: if for some transactions T_i and T_j and some memory address x, a sequence that satisfies $R_i(x), W_j(x) \prec \blacktriangleright_j \prec \blacktriangleright_i$, where $e_i \prec e_j$ denote that e_i precedes e_j, occurs.
- Admissible interchange set $\mathcal{A}$: e_i and e_j may be interchanged unless $\exists x, u, v.(W_j(x, u) \in T_j \wedge e_i = R_i(x, v) \wedge e_j = \blacktriangleright_j)$

Example: Trivial Implementation

The implementation module has the following data structures:

- *imp_mem*: $\mathbb{N} \longrightarrow \mathbb{N}$ – a persistent memory, init all 0;
- *pend_trans*: *array of lists* – where *pend_trans*[i] are the events of i's pending transaction;
- *imp_out* – latest occurring event;
- *history_q* – a queue that consists of all the pending transactions' events; It is an auxiliary variable introduced to simplify the proof;

Lazy version management – memory updated at commit;

Lazy conflict detection – conflicts detected at commit;

- In case of a conflict, the committing transaction is aborted;

Example: Refinement Mapping

A refinement mapping is defined from Trivial Implementation to Specification:

- $spec\_mem \leftarrow imp\_mem$;
- $q \leftarrow history\_q$;
- $spec\_out \leftarrow imp\_out$;

Example: Refinement Mapping

A refinement mapping is defined from Trivial Implementation to Specification:

- $spec\_mem \leftarrow imp\_mem$;
- $q \leftarrow history\_q$;
- $spec\_out \leftarrow imp\_out$;

verified, using this refinement:

Trivial Implementation correctly implements Lazy Invalidation.

Bounds of data structures:

- 2 clients;
- At most 4 events in each transaction;
- 2 memory addresses, 3 values;

Additional Implementations Verified

Using TLC we successfully verified other implementations:

- Eager conflict detection and lazy version management – conflicts are checked progressively as transactions read and write data, and the memory is updated only when a transaction is committed (LTM);
- Eager conflict detection and eager version management – conflicts are checked progressively, and the memory is updated immediately when a write event occurs (LogTM);

Accomplishments

- Defined and employed an abstract model for the specification of transactional memory;
- Defined a family of specifications of TMs;
- Showed that by appropriate adaptation of $\mathcal{A}$ we can capture conflicts that are mentioned in the literature (e.g. Scott's);
- Deductively verified some simple implementations;
- Successfully verified, using TLC, some standard implementations appearing in the literature (TCC, LTM, LogTM);

Future Work

- Prove liveness properties
  - if a client closes the same transaction infinitely many times, then it is committed infinitely many times;
  - (provided someone suggests an implementation that satisfies such properties...)
- Verify using a theorem prover;
- Prove more complex implementations:
  - memory access outside transactions;
  - nested transactions;

Reference

- [Lamport, 99] showed how to specify concurrent systems with TLA$^+$;
- [Scott, 06] offered a sequential specifications that embody conflict functions;
- [Herlihy and Moss, 93] proposed the first transactional memory;
- [Shavit and Touitou, 95] presented the first software-only transactional memory (STM);
- [Hammond et al., 04] proposed the model TCC (transactional memory coherence and consistency);
- [Ananian et al., 05] described UTM (unbounded transactional memory) and LTM;
- [Moore et al., 06] proposed LogTM– a log-based transactional memory;