Verification of Transactional Memories that support Non-Transactional Memory Accesses

Ariel Cohen [1], Amir Pnueli [1], and Lenore Zuck [2]

[1]New York University

[2]University of Illinois at Chicago

TRANSACT – February 2008

Objectives

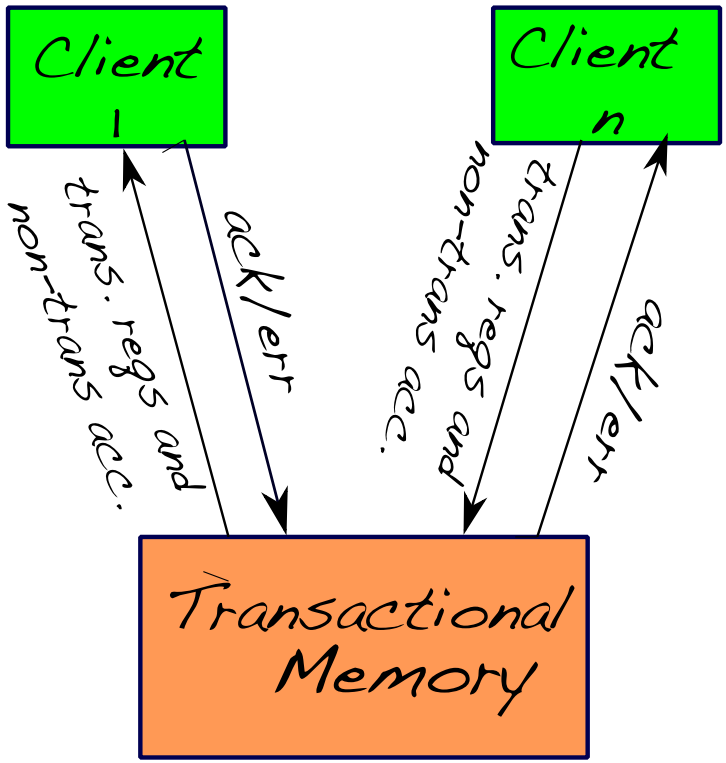
- A formal framework to reason about the correctness of TM implementations; Construct automatic tools to obtain formal verification of implementations (wrt to specifications)

Objectives

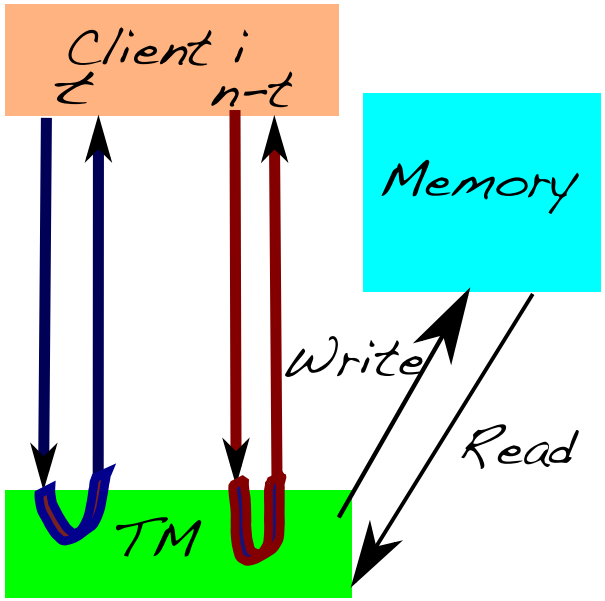
- A formal framework to reason about the correctness of TM implementations; Construct automatic tools to obtain formal verification of implementations (wrt to specifications)
- Construct generic specifications that depend on conflicts, their detection, and their arbitration. Construct a *deductive* proof rule that establishes that an implementations *refines* a specification. Prove soundness of proof rule, and obtain proofs, using TLPVS.

General Overview of TM

- *Clients* issue transactional and *non-transactional* requests;
- *TM* responds with *ack* (or values in the case of reads) or *err*;
- Non-transactional accesses never get *err*
- A *TS* is the sequence of *transactional* events (requests/responses) in the system;



Clients



- *Transactional* issues are $\blacktriangleleft_i, R_i(x), W_i(x, v), \blacktriangleright_i, \not\blacktriangleright_i$;
- Responses are *err*, *ack*, or value v;
- *Non-Transactional* issues are $R^n(x), W^n(x, v)$;
- Same client has at most a single transaction or a single non-transaction at a time. All transactions must end (commit/abort);
- Response is v (for a $R^n(x)$);

Notes on Transactions

- Inside a transaction, all $R_i(x)$ return the most recent value written by *same* transactions if it contains a prior $W_i(x, -)$;
- The only "interesting" responses (that cannot be locally determined) are:
  - Response for the first $R_i(x)$, which is a value;
  - Response to $\blacktriangleright$, which may be *ack* or *err*;
- We assume the req/res pair are consecutive;
- We omit the response when clear from context, and denote $R_i(x)$ with response v by $R_i(x, v)$;

Atomic and Serializable TSs

- A TS is *atomic* if after all events of aborted transactions are removed:
  - Transactions do not overlap;
  - Each (first) $R_i(x)$ returns the value of the most recent $W_j(x, -)$;
- A TS is *serializable* if, after all events of aborted transactions are removed, a series of *interchanges* of consecutive events results in an atomic TS.

Conflicts: Restrictions on Interchanges

Two consecutive events should *not* be interchanged when:

- They belong to the same transactions;
- Both are ▶ events (*strict serializability*);
- Their interchange will *resolve* a conflict;

Conflicts: Restrictions on Interchanges

Two consecutive events should *not* be interchanged when:

- They belong to the same transactions;
- Both are $\blacktriangleright$ events (*strict serializability*);
- Their interchange will *resolve* a conflict;

Example: An *overlap* conflict does not allow transactions to overlap.

$$\overbrace{\blacktriangleleft_1\ R_1(x,0)\ \underbrace{\blacktriangleleft_2\ R_2(x,0)\ W_2(x,1)\ \blacktriangleright_2}_{T_2}\ W_1(x,2)\ \blacktriangleright_1}^{T_1}$$

$$\blacktriangleleft_2\ R_2(x,0)\ W_2(x,1)\ \blacktriangleleft_1\ \blacktriangleright_2\ W_1(x,2)\ \blacktriangleright_1$$

$$\blacktriangleleft_2\ R_2(x,0)\ W_2(x,1)\ \blacktriangleright_2\ \blacktriangleleft_1\ W_1(x,2)\ \blacktriangleright_1$$

Do not allow $\blacktriangleleft_i\ \blacktriangleright_j$ interchanges!!

Admissible Interchanges

- Depend on definition of *conflict*;
- Describe which pairs of consecutive events can be *safely* interchanged;
- Expressible by temporal logic using *past operators*
- Can capture almost all [Sco06]'s conflicts (but for mixed invalidation, which requires future operators);

A TS is *serializable wrt to a conflict* c if, once events belonging to aborted transactions are removed, a sequence of interchanges that is allowed according to the admissible interchanges of c results in an atomic TS.

Specifying/Implementing TMs

Assume a conflict described by a set of *admissible interchanges*.

- A *Specification* for a TM, is a state machine that outputs *all* the TSs that are serializable wrt to the admissible interchanges. There are *fairness* requirements that capture the restriction that every transactions commits or aborts.

Specifying/Implementing TMs

Assume a conflict described by a set of *admissible interchanges*.

- A *Specification* for a TM, is a state machine that outputs *all* the TSs that are serializable wrt to the admissible interchanges. There are *fairness* requirements that capture the restriction that every transactions commits or aborts.
- Conflicts *detection* and *arbitration* is left for the implementation (though they *can* be added to the specifications).

Specifying/Implementing TMs

Assume a conflict described by a set of *admissible interchanges*.

- A *Specification* for a TM, is a state machine that outputs *all* the TSs that are serializable wrt to the admissible interchanges. There are *fairness* requirements that capture the restriction that every transactions commits or aborts.
- Conflicts *detection* and *arbitration* is left for the implementation (though they *can* be added to the specifications).
- An *implementation* of a TM is a state machine that outputs TSs. The implementation is *correct wrt to a specification* if, given some mapping between the variables of both systems, every computation of the implementation can be mapped to a computation of the implementation (preserving fairness).

Specification: Data Structures

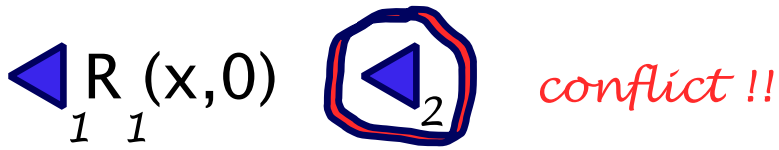
- $spec\_mem : \mathbb{N} \mapsto \mathbb{N}$ – a persistent memory, initially 0;
- $\mathbb{Q}$ – a "queue" of requests of pending transactions;
- $spec\_out$ – a req/res pair or $\perp$;
- $doomed : [1..n] \mapsto \{0, 1\}$ – $doomed[i]$ indicates that the transaction of client i is doomed to be aborted;

Specification: Actions

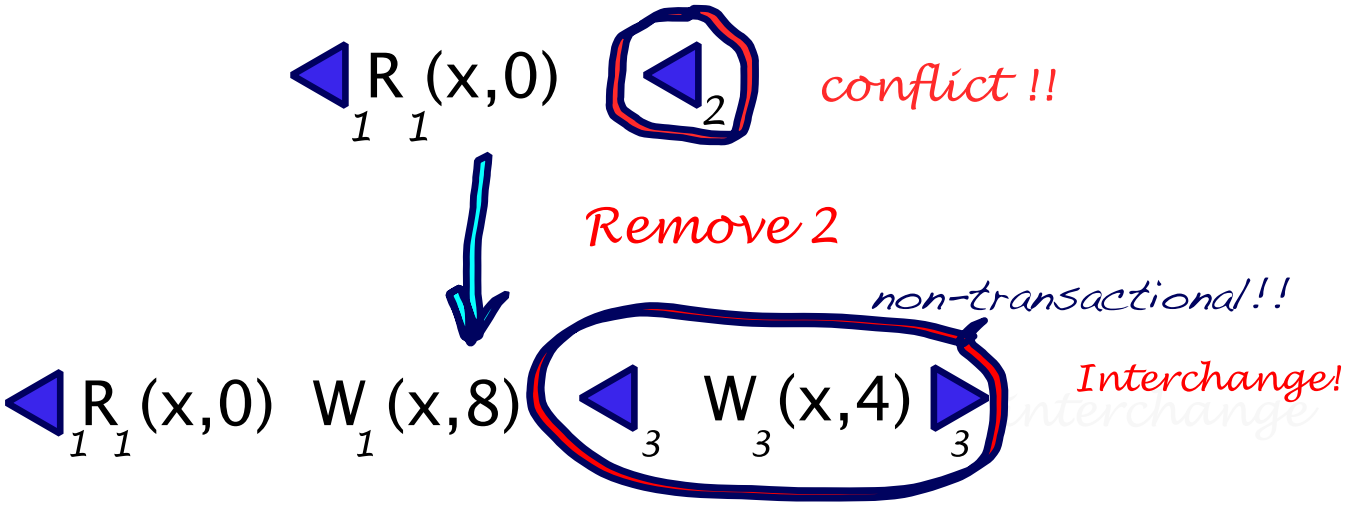
(assuming local correctness. $spec\_out$ in [] when not obvious):

- Append a $\blacktriangleleft_i, R_i, W_i$ and its response to $\mathbb{Q}$;
- Append $\blacktriangleleft_i\ R_i(x, spec\_mem[x])\ \blacktriangleright_i$ to $\mathbb{Q}$;
- Append $\blacktriangleleft_i\ W_i(x, v)\ \blacktriangleright_i$ to $\mathbb{Q}$;
- Append $\not\blacktriangleright_i$ to $\mathbb{Q}$ and remove i's pending transactional events from $\mathbb{Q}$;
- Set $doomed[i]$ to 1 and remove i's pending transactional events from $\mathbb{Q}$ $[\perp]$;
- If $doomed[i]$, set $spec\_out$ to $R_i(x, v)$ or $W_i(x, v)$;
- Perform any admissible interchange on $\mathbb{Q}$ $[\perp]$;
- Append $\triangleright_i$ ("almost commit") to $\mathbb{Q}$ $[\perp]$;
- If i's pending transaction is consistent and in the head of the queue, remove it and update $spec\_mem$ $[\blacktriangleright_i]$

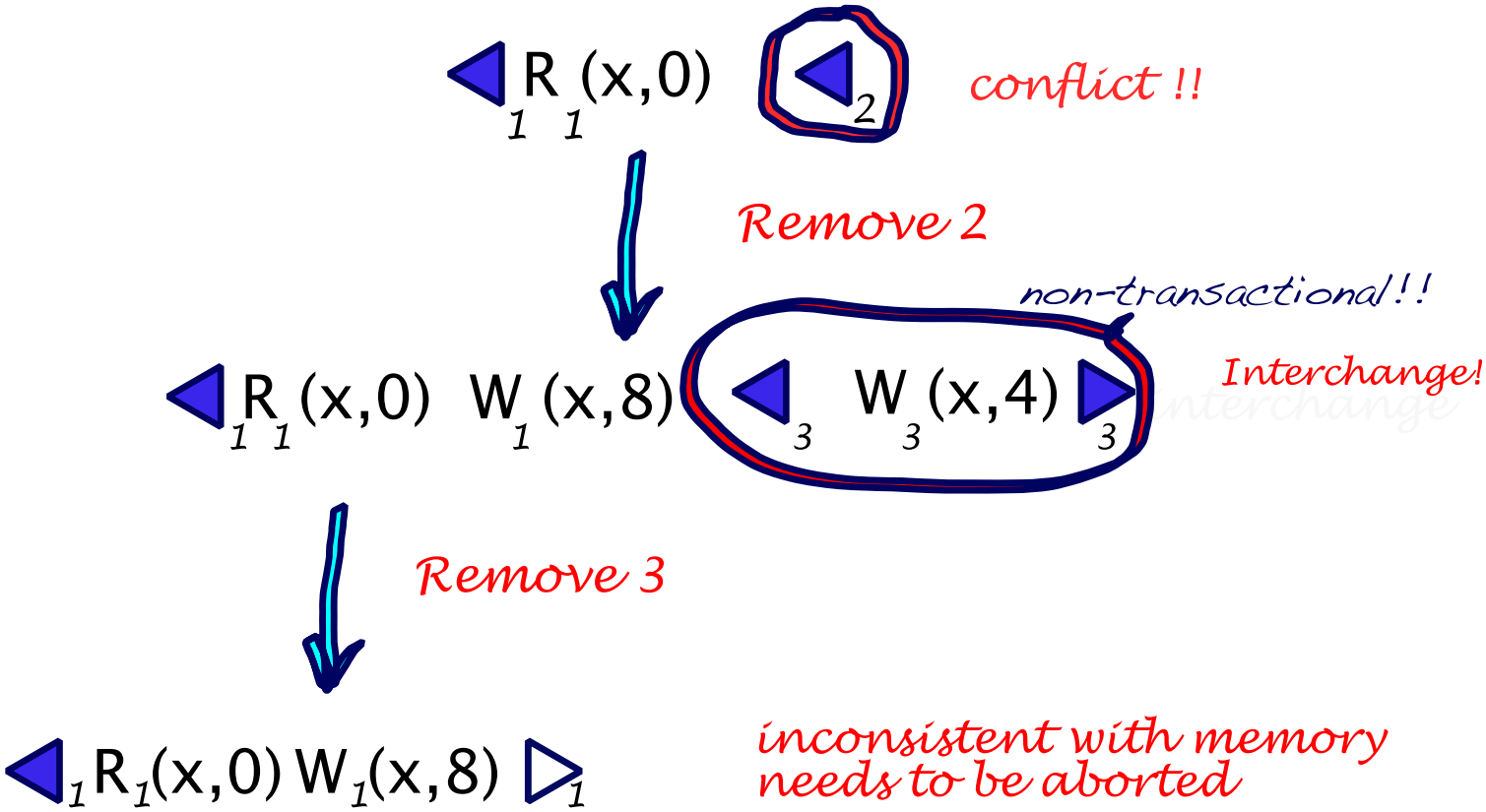
Example



Example


R 1 1 (x,0)
2
conflict !!
Remove 2
non-transactional!!
R 1 1 (x,0)
W 1 (x,8)
W 3 3 (x,4)
3
Interchange!

Example


R 1 (x,0)
1
2
conflict !!
Remove 2
non-transactional!!
R 1 (x,0)
W 1 (x,8)
W 3 (x,4)
3
3
3
Interchange!
Remove 3
R 1 (x,0) W 1 (x,8)
1
1
inconsistent with memory
needs to be aborted

Proving Refinement

- Both *specification* and *implementation* are given as (fair) state machines;
- They both have an *output* that displays the most recent *(req, res)* pair (or $\perp$ is there is none). The sequence of non-$\perp$ values should be equal;
- Each step of the implementation is simulated by one or more step of the specifications;
- Each *fair* computation of the implementation should map to a fair computation of the specifications;

Proving Refinement

- Both *specification* and *implementation* are given as (fair) state machines;
- They both have an *output* that displays the most recent *(req, res)* pair (or $\perp$ is there is none). The sequence of non-$\perp$ values should be equal;
- Each step of the implementation is simulated by one or more step of the specifications;
- Each *fair* computation of the implementation should map to a fair computation of the specifications;

The proof rule calls for a refinement mapping R between states of the two systems, and proves fair simulation between the two systems (similar to [AL])

Proving Refinement

- Both *specification* and *implementation* are given as (fair) state machines;
- They both have an *output* that displays the most recent *(req, res)* pair (or $\perp$ is there is none). The sequence of non-$\perp$ values should be equal;
- Each step of the implementation is simulated by one or more step of the specifications;
- Each *fair* computation of the implementation should map to a fair computation of the specifications;

The proof rule calls for a refinement mapping R between states of the two systems, and proves fair simulation between the two systems (similar to [AL])
Soundness proven by TLPVS

Example: TCC+

TCC can be represented as using the following data structures:

- $imp\_mem : \mathbb{N} \mapsto \mathbb{N}$ – a persistent memory 0 (similar to *spec_mem*);
- $cache[1] \ldots cache[n]$ – each $cache[i]$ stores the pending transaction of client i;
- *imp_out* – a req/res pair or $\perp$;
- $doomed\_mem : [1..n] \mapsto \{0, 1\}$ – Similar to *doomed*;

Example: TCC+ (cont)

TCC's actions are:

- Req/res pair belonging to pending transactions are stored in the appropriate *cache*'s, and commits/abort empty the corresponding cache;
- Each $R(x)$ is responded with $imp\_mem[x]$;
- When a transaction commits, all pending transactions that have reads intersecting with the writes of the committed transaction, are doomed to abort (*lazy invalidation*);
- Each *non-transactional* write causes all pending transactions with reads to the same memory location to be doomed to abort;

Applying the Rule on TCC+

The *admissible interchanges* forbid, in addition to the usual pairs, to interchange pairs of the form $R_i(x) \blacktriangleright_j$ if j's pending transaction includes some $W_j(x, -)$;

Applying the Rule on TCC+

The *admissible interchanges* forbid, in addition to the usual pairs, to interchange pairs of the form $R_i(x) \blacktriangleright_j$ if j's pending transaction includes some $W_j(x, -)$;

The refinement mapping consists of:

- $spec\_mem = imp\_mem$ (equality of the stable memories);
- $spec\_out = imp\_out$ (equality of the output; note that it is not required to hold at every step);
- $doomed = doomed\_mem$;
- For every client i such that $\neg doomed\_mem[i]$ (i is not doomed to abort), the projection of $\mathbb{Q}$ on i's events is exactly $cache[i]$;

Applying the Rule on TCC+

The *admissible interchanges* forbid, in addition to the usual pairs, to interchange pairs of the form $R_i(x) \blacktriangleright_j$ if j's pending transaction includes some $W_j(x, -)$;

The refinement mapping consists of:

- $spec\_mem = imp\_mem$ (equality of the stable memories);
- $spec\_out = imp\_out$ (equality of the output; note that it is not required to hold at every step);
- $doomed = doomed\_mem$;
- For every client i such that $\neg doomed\_mem[i]$ (i is not doomed to abort), the projection of $\mathbb{Q}$ on i's events is exactly $cache[i]$;

The *fairness* requirements of TCC+ are that each $cache[i]$ is eventually emptied (note: unlike our specifications, when a transaction is doomed its subsequent events do get recorded in its $cache$)

Applying the Rule on TCC+

The *admissible interchanges* forbid, in addition to the usual pairs, to interchange pairs of the form $R_i(x) \blacktriangleright_j$ if j's pending transaction includes some $W_j(x, -)$;

The refinement mapping consists of:

- $spec\_mem = imp\_mem$ (equality of the stable memories);
- $spec\_out = imp\_out$ (equality of the output; note that it is not required to hold at every step);
- $doomed = doomed\_mem$;
- For every client i such that $\neg doomed\_mem[i]$ (i is not doomed to abort), the projection of $\mathbb{Q}$ on i's events is exactly $cache[i]$;

The *fairness* requirements of TCC+ are that each $cache[i]$ is eventually emptied (note: unlike our specifications, when a transaction is doomed its subsequent events do get recorded in its *cache*)

(Mapping established in TLPVS)

Future Work

- Proving *liveness* properties
- Finding examples of systems that have both transactional and non-transactional accesses (and verifying them using technique)
- Handling nested transactions
- Handling non-transactional accesses when they are not easily recognizable as such