Decomposing Opacity

Mohsen Lesani Jens Palsberg

University of California, Los Angeles

Correctness of Transactional Memory

- Programming Model
- Correctness Criteria
- Algorithm Design
- Subtle Bugs
- Verification

Decomposition of Correctness

Is there a decomposition of the correctness condition to a conjunction of simpler and meaningful conditions?

- Understanding
- Algorithm Design
- Verification
- Studying Complexity

This Work

- Markability: a decomposition of Opacity to three separate invariants. It is proved that markability is required and sufficient for Opacity.
- Proof of Markability and hence Opacity of TL2
- Proof of a lower bound on the time complexity of TM

Markability

A transaction history is markable if and only if there exists a **marking** of it that is **write-observant**, **read-preserving**, and **real-time-preserving**.

Marking

A marking of a transaction history is a relation on the union of the **transactions** and the **read operations** in the history.

- **The effect order**:

  A total order of the transactions.

  It represents the order in which the transactions appear to take effect.

- **The access orders**:

  Writers of i: The committed transactions that have write operation(s) to location i.

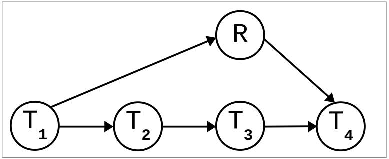
  Consider an unaborted read operation R on a location i.

  For each such R, the access order is an antisymmetric relation that orders R and every writer of i.

  The access order of R represents where the read access by R happens between the write accesses by the writers.

Marking

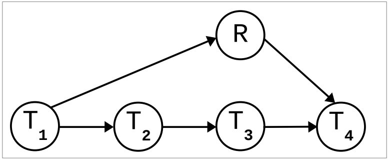
T_1	T_2	T_3	T_4
$write(i, v_1)$			
	$write(i, v_2)$		
$commit(): C$			
		$read(i): v_1$	
	$commit(): A$		
			$write(i, v_4)$
		$commit(): A$	
			$commit(): C$



Write-observation

At a high level, write-observation means that each read operation should read the most current value.

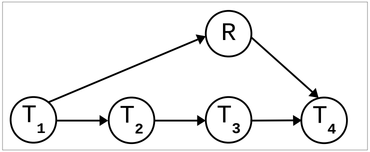
Consider an unaborted read operation R from a location i.
Pre-accessors: the writers of i that come before R in the access order for R.
Last pre-accessor: the pre-accessor that is greatest in the effect order.
Write-observation requires that the value that R reads be the same as the value written by the last pre-accessor.

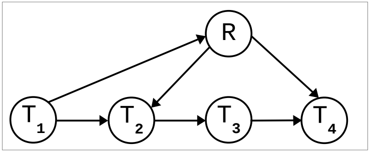


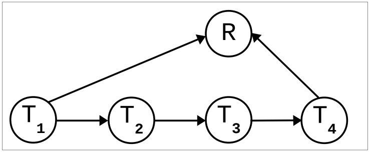
Read-preservation

At a high level, read-preservation means that the location read by a read operation is not overwritten between the points that the read takes place and the transaction takes effect.

Consider an unaborted read operation R by a transaction T from a location i.
Read-preservation requires that no writer of i comes between R and T in the marking relation.







Real-time-preservation

The real-time-preservation condition requires that if all the events of a transaction T happen before all the events of another transaction T', then T is less than T' in the effect order.

Marking Theorem

Theorem

Opacity = *Markability*

See Animation

Marking TL2

D:

def $init_t()$

$I01 \triangleright$ $snap = clock.read()$,

$I02 \triangleright$ $rver[t].write(snap)$,

$I03 \triangleright$ **return** ok,

def $read_t(i)$

$R01 \triangleright$ $pv = wset[t].get(i)$,

if $(pv \neq \bot)$

$R02 \triangleright$ **return** pv,

$R03 \triangleright$ $s_1 = ver[i].read()$,

$R04 \triangleright$ $v = reg[i].read()$,

$R05 \triangleright$ $l = lock[i].read()$,

$R06 \triangleright$ $s_2 = ver[i].read()$,

$R07 \triangleright$ $sver = rver[t].read()$,

if $(\neg(\neg l \ \wedge \ s_1 = s_2 \ \wedge \ s_2 \leq sver))$

$R08 \triangleright$ **return** $\mathbb{A}$,

$R09 \triangleright$ $rver[t].add(i)$,

$R10 \triangleright$ **return** v,

$\{R03 \rightarrow R04,\ R04 \rightarrow R05,\ R05 \rightarrow R06\}$,

def $write_t(i, v)$

$W01 \triangleright$ $wset[t].put(i, v)$,

$W02 \triangleright$ **return** ok,

def $commit_t()$

$C01 \triangleright$ **foreach** $(i \in wset[t])$

$C02 \triangleright$ $locked = lock[i].trylock()$,

if $(\neg locked)$

$C03 \triangleright$ $lset.add(i)$

else

$C04 \triangleright$ **foreach** $(i \in lset)$

$C05 \triangleright$ $lock[i].unlock()$,

$C06 \triangleright$ **return** $\mathbb{A}$,

$C07 \triangleright$ $wver = clock.iaf()$,

$C08 \triangleright$ $sver = rver[t].read()$,

if $(wver \neq sver + 1)$

$C09 \triangleright$ **foreach** $(i \in rset[t])$

$C10 \triangleright$ $l = lock[i].read()$,

$C11 \triangleright$ $s = ver[i].read()$,

if $(\neg(\neg l \ \wedge \ s \leq sver))$

$C12 \triangleright$ **foreach** $(i \in lset)$

$C13 \triangleright$ $lock[i].unlock()$,

$C14 \triangleright$ **return** $\mathbb{A}$,

$C15 \triangleright$ **foreach** $((i, v) \in wset[t])$

$C16 \triangleright$ $reg[i].write(v)$,

$C17 \triangleright$ $ver[i].write(wver)$,

$C18 \triangleright$ $lock[i].unlock()$,

$C19 \triangleright$ **return** $\mathbb{C}$,

Conclusion

- Markability correctness criterion as a conjunction of separate invariants
- Markability as a verification technique and markability of TL2

Lower bound

A TM algorithm is (weakly) progressive if and only if it forcefully aborts a transaction only when it conflicts with a live transaction.
A TM algorithm is (strictly) disjoint-access-parallel if and only if two transactions contend on a base object only if they access a common memory location.
A TM algorithm is invisible-reads if and only if the read operation does not mutate (i.e. change the state of) any base object.

Theorem

The time complexity of the commit operation of every opaque, progressive, disjoint-access-parallel and invisible-reads TM algorithm is $\Omega(|R|)$ where R is the read set.

Thanks for your attendance