SAPIENZA
UNIVERSITÀ DI ROMA
DIPARTIMENTO DI INFORMATICA

CONCURRENT SYSTEMS
LECTURE 5

Prof. Daniele Gorla

Software Transactional Memory

- Group together parts of the code that must look like atomic, in a way that is transparent, scalable and easy-to-use for the programmer
- Differently from monitors, the part of the code to group is not part of the definition of the objects, but is application dependent
- Differently form transactions in databases, the code can be any code, not just queries on the DB

Transaction: an atomic unit of computation (look like instantaneous and without overlap with any other transaction), that can access atomic objects.

→ *Assumption*: when executed alone, every transaction successfully terminates.

Program: set of sequential processes, each alternating transactional and non-transactional code (that both access base objects)

STM system: online algorithm that has to ensure the atomic execution of the transactional code of the program.

SAPIENZA
UNIVERSITÀ DI ROMA
DIPARTIMENTO DI INFORMATICA

Software Transactional Memory

To guarantee efficiency, several transactions can be executed simultaneously (the so called *optimistic execution* approach), but then they must be totally ordered

→ not always possible (e.g., when there are different accesses to the same obj, with at least one of them that changes it)

→ commit/abort transactions at their completion point (or even before)

→ in case of abort, either try to re-execute or notify the invoking proc.

→ possibility of unbounded delay

Conceptually, a transaction is composed of 3 parts:

[READ of atomic reg's] [local comput.] [WRITE into shared memory]

The key issue is ensuring consistency of the shared memory

→ as soon as some inconsistency is discovered, the transaction is aborted

Implementation: every transition uses a local working space

- For every shared register: the first READ copies the value of the reg. in the local copy; successive READs will then read from the local copy
- Every WRITE modifies the local copy and puts the final value in the shared memory only at the end of the transaction (if it has not been aborted)

4 operations:

* $\texttt{begin}_T\texttt{()}$: initializes the local control variables

* $\texttt{X.read}_T\texttt{()}$, $\texttt{X.write}_T\texttt{()}$: as described above

* $\texttt{try\_to\_commit}_T\texttt{()}$: decides whether a non-aborted trans. can commit

A Logical Clock based STM system

SAPIENZA
UNIVERSITÀ DI ROMA
DIPARTIMENTO DI INFORMATICA

Let T be a transaction; its *read prefix* is formed by all its successful READ before its possible abortion.

An execution is **opaque** if all committed transactions and all the read prefixes of all aborted transactions appear if executed one after the other, by following their real-time occurrence order.

We now present an atomic STM system, called *Transactional Locking 2* (TL2, 2006):

- CLOCK is an atomic READ/FETCH&ADD register initialized at 0
- Every MRMW register X is implemented by a pair of registers XX s.t.
  - XX.val contains the value of X
  - XX.date contains the date (in terms of CLOCK) of its last update
  - It is associated with a lock object (to guarantee MUTEX when updating the shared memory)
- For every transaction T, the invoking process maintains
  - lc(XX) : a local copy of the implementation of reg. X
  - read_set(T) : the set of names of all the registers read by T up to that moment
  - write_set(T) : the set of names of all the registers written by T up to that moment
  - birthdate(T) : the value of CLOCK(+1) at the starting of T

Idea: commit a transaction iff it could appear as executed at its birthdate time

Consistency:

- If T reads X, then it must be that XX.date < birthdate(T)
- To commit, all registers accessed by T cannot have been modified after T's birthdate (again, XX.date < birthdate(T))

A Logical Clock based STM system

```
beginT() :=
    read_set(T), write_set(T) ← ∅
    birthdate(T) ← CLOCK+1

X.readT() :=
    if lc(XX)≠⊥ then return lc(XX).val
    lc(XX) ← XX
    if lc(XX).date ≥ birthdate(T) then ABORT
    read_set(T) ← read_set(T) ∪ {X}
    return lc(XX).val

X.writeT(v) :=
    if lc(XX)=⊥ then lc(XX) ← newloc
    lc(XX).val ← v
    write_set(T) ← write_set(T) ∪ {X}

try_to_commitT() :=
    lock all read_set(T) ∪ write_set(T)
    ∀ X ∈ read_set(T)
        if XX.date ≥ birthdate(T)
        then release all locks
             ABORT
    tmp ← CLOCK.fetch&add(1)+1
    ∀ X ∈ write_set(T)
        XX ← ⟨lc(XX).val , tmp⟩
    release all locks
    COMMIT
```

Remark: to avoid deadlock, there is a total order on the registers and locks are required by respecting this order (the deadlock is avoided as in Solution 1 of the Dining Philosophers)

Virtual World Consistency

Opacity requires a total order on all committed transactions and on all read prefixes of all aborted transactions

→ this latter requirement can be weakened by imposing that the read prefix of an aborted transaction is consistent only w.r.t. its causal past (i.e., its vortual world)

Opacity: total order both on all committed trans.'s and on read prefixes of aborted trans.'s

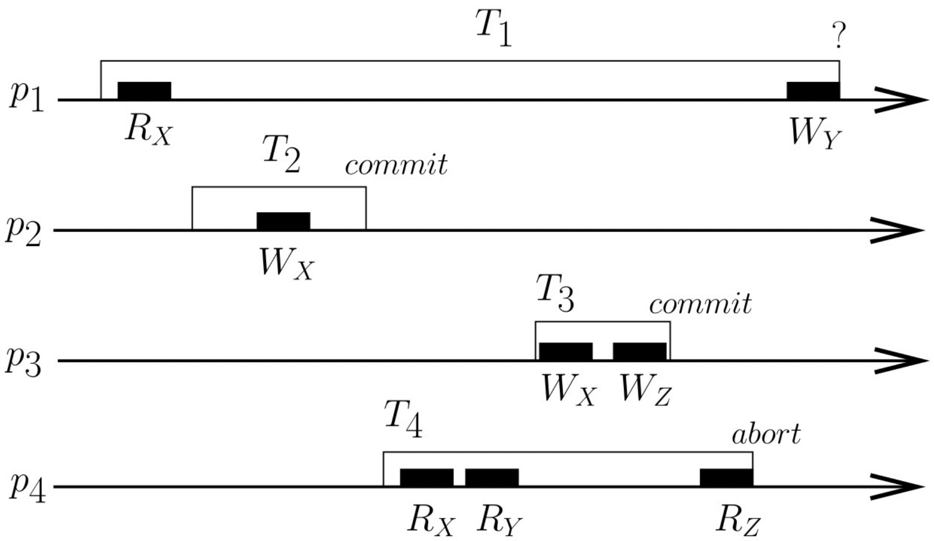
VWC: total order on all committed trans.'s + partial order on committed trans.'s and the read prefixes of aborted trans.'s

The **causal past** of a transaction T is the set of all T' and T'' such that

- T reads a value written by T', and
- T'' belongs to the causal past of T'

VWC allows more transactions to commit → it is a more liberal property than opacity

Virtual World Consistency



Without T4, an opaque execution is

T1 < T2 < T3

and they can all commit.

With T4:

- T4 must abort when reads Z (it cannot be executed atomically: before T3 for reading X and after T3 for reading Z)
- T1 < T2 < r_p(T4) < T3 , where r_p(T4) is {R_X, R_Y} is not opaque because the R_Y in T4 would read the value written by T1, unless also T1 aborts.

  → an aborted trans. leads T1 to abort

  → NOT GOOD!

With VWC we can let T1 commit:

- The total order on committed transactions is T1 < T2 < T3
- The partial order is T2 < r_pref(T4)

A Vector clock based STM system

We have m shared MRMW registers; register X is represented by a pair XX, with:

- XX.val the current value of X
- XX.depend[1…m] a vector clock s.t.
  - XX.depend[X] is the sequence number associated with the current val of X
  - XX.depend[Y] is the sequence number associated with the val of Y on which the current val of X depends from
- There is a starvation-free lock object associated to the pair

We have n processes; process pi has

- For every X, a local copy lc(XX) of the implementation of X
- $p\_depend_i[1…m]$ s.t. $p\_depend_i[X]$ is the seq.num. of the last val of X (directly or undirectly) known by pi

Every transaction T issues by pi has:

- read_set(T) and write_set(T)
- $t\_depend_T[1…m]$ a local copy of p_depend_i (this is used in the optimistic execution, not to change p_depend_i if T aborts)

A Vector clock based STM system

SAPIENZA
UNIVERSITÀ DI ROMA
DIPARTIMENTO DI INFORMATICA

```
begin_T(i) :=
    read_set(T), write_set(T) ← ∅
    t_depend_T ← p_depend_i

X.read_T(i) :=
    if lc(XX)=⊥ then
        lc(XX) ← newloc
        lc(XX) ← XX
        read_set(T) ← read_set(T) ∪ {X}
        t_depend_T[X] ← lc(XX).depend[X]
        if ∃ Y ∈ read_set(T) s.t.
            t_depend_T[Y] < lc(XX).depend[Y]
        then ABORT
        ∀ Y ∉ read_set(T) do
            t_depend_T[Y] ← max{t_depend_T[Y],
                                lc(XX).depend[Y]}
    return lc(XX).val
```

```
X.write_T(i,v) :=
    if lc(XX)=⊥ then lc(XX) ← newloc
    lc(XX).val ← v
    write_set(T) ← write_set(T) ∪ {X}

try_to_commit_T(i) :=
    lock all read_set(T) ∪ write_set(T)
    if ∃ Y ∈ read_set(T) s.t.
        t_depend_T[Y] < YY.depend[Y]
    then  release all locks
          ABORT
    ∀ X ∈ write_set(T) do
        t_depend_T[X] ← XX.depend[X]+1
    ∀ X ∈ write_set(T) do
        XX ← ⟨lc(XX).val, t_depend_T⟩
    release all locks
    p_depend_i ← t_depend_T
    COMMIT
```