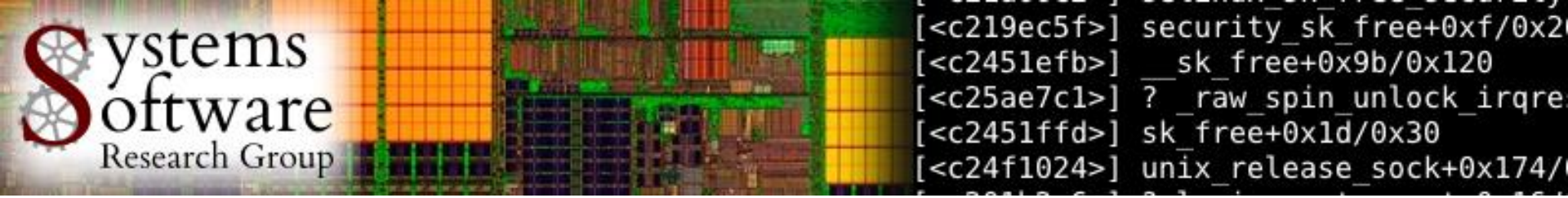


Software Transactional Memory (7)

Mohamed Mohamedin

Chapter 4 of TM Book

Long Jump

- **setjmp.h**
  - int setjmp(jmp_buf env)
    - Set the long jump location
    - Return value is zero if directly invocded
  - void longjmp(jmp_buf env, int value)
    - Do the long jump to the set location
    - Value is the returned value from setjump when the jump is executed

Long Jump

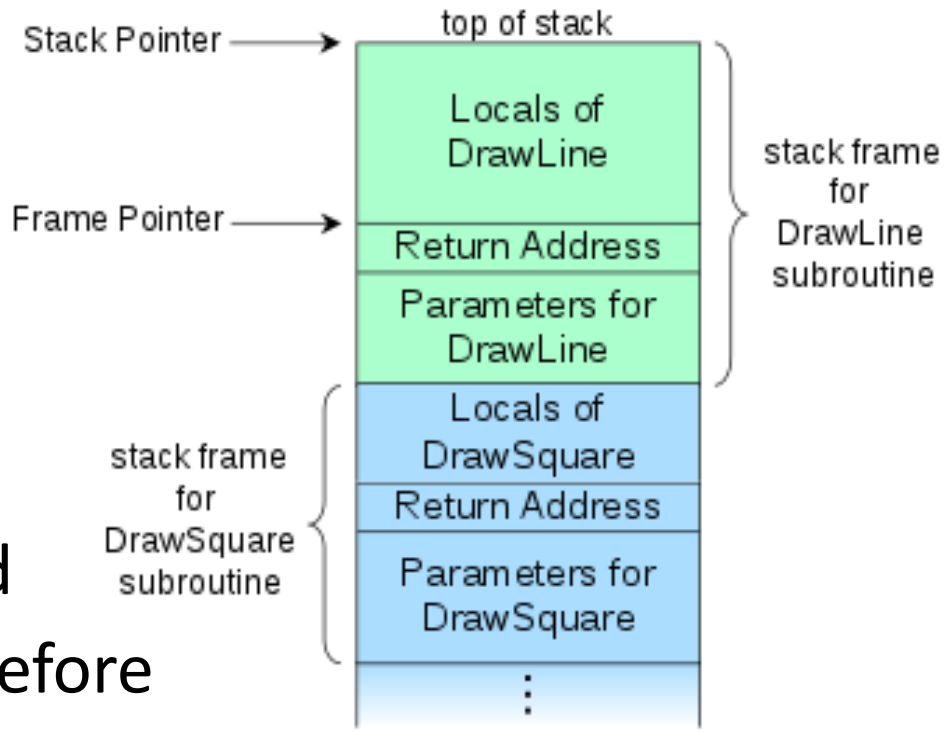
```c
#include <stdio.h>
#include <setjmp.h>

static jmp_buf buf;

void second(void) {
    printf("second\n");         // prints
    longjmp(buf,1);             // jumps back to where setjmp was called - making setjmp now return 1
}

void first(void) {
    second();
    printf("first\n");          // does not print
}

int main() {
    if (!setjmp(buf))
        first();                // when executed, setjmp returned 0
    else                        // when longjmp jumps back, setjmp returns 1
        printf("main\n");       // prints

    return 0;
}
```

Long Jump

- Call Stack

```
DrawSquare() {
    DrawLine();
}
```

- Setjmp saves the current call stack
  - So, the function called setjmp can not return before calling longjmp
    - Otherwise, the saved state will be invalid



Revising STM algorithms we studied so far

REVISION

NOREC

NOrec

- One global counter (acts as a global lock)
  - Global versioned lock
- Value-based validation
  - Read-set has the location and the read value
- Writes are buffered

NOrec

- tx_begin()
  - Reset read-set and write-set (write-buffer)
  - do
    - RV = Global-Clock
  - while((RV & 1) != 0)

NOrec

- tx_write(addr, value)
  - Add (or update) the addr and value to the write-set

NOrec

- tx_read(addr)
  - Find the addr is in the write-set
    - If found, return the value buffered in the write-set
  - val = *addr
  - while (RV != Global-Clock)
    - RV = tx_validate()
    - val = *addr
  - Add (addr & val) to read-set
  - Return val

NOrec

- tx_validate()
  - while(true)
    - time = Global-Clock
    - if ((time & 1) != 0) continue
    - for each entry in the read-set
      - if (*entry.addr != entry.val)
        - » tx_abort()
    - if (time == Global-Clock)
      - return time

NOrec

- tx_commit()
  - if (write-set.size == 0) // read-only tx
    - return
  - while(!CAS(&Global-Clock, RV, RV+1))
    - RV = tx_validate()
  - //Write back
  - For each entry in the write-set
    - *entry.addr = entry.value
  - //Unlock and update global clock version
  - Global-Clock = RV+2

NOrec

- tx_abort
  - //Just jump back to tx_begin to restart the transaction

RINGSTM

RingSTM

- Bloom Filters
  - It is an array of bits
  - Represents a Set
  - It is a probabilistic data structure
    - It can tell if an element is possibly in the Set
      - Has False Positives
    - BUT, it can 100% tell that an element is NOT in the Set
  - Contains return
    - Possibly in the Set
    - Definitely NOT in the Set
  - Has only Add & Contains. No Remove

RingSTM

- Array of bits of a given size *m*
  - Initially all bits are zeros
  - Each element is hashed using *k* hash functions
    - Each one map an element to a bit
    - Set those bits to ones

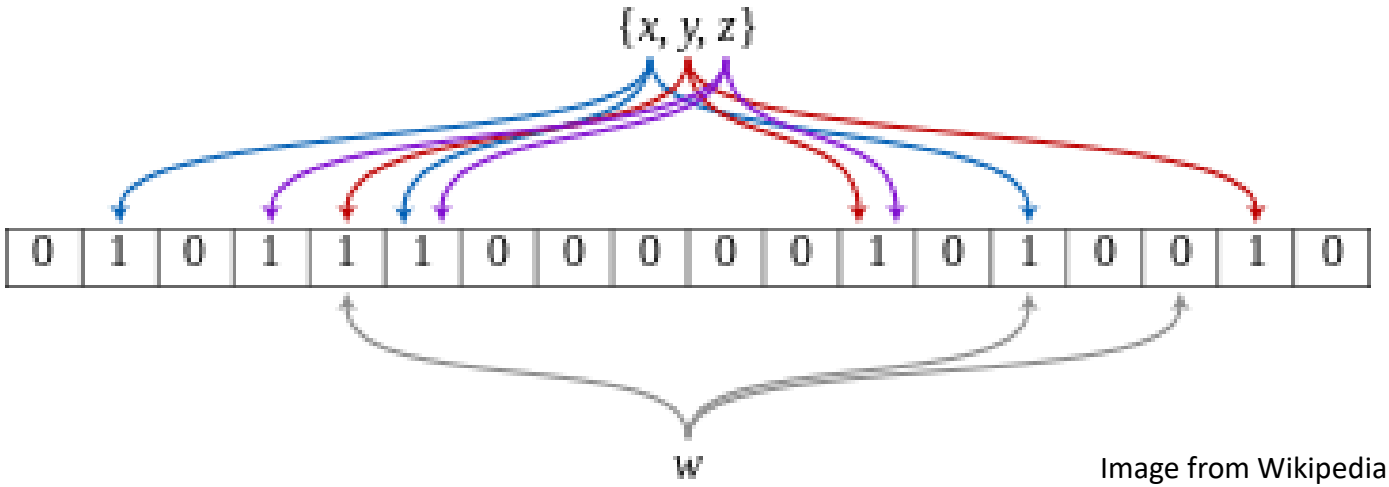


Image from Wikipedia

RingSTM

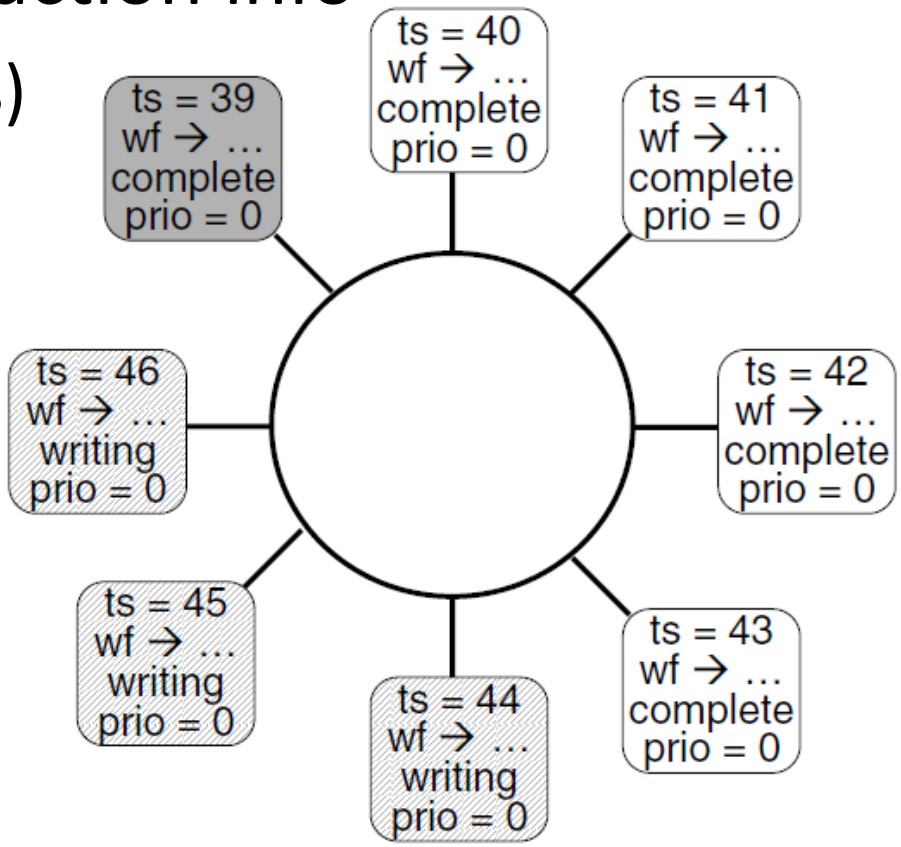
- Other methods can be defined
  - Intersect:
    - Check if two Bloom filters has common elements
  - Union:
    - Merge two sets (Bloom filters)

RingSTM

- How it works?
  - Metadata
    - Global:
      - Global-Clock (ring-index)
      - The Ring: “An ordered, fixed size ring data structure”
    - Thread-local:
      - Read-set signature
      - Write-set signature
      - Write-set
      - RV

The Ring

- Circular data structure
- Hold committed transaction info
  - Commit Timestamp (ts)
  - Write-signature (wf)
  - Status
  - Priority
  - Initially:
    - All have ts = 0
    - Status = complete

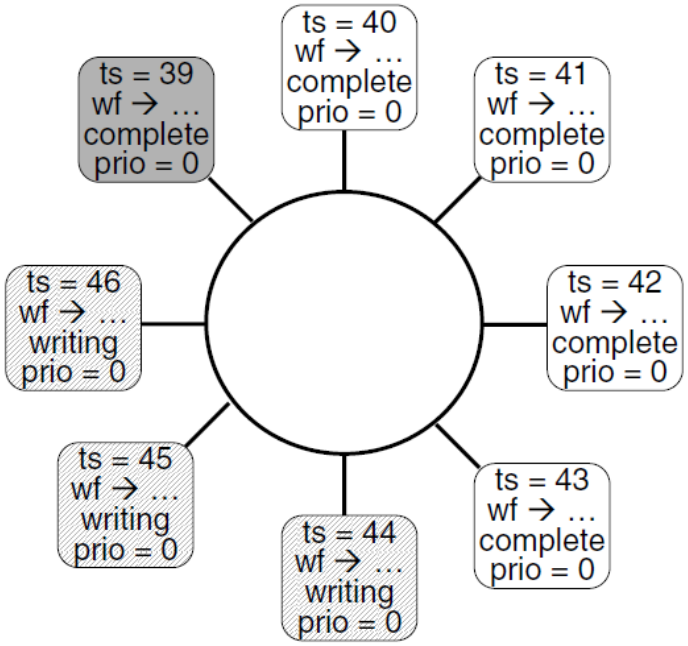


The Ring

- Only write transactions modifies the Ring
  - One CAS operation to add an entry
  - A transaction is committed (logically) once its entry is added to the Ring (status: writing)
  - After writing back is finished

    Status: complete

    Physically committed



RingSTM

- tx_begin()
  - Its idea is to find oldest entry in the ring that is still writing back.
  - It depends on this invariant

$$L_i.st = writing \implies \forall_{k>i}\ L_k.st = writing$$

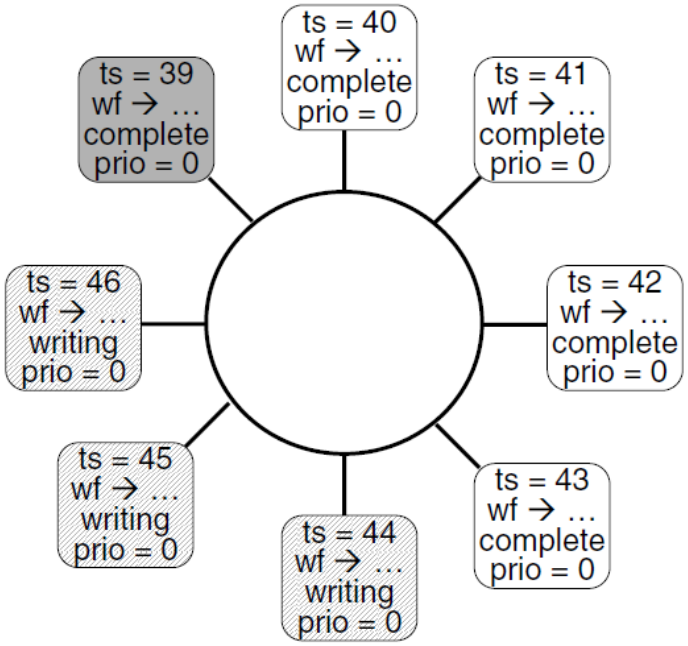
    - A transaction cannot change its status to complete if an older transaction is still writing
  - Guarantee detecting potential conflicts with the transactions still writing
    - Without waiting

RingSTM

- tx_begin()
  - Its idea is to find oldest entry in the ring that is still writing back.
  - It depends on this invariant

$$L_i.st = writing \implies \forall_{k>i}\ l$$

    - A transaction cannot change older transaction is still writir
  - Guarantee detecting potent transactions still writing
    - Without waiting



RingSTM

- tx_begin()
  - Reset thread-local metadata
  - RV = Global-Clock
  - while (ring[RV].status != complete ||
    ring[RV].timestamp < RV)
    - RV--

RingSTM

- tx_write(addr, value)
  - Add (or update) the addr and value to the write-set
  - Add the addr to the write-set signature

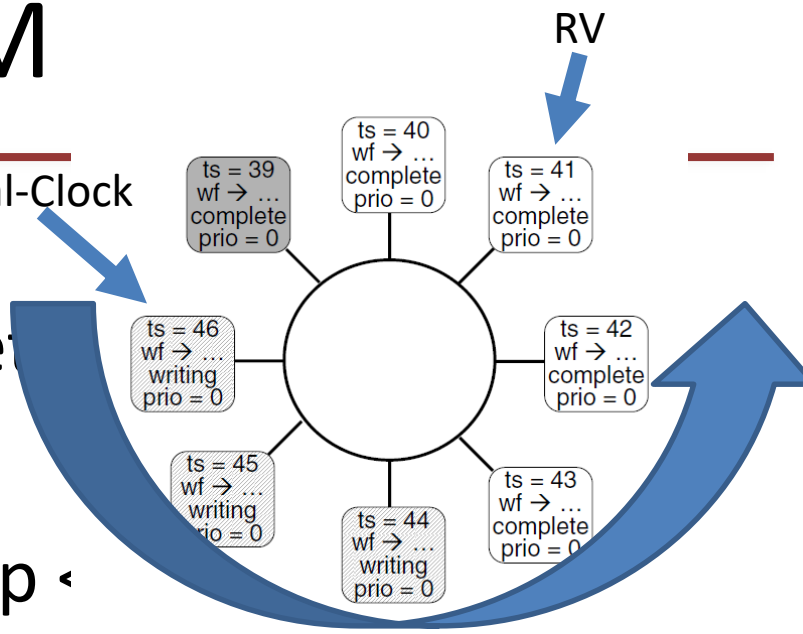
RingSTM

- tx_read(addr)
  - Find the addr is in the write-set signature
    - If found, find the addr is in the write-set
      - return the value buffered in the write-set
  - val = *addr
  - Add addr to read-set signature
  - tx_validate()
  - Return val

RingSTM

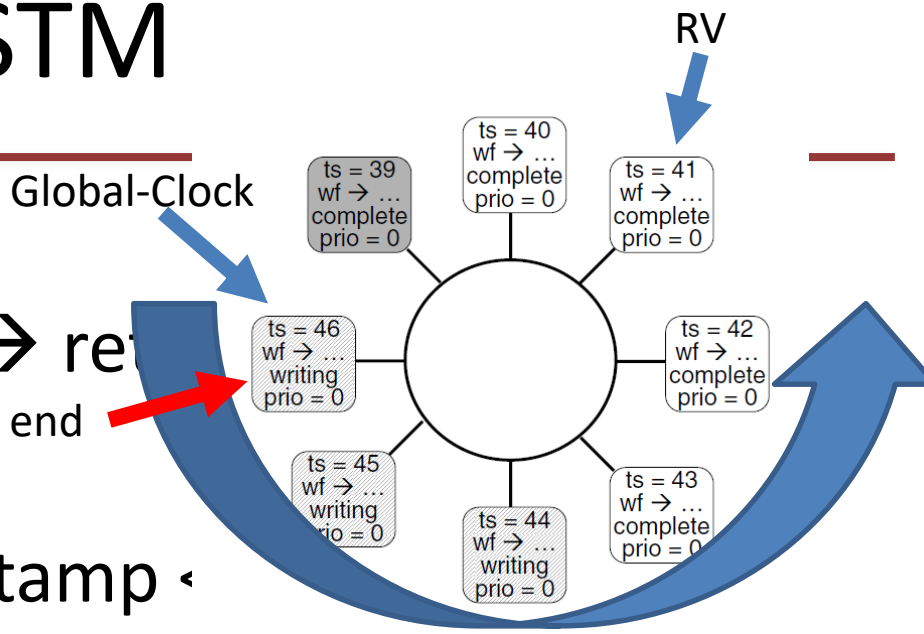
- tx_validate()
  - if Global-Clock == RV → return
  - end = Global-Clock
  - while (ring[end].timestamp < end) wait
  - for ring entries between Clobal-Clock & RV+1
    - if (ring-entry.write-sig ∩ read-set signature)
      - tx_abort()
    - if (ring-entry.status == writing)
      - end = (ring-entry-index) – 1
  - RV = end

RingSTM



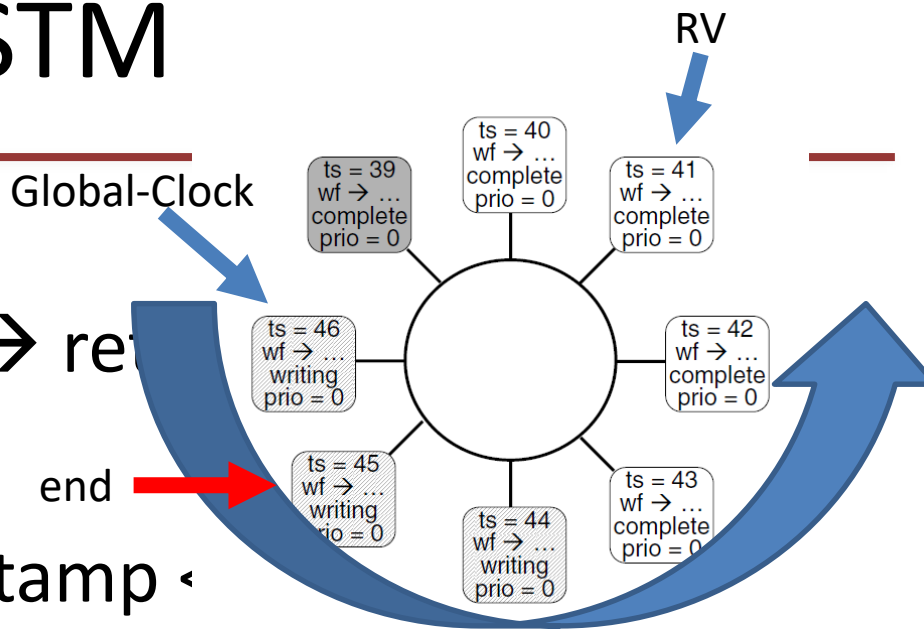
- tx_validate()
  - if Global-Clock == RV → re
  - end = Global-Clock
  - while (ring[end].timestamp
  - for ring entries between Clobal-Clock & RV+1
    - if (ring-entry.write-sig ∩ read-set signature)
      - tx_abort()
    - if (ring-entry.status == writing)
      - end = (ring-entry-index) – 1
  - RV = end

RingSTM



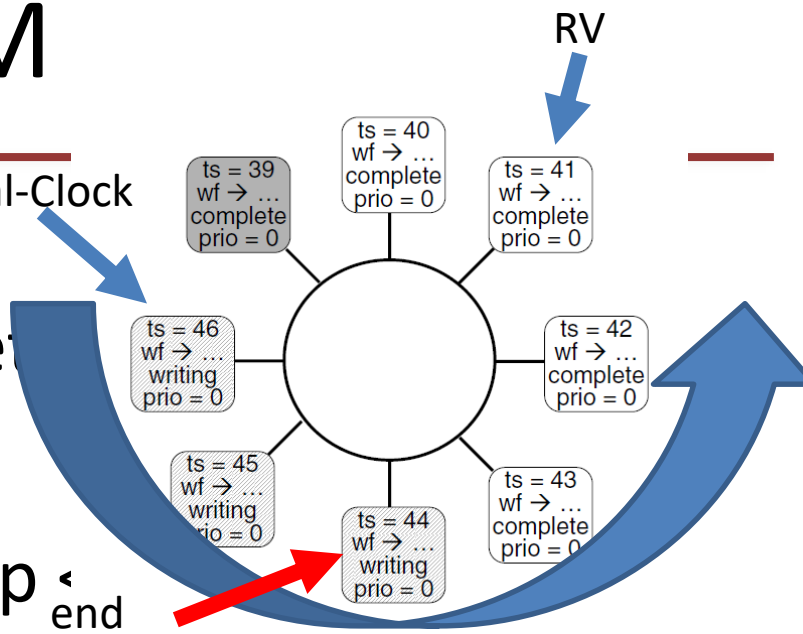
- tx_validate()
  - if Global-Clock == RV → re
  - end = Global-Clock
  - while (ring[end].timestamp
  - for ring entries between Clobal-Clock & RV+1
    - if (ring-entry.write-sig ∩ read-set signature)
      - tx_abort()
    - if (ring-entry.status == writing)
      - end = (ring-entry-index) – 1
  - RV = end

RingSTM



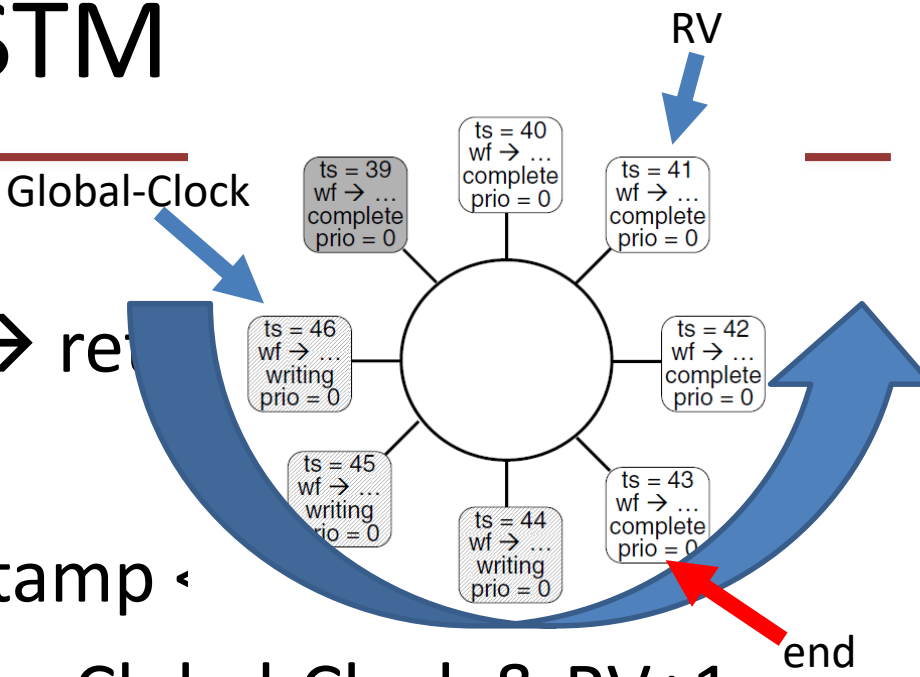
- tx_validate()
  - if Global-Clock == RV → re
  - end = Global-Clock
  - while (ring[end].timestamp
  - for ring entries between Clobal-Clock & RV+1
    - if (ring-entry.write-sig ∩ read-set signature)
      - tx_abort()
    - if (ring-entry.status == writing)
      - end = (ring-entry-index) – 1
  - RV = end

RingSTM



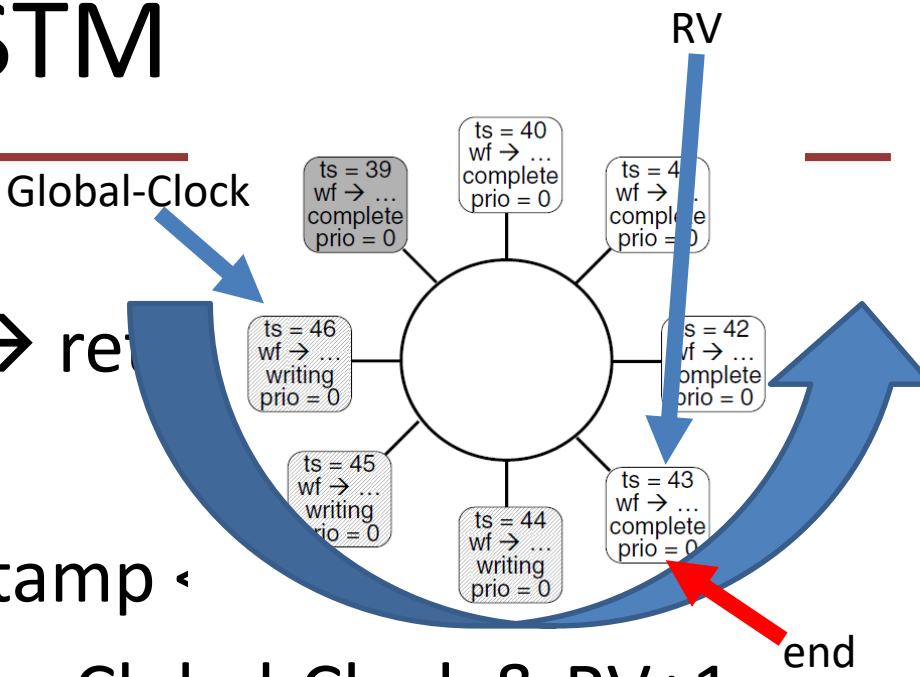
- tx_validate()
  - if Global-Clock == RV → re
  - end = Global-Clock
  - while (ring[end].timestamp <
  - for ring entries between Global-Clock & RV+1
    - if (ring-entry.write-sig ∩ read-set signature)
      - tx_abort()
    - if (ring-entry.status == writing)
      - end = (ring-entry-index) – 1
  - RV = end

RingSTM



- tx_validate()
  - if Global-Clock == RV → re
  - end = Global-Clock
  - while (ring[end].timestamp
  - for ring entries between Global-Clock & RV+1
    - if (ring-entry.write-sig ∩ read-set signature)
      - tx_abort()
    - if (ring-entry.status == writing)
      - end = (ring-entry-index) – 1
  - RV = end

RingSTM



- tx_validate()
  - if Global-Clock == RV → re
  - end = Global-Clock
  - while (ring[end].timestamp
  - for ring entries between Global-Clock & RV+1
    - if (ring-entry.write-sig ∩ read-set signature)
      - tx_abort()
    - if (ring-entry.status == writing)
      - end = (ring
  - RV = end

Timestamp extension. No need to check these entries again

RingSTM

- tx_commit()
  - if (write-set.size == 0) → return // read-only tx
  - *again*: commit_time = Global-Clock
  - tx_validate()
  - If (!CAS(&Global-Clock, commit_time, commit_time +1))
    - goto again
  - ring[commit_time + 1] = {writing, write-set-sig, commit_time +1}
  - for (i = commit_time *downto* RV + 1)
    - if (ring[i].write-sig ∩ write-set signature)
      - while(ring[i].status == writing) wait
  - For each entry in the write-set
    - *entry.addr = entry.value //Write back
  - while(ring[commit_time].status == writing) wait
  - ring[commit_time + 1].status = complete

RingSTM

- tx_commit()
  - if (write-set.size == 0) → return // read-only tx
  - *again*: commit_time = Global-Clo
  - tx_validate()
  - If (!CAS(&Global-Clock, commit
    - goto again
  - ring[commit_time + 1] = {writing,
  - for (i = commit_time *downto* RV + 1)
    - if (ring[i].write-sig ∩ write-set signature)
      - while(ring[i].status == writing) wait
  - For each entry in the write-set
    - *entry.addr = entry.value //Write back
  - while(ring[commit_time].status == writing) wait
  - ring[commit_time + 1].status = complete

As in TL2 and NOrec, nothing extra is done for committing read-only transaction. It is linearized at the last tx_read's validation

RingSTM

- tx_commit()
  - if (write-set.size == 0) → return // read-only tx
  - *again*: commit_time = Global-Clock
  - tx_validate()
  - If (!CAS(&Global-Clock, +1))
    - goto again
  - ring[commit_time + 1] = { time +1}
  - for (i = commit_time *do*
    - if (ring[i].write-sig ∩ write-set signature)
      - while(ring[i].status == writing) wait
  - For each entry in the write-set
    - *entry.addr = entry.value //Write back
  - while(ring[commit_time].status == writing) wait
  - ring[commit_time + 1].status = complete

Validation is done before adding an entry to the ring. This minimize contention window. Also, no need to add entries for transactions that will be aborted

RingSTM

- tx_commit()
  - if (write-set.size == 0) → return // read-only tx
  - *again*: commit_time = Global-Clock
  - tx_validate()
  - If (!CAS(&Global-Clock, commit_time, commit_time +1))
    - goto again
  - ring[commit_time + 1] = {writing
  - for (i = commit_time *downto* F
    - if (ring[i].write-sig ∩ write-set sig
      - while(ring[i].status == writing) w
  - For each entry in the write-set
    - *entry.addr = entry.value //Write back
  - while(ring[commit_time].status == writing) wait
  - ring[commit_time + 1].status = complete

If the transaction is valid, try to reserve an entry in the ring using the commit_time value (which is captured before the validation)

RingSTM

- tx_commit()
  - if (write-set.size == 0) → return // read-only tx
  - *again*: commit_time = Global-Clock
  - tx_validate()
  - If (!CAS(&Global-Clock, commit_time, commit_time +1))
    - goto again
  - ring[commit_tim = {writing, write-set-sig, commit_time +1}
  - for (i = con
    - if (ring[i]
      - while
  - For each e
    - *entry.addr = entry.value //write back
  - while(ring[commit_time].status == writing) wait
  - ring[commit_time + 1].status = complete

If CAS failed, this mains another transaction(s) has committed while doing the validation. This requires another validation (only for the newly committed transactions. Why?)

RingSTM

- tx_commit()
  - if (write-set.size == 0) → return // read-only tx
  - *again*: commit_time = Global-Clock
  - tx_validate()
  - If (!CAS(&Global-Clock, commit_time, commit_time +1))
    - goto again
  - ring[commit_time + 1] = {writing, write-set-sig, commit_time +1}
  - for (i = commit_time *downto* RV + 1)
    - if (ring[i].write-sig ∩ write-set signatu
      - while(ring[i].status == writing) wait
  - For each entry in the write-set
    - *entry.addr = entry.value //Write back
  - while(ring[commit_time].status == writing) wait
  - ring[commit_time + 1].status = complete

At this point, my transaction is committed (it is valid and it has reserved its entry in the ring)

RingSTM

- tx_commit()
  - if (write-set.size == 0) → return // read-only tx
  - *again*: commit_time = Global-Clock
  - tx_validate()
  - If (!CAS(&Global-Clock, commit_time, commit_time +1))
    - goto again
  - ring[commit_time + 1] = {writing,
  - for (i = commit_time *downto* R
    - if (ring[i].write-sig ∩ write-set sig
      - while(ring[i].status == writing) wa
  - For each entry in the write-set
    - *entry.addr = entry.value //Write back
  - while(ring[commit_time].status == writing) wait
  - ring[commit_time + 1].status = complete

This is the linearization point, a successful CAS

RingSTM

- tx_commit()
  - if (write-set.size == 0) → return // read-only tx
  - *again*: commit_time = Global-Clock
  - tx_validate()
  - If (!CAS(&Global-Clock, commit_time, commit_time +1))
    - goto again
  - ring[commit_time + 1] = {writing, write-set-sig, commit_time +1}
  - for (i = commit_time *downto* RV + 1
    - if (ring[i].write-sig ∩ write-set sig
      - while(ring[i].status == writing) wa
  - For each entry in the write-set
    - *entry.addr = entry.value //Write ba
  - while(ring[commit_time].status == w
  - ring[commit_time + 1].status =

Notice that the ring entry initially has ts = 0 (using our simplified infinite ring). At this line, the ring entry is populated. This is why we needed "ring[RV].timestamp < RV" in tx_begin and tx_validate

RingSTM

- tx_commit()
  - if (write-set.size == 0) → return // read-only tx
  - *again*: commit_time = Global-Clock
  - tx_validate()
  - If (!CAS(&Global-Clock, commit_time, commit_time +1))
    - goto again
  - ring[commit_time + 1] = {writing, write-set-sig, commit_time +1}
  - for (i = commit_time *downto* RV + 1)
    - if (ring[i].write-sig ∩ write-set signature)
      - while(ring[i].status == writing) wait
  - For each entry in the write-set
    - *entry.addr = entry.value //Write bac
  - while(ring[commit_time].status == wr
  - ring[commit_time + 1].status = (

The write-set-sig acts as a write-lock. This preserve write-after-write ordering

RingSTM

- tx_commit()
  - if (write-set.size == 0) → return // read-only tx
  - *again*: commit_time = Global-Clock
  - tx_validate()
  - If (!CAS(&Global-Clock, commit_time, commit_time +1))
    - goto again
  - ring[commit_time + 1] = {writing, write-set-sig, commit_time +1}
  - for (i = commit_time *downto* RV +
    - if (ring[i].write-sig ∩ write-set signatu
      - while(ring[i].status == writing) wait
  - For each entry in the write-set
    - *entry.addr = entry.value //Write back
  - while(ring[commit_time].status == writing) wait
  - ring[commit_time + 1].status = complete

If there is no common elements between writing transactions, we can proceed with writing back in parallel.

RingSTM

- tx_commit()
  - if (write-set.size == 0) → return // read-only tx
  - *again*: commit_time = Global-Clock
  - tx_validate()
  - If (!CAS(&Global-Clock, commit_time, commit_time +1))
    - goto again
  - ring[commit_time + 1] = {writing, write-set-sig, commit_time +1}
  - for (i = commit_time *downto* RV
    - if (ring[i].write-sig ∩ write-set sign
      - while(ring[i].status == writing) wait
  - For each entry in the write-set
    - *entry.addr = entry.value //Write back
  - while(ring[commit_time].status == writing) wait
  - ring[commit_time + 1].status = complete

Protect our invariant of not allowing a complete transaction until all previous transactions are complete

RingSTM

- tx_commit()
  - if (write-set.size == 0) → return // read-only tx
  - *again*: commit_time = Global-Clock
  - tx_validate()
  - If (!CAS(&Global-Clock, commit_time, commit_time +1))
    - goto again
  - ring[commit_time + 1] = {writing, write-set-sig, commit_time +1}
  - for (i = commit_time *downto* RV + 1)
    - if (ring[i].write-sig ∩ write-set signatu
      - while(ring[i].status == writing) wait
  - For each entry in the write-set
    - *entry.addr = entry.value //Write back
  - while(ring[commit_time].status == writing) wa
  - ring[commit_time + 1].status = complete

Finally, mark the transaction's ring entry as complete which means all data are written to the memory and all previous entries in the ring are complete also

RingSTM

- tx_abort
  - //Just jump back to tx_begin to restart the transaction

RingSTM

- Pros
  - Lightweight read-set
  - Fast validation
  - Low memory overhead
  - Low cache pressure (one CAS operation)
  - Concurrent commits
  - Livelock freedom
- Cons
  - Imprecise validation
    - False conflict aborts
    - Sensitive to the Bloom filter size and hash function