Software Transactional Memory (5)

Mohamed Mohamedin

Chapter 4 of TM Book

Lock-based STMs with Global Metadata

- RingSTM
- Another STM with Global Metadata, BUT
  - Doesn't use value-based validation
  - Instead, it uses signature-based validation

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

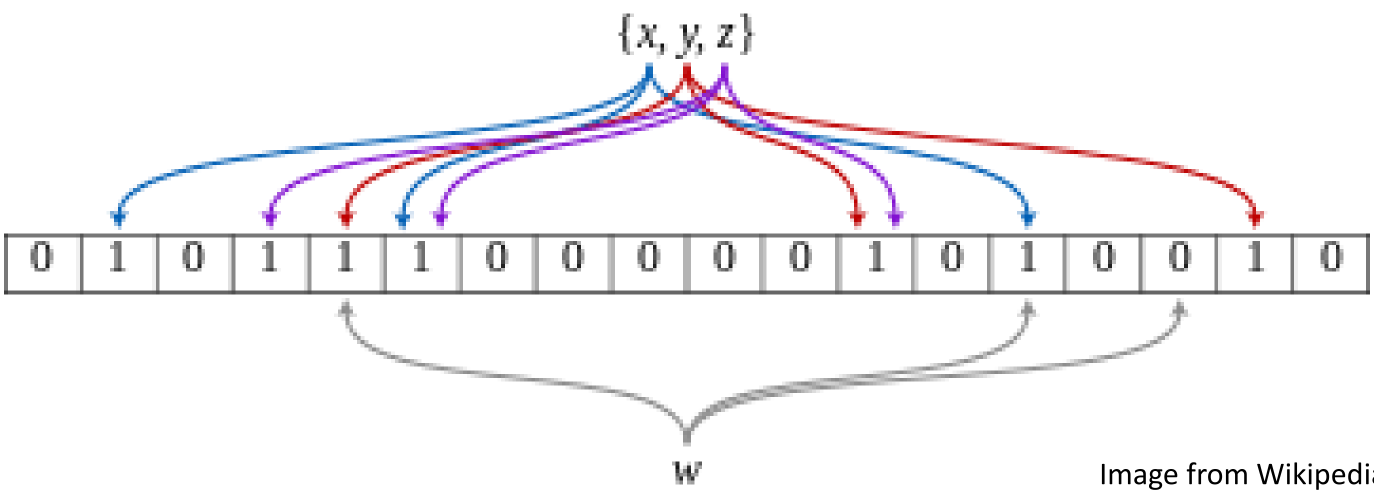


Image from Wikipedia

RingSTM

- Add(element)
  - Hash the element with each hash function
  - Set the corresponding bit to one
  - E.g., elements x, y, and z

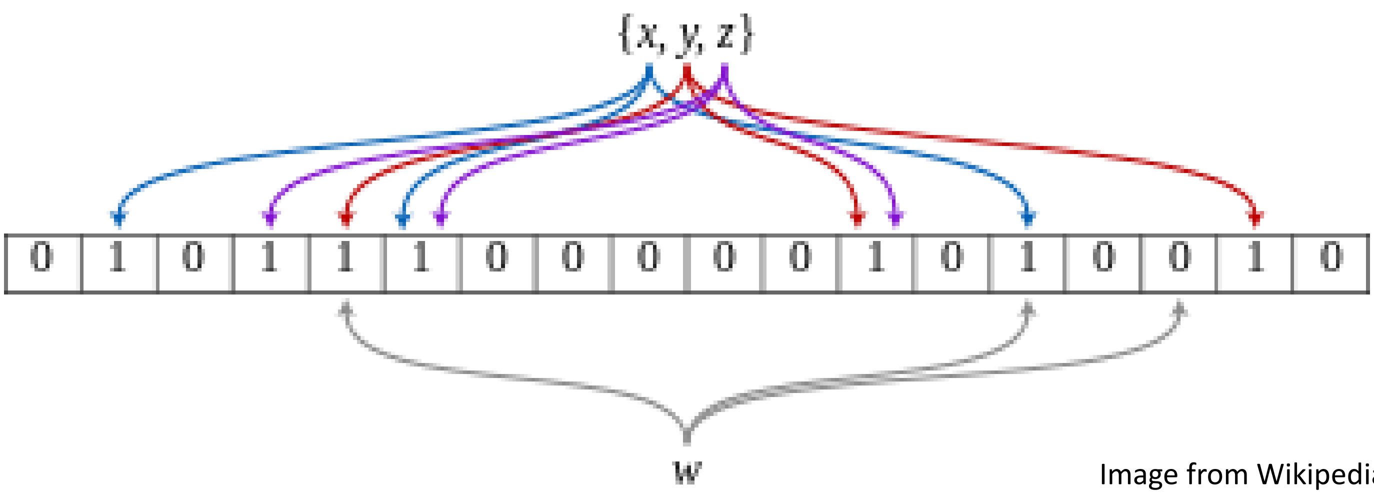


Image from Wikipedia

RingSTM

- Contains(element)
  - Hash the element with each hash function
  - If ALL corresponding bit are ones
    - Return true (possibly in the set)
  - Else: It is NOT in the set (definitely)
  - E.g., w is not in the Set

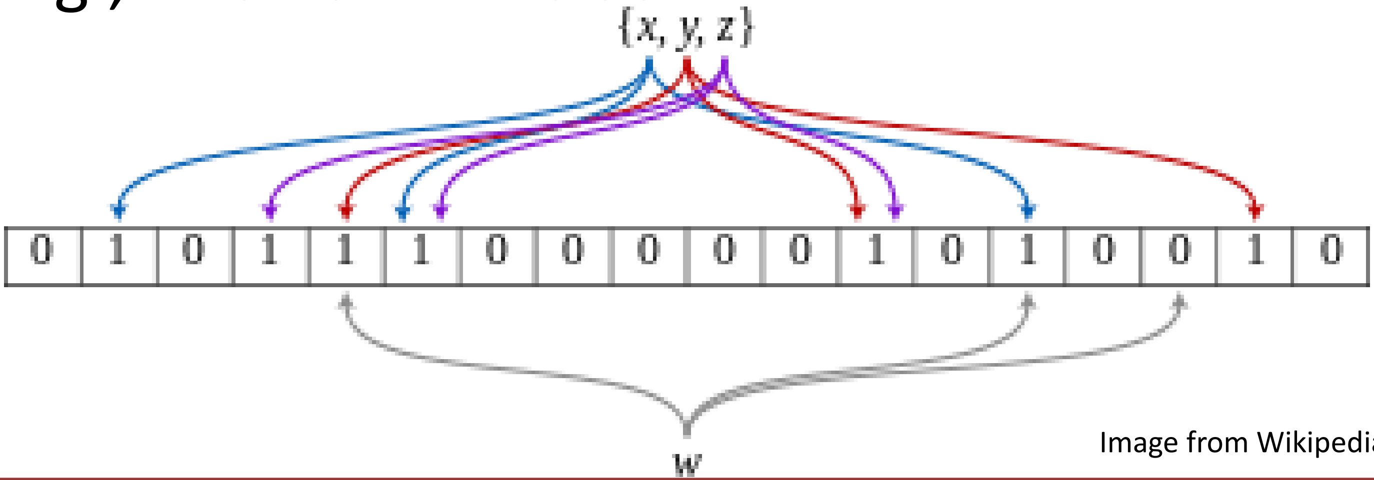


Image from Wikipedia

RingSTM

- Probability of false positives
  – Adding too many elements to a small Bloom filter increased false-positives significantly

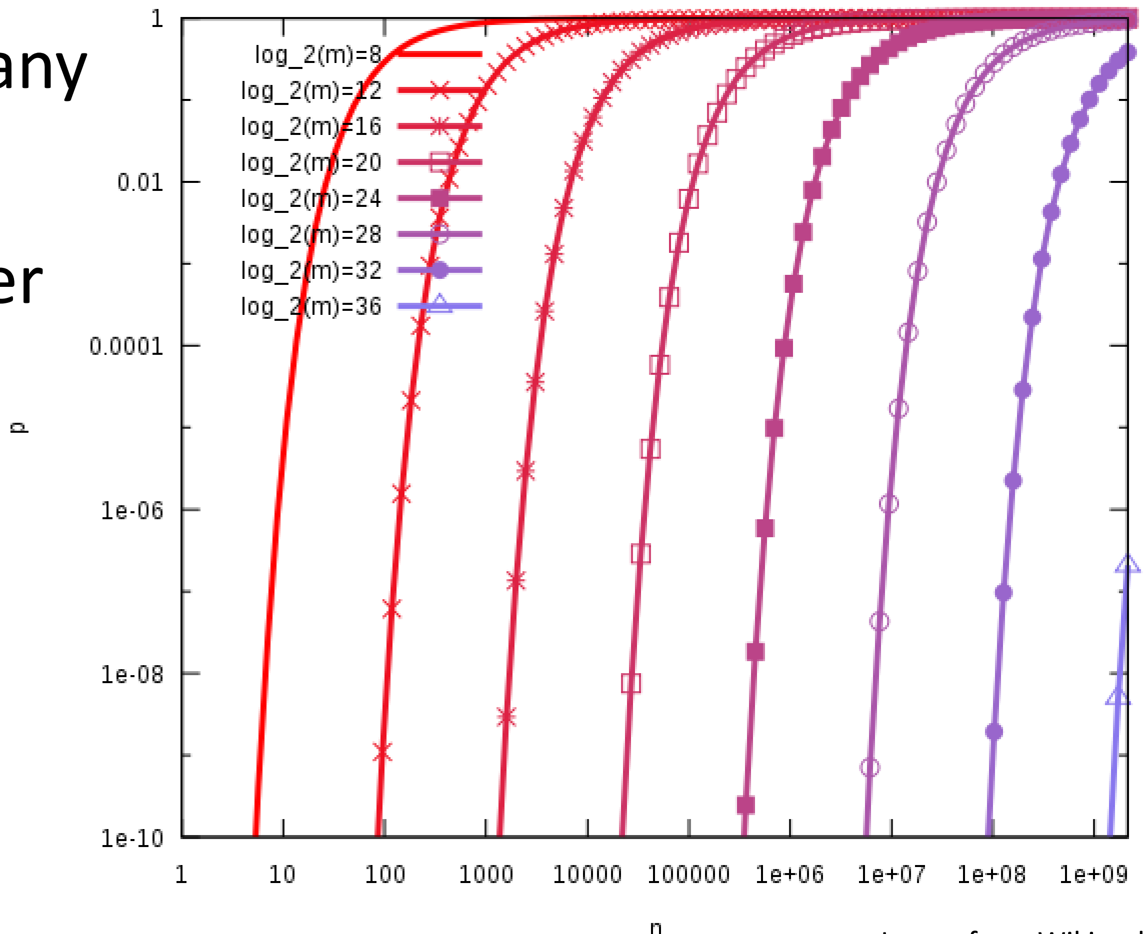


Image from Wikipedia

RingSTM

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

RingSTM

- A simplified Bloom filter implementation
  - From RSTM (http://cs.rochester.edu/research/synchronization/rstm/)
  - One hash function is used

RingSTM

```
template <uint32_t BITS>
class BitFilter
{
    static const uint32_t WORD_SIZE   = 8 * sizeof(uintptr_t);
    static const uint32_t WORD_BLOCKS = BITS / WORD_SIZE;

    uintptr_t word_filter[WORD_BLOCKS];

    static uint32_t hash(const void* const key)
    {
        return (((uintptr_t)key) >> 3) % BITS;
    }

    public:

    BitFilter() { clear(); }

    void add(const void* const val) volatile
    {
        const uint32_t index  = hash(val);
        const uint32_t block  = index / WORD_SIZE;
        const uint32_t offset = index % WORD_SIZE;
        word_filter[block] |= (1u << offset);
    }
```

RingSTM

```
bool lookup(const void* const val) const volatile
{
    const uint32_t index  = hash(val);
    const uint32_t block  = index / WORD_SIZE;
    const uint32_t offset = index % WORD_SIZE;

    return word_filter[block] & (1u << offset);
}

void unionwith(const BitFilter<BITS>& rhs)
{
    for (uint32_t i = 0; i < WORD_BLOCKS; ++i)
        word_filter[i] |= rhs.word_filter[i];
}

void clear() volatile
{
    for (uint32_t i = 0; i < WORD_BLOCKS; ++i)
        word_filter[i] = 0;
}
```

RingSTM

```
void fastcopy(const volatile BitFilter<BITS>* rhs) volatile
{
    for (uint32_t i = 0; i < WORD_BLOCKS; ++i)
        word_filter[i] = rhs->word_filter[i];
}

bool intersect(const BitFilter<BITS>* rhs) const volatile
{
    for (uint32_t i = 0; i < WORD_BLOCKS; ++i)
        if (word_filter[i] & rhs->word_filter[i])
            return true;
    return false;
}
```

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

RingSTM

- How it works?
  - Metadata
    - Global:
      - Global-Clock (ring-in
      - The Ring: "An ordere
    - Thread-local:
      - Read-set signature
      - Write-set signature
      - Write-set
      - RV

Listing 1 RingSTM metadata

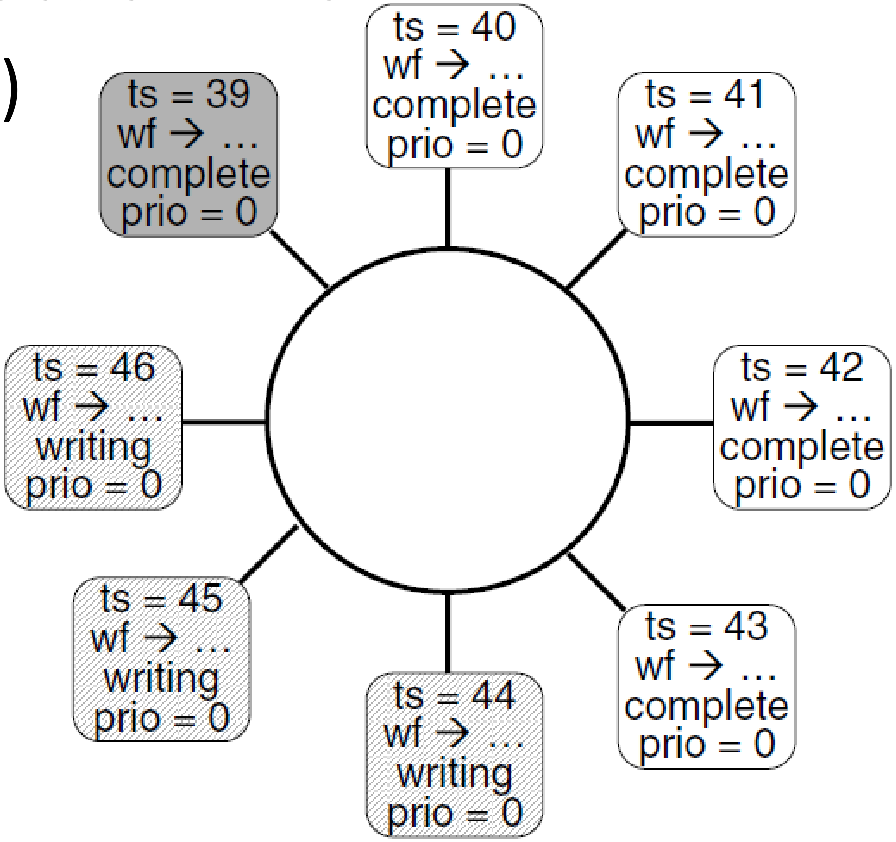
```
struct RingEntry
  int      ts             // commit timestamp
  filter   wf             // write filter
  int      prio           // priority
  int      st             // writing or complete

struct Transaction
  set      wset           // speculative writes
  filter   wf             // addresses to write
  filter   rf             // addresses read
  int      start          // logical start time

RingEntry ring[]          // the global ring
int       ring_index      // newest ring entry
int       prefix_index // RingR prefix field
```

The Ring

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

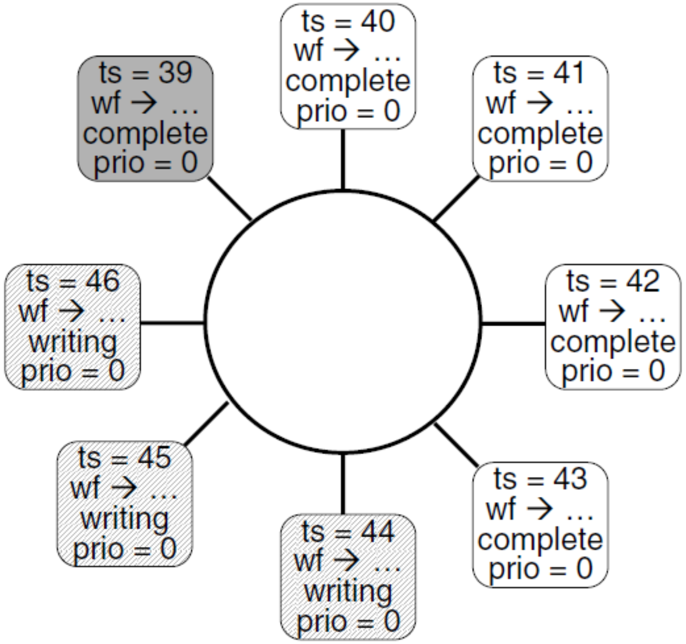


The Ring

- 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

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

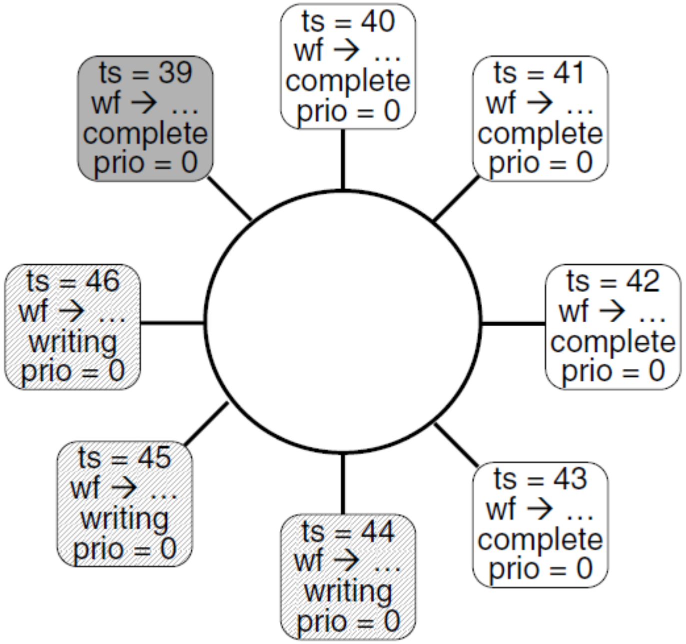
    - A transaction cannot change its status to complete if an older transaction is still writing
  - Guarantee detecting potential conflicts with the transactions still writing
    - Without waiting

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 i
      older transaction is still writir
  - Guarantee detecting potent
    transactions still writing
    - Without waiting

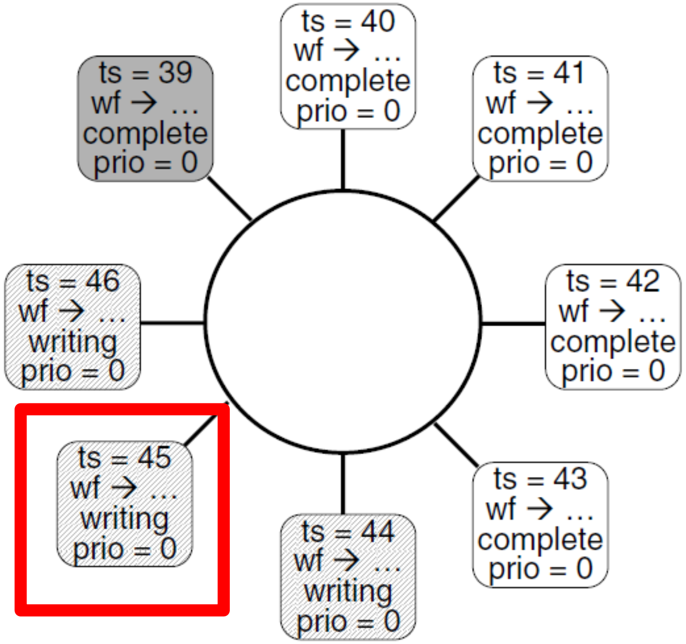


RingSTM

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



RingSTM

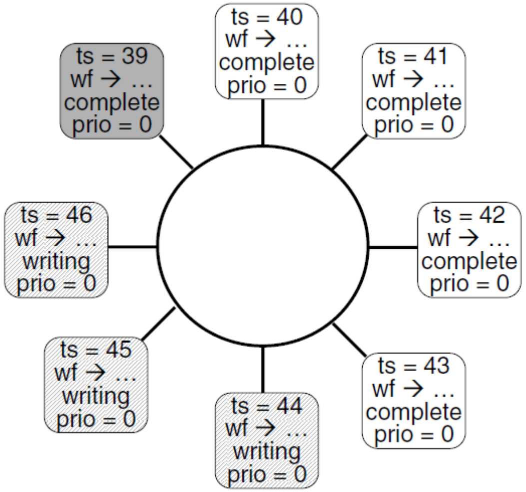
- tx_begin()
  - Reset thread-local metadata
  - RV = Global-Clock
  - while (ring[RV].status != complete | |
    ring[RV].timestamp < RV)
    - RV--

We will assume infinite ring for simplicity
Note: Global-Clock is the current ring-index

RingSTM

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

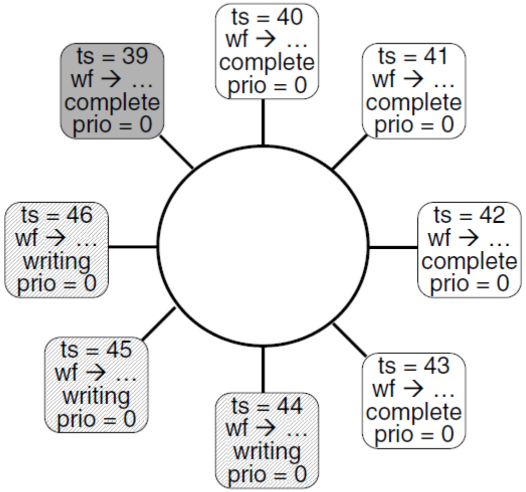
Lock backward for the first complete entry



RingSTM

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

The case when global-clock is incremented but the entry is not set yet



RingSTM

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

```
tm_begin:
 1 read ring_index to TX.start
 2 while ring[TX.start].st != complete ||
        ring[TX.start].ts < TX.start
 3  TX.start--
 4 fence(Read-Before-Read)
```

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

```
tm_write:
TX.wset.add(addr, val)
TX.wf.add(addr)
```

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_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
  - val = *addr
  - Add addr to read-set signature
  - tx_validate()
  - Return val

Bloom filters lookup is very fast [O(1)], so it can be used to save the expensive write-set lookup
Note: TL2 uses a Bloom filter also

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 sig
  - tx_validate()
  - Return val

But, Bloom filter has false-positive
So, we have to confirm the address
is actually in the write-set

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

No read-set, only a read-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

To support Opacity, a read must be validated

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

```
tm_read:
if addr in TX.wf && addr in TX.wset
 return lookup(addr, TX.wset)
val=*addr
TX.rf.add(addr)
fence(Read-Before-Read)
check()
return val
```

RingSTM

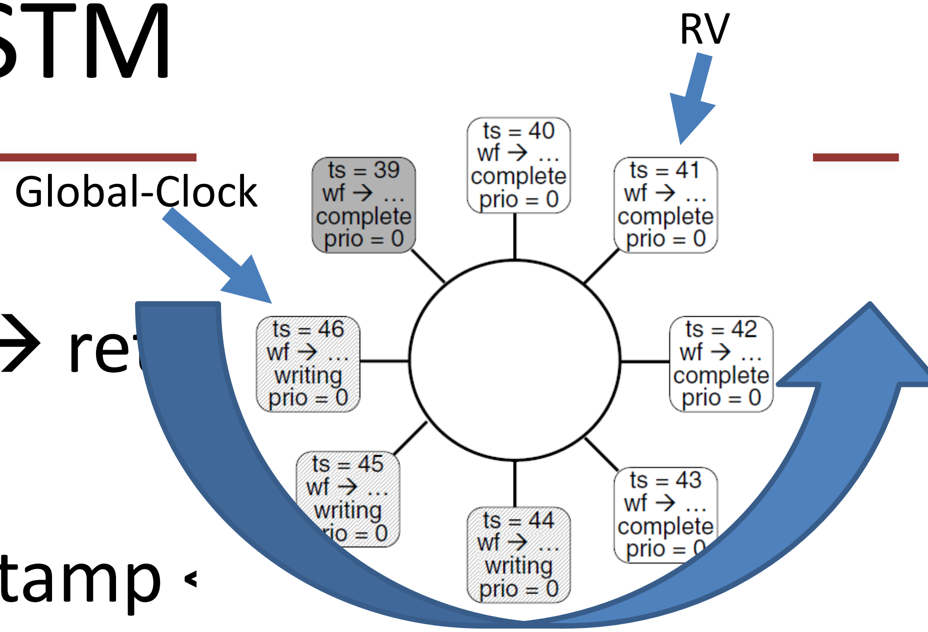
- tx_validate()
  - if Global-Clock == RV → return
  - end = Global-Clock
  - while (ring[end].timestamp < end) wait
  - for ring entries between Clobal-Clock & RV+1
    - if (ring-entry.write-sig ∩ read-set signature)
      - tx_abort()
    - if (ring-entry.status == writing)
      - end = (ring-entry-index)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

RingSTM

- tx_validate()
  - if Global-Clock == RV → return
  - end = Global-Clock
  - while (ring[end].timestamp < end) wait
  - for ring entries b … RV+1
    - if (ring-entry.wri … e)
      - tx_abort()
    - if (ring-entry.status == writing)
      - end = (ring-entry-index) – 1
  - RV = end

The case when global-clock is incremented but the entry is not set yet

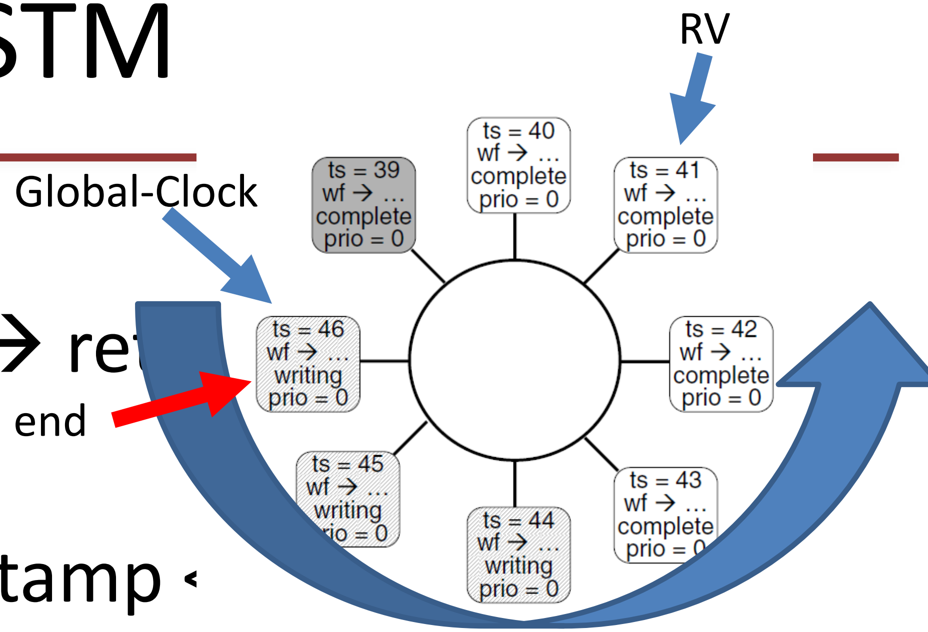
RingSTM



- 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

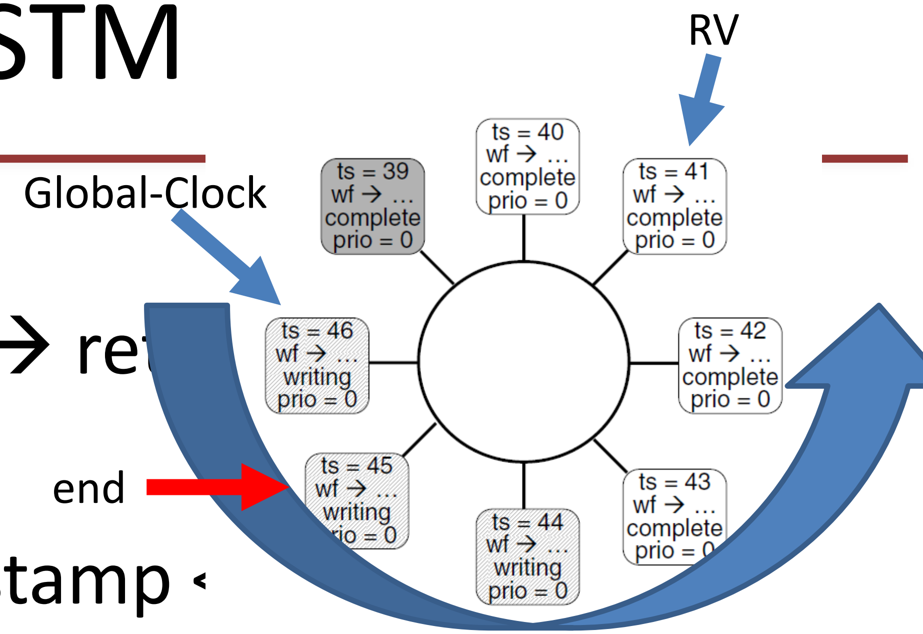
RingSTM

- 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



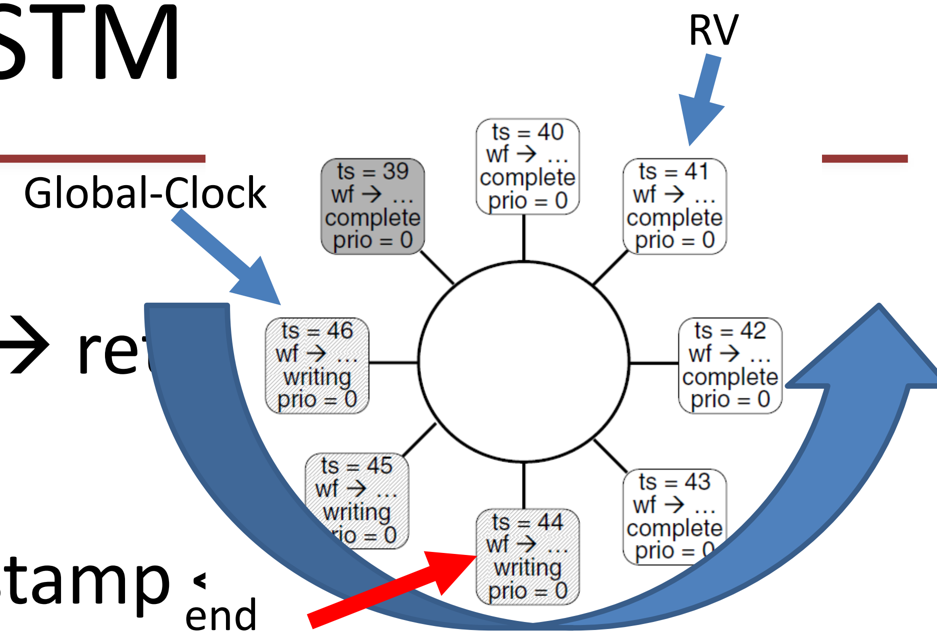
RingSTM

- 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

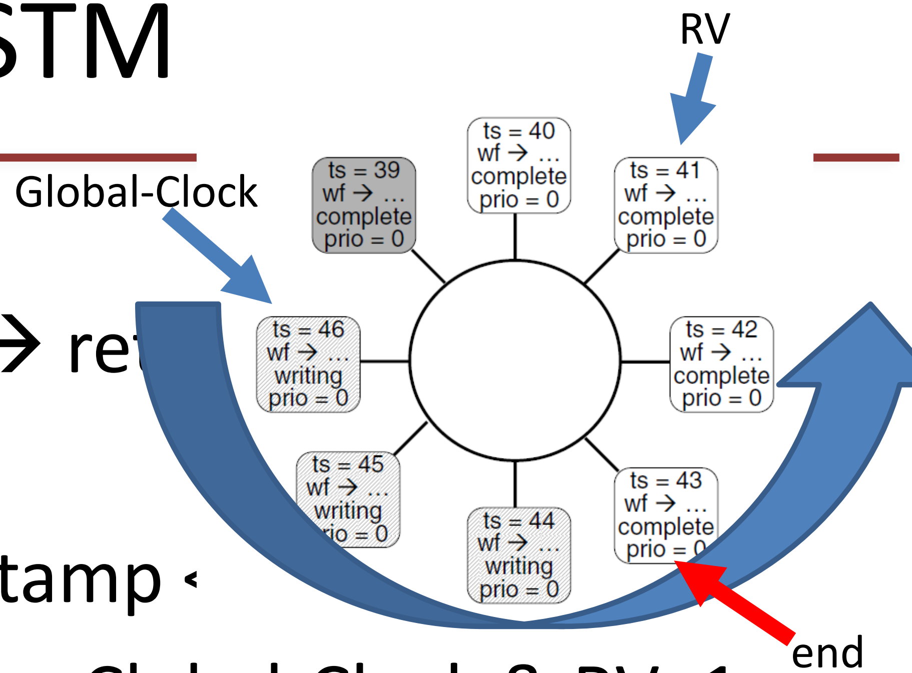


RingSTM

- tx_validate()
  - if Global-Clock == RV → ret
  - 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

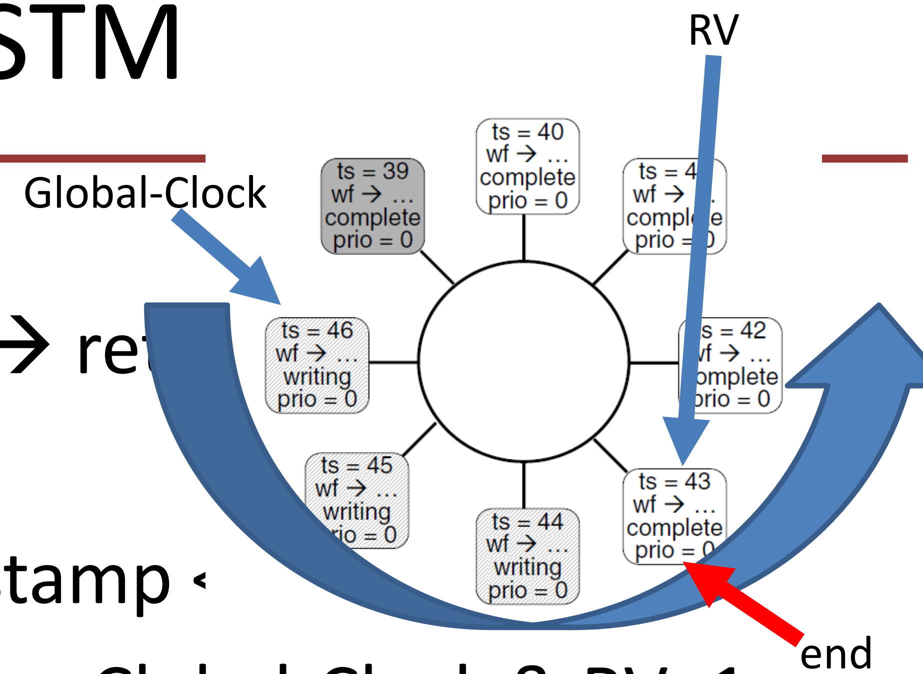


RingSTM



- tx_validate()
  - if Global-Clock == RV → ret
  - 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
  - RV = end

Timestamp extension. No need to check these entries again

RingSTM

```
check:
if ring_index==TX.start return
suffix_end=ring_index
while (ring[suffix_end].ts < suffix_end) SPIN
for i=ring_index downto TX.start+1
 if intersect(ring[i].wf, TX.rf)
  abort()
 if ring[i].st==writing
  suffix_end=i-1
TX.start=suffix_end
```

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-Cloc
  - 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 ... 1}
  - for (i = commit_time *downto* F
    - if (ring[i].write-sig ∩ write-set si
      - 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) wai
  - 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

```
tm_end:
 1 if read-only return
 2 commit_time=ring_index
 3 check()
 4 if !CAS(ring_index, commit_time, commit_time+1)
 5  goto 2
 6 ring[commit_time+1]=(writing, TX.wf, commit_time+1)
 7 for i=commit_time downto TX.start+1
 8  if intersect(ring[i].wf, TX.wf)
 9   while ring[i].st=writing SPIN
10 fence(ReadWrite-Before-Write))
11 for (addr, val) in TX.wset
12  write val at addr
13 while ring[commit_time].st==writing SPIN
14 fence(ReadWrite-Before-Write)
15 ring[commit_time+1].st=complete
```

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

Is RingSTM Opaque?

- Does it suffer from zombies?

T1:

- tx_begin
- tx_write(x, 2)
- tx_write(y, 4)
- tx_commit()

Initially x=1, y=2
Invariant: y = 2x

T2:

- tx_begin
- v1 = tx_read(y)
- v2 = tx_read(x)
- tx_write(z, 1/(v1 – v2))
- tx_commit()

Is RingSTM Opaque?

- Does it suffer from zombies?

T1:

- tx_begin
- tx_write(x, 2)
- tx_write(y, 4)
- tx_commit()

T2:

- tx_begin
- v1 = tx_read(y)
- v2 = tx_read(x)
- tx_write(z, 1/(v1 – v2))
- tx_commit()

NO zombies, it validates after each read. Any addition to the ring will trigger a validation after a tx_read. Y is in T2 read-set-sig and will intersect with T1 write-set-sig in the ring

RingSTM

T1:

- tx_begin
- v1 = tx_read(x)
- tx_write(y, v1 + 10)
- tx_commit()

T2:

- tx_begin
- v1 = tx_read(x)
- v2 = tx_read(y)
- tx_write(y, v1 + v2)
- tx_write(x, v1 + 1)
- tx_commit()

RingSTM

T1:

- tx_begin
- v1 = tx_read(x)
- tx_write(y, v1 + 10)
- tx_commit()

Will both transactions commit?
Are they Linearizable?

T2:

- tx_begin
- v1 = tx_read(x)
- v2 = tx_read(y)
- tx_write(y, v1 + v2)
- tx_write(x, v1 + 1)
- tx_commit()

RingSTM

T1:

- tx_begin
- v1 = tx_read(x)
- tx_write(y, v1 + 10)
- tx_commit()

IT DEPENDS!

T2:

- tx_begin
- v1 = tx_read(x)
- v2 = tx_read(y)
- tx_write(y, v1 + v2)
- tx_write(x, v1 + 1)
- tx_commit()

RingSTM

T1:

- tx_begin
- v1 = tx_read(x)
- tx_write(y, v1 + 10)
- tx_commit()

We need to know the Bloom filter size and hash function.
Assume a Bloom filter of size 10 and a single hash function {addr%10}
Address of x is 10
Address of y is 100

T2:

- tx_begin
- v1 = tx_read(x)
- v2 = tx_read(y)
- tx_write(y, v1 + v2)
- tx_write(x, v1 + 1)
- tx_commit()

RingSTM

T1:

- tx_begin
- v1 = tx_read(x)
- tx_write(y, v1 + 10)
  - (Addr of y) % 10 → 100 % 10 = 0
- tx_commit()
  - Write-set-sig = [1,0,0,0,0,0,0,0,0,0]
  - Global-Clock = 1

We need to know the Bloom filter size and hash function.
Assume a Bloom filter of size 10 and a single hash function {addr%10}
Address of x is 10
Address of y is 100

T2:

- tx_begin (RV = 0)
- v1 = tx_read(x)
  - (Addr of x) % 10 → 10 % 10 = 0
  - Read-set-sig = [1,0,0,0,0,0,0,0,0,0]
- v2 = tx_read(y)
  - tx_validate →
    [1,0,0,0,0,0,0,0,0,0] ∩ [1,0,0,0,0,0,0,0,0,0] != 0
    → tx_abort()
- tx_write(y, v1 + v2)
- tx_write(x, v1 + 1)
- tx_commit()

RingSTM

T1:

- tx_begin
- v1 = tx_read(x)
- tx_write(y, v1 + 10)
  - (Addr of y) % 10 → 100 % 10 = 0
- tx_commit()
  - Write-set-sig = [1,0,0,0,0,0,0,0,0,0]
  - Global-Clock = 1

NOW
Assume a Bloom filter of size 10 and a single hash function {addr%10}
Address of x is 1
Address of y is 100

T2:

- tx_begin (RV = 0)
- v1 = tx_read(x)
  - (Addr of x) % 10 → 1 % 10 = 1
  - Read-set-sig = [0,1,0,0,0,0,0,0,0,0]
- v2 = tx_read(y)
  - tx_validate →
    [1,0,0,0,0,0,0,0,0,0] ∩ [0,1,0,0,0,0,0,0,0,0] == 0
    → Valid → RV = 1
- tx_write(y, v1 + v2)
- tx_write(x, v1 + 1)
- tx_commit()

RingSTM

T1:

- tx_begin
- v1 = tx_read(x)
- tx_write(y, v1 + 10)
  - (Addr of y) % 10 → 100 % 10 = 0
- tx_commit()
  - Write-set-sig = [1,0,0,0,0,0,0,0,0,0]
  - Global-Clock = 1

With RingSTM, there can be false conflicts which depends on how the Bloom filter is implemented

T2:

- tx_begin (RV = 0)
- v1 = tx_read(x)
  - (Addr of x) % 10 → 1 % 10 = 1
  - Read-set-sig = [0,1,0,0,0,0,0,0,0,0]
- v2 = tx_read(y)
  - tx_validate →
    [1,0,0,0,0,0,0,0,0,0] ∩ [0,1,0,0,0,0,0,0,0,0] == 0
    → Valid → RV = 1
- tx_write(y, v1 + v2)
- tx_write(x, v1 + 1)
- tx_commit()

RingSTM

T1:

- tx_begin
- v1 = tx_read(x)
- tx_write(y, v1 + 10)
- tx_commit()

T2:

- tx_begin
- v1 = tx_read(x)
- v2 = tx_read(y)
- tx_write(y, v1 + v2)
- tx_write(x, v1 + 1)
- tx_commit()

RingSTM

T1:

- tx_begin
- v1 = tx_read(x)
- tx_write(y, v1 + 10)
- tx_commit()

T2:

- tx_begin
- v1 = tx_read(x)
- v2 = tx_read(y)
- tx_write(y, v1 + v2)
- tx_write(x, v1 + 1)
- tx_commit()

Will both transactions commit?
Are they Linearizable?

RingSTM

T1:

- tx_begin
- tx_write(x, 15)
- v1 = tx_read(x)
- tx_write(y, v1 + 15)
- tx_commit()

T2:

- tx_begin
- v1 = tx_read(x)
- v2 = tx_read(y)
- tx_write(y, v1 + v2)
- tx_write(x, v1 + 1)
- tx_commit()

RingSTM

T1:

- tx_begin
- tx_write(x, 15)

T2:

- tx_begin
- v1 = tx_read(x)
- v2 = tx_read(y)
- tx_write(y, v1 + v2)
- tx_write(x, v1 + 1)
- tx_commit()

- v1 = tx_read(x)
- tx_write(y, v1 + 15)
- tx_commit()

What about this scenario?

RingSTM

T1:

- tx_begin
- v1 = tx_read(x)
- tx_write(y, v1 + 15)
- tx_commit()

T2:

- tx_begin
- v1 = tx_read(x)
- v2 = tx_read(y)
- tx_write(y, v1 + v2)
- tx_write(x, v1 + 1)
- tx_commit()

RingSTM

T1:

- tx_begin
- v1 = tx_read(x)
- tx_write(y, v1 + 15)
- tx_commit()

T2:

- tx_begin
- v1 = tx_read(x)
- v2 = tx_read(y)
- tx_write(y, v1 + v2)
- tx_write(x, v1 + 1)
- tx_commit()

What about this scenario?

RingSTM

T1:

- tx_begin
- v1 = tx_read(x)
- tx_write(y, v1 + 15)
- tx_commit()

T2:

- tx_begin
- …read(x)
- …d(y)
- …+ v2)
- …+ 1)
- …t()

In the previous 3 scenarios, Bloom filter implementation will not affect! WHY?