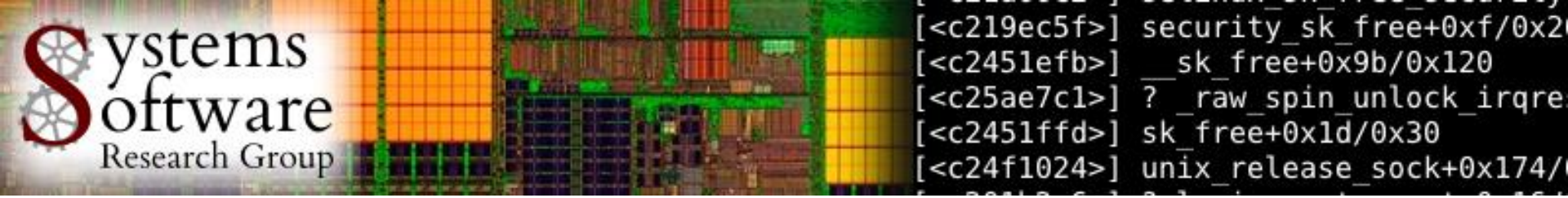


Hardware Transactional Memory (2)

Mohamed Mohamedin

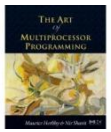
Chapter 5 of TM Book

HTM Limitation

- Best-efforts
  - Transactions are not guaranteed to commit
  - Must provide a software fallback path

Non-Speculative Fallback

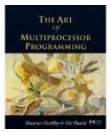
```
if (_xbegin() == _XBEGIN_STARTED) {
   read lock state
   if (lock taken) _xabort();
   work;
   _xend()
} else {
   lock->lock();
   work;
   lock->unlock();
}
```



Non-Speculative Fallback

```
if (_xbegin() == _XBEGIN_STARTED) {
  read lock state
  if (lock taken) _xabort();
  work;
  _xend()
} e
  -
  -
  w
  -
  -
}
```

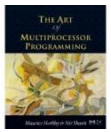
reading lock ensures that transaction will abort if another thread acquires lock



Non-Speculative Fallback

```
if (_xbegin() == _XBEGIN_STARTED) {
  read lock state
  if (lock taken) _xabort();
  work;
  _xend()
} else {
  lo
  wo
  lo
}
```

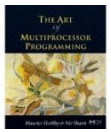
abort if another thread has acquired lock



Non-Speculative Fallback

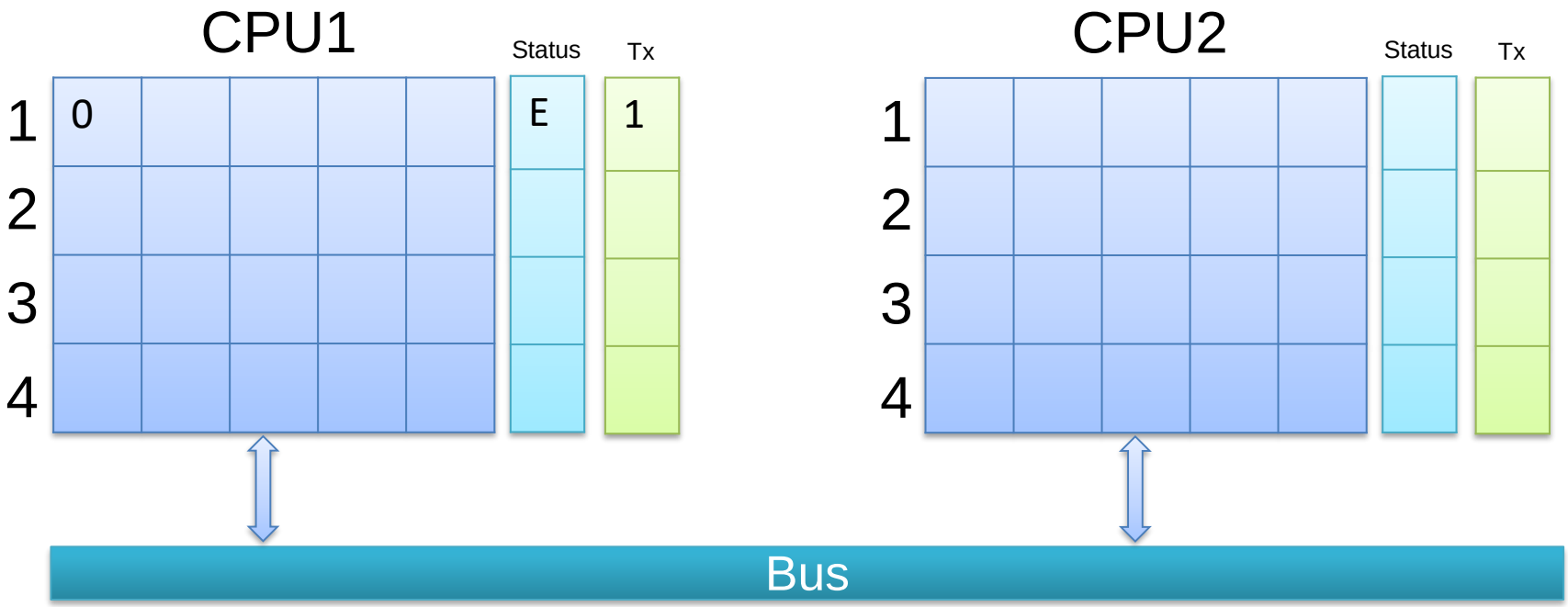
on abort, acquire lock & do work (aborting concurrent speculative transactions)

```
if
   re
   if (lock taken) _xabort();
   work;
   _xend()
} else {
   lock->lock();
   work;
   lock->unlock();
}
```



Global Lock Fallback

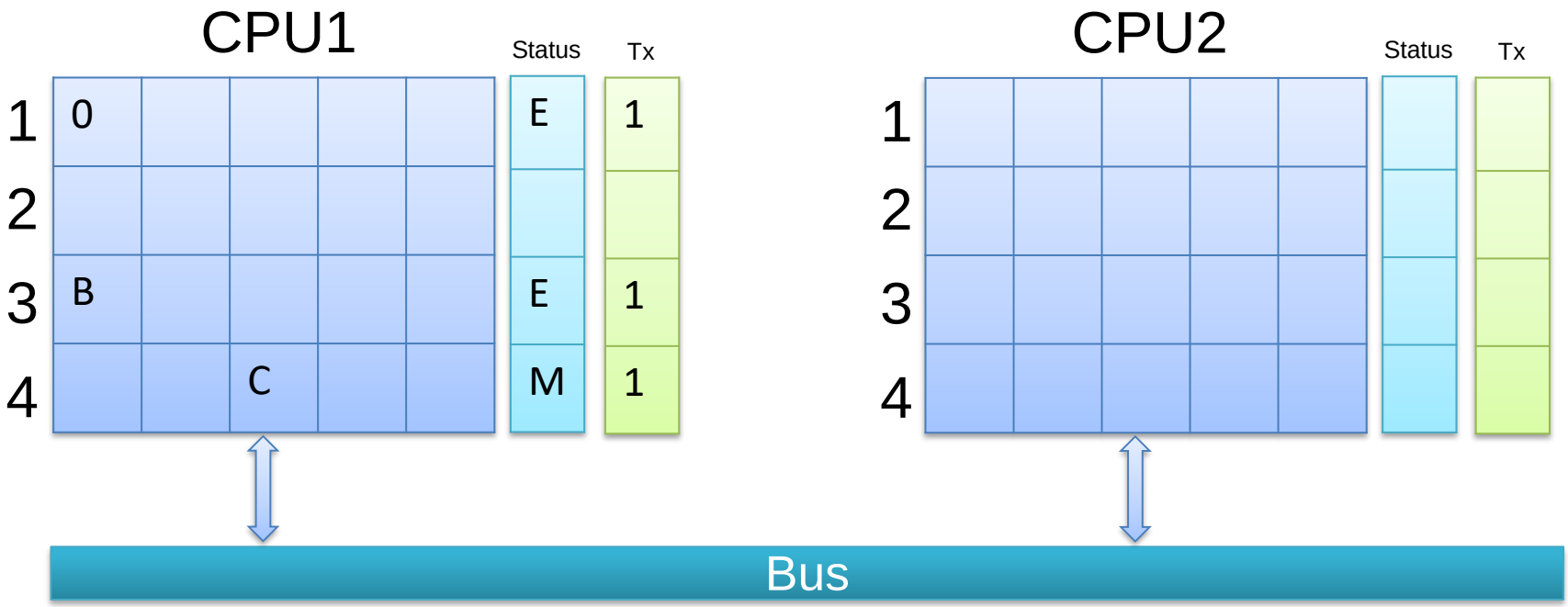
- Must define a software fallback path
  - Default is global lock



- _xbegin()
- if (read(lock)) == 1 then _xabort()

Global Lock Fallback

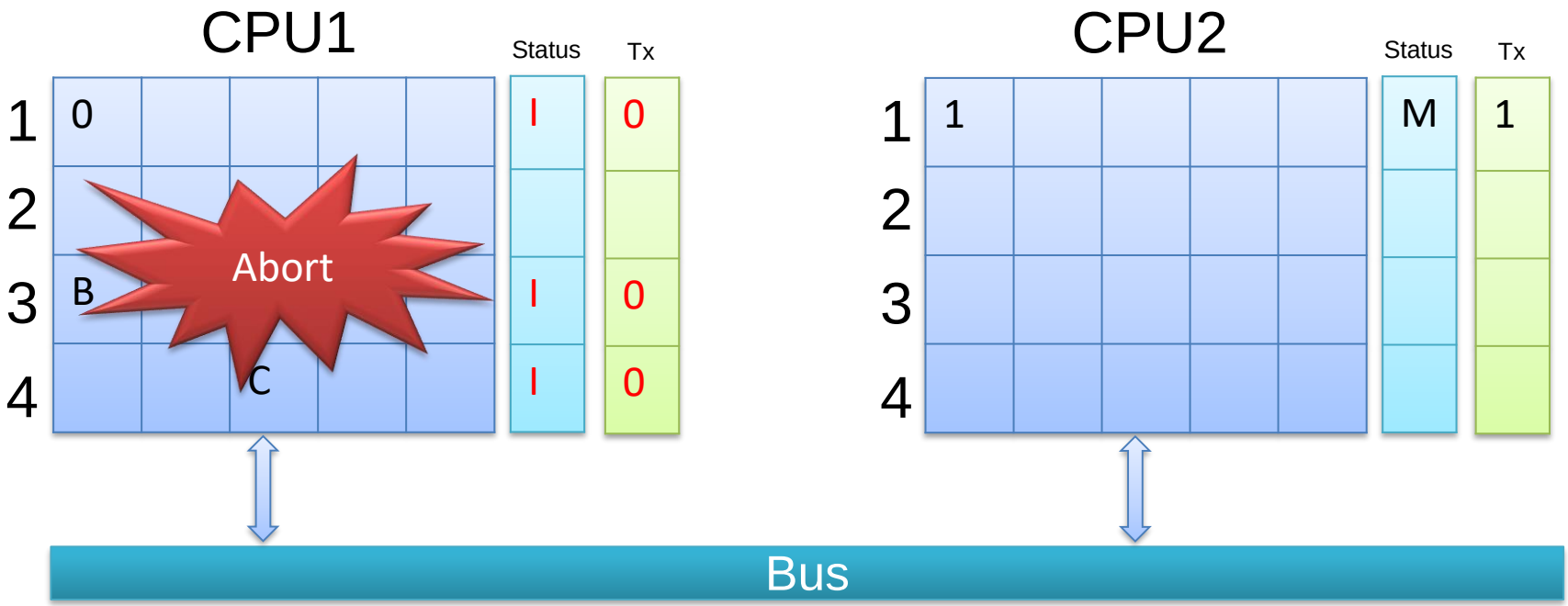
- Must define a software fallback path
  - Default is global lock



- _xbegin()
- if (read(lock)) == 1 then _xabort()
- do some work

Global Lock Fallback

- Must define a software fallback path
  - Default is global lock



- _xbegin()
- if (read(lock)) == 1 then _xabort()
- do some work

//non-transactionally
CAS(lock, 0 ,1)

Global Clock Fallback

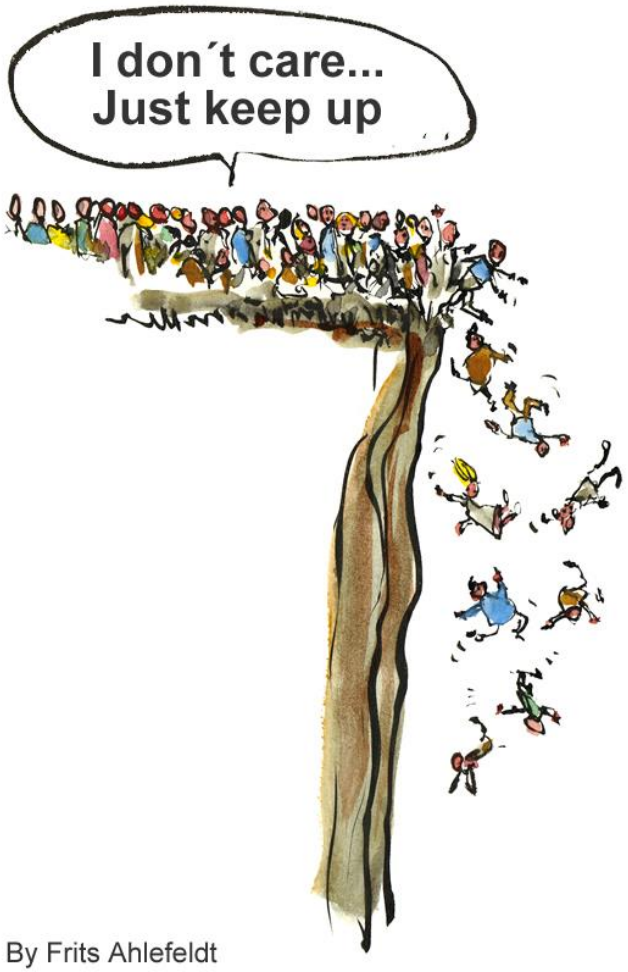
- Pros
  - Very easy to program
  - 100% safe
- Cons
  - Unnecessary aborts concurrent HTM transactions
  - Limit concurrency
  - Lemming effect
    - Cascading the failures in HTM, ending up with all transactions running in the global lock fallback path

Enhancing the Fallback Path

- Target
  - Enhance concurrency
    - Allow non-conflicting HTM transaction to run with the fallback path
  - Low overhead on HTM
    - An instrumentation is needed to communicate between HTM and the fallback path
      - E.g., The Global Lock
    - A heavy instrumentation would annul the advantages of HTM
  - Allow most transactions to commit in HW
    - No lemming effect

Lemming Effect

- Once a transaction take the fallback path → All remaining transactions follows
- How?



Lemming Effect

- One a transaction acquire the lock → all concurrent transactions abort

```
if (_xbegin() == _XBEGIN_STARTED) {
   read lock state
   if (lock taken) _xabort();
   work;
   _xend()
} else {
   lock->lock();
   work;
   lock->unlock();
}
```

Lock is in the read-set of the HTM transaction, then when it is locked in the fallback path (a write), a conflict aborts the HTM transacion

Lemming Effect

- Upon immediate retry, all HTM transactions will abort since the lock is already acquired

```
if (_xbegin() == _XBEGIN_STARTED) {
  read lock state
  if (lock taken) _xabort();
  work;
  _xend()
} else {
  lock->lock();
  work;
  lock->unlock();
}
```

Lock is already taken, so all HTM retries will abort immediately

Lemming Effect

- Hence, all HTM transaction will take the fallback path

```
if (_xbegin() == _XBEGIN_STARTED) {
   read lock state
   if (lock taken) _xabort();
   work;
   _xend()
} else {
   lock->lock();
   work;
   lock->unlock();
}
```

All HTM transaction will fallback to the global lock

Lemming Effect

- This behavior will continue until the next quiescent time

```
if (_xbegin() == _XBEGIN_STARTED) {
   read lock state
   if (lock taken) _xabort();
   work;
   _xend()
} else {
   lock->lock();
   work;
   lock->unlock();
}
```

Lemming Effect

- Eliminating lemming effect is easy
  - Wait for the lock to be free before retrying

```
while (lock taken) wait();
if (_xbegin() ==
   read lock stat
   if (lock taken)
   work;
   _xend()
} else {
   lock->lock();
   work;
   lock->unlock();
}
```

Wait until lock is free before trying an HTM transaction again

Lazy Subscription

- Move global lock checking to the end of a transaction

```
if (_xbegin() == _XBEGIN_STARTED) {
  //Do transaction work first
  work;
  //Check lock just before committing
  if (lock taken) _xabort();
  _xend()
} else {
  lock->lock();
  work;
  lock->unlock();
}
```

Lazy Subscription

- Why this is beneficial?
  - It allows more concurrency

```
if (_xbegin() == _XBEGIN_STARTED) {
  //Do transaction work first
  work;
  //Check lock just before committing
  if (lock taken) _xabort();
  _xend()
} else {
  lock->lock();
  work;
  lock->unlock();
}
```

Lazy Subscription

- Concurrent transactions will not immediately abort

```
if (_xbegin() == _XBEGIN_STARTED) {
  //Do transaction work first
  work;
  //Check lock just before committing
  if (lock taken) _xabort();
  _xend()
} else {
  lock->lock();
  work;
  lock->unlock();
}
```

Lazy Subscription

- New transactions will be allowed to start while the global lock is acquired

```
if (_xbegin() == _XBEGIN_STARTED) {
  //Do transaction work first
  work;
  //Check lock just before committing
  if (lock taken) _xabort();
  _xend()
} else {
  lock->lock();
  work;
  lock->unlock();
}
```

Lazy Subscription

- What about safety?
  - HTM transactions can access intermediate changes!

```
if (_xbegin() == _XBEGIN_STARTED) {
  //Do transaction work first
  work;
  //Check lock just before committing
  if (lock taken) _xabort();
  _xend()
} else {
  lock->lock();
  work;
  lock->unlock();
}
```

Lazy Subscription

- What about safety?
  - HTM is sandboxed and exceptions or infinite loops will the transaction

```
if (_xbegin() == _XBEGIN_STARTED) {
  //Do transaction work first
  work;
  //Check lock just before committing
  if (lock taken) _xabort();
  _xend()
} else {
  lock->lock();
  work;
  lock->unlock();
}
```

Lazy Subscription

- What about safety?
  - Unfortunately, current HTM implementation is not 100% sandboxed

```
if (_xbegin() == _XBEGIN_STARTED) {
  //Do transaction work first
  work;
  //Check lock just before committing
  if (lock taken) _xabort();
  _xend()
} else {
  lock->lock();
  work;
  lock->unlock();
}
```

Lazy Subscription Discussion

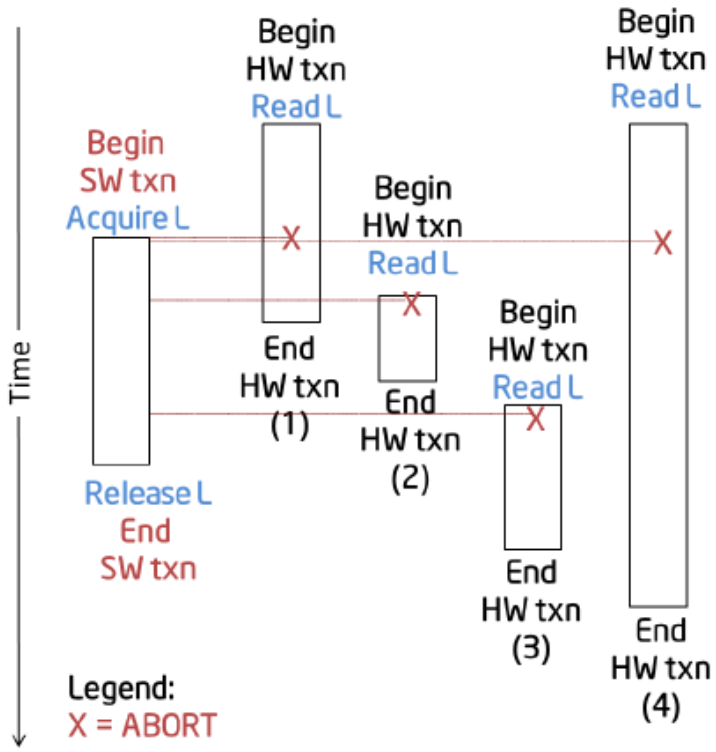


Figure 1. Obvious SGL Fallback implementation (E-SGL).

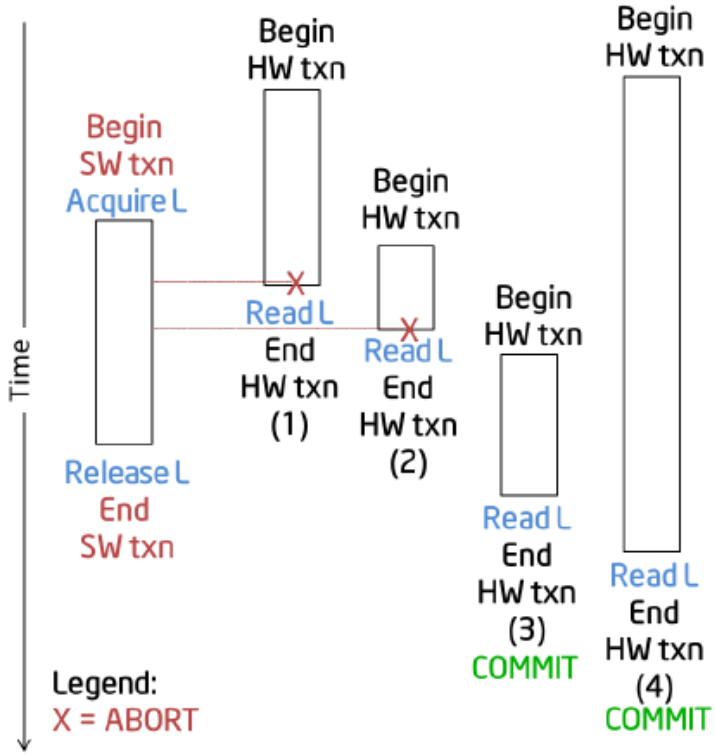


Figure 2. Lazy SGL (L-SGL).

Lazy Subscription Discussion

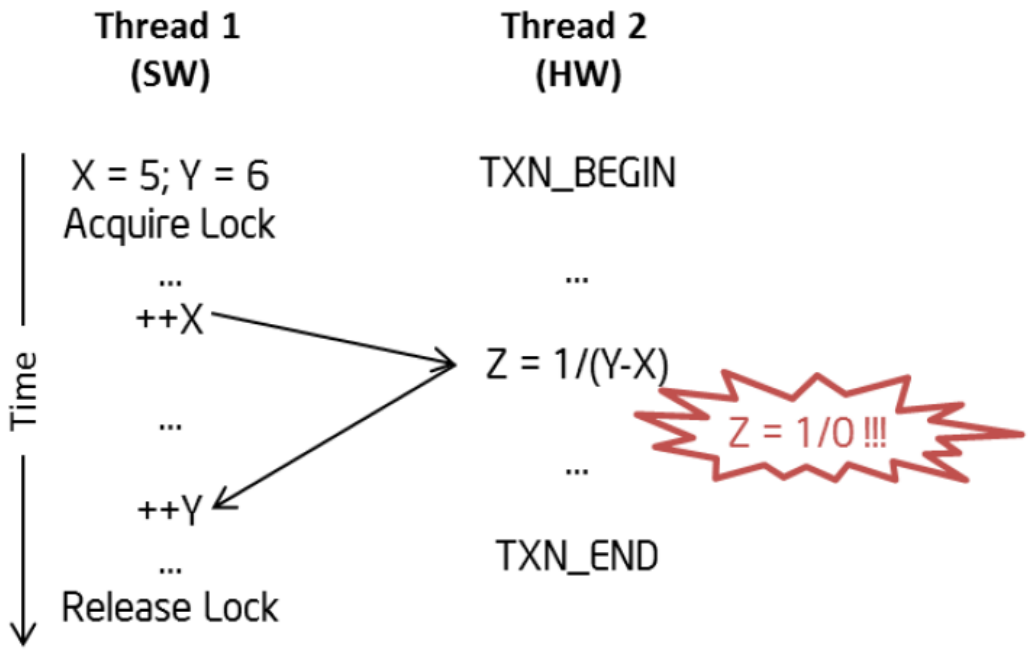


Figure 3. Inconsistent reads.

Lazy Subscription Discussion

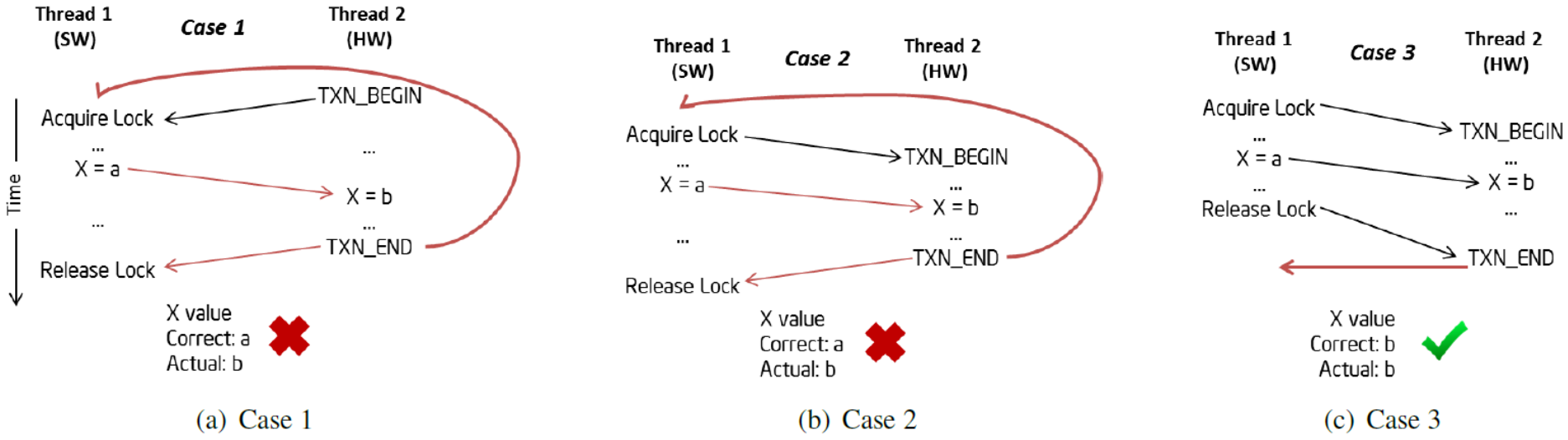


Figure 4. Correctness: Cases 1-3. Arrows denote the "happens-before" relation.

Lazy Subscription Discussion

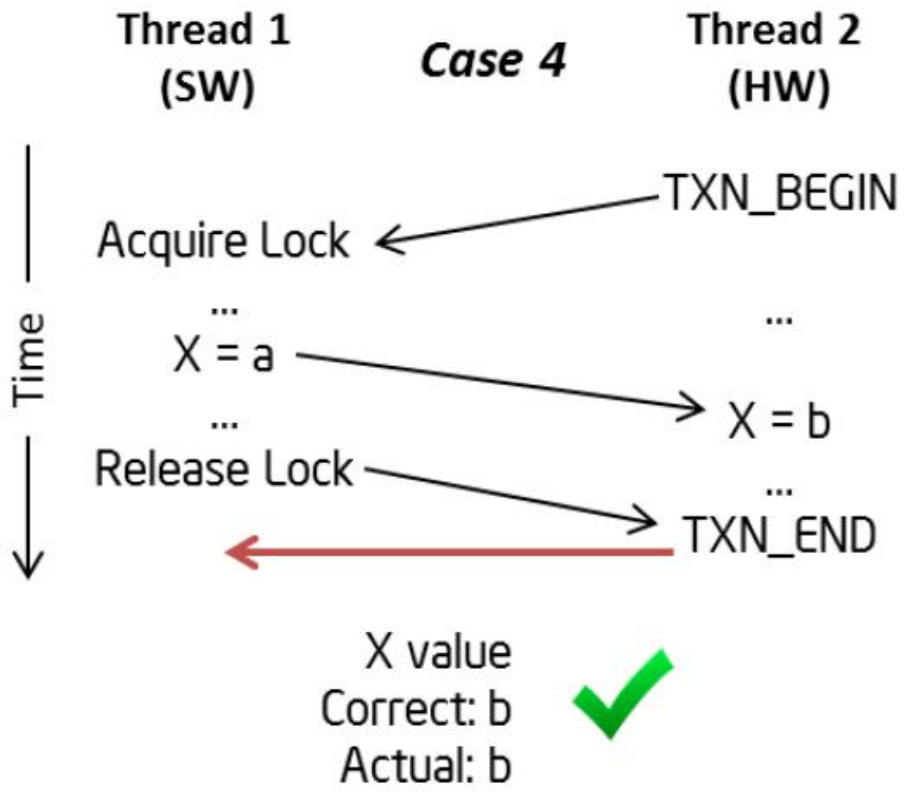


Figure 5. Correctness: Case 4.

Pitfalls of Lazy Subscription

- Observing inconsistent state

```
1  void (*method_table[2])() = {method1, method2};

3  int next_method = 0;

5  lock L;

7  void apply_next() {
8    acquire(L);
9    (*method_table[next_method])();
10   if (++next_method > 2)
11     next_method = 0;
12   release(L);
13 }
```

Pitfalls of Lazy Subscription

- Indirect branch

```
1  void (*method_table[2])() = {method1, method2};

3  int next_method = 0;

5  lock L;

7  void apply_next() {
8    acquire(L);
9    (*method_table[next_method])();
10   if (++next_method > 2)
11     next_method = 0;
12   release(L);
13 }
```

Pitfalls of Lazy Subscription

- Lock scribbling
- Subscribing to the wrong “lock”
- Self modifying code
- Corrupted return address

Pitfalls of Lazy Subscription

- Conclusion
  - Lazy Subscription is too risky on current architectures
  - HW must be modified first to support complete sandboxing

Retrying HTM Transactions

- When should we retry?
- How many retrials are enough?
- Should we try HTM in the first place?

Retrying HTM Transactions

Algorithm 1 TSX in GCC

```
 1: int attempts ← 2
 2: int status ← XBEGIN
 3: if status ≠ ok then
 4:     if attempts = 0 then
 5:         acquire(globalLock)
 6:     else
 7:         attempts−−
 8:         goto line 2
 9: if is_locked(globalLock)
10:     XABORT
11: ▷ ...transactional code
12: if attempts = 0 then
13:     release(globalLock)
14: else
15:     XEND
```

Abort code
`retry`: Transient failure
`conflict`: Contention to data
`capacity`: Exceeded cache
`explicit`: XABORT invoked
`other`: Faults, preemption, ...

Fig. 2. Error codes in TSX.

*Self-Tuning Intel Transactional Synchronization Extensions

Retrying HTM Transactions

Algorithm 1 TSX in GCC

```
 1: int attempts ← 2
 2: int status ← XBEGIN
 3: if status ≠ ok then
 4:     if attempts = 0 then
 5:         acquire(globalLock)
 6:     else
 7:         attempts−−
 8:         goto line 2
 9: if is_locked(globalLock)
10:     XABORT
11: ▷ ...transactional code
12: if attempts = 0 then
13:     release(globalLock)
14: else
15:     XEND
```

This code ignores the status code and just retry without considering the reason of the abort

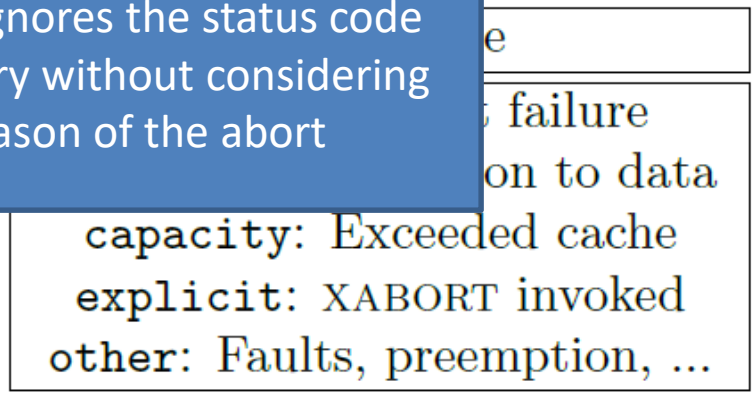


Fig. 2. Error codes in TSX.

*Self-Tuning Intel Transactional Synchronization Extensions

Retrying HTM Transactions

Algorithm 2 HEURISTIC based approach for TSX.

```
1: int attempts ← 5
   ▷ avoid the lemming effect
2: while(is_locked(global-lock)) do pause                    ▷ x86 instruction
3: int status ← XBEGIN
4: if status ≠ ok then
5:     if attempts = 0 then
6:         acquire(global-lock)
7:     else
8:         if status = explicit ∨ status = other then
9:             attempts ← attempts - 1                       ▷ skipped if transient
10:        else if status = capacity then
11:            attempts ← 0                                  ▷ give up, likely that it always fails
12:        goto line 2
13: ▷ ...code to run in transaction
14: if attempts = 0 then
15:     release(global-lock)
16: else
17:     if is_locked(global-lock) then
18:         XABORT                                           ▷ check for concurrent pessimistic thread
19:     XEND
```

*Self-Tuning Intel Transactional Synchronization Extensions

Retrying HTM Transactions

Algorithm 2 HEURISTIC based approach for TSX.

```
1: int attempts ← 5
   ▷ avoid the lemming effect
2: while(is_locked(global-lock)) do pause                    ▷ x86 instruction
3: int status ← XBEGIN
4: if status ≠ ok then
5:     if attempts = 0 then
6:         acquire(global-lock)
7:     else
8:         if status = explicit ∨ status =
9:             attempts ← attempts - 1                       ▷ skipped if transient
10:        else if status = capacity then
11:            attempts ← 0                                  ▷ give up, likely that it always fails
12:        goto line 2
13: ▷ ...code to run in transaction
14: if attempts = 0 then
15:     release(global-lock)
16: else
17:     if is_locked(global-lock) then
18:         XABORT                                           ▷ check for concurrent pessimistic thread
19:     XEND
```

Avoid lemming effect by waiting for global lock to be free before attempting an HTM transaction

*Self-Tuning Intel Transactional Synchronization Extensions

Retrying HTM Transactions

Algorithm 2 HEURISTIC based approach for TSX.

```
1: int attempts ← 5
   ▷ avoid the lemming effect
2: while(is_locked(global-lock)) do pause                    ▷ x86 instruction
3: int status ← XBEGIN
4: if status ≠ ok then
5:     if attempts = 0 then
6:         acquire(global-lock)
7:     else
8:         if status = explicit ∨ status = other then
9:             attempts ← attempts - 1                       ▷ skipped if transient
10:        else if status = capacity then
11:            attempts ← 0                                  ▷ give up, likely that it always fails
12:        goto line 2
13: ▷ ...code to run in transaction
14: if attempts = 0 then
15:     release(global-lock)
16: else
17:     if is_locked(global-lock) then
18:         XABORT                                           ▷ check for concurrent pessimistic thread
19:     XEND
```

If the abort is explicit or for "other" reasons, consume one attempt

*Self-Tuning Intel Transactional Synchronization Extensions

Retrying HTM Transactions

Algorithm 2 HEURISTIC based approach for TSX.

```
1: int attempts ← 5
   ▷ avoid the lemming effect
2: while(is_locked(global-lock)) do pause                    ▷ x86 instruction
3: int status ← XBEGIN
4: if status ≠ ok then
5:     if attempts = 0 then
6:         acquire(global-lock)
7:     else
8:         if status = explicit ∨ status = other then
9:             attempts ← attempts - 1                       ▷ skipped if transient
10:        else if status = capacity then
11:            attempts ← 0                                  ▷ that it always fails
12:        goto line 2
13: ▷ ...code to run in transaction
14: if attempts = 0 then
15:     release(global-lock)
16: else
17:     if is_locked(global-lock) then
18:         XABORT                                           ▷ check for concurrent pessimistic thread
19:     XEND
```

Capacity aborts consume all attempts as there is no benefit in retrying a transaction that doesn't fit in HTM

*Self-Tuning Intel Transactional Synchronization Extensions

Retrying HTM Transactions

Algorithm 2 HEURISTIC based approach for TSX.

```
1: int attempts ← 5
   ▷ avoid the lemming effect
2: while(is_locked(global-lock)) do pause                    ▷ x86 instruction
3: int status ← XBEGIN
4: if status ≠ ok then
5:     if attempts = 0 then
6:         acquire(global-lock)
7:     else
8:         if status = explicit ∨ status = other then
9:             attempts ← attempts - 1
10:        else if status = capacity then
11:            attempts ← 0
12:        goto line 2
13: ▷ ...code to run in transaction
14: if attempts = 0 then
15:     release(global-lock)
16: else
17:     if is_locked(global-lock) then
18:         XABORT                    ▷ check for concurrent pessimistic thread
19:     XEND
```

Conflict & Retry aborts do NOT consume any attempts as these are normal expected TM aborts

*Self-Tuning Intel Transactional Synchronization Extensions

Retrying HTM Transactions

Algorithm 2 HEURISTIC based approach for TSX.

```
1: int attempts ← 5
   ▷ avoid the lemming effect
2: while(is_locked(global-lock)) do pause                    ▷ x86 instruction
3: int status ← XBEGIN
4: if status ≠ ok then
5:     if attempts = 0 then
6:         acquire(global-lock)
7:     else
8:         if status = explicit ∨ sta
9:             attempts ← attempts - 1                       ▷ skipped if transient
10:        else if status = capacity
11:            attempts ← 0                                  ▷ give up, likely that it always fails
12:        goto line 2
13: ▷ ...code to run in transaction
14: if attempts = 0 then
15:     release(global-lock)
16: else
17:     if is_locked(global-lock) then
18:         XABORT                                           ▷ check for concurrent pessimistic thread
19:     XEND
```

When all attempts are consumed, acquire the global lock

*Self-Tuning Intel Transactional Synchronization Extensions

Retrying HTM Transactions

Algorithm 2 HEURISTIC based approach for TSX.

```
1: int attempts ← 5
   ▷ avoid the lemming effect
2: while(is_locked(global-lock)) do pause                    ▷ x86 instruction
3: int status ← XBEGIN
4: if status ≠ ok then
5:     if attempts = 0 then
6:         acquire(global-lock)
7:     else
8:         if status = explicit ∨ status = other then
9:             attempts ← attempts - 1                       ▷ skipped if transient
10:        else if status = capacity then
11:            attempts ← 0                                  ▷ give up, likely that it always fails
12:        goto line 2
13: ▷ ...code to run in transactio
14: if attempts = 0 then
15:     release(global-lock)
16: else
17:     if is_locked(global-lock) th
18:         XABORT                                           or concurrent pessimistic thread
19:     XEND
```

If attempt are zero, then I must have acquired the lock and need to unlock it

*Self-Tuning Intel Transactional Synchronization Extensions

Retrying HTM Transactions

Algorithm 2 HEURISTIC based approach for TSX.

```
1: int attempts ← 5
   ▷ avoid the lemming effect
2: while(is_locked(global-lock)) do pause                    ▷ x86 instruction
3: int status ← XBEGIN
4: if status ≠ ok then
5:     if attempts = 0 then
6:         acquire(global-lock)
7:     else
8:         if status = explicit ∨ status = other then
9:             attempts ← attempts - 1                       ▷ skipped if transient
10:        else if status = capacity then
11:            attempts ← 0                                  ▷, likely that it always fails
12:        goto line 2
13: ▷ ...code to run in transaction
14: if attempts = 0 then
15:     release(global-lock)
16: else
17:     if is_locked(global-lock) then
18:         XABORT                                           ▷ check for concurrent pessimistic thread
19:     XEND
```

Lazy subscription to the lock and _xend if lock is free

*Self-Tuning Intel Transactional Synchronization Extensions

Retrying HTM Transactions

Algorithm 3 TUNER adaptive configuration.

```
 1: int ucbBelief ← ▷ last configuration used
 2: int attempts ← ▷ last configuration used
 3: if reoptimize() then
 4:    long initCycles ← obtainRDTSC()
 5: while is_locked(global-lock) do pause
 6: int status ← XBEGIN
 7: if status ≠ ok then
 8:    if attempts = 0 then
 9:       if reoptimize() then tuneAttempts(ucbBelief)
10:       acquire(global-lock)
11:    else
12:       if status = capacity then
13:          ▷ set attempts according to ucbBelief
14:       else if status = explicit ∨ status = other then
15:          attempts ← attempts - 1
16:       goto line 5
17: ▷ ...code to run in transaction
18: if attempts = 0 then
19:    release(global-lock)
20: else
21:    if is_locked(global-lock) then XABORT
22:    XEND
23: if reoptimize() then
24:    long totalCycles ← obtainRDTSC() - initCycles
25:    ucbBelief ← UCB(totalCycles)                    ▷ rules of Section 4.2.
26:    attempts ← GRAD(totalCycles)                    ▷ rules of Section 4.3.
```

*Self-Tuning Intel Transactional Synchronization Extensions

Retrying HTM Transactions

Algorithm 3 TUNER adaptive configuration.

```
 1: int ucbBelief ← ▷ last configuration used
 2: int attempts ← ▷ last configuration used
 3: if reoptimize() then
 4:     long initCycles ← obtainRDTSC()
 5: while is_locked(global-lock) do pause
 6: int status ← XBEGIN
 7: if status ≠ ok then
 8:     if attempts = 0 then
 9:         if reoptimize() then tuneAttempts(ucbBelief)
10:         acquire(global-lock)
11:     else
12:         if status = capacity then
13:             ▷ set attempts according to ucbBelief
14:         else if status = explicit ∨ status = other then
15:             attempts ← attempts - 1
16:         goto line 5
17: ▷ ...code to run in transaction
18: if attempts = 0 then
19:     release(global-lock)
20: else
21:     if is_locked(global-lock) then XABORT
22:     XEND
23: if reoptimize() then
24:     long totalCycles ← obtainRDTSC() - initCycles
25:     ucbBelief ← UCB(totalCycles)                    ▷ rules of Section 4.2.
26:     attempts ← GRAD(totalCycles)                    ▷ rules of Section 4.3.
```

Number of attempts is adaptive

*Self-Tuning Intel Transactional Synchronization Extensions

Retrying HTM Transactions

Algorithm 3 TUNER adaptive configuration.

```
 1: int ucbBelief ← ▷ last configuration used
 2: int attempts ← ▷ last configuration used
 3: if reoptimize() then
 4:    long initCycles ← obtainRDTSC()
 5: while is_locked(global-lock) do pause
 6: int status ← XBEGIN
 7: if status ≠ ok then
 8:    if attempts = 0 then
 9:       if reoptimize() then tuneAttempts(ucbBelief)
10:       acquire(global-lock)
11:    else
12:       if status = capacity then
13:          ▷ set attempts according to ucbBelief
14:       else if status = explicit ∨ status = other the
15:          attempts ← attempts - 1
16:       goto line 5
17: ▷ ...code to run in transaction
18: if attempts = 0 then
19:    release(global-lock)
20: else
21:    if is_locked(global-lock) then XABORT
22:    XEND
23: if reoptimize() then
24:    long totalCycles ← obtainRDTSC() - initCycles
25:    ucbBelief ← UCB(totalCycles)
26:    attempts ← GRAD(totalCycles)
                                          ▷ rules of Section 4.2.
                                          ▷ rules of Section 4.3.
```

Consuming all attempts on capacity aborts is not always the best option

*Self-Tuning Intel Transactional Synchronization Extensions

Global Lock Fallback

- Even with all these optimizations
  - Once the global clock is acquired, no concurrency is allowed
  - A long running transaction acquiring the global clock will block the progress of all other transactions
    - Without having any mutual conflicts
- Can we do better?

Hybrid TM

- Mix HTM with STM
- Allows both HTM and STM to work concurrently (if possible)
  - Some proposals (e.g., PhaseTM) work in phases where all transactions run in HTM or STM
- Requires some instrumentation inside HTM path

Hybrid TM

- One naïve solution is to use per location metadata (Orec based STM)
  - For each read in HTM, check the associated metadata
  - Writes must acquire the associated locks to inform STM transactions of that write

Hybrid TM

- One naïve solution is to use per location metadata (Orec based STM)
  - The resulting algorithm performance is very bad
    - Consumes a lot of the limited HTM resources
      - Extra reads and writes due to metadata manipulation
    - Most HTM transactions abort
      - False aborts due to space limitation (metadata access)
      - False conflict due to global shared metadata access (e.g., global clock)
- What's the solution?

Hybrid TM

- Minimal instrumentation
- Minimal communication between HTM and STM
- For example, do NOT use Orecs

Hybrid TM

- Minimal instrumentation
- Minimal communication between HTM and STM
- For example, do NOT use Orecs
  - Best candidate from algorithms we learned

NOrec

Hybrid NOrec

- NOrec
  - A single shared global lock is used
    - Minimal communication between STM and HTM
  - Value-based validation
    - No per-location metadata
    - Minimal instrumentation inside HTM

Hybrid NOrec

- NOrec STM Algorithm

```
1   padded unsigned seqlock

3   thread local unsigned snapshot
4   thread local ReadSet reads
5   thread local WriteSet writes


6   SW_VALIDATE
7     snapshot = seqlock
8     if (snapshot & 1)
9       goto 7
10    foreach (addr, val) in reads
11      if (*addr != val)
12        SW_ABORT
13    if (snapshot != seqlock)
14      goto 7
```

```
15  SW_BEGIN
16    snapshot = seqlock
17    if (snapshot & 1)
18      goto 16


19  SW_COMMIT
20    if (writes.empty())
21      return
22    while (!CAS(&seqlock, snapshot,
          snapshot + 1))
23      SW_VALIDATE
24    foreach (addr, val) in writes
25      *addr = val
26    seqlock = seqlock + 1
27    reads.reset(), writes.reset()
```

```
28  SW_READ(addr)
29    if (addr in writes)
30      return writes.find(addr)
31    val = *addr
32    if (snapshot != seqlock)
33      SW_VALIDATE
34      goto 31
35    reads.append(addr, val)
36    return val

38  SW_WRITE(addr, val)
39    writes.append(addr, val)

41  SW_ABORT
42    reads.reset(), writes.reset()
43    /* restart transaction */
```

*Hybrid NOrec: A Case Study in the Effectiveness of Best Effort Hardware Transactional Memory

Hybrid NOrec

- Simply, subscribe to NOrec global lock at the beginning
- Increment NOrec global lock at the end to notify STM transaction of the change

```
 5   HW_POST_BEGIN
 6     if (seqlock & 1)
 7       while (true) // await abort

 9   HW_PRE_COMMIT
10     seqlock = seqlock + 2
```

Hybrid NOrec

- Simply, subscribe to NOrec global lock at the beginning
- Increment NOrec global lock at the end to notify STM transaction of the change

```
5   HW_POST_BEGIN
6     if (seqlock & 1)
7       while (true)

9   HW_PRE_COMM
10    seqlock = seqlock + 2
```

All HTM reading seqlock will abort one this line is executed!

Hybrid NOrec

- This improvement, reduces HTM-HTM false conflicts due to writing to seqlock
  - Shorter window of conflicts

```
1   padded unsigned seqlock
2   padded unsigned counter


5   HW_POST_BEGIN
6     if (seqlock & 1)
7       while (true) // await abort

9   HW_PRE_COMMIT
10    counter = counter + 1
```

Hybrid NOrec

- This improvement, reduces HTM-HTM false conflicts due to writing to seqlock
  - Shorter window of conflicts

```
1   padded unsigned seqlock
2   padded unsigned counter


5   HW_POST_BEGIN
6     if (seqlock & 1)
7       while (true) //

9   HW_PRE_COMMIT
10    counter = counter + 1
```

With using two locations for communication. Only HTM transactions executing this line concurrently will conflict with each other

Hybrid NOrec

- This improvement, eliminates HTM-HTM false conflicts due to writing to same counter

```
1   padded unsigned seqlock
2   padded unsigned counter[]
3   thread local unsigned id

5   HW_POST_BEGIN
6     if (seqlock & 1)
7       while (true) // await abort

9   HW_PRE_COMMIT
10    counter[id] = counter[id] + 1
```

Hybrid NOrec

- This improvement, eliminates HTM-HTM false conflicts due to writing to same counter

```
1   padded unsigned seqlock
2   padded unsigned counter[]
3   thread local unsigned id

5   HW_POST_BEGIN
6     if (seqlock & 1)
7       while (true) /

9   HW_PRE_COMMIT
10    counter[id] = counter[id] + 1
```

Now, each thread has its own location to indicate its writes to STM transactions, so no more false conflicts due to this communication

Hybrid NOrec

- This improvement, eliminates HTM-HTM false conflicts due to writing to same counter

```
1   padded unsigned seqlock
2   padded unsigned counter[]
3   thread local unsigned id

5   HW_POST_BEGIN
6     if (seqlock & 1)
7       while (true) /

9   HW_PRE_COMMIT
10    counter[id] = counter[id] + 1
```

BUT, software tx_commit will have more overhead checking all these counters