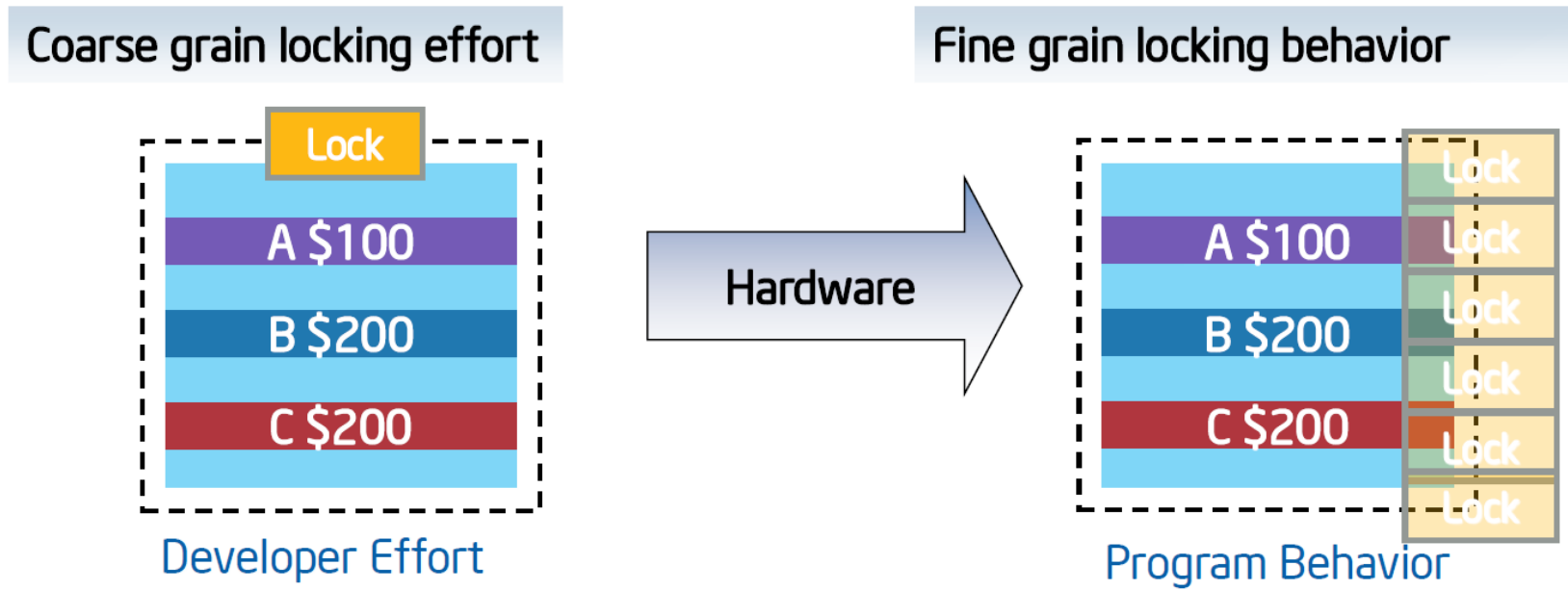
Intel TSX (Transactional Synchronization Extensions)

Mike Dai Wang and Mihai Burcea

What We Want...



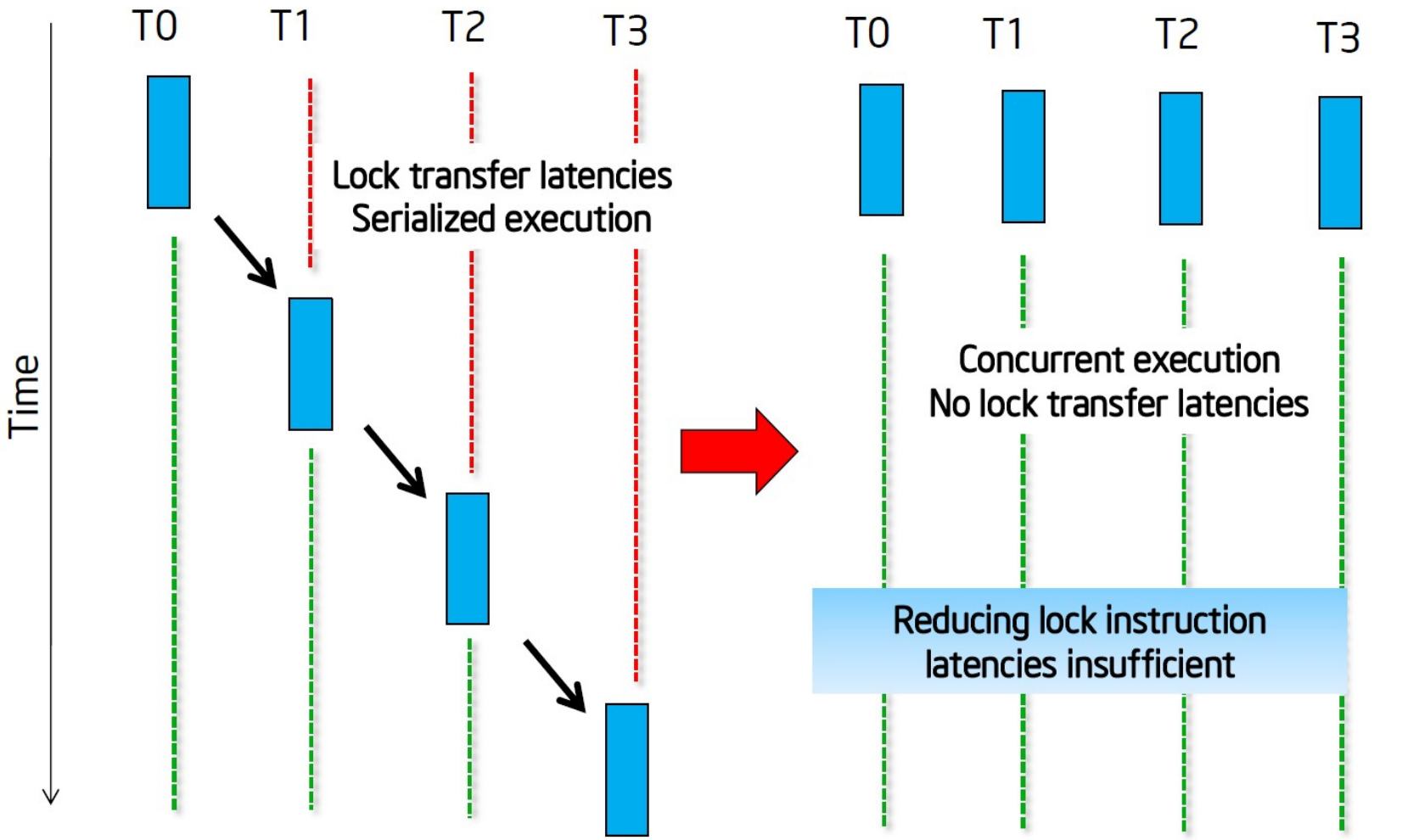
Developer uses coarse grain lock

Hardware elides the lock to expose concurrency

- Multiple threads don't serialize on the lock
- Hardware automatically detects **real** data conflicts

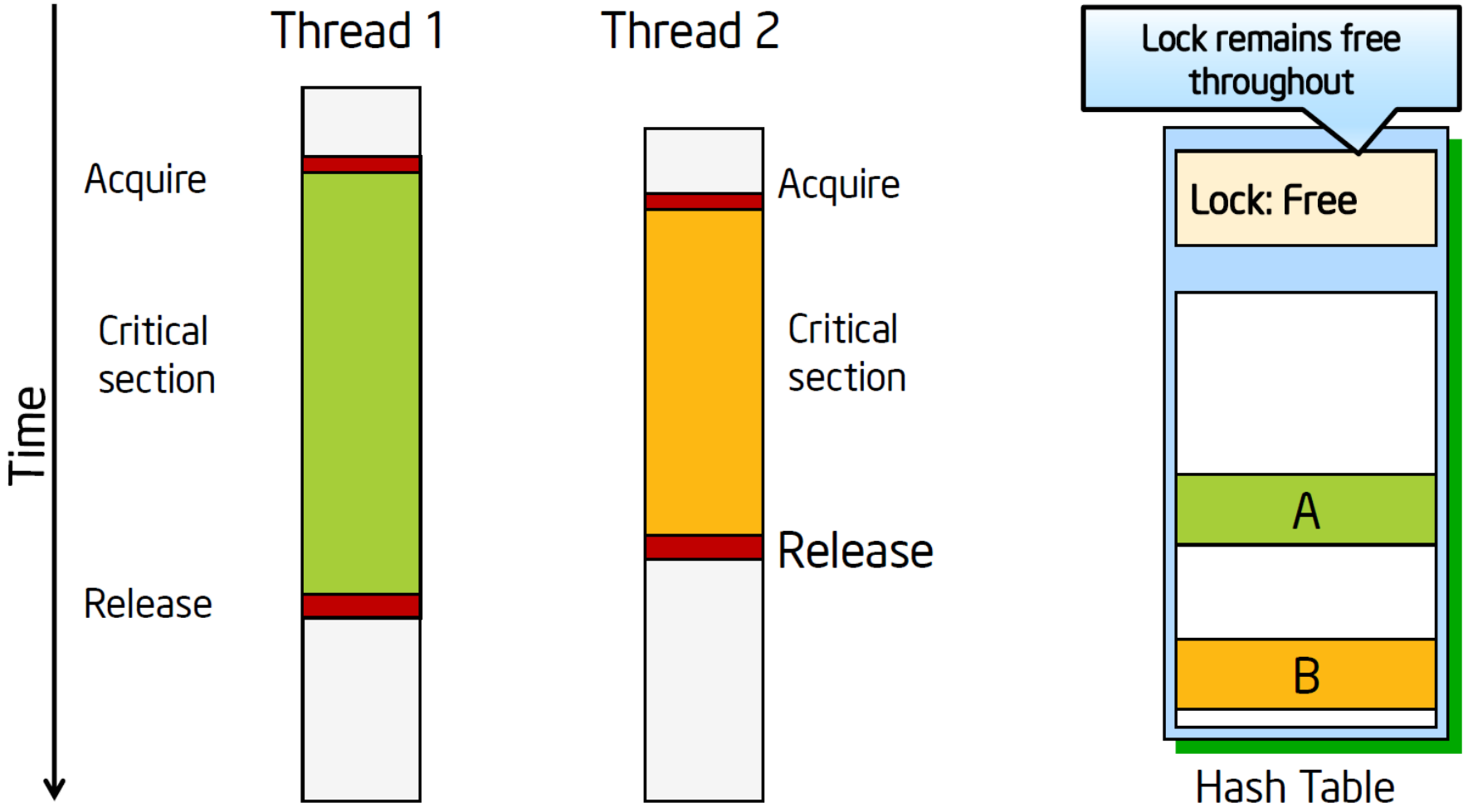
Lock Elision: Fine Grain Behavior at Coarse Grain Effort

Benefit of Lock Elision



Exposes Concurrency & Eliminates Unnecessary Communication

A Canonical Intel® TSX Execution



No Serialization and No Communication if No Data Conflicts

Intel® TSX Interfaces for Lock Elision

Hardware Lock Elision (HLE) – XACQUIRE/XRELEASE

- Software uses legacy compatible hints to identify critical section. Hints ignored on hardware without TSX
- Hardware support to execute transactionally without acquiring lock
- Abort causes a re-execution without elision
- Hardware manages all architectural state

Restricted Transactional Memory (RTM) – XBEGIN/XEND

- Software uses new instructions to specify critical sections
- Similar to HLE but flexible interface for software to do lock elision
- Abort transfers control to target specified by XBEGIN operand
- Abort information returned in a general purpose register (EAX)

XTEST and XABORT – Additional instructions

Flexible and Easy To Use

Intel® TSX Operational Aspects

1. **Identify and elide**
   - Identify critical section, start transactional execution
   - Elide locks, keep them available to other threads
2. **Execute transactionally**
   - Manage all transactional state updates
3. **Detect conflicting memory accesses**
   - Track data accesses, check for conflicts from other threads
4. **Abort or commit**
   - Abort discards all transactional updates
   - Commit makes transactional updates instantaneously visible

Identify and Elide: HLE

Hardware support to elide multiple locks

- Hardware elision buffer manages actively elided locks
- XACQUIRE/XRELEASE allocate/free elision buffer entries
- Skips elision without aborting if no free entry available

Hardware treats XACQUIRE/XRELEASE as hints

- Functionally correct even if hints used improperly
- Hardware checks if locks meet requirements for elision
- May expose latent bugs and incorrect timing assumptions

Hardware Management of Elision Enables Ease of Use

Implementation specific to the next general Intel® microarchitecture code name Haswell

Execute Transactionally

- **State updated during transactional execution**
  - State includes registers and memory
  - Hardware recovers register and memory state on aborts
- **Hardware manages all transactional updates**
  - Other threads cannot observe any intermediate updates
  - If lock elision cannot succeed, hardware restarts execution
  - Hardware discards all intermediate updates prior to restart

Software Does Not Worry About State Recovery

Intel® TSX Interface: HLE

```
        mov eax, 1
Try:    lock xchg mutex, eax
        cmp eax, 0
        jz Success
Spin:   pause
        cmp mutex, 1
        jz Spin
        jmp Try
```

```
        mov eax, 1
Try:    xacquire lock xchg mutex, eax
        cmp eax, 0
        jz Success
Spin:   pause
        cmp mutex, 1
        jz Spin
        jmp Try
```

Enter HLE execution

If lock not free, execution will abort either early (if pause used) or when lock gets free

Library

Application

```
acquire_lock (mutex)
; do critical section
; function calls,
; memory operations, …
release_lock (mutex)
```

Commit HLE execution

```
mov mutex, 0
```

```
xrelease mov mutex, 0
```

Legacy Compatible Enabling Within Libraries

Code example for illustration purpose only

Identify and Elide: HLE

mutex value	*self*	*others*
mutex:	0	0
mutex:	1	0

```
xacquire lock cmpxchg mutex, ebx
```

- Hardware executes XACQUIRE hint
- Hardware elides acquire write to mutex
- Hardware starts transactional execution

```
mov ecx, mutex
```

- Reading mutex in critical section sees last value written (1)
- Other threads reading see free value (0)

mutex:	1	0
mutex:	0	0

```
xrelease mov mutex, 0
```

- Hardware executes XRELEASE hint
- Hardware elides release write to mutex
- Hardware commits transactional execution

Hardware Automatically Manages Elided Locks

Intel® TSX Interface: RTM

```
        mov eax, 1
Try:    lock xchg mutex, eax
        cmp eax, 0
        jz Success
Spin:   pause
        cmp mutex, 1
        jz Spin
        jmp Try
```

Augment conventional lock libraries to support RTM-based Lock Elision

Library

Application

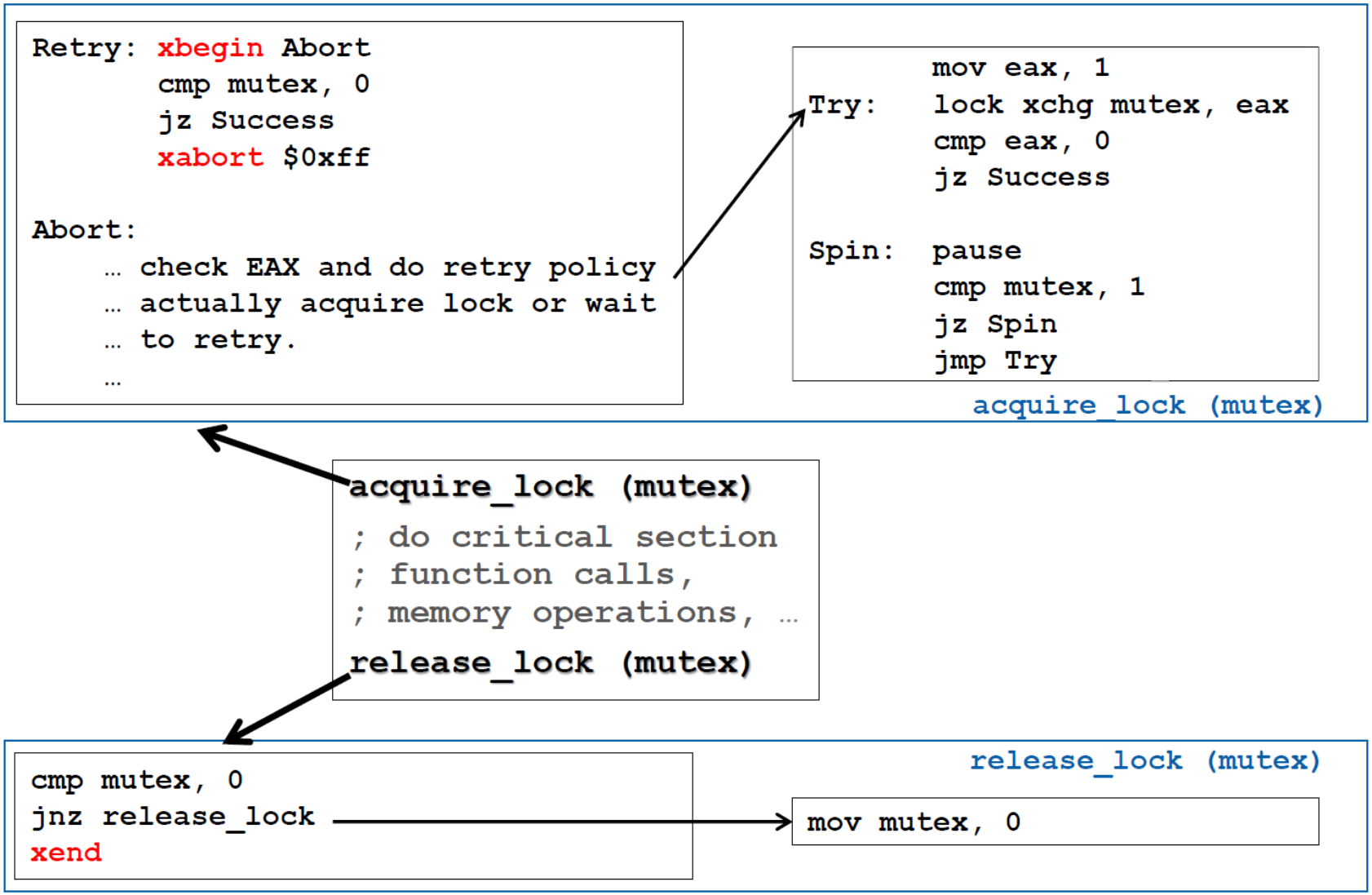
```
acquire_lock (mutex)
; do critical section
; function calls,
; memory operations, …
release_lock (mutex)
```

```
mov mutex, 0
```

Lock Elision using RTM Also Enabled Inside Libraries

Code example for illustration purpose only

Intel® TSX Interface: RTM



Code example for illustration purpose only

Intel® TSX Interface: RTM

```
Retry: xbegin Abort
       cmp mutex, 0
       jz Success
       xabort $0xff

Abort:
    … check EAX and do retry policy
    … actually acquire lock or wait
    … to retry.
    …
```

… Enter RTM execution, Abort is fallback path

… Check to see if mutex is free

… Abort transactional execution if mutex busy

… Fallback path in software

… Retry RTM or explicitly acquire mutex

```
acquire_lock (mutex)
; do critical section
; function calls,
; memory operations, …
release_lock (mutex)
```

```
cmp mutex, 0
jnz release_lock
xend
```

… Mutex not free →was not an RTM execution

… Commit RTM execution

Code example for illustration purpose only

Identify and Elide: RTM

mutex value	*self*	*others*
mutex:	0	0
mutex:	0	0

`xbegin <fallback_path>`

- Hardware executes XBEGIN
- Hardware starts transactional execution
- Software checks for a free mutex, skips acquire

`mov ecx, mutex`

- Reading mutex in critical section sees 0
- Other threads reading see free value (0)

mutex:	0	0
mutex:	0	0

`xend`

- Hardware executes XEND
- Hardware commits transactional execution

RTM Provides Increased Flexibility for Software

Execute Transactionally

Hardware manages all transactional updates

- Other threads cannot observe any intermediate updates
- If transactional execution doesn't succeed, hardware restarts execution
- Hardware discards all intermediate updates prior to restart

Transactional abort

- Occurs when abort condition is detected
- Hardware discards all transactional updates

Transactional commit

- Hardware makes transactional updates visible instantaneously
- No cross-thread/core/socket coordination required

Software Does Not Worry About State Recovery

Implementation specific to the next general Intel® microarchitecture code name Haswell

Execute Transactionally – Memory

Buffering memory writes

- Hardware uses L1 cache to buffer transactional writes
  - Writes not visible to other threads until after commit
  - Eviction of transactionally written line causes abort
- Buffering at cache line granularity

Sufficient buffering for typical critical sections

- Cache associativity can occasionally be a limit
- Software always provides fallback path in case of aborts

Hardware Manages All Transactional Writes

Implementation specific to the next general Intel® microarchitecture code name Haswell

Detect Conflicts

Read and write addresses for conflict checking

- Tracked at cache line granularity using physical address
- L1 cache tracks addresses written to in transactional region
- L1 cache tracks addresses read from in transactional region
  - Cache may evict address without loss of tracking

Data conflicts

- Occurs if at least one request is doing a write
- Detected at cache line granularity
- Detected using existing cache coherence protocol
- Abort when conflicting access detected

Hardware Automatically Detects Conflicting Accesses

Implementation specific to the next general Intel® microarchitecture code name Haswell

Intel® TSX Usage Environment

- **Available in all x86 modes**
- **Some instructions and events may cause aborts**
  - Uncommon instructions, interrupts, faults, etc.
  - Always functionally safe to use any instruction
- **Software must provide a non-transactional path**
  - HLE: Same software code path executed without elision
  - RTM: Software fallback handler must provide alternate path

Architected for Typical Lock Elision Usage

IDF2012
INTEL DEVELOPER FORUM

Focus: Simplify Developer Enabling

- **Easy to start using Intel® TSX**
  - Simple and clean instruction set minimizes software changes
    - Can be hidden in software synchronization libraries
    - Supports nesting critical sections
  - Minimizes implementation-specific causes for aborts
    - Micro-architectural events such as branch mispredicts, cache misses, TLB misses, etc. do not cause aborts
    - No explicit limit on number of instructions inside critical section
- **Simplify decision process of when to use**
  - Designed to support typical critical sections
  - Competitive to typical uncontended critical sections

Developer Focused Architecture and Design

Implementation specific to the next general Intel® microarchitecture code name Haswell
Intel® Transactional Synchronization Extensions (Intel® TSX)

Core Cache Size/Latency/Bandwidth

Metric	Nehalem	Sandy Bridge	Haswell
L1 Instruction Cache	32K, 4-way	32K, 8-way	32K, 8-way
L1 Data Cache	32K, 8-way	32K, 8-way	32K, 8-way
Fastest Load-to-use	4 cycles	4 cycles	4 cycles
Load bandwidth	16 Bytes/cycle	32 Bytes/cycle (banked)	64 Bytes/cycle
Store bandwidth	16 Bytes/cycle	16 Bytes/cycle	32 Bytes/cycle
L2 Unified Cache	256K, 8-way	256K, 8-way	256K, 8-way
Fastest load-to-use	10 cycles	11 cycles	11 cycles
Bandwidth to L1	32 Bytes/cycle	32 Bytes/cycle	64 Bytes/cycle
L1 Instruction TLB	4K: 128, 4-way 2M/4M: 7/thread	4K: 128, 4-way 2M/4M: 8/thread	4K: 128, 4-way 2M/4M: 8/thread
L1 Data TLB	4K: 64, 4-way 2M/4M: 32, 4-way 1G: fractured	4K: 64, 4-way 2M/4M: 32, 4-way 1G: 4, 4-way	4K: 64, 4-way 2M/4M: 32, 4-way 1G: 4, 4-way
L2 Unified TLB	4K: 512, 4-way	4K: 512, 4-way	4K+2M shared: 1024, 8-way

All caches use 64-byte lines

Intel® Microarchitecture (Haswell); Intel® Microarchitecture (Sandy Bridge); Intel® Microarchitecture (Nehalem)

Example: toy banking application with RTM

- Code written and tested in emulator, not in hardware!
  - Hardware did not exist yet
- Uses STL *std::vector<int>* as an array of accounts
- Perform a few operations on two accounts concurrently
- Protect each operation by protecting its entire scope

Classic lock-based implementation

```
  {
    std::cout << "open new account" << std::endl;
    TransactionScope guard; // protect everything in this scope
    Accounts.push_back(0);
  }
  {
    std::cout << "put 100 units into account 0" <<std::endl;
    TransactionScope guard; // protect everything in this scope
    Accounts[0] += 100; // atomic update due to RTM
  }


class TransactionScope {
  SimpleSpinLock & lock;
  TransactionScope(); // forbidden
public:
  TransactionScope(SimpleSpinLock & lock_): lock(lock_) {lock.lock();}
  ~TransactionScope() { lock.unlock(); }
};
```

Naive RTM implementation

```
class TransactionScope {
public:
  TransactionScope() {
     int nretries = 0;
     while(1) {
        ++nretries;
        unsigned status = _xbegin();
        if (status == _XBEGIN_STARTED) return; // successful start
        // abort handler
        std::cout << "DEBUG: Transaction aborted "<< nretries <<
            " time(s) with the status "<< status << std::endl;
     }
 }
 ~TransactionScope() { _xend(); }
};
```

Output

open new account
DEBUG: Transaction aborted 1 time(s) with the status 0
DEBUG: Transaction aborted 2 time(s) with the status 0
DEBUG: Transaction aborted 3 time(s) with the status 0
DEBUG: Transaction aborted 4 time(s) with the status 0
…

Improved RTM implementation

Acquire a fall-back spin lock non-transactionally after a specified number of retries

```
TransactionScope(SimpleSpinLock & fallBackLock_, int max_retries = 3) : fallBackLock
    (fallBackLock_) {
int nretries = 0;
while(1) {
  ++nretries;
  unsigned status = _xbegin();
  if (status == _XBEGIN_STARTED) {
      if (!fallBackLock.isLocked()) return; //successfully started transaction
      /* started txn but someone is executing the txn  section non-speculatively (acquired the
         fall-back lock) -> aborting */
      _xabort(0xff); // abort with code 0xff
  }
  // abort handler
  InterlockedIncrement64(&naborted); // do abort statistics
  std::cout << "DEBUG: Transaction aborted "<< nretries << " time(s) with the status "<<
    status << std::endl;
  // handle _xabort(0xff) from above
  if( (status & _XABORT_EXPLICIT) && _XABORT_CODE(status)==0xff && !(status &
    _XABORT_NESTED)) {
     // wait until the lock is free
     while (fallBackLock.isLocked())          _mm_pause();
  }
  // too many retries, take the fall-back lock
  if (nretries >= max_retries) break;
} //end while
fallBackLock.lock();
}
```

```
~TransactionScope() {
   if (fallBackLock.isLocked())
      fallBackLock.unlock();
   else
     _xend();
}
```

```
open new account
DEBUG: Transaction aborted 1 time(s) with the status 0
DEBUG: Transaction aborted 2 time(s) with the status 0
DEBUG: Transaction aborted 3 time(s) with the status 0
open new account
put 100 units into account 0
transfer 10 units from account 0 to account 1 atomically!
```

- All txns except the first one, succeeded
- First txn failed 3 times and then took the fallback path
  - It failed repeatedly because of its expensive operations: reserving and touching memory for its vector from the OS, which involves:
    - System calls
    - Privilege transitions
    - Page faults
    - Initialization/zeroing of big memory chunks that may not fit in the transactional buffer

Leveraging RTM Abort Status Bits

RTM EAX abort codes

EAX Register bit position	Meaning
0	Set if abort set by XABORT
1	If set, txn may succeed on retry; clear if bit 0 is set
2	Set if another CPU conflicted with a mem addr part of the aborted txn
3	Set if an internal buffer overflowed
4	Set if debug breakpoint was hit
5	Set if an abort occurred during nested txn
23:6	Reserved
31:24	XABORT argument (only valid if bit 0 set, otherwise reserved)[33]

- In case of such hard aborts, the “retry” bit (position 1) is not set
- If the hardware thinks the txn will succeed on retry, it will set the bit
- The optimization: if the bit is not set, choose the fallback already
- The idea is to make faster progress, rather than keep aborting and retrying when it seems unlikely that the txn will succeed on subsequent attempts

```
if( (status & _XABORT_EXPLICIT) && _XABORT_CODE(status)==0xff
    && !(status & _XABORT_NESTED)) {
      // wait until the lock is free
      while (fallBackLock.isLocked())          _mm_pause();
} else if(!(status & _XABORT_RETRY)) break; /* take the fall-back lock
    if the retry abort flag is not set */
// too many retries, take the fall-back lock
if (nretries >= max_retries) break;
```

---

open new account

DEBUG: Transaction aborted 1 time(s) with the status 0

open new account  **Faster progress!**

put 100 units into account 0

transfer 10 units from account 0 to account 1 atomically!

atomically draw 10 units from account 0 if there is enough money

add 1000 empty accounts atomically

More Statistics

- Run two worker threads that randomly choose two accounts from the array, and write to both accounts (transfer money from one to the other)
- Outcome: 100-300 aborts for 20k txns
- Testing if it all works correctly:
  - Introduce an update/increment of global counter in each txn (create many conflicts)
  - Outcome: 5k-15k aborts