Practical Experience with Transactional Lock Elision

Tingzhe Zhou, PanteA Zardoshti and Michael Spear

Lehigh University, Bethlehem, PA

Two Programming Models

- Transactions First
  - Input: TM features (self-abort, defer actions, CM)
  - Create new concurrent programs/modules

Scalability

Programmability

Create concurrent program from scratch.

- Transactional Lock Elision (TLE)
  - Input: lock based program
  - TM as a mechanism for achieving lock elision

Scalability

Easier to use in existing concurrent programs.

Contributions

- Evaluate the effectiveness of TLE on real-world programs
  - Transactionalizing two highly optimized programs (PBZip2 and x265)
  - C++ TM technical specification (TMTS)
- Extend TM API
  - TM.NoQuiesce()
- New insights
  - Existing tools and libraries
  - Obstacles unique to TLE

Outline

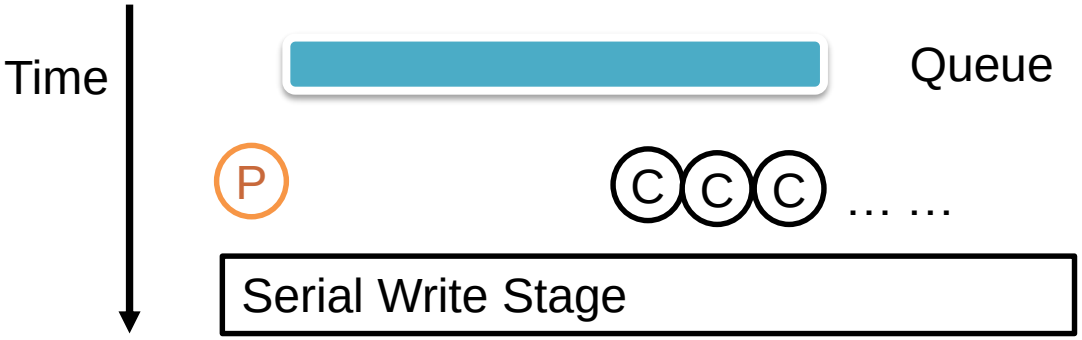
- PBZip2 and x265
- Quiescence and Lock Elision
- Obstacles, solutions and open challenges
- Evaluation

PBZip2[1]

PBZip2 is the parallel version of the bzip2 file compression algorithm.

Data Format

Execution



[1] Source: http://compression.ca/pbzip2/

X265

Data Format

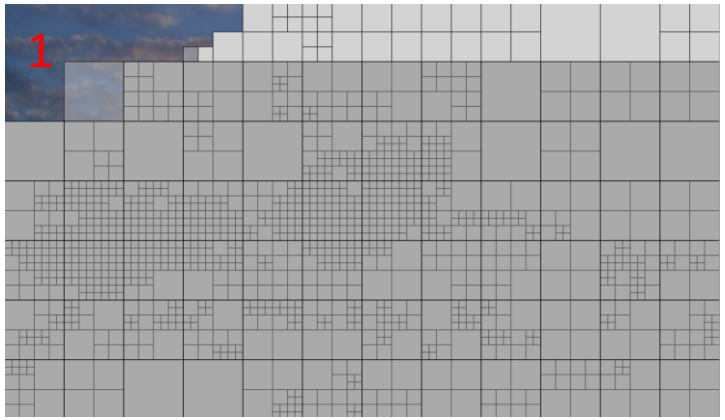
Frame

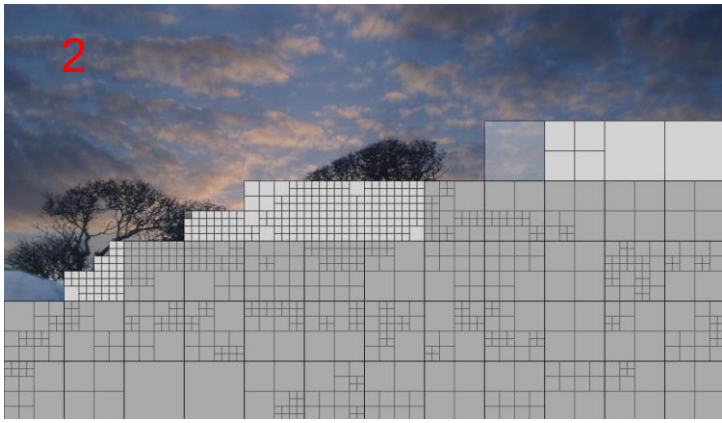
↓

Slices

↓

Coding Tree Units





HEVC wavefront parallel processing

Threading

1 main thread, m frame threads and n threads in pools.

Outline

- PBZip2 and x265
- **Quiescence and Lock Elision**
- Obstacles, solutions and open challenges
- Evaluation

Privatization Problem

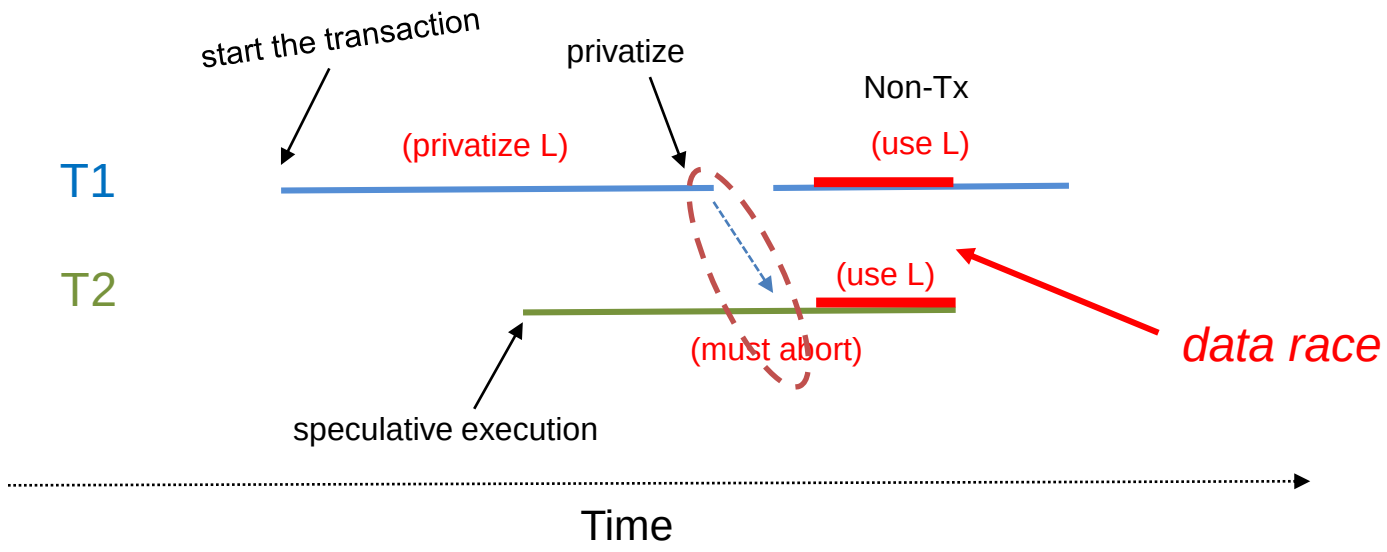
T1

```
__ transaction_atomic {
      node = L->head
      L->head = null
}

// L is privatized
process(node)
```

T2

```
__transaction_atomic{
      i_node = locate(L, i)
      if (i_node != null)
            i_node->data = process(i_node)
}
```



Quiescence in C++ TMTS

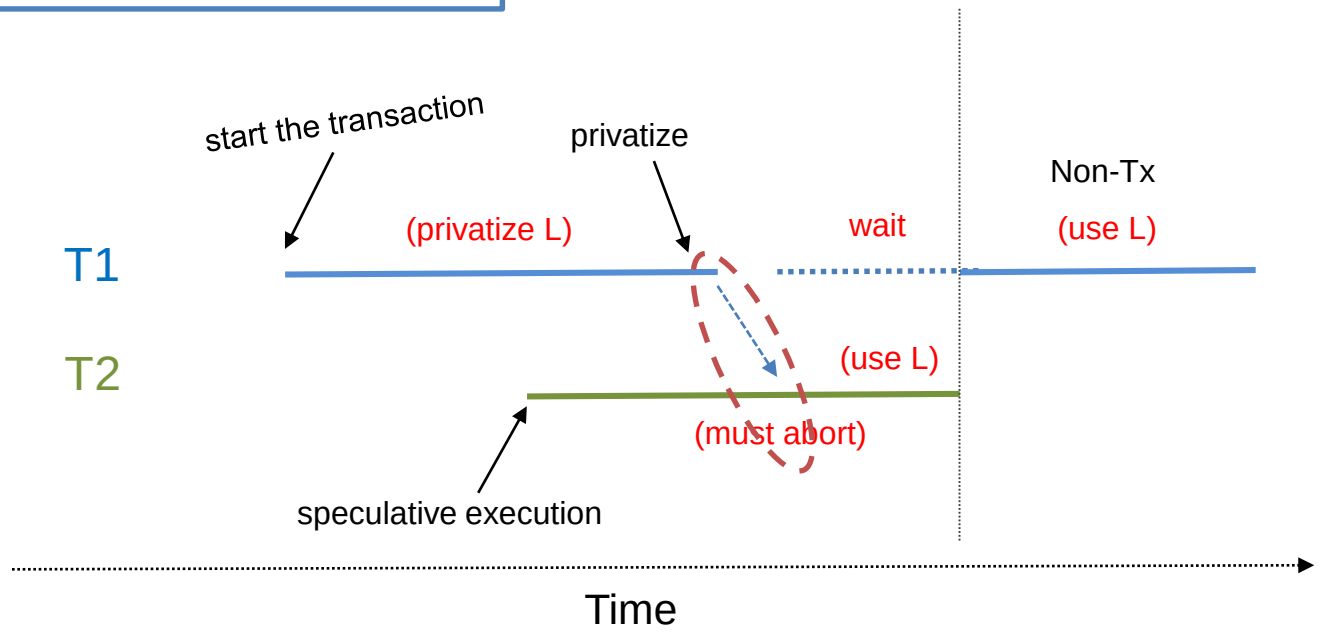
T1

```
__ transaction_atomic {
      node = L->head
      L->head = null
}

// L is privatized
process(node)
```

T2

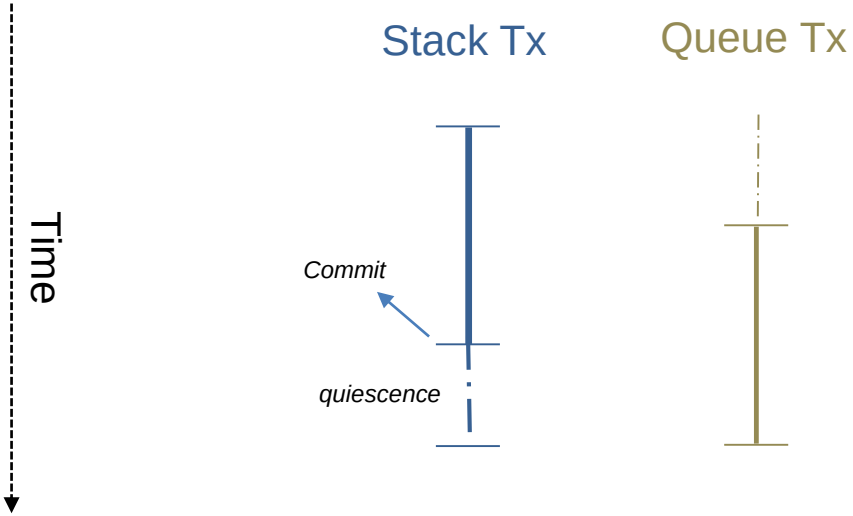
```
__transaction_atomic{
      i_node = locate(L, i)
      if (i_node != null)
            i_node->data = process(i_node)
}
```



Quiescence and Lock Elision

Three Problems

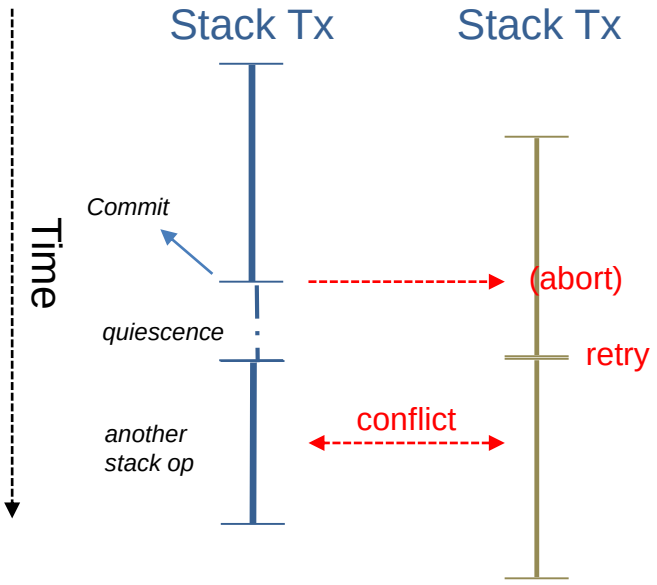
- Linear overhead
  - implementation
- Force the transaction to delay after it commits
  - granularity of quiescence



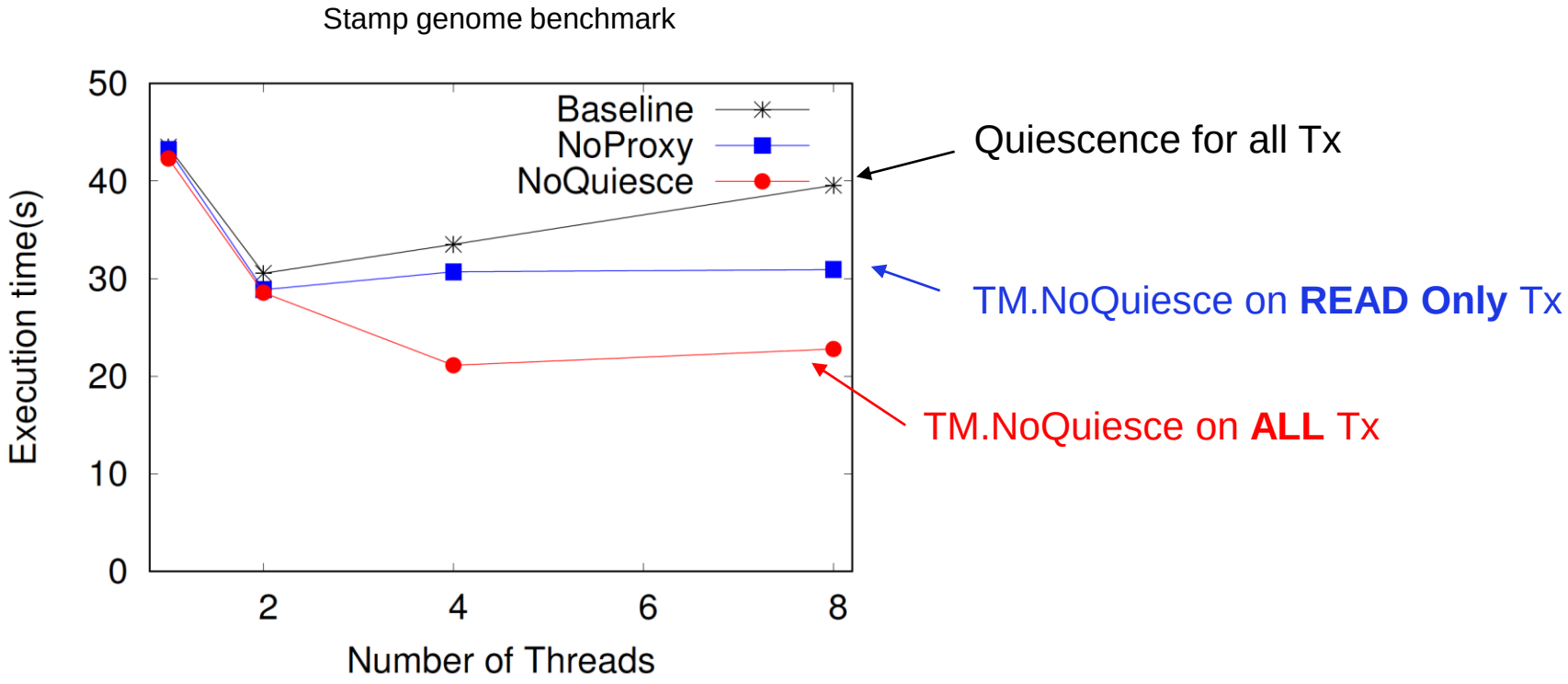
Quiescence and Lock Elision

Three Problems

- Linear overhead
- Force the transaction to delay after it commits
- **Transaction congestion**



Quiescence overhead in C++ TMTS



Programmatically Avoiding Quiescence

- Transactions need quiescence
  - transition data to non-transactional state
  - last transaction executed by the thread before it accesses data non-transactionally
- Disable quiescence for all transactions
  - improve performance
  - not compositional

- A new API function: TM.NoQuiesce
  - indicates transaction should NOT quiesce after it commits
  - free to be ignored (especially in HTM)

Programmatically Avoiding Quiescence

Listing 2: Producer/consumer workload

```
// Producer thread
1 while true do
2     atomic_cancel
3         if ¬c.full() then
4             c.insert(produce())
5         TM.NoQuiesce()
```

```
// Consumer thread
1 while true do
2     atomic_cancel
3         if ¬c.empty() then
4             tmp ← c.get()
5         else
6             tmp ← nil
7             TM.NoQuiesce()
8     if tmp ≠ nil then
9         use(tmp)
```

- The producer never needs quiesce
- The consumer only quiesces if it succeeds in getting an element

Pitfalls

- TM.NoQuiesce can increase scalability

TM.NoQuiesce

correct program → racy program

- Tx SC requires a global total order
- TM.NoQuiesce asserts dependencies with transactions in one thread are enough to provide happens-before
- We expect these errors to be easy to identify and fix using transactional race detectors

Outline

- PBZip2 and x265
- Quiescence and Lock Elision
- **Obstacles, solutions and open challenges**
- Evaluation

Naïve Transactionalization

- It should be easy to transactionalize code
- Critical sections in PBzip2 are transaction friendly
  - small critical sections
  - No expensive functions and system calls (file ops)
- x265 could NOT be naively transactionalized
  - Pattern of lock acquisitions and releases was clearly not two-phase locking

Problems of Naïve Transactionalization

Listing 3: Example of non-serializable critical section in x265.

```
   // LOCK for output queue(i)
1  OutputQueue.lock()
2  element = newQueueNode()
3  OutputQueue.enqueue(element)
4  process(element)
5  OutputQueue.unlock()

   // LOCK for output queue(ii)
6  OutputQueue.lock()
7  OutputQueue.dequeue()
8  OutputQueue.unlock()

   // Process(element)
   process(element)
9      m_lock.lock()
10     m_task = element.size()
11     m_lock.unlock()
12     sub_working()
13     Wait()
14     return

   // Task for working threads
   sub_working()
       ...
15     m_lock.lock()
16     m_task − −
17     m_lock.unlock()
       ...
```

Listing 4: A `ready` flag avoids lock nesting, facilitating transactionalization.

```
   // LOCK for output queue(i)
1  OutputQueue.lock()
2  element = newQueueNode()
3  OutputQueue.enqueue(element)
4  element.ready = false
5  OutputQueue.unlock()
6  process(element)
7  OutputQueue.lock()
8  element.ready = true
9  OutputQueue.unlock()

   // LOCK for output queue(ii)
10 OutputQueue.lock()
   if OutputQueue.peek().ready
11     element = OutputQueue.dequeue()
12 OutputQueue.unlock()
```

Frame thread

```
Lock(&A)
… …
    Lock(&B)
        m_task  = elem.size()
    unLock(&B)
    wait until m_task == 0
… …
unLock(&A)
```

worker threads

```
… …
    Lock(&B)
        m_task --;
    unLock(&B)
… …
```

1. Add ”ready flag”
2. Move process outside of TM
3. Check the flag

Two Phase Locking ?

- Can it be proven that naive transactionalization is always correct for critical sections that obey two-phase locking?
- Under what conditions will naive transactionalization of non-two-phase locking code remain safe?

Outline

- PBZip2 and x265
- Quiescence and Lock Elision
- Obstacles, solutions and open challenges
- **Evaluation**

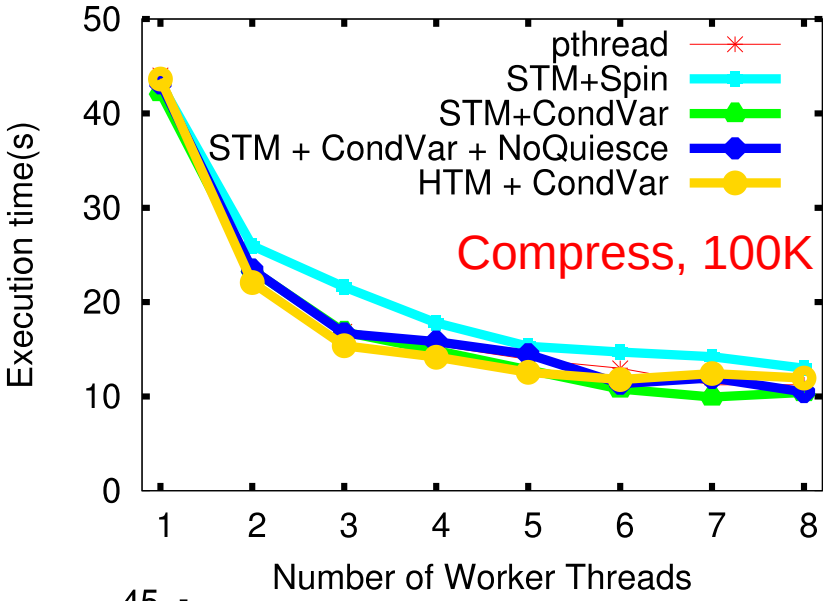
Evaluation

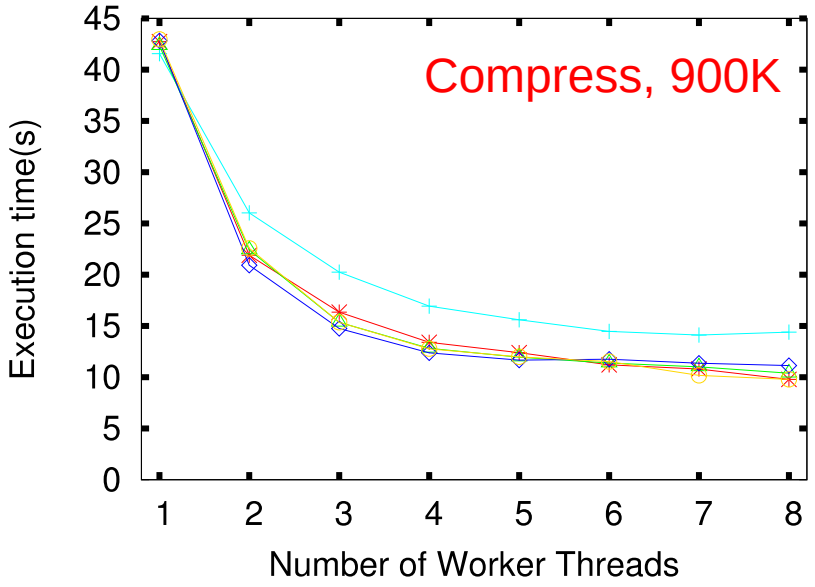
- Preserve the original structure of the source code
  - “ready” flag in x265
  - Additional TM.NoQuiesce calls
  - Minor refactoring to be able to use TMCondvars
- Environment
  - 4 core/8 thread Intel Core i7-4770 CPU, 3.40GHz, 8GB RAM, Intel TSX for HTM.
  - Linux 4.3, GCC 5.3.1
- Fallback-strategy (GCC default)
  - ml_wt (TinySTM) X 100 → serial path
  - htm (RTM) X 2 → serial path

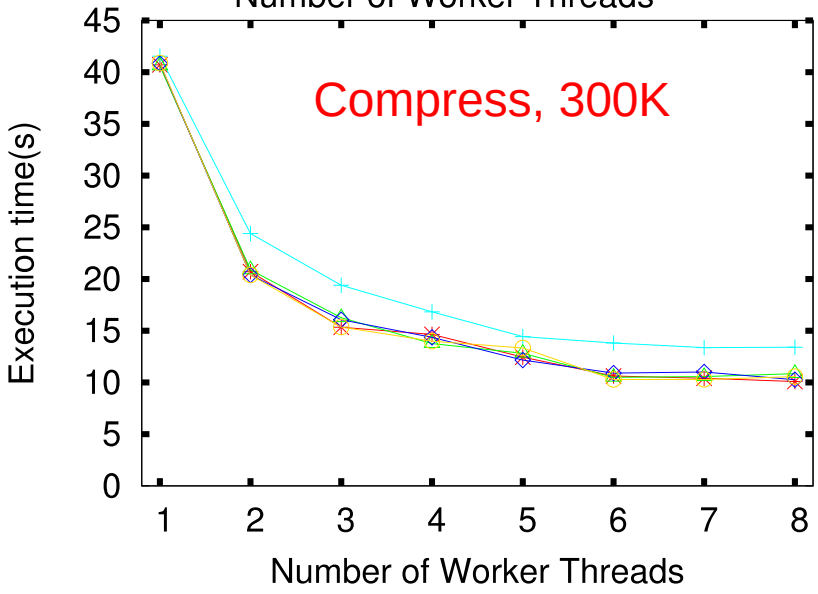
PBZip2

- Two independent operations
  - Compress
  - Decompress
- Variables configuration
  - worker threads: 1 to 8
  - block size: 100K, 300K, 900K. (650M test file)
- Five algorithms
  - pthread
  - STM + spin
  - STM + condVar
  - STM + condVar + TM.NoQuiesce
  - HTM + condVar

PBZip2

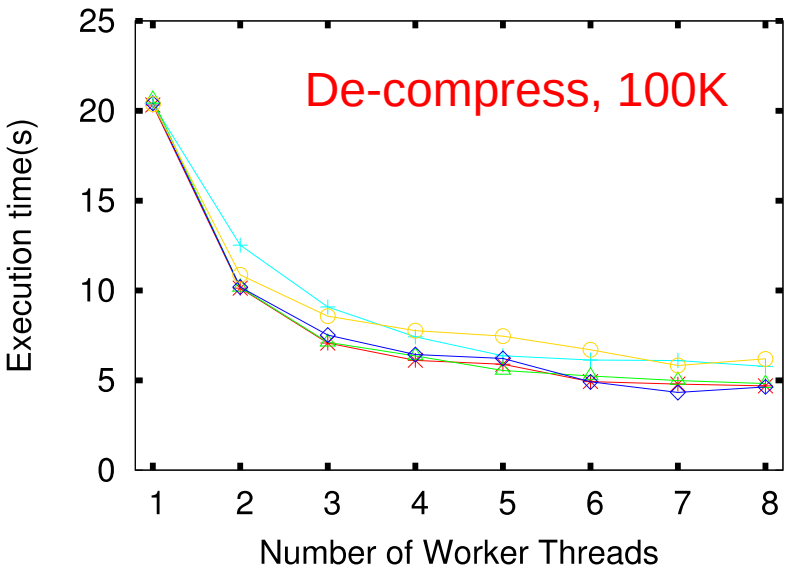


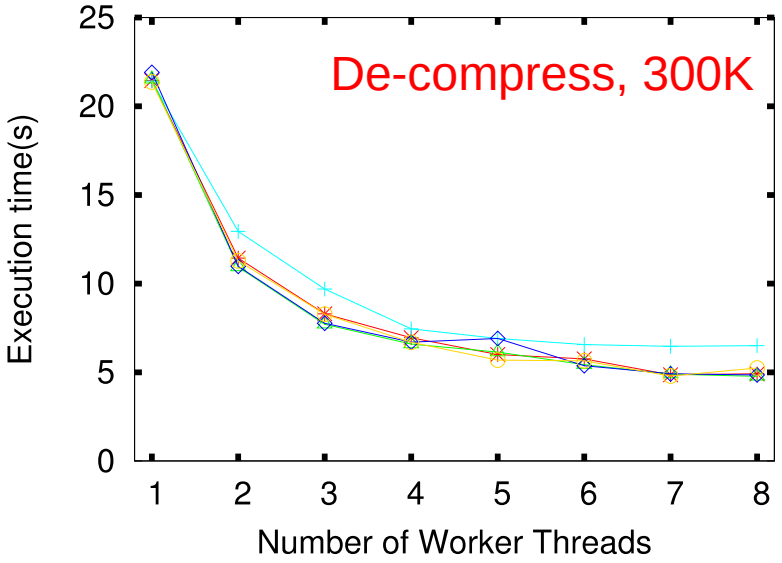


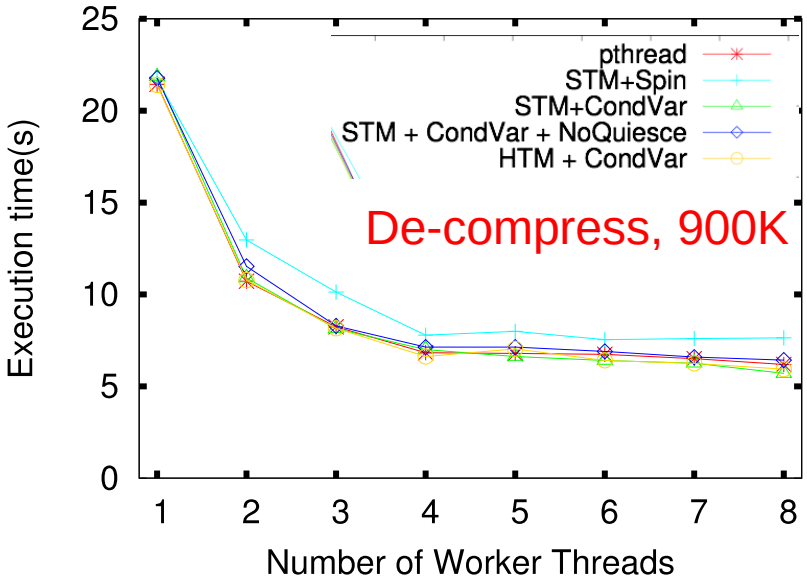


1. STM + Spin performs the worst in all conditions.
2. HTM has good performance in most cases, although 13% to 18% of transactions abort twice and fall back to serial path.
3. HTM outperforms the baseline, achieving a peak speedup of 8.5%.

PBZip2





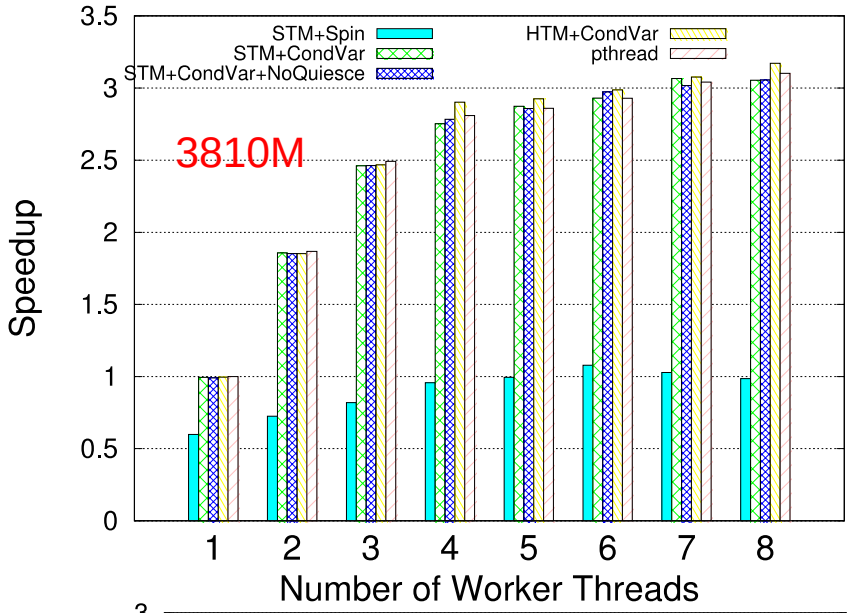


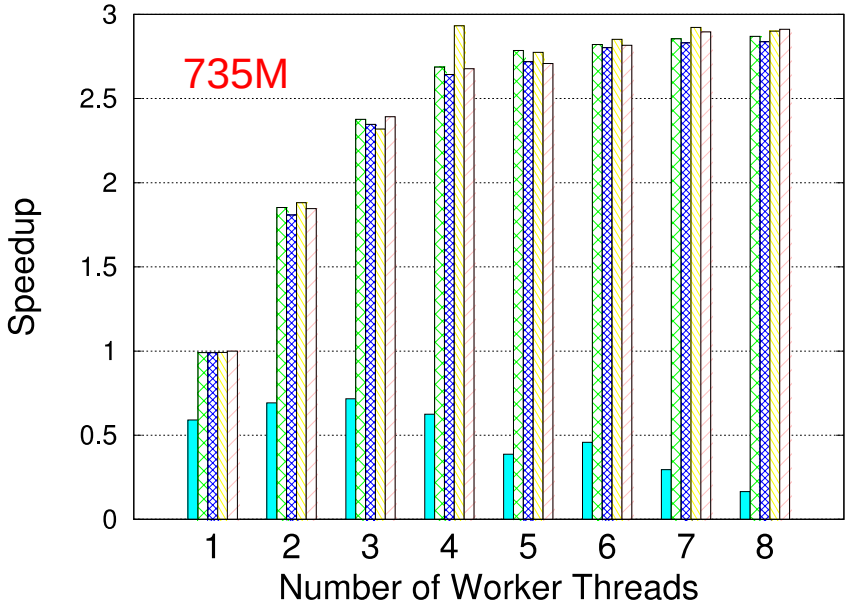
1. HTM performs worse than STM because nearly 20% of HTM transactions fall back to the serial path.

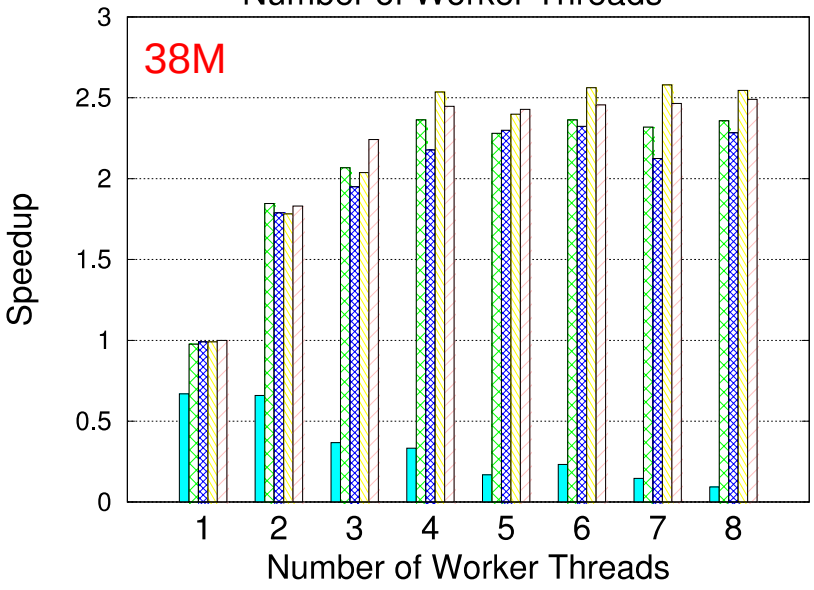
2. STM+Cond and NoQuiesce both could outperform the base line.

3. Disabling quiescence offers mixed results

x265







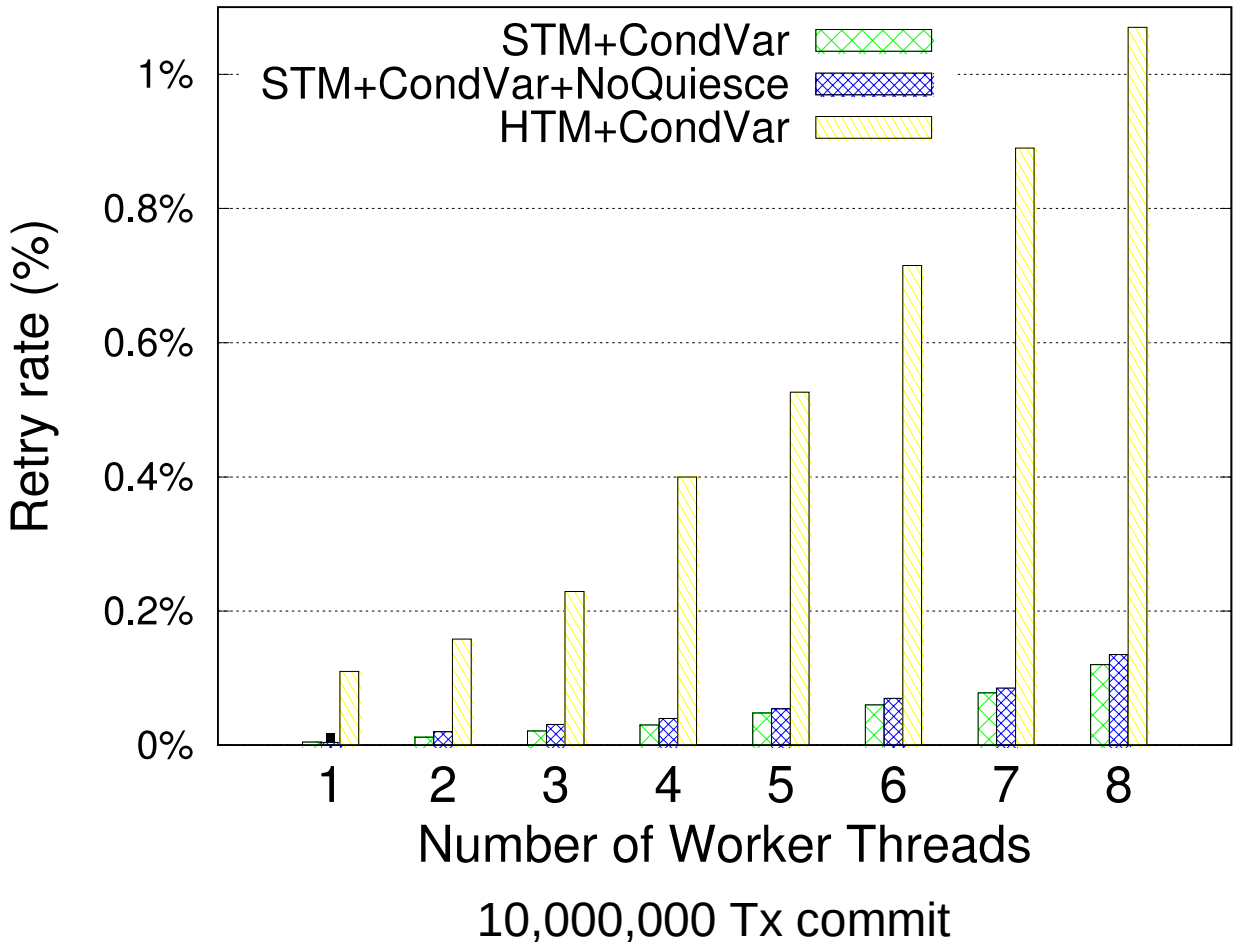
Base line: single-thread pthread execution.

1 main thread , 3 frame threads, X worker threads

The peak performance of HTM is 9.5% better than pthreads at 4 threads

TM.NoQuiesce performance is unstable.

x265


STM+CondVar
STM+CondVar+NoQuiesce
HTM+CondVar
Retry rate (%)
0%
0.2%
0.4%
0.6%
0.8%
1%
1
2
3
4
5
6
7
8
Number of Worker Threads
10,000,000 Tx commit

Conclusions and Future Work

- Applied C++ TMTS to elide locks in two programs
- Improved performance by up to 9%
- Our experience does not validate the expectation that transactional lock elision will be easy
- Quiescence avoidance need not be thought of either YES or NO
- We are at a point where we can start making small improvements that make a big difference!

Q & A

- Thank you !