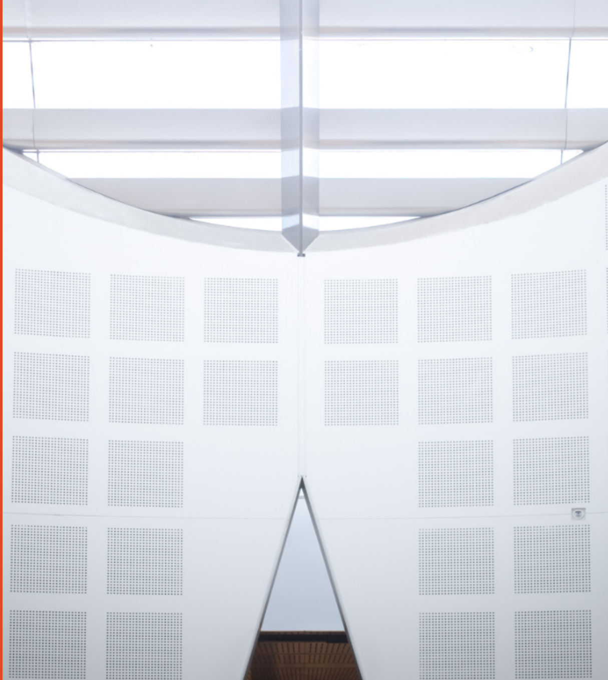
Hardware Transactional Memory

...and its relation to Soufflé

Jonathan Chung

March 23, 2018

THE UNIVERSITY OF SYDNEY



Overview

- Locks
  - Identify critical sections of code
  - Only one thread can execute code within it at a time

Overview

- Locks
  - Identify critical sections of code
  - Only one thread can execute code within it at a time
- Transactional Memory
  - Sections of code are treated as transactions - similar to a database
  - Optimistic - Resources are not immediately locked; compare the start and end states of the resource and commit updates/rollback accordingly
  - Monitor some form of *transaction variables* to see if they have been modified
  - **Serialisable** - the result of running something concurrently is the same as running them separate from each other
  - **Atomic** - either everything that is changed is commited, or nothing is
  - *What* do I want to execute atomically vs. *how* should I make it execute atomically
  - Can be done in software, but incurs significant overheads

Hardware Transactional Memory (HTM)

- Similar to transactional memory

Hardware Transactional Memory (HTM)

- Similar to transactional memory
- ...but with *hardware*

Hardware Transactional Memory (HTM)

- Similar to transactional memory
- ...but with *hardware*
- Requires modifications to hardware to support transactions - processors, cache, bus protocol
- Simple semantics - designate transactional area
- Intended to avoid common problems with locking (deadlocks, race conditions, etc.)
- Has yielded considerable performance improvements in certain previous applications

TSX Implementations

- **Transactional Synchronisation Extensions (TSX)**
  - Documented by Intel in 2012
  - First released on the Haswell microarchitecture in 2013

TSX Implementations

- **Transactional Synchronisation Extensions (TSX)**
  - Documented by Intel in 2012
  - First released on the Haswell microarchitecture in 2013
- Two interfaces:
- **Hardware Lock Elision (HLE)**
  - Can be used on TSX-incompatible hardware - its prefixes uses same byte encoding as existing prefixes for string-manipulation instructions
  - A write lock is not acquired at the start of the transaction (it is *elided*)
  - Write addresses are tracked, so if they is modified externally the transaction aborts
  - On an abort, the transaction restarts and acquires the lock

TSX Implementations

- **Transactional Synchronisation Extensions (TSX)**
  - Documented by Intel in 2012
  - First released on the Haswell microarchitecture in 2013
- Two interfaces:
- **Hardware Lock Elision (HLE)**
  - Can be used on TSX-incompatible hardware - its prefixes uses same byte encoding as existing prefixes for string-manipulation instructions
  - A write lock is not acquired at the start of the transaction (it is *elided*)
  - Write addresses are tracked, so if they is modified externally the transaction aborts
  - On an abort, the transaction restarts and acquires the lock
- **Restricted Transactional Memory (RTM)**
  - Requires TSX-compatible hardware
  - Allows greater flexibility to specify abort conditions, use or omit locks
  - Fallback path is required in case of transaction failure, which is also programmer-defined

RTM Instructions

- **_xbegin** - starts transactional execution for processor; returns value corresponding to success or status of abort (e.g. conflict, capacity)
  - Specifies fallback path in event of transactional failure
  - The abort status of _xbegin is stored in the EAX register
- **_xend** - specifies end of transactional code region, initiates commit
- **_xabort** - forces transaction to abort explicitly
- **_xtest** - check if processor is currently executing in a transactional region

Caches

- **Cache lines** - Transfers data between memory and cache in fixed size blocks
- **Associativity** - The number of places in the cache that can be mapped to memory

Caches

- **Cache lines** - Transfers data between memory and cache in fixed size blocks
- **Associativity** - The number of places in the cache that can be mapped to memory
- The processor tracks its sequence of accesses, known as read and write sets, which are stored in some hardware cache
- Which cache the sets are stored may differ between processors
  - In Skylake processors, read sets are tracked in the L3 cache (65536 cache lines, with associativity 16)
  - Write sets are brought to the L1 cache (512 cache lines, with associativity 8)
- The further away the cache, the less performant it is

Caches

- **Cache lines** - Transfers data between memory and cache in fixed size blocks
- **Associativity** - The number of places in the cache that can be mapped to memory
- The processor tracks its sequence of accesses, known as read and write sets, which are stored in some hardware cache
- Which cache the sets are stored may differ between processors
  - In Skylake processors, read sets are tracked in the L3 cache (65536 cache lines, with associativity 16)
  - Write sets are brought to the L1 cache (512 cache lines, with associativity 8)
- The further away the cache, the less performant it is
- If one thread's cache line in the read or write set is modified by another thread, the transaction aborts
- If a new access cannot be recorded in the read or write set, the transaction aborts

Causes of Aborted Transactions

- **Conflicts** - a thread's cache line is modified by another thread during a transaction
- **Capacity** - the internal buffer overflowed (hardware/resource constraints)
- **Explicit** - a forced abort, caused by calling `_xabort()`

Causes of Aborted Transactions

- **Conflicts** - a thread's cache line is modified by another thread during a transaction
- **Capacity** - the internal buffer overflowed (hardware/resource constraints)
- **Explicit** - a forced abort, caused by calling `_xabort()`
- Other causes include:
  - Ring transitions - functions that require changing levels of privilege
  - Using unsupported functions: `strcmp`, `strcpy`, `new`, `delete`
  - Interrupts
  - x86 instructions - PAUSE, CPUID instructions (returning processor information)

Causes of Aborted Transactions

- **Conflicts** - a thread's cache line is modified by another thread during a transaction
- **Capacity** - the internal buffer overflowed (hardware/resource constraints)
- **Explicit** - a forced abort, caused by calling `_xabort()`
- Other causes include:
  - Ring transitions - functions that require changing levels of privilege
  - Using unsupported functions: `strcmp`, `strcpy`, `new`, `delete`
  - Interrupts
  - x86 instructions - PAUSE, CPUID instructions (returning processor information)
- Usually, retry the transaction if allowed to
- If capacity reached or out of retries, revert to a fallback software lock

B-Trees

- Used to store relations
- Allows a certain range of keys per node
- Self-balancing during inserts and removals
- Optimised for reading and writing large amounts of data - $O(\log n)$
- Soufflè implementation differs slightly:
  - Hints
  - Read/write locks
  - No key removal

Old Code - "BTree.h"

```
while (root == nullptr) {
    if (!root_lock.try_start_write()) {
        continue;
    }
    if (root != nullptr) {
        root_lock.end_write();
        break;
    }
    leftmost = new leaf_node();
    leftmost->numElements = 1;
    leftmost->keys[0] = k;
    root = leftmost;
    root_lock.end_write();
    hints.last_insert = leftmost;
    return true;
}
```

New Code - "htmx86.h"

```
#define IS_LOCKED(lock) \
    (__atomic_load_n((long int*)&fallback_lock,
      __ATOMIC_SEQ_CST) != fallback_unlocked_value)
#define TX_RETRIES(num) int retries = num;
#define TX_START(type)                              \
    while (1) {                                     \
        while (IS_LOCKED(fallback_lock))            \
            ;                                       \
        unsigned status = _xbegin();                \
        if (status == _XBEGIN_STARTED) {            \
            if (IS_LOCKED(fallback_lock)) _xabort(1); \
            break;                                  \
        } else {                                    \
            if (!(status & _XABORT_RETRY))          \
                retries = 0;                        \
```

New Code - ”htmx86.h”

```
            else                                                \
                retries--;                                      \
        }                                                       \
        if (retries <= 0) {                                     \
            fallback_lock.lock();                               \
            break;                                              \
        }                                                       \
    }

#define TX_END                        \
    if (retries > 0) {                \
        _xend();                      \
    } else {                          \
        fallback_lock.unlock();       \
    }
```

Thanks to Vincent Gramoli for providing an RTM template

New Code - "BTree.h"

```
TX_RETRIES(maxRetries());
if (isTransactionProfilingEnabled()) {
    TX_START_INST(NL, (&tdata));
} else {
    TX_START(NL);
}
if (empty()) {
    leftmost = new leaf_node();
    leftmost->numElements = 1;
    leftmost->keys[0] = k;
    root = leftmost;
    hints.last_insert = leftmost;
    TSX_END;
    return true;
}
```

The DOOP Experiment

- "A collection of various analyses expressed as Datalog rules"
- Object-sensitive analyses with varying degrees of complexity

The DOOP Experiment

- "A collection of various analyses expressed as Datalog rules"
- Object-sensitive analyses with varying degrees of complexity
  - 1-object-sensitive+heap, 2-object-sensitive+2-heap, 3-object-sensitive+3-heap (1o1h, 2o2h, 3o3h respectively)
  - A flavour of *context sensitivity*, which qualifies variables and abstract objects with context information
  - Object-sensitive analysis has a *calling context* for object abstractions (i.e. allocation sites), plus a *heap context* for heap abstractions
  - 2o2h, 3o3h have calling and heap contexts of two and three allocation sites

The DOOP Experiment

- "A collection of various analyses expressed as Datalog rules"
- Object-sensitive analyses with varying degrees of complexity
  - 1-object-sensitive+heap, 2-object-sensitive+2-heap, 3-object-sensitive+3-heap (1o1h, 2o2h, 3o3h respectively)
  - A flavour of *context sensitivity*, which qualifies variables and abstract objects with context information
  - Object-sensitive analysis has a *calling context* for object abstractions (i.e. allocation sites), plus a *heap context* for heap abstractions
  - 2o2h, 3o3h have calling and heap contexts of two and three allocation sites
- DaCapo 2006 benchmarks: antlr, bloat, chart, eclipse...

The DOOP Experiment

- "A collection of various analyses expressed as Datalog rules"
- Object-sensitive analyses with varying degrees of complexity
  - 1-object-sensitive+heap, 2-object-sensitive+2-heap, 3-object-sensitive+3-heap (1o1h, 2o2h, 3o3h respectively)
  - A flavour of *context sensitivity*, which qualifies variables and abstract objects with context information
  - Object-sensitive analysis has a *calling context* for object abstractions (i.e. allocation sites), plus a *heap context* for heap abstractions
  - 2o2h, 3o3h have calling and heap contexts of two and three allocation sites
- DaCapo 2006 benchmarks: antlr, bloat, chart, eclipse...
  - (These are existing tools written in Java)

The DOOP Experiment

- "A collection of various analyses expressed as Datalog rules"
- Object-sensitive analyses with varying degrees of complexity
  - 1-object-sensitive+heap, 2-object-sensitive+2-heap, 3-object-sensitive+3-heap (1o1h, 2o2h, 3o3h respectively)
  - A flavour of *context sensitivity*, which qualifies variables and abstract objects with context information
  - Object-sensitive analysis has a *calling context* for object abstractions (i.e. allocation sites), plus a *heap context* for heap abstractions
  - 2o2h, 3o3h have calling and heap contexts of two and three allocation sites
- DaCapo 2006 benchmarks: antlr, bloat, chart, eclipse...
  - (These are existing tools written in Java)
- Measure runtime and memory footprint

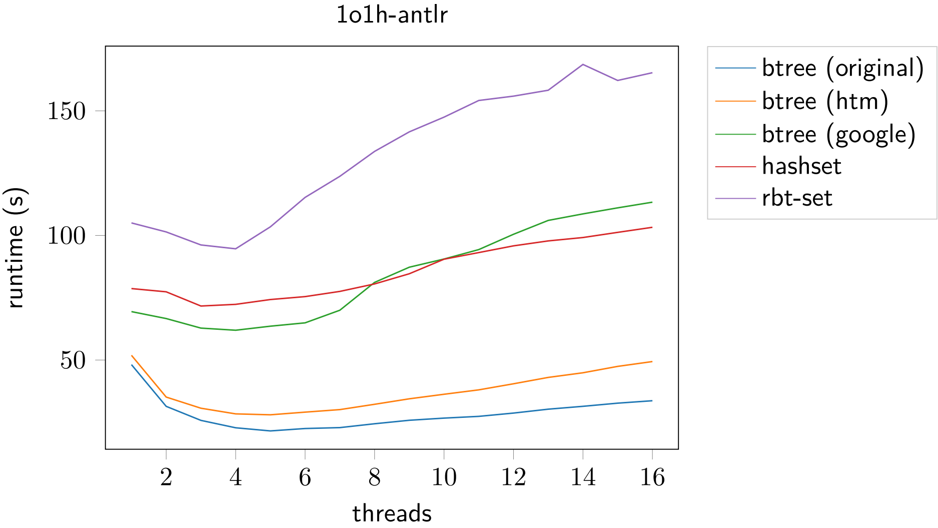
Data Structures

- B-Tree (original): our original, existing, lovingly-optimised implementation
- B-Tree (HTM): our new implementation using HTM for insertion (particularly, Restricted Transactional Memory)
- B-Tree (Google): a Google implementation of B-Trees, from which the current original implementation was derived

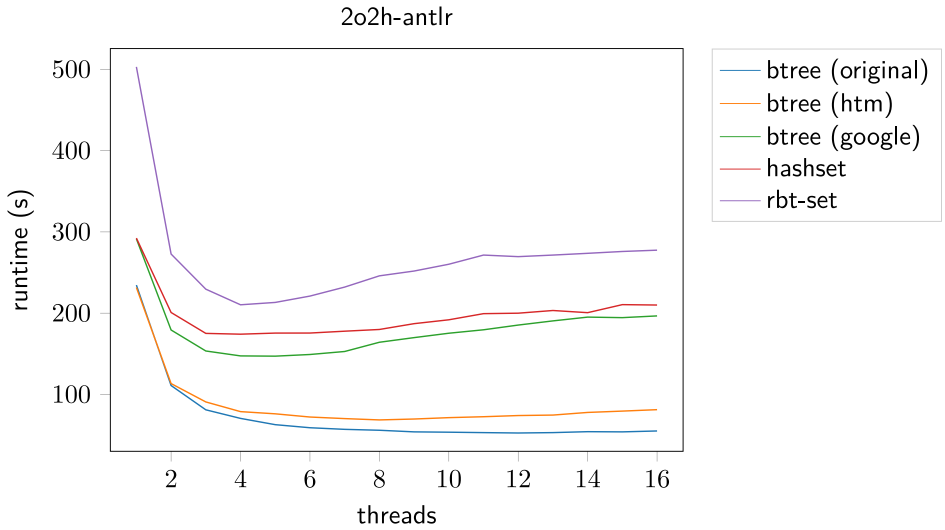
Data Structures

- B-Tree (original): our original, existing, lovingly-optimised implementation
- B-Tree (HTM): our new implementation using HTM for insertion (particularly, Restricted Transactional Memory)
- B-Tree (Google): a Google implementation of B-Trees, from which the current original implementation was derived
- Unordered Hashset: a hash-based data structure using STL's unordered sets promising fast lookups; must recursively hash each relation/tuple
- Ordered Hashset (RBT-set): similar, but using STL's ordered sets; now based on red-black trees

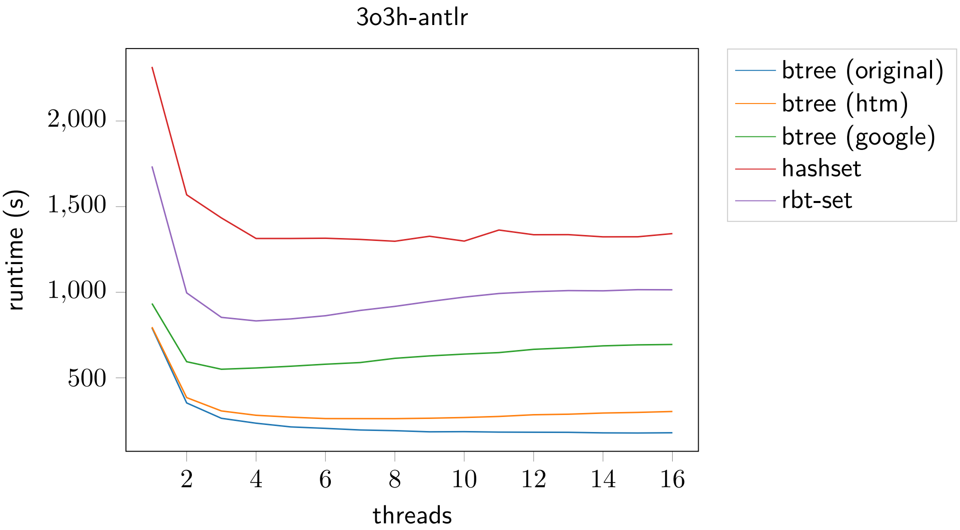
Runtime


1o1h-antlr
150
100
50
runtime (s)
2
4
6
8
10
12
14
16
threads
btree (original)
btree (htm)
btree (google)
hashset
rbt-set

Runtime

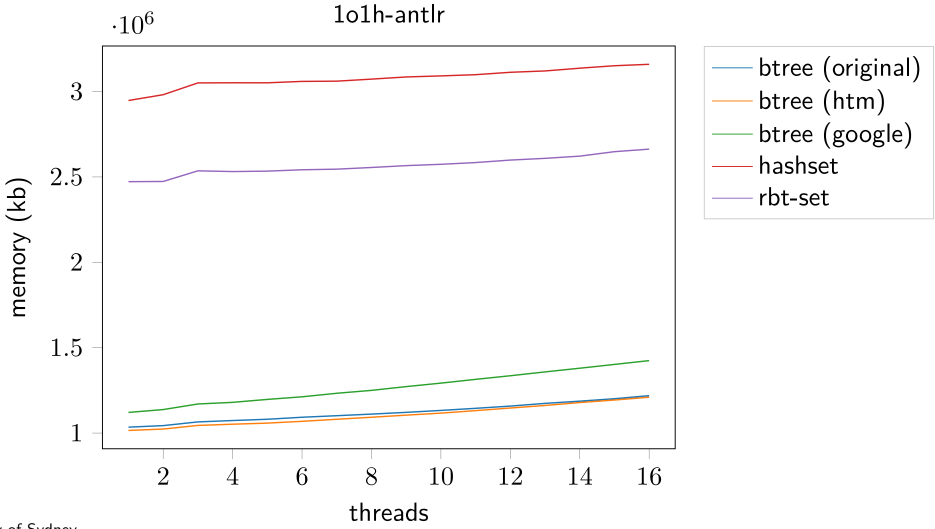

2o2h-antlr
500
400
300
200
100
runtime (s)
2
4
6
8
10
12
14
16
threads
btree (original)
btree (htm)
btree (google)
hashset
rbt-set

Runtime

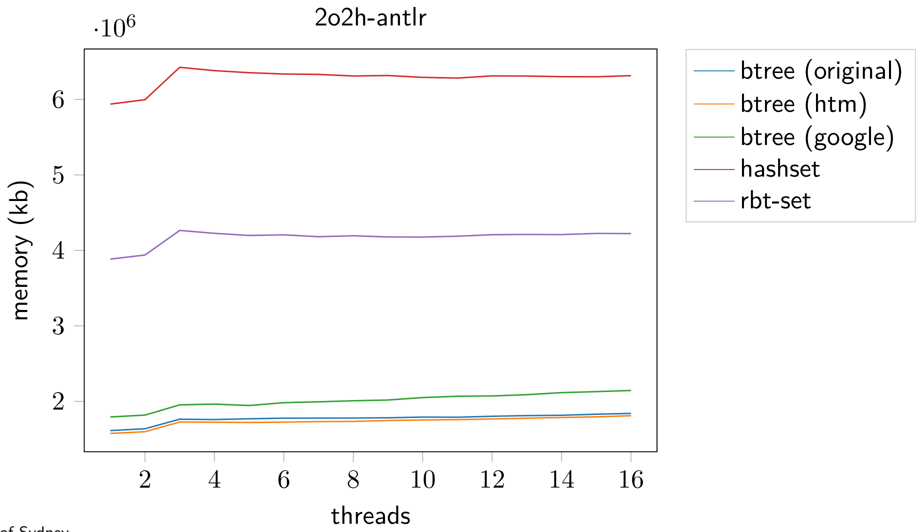


3o3h-antlr
runtime (s)
2,000
1,500
1,000
500
2
4
6
8
10
12
14
16
threads
btree (original)
btree (htm)
btree (google)
hashset
rbt-set

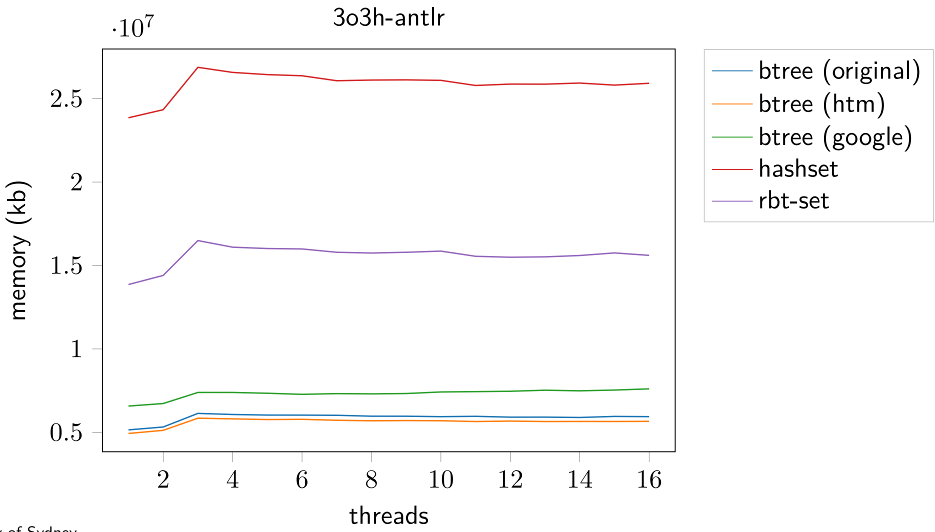
Memory


1o1h-antlr
$\cdot 10^6$
memory (kb)
3
2.5
2
1.5
1
2
4
6
8
10
12
14
16
threads
btree (original)
btree (htm)
btree (google)
hashset
rbt-set

Memory


2o2h-antlr
·10^6
memory (kb)
6
5
4
3
2
2
4
6
8
10
12
14
16
threads
btree (original)
btree (htm)
btree (google)
hashset
rbt-set

Memory


3o3h-antlr
$\cdot 10^7$
memory (kb)
2.5
2
1.5
1
0.5
2
4
6
8
10
12
14
16
threads
btree (original)
btree (htm)
btree (google)
hashset
rbt-set

What happened?

- Worse runtime than original B-Trees, though still considerably better than Google's implementation (and a lot better than hash-based data structures)
- Marginally better memory footprint than original B-Trees
- Doesn't scale as well with threads - reaches its peak at 4/8 cores

What happened?

- Worse runtime than original B-Trees, though still considerably better than Google's implementation (and a lot better than hash-based data structures)
- Marginally better memory footprint than original B-Trees
- Doesn't scale as well with threads - reaches its peak at 4/8 cores
- Why is HTM slower?

What happened?

- Worse runtime than original B-Trees, though still considerably better than Google's implementation (and a lot better than hash-based data structures)
- Marginally better memory footprint than original B-Trees
- Doesn't scale as well with threads - reaches its peak at 4/8 cores
- Why is HTM slower?

- One thread:
- 4268538 transactions
- 105967 total aborts
- 4772 aborts due to conflicts
- 1472 aborts due to capacity
- 99723 'other' aborts
- 101671 software fallbacks

- Two threads:
- 7305118 transactions
- 3723216 aborts
- 3477314 aborts due to conflicts
- 2251 aborts due to capacity
- 243651 'other' aborts
- 247557 software fallbacks

What happened?

- Worse runtime than original B-Trees, though still considerably better than Google's implementation (and a lot better than hash-based data structures)
- Marginally better memory footprint than original B-Trees
- Doesn't scale as well with threads - reaches its peak at 4/8 cores
- Why is HTM slower?
  - One thread:
  - 4268538 transactions
  - 105967 total aborts
  - 4772 aborts due to conflicts
  - 1472 aborts due to capacity
  - 99723 'other' aborts
  - 101671 software fallbacks

  - Two threads:
  - 7305118 transactions
  - 3723216 aborts
  - 3477314 aborts due to conflicts
  - 2251 aborts due to capacity
  - 243651 'other' aborts
  - 247557 software fallbacks
- Coarse granularity, large transaction size

What now?

- Finer granularity of transactions for HTM in the insert operation - potentially reducing conflicts?
- Could HTM be used elsewhere in the B-Tree/Soufflé to greater success?
- Switching out Soufflé's custom read/write locks for C++'s new standard implementations (e.g. `shared_mutex`)?
- Which is the greater bottleneck: lookups or insertion?