Hardware Transactional Memory

In relation to Software Transactional Memory and to concurrent data structures design

Jason Dellaluce, Michal Guzek

Traditional database transactions

Have to follow **ACID** properties:

- **A**tomicity - treat multiple operations as an indivisible unit
- **C**onsistency - each transaction's outcome results in a valid database state
- **I**solation - concurrent execution of multiple transactions have the same effect as if they were executed in a serialized manner
- **D**urability - the transaction's outcome will be preserved once the database is shut down

Software Transactional Memory

Two main properties of interest:

- **Atomicity -** treat multiple operations as an indivisible unit
- **Isolation -** concurrent execution of multiple transactions have the same effect as if they were executed in a serialized manner

STM can use locks or be implemented as a lock-free algorithm. It usually refers to its global version clock when reading/writing to a variable, so as to verify whether another thread hasn’t updated a given memory location in the meantime.

STM - now we can do better than that

Common problems with STM:

- Versioning/bookkeeping code is still handled explicitly on the software level
- Some implementations defer version comparison of memory location until commit time - might be too late if some errors have already occured due to the data races
- If an implementation use excessive amount of locking, the ultimate gain is negligible

Hardware Transactional Memory - concept

Ability to perform speculative execution of operations across many memory locations.
Transactions are executed in parallel, and aborted if data inconsistency is detected (roll-back)
Utilizes cache coherence protocols (like MESI) in order to ensure data integrity

- Read-set: Memory locations readed by a transaction
- Write-Set: Memory locations written by a transaction

Aborts take place when:

- Memory locations in read-set have been externally modified before ending the transaction
- Some other thread wants an exclusive access on a memory location in our write-set
- Cache line needs to be evicted from the cache itself (L1 size of particular importance here, in Haswell it’s 64KB per core)

Not every hardware transaction is guaranteed to succeed!

Hardware Transactional Memory - history

1993 - first mentioned in academic papers

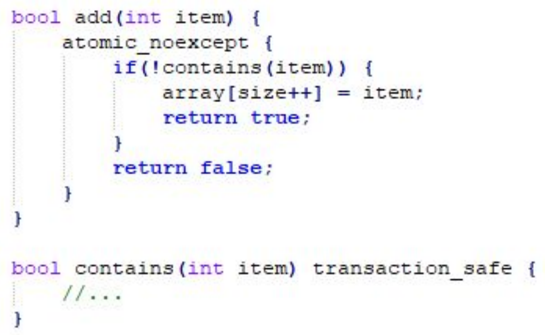
2012 - Intel announces the concept to be implemented by its mainstream processors

2013 - Haswell microarchitecture with HTM support released in 2013

Hardware and Software-level implementations

38-page long C++ working draft from 2015 focuses entirely on transactional memory and introduces new keywords such as: `synchronized` and `atomic` blocks, `transaction_safe` function specifier

g++ implements this features in ITM library starting from 4.7 version under `–fgnu–tm` flag



```
bool add(int item) {
    atomic_noexcept {
        if(!contains(item)) {
            array[size++] = item;
            return true;
        }
        return false;
    }
}

bool contains(int item) transaction_safe {
    //...
}
```

Drawbacks:

- g++ still reports internal compiler errors while using more advanced features, especially `transaction_safe_dynamic` function specifiers for virtual functions
- Marking functions as `transaction_safe` can be cumbersome (the bulk of functions from the standard library is not `transaction_safe`)
- Even when the transaction eventually succeeds, there's no guarantee that HTM is used, even when its support is present - it might just as well fall back on STM (Hybrid-Transaction approach)

Hardware and Software-level implementations

Intel **Transactional Synchronization Extensions (TSX)**, an extension to its base instruction set, gives us a direct access to low-level HTM instructions.

Intel TSX is comprised of two main features:

- Hardware Lock Elision (HLE) - provides HTM-like support for older architectures, not covered in our work
- Restricted Transactional Memory (RTM) - new instructions to leverage HTM, such as `XBEGIN`, `XEND` (to mark the transaction boundaries) and `XABORT` (to abort the transaction explicitly in the source code)

RTM is supported in g++ with `-mrtm` flag and `<immintrin.h>` header

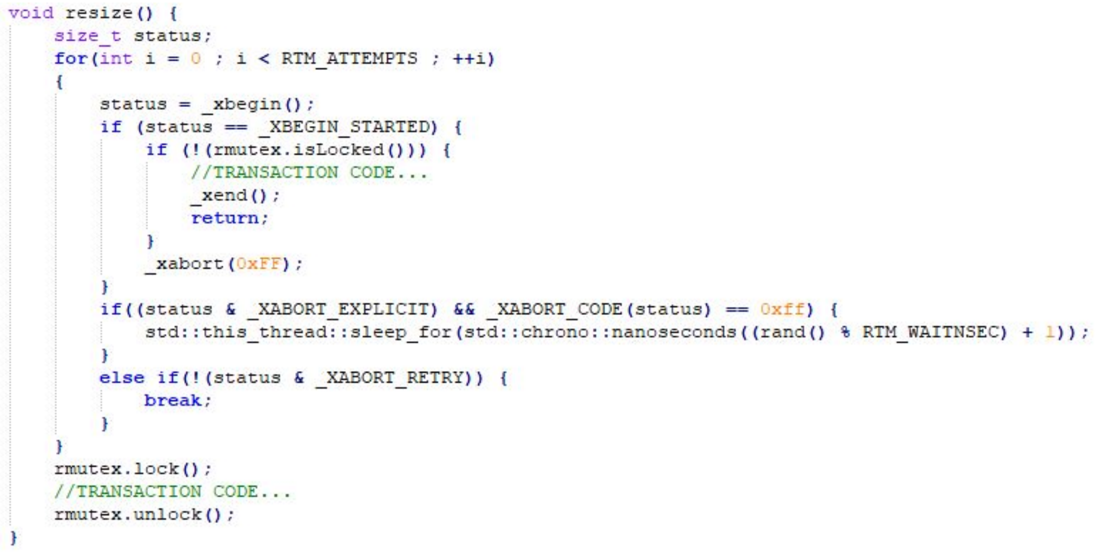
Since RTM and the underlying hardware instructions provide at most best-effort when it comes to executing a transaction by means of HTM support, we always have to provide a **fallback section**, which is usually surrounded by an ordinary lock

RTM general “template” in C++

Intel **TSX**’s general use case looks in the following way:

1. If transaction’s start was successful:
   a. Make sure the lock variable allows for lock elision
   b. Execute the transaction code
2. Else:
   a. Make sure the transaction can be retried and attempt it again
   b. If not, proceed to fall back section

A fall back section is executed whenever we cannot retry a transaction or we have already reached the retry threshold



```
void resize() {
    size_t status;
    for(int i = 0 ; i < RTM_ATTEMPTS ; ++i)
    {
        status = _xbegin();
        if (status == _XBEGIN_STARTED) {
            if (!(rmutex.isLocked())) {
                //TRANSACTION CODE...
                _xend();
                return;
            }
            _xabort(0xFF);
        }
        if((status & _XABORT_EXPLICIT) && _XABORT_CODE(status) == 0xff) {
            std::this_thread::sleep_for(std::chrono::nanoseconds((rand() % RTM_WAITNSEC) + 1));
        }
        else if(!(status & _XABORT_RETRY)) {
            break;
        }
    }
    rmutex.lock();
    //TRANSACTION CODE...
    rmutex.unlock();
}
```

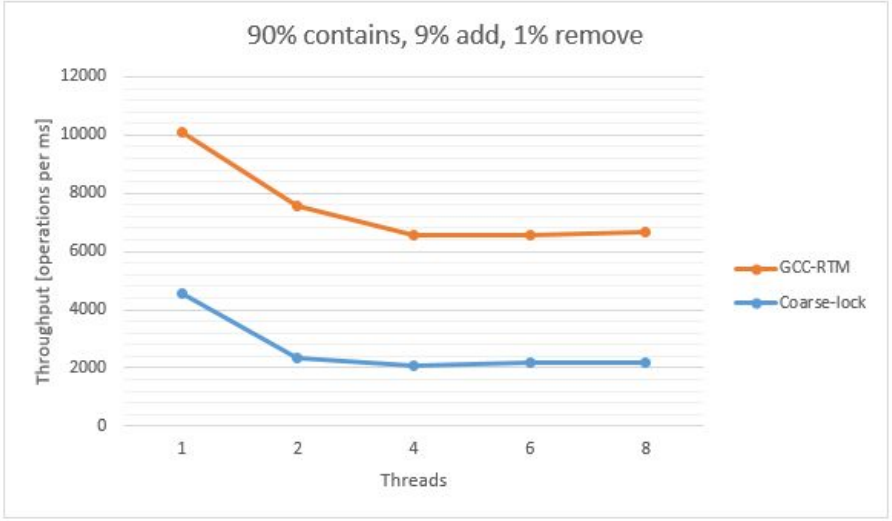
Good practices towards concurrent objects

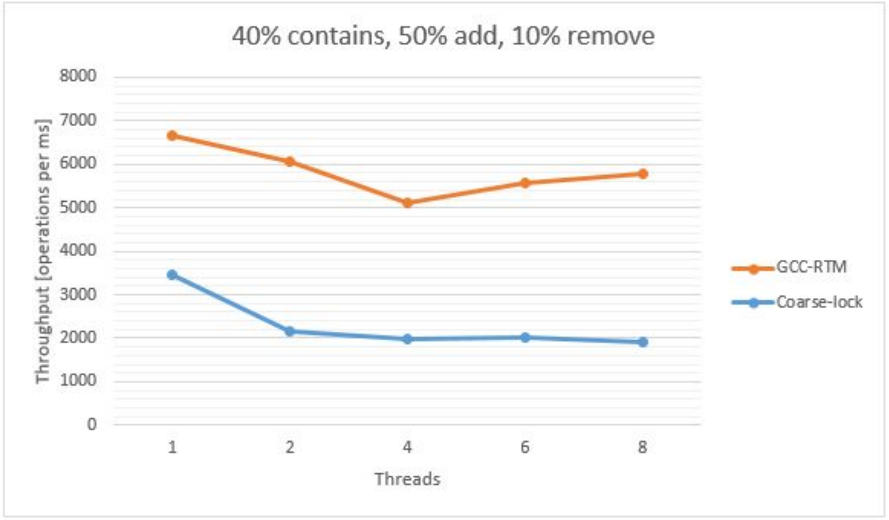
There are a few important things to keep in mind while writing hardware transactional code with RTM:

- Retrying RTM loop forever yields ill-formed code due to the fact that HTM-backed transaction is not guaranteed to always succeed, even in the absence of data races whatsoever
- Retrying RTM loop too quickly might cause a thread to reach a retry limit prematurely, in case some lock is being held for longer amount of time. Also, contention would be increased. → **Backoff for a while!**
- Lock variable needs to be explicitly read by a transaction so as to keep track of that cache line throughout the transaction (simple `lock.is_free()` call that returns boolean value is not enough); this concept is known as **lock elision** and underlies the whole HTM methodology
- HTM transactions cannot be applied to some scenarios where they cannot undo their operations, like handling I/O for instance, so don't even try them there

Case study: HashSet

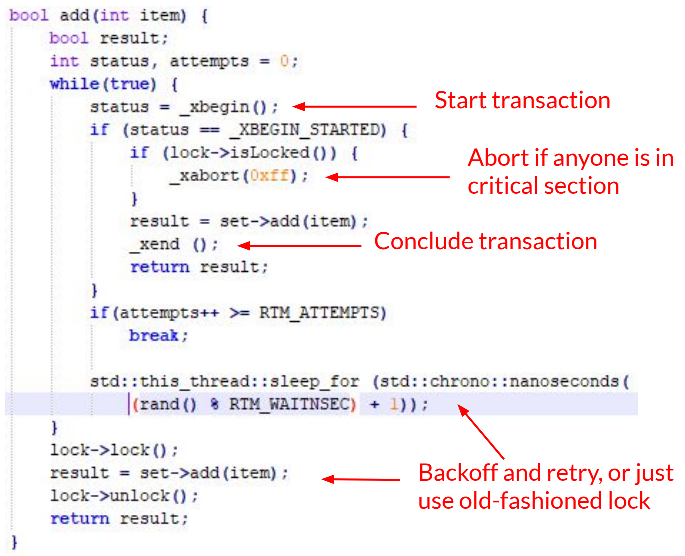
- HashSet implementation based on coarse-grained lock version from the textbook
- Employing RTM support causes significant improvement in the throughput
- 90% contains, 9% add and 1% remove is a typical workload scenario for most applications
- Tested on an Intel Core i9-9880H CPU @ 2.30GHz, 8 physical cores (with TSX support)





Case study: Linked-List-based Set

- Implemented a **Set based on a Linked-List**
- Used same approach discussed during the lectures
- Revisited in order to leverage HTM
- Used both GNU **LibITM** and **Intel's RTM**
- Performance have been compared to **coarse-grained** locking and **lock-free** approaches

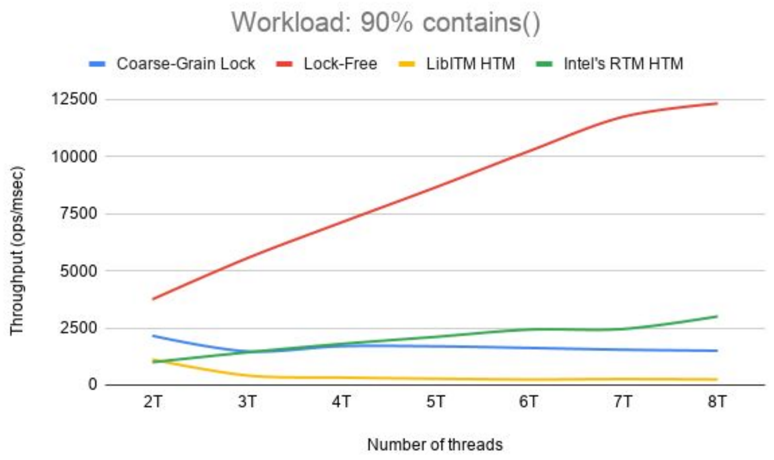

Start transaction
Abort if anyone is in critical section
Conclude transaction
Backoff and retry, or just use old-fashioned lock

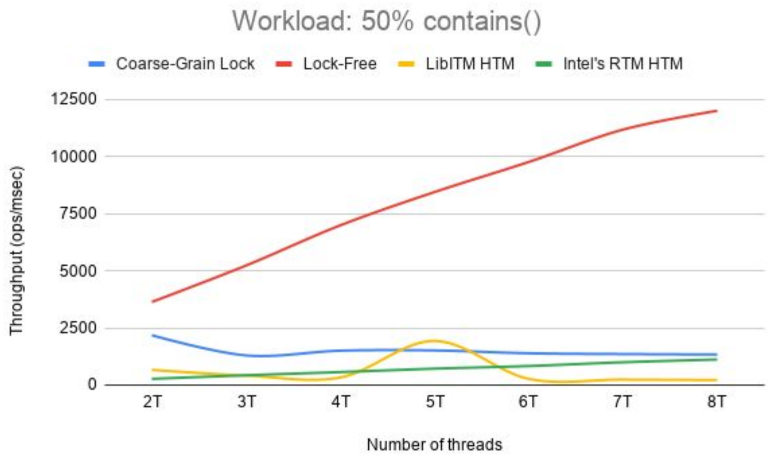
```
bool add(int item) {
    bool result;
    int status, attempts = 0;
    while(true) {
        status = _xbegin();
        if (status == _XBEGIN_STARTED) {
            if (lock->isLocked()) {
                _xabort(0xff);
            }
            result = set->add(item);
            _xend ();
            return result;
        }
        if(attempts++ >= RTM_ATTEMPTS)
            break;

        std::this_thread::sleep_for (std::chrono::nanoseconds(
            (rand() % RTM_WAITNSEC) + 1));
    }
    lock->lock();
    result = set->add(item);
    lock->unlock();
    return result;
}
```

Set case study: Performance comparison

- Benchmark addresses different workloads
- Tested on an Intel Core i9-9880H CPU @ 2.30GHz, 8 physical cores (with TSX support)
- Lock-free is still the best in the most common cases, but HTM competes with classical locking!





Linked List: Some considerations

- CONS
  - Transactions' **read-set can include the whole list** in a full traversal (worst case)
  - A transaction certainly aborts if an add/remove is performed on an already-traverse node
  - **Modifying head's pointer aborts all other transactions** (worst case)
- PROS
  - **Contains** (highest % of operations) transactions can freely **run in parallel**
  - Contains and add/remove can run in parallel if respectively accessing first and second half of list
- **Would a more fine-grained approach help?**
  - Can substitute locks in the fine-grain linked list implementation saw during lectures
  - **BUT** → Quantity of transactions will dramatically increase with **O(N)** for N nodes
    - Would the HTM backend scale well?

Summary

- HTM and lock elision paradigm can be used when lock-free implementation is infeasible/too cumbersome to write but we want to considerably speed up an existing coarse-grained lock implementation
- Simpler, declarative interface for managing concurrency
- Reducing data contention inside HTM transaction is crucial in order to fully leverage its potential
- At the moment, software and programming language-level implementations do not provide any guarantees whether underlying HTM technology will be actually used → for such guarantees, a lower-level framework, such as RTM, is needed
- Creating more and smaller transactions doesn’t scale in TSX → The abortion rate increases!
- Few CPU models currently provide HTM support

Q & A