Design and analysis of concurrent data structures using Hardware Transactional Memory

Jason Dellaluce
jdellalu@uci.edu

Michal Guzek
mguzek@uci.edu

Abstract — *Hardware Transactional Memory (HTM) is a relatively recent mechanism for optimizing concurrent code that utilizes coarse-grained locks. Due to lock elision phenomenon, CPUs are now able to speculate about concurrent code execution within transactions and allow those transactions to commit in the event of no data races. That way, if an implementation of lock-free version of an algorithm is infeasible or otherwise cumbersome, we can still gain a significant speed-up just by fine-tuning coarse-grained lock version. This paper presents performance analysis of multiple C++11 implementations of two data structures. Some of those implementations use HTM and, as the analysis shows, they are viable alternatives to coarse-grained lock versions.*

1. Introduction

Coarse-grained lock programming is one of the most approachable paradigms in the concurrency realm. It allows to surround multiple statements with the same single lock and thus makes for easier reasoning about a concurrent execution. One of the main drawbacks of this approach is the fact that the lock quickly becomes a sequential bottleneck and threads are essentially serialized around it. Hardware Transactional Memory, a recent addition to mainstream CPUs, allows for a lock elision phenomenon under certain conditions and thus speeds up the execution of algorithms based on a coarse-grained lock because the transactions can now run in parallel. HTM provides direct hardware support to its predecessor, Software Transactional Memory (STM). This paper focuses on Intel TSX's RTM implementation of HTM.

2. Objective

The purpose of this paper is to present performance analysis of two data structures with multiple implementations in order to assess general utility of HTM. The first data structure is a HashSet with two implementations: the first one revolves around a coarse-grained lock and the second one uses Intel's RTM. The second data structure is a set implemented as a linked-list and its implementations use the following approaches: a coarse-grained lock, GNU libitm, Intel's RTM and lock-free algorithms.

3. Execution environment

The data structures were implemented in C++11 and compiled with g++ 7.5.0 on Ubuntu Linux x64. The benchmarks have been executed on Intel Core i9-9880H CPU, which has 8 physical cores and a frequency of 2.30GHz. The chipset supports the TSX instruction set.

4. HashSet

The HashSet data structure was implemented with hash collisions resolved by separate chaining. It consists of a base class `HashSet` on which two the implementations are based.

4.1. HashSet: a coarse-grained lock

A coarse-grained lock approach consists of simply surrounding each method body with a single lock in `CoarseGrainedHashSet` class.

4.2. HashSet: Intel's RTM

The technique based on Intel's RTM mechanism can directly leverage the underlying HTM support. To that end, methods in `HashSetGCC_RTM` class follow the RTM template advised by Intel. After an unsuccessful transaction start attempt, a thread sleeps for a random number of nanoseconds (up to 50 ns) in order not to retry a transaction too quickly in case some other thread still holds the physical lock. Higher thresholds (above 50 ns) caused RTM performance to degrade significantly. When the number of attempts has reached the threshold (being equal to three, in a HashSet setting) or a transaction cannot be retried anymore, a given method falls back on executing the code as if it was surrounded by an ordinary lock. That ordinary lock has to be modified, however, in order for the transaction to be able to place the lock variable in its read-set and speculate about it during its execution. Accordingly, we provided a wrapper class on a standard C++ mutex which explicitly maintains a lock variable in addition to locking and unlocking the underlying lock. Transaction code then refers to that wrapper class' `isLocked()` method in order to read the lock variable itself.

4.3. HashSet: a GNU libitm attempt

We have also made an attempt to implement a HashSet by using C++ Transactional memory features delineated in a C++ working draft from 2015. g++ supports those constructs starting from version 4.7 with `-fgnu-tm` flag, by means of its libitm library. However, that attempt failed due to several drawbacks that C++ TM implementation has:

- In order to enclose multiple statements in an `atomic` block, those statements, and the functions they call, must be `transaction_safe` (C++ mandates that rule because the transaction code cannot manipulate `volatile` variables of some kind).
- g++ still reports internal compiler errors and asks for submitting a bug report while using more advanced features, especially `transaction_safe_dynamic` function specifiers for virtual functions.
- Finally, even when a transaction succeeds, C++ TM mechanism provides no guarantees whatsoever that the underlying HTM support will be leveraged, even if the CPU supports it. That is, a transaction might just as well be executed by using higher-level approaches such as STM, which do not provide HTM-like performance.

As per the first bullet point, the majority of functions in the standard library are not marked with `transaction_safe` specifier and neither is the exception that the global `new []` `operator` might throw. Hence, we were not able to implement transaction-safe HashSet buckets that would support self-resizing when necessary. The only array-like container in C++ standard library that is transaction-safe, as of this writing, is `std::array`, however its template parameter denoting the array size needs to be a compile-time constant. Therefore, we have decided not to implement a HashSet based on GNU libitm library.

5. HashSet performance analysis

The performance analysis for the two HashSet implementations is shown in *Fig. 1* and *2*.

Fig. 1 presents analysis for the typical scenario when HashSet methods have the following distribution: 90% `contains()` calls, 9% `add()` calls, and 1% `remove()` calls.

Fig. 2 presents analysis for the following HashSet methods distribution: 40% `contains()` calls, 50% `add()` calls, and 10% `remove()` calls.

As is evident from the figures, an approach based on Intel's RTM leverages the hardware support for transactional memory and thus provides a significant increase in the throughput over a single coarse-grained lock, due to the lock elision phenomenon.

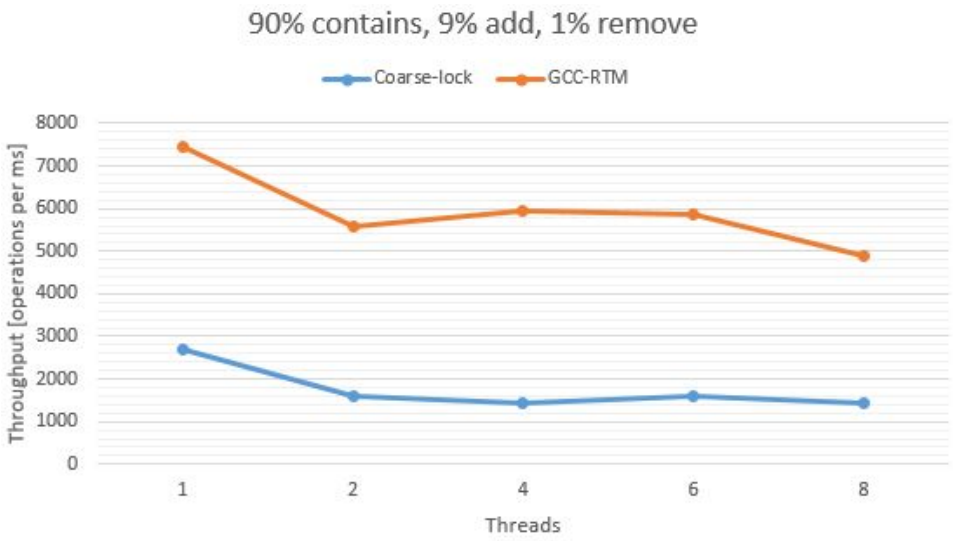

Figure 1: 90% contains, 9% add, and 1% remove

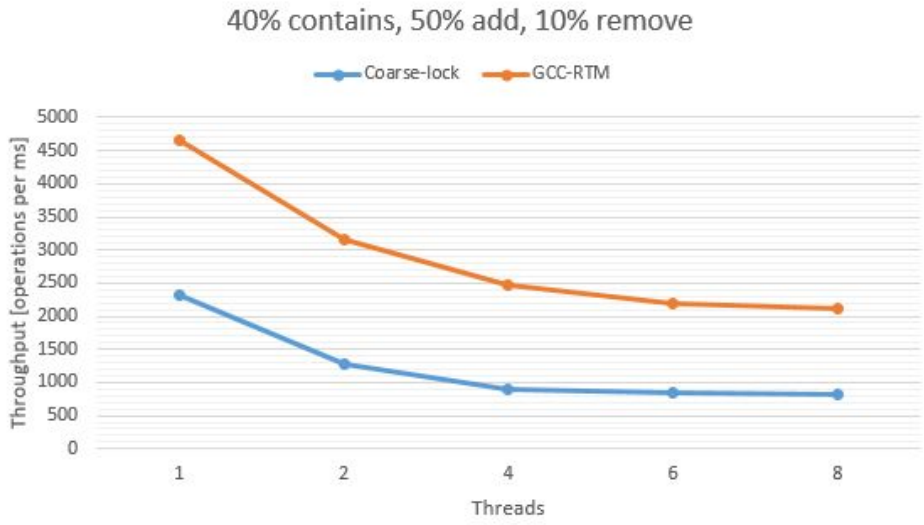

Figure 2: 40% contains, 50% add, and 10% remove

6. Linked-list set data structure

We developed four different versions of the set data structure based on a linked-list in order to compare how HTM-based approach stacks up with the other implementations when it comes to performance. The first implementation is based on GNU libitm implementation of C++ Transactional Memory working draft. C++ TM offers a simple declarative language construct for defining memory transactions, and hides all the complexity of the underlying hardware and software support. If the host machine doesn't support HTM, or in the event of a transaction failure, the library falls back on software transactional memory, effectively proposing a hybrid approach. Our second implementation directly leverages Intel's RTM instruction set to support transactions at the hardware level. The third implementation uses a single coarse-grained lock in order to provide a performance perspective. The granularity of these three implementations had been coarse, namely they serve as a wrapper over a sequential implementation of the linked-list set. The fourth implementation uses lock-free algorithms optimized with a lazy marking strategy for elements removal, again to provide an additional performance perspective.

6.1 Observations on linked-list set implementations

Given our design choices for set implementations based on transactional memory, there are some facts that can be observed:

- Each transaction implies a traversal of the underlying linked list. This causes the read-set of memory locations of each

transaction to grow linearly when nodes of the list are dereferenced. In the worst case, this means that the entire list can be part of transactions' read-set.

- Transactions with overly large read-set or long duration are probabilistically more likely to fail. The data might be also easily evicted from the L1 cache (which is usually 64 KB), thus causing the transaction to fail.
- Modifying a list node pointer, either by adding or removing an element, causes the failure of all the uncommitted transactions that have traversed such a node already.
- The best case scenario is when a node is appended or removed at the end of the list. The worst case scenario is when a modification happens at the list's head node, causing the abortion of all other uncommitted transactions.

7. Linked-list set performance analysis

The charts in *Fig. 3* and *4* provide performance analysis of our four linked-list set implementations. *Fig. 3* shows a workload consisting of about 90% `contains`, 5% `add`, and 4% `remove` operations. *Fig. 4* shows a workload consisting of about 50% `contains`, 35% `add`, and 25% `remove` operations. In both cases, the performance has been measured as the number of operations per millisecond.

While looking at the charts, it is clearly visible that the lock-free implementation still exhibits the best performance. The transactional memory implementation based on GNU libitm demonstrated the poorest performance of all the implementations. Therefore, even though HTM was supported by the underlying hardware, GNU libitm library incurred some additional overhead and is not preferable even to a simple coarse-grain locking. On the other hand, the implementation based on Intel RTM appeared to be slightly better than a simple coarse-grain locking. Obviously, this is dependent on the workload. In fact, increasing the percentage of modifying operations on the linked-list set, caused the RTM implementation to lose performance due to the increased abortion rate of the transactions.

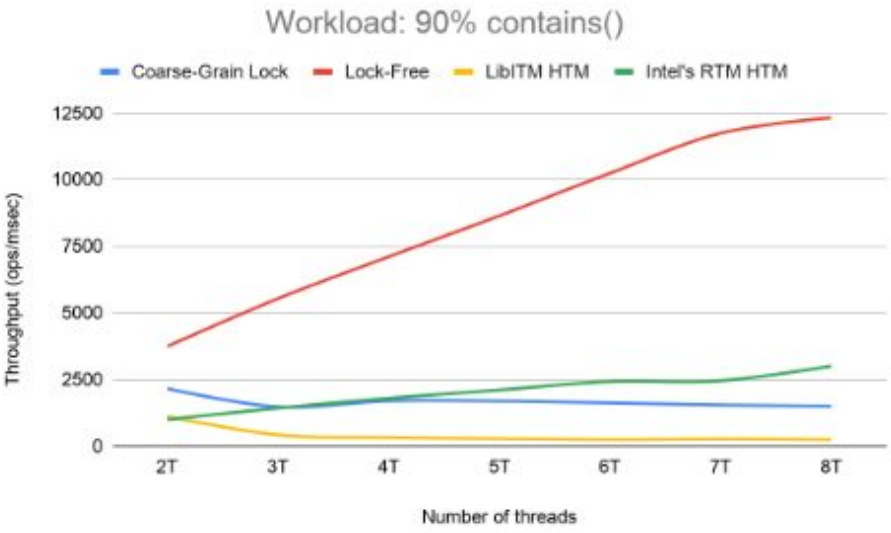

Figure 3: 90% contains, 5% add, and 4% remove

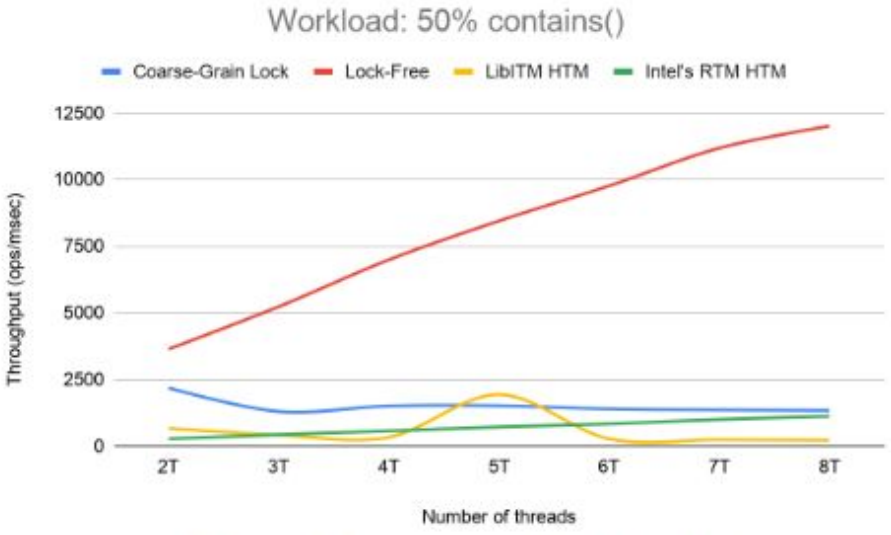

Figure 4: 50% contains, 35% add, and 25% remove

Finally, it is worth discussing the possibility of lowering the volume of set transactions code by designing them with a different granularity. Such an approach, that would require considerable reasoning, is not guaranteed to scale well. For instance, let's consider the case in which a transaction is created each time a single node of the list is read or written, similarly to other well known fine-grain locking designs. In that case, the number of transactions will increase linearly with the length of the list. Each operation invocation performed by threads would cause a very large number of smaller transactions to be created. However, state-of-the-art proposals for HTM (including Intel's TSX) provide a *best-effort* speculative approach, that is, transactions are not guaranteed to ever succeed. In such a worst-case scenario, a large increase of transaction count would cause an overly high abortion rate, forcing the application to execute the fallback path frequently and not leverage transactional memory benefits.

8. Conclusion

As the performance analysis shows, transactions leveraging HTM are an interesting alternative to existing solutions utilizing single coarse-grained locks. If the data contention within transactions remains low, the majority of transaction executions can be parallelized and thus outperform single coarse-grained locks.

On the other hand, data structures that are lock-free still tend to outperform HTM-based approaches. That is, of course, expected due to the fact that HTM algorithms are inherently still blocking.

HTM is yet not supported by many architectures but it seems to become more prevalent in the near future due to the advantages it provides.

9. References

[1] Transactional memory - Wikipedia

[2] Software transactional memory - Wikipedia

[3] Optimistic concurrency control - Wikipedia

[4] Transactional Synchronization Extensions - Wikipedia

[5] Understanding Hardware Transactional Memory - infoq.com

[6] The Case for Hardware Transactional Memory - cs.cmu.edu

[7] Transactional memory - cppreference.com

[8] Technical Specification for C++ Extensions for Transactional Memory, 2015

[9] Transactional Memory - MODERNES C++

[10] TransactionalMemory - GCC Wiki

[11] Hans-J. Boehm - Transactional Memory in C++

[12] Intel's TSX support - GCC

[13] TSX anti patterns in lock elision code - Intel

[14] TSX fallback paths - Intel

[15] Intel TSX - ARCS004

[16] Transactional Synchronization with Intel® Core™ 4th Generation Processor - Intel

[17] Coarse-grained locks and Transactional Synchronization explained - Intel