Brief Announcement: HTM-Assisted Combining

Alex Kogan, Yossi Lev

Brief Announcement: HTM-Assisted Combining

Alex Kogan Yossi Lev

Oracle Labs

Abstract. We present `HCF`, an algorithm that uses hardware transactional memory to combine operations that are conflicting with each other, while allowing other operations to run concurrently.

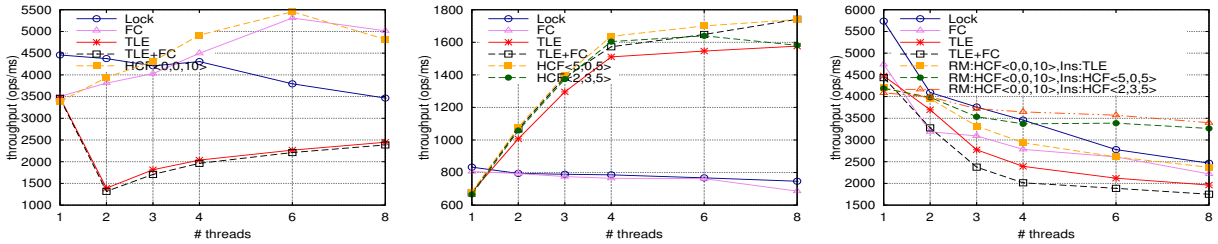
1 Introduction

Transactional lock elision (TLE) [2] and flat combining (FC) [1] were in the focus of recent work on improving concurrency in lock based programs. TLE uses hardware transactional memory (HTM) to execute a critical section *speculatively* using a hardware transaction while confirming that the lock associated with the critical section is not held. Thus, as long as no thread acquires the lock due to consistent failures of speculative attempts, multiple threads can run the critical section concurrently. FC, on the other hand, is effective when *many threads are trying to acquire the lock*; the thread holding the lock (denoted as *combiner*) helps threads that are waiting for it by executing operations on their behalf. This approach improves performance not only because of better cache locality, but also because many data structures allow optimizing a sequence of operations either by packing them together into a more compact operation, or via elimination of one operation with another. Therefore, while TLE can significantly improve performance for data structures whose operations rarely conflict with each other (e.g., a binary tree), FC is more beneficial for data structures whose operations are inherently conflicting (e.g., a stack or a queue).

In our work we explore how to combine these two techniques and provide a "best of both words" solution. A trivial solution can use the original FC algorithm if and when threads that use TLE fail to the lock; unfortunately, as we demonstrate, this approach is rarely helpful because it still prevents other threads, that are not waiting for the lock, to execute the critical section. We therefore introduce the *HTM-assisted Combining Framework* (`HCF`), that enables multiple (combiner and non-combiner) threads to access the data structure concurrently using HTM. Our algorithm allows multiple threads to act concurrently as combiners for different kinds of operations, and hence is well suited for data structures with operations of different nature. For example, with a double ended queue (deque), we expect operations on one end of the queue to conflict with each other, while operations on opposite ends of the queue are unlikely to conflict. Another example is a priority queue, where we expect all RemoveMin operations to conflict with each other, while Insert operations can still run in parallel with all other operations. With `HCF`, we can associate the operations on each end of the deque with a different combiner, and with the priority queue, use TLE for Insert operations while concurrently combining RemoveMin operations. Importantly, due to the use of HTM, this is achieved using a simple sequential implementation of the data structure protected by a global lock, without the need to reason about fine-grained synchronization. In particular, the choice of how many combiners to use and which operations to combine can only affect performance, not correctness.

2 Overview of the `HCF` Algorithm

With `HCF`, each operation Op is associated with a publication array, PA(Op), that has at most one thread executing as a combiner for the operations held in PA(Op). Using multiple publication arrays allows concurrent execution of operations that are unlikely to conflict with each other, even if some of them are executed by combiner threads. A thread T executes Op in one of four phases that it attempts one after another until Op is completed. The first `OwnerPreAnnounce` phase simply tries to execute Op using a hardware transaction (TX). If failed (perhaps multiple times), the second `OwnerPostAnnounce` phase keeps trying to execute Op



(a) RemoveMin only (b) Insert only (c) 50%-50% mix

Fig. 1: Skip-lists-based priority queue throughput results. Experiments were done on an Intel Haswell (Core i7-4770) 4-core hyper-threaded machine running at 3.40GHz and powered by Oracle Linux 7.

using a TX, but only after announcing it in PA(Op), *and* only as long as the operation is not being helped by a combiner thread. If T does not complete Op in the `OwnerPostAnnounce` phase, it continues to the `CombinerHTM` phase, where it becomes a combiner, and tries to execute a subset of operations in PA(Op) (including Op) using one or more TXs. Finally, if all three phases fail to execute Op, T acquires the data-structure lock at the `CombinerLock` phase, and applies the subset of operations left while holding the lock. The last two phases allow any data-structure specific combining and elimination techniques that can reduce the contention on the main data structure, and help executing operations faster. Critically, like with TLE, any TX run by `HCF` respects the data structure's lock, testing that it is not held and aborting otherwise.

3 Performance Overview

We denote by `HCF`<X,Y,Z> a variant of the `HCF` framework that executes up to X,Y and Z HTM trials in the `OwnerPreAnnounce`, `OwnerPostAnnounce` and `CombinerHTM` phases, respectively. We note that `HCF`<0,0,0> that helps all operations in the publication array is equivalent to the original FC algorithm [1], and `HCF`<X,0,0> that helps only the combiner's own operation is equivalent to the TLE algorithm [2].

We evaluated a priority queue data structure, implemented as a skip-list, so we can efficiently combine N RemoveMin operations by chopping off the first N elements in the list. Using the `HCF` framework, we let RemoveMin and Insert operations to use separate publication arrays. For RemoveMin operations, we used a `HCF`<0,0,10> variant, as we expect them all to conflict with each other, and they can be efficiently combined. For Insert, we experimented with either TLE (`HCF`<10,0,0>), `HCF`<5,0,5>, or `HCF`<2,3,5> variants.

Throughput results are presented in Figure 1. As expected, when running only RemoveMin operations, our `HCF` algorithm achieves results that are competitive with those of FC. TLE, on the other hand, performs poorly as all the operations conflict with each other. Even the simple TLE+FC solution, that uses the FC algorithm when TLE fails to the lock, does not provide any benefit. On the other hand, when running only Insert operations, FC performs poorly while TLE is doing pretty well, but is still outperformed by our `HCF` variants. This is because with `HCF`, conflicting operations serialize among themselves at the `CombinerHTM` phase, but can run concurrently with other non-conflicting operations. The biggest advantage of `HCF`, however, is evident in the 50%-50% operation mix experiment, where the variants that use two different configurations of `HCF` for Insert and RemoveMin outperforms all other mechanisms (including the simple TLE+FC variant) by a large margin. This demonstrates one of the most important benefits of our algorithm: the ability to easily apply different combining policies in parallel for different operations executed on the same data structure.

References

1. Hendler, D., Incze, I., Shavit, N., Tzafrir, M.: Flat combining and the synchronization-parallelism tradeoff. In: Proceedings of the ACM Symposium on Parallelism in Algorithms and Architectures (SPAA). pp. 355–364 (2010)
2. Rajwar, R., Goodman, J.R.: Speculative lock elision: Enabling highly concurrent multithreaded execution. In: Proceedings of the 34th Annual ACM/IEEE International Symposium on Microarchitecture. pp. 294–305 (2001)