Assignment 2 (CS 798: Multicore programming—Fall 2018)

Due date: November 14 (tentatively)

Submit: a single ZIP file on Piazza, including a PDF file in your ZIP.

Download and setup

- You will need hardware transactional memory (HTM) support to complete the assignment. If you don't have HTM support in your processor (see `htm_hello_world` below to test for support), you will need to download the appropriate Intel SDE for your platform[1], and use it to run your program[2].
- Do your development on your machine, on Intel SDE if needed. You will run your final experiments on powerful servers with HTM. I am currently trying to arrange access to three such servers for all students enrolled in the class.
- Download the starting C++ code from `http://tbrown.pro/multicore/a2code.zip`.
- `make` creates three binaries: `htm_hello_world`, `benchmark_kcas` and `benchmark_set`.
- `htm_hello_world` is what it sounds like. Run it to see if you have HTM support. It should output `committed empty hardware tx successfully` (and no error messages) if you have HTM support. The other two binaries are described below.
- Note the `-mrtm` compilation flag. This compiles for an architecture that supports Intel's HTM. I've already included this flag, but if you do your own HTM development in your own projects, note that you need this flag to use HTM.

Questions about Hash Tables

These first few questions are about building non-trivial, full featured hash tables using HTM. That means insertion, deletion and expansion. (We aren't worrying about contains, simply because it's easy once you have the other operations.) We will be working with `benchmark_set`, which is a simple test harness for evaluating *sets* that provide insert and delete operations. In this benchmark, n threads repeatedly perform 50% insertions and 50% deletions of uniform random keys drawn from a fixed key range for s seconds. Run `benchmark_set` with no arguments to see usage instructions.

(Note: as a sanity check, we do some checksum validation on the data structure at the end of each execution [see line 149 of `benchmark_set.cpp`]. It might be interesting to think about how this checksum validation works.)

[1]Intel SDE site: `https://software.intel.com/en-us/articles/intel-software-development-emulator`

[2]Example, to run `./benchmark_kcas [args]`, you use a command like `sde -tsx -- ./benchmark_kcas [args]`.

1 Explaining the performance of a hash table

To start you off, I provide a simple lock-free hash table with insert and delete but no expansion (one of the implementations we presented in class) in the file `set_hashtable_lockfree.h`.

Experiment with this algorithm by running with different command line arguments.

- What kinds of results do you see? Are there any performance anomalies, or strange emergent behaviours that this hash table exhibits in this test?
- To aid in your exploration, try collecting and printing some extra information about your executions in the function `printDebuggingDetails()` in `set_hashtable_lockfree.h`). (This function is executed at the end of an execution, after all other threads have been shut down.)
- Explain in a few sentences what is happening to the table throughout an execution of the micro benchmark, and how it affects performance, and the usefulness of the data structure.

2 Hash table with expansion

Of course, hash table expansion is important. But, it's hard to implement expansion using CAS. This question is about using HTM to simplify the problem of expanding a hash table. Recall transactional lock elision (TLE), which we covered in class.

Implement a TLE-based hash table with insert, delete and expansion.

- I've provided a skeleton for the required data structure interface in `set_unfinished.h` that you can use as a starting point. (See the bottom of `benchmark_set.cpp` to integrate your implementation into the benchmark.)
- Don't worry about making the expansion itself very efficient in this first implementation (single thread doing the expansion is fine).
- Expansion should happen (as part of insertion—invisible to the user) whenever table is more than half occupied (where tombstones count as occupied), and should *double* the table size.
- Hint: how can you *efficiently* determine *when* you should expand?

Experiment with your expandable hash table to understand its behaviour.

- How many transactions succeed vs abort in your executions? (Remember to avoid contention and false sharing when gathering such statistics.)
- How many (successful) expansions happen on the fast path vs the fallback code path? Why?
- Modify your code so that a time stamp is printed to stdout whenever table expansion occurs. (Can you do this without noticably slowing down your hash table?) Do any patterns emerge in the expansion times? Explain. (Careful: do not print or make any syscalls while inside a transaction, because this will abort it.)

3 Collaborative hash table expansion

Implement

- Use OpenMP to parallelize expansions that occur on the *fallback code path* (when the global lock is held). In other words, if an expansion happens on the fallback code path, it should be performed by several threads, who collaborate to create the new, larger table.
- Remember to use synchronization as appropriate to guarantee that keys are correctly copied to the new table (ensuring that no keys are lost, etc.).

Experiment

- How much does this improve performance vs your previous TLE-based hash table?

Questions about KCAS

The remaining questions are about implementing KCAS. We will be working with `benchmark_kcas`, which is a simple test harness for evaluating KCAS implementations. In this benchmark, KCAS is used to implement a silly array-based data structure with one operation, called *atomicIncrementRandomK*, that picks a random index i in the array, and atomically increments k consecutive locations in the array starting with i. In each execution of the benchmark, n threads repeatedly perform *atomicIncrementRandomK* operations for s seconds. (This is similar to a synthetic benchmark we saw in class while discussing KCAS performance.) Run `benchmark_kcas` with no arguments to see usage instructions.

4 TLE-based KCAS

To start you off, I've provided `kcas_hashtable_lockfree.h`, which contains a lock-free implementation of KCAS (using my descriptor reuse technique that I only briefly mentioned in class). Its implementation is rather complicated, and you don't have to look at it. Incidentally, this implementation should be competitive with state of the art KCAS implementations.

Implement KCAS using TLE.

- I've provided a skeleton for the required KCAS interface in `kcas_unfinished.h` that you can use as a starting point. (See the bottom of `benchmark_kcas.cpp` to integrate your implementation into the benchmark.)

Experiment

- Compare the performance of your TLE-based KCAS with the lock-free KCAS algorithm.

5 Try-lock based KCAS

Implement the try-lock based KCAS algorithm we discussed in class.

- You can use the try-lock implementation in `util.h`.
- Recall that to ensure correctness in KCASRead, we have to guarantee that the address is not locked when we read the address. To guarantee this, just do the “dumb thing” and acquire a lock on the address before reading it in KCASRead.

Experiment

- How does the performance of this try-lock based algorithm compare with the other algorithm?
- How much does the locking in KCASRead hurt performance? To test this, try running *without* the lock in KCASRead. Your answers might be incorrect, but you will still get performance results.
- Can you find a workload where checksum errors (failed validation at the end of a trial) occur if you *don’t* lock in KCASRead?
- Use HTM to guarantee the required correctness property in KCASRead more efficiently (than locking). How much does this improve performance?

Next assignment, you’ll implement one or more data structure(s) using KCAS, and you’ll be able to see how improving KCAS performance affects data structure performance, using your KCAS implementations.

Bonus (optional)

- Implement the version-locking variant of KCAS discussed in class, and show how much this improves performance vs the try-locking KCAS that acquires a lock in KCASRead. (How does this compare to the HTM-based fix to KCASRead?)