Zexuan Wang (99885173)

Q1. Because un-necessary code can result low cache hit rate when we fetch instructions and thus lowering program runtime performance. This will lead to inaccurate benchmarking result when we do Part 4.

Q2. TM is very easy to implement given the global lock implementation. All one needs to do is to replace "pthread_mutex_lock" to "__transaction_atomic{}".

Q3. No. Because in order to create list-level locks, we need to know the internal implementation of the hashtable in order to find the list where a particular key sits under and lock that list ONLY.

Q4. No. Changing "lookup" and "insert" functions does not allow us to lock the "count++" statement, which is a write instruction followed by a read instruction and should be locked in multithreading. Solely changing "lookup" and "insert" functions do not provide us the ability to access to the frequency counter in the hashtable.

Q5. No. This allows the possibility to create an atomic function to insert a new value into the hashtable by combining the read (lookup) and write (insert) into one function. However, this does not solve the potential racing condition in "count++" from Q4.

Q6. Yes. We can lock the sample "count++" within "lock_list" and "unlock_list" and also lock the insert() and lookup() statements. This way, all potential racing conditions have been addressed.

Q7. TM is a lot easier to implement comparing to the list-level locks and TM has a better runtime benchmarking performance comparing to list-level locks as well.

Q8. Pros of using multiple private hashtable copies:

- Solves the data-racing problem in the multithreaded program.
- Faster runtime within each thread since: 1) Do not need to wait for data from other threads; 2) possibly smaller cache collision list to search from; 3) Better cache hit probability for accessing the private hashtable since less entries comparing to the global one.

Cons of using multiple private hashtable copies:

- Larger memory consumption for the extra copies of private hashtables.
- Extra runtime for the main thread to combine those local private hashtables into the global one.
- May not scale well when the user decides to parallelize it with more than four cores since it will requires multiple copies of the hashtable and the overhead combining them may become significant.

Q9. Benchmarking result with samples_to_skip = 50:

Note: - All values measured as an average of 5 runs and measured using "/usr/bin/time" with the wall-clock / real time.
- Can be produced by directly running "bash benchmarking.sh"

Number of Threads	1	2	4
randtrack	10.34 sec	N/A	N/A
randtrack_global_lock	10.51 sec	5.70 sec	4.98 sec
randtrack_tm	11.27 sec	5.43 sec	2.96 sec
randtrack_list_lock	10.66 sec	5.58 sec	3.11 sec
randtrack_element_lock	10.65 sec	5.52 sec	2.92 sec
randtrack_reduction	10.33 sec	5.23 sec	2.75 sec

Overload for each of the parallel version is reported in the following table:

	Overhead
randtrack	N/A
randtrack_global_lock	1.02 (10.51 sec / 10.34 sec)
randtrack_tm	1.09 (11.27 sec / 10.34 sec)
randtrack_list_lock	1.03 (10.66 sec / 10.34 sec)
randtrack_element_lock	1.03 (10.65 sec / 10.34 sec)
randtrack_reduction	1.00 (10.33 sec / 10.34 sec)

Q10. For all approaches, the runtime performance increases as we increases the number of threads used for parallelization. This indicates that the overhead for parallelization is relatively small comparing to the speedup it offers.

Q11. The following tables summarizes the experimental result after setting samples_to_skip = 100:
Note: - All values measured as an average of 5 runs and measured using "/usr/bin/time" with the wall-clock / real time.
- Can be produced by directly running "bash benchmarking.sh"

Number of Threads	1	2	4
randtrack	20.47 sec	N/A	N/A
randtrack_global_lock	20.63 sec	11.03 sec	5.95 sec
randtrack_tm	21.36 sec	10.48 sec	5.57 sec
randtrack_list_lock	20.81 sec	10.63 sec	5.72 sec
randtrack_element_lock	20.79 sec	10.62 sec	5.68 sec
randtrack_reduction	20.47 sec	10.32 sec	5.52 sec

Overload for each of the parallel version is reported in the following table:

	Overhead
randtrack	N/A
randtrack_global_lock	1.01 (20.63 sec / 20.47 sec)
randtrack_tm	1.04 (21.36 sec / 20.47 sec)
randtrack_list_lock	1.02 (20.81 sec / 20.47 sec)
randtrack_element_lock	1.02 (20.79 sec / 20.47 sec)
randtrack_reduction	1.00 (20.47 sec / 20.47 sec)

Difference comparing to samples_to_skip = 50:

- Smaller ratio for overhead, this is because more execution time spending on skipping the samples and thus smaller proportion of runtime is actually synchronizing the locks as overhead.
- Randtrack_tm has better performance than randtrack_element_lock when samples_to_skip = 100, but this was not the case for samples_to_skip = 50. This may also attribute to the fact that randtrack_tm has a large overhead when samples_to_skip = 50 but its performance improvement leverages out the overhead at samples_to_skip = 100.

Q12. I would suggest OptsRus to do the following with their software: Check how many cores the customer has on their machine, if it's less than four cores, then run randtrack_reduction; otherwise, run randtrack_tm.

Reason for running randtrack_reduction if the machine has less than four cores:

As shown from the experiment, it has the fastest runtime although with tradeoff in a constant multiple of memory overhead.

Reason for running randtrack_tm if the machine has more than four cores:

Comparing to other locking mechanisms, randtrack_tm has the best flexibility and code maintainability. For example, if OptsRus decides to modify / add operations to the hashtable in the future, all randtrack_tm needs to do it to wrap "__transaction_atomic{}" around the new critical session without modifying the underlying code for hashtable / list since developers are pretty expensive nowadays. Randtrack_reduction may not scale well with more parallelization as discussed previously.