Q1.Why is it important to #ifdef out methods and data structures that aren't used for different versions of randtrack?

Methods and data structures that aren't used occupy memory from the data and code. By surrounding these with #ifdef, the compiler doesn't compile them and makes the program faster by saving memory.

Q2. How difficult was using TM compared to implementing global lock above?

It required a little less code that implementing a global lock as the global lock creation, initialization, lock and unlock code was eliminated. The critical section already identified in the global_lock code, surrounded by lock and unlock code lines, was replaced by __transactional_atomic{}. It's a little easier in comparison.

Q3. Can you implement this without modifying the hash class, or without knowing its internal implementation?

No, this cannot be implemented without modifying the hash class or without knowing its internal implementation. This requires implementing locks to each list in the hash table, which can only be accessed at the list level. This can be only be achieved by modifying the hash class.

Q4. Can you properly implement this solely by modifying the hash class methods lookup and insert ? Explain.

No, this cannot be done as there is a chance for race condition and inconsistency. For instance, two threads can lookup the value for the same key one after the other, as a result of list lock, in the hash table and find it's empty. They then both proceed to insert a value into the hash table one after the other, as a result of list lock. This shall result in double insertion, which is unintended.

In addition, an inconsistency on the counter (s->count++) may occur if two threads are accessing the same element and incrementing a counter. The outside counter cannot be made atomic solely by modifying internal lookup and insert functions.

Q5. Can you implement this by adding to the hash class a new function lookup_and_insert_if absent? Explain.

Yes the first problem mentioned in Q4 can be solved by the implementation of this new function through performing lookup and insert together atomically. However, since this function is implemented at the list level, the counter problem is not still resolved.

Q6. Can you implement it by adding new methods to the hash class lock_list and unlock_list ? Explain. Implement the simplest solution above that works (or a better one if you can think of one).

Yes these methods allow implementing locks surrounding the counter section and solves the second problem mentioned in Q4. In addition, they also solve the first problem if the whole critical section is locked using them as this only locks one list.

Q7. How difficult was using TM compared to implementing list locking above?

Implementing TM was much easier than list locking above as there was no need to modifying internal implementation of the hash.h by adding lock_list and unlock_list functions.

Q8. What are the pros and cons of this approach (reduction)?

The pro for this approach is that it eliminates race conditions. Since each thread is modifying its own copy of the hash table, there is no need for any locks or waits which increase performance. The con, however, is that this takes up n times more memory space than other approaches, where n is the number of threads. This also results in an increasing number of misses for reads and writes to cache, which can be expensive. This becomes a larger problem for larger sizes of the hash table, if an increasingly more amount of tests are done.

Q9. For samples to skip set to 50, what is the overhead for each parallelization approach? Report this as the runtime of the parallel version with one thread divided by the runtime of the single-threaded version.

	Runtime with num_threads: 1	Runtime with num_threads: 2	Runtime with num_threads: 4	Overhead
randtrack	10.42			-
randtrack_global_lock	10.53	5.81	4.85	1.011
randtrack_tm	11.25	9.64	5.65	1.080

randtrack_list_lock	10.67	5.78	3.25	1.024
randtrack_element_lock	10.68	5.49	3.02	1.025
randtrack_reduction	10.41	5.31	2.84	0.999

Each runtime is reported in seconds. Overhead is calculated by the runtime of the parallel version with one thread divided by the runtime of the single-threaded version for each approach. Samples to skip is set to 50.

Q10. How does each approach perform as the number of threads increases? If performance gets worse for a certain case, explain why that may have happened.

As the number of threads increase, each approach performs better and better. This is due to parallelizing the operations using the threads.

Q11. Repeat the data collection above with samples to skip set to 100 and give the table. How does this change impact the results compared with when set to 50? Why?

	Runtime with num_threads: 1	Runtime with num_threads: 2	Runtime with num_threads: 4	Overhead
randtrack	20.51			-
randtrack_global_lock	20.64	10.91	5.83	1.006
randtrack_tm	21.40	14.62	8.27	1.043
randtrack_list_lock	20.79	10.83	5.84	1.014
randtrack_element_lock	20.81	10.53	5.78	1.015
randtrack_reduction	20.51	10.31	5.52	1

Each runtime is reported in seconds. Overhead is calculated by the runtime of the parallel version with one thread divided by the runtime of the single-threaded version for each approach. Samples to skip is set to 100.

With samples to skip set to 100, the runtimes have doubled for single thread performance, and increased by less than a factor of 2 for 2 and 4 threaded performances. This is due to the inner loop of the randtrack function, which iterates once for each sample to skip and thus iterates more when set to 100.

Q12. Which approach should OptsRus ship? Keep in mind that some customers might be using multicores with more than 4 cores, while others might have only one or two cores.
Based on the performances measured as seen in the table, we think OptsRus should ship with the randtrack_reduction version. This is the best approach provided the hash tables don't take too much memory, since for each multithreaded performance, it performs better than the other 4 approaches. For the case of the single threaded runtime, it has almost no overhead at all as well, compared to the original randtrack - with reported overhead of 0.999 and 1 for 50 and 100 samples_to_skip. As such, randtrack_reduction provides the best performance regardless of how many cores customers might be using.