Homework 3 JMM Report

Zhehao Wang <zhehao@cs.ucla.edu> 404380075

Environment information (lnxsrv07.seas.ucla.edu)

Java version "1.8.0_51"
CPU: Intel(R) Xeon(R) CPU E5-2640 v2 @ 2.00GHz
Memory: 64GB

Results and discussion

In this assignment we implemented “Unsynchronized”, “BetterSorry”, “GetNSet”, and “BetterSafe” models for testing Java shared memory performance races.
The data race comes from a) non-atomic increment and decrement operators. One scenario’s illustrated in Figure 1, in which two increment operations on the shared memory will result in only one increment of the variable X.

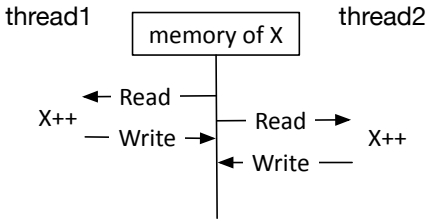

Figure 1. Data racing scenario

Another problematic scenario is b) values in the shared memory can fall below the lower bound, or go beyond the upper bound, if multiple threads passed the boundary check before any of them writes the incremented or decremented value. The program may enter a dead loop if all elements are at the boundaries, or beyond them, so we introduced an arbitrary timeout to make sure the program exits.

The unsynchronized implementation suffers from a), and we can hardly maintain 50% reliability with 4 threads and 100 transitions; b) also occurs frequently, with the chance of successful execution getting higher as larger upper limits are given.

Our GetNSet implementation uses AtomicIntegerArray, which guarantees data race as described in scenario a) would not happen. Scenario b) could still happen. Interestingly, b) happens more often in large number of threads, low amount of transitions cases. (e.g. 16 threads, 30 transitions)

The BestSafe implementation uses ReentrantLock, which is usually considered as a more flexible synchronization method than Synchronized. By locking the increment and decrement operations, we can guarantee that scenario a) wouldn’t appear. And by adding the boundary condition check into this critical section, we can guarantee that b) wouldn’t happen either. Thus we have 100% reliability.

BestSafe outperforms Synchronized model in higher contention rate scenarios, with the possible reason being that the ReentrantLock has a smaller scope, and a lighter-weight internal implementation than the synchronized keyword.

Our BestSorry implementation creates an AtomicInteger before increment and decrement, to guarantee that these operations are atomic.
Similar as GetNSet, scenario a) won't happen, while b) may still happen; Thus we consider this approach not 100% reliable, though much better than Unsynchronized. (8 threads 20 transitions has ~5% chance of overflowing)
The performance is similar with GetNSet in general, and slightly better in low contention scenarios. It outperforms BetterSafe in a high contention environment, potentially because AtomicInteger utilizes lock-free concurrency facilities from the hardware.
An interesting observation is that BestSorry's reliability drops as thread number increases and transition number decreases. Also, under the test scenarios listed in the appendix, an array of AtomicInteger performs worse than allocating AtomicInteger when swap is called.

It is worth mentioning that the test results in this assignment are also subject to

1. Time used in picking two random values that can swap. For all models, especially those that are not 100% reliable, "picking time" may have a large variance because of the random numbers chosen, and CPU's execution scheduling of our multiple threads.
2. Java version, as compile time code optimization, and JIT's runtime optimization may be different.
3. The hardware architecture may also influence the test results, as machine code optimization, and hardware's high-level features may be different.

We provide the initial performance measurement and analysis of the 5 models under different scenarios in the appendix. The initial test results with statistics (average, as well as standard deviation, max and min) are shown here (https://docs.google.com/spreadsheets/d/1OGH-9k8wBQ3WAVNs2_Sat6FSs5d8QiAtKaGBxd8zNnI/edit?usp=sharing). (in several sheets each with different controlled variables.)
A test script is used for automating tests under different scenarios.

Appendix

(Unsynchronized model has a very low reliability under any of the test scenarios below, and is often terminated by the arbitrary timeout. Thus we don't show Unsynchronized in our results.)

Tests are carried out with an array of size 5 and maximum value of 100. The average time per transaction comes from the time measured in 20 sequential executions. These tests have 16 threads at most, to avoid frequent context switching in the 8-core, 16-hyperthreading CPU in the test environment.

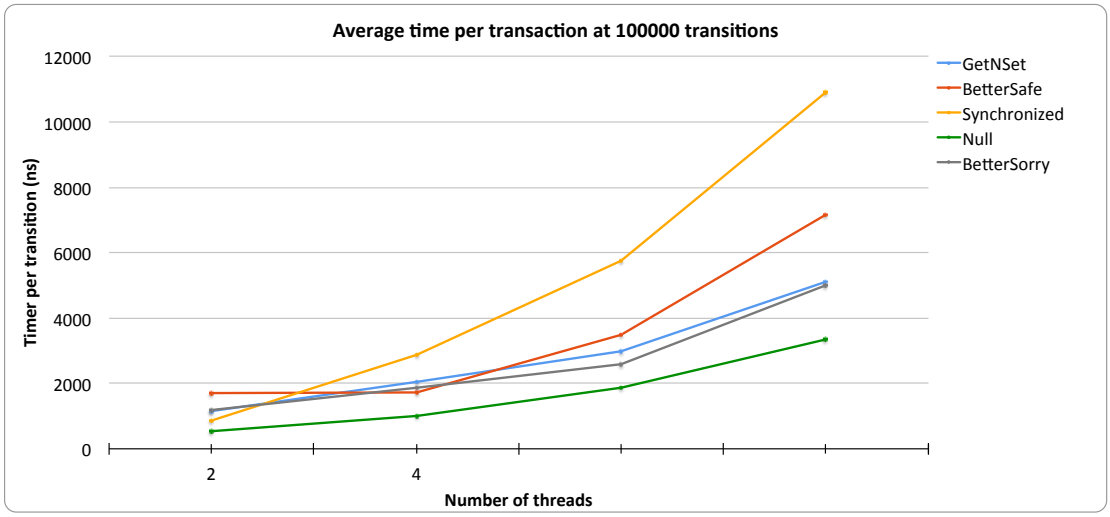

Figure 2. Performance of the five models under 100000 transitions scenario

This result shows that under high contention scenarios with a considerable amount of transitions, the order of performance from worst to best is Synchronized, BetterSafe, GetNSet, and BetterSorry (almost tie).

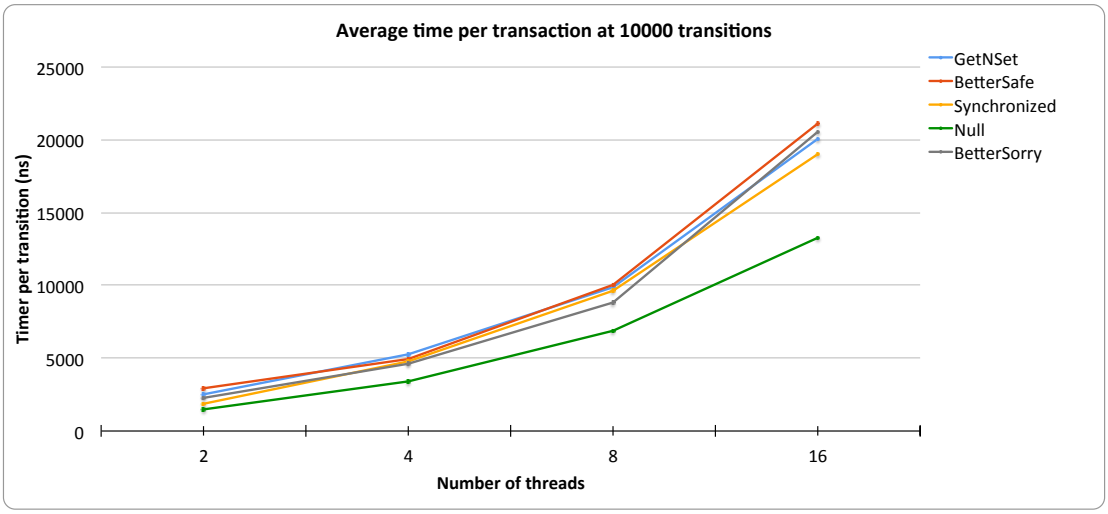

Figure 3. Performance of the five models under 10000 transitions scenario

This result shows that with less amount of transitions, BetterSafe is worse than Synchronized, and GetNSet is worse than BetterSorry.
The average time is also much longer than Figure 1, whose reason could be that in this case, higher speed CPU caches, and other hardware optimizations do not have enough amounts of operations to fully demonstrate their advantage.
BestSorry reported one negative output out of the twenty sequential runs in the 16-thread test in this scenario.

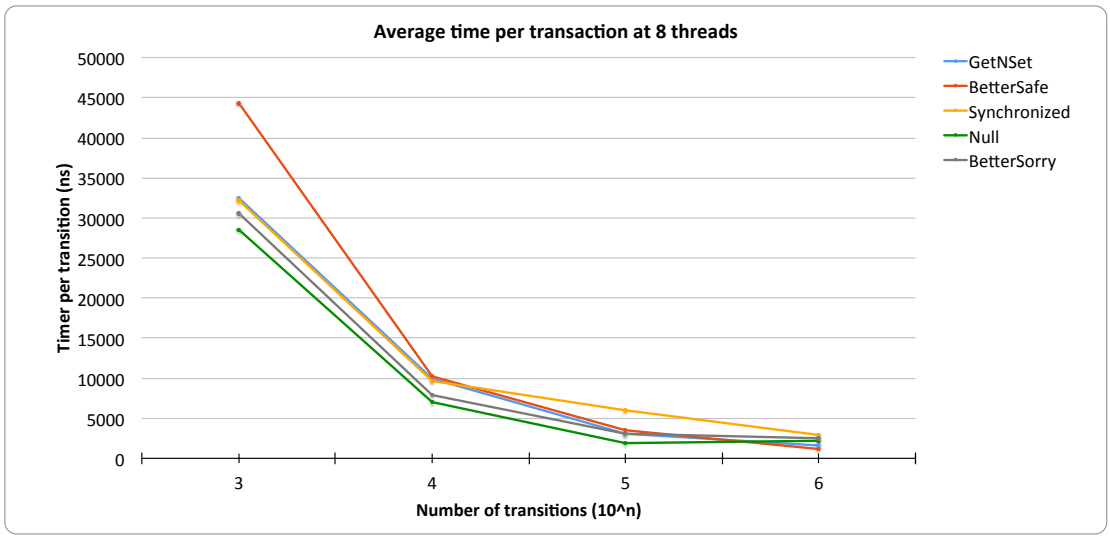

Figure 4. Performance of the five models under 8 threads scenario

In this figure, we use a fixed number of threads and observe the performance change as more transitions are performed. Synchronized becomes more costly as the number of transitions grows. One thing that we find hard to explain is that the performance of BetterSafe and GetNSet becomes as good as Null (here it suggests that they are even slightly better than Null). It's possible that other tasks are carried out at the time of Null test, however, this behavior seems reproducible in tests conducted at different times.