CS5300: PARALLEL AND CONCURRENT PROGRAMMING, Fall 2020

Programming Assignment 3: Wait-Free Atomic Snapshot Implementations Comparing the Solutions of MRSW and MRMW

SAI HARSHA KOTTAPALLI

CS17BTECH11036

GOAL

The goal of this assignment is to implement the two Atomic Wait-Free Snapshots algorithms: the solutions for taking atomic snapshots of MRSW and MRMW registers

DESIGN & IMPLEMENTATION

The design has already been discussed in class with the pseudocode. For implementation, it is mostly reproducing the pseudocode to C++ language. The only difficulty was creating an array of atomic registers. For this, I have created a vector of atomic pointers to Object pointers so that they are trivially constructible and are accepted by c++ standard. Any update would only happen after referencing the atomic pointer so that all register updates are atomic. We also correspondingly free the allocated memory wherever necessary(which was told to be not required later).

Here, collect() is the non atomic act of copying the register values one-by-one into an array. So whenever two collects give the same set of values(the whole identification set) we call this a clean double collect and return what has been collected as the snapshot.

It follows from this and pigeonhole principle that as there are n threads in total and of which n-1 threads which can interrupt, (n+1) collect calls will guarantee clean double collect.

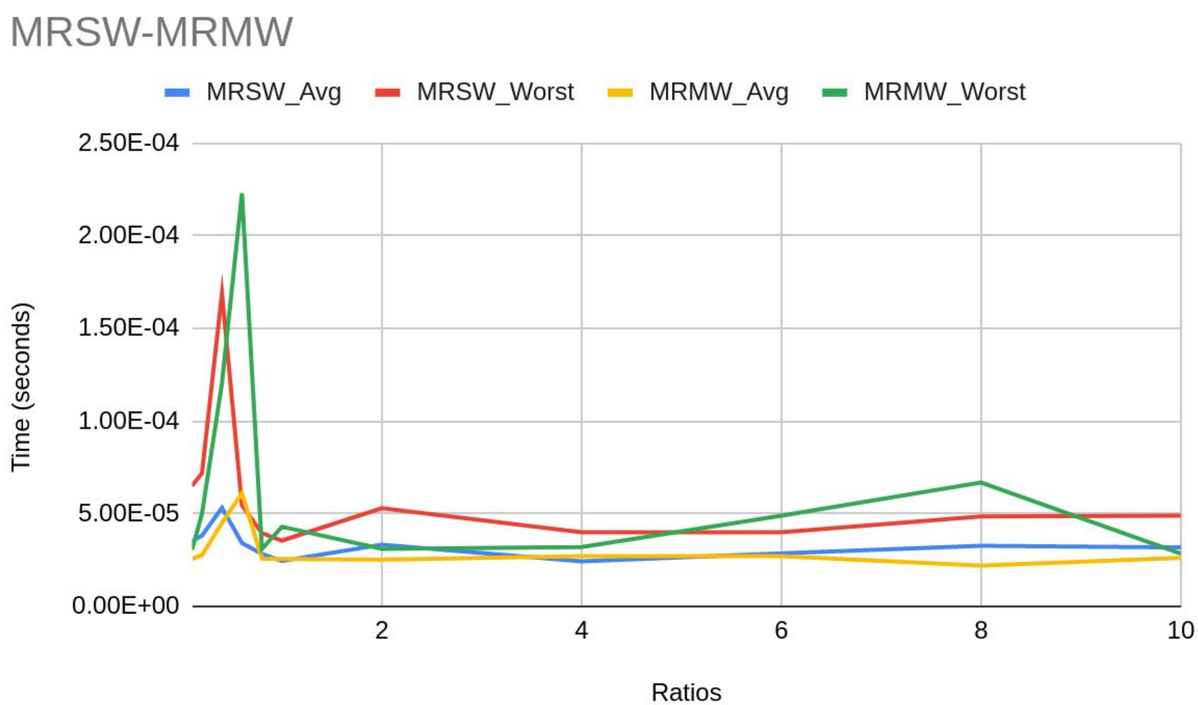
Each update() call helps scan() by storing the snapshot before modification. If same thread moved twice during the snapshot, then thread can use the moving thread's snapshot.

One major change in mrsw and mrmw implementation is that, in mrsw writers can only write to its own register while inn mrmw writers can write to any of the available set of registers.

GRAPHS

Note: Data points are the average of many observations. This is because mrmw writers write to random locations and time taken depends on how distributed the random values are.

Ratio = mew_s / mew_w



OBSERVATIONS

- We notice that peak occurs when ratio ~1 and on either side of extremes it is minimum.

- For left side of extreme, when snapshotter thread delay is smaller than writer delay, it can quickly grab the snapshot via scan() before a writer possibly writes and hence get the clean double collect quickly. This explains why it is minimum here,
- For right side of extreme, when snapshotter thread delay is larger than writer delay, that is writer threads writes its updates frequently. Here, this would cause a higher probability for snapshotter thread coming across a thread which writes twice during the collect comparison, then we would use the thread’s snapshot which moved twice. Hence, here also time taken is minimum.
- Now whats left is when the delay values are similar, that is ratio ~1 which might cause different writer threads to interrupt the collect compare process instead of the same thread. As expected, the peak occurs here(refer graph).
- Comparing the peaks, it seems that MRSW is faster than MRMW implementation. But this is not uniform everywhere possibly due to the fact that MRMW is writing to random locations which might be concentrated or more distributed.