Wait Free Atomic Snapshot Implementations

Sharan Narasimhan, CS20MTECH14003

November 11th, 2020

# 1.Foreword to Evaluators:

I have kept the code explanation concise for the sake of brevity. Please read the code comments for more specific explanations.

The experiments for mr\*w are written to experiments\_mr\*w.txt and the timeline of execution is written to snapshots\_file\_mr\*w.txt.

The helpers.cpp contains common methods used for both snapshots.

# 2. MRSW Snapshot

**Step 1:**

The main() consists of the usual steps such as initialising files, getting input parameters, initialising the snapshot object, creating threads and running the process inside #pragma omp critical. This all is included in Line 165-210 inside main().

**Step 2:**

N+1 threads are created, where the thread with id = n is the snapshot thread, which will execute the while loop to attain a clean snap k times, as seen in figure 1.

Text

Description automatically generated

Figure 1

The snapshot thread obtains a clean snap, writes it to file, computes relevant statistics, and sleep for a random time.

**Step 3:**

All the other writer threads will execute update() indefinitely till the snapshot thread sets stop\_writing to true. After this all threads exit the parallel section and the file streams are closed. All this is encapsulated in figure 2.

Text

Description automatically generatedFigure 2

**The snap\_value class:**

Each atomic register contains an object of type snap\_value which looks like:

Graphical user interface, text

Description automatically generated

Figure 3

**The mrsw\_snapshot\_obj class:**

This class encapsulates the array of atomic<snap\_value> MRSW register and all the necessary functions such as collect(), update() and scan().

Text

Description automatically generated

Figure 4

Text

Description automatically generatedThe update() function gets a clean snapshot, takes the old snap\_value object and creates the new entry to be written to s\_table\_snap\_values as seen below:

Figure 5

The collect() function is fairly straightforward.

The scan() function follows the strategy, “return snap if clean collect is obtained, else steal snapshot of first thread that moved twice”. This logic can be seen below:

Text

Description automatically generated

Figure 6

# 3. MRMW Snapshot

This algorithm is fairly similar to the previous MRSW algorithm except 1. Update() follows the strong freshness policy, 2. We use 2 objects in the MRMW class as explained below.

a) s\_table[m] contains objects of type mrmw\_entry as seen below:

Text

Description automatically generated

Figure 7

b) help\_snaps[n] is an array of clean snaps corresponding to each thread. Its declaration can be seen in line 48 of the below figure.

Text

Description automatically generated

Figure 8

I was forced to use std::array instead of C++ arrays for this algorithm as the atomic<> type does not accept arrays.

The update() follows strong freshness and also includes the location of the register to update along with the value. Apart from that it is fairly straightforward.

The snap() also works the same as before whenever a double move is detected, the scanning thread will take the help\_snap of the array that moved twice. All this logic can be seen below:

Text

Description automatically generated

Figure 9

**4. Correctness:**

Please compile and run mr\*w\_snapshot.cpp and refer to the snapshots\_file\_mr\*w.txt to confirm the correctness of the algorithm.

Please note that writing to these files was done sequentially by each thread to ensure that the timeline was in increasing order of time. Writes to files were done using the function write\_to\_file() in helpers.cpp. This will not affect the snapshot objects behaviour in any way, since writes are only done after an update() or a scan().

**5. Analysis:**

Please refer to experiments\_mr\*w.txt for all the data used for analysis.

**a) MRSW snapshot analysis:**

The parameters were n = 10, k = 50 for all experiments. Each datapoint was obtained by averaging 5 iterations. Since it was not specified how to obtain the u\_s/u\_r ratio, I am keeping u\_s set to 100 micro seconds and varying u\_r.

**a) MRMW snapshot analysis:**

The parameters were n = 10, m = 20, k = 50 for all experiments. Each datapoint was obtained by averaging 5 iterations. Since it was not specified how to obtain the u\_s/u\_r ratio, I am keeping u\_s set to 100 micro seconds and varying u\_r.

**Observations:**

**1. Both MRMW and MRSW have similar average times**

This at first seems odd as we expected MRMW to experience more collisions since now collision occurs through multiple threads. However the probability that a register is updated is still 1/n\_threads even in the case of MRMW. This is the same case for MRSW since at any given time the probability that the snapshot thread experiences a collision is 1/n\_threads also.

Since #collosions is the primary factory affecting time taken for scan(), both the algos will perform similarly.

**2. There is no relation between u\_s/u\_w and avg time**

This is a little odd and hard to explain. I have no concerete reasoning as to why this is occurring.

**3. Worst time is randomly fluctuating:**

This is expected, and we can see the “avg worst time” does not vary a lot.