Comparing Obstruction-free and Wait-Free Snapshot Algorithms
CO21BTECH11004

Implementation: -

- Global variables nW, nS, M, k, muW, muS are created.
- Global atomic variables are created to measure time.
- Time is measured in microseconds using the Chrono library.
- Global Snapshot object is created.
- Random numbers to decide the position and value are generated using uniform_int_distribution, while that for sleep is generated using exponential_distribution using a random library.
- The atomic MRMW array of stampedValue is created by making an atomic wrapper class, where the constructor, copy constructor, copy assignment operator, and != operators are defined.
- For log, a global vector of pair<string, long> is created, which contains the log and time. Each thread maintains a localLog vector and updates the global log before exiting.
- Before writing the log into the output file, it is sorted with respect to time in ascending order and written.

Obstruction-free: -

- Using stampedSnap instead of stampedValue.
- The following changes are made in the update function given in the book: -
    - Instead, each thread updates the value at threadID, The position is passed, so an update is made to that position.
    - Each writer thread maintains a local stamp, which is incremented by 1 and written along with value and threadID.
- In the scan function, stamp and threadID are compared for equality check rather than only stamp value.

Wait-free: -

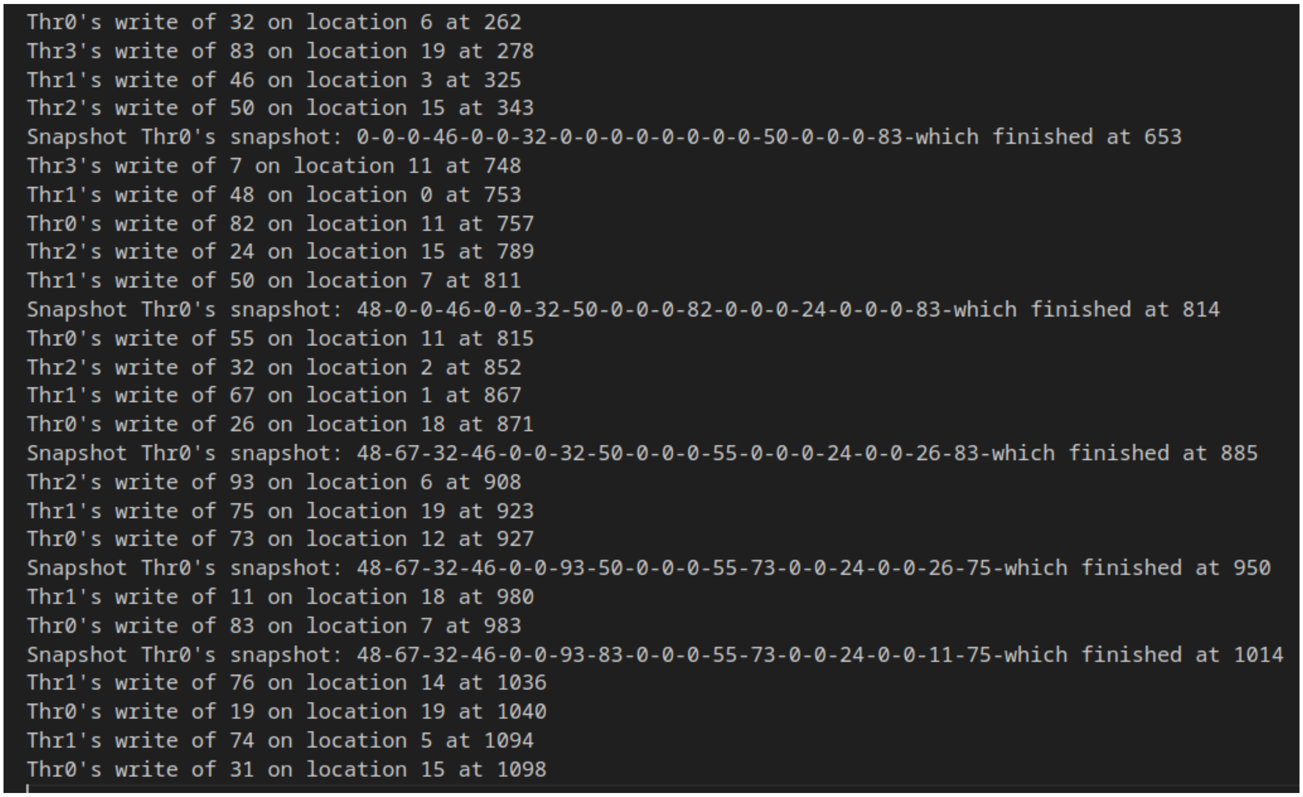
- The algorithm given in the research paper is implemented.
- StampedSnap is used.
- Reg array is a vector of atomic stampedSnap of size capacity.
- The helparray is of nW * capacity.
- Each thread maintains its local stamp, which is incremented by 1 and stored in the update function.

Difficulty faced: -

- Making an MRMW array in which each element stores three values (here value, stamp, and threadID).
- For this, an atomic wrapper class is created.
- Operator overloading is done for ‘!=’ operator, which compares stamp and threadID.
- Copy assignment operator, also created, which is used for making a copy in collect and making result array in the scan.
- ‘-latomic’ tag should be used while comping the file.

Output log for obstruction-free implementation: -

- nW=4, nS=1, M=20, muW = muS = 0.5, k=5

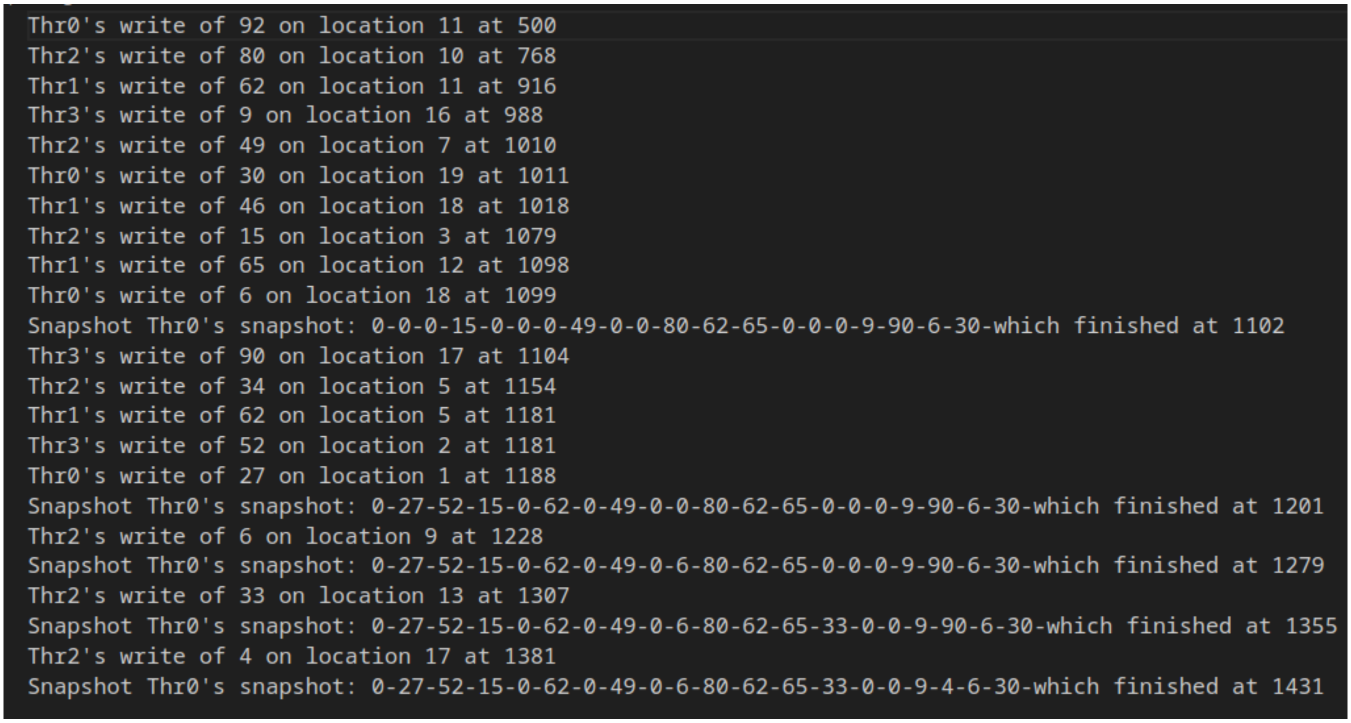


```
Thr0's write of 32 on location 6 at 262
Thr3's write of 83 on location 19 at 278
Thr1's write of 46 on location 3 at 325
Thr2's write of 50 on location 15 at 343
Snapshot Thr0's snapshot: 0-0-0-46-0-0-32-0-0-0-0-0-0-0-50-0-0-0-83-which finished at 653
Thr3's write of 7 on location 11 at 748
Thr1's write of 48 on location 0 at 753
Thr0's write of 82 on location 11 at 757
Thr2's write of 24 on location 15 at 789
Thr1's write of 50 on location 7 at 811
Snapshot Thr0's snapshot: 48-0-0-46-0-0-32-50-0-0-0-82-0-0-0-24-0-0-0-83-which finished at 814
Thr0's write of 55 on location 11 at 815
Thr2's write of 32 on location 2 at 852
Thr1's write of 67 on location 1 at 867
Thr0's write of 26 on location 18 at 871
Snapshot Thr0's snapshot: 48-67-32-46-0-0-32-50-0-0-0-55-0-0-0-24-0-0-26-83-which finished at 885
Thr2's write of 93 on location 6 at 908
Thr1's write of 75 on location 19 at 923
Thr0's write of 73 on location 12 at 927
Snapshot Thr0's snapshot: 48-67-32-46-0-0-93-50-0-0-0-55-73-0-0-24-0-0-26-75-which finished at 950
Thr1's write of 11 on location 18 at 980
Thr0's write of 83 on location 7 at 983
Snapshot Thr0's snapshot: 48-67-32-46-0-0-93-83-0-0-0-55-73-0-0-24-0-0-11-75-which finished at 1014
Thr1's write of 76 on location 14 at 1036
Thr0's write of 19 on location 19 at 1040
Thr1's write of 74 on location 5 at 1094
Thr0's write of 31 on location 15 at 1098
```

- For Thr0’s snapshot at 653, 4 values 46 (pos 3), 32 (pos 6), 50 (pos15), and 83 (pos 19) are there, which are the latest modified.
- Similarly, for all other snapshots, this holds true, hence, snapshots are consistent, i.e., linearizable.

Output Log for wait-free implementation: -

- nW=4, nS=1, M=20, muW = muS = 0.5, k=5



```
Thr0's write of 92 on location 11 at 500
Thr2's write of 80 on location 10 at 768
Thr1's write of 62 on location 11 at 916
Thr3's write of 9 on location 16 at 988
Thr2's write of 49 on location 7 at 1010
Thr0's write of 30 on location 19 at 1011
Thr1's write of 46 on location 18 at 1018
Thr2's write of 15 on location 3 at 1079
Thr1's write of 65 on location 12 at 1098
Thr0's write of 6 on location 18 at 1099
Snapshot Thr0's snapshot: 0-0-0-15-0-0-0-49-0-0-80-62-65-0-0-0-9-90-6-30-which finished at 1102
Thr3's write of 90 on location 17 at 1104
Thr2's write of 34 on location 5 at 1154
Thr1's write of 62 on location 5 at 1181
Thr3's write of 52 on location 2 at 1181
Thr0's write of 27 on location 1 at 1188
Snapshot Thr0's snapshot: 0-27-52-15-0-62-0-49-0-0-80-62-65-0-0-0-9-90-6-30-which finished at 1201
Thr2's write of 6 on location 9 at 1228
Snapshot Thr0's snapshot: 0-27-52-15-0-62-0-49-0-6-80-62-65-0-0-0-9-90-6-30-which finished at 1279
Thr2's write of 33 on location 13 at 1307
Snapshot Thr0's snapshot: 0-27-52-15-0-62-0-49-0-6-80-62-65-33-0-0-9-90-6-30-which finished at 1355
Thr2's write of 4 on location 17 at 1381
Snapshot Thr0's snapshot: 0-27-52-15-0-62-0-49-0-6-80-62-65-33-0-0-9-4-6-30-which finished at 1431
```

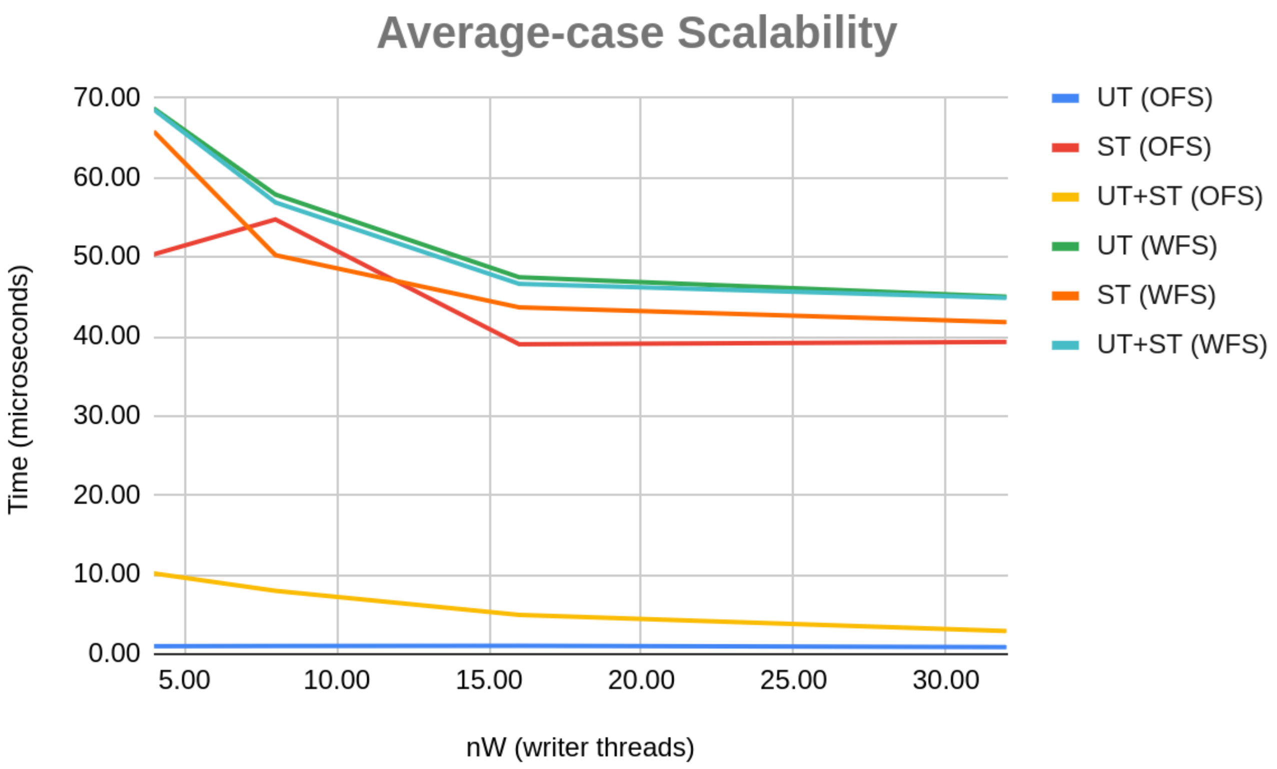
- After first Thr0’s snapshot at 1102, 5 writes occur.
- Second Thr0’s snapshot at 1201, incorporated the latest changes for all 5 writes, i.e., 27 (pos 1), 52 (pos 2), 62 (pos 5), and 90 at (pos 17).
- Similarly, for all snapshots, this holds. Hence, snapshots are consistent, i.e., linearizable.

The values generated by the writer thread are from 1 to 100. This range was chosen because the array length (M) is 40 or 20.

Also, there is a slight difference in the time format in the output file to what is given in the problem statement.
The time showing in the output file is from some reference start initialized globally and in microseconds.

Average-case Scalability

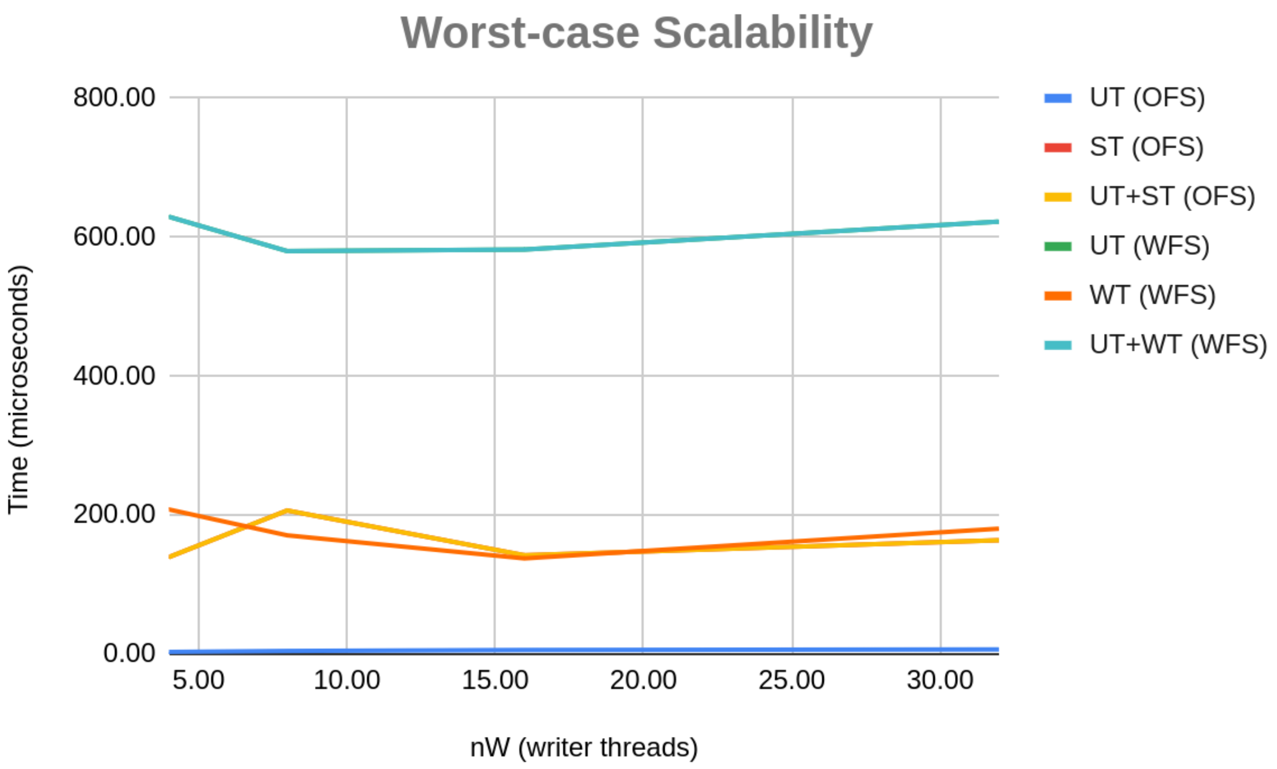
- nW/nS = 4, M = 40, muS = muW = 0.5, k = 5



- UT: - average update time, ST: - average scan time, OFS: - obstruction free snapshot, WFS:- wait-free snapshot.
- For OFS, the average update time is almost 1, as the update function only changes the value at a specific index in the array.
- For WFS, the average update time is comparable to the average snapshot time, because the update thread updates the value and also takes a snapshot.
- The average scan time is almost identical for both OFS and WFS implementations. But it differs from our intuition, we thought the WFS scan would take less time than the OFS scan. This can also happen because the array size (M) is small (40).
- As the number of updates is significantly greater than the number of snapshots, for both OFS and WFS, the total thread (update + scan) average time is highly influenced by the average time of update threads and is near to that.

Worst-case Scalability

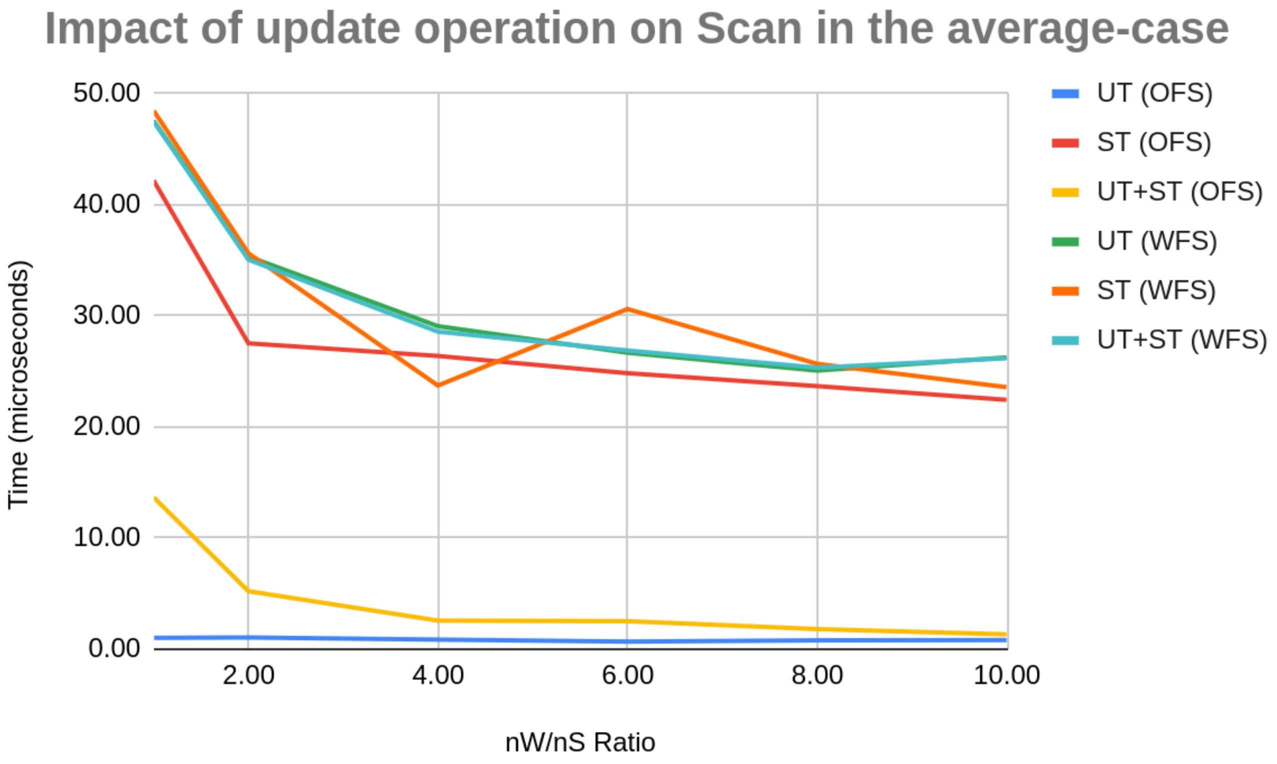
- nW/nS = 4, M = 40, muS = muW = 0.5, k = 5



- UT: - worst update time, ST: - worst scan time, OFS: - obstruction free snapshot, WFS:- wait-free snapshot
- For OFS, the worst update time is in the range [2,5], which increases from 2 to 5 as nW increases. The update function only changes the value at a specific index in the array, so it takes very little time.
- For WFS, the worst update time is the highest because the update thread updates the value and takes a snapshot.
- ST for OFS and WFS are almost identical. But it differs from our intuition, we thought the WFS scan would take less time than the OFS scan. This can also happen because the array size (M) is small (40).
- UT+WT for OFS coincides with the scan time as it takes the worst time.
- UT+WT for WFS coincides with the update time, as it takes the worst time.

Impact of update operation on Scan in the average-case

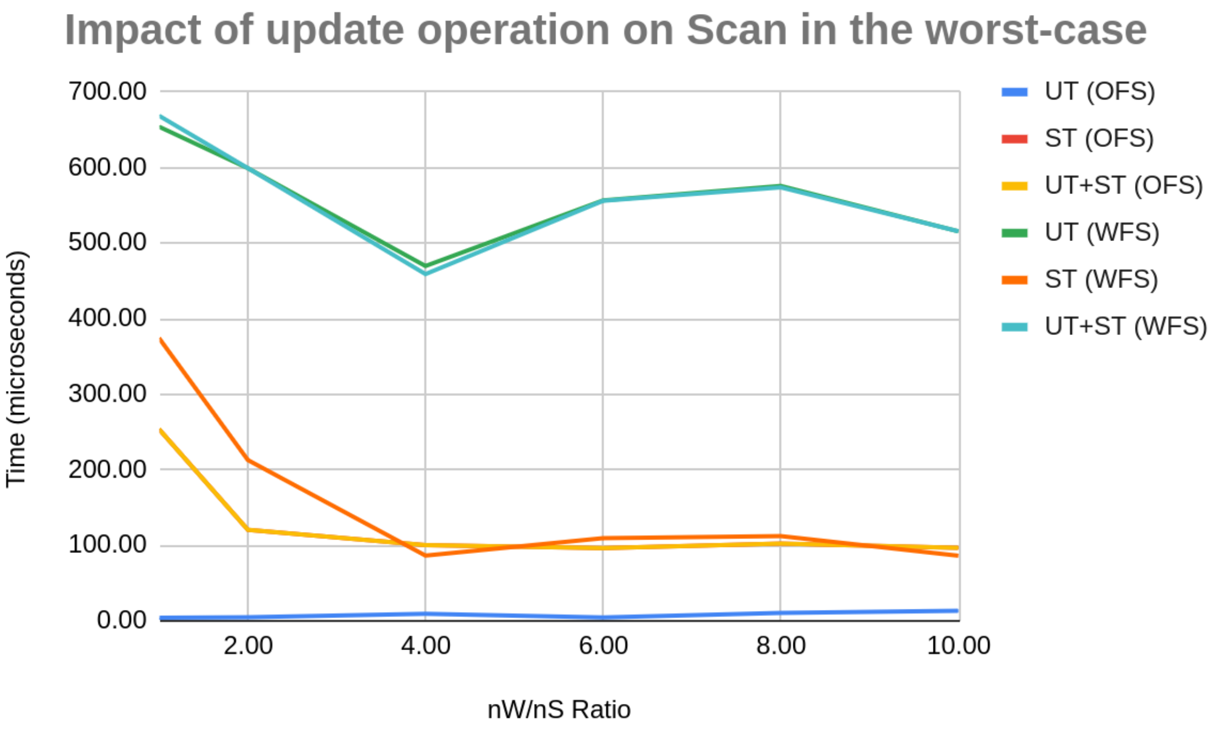
- nW/nS = 4, M = 40, muS = muW = 0.5, k = 5



- UT: - average update time, ST: - average scan time, OFS: - obstruction free snapshot, WFS:- wait-free snapshot
- For OFS, the average update time is almost 1, as the update function only changes the value at a specific index in the array.
- For WFS, the average update time is comparable to the average snapshot time because the update thread updates the value and takes a snapshot.
- ST for WFS is slightly larger than ST for OFS. But intuitively, as writes are increasing, OFS should take more time than WFS, but this is not true here. This can also happen because the array size (M) is small (40).
- As the nW/nS ratio increases, UT+ST (OFS) decreases and becomes comparable to UT (OFS) because the number of updates becomes significantly more significant than the number of scans, so the UT+ST average is highly influenced by UT average time.

Impact of update operation on Scan in the worst-case

- nW/nS = 4, M = 40, muS = muW = 0.5, k = 5



- UT: - worst update time, ST: - worst scan time, OFS: - obstruction free snapshot, WFS:- wait-free snapshot
- Worst update time for OFS increases from 3 to 12 microseconds as nW/nS ratio increase, but still low as compared to other times as the update function only changes the value at a specific index in the array.
- ST for WFS is less than UT for WFS because the update function also calls an additional scan apart from updating the value. The update is helping in scanning while taking snapshots, so the worst-case time of ST is less than UT.
- ST for OFS and WFS are almost identical. But it differs from our intuition, we thought the WFS scan would take less time than the OFS scan. This can also happen because the array size (M) is small (40).
- UT+WT for OFS coincides with the scan time as sacn takes the worst time.
- UT+WT for WFS coincides with the update time, as update takes the worst time.