Comparing Obstruction-Free and Wait-Free Snapshot Algorithms

Aaryan, CO21BTECH11001

Program Design:

1. **Important for implementation:** In both of the programs, it is required to have an array, named `a_table` , of atomic MRMW registers, which means that we need to create an array of `atomic<StampedValue<int>>` objects.
   I have used C++ atomics library to implement the same. According to the documentation, it is necessary that the object passed inside the `atomic` class (which is `StampedValue<int>` in this case) should have a size which is a multiple of 4 bytes.
   Therefore, even though we need only three variables (`value, stamp` and `pid`) in theory, we have to define an extra garbage variable for the implementation. I have named it '_' (underscore) in the program.
   Also, we can't use an "`int`" data type for the above four variables. Because in that case, the size of the `StampedValue<int>` object will become 16 bytes and the limit on the size (according to the documentation) is 8 bytes.
   Therefore, I have used a "`short int`" datatype in this case. So, the size of the `StampedValue<int>` object is 8 bytes.

   ```
   #define int short int // Line 23 in both of the files
   ```

2. **Changes in code for obstruction free implementation to make it work for multiple writers:** In the algorithm described in Fig 4.17 of the book, there is only a single writer. Therefore, there are no overlapping updates on any location of the array. However, in a multi-writer algorithm, it is possible to have overlapping updates on a single location.
   To address this problem, it is required to store the thread-id in the `StampedValue` class. Therefore, whenever a thread obtains a clean collect, it means that there is no update in the array between the two collects.
   For comparing the arrays `oldCopy` and `newCopy`, we have to compare all the three fields, namely `value`, `stamp` and `pid`, of every location in the

array. Therefore, if two threads, say `i` and `j`, are doing overlapping `update` at the same location, say `k`, and they have read the same value of `stamp` from the `oldValue` and their `newValue` have same value of `stamp`, their thread-ids will be different. So, if a thread performing `snapshot` operation has taken the first collect after thread `i` has performed the `update` and the second collect after thread `j` has performed the `update`, it will be able to detect that `oldCopy[k] != newCopy[k]`, because the `pid` field of these two values are different, even though the `stamp` field of these two values are the same.

3. **Storing the logs:** To store the log strings of the threads in order of their time of execution, an array of strings named `log_strings` is initialized to a large size (`SIZE`). To write its log, a thread has to access an atomic variable named `log_index` using the `fetch_add` method and write its log at that location in the array. Finally, all the logs are printed in a log file.

4. **Storing times:** To store the time taken for an operation, two 2D arrays named `updateOperationTime` and `snapshotOperationTime` are created. A thread performing an update operation will push the time taken in the array `updateOperationTime[thread_id]`. Similarly, a thread performing a snapshot operation will push the time taken in the array `snapshotOperationTime[thread_id]`.
   Note: Time taken is measured in nanoseconds in both of the programs.

5. **Random numbers:** To ensure that all the threads produce a different sequence of random numbers , the random number generator is seeded with a value equal to the product of the thread-id and the current time. This also ensures that different random numbers are generated in different executions of the program.

- **Average-Case Scalability:**

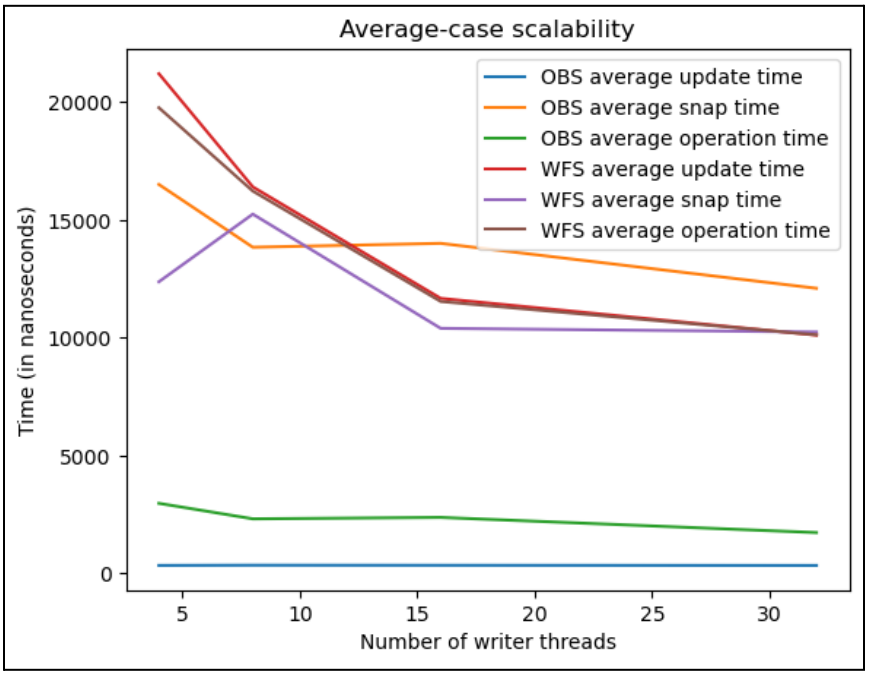

Figure 1: Average-Case Scalability

- **Worst-Case Scalability:**

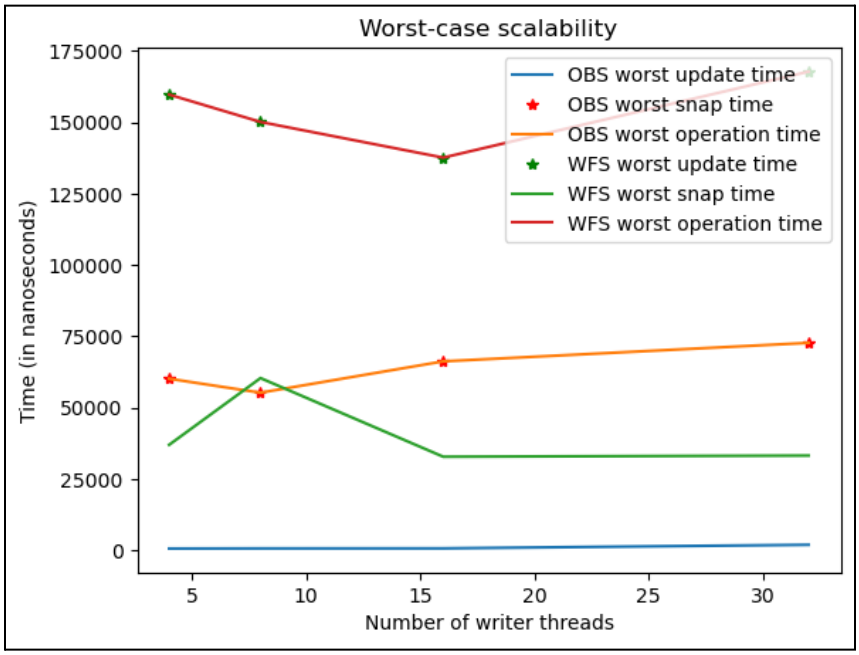

Figure 2: Worst-Case Scalability

- **Impact of Update operation on Scan in average-case:**

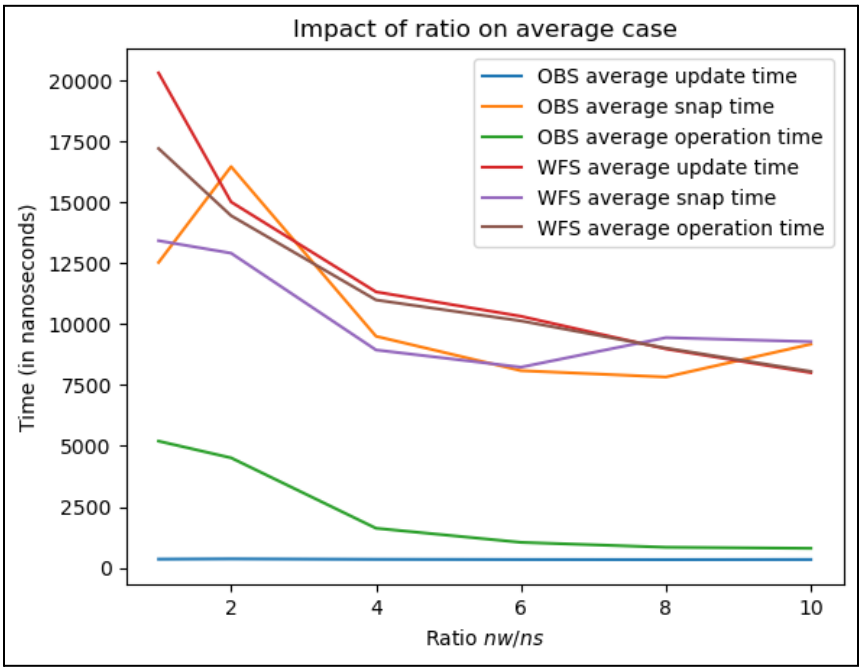

Figure 3: Impact of update operation on average-case

- **Impact of Update operation on Scan in worst-case:**

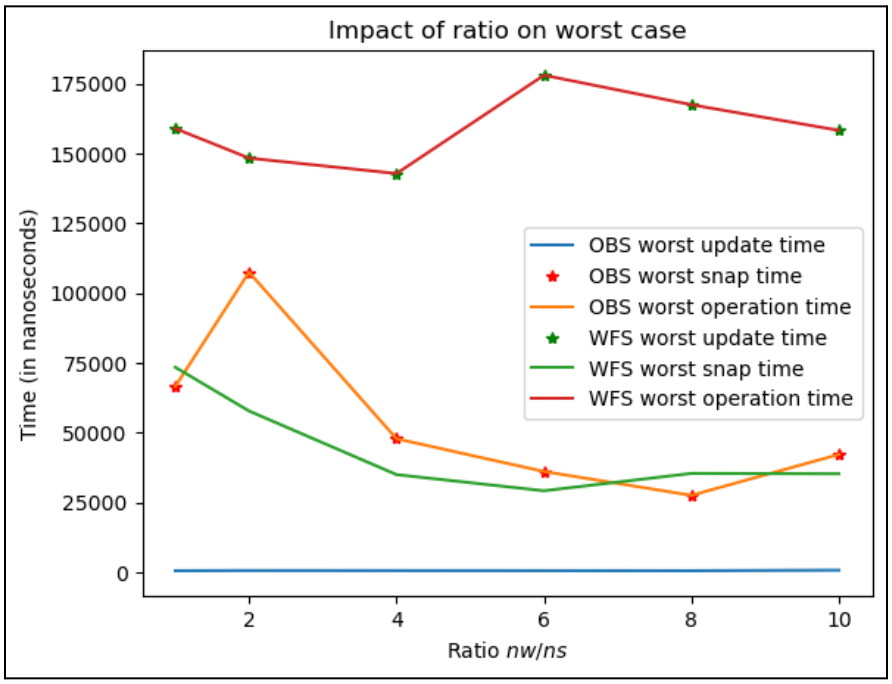

Figure 4: Impact of update operation on worst-case

- **Observations and Conclusion:**

1. In all of the figures above, we can see that the time taken (*average as well as worst case*) for a `snapshot` operation in the Wait-Free implementation is less than in the Obstruction-Free implementation in almost every case. This result is in accordance with the intuition that a Wait-Free snapshot algorithm is faster than an Obstruction-Free snapshot algorithm.

2. However, notice that the operation time (*average as well as worst case*) in the Obstruction-Free algorithm is less than the operation time in Wait-Free algorithm.
   It can be explained as follows:

   a. The `update` method in the Obstruction-Free algorithm is lightning fast as compared to the Wait-Free algorithm. The reason is that in the Wait-Free algorithm, the writer thread takes a snapshot along with updating the value. But that is not the case in the Obstruction-Free algorithm.

   b. The number of `update` operations are way more than the number of `snapshot` operations. This is because the ratio of the number of writer threads and the number of snapshot threads is always more than or equal to 1. Also, a snapshot thread performs a finite number of operations, whereas the writer threads keep performing write operations until all the snapshot threads terminate.

   c. Because of the above two reasons, we can say that the average operation time is affected more by the `update` operations than the `snapshot` operations. Therefore, the average operation time in Wait-Free algorithm is much higher than the Obstruction-Free algorithm, even though the `snapshot` operation is faster in Wait-Free algorithm.
      Similarly, the worst case operation time in Wait-Free algorithm is much higher than the Obstruction-Free algorithm, because of the time consuming `update` operation in Wait-Free algorithm.

3. In conclusion, the Wait-Free implementation performs a faster `snapshot` operation than the Obstruction-Free algorithm. However it performs a slower `update` operation.
   In my opinion this behavior of the Wait-Free algorithm is justified because an `update` operation is more “demanding” than a snapshot operation, in the sense that it demands a change in the existing value(s). Therefore, it is justified to give the `snapshot` operation more “preference” than the `update` operation and return the snapshot in less amount of time while bearing the cost of a time consuming `update` operation.