VU Advanced Multiprocessor Programming
SS 2023
Project 3: Snapshots

Alexandre Gunter 12202294, Serban Ionescu 12202437, Gabriele Cimador 12231849

June 2023

Contents

1 Introduction

In the context of this project, our team had to implement, test and benchmark two different snapshots providing a scan and an update operation. As a reminder, a snapshot is an object composed of n registers, each supporting read and write accesses through the scan and update operations respectively. Throughout the project, we have used the C language with the OpenMP library, and a Python script to generate plots during our benchmarks.

In the implementation section of this paper, we will detail the project structure and the assumptions on which we have relied to construct our snapshots and algorithms. In the testing section, we will explain how we have tested the correctness of our implementations. As the second implementation, the MRMW Partial Snapshot, was based on a research paper, we've also included extended tests in an attempt to further verify its linearizability. Finally, we will explain our bench-marking procedure and will show the results.

We would like to mention that our team is composed of three Erasmus students studying at the university for a semester. We have chosen this course as it was mandatory for us to take an introduction lecture to Multiprocessor Architecture. However, the content of this course goes way beyond the scope of an introduction course, and as we are currently only completing our Bachelor degrees, we would be grateful if our project could be graded accordingly. We thank you for your consideration and wish you a pleasant reading.

2 Implementation

2.1 Wait-Free MRSW Snapshot

The first implementation is based on the work of Maurice Herlihy and Nir Shavit in the book "The Art of Multiprocessor Programming" [1]. The corresponding code can be found in the `src` folder within the `WFSnapshot` directory.

Our goal was to develop a wait-free snapshot using atomic multi-readers single-writer (MRSW) registers. To achieve this, we implemented atomic MRSW registers that allow concurrent read and write accesses. These registers are composed of atomic single-reader single-writer (SRSW) registers, which store values as Time-Stamped values. For our implementation, we relied on the Java implementation described in the aforementioned book. However, we simplified our implementation by storing only integers in the registers instead of generic values. Furthermore, to streamline our code, we made the thread-local values in the SRSW register implementation register-local. This decision was based on the observation that, in our MRSW register implementation, each SRSW register is always read and written by the same thread. Finally, our SRSW registers store stamped-stamped values, as each atomic SRSW register stores a stamped value that is itself stamped by our atomic MRSW register.

The snapshot is constructed using these registers to ensure concurrency, and it employs an algorithm that relies on mutual thread assistance to achieve wait-freeness. The code for each component is stored in corresponding .h/.c files. For example, the implementation of the stamped value is found in the files `stampedValue.c` and `stampedValue.h`.

2.2 MRMW Partial Snapshot

The second implementation is based on an article written by Damien Imbs and Michel Raynal [2]. The corresponding code can be found in the `src` folder within the `PSnapshot` directory.

Our goal was to develop a snapshot composed of atomic multi-readers multi-writer (MRMW) registers that could allow partial scans of the shared memory. To achieve this, we implemented MRMW registers using the `std_atomic` library. Although the paper also used MRSW registers for some data structures needed to build the algorithm, for simplicity and efficiency purposes, we opted to use the MRMW registers for these data structures as well. Since the type of the stored values was not specified, we chose to use atomic integers to maintain consistency with our other snapshot implementation.

Additionally, the provided data structures included an array of active-set objects, which were implicitly defined by referring to other research papers. After reviewing these papers, we found the content to be quite advanced. To simplify the implementation, we decided to represent each active-set object as an array of booleans, where each element represents whether a thread is scanning a particular register. Through thorough testing, we found that this approach worked well for any number of threads, although we cannot guarantee its suitability for concurrent use.

This snapshot also utilizes a "Help when needed, but no more" and "Write first, help later, and help individually" algorithm. It relies on mutual thread assistance to guarantee wait-freeness and is implemented using multiple data structures defined in our `PSnapshot.h` file.

3 Testing

3.1 Base Tests

Both implementations offer four base tests that they have in common. These tests aim to ensure that the implementation is running smoothly and to verify its correctness (and linearizability). All tests can be found and executed from the `test` sub-folders. One test sub-folder is provided for each implementation, and the base tests can be run by using the command $./PSnapshotTest$ and $./WFTest$

for the Partial Snapshot and the Wait-Free MRSW Snapshot implementations respectively. To compile the base tests, run the command *make test* from the root folder containing the Makefile. However, we must warn you that our implementation is not entirely free of memory leaks, even though we have spent many hours trying to remove as many as possible. Executions can become memory-consuming, so it is strongly recommended to run all tests using *srun* to avoid segmentation faults.

The first test verifies that the snapshot supports sequential operations. Only one thread runs in this test. The snapshot array is initialized to 1, the thread scans and reads 1, the register is updated to 2, and the thread scans again and reads 2.

The second test aims to ensure that parallel writes update the registers in a wait-free concurrent manner. A first scan is performed, then all threads update their respective registers with their thread ID, and a second scan is done to verify that all updates have been successfully completed. This test is executed for n threads in parallel, $n \in \{2, 4, 68, 18, 32, 64\}$.

The third test aims to test that all write and read operations occur in a linearizable way. All threads except one update all their registers from 0 to 1000, while one thread scans twice during this period. Then, both snapshots are compared to verify that the values are monotonically increasing, which means that for each register, the value stored in the second scan is greater than the value stored in the first scan. Once again, this test is executed for n threads in parallel, $n \in \{2, 4, 68, 18, 32, 64\}$.

Finally, the last test verifies the behavior of our implementation during concurrent reads and writes. The idea of the test is the same as in test three, but now two threads are scanning instead of one, and we are checking that the scans for both threads respect the monotonic increase of the written values. This test is also executed with up to 64 parallel threads.

Overall, when put all together, these tests should help to demonstrate the correctness of linearizable execution of operations on both snapshots.

3.2 Extended Tests

As the Partial Snapshot implementation uses MRMW registers and supports partial scans, we needed to extend the base tests to verify correctness for execution of linearizable operations.

In the `TestPSnapshot.c` file, a fifth test was added to the base test. This test is a copy of test four, but the scanning threads are reading a random number of registers between 1 and the number of stored registers instead of all registers. Thus, we can check that a partial scan works the same way a classic memory scan does.

Furthermore, we have created a `TestLinerizability.c` file which contains additional linearizable execution cases and which tests if the output values are following an expected order according to the linearization points. In the first test, two threads are concurrently writing to the same register then reading from this register, and we want to ensure that these operations happen in atomic order. To be more precise, such an execution shouldn't occur : $update_1(r, 10) \rightarrow update_2(r, 11) \rightarrow scan_1(r, 11) \rightarrow scan_2(r, 10)$, where r is our register, and $update_1$ is an update by thread 1. By ensuring that $scan_1(r, 11)$ can never be followed nor preceded by $scan_2(r, 10)$ in the same execution, we can guarantee that such an execution order can not happen.

In the second test, we extend the above test to support concurrent writes to two registers, once again checking all possible execution orders. For instance, an execution like the following one should not occur: $update_1(r_0, 10) \rightarrow update_1(r_1, 10) \rightarrow update_2(r_0, 11) \rightarrow update_2(r_1, 11) \rightarrow scan_1((r_0, r_1), (10, 11)) \rightarrow scan_2((r_0, r_1), (10, 10))$.

Hence, our tests serve as benchmarks for correctness and show that the claims made by the given paper are true: the given algorithms do provide an efficient and linearizable concurrent partial snapshot implementation.

4 Benchmarking and Results

4.1 Benchmarking Design

In the context of our project, benchmarking was far from an easy task. The implementations were not optimal in terms of running speed, especially in the case of the Wait-Free MRSW Snapshot where no built-in atomic variables were used to build the registers. Additionally, the algorithms and the $n \times n$ matrix used for each MRSW register resulted in exponentially increasing memory usage and run time. Since the project instruction set a maximum benchmark execution time of 1 minute, and considering that our implementations were in separate folders to simplify our project structure, our objective was for each benchmark to run under 30 seconds while gathering a meaningful amount of data. Here's how we tried to achieve this.

First, for both snapshots, the benchmark consists of a `benchmark.py` Python script and a `snapshotBench.c` C file, as provided in the project template. The `snapshotBench.c` file is exactly the same for both implementations as we wanted to compare the results between them. This file generates six values: three latency metrics and three throughput metrics. It is divided into three sections, each resulting in one latency and one throughput metric.

The first section benchmarks the efficiency of the update operations by having a given number of threads n write in parallel to n registers. Throughput is measured by dividing the number of executed operations by the time between the first and last write. Latency is measured by storing the time taken by each thread to update and averaging the results over all executions. Therefore, if 32 threads are executed in parallel, latency will be measured 32 times and then averaged.

The second section follows the same idea, but this time it benchmarks the efficiency of the scan operation, also known as snapshot in the paper. n threads execute the scan operation concurrently and then return. Throughput and latency are measured in the same way as mentioned above. Finally, we added a "mixed" section, the third section, to evaluate the efficiency of the scan operation when concurrent updates are being executed and to compare it with the efficiency of concurrent scan operation without concurrent update. In this section, threads are divided into two equal groups: one group writes to the registers, and the other group reads. Throughput is calculated by measuring the time taken for all operations to finish, and latency is measured by averaging the results over all reads. For example, if 32 threads are executing in parallel, scan latency will be measured for 16 threads and then averaged.

To measure all these metrics, we had to limit the number of concurrent threads to 32 and 64 for the Wait-Free MRSW Snapshot and Partial Snapshot implementations, respectively. This was necessary because the latency of each operation was increasing exponentially with the number of threads, especially in the case of the Wait-Free Snapshot implementation described in the lecture. Furthermore, we executed the benchmarks only 3 times from the Python script, which made our results less precise for a small number of threads but still meaningful for larger numbers. Even by removing one of the sections mentioned above, we still struggled to run our benchmark more than 10 times while keeping the execution time below 30 seconds.

Nevertheless, we believe that our mix of operations is representative of typical use cases of a snapshot, combining individual updates and scans, and mixed updates and scans. We have also chosen these operations as they are usable for both implementations, though the benchmark for the Partial Snapshot also contains a commented section that can evaluate metrics for concurrent writes.

To compile and run our benchmark, use the *make small−bench* command inside of the *srun* command as follows: srun -p q_student -N 1 -c 64 make small-bench. A `data` sub-folder will then be created for each implementation, containing the described metrics.

4.2 Results Analysis

First of all, let's analyze the data obtained for the latency metrics. As expected, latency rises as the number of used thread grows. Furthermore, when using a large number of threads, the Partial Snapshot implementation is much more efficient than the Wait-Free MRSW Snapshot, which is also an expected result. One very interesting observation that can be drawn from our data is that, in the case of the Partial Snapshot algorithm, the average scan latency with concurrent updates scales better than the average scan latency without the concurrent updates. This could be an expected behavior, as the Partial Snapshot algorithm uses a helping technique that is supposed to guarantee wait-freeness, and in which a thread completing an update always helps, if needed, a blocked scanning thread to terminate by using his own scan. However, as shown by the plots provided below, this does not apply to our Wait-Free MRSW Snapshot implementation. We could hypothesize that it is probably because the writing process of our atomic MRSW registers is much more time consuming than the one of our atomic MRMW registers in the Partial Snapshot implementation.

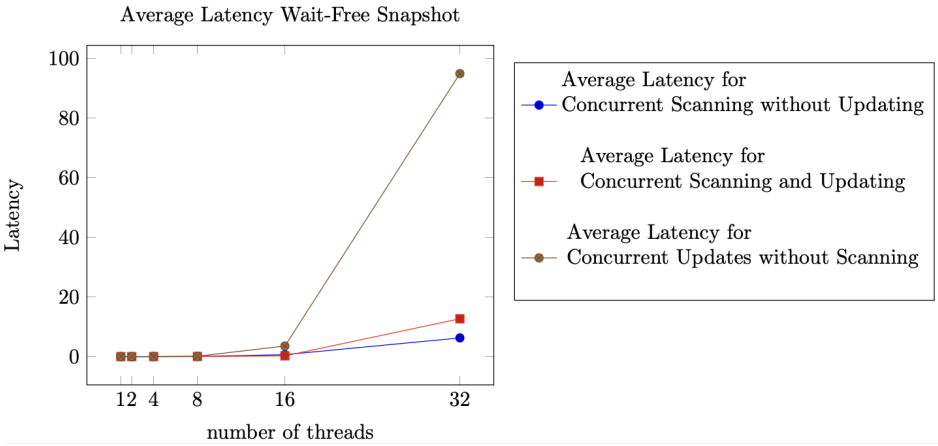

Figure 1: Average latency for the Wait-Free MRSW Snapshot operations in ms

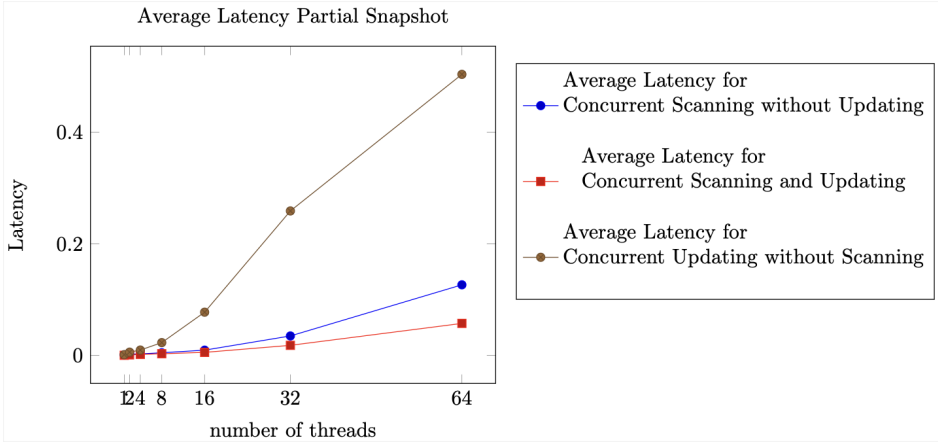

Figure 2: Average latency for the Partial Snapshot operations in ms

Another interesting observation is that the update operation alone always takes much more time to execute than the scanning operations. More precisely, it takes around 5 times more time for the Partial Snapshot update to execute than for a Partial Snapshot scan, and up to around 10 times more time for a Wait-Free MRSW Snapshot. In the context of the Wait-Free Snapshot, this could be explained

by the complex algorithm used to update the MRSW registers. However, such an argument does not hold for the Partial Snapshot implementation. Therefore, the explanation for such a difference might be memory related, even though we have not found any suitable explanation for this observation. Unfortunately, this might show that our data is to some extent unreliable.

For the throughput, the number of concurrent operations by slot of time is once again larger for the Partial Snapshot than for the Wait-Free Snapshot. The different throughput values also tend to decrease drastically and converge towards a similar value as the number of thread grows. However, the plots provided below are not very visually representative of our results as the throughput is still relatively large for all operations.

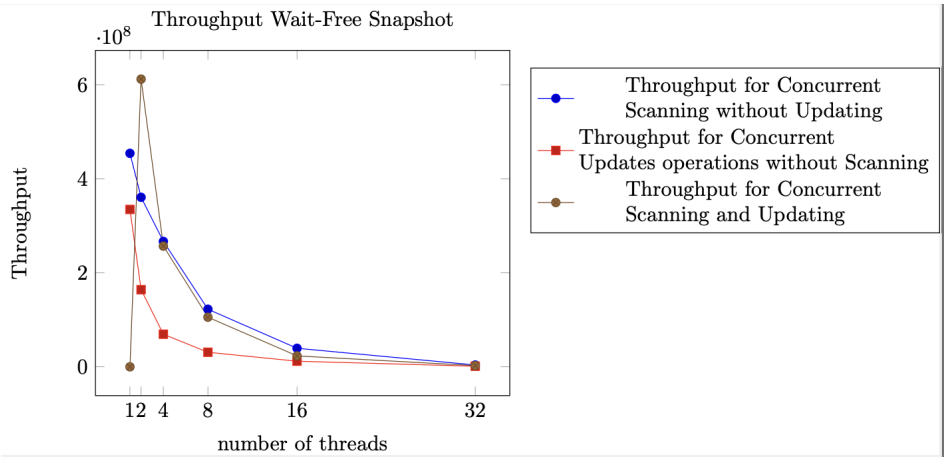

Figure 3: Average throughput for the Wait-Free MRSW Snapshot operations

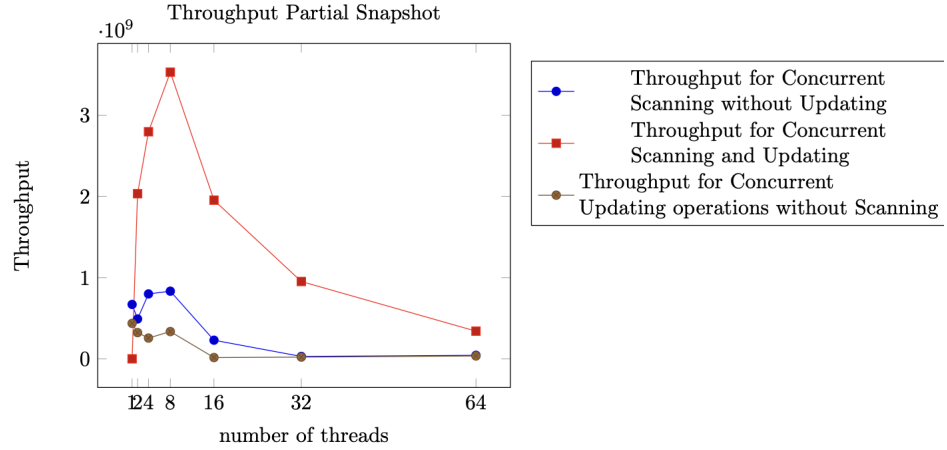

Figure 4: Average throughput for the Partial Snapshot operations

Finally, a last interesting observation is that the average throughput of concurrent reads and writes operations in the Partial Snapshot implementation is much higher than the other throughput values, which might be an indicator of the efficiency of its helping algorithm.

5 Conclusion

In conclusion, even though it is quite difficult to estimate how reliable our gathered data is, it is undeniable that both implementations provide interesting concurrent snapshots implementations. The Partial Snapshot implementation is a much more suitable choice when it comes to efficiency, but it could still use a few algorithmic optimizations that have been quickly described in the paper. Its partial snapshot functionality also accounts for an interesting improvement, though we haven't been able to benchmark its efficiency due to the time constraints. On the other hand, the Wait-Free MRSW Snapshot is definitely more of a practical example and is probably not an efficient implementation.

References

[1] M. Herlihy and N. Shavit, *The Art of Multiprocessor Programming.* Morgan Kaufmann Publishers, pages 71-93, 2008.

[2] D. Imbs and M. Rayna, *Help when needed, but no more: Efficient read/write partial snapshot.* J. Parallel Distrib. Comput., 2012.