ASAM OSI Flatbuffers

Performance Report

Clemens Linnhoff, Philipp Rosenberger

March 2022

ASAM OSI Flatbuffers

Performance Report

Abstract

In the current phase of the ASAM Open Simulation Interface (OSI) development, the OSI Change Control Board (CCB) is discussing a new release 4.0 that may disrupt backward compatibility. Such a step in a standard needs well-founded arguments, but on the other hand it also enables new possibilities. In order to evaluate the application and performance of Google Flatbuffers instead of the currently used Protobuf for serialization and deserialization of data, the OSI CCB has commissioned Persival GmbH.

This second report contains a detailed performance analysis of Flatbuffers. Application experiences for OSI during the implementation of working examples are given in the first report. The overall project goal is to support the decision of ASAM OSI to change the standard's serialization method in future releases. The project period has been from Jan 18 - Mar 31, 2022.

At first, the general performance measurement concept is introduced. This includes an explanation at what steps time stamps are captured during co-simulation and additionally the definition of the performance logfile format.

The second chapter includes the evaluation of all performance measurements and a detailed comparison. All tests are performed using OSI v3.4.0 which already includes Flatbuffers for experimental build. Performance is evaluated for transfer of different numbers of reflections and objects. The number of rays thereby reflects typical amounts of reflections for super-sampling all beams of lidars like the Velodyne VLP32^{TM}. An additional evaluation between debug and release builds and of the Flatbuffers Object API is performed. Furthermore, as it matters a lot for performance of Flatbuffers, the difference between using tables or structs for the data is evaluated. Last, it is evaluated what it takes to replay previously captured OSI trace files in binary format and all results are summarized in a table at the end.

Finally, a short conclusion on Flatbuffers performance is provided accompanied by recommendations for the future of the Open Simulation Interface.

Contents

1 Performance Logging

1.1 General Concept

To assess the performance of Flatbuffers vs. Protobuf, the general concept is to connect an extended OSMPDummySource FMU to an OSMPDummySensor with performance logging implemented in both FMUs. The OSMPDummySource was extended to generate different quantities of equal dummy osi3::LidarSensorView::Reflections in addition to the moving objects it already creates. The OSMPDummySensor was extended with a section that creates osi3::LidarDetections out of a number of reflections to consider data access in the sensor model. Furthermore, different concepts of using Flatbuffers are investigated, like using the Object API or changing certain Tables to Structs.

The data from the performance log files is processed by a Python script to generate box plots. The script along with a more detailed README on how to use it is available open source[1].

1.2 Time Measurements

In the OSMPDummySource time stamps are recorded at three locations in the code. The first one is at the beginning of the *doCalc()* function. In the Protobuf version, the second time stamp is taken right before the *set_fmi_sensor_view_out()* function is called. That is where the serialization in Protobuf is done. In the Flatbuffers version, the second time stamp is taken before the builder is finished, as there is no single point in the code, where the serialization happens. The third time stamp is recorded in both implementations, after the *set_fmi_sensor_view_out()* function is called. The differences between these three time stamps result in two time measurements:

- **Source generation**: The time it takes from the beginning of *doCalc()* until the data is serialized in Protobuf. In Flatbuffers the serialization happens during the object generation, so it is not separable. This needs to be kept in mind when comparing the two.
- **Source serialization**: The time it takes to serialize the data in Protobuf and the time it takes in Flatbuffers to finish the builder and set the FMU output.

In the OSMPDummySensor time stamps are recorded at four locations. The first one is again at the beginning of the *doCalc()* function. The second time stamp is taken after the *get_fmi_sensor_view_in()* function is called. This is where the input is deserialized in Protobuf. The third time stamp is recorded before the *set_fmi_sensor_data_out()* function is called in Protobuf and before the builder is finished in Flatbuffers. The last time stamp is taken at the end of the *doCalc()* function after *set_fmi_sensor_data_out()* is called. The differences between these four time stamps result in three time measurements:

- **Model deserialization**: The time it takes for Protobuf to deserialize the input and for Flatbuffers to get the pointer to the input.
- **Model calculation**: The time it takes for the model to calculate its detection and object output.
- **Model serialization**: The time it takes to serialize the data in Protobuf and the time it takes in Flatbuffers to finish the builder and set the FMU output.

[1] https://gitlab.com/persival-open-source/open-simulation-interface/osi-performance-evaluation

1.3 Logfile Format

The time stamps are written along with further meta data, like the message sizes and which FMU is used to a JSON file for every FMU. The JSON format was adapted from a logging format developed in the SET Level project. An example of the format is given the performance log file excerpt in Code Snippet 1.

```
{
  "Header": {
    "OsiMessages": ["osi3::SensorView", "osi3::SensorData"],
    "EventFields": ["EventId", "GlobalTime", "SimulationTime", "MessageId",
          "SizeValueReference", "MessageSize"],
    "EventTypes": ["StartSourceCalc", "StartOSISerialize",
          "StopOSISerialize", "StartOSIDeserialize", "StopOSIDeserialize"],
    "FormatVersion": {
      "Major": 1,
      "Minor": 0,
      "Patch": 0,
      "PreRelease": "beta"
    }
  },
  "Data": [
    {
      "Instance": {
        "ModelIdentity": "OSMPDummySource Flatbuf"
      },
      "OsiEvents": [
        [0, 1648482698.771, 0.1, 0, 5, 23226752],
        [1, 1648482698.817, 0.1, 0, 5, 23226752],
        [2, 1648482698.83, 0.1, 0, 5, 23226752],
        [0, 1648482698.841, 0.2, 0, 5, 23226752],
        [1, 1648482698.882, 0.2, 0, 5, 23226752],
        [2, 1648482698.891, 0.2, 0, 5, 23226752],
        ...
      ]
    }
  ]
}
```
Code Snippet 1: Except from an OSI performance log file. The JSON format was developed in the SET Level project.

2 Performance Comparison

2.1 Different Reflection and Object Counts

In this first section of the performance evaluation several different configurations of lidar reflections and moving objects are assessed. The first case is the simulation of a typical reflection count for a Velodyne VLP32™ lidar. In the current ASAM OSI release, rays are defined in an equidistant pattern. This is not feasible for lidars with gaps between the layers, as those gaps would also be covered with rays leading to a not processible number of reflections. It is presumed that this will be addressed in a future OSI release, as proposed by the SET Level project.

Therefore, an improved reflection version is assumed for the definition of the number of reflections, where rays are only shot inside the boundaries of the lidar beams. The VLP32 ™ has 32 layers vertically and 1800 beams per layer over a 360° revolution. Super-sampling of the beams with 3 rays vertically and 6 rays horizontally per beam is defined, resulting in 18 rays per beam. With an assumption, that about 80 % of the rays produce a result, a total of 829,440 lidar reflections is generated. In addition to the reflections, the default amount of 10 moving objects from the OSMPDummySource is also created. In the OSMPDummySensor a section was added to generate a lidar detection out of every 18th reflection. The following sections on Debug, Object API and Structs also use this configuration and are therefore comparable.

Figure 1 shows the results of the performance measurement. The time it takes to generate and serialize the source data increases in Flatbuffers by around 60 %. But because there is virtually no deserialization in the model, the time in the model is drastically decreased by about 86 %. The total performance improvement of Flatbuffers over Protobuf in this configuration is 44 % from 93 to 52 ms.

The size of the SensorView generated and serialized by the source is about 16 MB with Protobuf and 23 MB with Flatbuffers. So, there is an increase by 44 % in size from Protobuf to Flatbuffers.

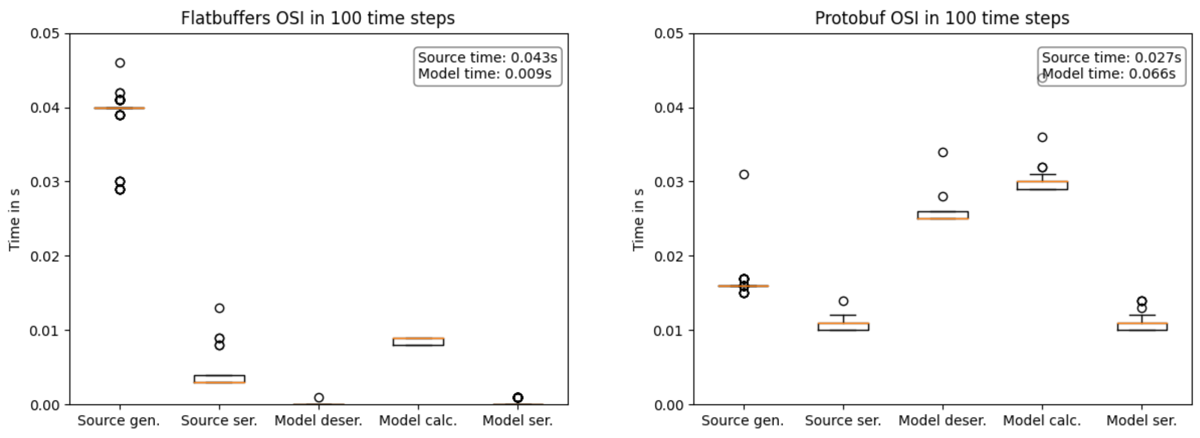

Figure 1: 829,440 lidar reflections and 10 moving objects

In the next evaluation, the reflection count was reduced by about half to 400,000 reflections. The results are shown in Figure 2. The performance scales roughly linearly with the number of reflections totaling in 24 ms in Flatbuffers and 45 ms in Protobuf. The size increases from 8 MB to 11 MB from Protobuf to Flatbuffers.

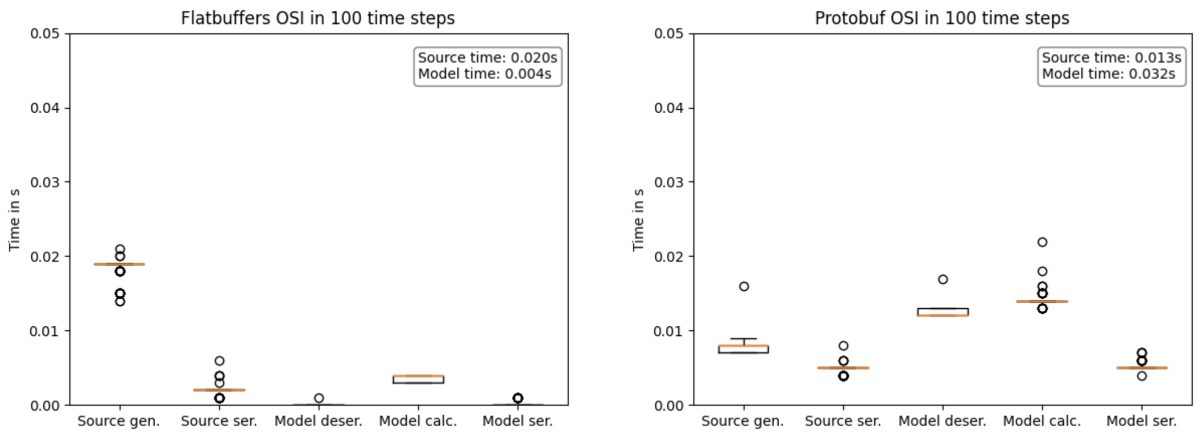

Figure 2: 400,000 lidar reflections and 10 moving objects

Further reducing the reflection count to 100,000 reflections confirms the linear scaling, as can be seen in Figure 3.

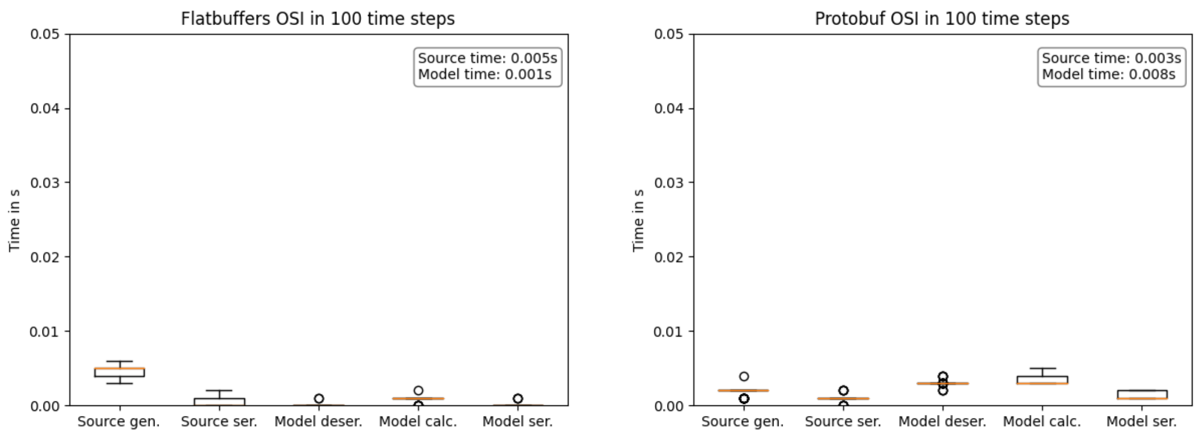

Figure 3: 100,000 lidar reflections and 10 moving objects

Removing the reflections altogether and just creating the remaining 10 moving objects yields a run time, that is not quantifiable in ms. For the sake of completeness, the plot is included anyways in Figure 4. The SensorView size for Protobuf and Flatbuffes is 2.2 kB and 2.4 kB respectively.

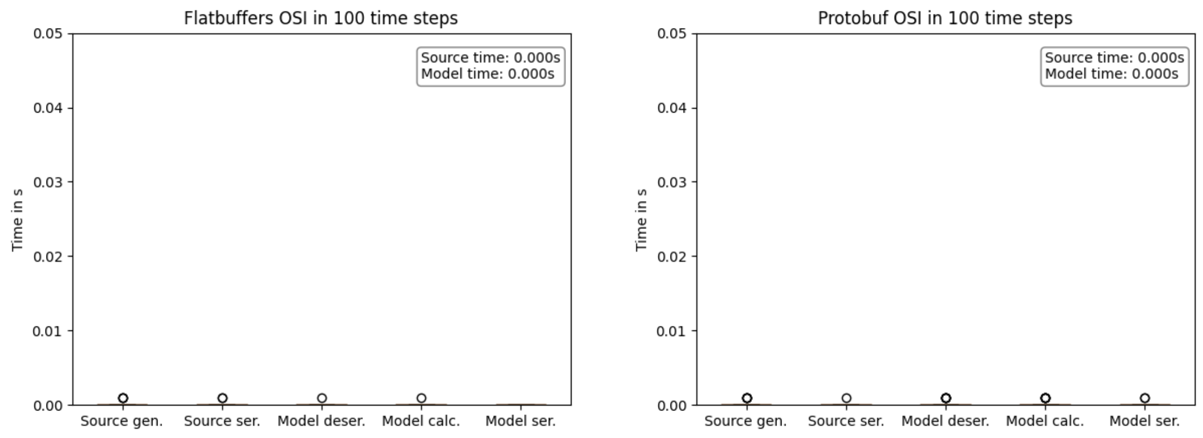

Figure 4: 10 moving objects

Increasing the number of moving objects, also increases the run time, but no difference between Flatbuffers and Protobuf is visible, as depicted in Figure 5 and Figure 6. The SensorView sizes are 41 kB for both Protobuf and Flatbuffers in the 200 object configuration and about 103 kB for 500 objects.

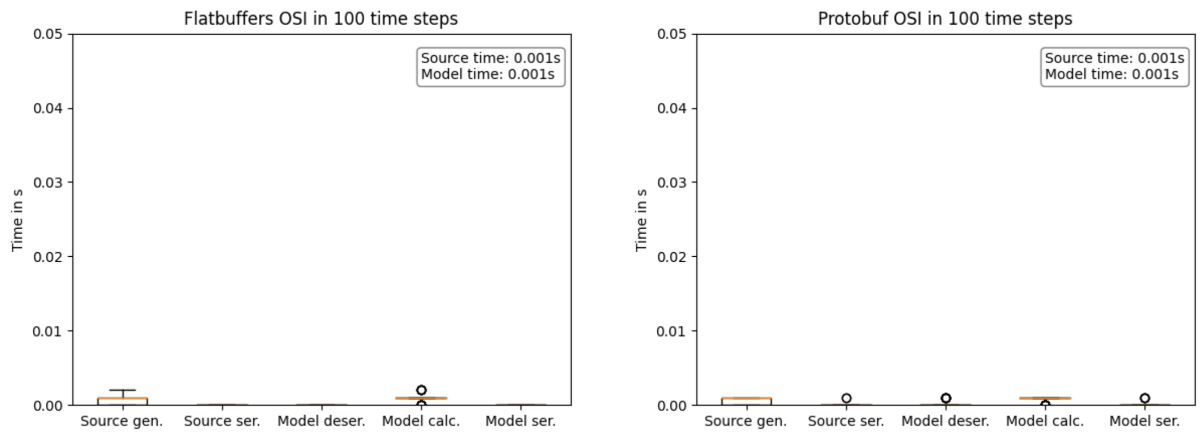

Figure 5: 200 moving objects

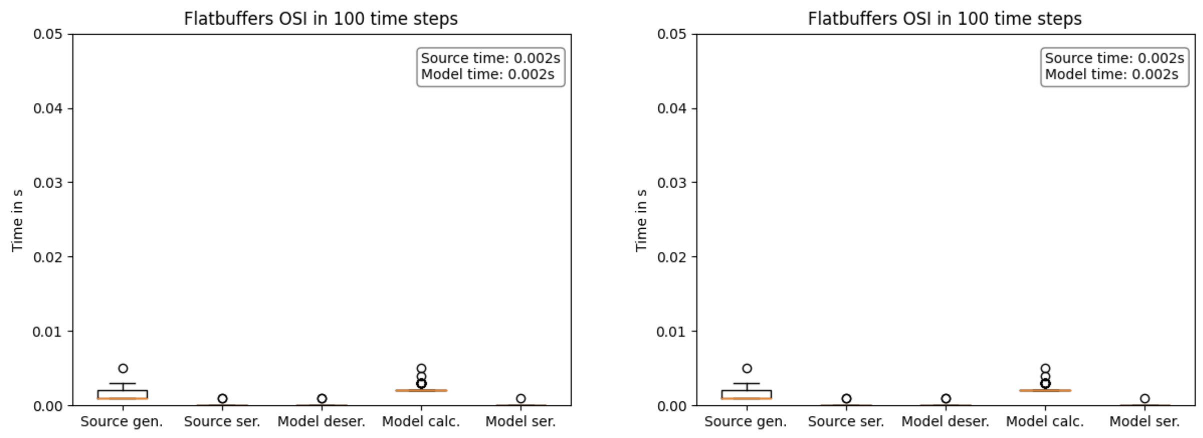

Figure 6: 500 moving objects

2.2 Release vs. Debug

The build configuration plays a major role. There is a big performance difference when building in Debug vs. in Release. All other performance evaluations in this report are done in the Release configuration. Figure 7 shows the result of Debug FMUs with the first configuration of 829,440 reflections. Please consider the changed scaling on the ordinate ranging from 0.0 to 0.5 s now. Run time is increased by around 880 % with Flatbuffers and 420 % in Protobuf. So, with Debug, Flatbuffers is actually 17 % slower than Protobuf. The sizes of the SensorViews are not affected and stay at 16 MB and 23 MB respectively.

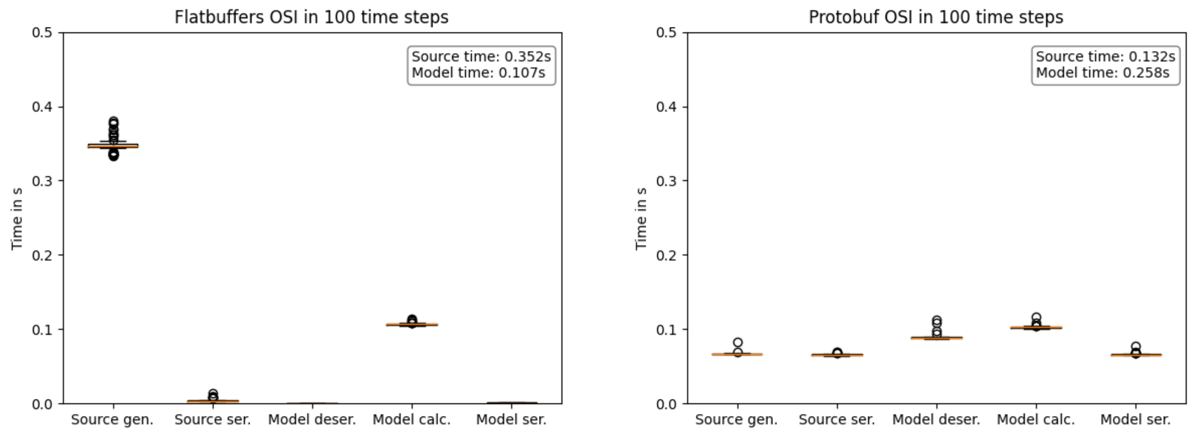

Figure 7: Flatbuffers and Protobuf built in Debug

2.3 Object API

As described in the application report, Flatbuffers also comes with an Object API, making the programming a more intuitive. This comes at the price of performance. Using the Object API over plain Flatbuffers increases the run time by around 48 %. The size of the SensorView does not change and remains at 23 MB.

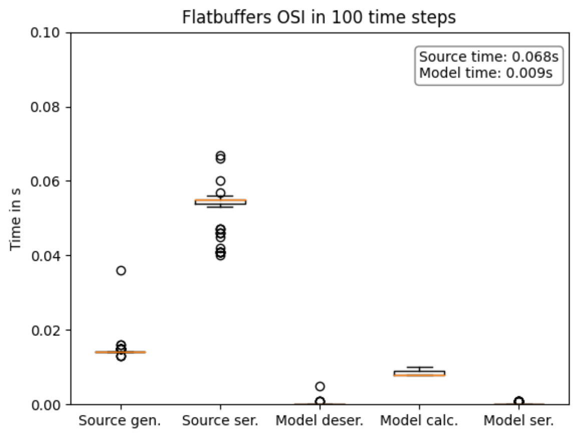

Figure 8: Flatbuffers Object API

2.4 Tables vs. Structs

Although using structs instead of tables for the reflections is not advisable, because of backward compatibility issues, the performance is still analyzed. For this evaluation, the reflection table as well as the more primitive fields like Vector3D, Dimension3D, Identifier etc. are changed to structs. The results in Figure 9 show an improvement both in the source and in the model. The source generation time decreases from 40 ms to around 20 ms and the model calculation from 9 ms to around 5 ms. The overall run time is reduced by 63 %. The size also slightly decreases from 23 MB to 20 MB. However, this remains a purely academic result, as backward compatibility is not possible with structs instead of tables.

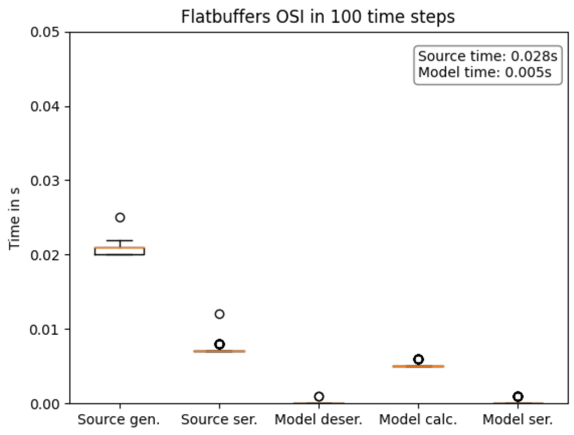

Figure 9: Lidar reflections and Vector3D, Dimension3D etc. as structs

2.5 Trace Files

Originally not intended for the performance evaluation is the run time evaluation for playing back trace files. Nevertheless, as the trace files including their players were developed during this project for the application assessment, they were also analyzed in terms of performance. The number of reflections however is not comparable to the previous examples, because an IbeoLUX ™ 2010 was simulated with the original ASAM OSI 3.4 reflection configuration instead of a Velodyne VLP32 ™ with its assumed SET Level improvements. Exactly 668,052 reflections are transmitted for every simulation time step along with 10 moving objects.

Figure 10 shows the results with a drastic performance improvement with Flatbuffers of 86 %. Flatbuffers increases the playback speed by around 7 times. The Protobuf trace file has a size of 968 MB containing 6 s of simulation time. The Flatbuffers trace file is slightly larger with 1.05 GB.

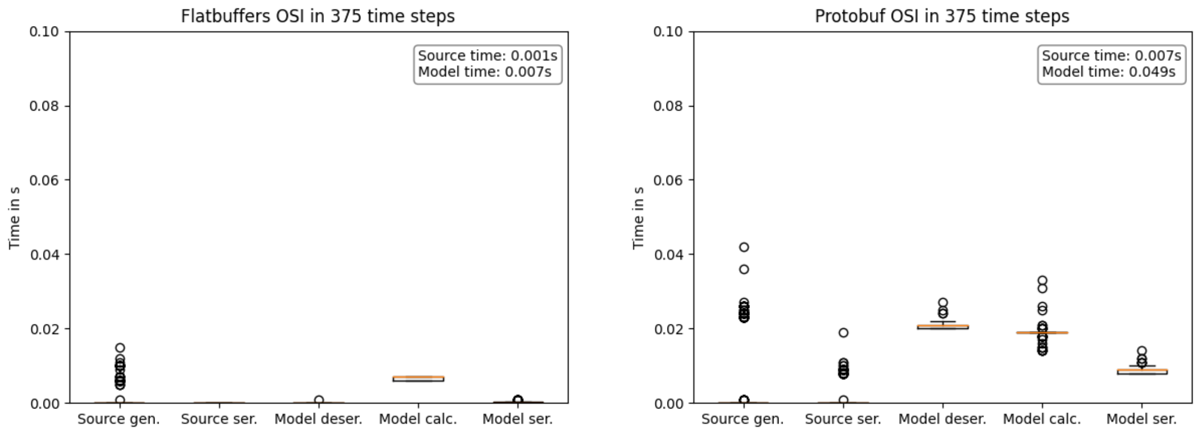

Figure 10: Trace file player with 668,052 lidar reflections

2.6 Summary

The following Table 1 summarizes the performance benchmarks of the evaluated tests. When large numbers of reflections are involved, Flatbuffers reduces the run time by around half compared to Protobuf. For configurations without reflections no performance gain was observed. Building the FMUs in Debug heavily increases run time as well as the Object API. Playing OSI trace files is around 7 times faster with Flatbuffers than with Protobuf.

Table 1: Summary of all performance evaluations

Evaluation Subject	**Flatbuffers**		**Protobuf**	
	Source in ms	Model in ms	Source in ms	Model in ms
829,440 reflections, 10 objects	43	9	27	66
400,000 reflections, 10 objects	20	4	13	32
100,000 reflections, 10 objects	5	1	3	8
10 objects	0	0	0	0
200 objects	1	1	1	1
500 objects	2	2	2	2
Debug build	352	117	132	258
Flatbuffers Object API	68	9	-	-
Reflections, Vector3D etc. as Structs	28	5	-	-
Trace File Player (700,000 refl.)	1	7	7	49

3 Conclusion

At first, the general performance measurement concept has been introduced including an explanation of the selection of points for time capturing during co-simulation. Therefore, five intervals are taken into account: The time for generating the source data, the time it takes to serialize the source data, the duration of the deserialization within the sensor model, the calculation time of the model and finally the data serialization time of the model.

The results within the second chapter show that computation time scales with the number of reflections and objects. As Flatbuffers does not need a deserialization step, data access is 7-8 times faster compared to Protobuf. Nevertheless, serialization of many reflections even takes around 60 % longer with Flatbuffers. Overall, the co-simulation of Reflection/ObjectSource and SensorModel with Flatbuffers takes half the time of the Protobuf implementation. This is a performance enhancement, but probably not as much as it was expected to be.

The Flatbuffers Object API while providing ease of use also comes with a lower performance by around 50 %. Structs can be used to increase performance, but are not backward compatible. The evaluation of the binary OSI trace files indicates that replay of binaries is 7 times faster with Flatbuffers, while again deserialization of Protobuf takes some time.

While there is definitely more application experience needed by other users of the OSI standard, the here evaluated lidar/radar reflections can be expected to be the worst-case regarding packet size per frame to transfer. The results show that object lists of moving objects are already quite fast with Protobuf and do not benefit from switching to Flatbuffers. However, the results for reflection transfer show that while serialization of the data cannot be accelerated with Flatbuffers, it does not need a deserialization step and therefore data access within the model after transfer is much faster.

After all application and performance evaluation of Flatbuffers against Protobuf, we as the service provider for this first trial would not recommend to switch to Flatbuffers. Possibly other data encoding for lidar/radar reflections/detections that e.g. uses some regular binary scheme in the first place, as it is already the case for camera data, seems to be more promising regarding data amount and serialization/deserialization time for now. Such data encoding would bring the possibility of applying data reduction and does not need a deserialization step, as is the case with Flatbuffers. If lidar/radar binary encoding would be added to OSI as just another field, this would keep backward compatibility and would not bring the overhead of the Flatbuffers application that is “definitely not the simple implementation solution to all performance problems that it was promised to be”, as already summarized at the end of the application report.