ScOSA: application development for a high-performance space qualified onboard computing platform

Kurt Schwenk, Moritz Ulmer, Ting Peng
ScOSA - Application Development for a High-Performance Space Qualified Onboard Computing Platform

Kurt Schwenk\textsuperscript{a}, Moritz Ulmer\textsuperscript{b}, and Ting Peng\textsuperscript{c}

\textsuperscript{a}German Aerospace Center (DLR), German Space Operations Center (GSOC), Münchner Str. 20, 82234 Wessling, Germany
\textsuperscript{b,c}German Aerospace Center (DLR), Simulation and Software Technology, Lilienthalplatz 7, 38108 Braunschweig, Germany

ABSTRACT

Future onboard computing applications require significantly greater computing performance than is currently provided by standard space qualified on-board computers. Examples of such applications include onboard data analysis and (rendezvous) navigation tasks. Therefore, the German Aerospace Center is currently developing Scalable On-board Computing for Space Avionics (ScOSA). The aim of the ScOSA onboard computing platform is to deliver high performance, reliability, scalability and cost-efficiency.

To reach these properties, a distributed computing platform approach is used, by which reliable radiation hardened computing nodes (LEON3’s) are combined with several high performance computing nodes (Xilinx Zynq), connected over a high-bandwidth SpaceWire network. The execution platform consists of a distributed task-based framework.

In this paper, the architecture, features and capabilities of the ScOSA onboard computing platform are presented from an application developer’s view. A brief summary of the design goals and the general hardware and software architecture of the ScOSA system will be introduced. This is followed by a description of the programming model and the application interface, with a focus on how the distributed nature of the ScOSA system is handled. It will also be shown how an existing application can be integrated in the ScOSA system.

The main part of this paper will focus on the computing performance attainable from the ScOSA platform. There will be a comparison of the computing performance of an example application executed on the ScOSA system versus a standard PC. It will also be demonstrated how the performance of an application can be improved by adapting it to the distributed computing architecture of the ScOSA platform. Furthermore, a short overview of the failure detection and recovery features of the ScOSA platform are described and how they can be integrated into an application.

1. INTRODUCTION

1.1 Background

For many future space missions, the available on-board computing performance is a crucial factor. It’s a requirement for many visionary application, such as advanced onboard image analysis, autonomous landing, robotic control and rendezvous navigation-systems.\textsuperscript{1}

However, due to the harsh conditions imposed by operating in the space environment, it is difficult to provide high computing performance on a spacecraft. There are restrictions in the system size, power consumption, and the heat emission of a spacecraft computing system. In addition, the computing system has to be able to withstand heavy physical stresses during the launch to orbit and has to be tolerant to the heavy radiation in a space environment. Also, due to the very restricted supportability of the system, it has to be designed to operate...
reliably until the end of the mission, despite the environmental constraints in space such as temperature and radiation drain.

As a result, current classical onboard computing systems are designed with reliability and robustness as the primary drivers. Since the market of space computer systems is small and developing in addition to certifying space hardware is expensive, the systems used today are several generations behind the current technological level used on the ground and still relatively expensive. For comparison, even the newest available space processors such as the GR740 Quad-Core LEON4 SPARC V8 Processor are no match for a modern smartphone processor like Qualcomm’s Snapdragon 810 Processor or Apples A11 Bionic.

1.2 Motivation
The main problem lies within the limited computing performance provided by current onboard computers. This directly impacts the scope of applications that can be realized on the given hardware. Therefore, the The German Aerospace Center (DLR) decided to develop a high-performance onboard computing platform. The main design goals of the system are to provide an adequate platform for future space applications.

First of all, it aims to provide significantly higher computing performance than classical approaches while considering mission requirements which consist of power consumption, reliability, mass, and dimension. This will directly impact the choice and effectiveness with which future space missions can be executed. This is especially relevant for exploration and earth observation missions where autonomy and large data quantities come into play, respectively. Secondly, in order to achieve high reliability, the computing units have to be able to reconfigure themselves when a unit fails. This reconfiguration process is built to respect the timing constraints imposed by real-time applications. Another important aspect of the system is the ability to reconfigure the system for different mission phases. This means that for the first stage of an exploration mission, the computing resources can be allocated for Attitude and Orbit Control System (AOCS) tasks. After the probe has landed, these resources can be reallocated for science tasks. And finally, as added by the authors, to provide a simple and standard Application Programming Interface (API), so that people who are not onboard computing experts can develop onboard applications without knowing system internals, while still being able to fully exploit the safety and performance features of the system. This goes hand in hand with being able to port existing applications to the onboard system without needing to rewrite them from scratch.

1.3 Knowledge Gaps
The question is how such an onboard computing system should look like. How can the opposing requirements for high performance, environmental robustness, high reliability, developer friendliness be combined while staying cost-efficient. The team developing the onboard computer has come to the conclusion that this is not possible with the standard approach of using only space qualified radiation-hardened components that are significantly slower than commercial components. Reasons for this include the large space and computing overhead incurred by adhering to conventional safety concepts. This results in redundant counterparts for critical components, which are not utilized the majority of the time. Another aspect of this is that a private computing system is utilized for each subsystem instead of sharing the limited resources on spacecraft. This is only exacerbated by providing a specialized and in most cases very basic non-standard APIs. The consequence of these limitations for application developers is that their applications often have to be reimplemented by an expert of the onboard computing system and cannot be reused in the subsequent missions.

1.4 Objectives
The goal of this paper is to provide an overview of the system from an application developer’s view. This will be conveyed by examining an application developed for the system as well as describing the system’s architecture. This will highlight the system’s most important features and demonstrate its capabilities. In addition, the programming model will be explained and a brief look at the API will be given. As a final point, the failure recovery features of the system will be described. This is a crucial aspect due to the use of Commercial off-the-shelf (COTS) components.

*when comparing the technical specification.
2. SYSTEM ARCHITECTURE

To understand the application, first, an overview over the system architecture is presented. In order to design a powerful, reliable and cost-efficient onboard computer, it was decided to build a distributed computing system, consisting of several COTS High Performance Nodes (HPNs) and space qualified Reliable Computing Nodes (RCNs). The computing nodes are independent computing systems consisting of a CPU, RAM, non-volatile memory and network interfaces. The computing nodes are connected over Ethernet and SpaceWire network connections. A setup consisting of six HPNs and two RCNs is presented in figure 1. The high computing performance is delivered by the high-performance COTS nodes. They deliver an order of magnitude higher computing performance than their space qualified counterparts at quite significantly lower price point. These nodes are expected to be able to work under space conditions with a high degree of reliability. However, due to the fact that these parts are meant to be used on Earth, they are not specifically hardened for space conditions. Therefore, they are observed by other nodes, and in case of a node failure, the computing tasks of this node are transferred to other nodes. The failing of high performance nodes is thought to happen due to radiation events, but in most cases the effects should only be temporary and the affected computing node should be able to recover. Once it has been power cycled and its self-tests have passed, it should be reintegrated into the distributed computing system. For very critical tasks, the RCNs are used. Interface Nodes (IFNs) are a special type of RCN that are used to interface with external components such as sensors and actuators. An IFN offers interfaces to connect with external devices. As external components can often be connected only to one IFN, it is not acceptable for IFNs to fail. Therefore, for this type of node, space qualified components are chosen. With the approach, it is possible to harness the high performance of the COTS world and the safety of space qualified components.

2.1 Hardware Architecture

To verify the design and provide demonstrations, various prototype systems were built at the DLR. The demonstration system consists of three HPNs, an RCN and two camera nodes while this application will focus on the HPNs. The setup is shown in figure 2. For the HPN, commercial Xilinx Zynq-Boards were chosen. The Xilinx Zynq Z7020 System-on-a-Chip is a hybrid system consisting of a Field Programmable Gate Array (FPGA) with a built-in dual-core ARM Cortex processor\(^1\). On the current HPN, 1 GiB of RAM and 4 GiB of flash memory are available. For more technical specification please refer to the vendor’s product website.\(^2\) The Field Programmable Gate Array (FPGA) should be used as an accelerator for compute-intensive algorithms. Utilizing all three nodes, the CPU and the FPGA, up to 516 MFlops/s of Double Precision performance\(^3\) should be attainable. For the network connection, a SpaceWire link should be used with a bandwidth of 150Mbit/s. For

---

\(^1\)Dual-core ARM Cortex-A9 MPCore @866 MHz
\(^2\)IEEE754
Figure 2: The setup consists of the RCN on the left, three HPNs in the middle and two camera nodes on the right. The cameras and HPNs are connected via Ethernet while the RCN and HPNs are connected via SpaceWire. The HPNs act as a bridge between the two networks.

the RCNs, a space qualified LEON3 processor is used, synthesized in a space qualified FPGA. The interface for the RCN includes two SpaceWire ports, multiple UARTs and a telemetry and telecommand (TM/TC) interface conforming to the CCSDS standards which can be connected to traditional satellite buses. It should also be able to handle Command and Data Handling (C&DH) tasks.

2.2 Software Architecture

The software stack of one processing node is presented in figure 3. To implement the distributed system a middleware design approach was chosen. General system management is handled by the Operating System (OS). The middleware in user space is responsible for setting up the distributed tasking computing system. Currently, the middleware supports running on both Linux and Real-Time Executive for Multiprocessor Systems (RTEMS). This article focuses only on Linux setups. Applications interfacing with the middleware, can also access the functionality provided by the OS. The middleware is responsible for task execution, distributing tasks between different nodes, handling communication between the nodes and failure detection (through the observation of other nodes) and recovery. The middleware also offers an API for the applications, for using the distributed computing and failure recovery features of the distributed Tasking framework. The middleware is not tagged to the onboard computing hardware. As it supports Linux, the application can be developed on a standard workstation before subsequently being recompiled for the HPNs. Each node then simply represents its own Linux process and for networking, the Linux network stack uses a virtual network card for each virtual node. For the HPNs, the Poky-Linux distribution from the Yocto Project is used. It is a Linux distribution specially designed for embedded devices. One of its major features is that it offers a significant portion of widely used open-source projects as packages. This allows them to be defined as requirements and have them automatically built into the deployed images. This includes OpenCV, the Boost libraries, Eigen3 and libjpeg. This significantly simplifies porting existing software projects to the onboard computer system, because there exists a high probability that the external open-source libraries being used are already available as packages.

2.3 Application Programming Interface

In the following, the API of the middleware is presented. Since a distributed computing architecture was chosen for the onboard computer design, the middleware API should reflect this fact. This was done by choosing a message based data-flow architecture as the programming model. In this model the basic entity of program execution is a task. Tasks communicate with one another through so called channels. There are generally two types of tasks, processing and event tasks. Processing tasks receive input messages, process them and optionally send resulting messages to other tasks. Processing tasks are triggered as soon as all required input messages are available (see figure 4a). This behavior can be modified to require a specific number of messages per input or execute the task as soon as a specific input receives a message. The second kind of tasks are event tasks (see
Figure 3: The system layers that constitute a ScOSA application. The middleware and libraries isolate the application from the OS which allows it to be cross-platform compatible. For testing purposes, the native OS API may still be called.

![Diagram](https://www.iconarchive.com/show/mushroom-icons-by-jommans/Clock-icon.html)

(b) Event task with one output, triggered by a clock^4.  

Figure 4b). Event tasks do not receive data, they only output it. Therefore, they cannot be triggered by incoming data but are executed by external events. To trigger them periodically or at a specific time, timer channels are used.

In figure 5 the integration of a simple filter function into a processing task which has two input arguments and two output arguments is shown.

When building an application, the different tasks have to be linked to one another. A data flow schema of a typical analysis application is presented in figure 6. In this figure, it can be seen that performance can be gained through parallelization. In the example, the two most important patterns are demonstrated.

```cpp
1  Fifo<T1> input_channel1;
2  Fifo<T2> input_channel2;
3  Fifo<T3> output_channel1;
4  Fifo<T4> output_channel2;
5
6  //defined in an external library.
7  extern std::tuple<T3, T4> filter(T1, T2);
8
9  ExecutorTask filter_task( 
10     FifoTuple<T1, T2>(inputChannel1, inputChannel2),
11     filter ,
12     FifoTuple<T3, T4>(outputChannel1, outputChannel2)
13 )
```

Figure 5: C++ implementation of the task presented in figure 4a.
1. Pipeline processing
Dividing a task into a series of independently executable subtasks where the output of one element is used as the input of the next one. If this subtask can be executed in parallel, a throughput increase of factor $N$ can be gained.

2. Single Instruction Multiple Data (SIMD)
Performing the same operation on multiple data. In the above example, this is realized through the use of three parallel pipelines (each consisting of a process A and B task). The data source alternates between sending the data to be processed by the three different execution pipelines. In this case, a throughput increase of $M$ can be gained.

In the example, both patterns can be combined which results in a speed up given by, $M*N = 3*3 = 9$. This ignores the overhead incurred by combining both approaches.

Task execution is handled by the Tasking framework. The number of tasks that can be defined is independent of the physically available resources. If for example only one CPU core is available, the tasks are executed sequentially. If more CPU cores are available, the Tasking framework will execute several tasks in parallel. Also, tasks can be deployed on different nodes. The data transport from one node to another is automatically handled by the Tasking framework. For example, a typical configuration of the example in figure 6 uses both available CPU cores of each node. The task-node mapping configuration has to currently be written manually. Figure 8 shows the part where registration is handled. In the future, task distribution should be generated automatically. Task node distribution is defined independently of the application task data flow schema. So for different hardware setups, different configuration files can be used, without changing the application implementation. As a last remark, there are also system tasks running on the nodes, that are responsible for failure detection and recovery and system (re-)configuration. One example consists of heartbeat tasks that check if other nodes are running correctly. As these tasks are not relevant for application development, they are not discussed in detail. More information can be found in the paper describing the preceding project, Onboard Computer - Next Generation (OBC-NG).³

---
³For trivially copyable types (see https://en.cppreference.com/w/cpp/named_req/TriviallyCopyable).
To illustrate the development, the porting of a ship detection application to the ScOSA onboard computing platform will be demonstrated. Details of the ship detection application can be found in. The application was available as an experimental version for demonstrating and analyzing the capabilities of a ship detection method. The application is a C++ application using the open-source computer vision library OpenCV for reading in image data and for applying various filter functions to the images. Also, the Boost library was used for various components, such as file system access and the unit-test framework. For development, a desktop Linux distribution was used.

The first analysis that was done was checking if there are any incompatible dependencies regarding the ScOSA framework. The critical dependencies are, in most cases, external software and hardware components. If external components are required that can not be integrated in the ScOSA system, they have to be rebuilt completely. This is often the case with proprietary hardware and software where only binary libraries or drivers are available. Integrating these components into the ScOSA system would have incurred untenable overhead. In our example case, we had no special hardware dependencies. The external software components we used where the open source libraries OpenCV and Boost which meant that the source code was freely available. Gladly these libraries are supported by Embedded Linux distribution from the Yocto Project (Poky-Linux). The only issue we had in this regard was, that the standard Poky-Linux distribution we used in project was out-of-date, and did not support a more recent version of the Boost library we used. But as there are newer Poky versions available with relative up-to-date software packages, we could have easily built a more recent version of Poky and deployed it on the board. This is also an advantage of using COTS components for hardware and software and developing against standard COTS interfaces. This meant that even central components such as the operating system could be easily replaced and updated.
We also checked the hardware requirements of the application, as even though ScOSA is powerful for an onboard system, a modern desktop computer offers significantly more computing resources. CPU performance was not an issue since we had no real-time requirements. It would simply take longer to process the data. A real issue was the size of the images. As RapidEye images have a original size of $5000 \times 5000$ pixels at 16 bit per pixel for each channel, this results in around 47 MiB per image. This has a direct effect for the amount of Random-access memory (RAM) required to run the application. As 5 channels\(^5\) are used, this adds up to $47 \text{ MiB} * 5 = 235 \text{ MiB}$. Memory efficiency was not taken into account when implementing the original ship detection algorithm. Therefore, we were not able to use full images on our test hardware. As a result, it was decided to use smaller images to circumvent the memory limitation. At a later stage, we could optimize the implementation if necessary. Through that analysis, we did not see any obstacle for porting our application to the onboard computer.

### 3.2 Application Architecture

After doing the compatibility analysis, we thought about how the application could be integrated into the ScOSA system most efficiently. In former projects, like VIMOS - Modular Commanding and Execution Framework for Onboard Remote Sensing Applications\(^{13}\) (VIMOS), we simply took the most current stable version of an application and ported it to the onboard system without thinking of code re-usability. As the API of the onboard computing system of this project was a non-standard implementation of the C++ API, this seemed to be the only viable way. But with this approach we always have the disadvantage, that we cannot use the application, in another context without heavy re-factoring or reimplementation. For this application we decided to share as much code as possible of the ship-detection code base with other projects. Also, if changes to the ship detection algorithm were made in other projects, it should be trivial to backport them into the ScOSA application variant.

For that reason we restructured the software in the form illustrated in figure 9. The whole application processing routines should be separated into application libraries. This library and the parts of external sub-libraries used, should be built on a general API, that is supported by all target platforms. In our case, our application library and the parts of external libraries used are compatible to the ISO-C++ standard.\(^{14}\) By using the general ISO-C++ standard, we have a good chance that it will be compatible with future project system environments. The system relevant parts of the application should be spun out into their own interface library. To form an application, these two libraries are then combined into a system dependent application integration file. This should, in ideal cases, only be a small glue code file. With that approach, we want to achieve a separation between the application code which is system independent and that which is system specific. This is a big advantage of the ScOSA system supporting an general API\(^6\), because it allows code reusability.

### 3.3 Development Environment

When starting development, the first critical point is having an efficient development system at your disposal. In the ideal case, the developer should be able to develop and run the whole application on their desktop computer. This speeds up the development process enormously and allows development without having access to the ScOSA onboard computing hardware. This is possible with the ScOSA architecture. As seen in figure 3 the ScOSA application is built upon the ScOSA middleware and the OS layer. As the middleware is completely Linux compatible, we could build a version of the application on a desktop Linux development system. Furthermore, since Poky Linux is used as the ScOSA onboard computing operating system, we can use the same middleware source code for the development version as for the onboard computer version. Only when building the application do we have to decide which hardware platform we will be targeting. This means that the same code base can be used for the development system as well as for the onboard computing system, which is a big plus for compatibility.

As stated in the beginning, the ScOSA onboard computer is a distributed computing system as shown in figure 1. This is taken advantage of by executing the same binary on each node with varying environment variables. For communication, the Linux network API is used. We, therefore, can set up a distributed computing network

---

\(^5\)blue,green,red,red edge, NIR

\(^6\)Linux API
simply by generating virtual network interfaces and letting the binaries for each node run in parallel on our development system.

For testing the performance of the application, the desktop development system is not the best choice, because it is significantly more powerful than the onboard computer. We used the COTS ZedBoard development kit, which contains a Xilinx Zynq-7000 SoC XC7Z020-CLG484-1 processing unit. As Xilinx Zynq-7000 chips are also used for the onboard computer’s HPNs, the performance mirrors that of an HPN. The only significant difference is that on the ZedBoard only 512 MiB RAM are available instead of the 1024 MiB on the HPNs. One big advantage of the ZedBoard is that it has an SD-Card slot, which the HPNs do not have. This makes deployment of the software more convenient. As ZedBoards also have an Ethernet interface, we could build a distributed computing network consisting only of ZedBoards, which is comparable to the onboard computer configuration. This is also an advantage of using COTS components, because a prototype system can be acquired at once, at a relative low price point. This allows application developers to test on a comparable system, even if the onboard computer hardware is not available. Within our development time, this was often the case, because the prototype system at first was still in development and later other project partners urgently needed it multiple times.

3.4 Software Development Process

In this chapter I will give a short overview of how we built a first ScOSA application version, based on a standard C++ application. We started with a C++ application that was in the form of 9, that is separated in system-dependent interface functionality and general application library functions. The application source code is structured as shown in figure 10. When an application is in that form, it is straight forward to integrate it into the ScOSA system. We simply added a few adapter functions to bring the functionality into the form required by the ScOSA Tasking framework. This is represented in figure 11. With this we then could easily construct a simple distributed Tasking application presented in figure 12. The accompanying implementation integration file is presented in figure 13.

3.5 Performance Optimization

When a first feature complete version of the ScOSA application was finished, we started doing some performance analysis on our development system and the ZedBoard. The ZedBoard represents one node of onboard computer. This can be done very comfortable with the ScOSA’s Tasking framework, because it offers built-in

**we have not done this.

††Intel(R) Core(TM) i7-4810MQ CPU @ 2.80GHz, 16 GiB Ram.
‡‡Xilinx Zynq-7000 SoC XC7Z020-CLG484-1, 512 MiB Ram
while (true)
{
    Image_t input_data = readData();
    Result_t result_data = ship_detection(input_data);
    writeData(result_data);
}

Figure 10: Form of the original C++ application

std::tuple<Image_t> readData()
{
    Image_t input_data = readData();
    return std::tuple(input_data)
}

std::tuple<Result_t> shipDetection(Image_t&& input_data)
{
    Result_t result_data = ship_detection(input_data);
    return std::tuple(std::move(result_data));
}

void writeData(Result_t&& result_data)
{
    writeData(result_data);
}

Figure 11: Wrapping the original functions presented in figure 10 to make them compatibly to the ScOSA API presented in figure 5

Figure 12: Schema of the application ship detection data flow.
Fifo<Image_t> inputData;
Fifo<Res_t> resData;

#define FT FifoTuple

EventTask TaskReadData(readData, FT<Image_t>(inputData));
ProcessTask TaskShipDetect(
  FT<Image_t>(inputData), shipDetection, FT<Res_t>(resData));
ProcessOutOnlyTask TaskWriteData(FT<Res_t>(resData), writeData);

Tasking::initialize();
Tasking::start();
auto period = 1s;
auto cycles = 10;
TaskReadData::start(period, cycles);
std::this_thread::sleep_for(15s);
Tasking::waitUntilEmpty();
Tasking::Terminate();

Figure 13: Core parts of the system depended application integration file (see figure 9), implementing the data flow presented in figure 12, by using the functions in figure 11.

<table>
<thead>
<tr>
<th>Task</th>
<th>Execution Count</th>
<th>Intel Core <a href="mailto:i7-4810MQ@2.80GHz">i7-4810MQ@2.80GHz</a></th>
<th>Xilinx Zynq-7000 SoC XC7Z020</th>
</tr>
</thead>
<tbody>
<tr>
<td>TaskReadData</td>
<td>2500</td>
<td>0ms</td>
<td>0ms</td>
</tr>
<tr>
<td>TaskShipDetect</td>
<td>2500</td>
<td>1942ms</td>
<td>64614ms</td>
</tr>
<tr>
<td>TaskWriteData</td>
<td>2500</td>
<td>0ms</td>
<td>0ms</td>
</tr>
<tr>
<td>Total Computation Time</td>
<td>3869ms</td>
<td>67250ms</td>
<td></td>
</tr>
</tbody>
</table>

Figure 14: Performance analysis of the ship detection application with data flow layout as presented in figure 12.

computation time tracking and execution counters. For the version presented in figure 13, the results are shown in figure 14. The first result was that most computing time is spent in the TaskShipDetect task. This is positive since it shows that there is no bottleneck in the disk IO operations or while passing data between tasks. The time spent in the other tasks is below the resolution of the computation time tracker. The second result is that the version on the desktop development is around 30 times faster than the version running on the ZedBoard. This is no surprise and provides a frame of reference of the computing performance that can be expected of a single node. To measure the performance difference, the application was structured to enable parallel processing. Its design, as shown in figure 6, consists of three ship detection tasks running in parallel. The results can be seen in figure 15. The input data is distributed among the three TaskShipDetect tasks, but no additional performance is gained. Therefore, TaskShipDetect is also not a real bottleneck, and for the optimization for one node we are finished. This shows that the ship detection application running inside TaskShipDetect is quite optimized for multi-core CPUs, and splitting it up further is useless for a single node configurations. The reason is that the thread count begins to exceed the available cores. This can be clearly seen when comparing the computation time of tasks TaskShipDetect 1-3. One outlier is the difference in computation time between TaskShipDetect 1 and the other two TaskShipDetect tasks. For more optimizations, we would have to optimize the ship detection routines themselves, which was not planned for the project. As a final note, the complete computation time is greater than the sum of each of the tasks' computation time. This time is attributable to static initialization and the setup routines. Further performance evaluations will be presented in future publications which will include a detailed looked at the performance gains attainable through the use of a multi-node system.
### 4. CONCLUSION

The first part of this article presented the architecture of the onboard computer developed at the DLR. It uses a distributed computing network based on a mixture of classical space qualified and COTS hardware computing nodes to archive high computing performance and robustness in a space environment. Mirroring this approach on the software side, a widely used operating system, standard toolchains and common libraries are used. In the second part, we showed how an application development process for the onboard computer platform looks like, starting from an already existing application. We showed what has to be checked to ensure that the application can be ported to it. One big focus was code re-usability. Existing software components should be reusable for an onboard version of an application, to save development costs and archive a higher quality. This is possible with the ScOSA software platform, due to using general system interfaces, particularly a Linux based software stack, and a Linux compatible middleware. This is an advantage over classical onboard architectures, because due to their proprietary API, software often requires significant restructuring. Another advantage of using a Linux based software stack is that the whole application development including the distributed computing layout can be run on a Linux development system. This significantly speeds up the development process. First of all, it allows development without needing to have access to the onboard computing hardware. Another equally important reason is that it accelerates software testing and allows continuous integration on the development system. In the end, we started to show how the computing performance of an application can be measured within the ScOSA API. We measured the performance that can be reached with one computing node and compared it with the performance of a desktop computer. Testing showed that the desktop computer was around 30 times faster compared to an onboard computing node. A multi-node configuration and the integration of the FPGA will allow us to leverage the complete computing power provided by the hardware. In addition, we hope to be able to track the advances in COTS computing power through the flexibility afforded by ScOSA’s system architecture. Unfortunately, due to the heavy development state of the project, we were not yet able to analyze the performance of a multi-node configuration. This should happen at a later time point.

### REFERENCES


[5] “SoCs with hardware and software programmability.”

