

Master Thesis

Deggendorf Institute of Technology, Campus Cham

Faculty of Applied Natural Sciences and Industrial Engineering

Master Mechatronic and Cyber-Physical Systems

German title:

Leistungsbewertung für einen verteilten On-Board-Computer

English title:

Performance Evaluation for a Distributed On-Board Computer

Master thesis to obtain the academic degree:

Master of Engineering (M.Eng.)

Submitted by: **Mahmoud Mostafa Elbarrawy**
Matriculation: **00822092**

First examiner: **Prof. Dr. Frank Denk**

Germany, January 25, 2023

Declaration

Name of the student:

Mahmoud Mostafa Hussein Hassan Elbarrawy

Name of the first examiner:


Prof. Frank Denk

Title of the master thesis:

Performance Evaluation for a Distributed On-Board Computer

1. I hereby declare that I have written this master thesis independently. I have not submitted it for any other examination purposes. I have not used other references or material than mentioned in the bibliography and I have marked all literal and analogous citations.

Cham, (date) 24.01.2023


signature of student: Mahmoud Elbarrawy  Digitally signed by Mahmoud Elbarrawy
Date: 2023.01.24 23:06:16 +01'00'

2. I agree that my master thesis may be made available to a broader public by the DIT library. Therefore, I hand in a further bound copy of my master thesis.

yes

no

Cham, (date) 24.01.2023

signature of student: Mahmoud Elbarrawy  Digitally signed by Mahmoud Elbarrawy
Date: 2023.01.24 23:06:56 +01'00'

I declare and take the responsibility that I am the exclusive owner of all rights concerning the master thesis, including the right of disposal concerning drafts and attached illustrations, plans or similar and that no third party rights or claims or legal requirements will be made upon making this master thesis publicly available.

To be filled in by the first examiner in the event that the author agrees with public accessibility of the master thesis:

Adding a copy of the master thesis into the stock of the library and lending the copy is:

approved

not approved

Cham, (date) _____

signature of first examiner: _____

Acknowledgments

I would like to thank God for everything. Thank God for such wonderful, supportive parents. I would like to thank M. Eng. Patrick Kenny, Dr. Carlos Gonzalez, and Dipl.-Ing. Daniel Lüdtké for supporting me through this master's thesis project. I can't find the words to express my appreciation for all of you. Special thanks to my professor, Frank Denk, for his continued support. Thanks to Dr. Andreas Lund for always being there when I needed help, and thanks to all my teammates. Last but not least, I'd like to express my gratitude to the Deggendorf Institute of Technology and the Deutsches Zentrum für Luft- und Raumfahrt (German Aerospace Center DLR)-Institute of Software Technology for allowing me to work on this fascinating master's thesis.

Abstract

Autonomous spacecraft, satellite communications, and Earth observation are examples of the growing significance of onboard applications with high demands for the processing power of onboard computers in the space domain. The space-grade hardware used for onboard computers does not cope with the computational demands of the current space applications. As a result, engineers are increasingly using Commercial off-the-shelf (COTS) components to build spacecraft. However, the COTS are vulnerable to cosmic radiation. The German Aerospace Center (DLR) addressed this problem and found a solution by creating a distributed system, the Scalable On-board computing for Space Avionics (ScOSA) with a heterogeneous number of nodes. This thesis focuses on selecting and applying a suitable benchmark for performance evaluation of the ScOSA distributed system. The results of the selected benchmark (OBPMark-Image Processing) show an increase in performance from 0.968 megapixels/s in one node to 1.538 megapixels/s while using three nodes.

Keywords: space domain, benchmarking, performance evaluation, floating-point operations, image processing.

Contents

Acknowledgments	ii
List of Abbreviations	vii
List of Figures	viii
List of Tables	ix
1 Introduction	1
1.1 Motivation	1
1.2 Problem statement	3
1.3 Contribution and scope	4
1.3.1 Selection of appropriate benchmarks	4
1.3.2 Adaption of benchmarks to ScOSA system	5
1.4 Structure	5
2 Literature Review	6
2.1 Benchmark definition	6
2.2 History of benchmarks	6
2.3 Benchmark types for performance evaluation	8
2.4 Common performance Benchmarks	9
2.4.1 Stone age Benchmarks	9
2.4.2 General purpose CPU Benchmarks	11

2.4.3	Embedded and media benchmarks	13
2.5	Benchmarks for distributed systems	15
2.6	Benchmarks for the space domain	17
2.7	Benchmark comparisons	20
3	Methodology	22
3.1	ScOSA Middleware architecture	22
3.2	Benchmarking:	25
3.3	Benchmark selection	26
3.4	Hardware	27
3.5	Benchmark Implementation	28
3.6	LINPACK Implementation	28
3.6.1	LINPACK One Node	28
3.6.2	LINPACK three Nodes	28
3.7	OBPMark in Details	29
3.7.1	OBPMark One Node	32
3.7.2	Benchmark Distribution	33
3.7.3	Modeling Approach for Distribution	36
3.7.4	OBPMark three Node	36
4	Results and Discussion	39
4.1	LINPACK	39
4.2	OBPMark	41
4.2.1	OBPMark One Node	41
4.2.2	OBPMark Three Nodes	41
4.2.3	Team server x86_64 Results Discussion	43
4.2.4	Zynq-7000 Results Discussion	45
4.3	Verification	46
5	Conclusion	47

6 Future Work	49
7 Appendix	50
References	53

List of Abbreviations

ScOSA	Scalable On-board computing for Space Avionics
SPEC	Standard performance evaluation corporation
TPC	Transactions processing council
IC	integrated circuits
COTS	Commercial off-the-shelf
DLR	The German Aerospace Center
OBC-NG	On-Board Computer—Next Generation
CPU	central processing units
BLAS	Basic Linear Algebra Subroutines
HPN	High-performance node
RCN	Reliable communication nodes

List of Figures

1.1	ScOSA-illustration example [1]	3
2.1	EEMBC-product-timeline [2]	14
3.1	The stack software architecture layout of ScOSA. The middleware is made up of the three components labeled in blue — System Management Services, the Distributed Tasking Framework, and the SpaceWire-IPC protocol [1].	23
3.2	Benchmark Flow Chart	26
3.3	ScOSA HPN Software Development Model	27
3.4	LINPACK one node setup	28
3.5	LINPACK three node setup	29
3.6	OBPMark Image-processing pipeline [3]	30
3.7	OBPMark one node setup	32
3.8	Distribution into four quarter sub-frames	33
3.9	Overlapping Distribution	35
3.10	OBPMark three node setup	37
4.1	OBPMark three node Event, Time Diagram Team-server	43
4.2	OBPMark Team server Performance	44
4.3	OBPMark three node Event, Time Diagram HPNs	45
4.4	OBPMark HPNs Performance	46

List of Tables

2.1	GPU Bench workloads	19
2.2	OBPMark overview [4]	20
2.3	Benchmark Comparison	21
4.1	Hardware Specification	39
4.2	LINPACK Results	40
4.3	OBPMark One Node results	41
4.4	OBPMark distributed results	42
7.1	LINPACK One Node development machine detailed three-run results	50
7.2	LINPACK One HPN Node detailed three-run results	50
7.3	LINPACK Three Nodes development machine detailed three-run results	50
7.4	LINPACK Three HPN Nodes detailed three-run results	51
7.5	OBPMark one Node results detailed three-run results	51
7.6	OBPMark distributed results for the development machine three runs	51
7.7	OBPMark distributed results for the HPNs three runs	52

Chapter 1

Introduction

1.1 Motivation

Space domain applications, such as autonomous spacecraft, satellite communications, and Earth observation, are examples of the growing significance of on-board applications with high demands for processing power in the space domain. To interpret and evaluate the massive volumes of data produced by sensors and equipment in outer space, these applications will require cutting-edge computer systems and algorithms.

On-board computers are an essential part of spacecraft and are responsible for managing various systems and functions on the craft. Unlike typical embedded systems, onboard spacecraft computers must have certain qualities (e.g., high performance and reliability) due to the harsh nature of space. The software and hardware for the spacecraft cannot be directly maintained after the mission has launched, which led to the need for reliability on both the software and hardware sides. Moreover, the integrated circuits (IC) inside the system are vulnerable to cosmic radiation. This may cause soft errors [1] in electronics, which in turn will cause a wide variety of issues in the spacecraft's operational software.

To avoid such problems, engineers usually use space-grade hardware. Those types of hardware processing units are referred to as "radiation-hardened" reliable computing nodes. However, they cost much more than normal processing units, and the computation performance is lower compared to cutting-edge embedded hardware on the market. For this reason, engineers are increasingly using commercial off-the-shelf COTS parts to construct/build spacecraft, improving performance and decreasing costs at the expense of reliability. Obtaining both reliability and performance without sacrificing either is one of the scientific challenges in the space domain at the moment.

DLR started exploring this problem in 2012, which led to the 2013 On-Board Computer—Next Generation (OBC-NG) project [5] and the 2017 ScOSA project [6]. These projects resulted in a heterogeneous onboard computer architecture by combining radiation-hardened hardware with high-performance COTS processors. The ScOSA architecture is designed to ensure that the spacecraft has access to high-performance and reliable computing power while in orbit. To accomplish this, a distributed computer with several heterogeneous processors connected through a network is used. The ScOSA middleware layer is utilized to control the heterogeneous system and to create distributed computing applications without worrying about the complexity of an underlying distributed system.

After the ScOSA project, another project began with the title of the ScOSA Flight Experiment, which started in January 2020. This project goal is an in-orbit demonstration of the ScOSA onboard computer.

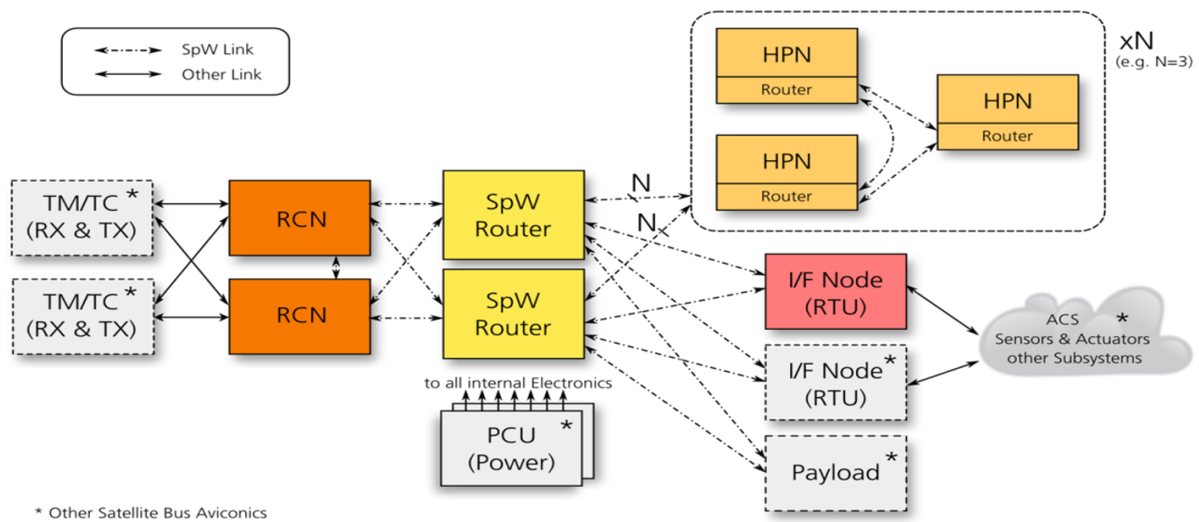


Figure 1.1: ScOSA-illustration example [1]

Figure 1.1 shows one possible way of configuring ScOSA system architecture, but not the only one. In order to communicate with the telecommand and telemetry units, the system relies on two Reliable communication nodes (RCN)s. SpaceWire routers connect the RCNs to the N high-performance nodes and the interface nodes, which in turn link the sensors and actuators.

1.2 Problem statement

There are a lot of challenges that distributed computers face. Especially in distributed computing systems that consist of multiple interconnected components (nodes) across a network, typical challenges that engineers try to overcome while constructing those systems are scalability (increasing the number of hardware nodes while ensuring that the overall system functions efficiently), debugging and monitoring the system while overcoming potential communication failures, optimizing the latency of data transfer, efficiently managing data across different nodes in an organized way, and interoperability (of heterogeneous nodes), making sure that they are compatible to function together and can overcome potential node failures by continuing to function.

The ScOSA system is comprised of N nodes, as illustrated in Figure 1.1. The system is in continuous development to overcome the previously stated challenges. The challenge that ScOSA faces is due to the flexibility of the heterogeneous system, which offers the application engineers a wide range of options for the number of processing nodes to select from.

Thus, this thesis addresses the following research question:

”How much processing power do we have and how many nodes should I add to reach the required performance?”

Evaluating how well these systems and algorithms function (e.g., ScOSA) is difficult due to the heterogeneity, distribution, and space-domain complexity of their systems. One way to evaluate the performance and reliability of spacecraft onboard computers is through the use of benchmarks. A benchmark is a set of tests or metrics that are used to measure the performance of a system or device. In the context of onboard computers, benchmarks can be used to measure factors such as computational speed, memory bandwidth, power consumption, and reliability.

This in return generates the following further relevant secondary questions:

- Which benchmarks are more suitable for our distributed system?
- How to adapt or create benchmarks for the ScOSA system?

1.3 Contribution and scope

The goal of this Master Thesis is to design and carry out a performance evaluation of the ScOSA system by selecting and applying appropriate benchmarks. Determining which is an appropriate benchmark for this distributed system would be valuable for future developments.

The proposed work includes the following steps:

1.3.1 Selection of appropriate benchmarks

A range of standard benchmarks for computing systems exists, including those aimed at generic computer systems, embedded systems, and space-specific benchmarks. As ScOSA is a space-specific, heterogeneous, and distributed computing system, not all benchmarks will be appro-

priate. Benchmarks will be selected based on criteria such as their appropriateness for the ScOSA hardware and software, including their source-code availability and licensing requirements, their feasibility of implementation, and their relevance to the application types used in ScOSA.

1.3.2 Adaption of benchmarks to ScOSA system

The selected benchmarks must be implemented to run on the ScOSA system. This may require adapting them to the C++ programming language used in ScOSA and having them built using the ScOSA middleware's Scons build system for the embedded targets. It will also require the design of a parallelization strategy to make effective use of all available processing nodes. In this thesis, the benchmarks will be run on the software development model of the ScOSA system.

1.4 Structure

The thesis is structured into six chapters. Chapter One is the introduction, which contains the motivation and importance of performance evaluation for a distributed system as well as our contribution to the scientific field. Chapter Two is a literature review of the previous techniques used for performance evaluation and benchmarking. Chapter Three is the methodology part, where a description and explanation of the thesis work are shown. The results and outcomes of the thesis work are discussed in Chapter Four. Chapter Five is a conclusion and discussion. Chapter Six would be a future recommendation for possible scientific explorations in the same area of research.

Chapter 2

Literature Review

This literature review focuses on reviewing benchmarks for performance evaluation. The literature shows benchmark definitions, history, different types of benchmarking techniques, current benchmarks for distributed systems, a comparison between current benchmarks with various aspects, and compatibility for embedded onboard computing systems.

2.1 Benchmark definition

A benchmark is "a standard method for evaluating and comparing competing systems or components based on certain qualities, such as performance, reliability, and security" [7]. This definition is one of the definitions that focus on the competitive features of benchmarks [8].

2.2 History of benchmarks

Performance benchmarks have contributed significantly to the development of the computing domain [9]. Since the seventies, there have been standardized tests of central processing units (CPU) and other processors since the seventies. Modern computer workloads contain different types of programs, making it difficult to establish standard benchmarks [10]. Benchmark-based performance assessment has always been contentious. Even recently, peo-

ple have purchased computers primarily based on clock speed or memory capacity, not test results. In the 1980s, computer performance was measured using small program portions (kernel) taken from applications (e.g., Lawrence Livermore Loops, Linpack, Sorting, the Sieve of Eratosthenes, the 8-Queens Problem, and the Tower of Hanoi) or synthetic programs such as Whetstone or Dhrystone [11]. Synthetic benchmarks serve no practical purpose other than benchmarking, and they have no real-life application usage. Whetstone is a synthetic floating-point benchmark based on numerous programs. When evaluating floating-point arithmetic performance, the Whetstone benchmark is used. Dhrystone is an additional integer performance benchmark created in the 1980s. Both were popular for years. Both programs were rudimentary attempts to generate a benchmark suite application. However, many of the findings calculated by the programs were never published or utilized. Compilers could easily delete a huge portion of dead code because of the optimization properties of the compiler, or the results would be invalid because of the compiler's debug mode, which gets more safety information on the compiled code to help debug, and which adds overhead to the performance time. Also according to Weicker's 1990 study, manufacturers misused/abused synthetic benchmarks to get a high score for their hardware [12]. Because of all that was mentioned, there was a need for a standard, agreed-upon way for developers to benchmark the performance. The Standard performance evaluation corporation (SPEC) [13] and Transactions processing council (TPC) [14] was founded in 1988 to enhance benchmarking and make sure there is no abuse of the compiler optimization properties.

The user's application is often cited as the gold standard for benchmarking, based on the applications that the machine will perform; the evaluation of how well this machine compares to others is comprehended. However, this is usually unattainable since the software cannot be installed on all of the machines involved in the system. Also, a slight change in hardware may cause a speed difference on the software side; an example of that is the cooling of the processing unit. If the cooling is not good enough, it may cause a lag on the software side. The complexity of benchmark software increases in proportion to its resemblance to actual applications. This makes measurements more challenging [11]. The three most popular benchmarks are Whet-

stone, Dhrystone, and Linpack, which are discussed in detail in section 2.4. Results from these standard benchmarks are often referenced in manufacturer and industry publications. Whetstone and Dhrystone are examples of synthetic benchmarks. Linpack is a benchmark that was derived from a practical program.

2.3 Benchmark types for performance evaluation

Benchmarks are categorized into four different types: real applications, small benchmarks, synthetic benchmarks, and benchmark suites [15]. But those categories do not expand to the types of distributed or heterogeneous system benchmarks. It was a general explanation of a benchmark. In reality, the systems are constructed from distributed nodes as resources, and in some cases, those nodes are heterogeneous. However, the explanation of these types is essential for literature.

Real application benchmarks are based on programs that have already been made. They are made up of the tasks that a typical user does during the day. The benefits of running real applications are that they directly show how fast programs run. But these benchmarks are usually very large and take longer to run and move to other machines. Also, it can be hard to find a processing bottleneck and figure out what instructions need to be improved.

Small benchmarks are only small segments of codes like loops and recursions. It is easily simulated during design to test functionality, and it offers developers a solid understanding of which part of their workflow is slowing down. However, designers may misuse this information and create an algorithm that performs well for a single task, but when multitasking occurs in a real application, the processing power, and speed drop.

Synthetic benchmarks are aiming to provide similar functionality to their more resource-intensive counterparts while using much less memory and CPU. Although they do not use the processor's capabilities in a meaningful task, they can accurately portray the processing power. When performing a benchmark, several iterations should always be performed; the timings may vary, therefore taking an average will provide more reliable results [16].

A benchmark suite is a collection of individual benchmarks from several sectors designed to simulate a wide range of computational demands on a system. To simulate a user's usual workload, many organizations collaborated to agree on a standard collection of applications. Suites are helpful since they cover a wide variety of qualities and traits. However, they need regular upgrades to keep up with the changing nature of workloads.

2.4 Common performance Benchmarks

In this section, the most common performance benchmarks are reviewed and categorized into three categories which are stone age benchmarks, general-purpose CPU benchmarks, and embedded system benchmarks.

2.4.1 Stone age Benchmarks

Stone age benchmarks are the first benchmarks introduced for performance evaluation, and they are well-known in the scientific communities [11].

Whetstone:

Whetstone is the first benchmark software for numeric programming. The whetstone modules have statements of integer arithmetic, floating-point arithmetic, "if" expressions, function/procedure calls, etc., which are "Whetstone instructions" of synthetic code [11]. The Whetstone benchmark produces an output metric of KWIPS or MWIPS (mega Whetstone instructions per second). Mathematical library functions take a considerable amount of time to execute. However, it may not apply to most numerical applications nowadays. Since the speed of these subroutines, or microcode, determines Whetstone's performance, manufacturers may modify the run-time library, which would be a manipulation of measuring the performance; that's why Whetstone is not representative of modern programs.

Dhrystone:

Dhrystone was created similarly to Whetstone in 1984 [11]. The original language was Ada. However, it utilizes just the Pascal subset and is easily translated into the Pascal and C languages. Dhrystone is a non-numeric benchmark that measures system-type programming language characteristics (e.g., operating systems, compilers, editors, etc.). System-type programming includes fewer loops, simpler computations, and more "if" and function calls. The Dhrystone benchmark evaluates the integer operations that processing units can perform [17]. The measuring unit of the benchmark is Dhrystones per second. In addition, Dhrystone Million Instructions Per Second (DMIPS) is still another representation of Dhrystone's result. Dhrystone's measurement uses no floating-point computations; instead, it uses string functions to evaluate the performance, which consumes most of the benchmark's execution time. Dhrystone's primary measurement loops have fewer loops than Whetstone's. Microprocessors with short instruction caches below 1,000 bytes always face the well-known problem of cache misses. Compilers are in continuous development, and that results in an optimization of the code in release mode, which might recognize numerous statements in Dhrystone as unnecessary since it is a synthetic benchmark and will not execute the instructions.

LINPACK:

LINPACK was not originally a benchmark, according to its inventor. In 1976, it was a bundle of linear algebra subroutines for Fortran applications. It was meant to give an idea of how long it would take to solve a set of linear equations. Dongarra [18], who gathers and distributes LINPACK results, has turned it into a benchmark with many versions [19]. The benchmark assesses the performance by generating a dense matrix problem $Ax = B$ and measuring how much time it will take to solve this matrix. The measuring unit is float point operations per second (Flops) or megaflops. The software uses a large matrix (a two-dimensional array), yet Basic Linear Algebra Subroutines (BLAS) manage it as a one-dimensional array [20]. BLAS serves as the foundation for the LINPACK software since BLAS handles much of the floating-point work inside the LINPACK algorithms. The floating-point operation rate can be determined by the

array size (the number of elements in the matrix). Non-floating-point operations are ignored, or their execution time is added to floating-point operations. However, this might be deceptive when floating-point operations become quicker than integer operations, but that was in the old versions. Also, data mapping for the cache line in the LINPACK benchmark is highly affected by the matrix size (array size), and when dealing with big matrices, it might cause a cache miss and eventually lead to a low Mflops rate. Currently, single and double floating-point precision versions are available for LINPACK.

LAPACK:

LAPACK uses Fortran 90 to solve eigenvalue, singular value, eigenvalue systems, and simultaneous linear equations. Related computations include rearranging Schur factorizations and calculating condition numbers (LU, Cholesky, QR, SVD, Schur, generalized Schur). Real and complex matrices in single and double precision use the same operations [21].

The LAPACK optimizes the LINPACK libraries on shared-memory vectors and parallel computers. LINPACK is inefficient on these processors because their memory access patterns don't take advantage of multi-layered memory hierarchies, wasting time transporting data instead of executing floating-point calculations. LAPACK overcomes this problem by putting matrix multiplication into algorithm inner loops. However, LAPACK is not much different from Linpack from a point of view of benchmarking as it only optimizes the algorithm order itself and it is not giving a clear precise value of processing power for space applications.

2.4.2 General purpose CPU Benchmarks

General-purpose CPU benchmarks are mostly for PCs and general-purpose computing power, not for specific workloads. However, for the sake of the literature review and history of benchmarks, it is necessary to include the most famous ones that played a vital role in shaping the standardization of benchmarks.

TPC:

The Transaction Processing Council (TPC) was created in 1988 to develop transaction processing and database benchmarks and publish objective, verifiable transaction-processing performance statistics [14]. Many commercial and computing operations are called "transactions." A "transaction" is a computer function that involves disk access, operating system calls, or data transfer across subsystems. TPC defines a transaction as an exchange of products, services, or money. A typical TPC transaction involves updating a database for inventory management (goods), airline bookings (services), or banking (money). Customers or service reps enter and manage transactions on a terminal or desktop computer linked to a database. TPC benchmarks assess transaction processing (TP) and database (DB) performance by how many transactions a system and database can do per unit of time, such as per second or minute [22]. As previously stated, the TPC focuses on benchmarking computers for domains unrelated to space. However, it illustrates how a benchmark measures performance in a distributed system for data exchange between processing units by measuring the latency of data transformation and evaluating based on transmission time. The master node is used as the main node that evaluates other nodes in the system. But it does not evaluate itself.

SPEC:

In 1988, a small group of workstation providers recognized the industry needed realistic, standardized performance assessments [22]. SPEC provides the benchmark with a standardized suite of source code from existing programs that its members have adapted to several platforms. SPEC source code benchmarks are based on actual user applications. These benchmarks examine processor, memory, and compiler performance. Each program in SPEC has many input data sets. The reference input set is huge, but test and train inputs are provided to verify it. However, SPEC is more of general purpose application benchmark which is difficult to apply on an embedded system.

GeekBench:

GeekBench comes from Primate Labs and works on several platforms [23]. It includes a suite of tests designed to mimic the typical use of the CPU. GeekBench is equipped with its exclusive method of gauging performance. One core's score (in points) is added to the total score. Better performance is reflected by a higher score. The issue with Geekbench is that in order to apply their latest version then there are minimum software/hardware requirements. e.g. at least for Linux distribution: Ubuntu 16.04 LTS (64-bit) or later, 2GB of RAM, and Intel Core 2 Duo or later for CPU unit. This is not the case for all nodes that we have for our space application and that makes it more in the general purpose category than utilizing it for space applications.

2.4.3 Embedded and media benchmarks

Embedded system benchmarks are used to evaluate the embedded software and hardware components of a system for specific types of application workloads. For space applications, embedded benchmarks are the fundamental basis for measuring performance. Also, it is the backbone for developing special kinds of benchmarks for the space domain.

EEMBC benchmarks:

The EDN Embedded Microprocessor Benchmark Consortium (EEMBC) was created in April 1997 to produce embedded processor benchmarks. EEMBC benchmarks [2] both real-world and synthetic benchmarks. Automotive, industrial, consumer, networking, office automation, and telecommunications benchmarks are covered. These benchmarks target engine control, digital cameras, printers, cellular phones, modems, and other microprocessor-based devices. The EEMBC team deconstructed applications from these fields and developed 37 benchmark algorithms. EEMBC's Certification Labs (ECL) in Texas and California provide certified benchmarking findings. EEMBC is sponsored by most of the processor industry and is the industry standard embedded processor benchmarking organization.

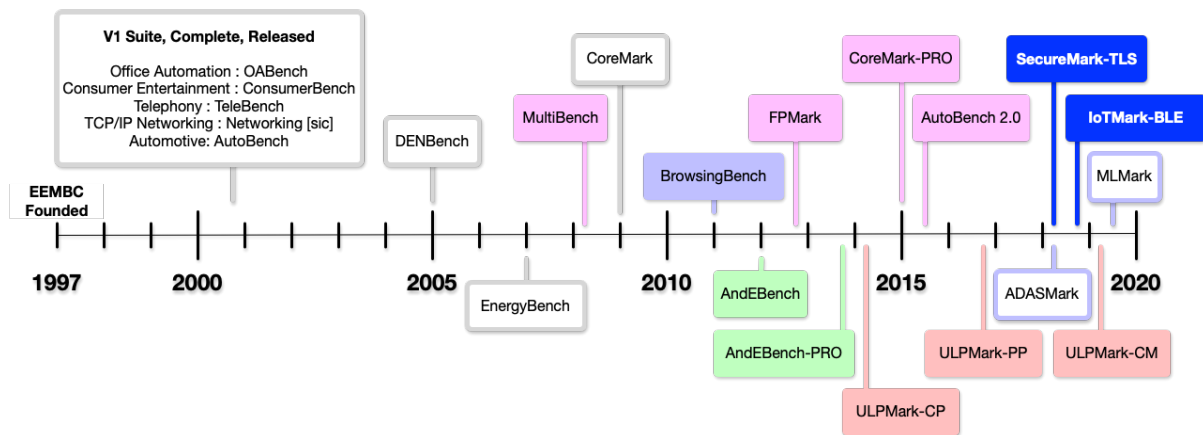


Figure 2.1: EEMBC-product-timeline [2]

Figure 2.1 shows the timeline of EEMBC product benchmarks and as shown there are a lot of versions of it. Which makes it a clear powerful candidate for the space domain since it is supported by most industry standards.

MiBench:

Freely available online [24, 25], MiBench is an embedded benchmark package with many of the same tools as the EEMBC suite. The EEMBC software bundle is not easily available to universities. Michigan researchers built the MiBench suite of 35 embedded applications to address this issue. The EEMBC suite serves as the basis for the categorization of software into the following groups: transportation, home use, office computing, mobile devices, security, and privacy. The C source code for all the apps is provided. The issue with MiBench is that it is not as well known as EEMBC, and that is why the reported results will be difficult to compare with others.

MediaBench:

The MediaBench is an academic initiative that aims to compile different benchmarks for media processing. Many image processing, networking, and digital signal processing (DSP)

applications are included in the MediaBench benchmark suite. e.g., voice compression, image compression, Ghostscript (a portable document format), and adaptive differential pulse code modulation for signal encoding. The outcomes of the 30th International Symposium on Microarchitecture provides an example of how these benchmarks are utilized [26]. But as mentioned with the MiBench, the MediaBench was created for academic purposes, which makes it unknown for users to report their results for comparison and verification.

BDTI benchmarks:

Formed in 1991, Berkeley Design Technology, Inc. (BDTI) has provided technical services with an exclusive emphasis on digital signal processing [27]. To evaluate how well various apps perform, BDTI creates bespoke benchmarks. Some examples of DSP procedures used in the benchmarks include FIR filters, IIR filters, FFT, dot product, and Viterbi decoder. However, the BDTI is not open source, and most of the claims that the benchmark assesses the processing power accurately are from their website.

2.5 Benchmarks for distributed systems

Until now, there has been no distributed benchmark application that is compatible with all of the system processing units without tweaking the algorithm to make it suitable, specifically if it is a heterogeneous system. However, there are some attempts to achieve a fully automated benchmark that works with distributed systems without worrying about the system setup. Their benchmark workloads are inspired by the common benchmarks that are mentioned in the above section. This section will provide an overview of the most well-known benchmarks for distributed systems.

Real-time distributed systems (RTDSs) can be set up in practically countless different ways. These differences come from the fact that hardware, system software architectures, and application needs are all different from one system to another. Because of these differences,

benchmarking RTDS depends more on how the system is configured, the nature of the hardware used, and the type of software running on those systems. In [28] the authors introduce benchmarking techniques as an application for the RTDS. Also, they introduced one way of modeling the RTDS as three scheduling domains: the node's processor scheduling domain, the node's communication scheduling domain, and the node's priority domain. A benchmark can only combine these three scheduling domains and be independent of hardware and system software architecture if it presents itself to the RTDS as an application. From an application's point of view, the RTDS can be thought of as a group of tasks (processes) that run on different nodes and send messages to each other. Both the tasks and the messages are time-constrained, and if the deadline is passed, then the benchmark fails or gets a low score for that task. However, the authors didn't present any rules or constraints for modeling the distributed benchmarks, but their technique is worth mentioning.

In [29] the authors introduced an approach for modeling the performance behavior of the distributed system, which depends on the execution state of distributed programs and subprograms. In their approach, the modeling depends on two factors: the available data and the transmission paths available for it. Capturing the time for this model is their main target. Before proceeding, it is a good idea to evaluate the available paths and the time taken by the data to move in this path so that the system can select the shortest time path to run. We can consider this in distributing benchmarks in general. But the paper did not discuss comparing or benchmarking their approach with other available approaches. Furthermore, their work is more of a simulation than an actual run on real hardware.

PEEL is a framework for benchmarking distributed systems and algorithms [30]. The purpose of the PEEL benchmarking framework is to provide a standardized way to develop, run, evaluate, and share experiments for evaluating the performance of distributed systems and algorithms. Currently, PEEL supports distributed stream processing frameworks of PC clusters such as Flink, and Spark [31]. Peel's key principles, such as experiment definitions and its experimentation procedure, were shown in an example of a supervised machine learning work-

load. However, the authors are more focusing on the group of PC computers and they are not discussing the low-level implementation of embedded hardware nodes.

The German Federal Ministry of Education, Science, Research and Technology (BMBF) initiated the IPACS-Project (Integrated Performance Analysis of Computer Systems) to provide a new standard distributed benchmark-execution framework by which to evaluate the efficiency of distributed systems [32]. The project targets main memory and disk access patterns for internal node communication, software design, and workload patterns to evaluate high-performance PC clusters. This is similar to [30]. The project did not expand the distributed benchmarking process for embedded systems. However, they provide a modeling approach for applying LINPACK benchmarks in a distributed system by using the framework to select the LINPACK and selecting the targeted computer then everything will work automatically. First, the source code is fetched from the NETLIB website, then the framework copied the code for the executable after that executable is copied to the target and executed, then the results are sent automatically to the user or the website for comparison. If the results are not logical in comparison to other CPUs of the same type on the website, the framework notifies the user about something wrong in the system. This notion is worth noting since it may be beneficial if we want to create a framework like that for space applications.

2.6 Benchmarks for the space domain

In [33] the authors introduced a framework to analyze processor architectures for onboard space computing. Their main metric measurements are computational density (CD) and CD per watt (CD/W) device metrics which are simply the processor's raw performance in terms of addition and multiplication operations per second [34]. The authors are looking for performance from two points of view: the benchmark performance and the device metrics that they use. The performance model for the work introduced is from the University of California. The model consists of space-computing taxonomies, which are similar to benchmark workloads for space applications [35]. They compared the serial and parallel performance of different processor

architectures with multiple cores; however, they did not discuss multiple nodes for their work, and the paper does not provide guidelines for implementation.

NPB:

NASA developed the NAS Parallel Benchmarks (NPB) to measure the efficiency of massively parallel supercomputers [36]. The benchmarks are implemented in software for computational fluid dynamics (CFD)-related tasks. The use cases for NPB are rare in onboard processing applications. However, there is an implementation of the fast Fourier transform (FFT) workload that might be useful, because FFT is a common workload in space domain applications [35].

EEMBC for space applications:

Several processors and embedded system benchmarks have been published by the EEMBC as mentioned in section 2.4.3. The CoreMark, designed to replace the Dhrystone standard, has gained widespread adoption. It has the same foundation as Dhrystone, mainly synthetic applications. However, it fixes problems inherent to Dhrystone such as the difficulty of implementing compiler optimizations and the lack of a standardized method for reporting Dhrystone findings. CoreMark has been utilized in the space domain, e.g., the single- and multi-core LEON processors [37] and [38]. However, comparing CPUs (central processing units) against other kinds of processing devices like FPGAs (field-programmable gate arrays), GPUs (graphics processing units), and ML accelerators is not a good use of such benchmarks. Other EEMBC benchmarks including Multi bench for measuring the performance of multicore processors and FPMark for measuring the performance of multithreaded floating point operations are available in the market. Additionally, two heterogeneous system benchmarks, advanced driver assistance systems (ADASMark) and machine learning (MLMark), have been made public. System-on-chips that perform (ADAS) activities for autonomous driving are the focus of ADASMark. The image processing pipeline incorporates standard ADAS features of pre-processing and object detection [4]. The benchmark is available to download in OpenCL and may be used to run on

CPUs, GPUs, or DSPs. Although the method and image processing, in general, are similar to those performed aboard spacecraft, they do not completely translate to the image processing tasks required by space applications. Although it is not an accurate representation of space systems, it may be used as a good baseline for general image processing.

GPU4S Bench:

The project GPU4S Bench (graphics processing unit for space application) aims to design and implement an open-source GPU benchmarking suite for space onboard computers. The project is targeting the GPU specifically because of its high potential for image processing and filter implementations that are used in space missions [39, 35]. The project has helped in creating an onboard processing benchmark (OBPMark), a new benchmark suite that is not only for GPUs but also for other different processing units available in the market, which is a major step in the benchmarking for the space domain (reviewed in 2.6). The building blocks of this GPU bench serve as a foundation for the main workloads that the space mission encounters in action. Table 2.1 contains the relevant workloads.

Table 2.1: GPU Bench workloads

Building Block	Domain
Fast Fourier Transform	Image processing and signal processing
Finite Impulse Response Filter	Vision based navigation and signal processing
Discrete Wavelet Transform	Data compression
Matrix Computation	Vision-based navigation, image processing and neural network processing
Convolution	Neural network processing, vision-based navigation and image processing
Correlation	Signal processing, vision based navigation and image processing
Memory Allocation	Image processing, vision-based navigation, signal processing and neural network processing

OBPMark:

The European Space Agency (ESA) and the Barcelona Supercomputing Center (BSC) have begun work on a set of benchmarks called OBPMark (On-Board Processing Benchmarks) that will encompass software often used on spacecraft [4]. The development of this benchmark, as

mentioned in section 2.6 comes from the GPU4S Bench project. However, it was also inspired by the industrial benchmarks of the EEMBC suites 2.6 in its implementation, making it more suitable for performance evaluation in space applications. Currently, the benchmark is in beta version with standard C, openMP, OpenCL, and CUDA implementations. Table 2.2 describes the standard workload domains for this suite.

Table 2.2: OBPMark overview [4]

Benchmark Name	Workload
Image Processing	Image calibration/correction and radar image processing
Standard Compression	Data compression, image compression and hyperspectral image compression
Standard Encryption	AES Encryption
Processing Building Blocks	FIR filter, FFT processing, convolution, matrix multiplication
Machine Learning Inference	Object detection and cloud screening

2.7 Benchmark comparisons

There are headlines to follow for building a benchmark from scratch depending on certain aspects that can also be used to select a specific benchmark. These aspects can be formulated into questions [7].

- **Relevance:** Is it related to the work domain of the measured processor or not?
- **Verifiability:** Does it have data to verify the output results?
- **Scientific popularity:** Is it famous and has a vivid well known measuring unit for the public?
- **Openness:** Does it have an open source code and is available for use without restrictions?
- **Access to reported data:** Is the reported data available to the public comparison without restrictions or not?

In response to those questions, we had a comparison between benchmarks that we have revised to select the most suitable one for our system.

Table 2.3: Benchmark Comparison

Benchmark	Relevance	Verifiability	Scientific popularity	Openness	Access to reported data	Type
Whetstone	0	++	++	++	++	Synthetic Benchmarks
Dhrystone	0	++	++	++	++	
LINPACK	0	++	++	++	++	Application
LAPACK	0	++	++	++	++	Benchmarks
TPC	--	+	+	--	0	General purpose CPU Benchmarks
SPCE	--	+	+	--	0	
GeekBench	--	+	+	+	++	
MiBench	0	+	+	++	0	Embedded Benchmarks
MediaBench	0	+	+	++	0	
BDTI	0	0	0	--	-	
EEMBC	0	+	++	0	+	
GPU4S	++	+	+	++	++	Space application
OBPMark	++	+	+	++	++	Benchmarks

(+ +): Excellent. (+): Good. (0): Neutral. (-): Bad. (- -): Worse

The aspects that the comparison is built on are very important for selecting a benchmark to ScOSA. From Table 2.3 The closest ones to reach a level of compatibility with evaluating all embedded systems would be the EEMBC benchmarks, but they are mostly in the automotive domain and are not tailored specifically for space applications. On the other hand, when looking at synthetic and application benchmarks, it is better to go on with application types so that they match the behavior of the system in real-life applications.

The only disadvantage of space application benchmarks is that they are significantly new. For example, the OBPMark suit is currently a beta version. When it comes to general-purpose CPU benchmarks, attempting to implement them in the space domain is a bad idea because they are not designed to evaluate the workloads of space applications.

Chapter 3

Methodology

In the methodology section, the workflow will be discussed based on the findings of the literature review. As seen in the previous table, there are several different benchmark applications. On our ScOSA system, we are looking to assess the performance of a distributed system with a relevant workload that is popular for benchmarking in the scientific community for space applications so that the results can be compared with other distributed onboard computers.

3.1 ScOSA Middleware architecture

In the introduction section, we talked about the ScOSA project in brief detail, but before diving into benchmark implementation, let's talk in detail about the ScOSA middleware architecture and what we need to be concerned about while using it.

The middleware of ScOSA consists of three main components

1. System Management services.
2. Network protocol (SpaceWire or Ethernet).
3. Distributed Tasking Framework.

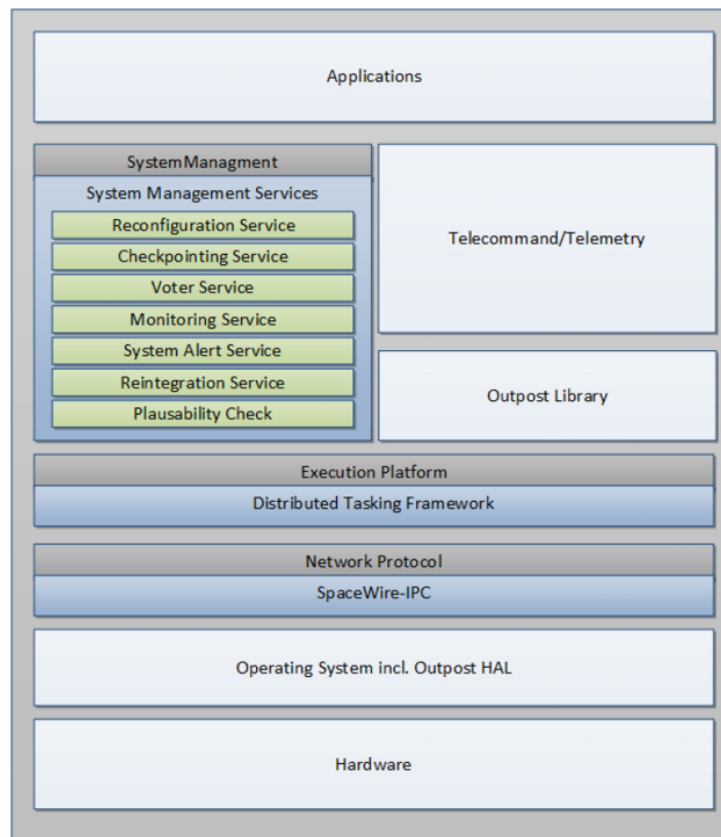


Figure 3.1: The stack software architecture layout of ScOSA. The middleware is made up of the three components labeled in blue — System Management Services, the Distributed Tasking Framework, and the SpaceWire-IPC protocol [1].

As explained in the figure 3.1, the stack layers show that the application is at the top and the rest of the ScOSA is at the bottom with all the protocols, libraries, and hardware. The middleware sorts the nodes into different types, such as observer, worker, or coordinator nodes.

The most important component for integrating and modeling the benchmark into ScOSA will be the distributed tasking framework, which will be explained in detail later. as a brief explanation of the other two components.

System management service: provides the middleware with fault detection, isolation, and recovery (FDIR) capabilities. As shown in Figure 1.1 example, the system consists of nodes, and because of the harsh nature of space, those nodes might face some errors, or for any reason, some of those nodes might not function normally. ScOSA middleware offers a reconfiguration

service that activates FDIR when such problems occur. To facilitate interaction across instances running on different nodes, the services will be created as threads and linked to the network stack. This program section is also accountable for archiving the system's present configuration settings and parameters, as well as the nodes' current statuses and assigned responsibilities such as "observer," "worker," and "coordinator."

Network protocol: it is the internal communication protocol between the nodes of the system. If you're familiar with the ISO/OSI reference model, you'll recognize SpaceWire-IPC as a component of Layer 4 [1]. Inter-process communication (IPC) between nodes and management data to and from the coordinator node is the primary goal of this layer, which supports Ethernet and SpaceWire data transmission protocols.

Distributed Tasking Framework: The distributed tasking framework is based on the tasking framework, which is an event-driven multithreaded execution platform for real-time on-board software systems [40]. The tasking framework is also developed by DLR internally, and it is an open-source framework [41]. The idea of a tasking framework is inspired by Petri nets, which are a discrete event-based modeling approach for the system. The main target is to separate the functionality of the tasks from the inputs and outputs so that the tasks always have the same functionality independently. This type of programming is well known as functional programming.

The framework allows the user to run multiple functions (parallel computing) at the same time without worrying too much about the complexities going on underneath.

The framework offers:

1. Tasks: the functionality is defined inside them.
2. Events: an object event that can be used to trigger tasks periodically or just once, according to the user's implementation.

3. Channels: these are the buffer containers where data is stored inside to separate them from functionality and to transfer them between tasks when needed by the application.
4. Writer/reader: is used when a node wants to communicate with another node in the model. The writer/reader is a spatial type of task in a distributed tasking framework that allows the nodes to send data to each other.

The most important component for integrating and modeling the benchmark into ScOSA will be the distributed tasking framework, which will be explained in detail later. as a brief explanation of the other two components.

3.2 Benchmarking:

The definition of benchmarking was explained in section 2.1. However, the technical implementation of any benchmark will follow the flow chart diagram in Figure 3.2.

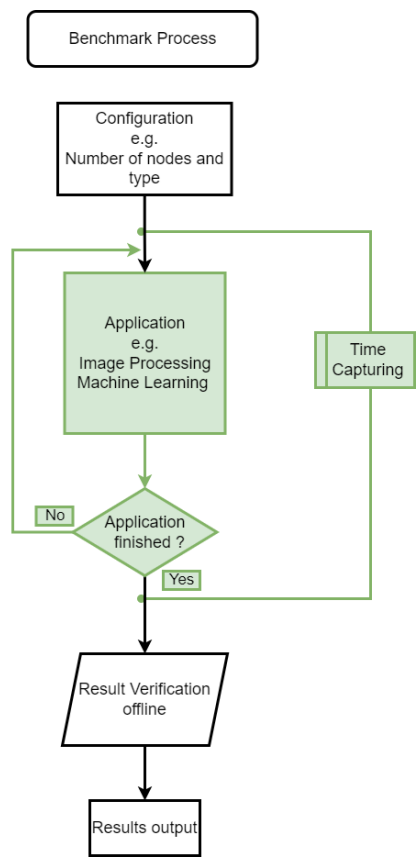


Figure 3.2: Benchmark Flow Chart

The benchmarking flowchart starts with the configuration of the system and then starts the benchmark workload; after that, check if the benchmark is finished or not. Time recording should be active while the benchmark is running. After the benchmark is finished, a verification step should be executed, but this step is not included in the benchmark time. Then we get the outputs; typically, the outputs are the time for execution, the rate of execution of operations used, etc...

3.3 Benchmark selection

From the literature review and the comparison Table 2.3, it was decided to proceed with implementing an application benchmark because it demonstrates the system's actual behavior in

real-life scenarios.

The LINPACK benchmark was chosen to assess the performance of a system with integer operations. Float-point operations are not exclusively used in space applications; they are used in any application, which is why it made more sense to use this benchmark as a first step toward evaluating the system since it is famous enough for benchmarking.

However, LINPACK is not enough to evaluate a distributed onboard computer for space avionics, so research work is done to investigate a benchmark suite, which is a collection of benchmarks for evaluating the system, and from the literature review, it was clear that there is a dedicated benchmark suite for onboard computers, which is the OBPMark 2.6.

3.4 Hardware

The current hardware that runs our benchmark is shown in Figure 3.3. The hardware used is the one highlighted in red; it is a Xilinx Zynq-7000 ARM Cortex-A9 [42], which consists of three CPU units (the three high-performance nodes).

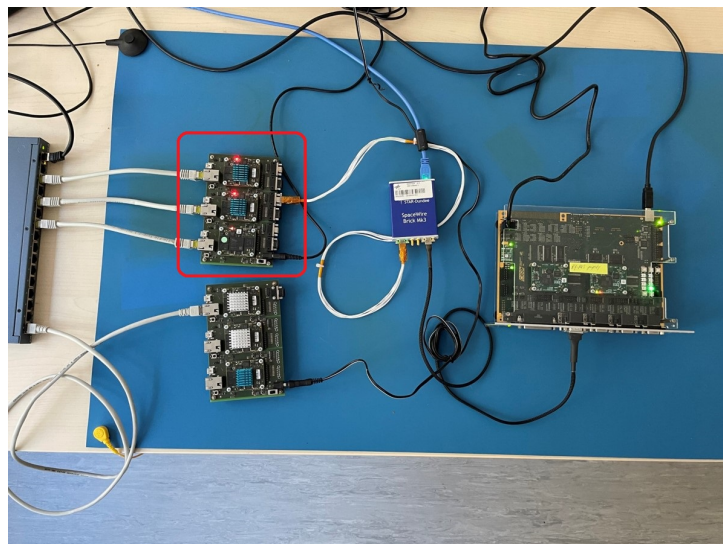


Figure 3.3: ScOSA HPN Software Development Model

3.5 Benchmark Implementation

For the distributed tasking framework, the programming sequence can be modeled in diagrams. In the upcoming sections, you can find the diagrams implementation diagrams of LINPACK and OBPMark integrated within ScOSA.

3.6 LINPACK Implementation

In the first attempt to evaluate our system, we started as simply as possible. The entire setup is running on one HPN node. An Event is triggering a task of LINPACK to start the LINPACK benchmark. The LINPACK Task starts by calculating matrix equations of size $N=1000$ for a linear algebra problem. Then a result of Mflops rate is presented at the end of the execution.

3.6.1 LINPACK One Node

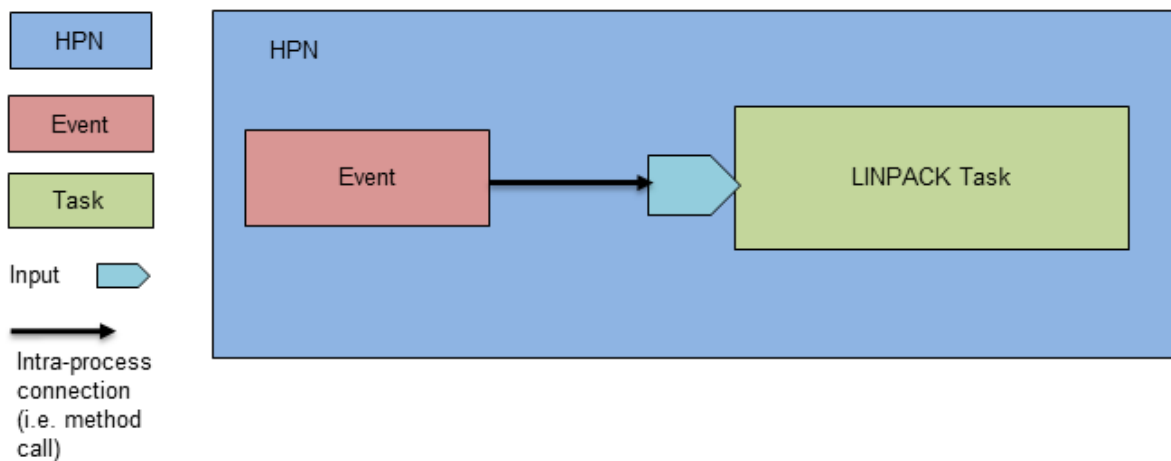


Figure 3.4: LINPACK one node setup

3.6.2 LINPACK three Nodes

After running a one-node setup, LINPACK is scaled into multiple nodes to evaluate the distributed system.

The next setup shows the same process of the LINPACK benchmark but in three different nodes,

and one of those nodes is collecting the results of Mflops rate for the system to investigate how many floating points the three CPU processing units can handle.

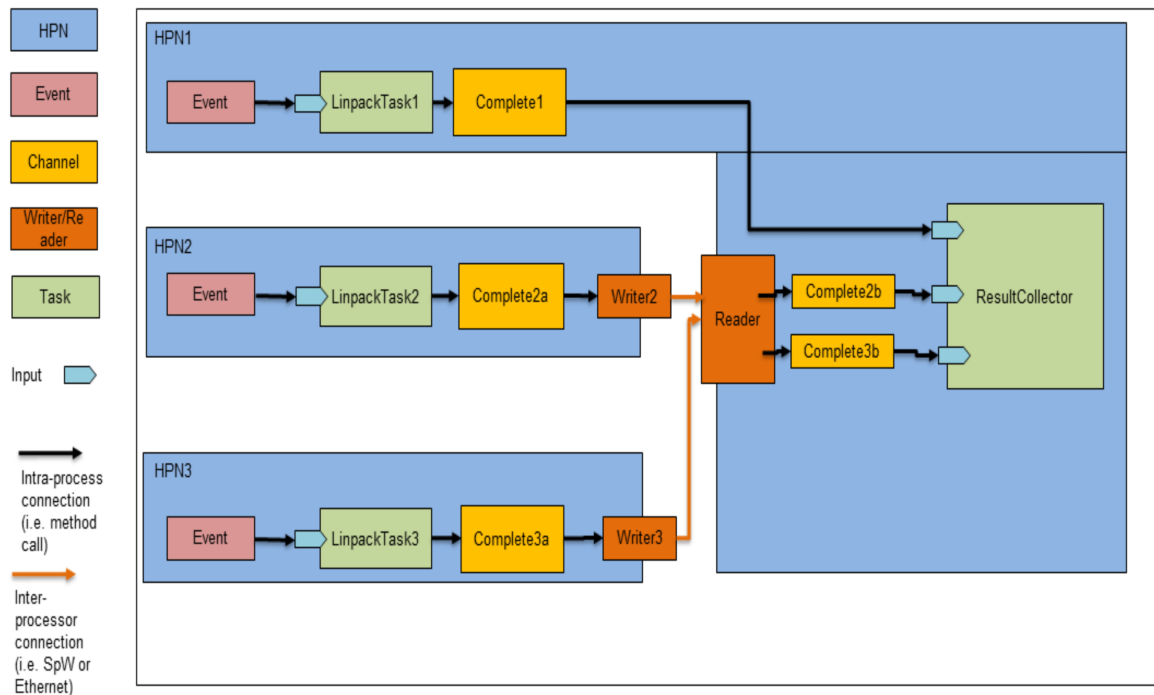


Figure 3.5: LINPACK three node setup

Figure 3.5 shows the three tasks of LINPACK in three different High-performance node (HPN)s , and there are extra three channels to hold the results of Mflops so that the result collector task can calculate the sum of the three nodes.

Also, there are a writer/reader tasks that helps the other two nodes to write into the first HPN and facilitate communication between them.

3.7 OBPMark in Details

The OBPMark, as explained in section 2.6 was developed through a collaboration between ESA and the Barcelona Supercomputing Center; currently, it is an open-source beta version.

I have been in contact with the ESA team to give more information about the implementation and to provide documentation if possible. They provided technical notes [3] and gave access to their internal repository so that I could see the latest version of OBPMark before releasing it to the public.

The first benchmark in this suite was the image processing benchmark. The technical notes are clear enough to help with the implementation of the image processing benchmark with ScOSA. The OBPMark-image processing is based on typical image processing tasks for onboard computers, such as deep-space telescopes with long exposure times.

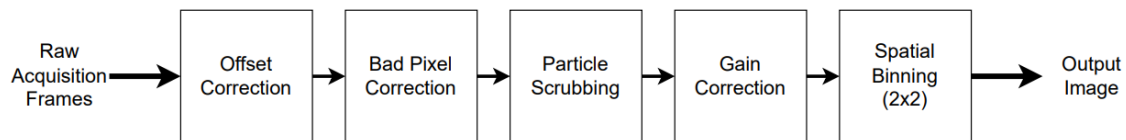


Figure 3.6: OBPMark Image-processing pipeline [3]

Figure 3.6 is the pipeline implementation of the Image Processing benchmark, where the input for the pipeline is a set of frames (8 images) and the output is only one frame processed from all of the inputs. For every input frame, the steps of the pipeline are executed, and they are governed by the equations from 3.1 to 3.6:

Where,

I : is the current input pixel.

J : is the output pixel.

x, y : are the position of the pixel inside the frame.

Image offset correction

In the image offset correction step, a constant offset value, $C_{x,y}$, is set to each input pixel and subtracted from the input pixel if the input is greater than the offset. These offset values are

predefined by the benchmark as auxiliary data.

$$J_{x,y} = I_{x,y} - C_{x,y}, \quad (3.1)$$

Bad pixel correction

The bad pixel correction is performed by determining whether the pixel is bad or not using a configurable look-up table $M_{x,y}$. If it is not a bad pixel, then the output is the same as the input. However, if there is no bad, the output pixel is a function of F .

F function is the average value of the **3x3** mask pixels' neighboring good pixels. If a bad pixel is found in the corner, the algorithm takes the average of **2x2**. In the case of edges, the mask of neighboring pixels is **2x3** for the left/right edge and **3x2** for the top/bottom edge.

$$M_{x,y} \begin{cases} 1 & , J_{x,y} = F(I_{x,y}, x, y) \\ 0 & , J_{x,y} = I_{x,y} \end{cases} \quad (3.2)$$

Radiation scrubbing

The previous steps were one-to-one operations, meaning that the operation was performed for every pixel inside every frame based on auxiliary data predefined by the benchmark. However, for radiation scrubbing, the operation depends on the previous two and the next two frames regarding the time of capturing the frames. For the 8 frames from $t = 0$ to $t = 7$, the benchmark already provides an additional 4 frames ($t = -2, t = -1, t = 8, t = 9$) as auxiliary data. Then the average value of those pixels is calculated by the equation below and used if the current pixel frame suffers a radiation disturbance.

$$J_{x,y} = \frac{I_{(x,y),t-2} + I_{(x,y),t-1} + I_{(x,y),t+1} + I_{(x,y),t+2}}{4} \quad (3.3)$$

Gain correction

Gain correction is simply a multiplication of a constant value G .

$$J_{x,y} = I_{x,y} \cdot G_{x,y} \quad (3.4)$$

Spatial binning

Every 2×2 pixels within each frame is added together to form only one pixel, meaning that each frame will shrink in size to a quarter of the original one.

$$J_{x,y} = I_{x,y} + I_{x+1,y} + I_{x,y+1} + I_{x+1,y+1} \quad (3.5)$$

Temporal binning

The output frame in the final step is the sum of every pixel in the 8 frames set (pixel by pixel in order of their positions).

$$\sum_{t=0}^7 f_{i,t} \quad (3.6)$$

3.7.1 OBPMARK One Node

As with LINPACK, the same strategy of implementing the benchmark within ScOSA was used. First, we have only one node that contains the event and task; the task runs OBPMARK image processing benchmark function.

The event triggers the benchmark task to start, and after a while, the result output frame is constructed.

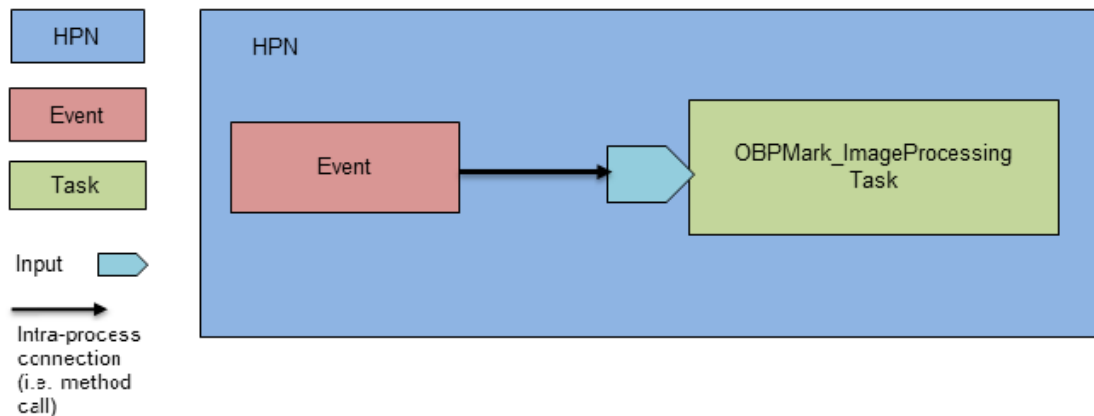


Figure 3.7: OBPMARK one node setup

3.7.2 Benchmark Distribution

The next step is to benchmark the distributed system and scale the benchmark to different nodes. However, unlike LINPACK, simply repeating the same task in the other nodes does not make sense this time. LINPACK just gives an idea of the Mflops that every node can process, but in image processing, this is not the actual behavior of an application on the onboard computers.

The idea of splitting the input frames between the different nodes inside the system and keeping the pipeline as it is without splitting it because there are dependencies between steps was the first that came to mind.

Splitting is done in such a way that every node takes a sub-frame, as shown in figure 3.8 below, and performs the same pipeline steps on every sub-frame.

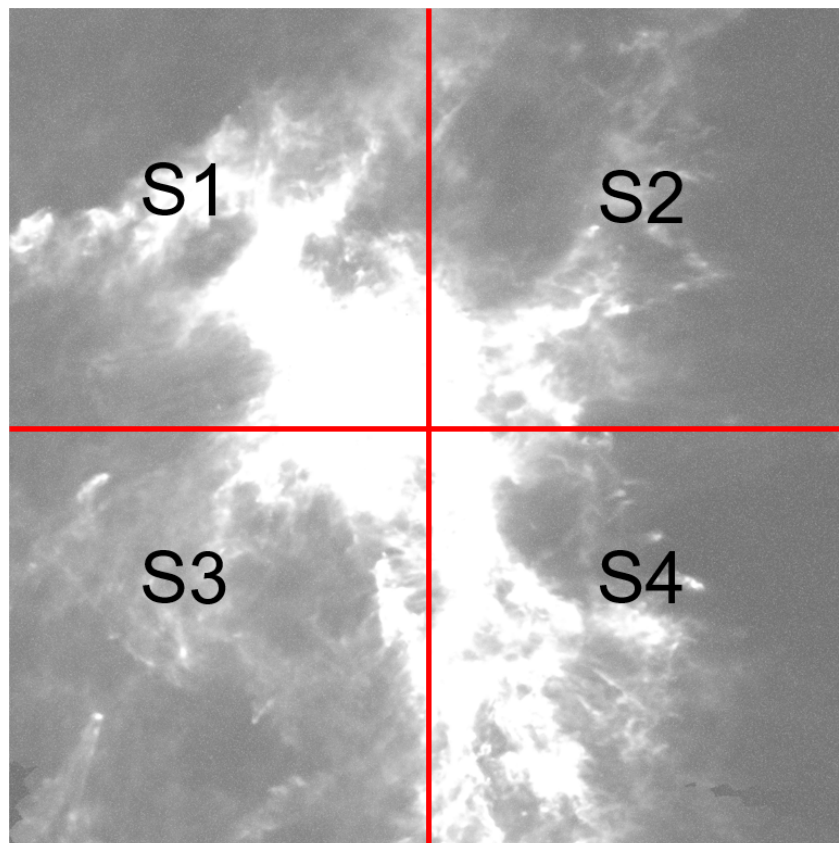


Figure 3.8: Distribution into four quarter sub-frames

However, after reviewing the implementation of OBPMark and knowing that the input frames are all in the form of binary format files (*.bin*) in one-dimensional data representation (the pixels are all next to each other and not in the form of a 2-D array), It was prudent to divide the image only by its width or height so that we reduce the complexity of splitting it into one dimension.

From equation 3.2 of bad pixel correction if we just split the frames without taking care of the new edges and corners constructed, the output will not be the same as the original benchmark. Because the F function handles the edges and corners in a different way than the rest of the pixels inside the frame, If there was a bad pixel at the new constructed edge in the original benchmark, it requires information on the neighboring pixels, which would be in other sub-frame.

Thus, the solution was splitting the sub-frames considering an overlapping area as shown splitting algorithm 1.

Also, the spatial binning in equation 3.5 equation shows that for every **2x2** pixels block, it will shrink in size into one pixel, which means if we have two straight lines of pixels in a 2-D representation of the image for width, they will be reduced to only one straight line of pixels. This also means that when splitting, we must ensure that every two straight-line ordered pixels are on the same sub-frame so that the spatial binning process works properly as explained in splitting algorithm 1.

Algorithm 1 Splitting

```

1:  $s = height / n_{divisions}$ 
2:  $k_{up} = 2$  # Overlap rows up
3:  $k_{down} = 2$  # Overlap rows down
4: for  $i$  in  $n_{divisions}$  do
5:   if  $i == 0$  then
6:      $k_{up} = 0$ 
7:   else if  $i == n_{divisions} - 1$  then
8:      $k_{down} = 0$ 
9:   end if
10:   $F_i = I[s * i - k_{up} : (s + 1) * i + k_{down}][width]$ 
11: end for

```

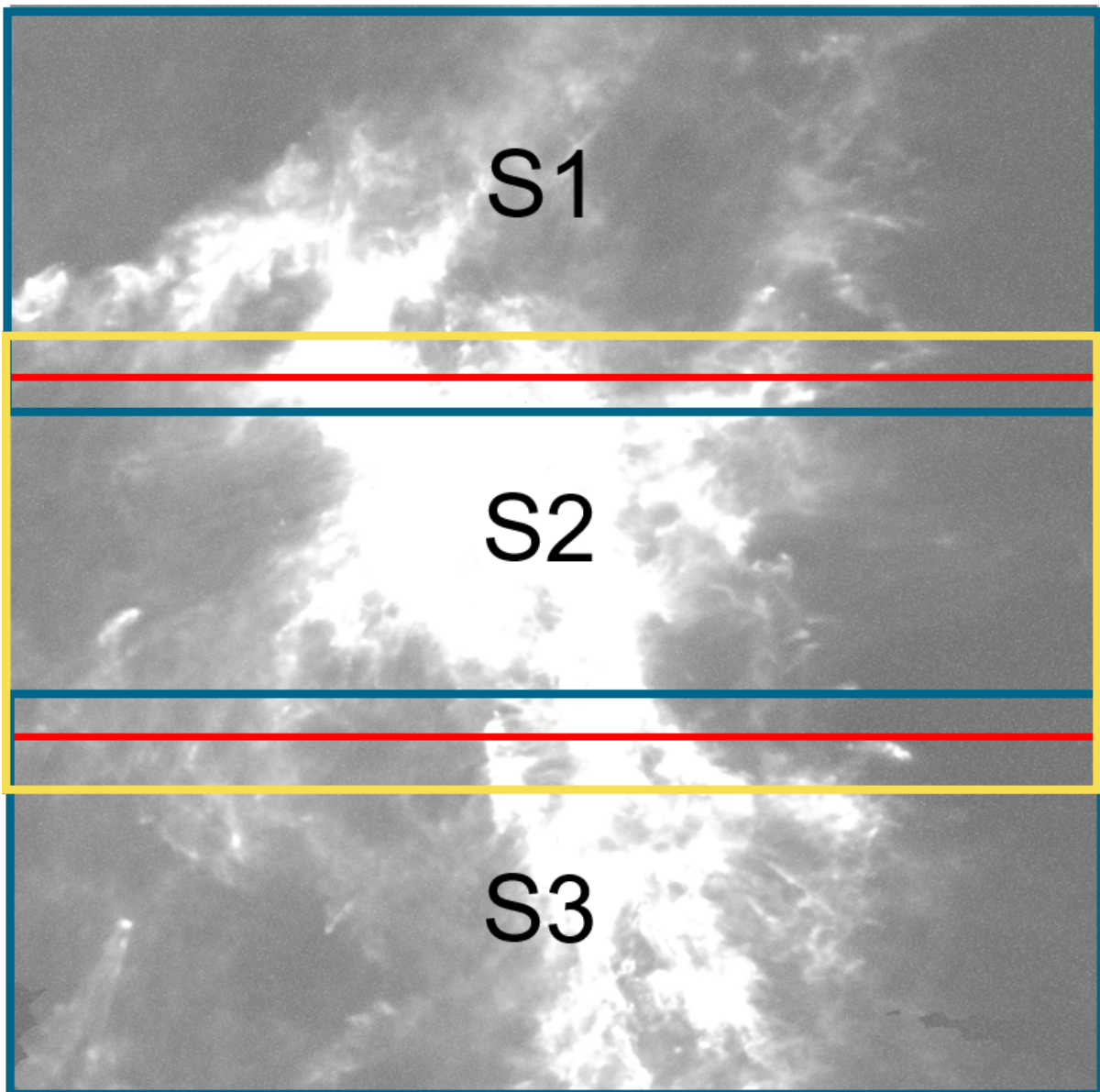


Figure 3.9: Overlapping Distribution

Figure 3.9 shows the distribution of the image frame into three sub-frames. The red lines represent the fair split between the nodes without overlapping, then the algorithm creates an overlapping area for every sub-frame, so the top blue rectangle represents the first sub-frame and the bottom blue rectangle represents the last sub-frame. The yellow rectangle represents the middle sub-frame. As shown, the middle sub-frame holds an overlap area from the previous and

next sub-frames to help in handling the pipeline steps in the correct way. This is a simplified illustrative example of how only one frame can be split between three nodes; however, the algorithm can handle more than three nodes with the constraint that the number of nodes is less than the height size of the frame.

3.7.3 Modeling Approach for Distribution

After splitting the frames into sub-frames, the next step is to design the benchmark with a distributed tasking framework diagram. Modeling the the benchmark distribution/parallelization was done using the MapReduce approach. MapReduce is a programming model that is based on functional programming. It was first introduced for parallelization of big data between a cluster of computers. However, it adds overhead to sending data through nodes [43]. In our case, the data would be the sub-frame slices from all input frames.

3.7.4 OBPMark three Node

Distribution strategy,

1. Split: in the split step the data are loaded from the file system and then divided into three sub-frames.
2. Serialize data: is to convert data complex objects into a structure of data stream of bytes.
3. Push the message: this is to start transmitting this stream of bytes either by the network or internally in the same node.
4. Receive message: to accept the data transmission.
5. Deserialize data: to return it back to the same complex object form.
6. Continue the benchmark process.

Distributed Diagram of OBPMark within ScOSA

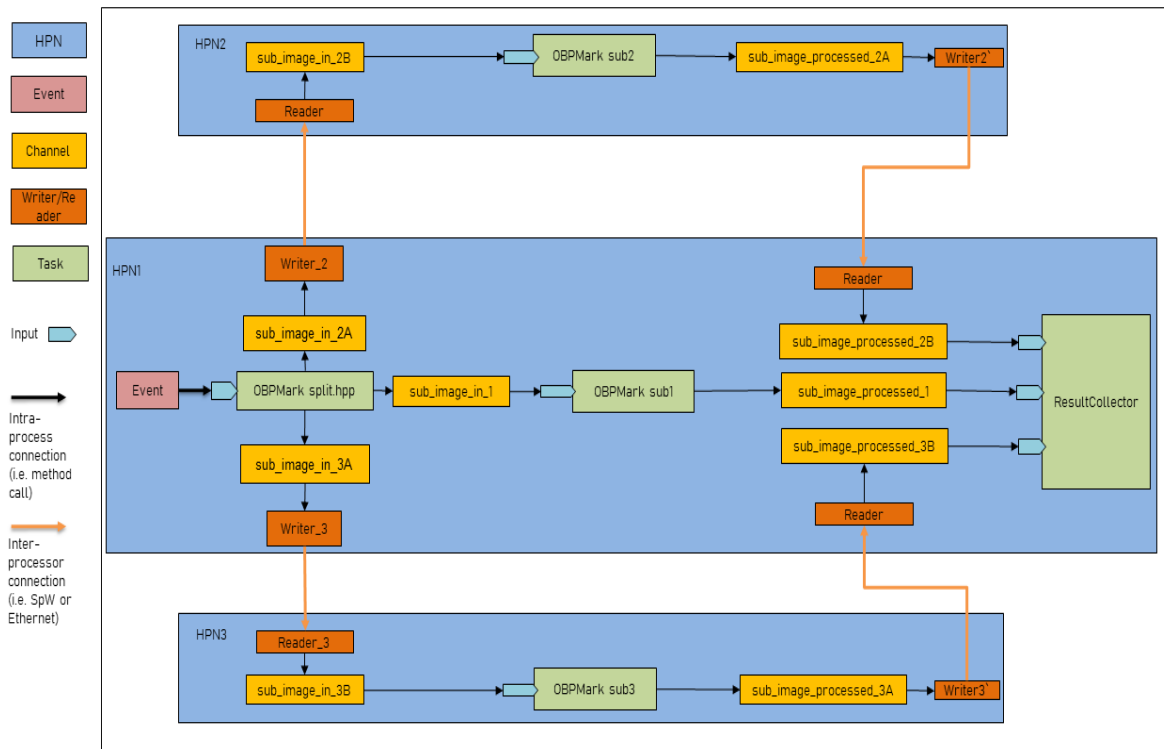


Figure 3.10: OBPMARK three node setup

Figure 3.10 shows the implementation of distributed OBPMARK. The diagram consists of three nodes of HPN nodes.

The split task sends buffer data to other OBPMARK-sub-task which in return performs the image processing benchmark. Then it sends the outputs via buffers to the result collector task which in return constructs the *output.bin* frame. The time capture is executed from the start of the benchmark until the output frame is constructed.

The messages that had been used are:

Listing 3.1: Structures used for data serialization with a message size of 12185642 bytes approximately 12 MiB.

```

1
2 constexpr size_t n_nodes = 3; // Number of nodes
3 #define MSG_SIZE 350 * 1024 // Height*Width
4 #define MSG_SIZE_HALF 175 * 1024 // Height*Width

```

```
5 #define NUM_FRSME      12          // Fixed
6 // struct for the Serialize data
7 typedef struct
8 {
9     uint16_t frame[MSG_SIZE] = {0};
10 } frame_msg;
11 struct image_data_msg
12 {
13     unsigned int num_frames = {0};
14     frame_msg frames[NUM_FRSME];
15     uint16_t offsets[MSG_SIZE] = {0};
16     uint16_t gains[MSG_SIZE] = {0};
17     uint8_t bad_pixels[MSG_SIZE] = {0};
18     uint8_t scrub_mask[MSG_SIZE] = {0};
19     uint32_t binned_frame[MSG_SIZE_HALF] = {0};
20     uint32_t image_output[MSG_SIZE_HALF] = {0};
21 };
22 /**
23  * \brief struct to send data msg.
24  */
25 struct msg
26 {
27     image_data_msg image_msg;
28     unsigned int w_size = {0};
29     unsigned int h_size = {0};
30     unsigned int h_overlap_array[n_nodes] = {0};
31     unsigned int h_output_array[n_nodes] = {0};
32     std::chrono::time_point<std::chrono::_V2::system_clock,
33                             std::chrono::_V2::system_clock::duration>
34         starttime;
35 };
```

As shown in the message structure, there is a time point that holds the starting time of sending data so that the data transferring latency is calculated on the receiving node.

Chapter 4

Results and Discussion

The results are presented in two categories: the LINPACK results and the OBPMARK-Image Processing benchmark. Each category is investigated in two scenarios: operating the benchmark program using one node or three nodes. The output results are the average of three different runs on the machine used to develop the benchmark (Team server x86_64) and the Zynq7000. The detailed specifications are in Table 4.1 and the detailed results are available in the Appendix.

Table 4.1: Hardware Specification

	Model name	Architecture	CPU Frequency	RAM size
Team server	Intel(R) Core(TM) i9-10980XE CPU	x86_64	3 GHz	125 GB
One Node Zynq-7000	ARM Cortex-A9 Based	ARMv7 Processor rev 0 (v7l)	886 MHz	1 GB

4.1 LINPACK

Table 4.2 shows the results of the floating-point operation that the nodes perform. In the case of the machine used to develop and integrate the benchmark with ScOSA (the Team server), the nodes are treated as the same CPU but with a different internal core. However, in the case of the Zynq-7000 HPN nodes, it is separated into three CPU units. In addition, the Team server CPU has more powerful processing cores than the Zynq nodes, as evidenced by the results'

Table 4.2: LINPACK Results

		Team server x86_64 Mflops	Zynq-7000 Cortex A9 Mflops
Single precision	One Node	4206.863	75.022
	Three Nodes	11064.266	300.109
Double precision	One Node	3225.960	46.638
	Three Nodes	8062.116	190.032
Matrix order N = 1000			

megaflops rate. However, the Zynq nodes are the ones that are going to fly into space, but the representation of the Team server results is for making sure that the results make sense and to help the engineers and developers improve the tools that are used in performance evaluation.

The results of the LINPACK single-precision representation have the highest megaflops so far when compared to the double-precision representation. This was expected because the 32-bit LINPACK data representation uses less memory than the 64-bit LINPACK data representation, and this led to fewer instructions when fetching data from 32-bit. When the results of one node are compared to those of three nodes, the result is nearly three times as large, which is also justified in the implementation because it is the same benchmark task that runs on three different nodes. However, the time consumption did not improve by using three nodes compared to one node because it performed the same operation on all three nodes. In normal distributed system operation, if a task takes a long time, the processing nodes shear the same task to finish it faster. but that was not the case with this LINPACK setup.

LINPACK was only the beginning of getting the distributed system to work and learning more about how to find the best strategy to benchmark ScOSA. It also worked very well in giving an idea of how much the Zynq-7000 processors can perform in float-point operations; however, for space applications and distributed systems, LINPACK is not enough.

4.2 OBPMark

4.2.1 OBPMark One Node

In the first scenario, the OBPMark-Image Processing benchmark runs only at one node, and the event is triggering the benchmark to start loading the 8 frames and all auxiliary data from the file system. Then the process feeds the image processing pipeline a total of 8 frames with a size of 1024 x 1024 pixels each.

Table 4.3: OBPMark One Node results

Number of processed pixels (frames*Height*Width) = 8*1024*1024	Time in ms
OBPMark_ImageProcessing Team server x86_64 total time	373.639
OBPMark_ImageProcessing Zynq-7000 Cortex A9 total time	8659.616

The total processing time for one node in the Team server is 0.373 seconds, which is significantly faster in comparison to the 8.659 seconds for one HPN node. However, that result is not a surprise; it is expected that the Team server machine will be fast. To calculate the megapixels per second (Mpixels/s) that the CPU units perform, we need simply the number of processed pixels, which is the total number of frames multiplied by the number of pixels on each frame, which is 8x1024x1024, and then divided by the total processing/execution time of the benchmark.

Team server:

$$\frac{8 * 1024 * 1024 * 10^{-6}}{0.373} = 22.489 \text{ Mpixels/s} \quad (4.1)$$

One node Zynq-7000:

$$\frac{8 * 1024 * 1024 * 10^{-6}}{8.659} = 0.968 \text{ Mpixels/s} \quad (4.2)$$

4.2.2 OBPMark Three Nodes

OBPMark-Image Processing Multiple Node Configuration differs from LINPACK; in OBPMark, it is an actual distribution of data for the same task between different nodes to benchmark

the distributed system in resemblance to real-life scenarios.

The distribution is done simply by splitting the eight frames and the auxiliary data into sub-frames based on the number of nodes available on the system. This setup sends big buffers of data between the nodes. In the OBPMark shown in Figure 3.10 there are three channel buffers connected to the split task. Those buffers carry messages of size 12 MiB approximately each. On the other side, there are three other channels (BuffersProcessed) that carry messages of size 0.683 MiB (the output subframes) to be collected with the result collector task. With all of those buffers, there is an overhead in sending data between tasks and nodes. This overhead is shown in the latencies of those buffers. Table 4.4 shows the results of the distributed image processing application benchmark and the latencies of data transmission.

Table 4.4: OBPMark distributed results

	Team server x86.64 Time in ms	Zynq-7000 Cortex A9 Time in ms
latency of BufferInput 1	2.621	53.683
latency of BufferInput 2	1970.293	1877.117
latency of BufferInput 3	1972	1624.984
BufferInput size = 12185644 bytes		
latency of BufferProcessed 1	2099.646	2201.562
latency of BufferProcessed 2	121.462	597.561
latency of BufferProcessed 3	122.294	831.292
BufferProcessed size = 716832 bytes		
Split task time	72.447	681.105
Sub-frames 1 processe time	126.089	2794.341
Sub-frames 2 processe time	133.144	2464.051
Sub-frames 3 processe time	126.664	2379.930
ResultCollector task time	4.944	41.175
Number of processed pixels (frames*Height*Width) = 8*1024*1024		
Total Execution time	2291	5453.666

The latency of data transmission on the Team server is approximately two seconds for twelve MiB, which is about 6 MiB/sec of transmission rate between the nodes using the network layer

of ScOSA. Also, this value is nearly the same for data transmission between the Zynq-7000 nodes. The BufferInput 1 is on the same node, so the transmission is internal and not using the communication protocol, and the transmission time is low. By looking at the BufferProcessed 1 time, you might think that it should also be the lowest since it is on the same node as well, but that is not the case, and there is a good explanation for that: The result collector is waiting for the other two nodes to send their data so that the result collector task can start.

4.2.3 Team server x86_64 Results Discussion

By analyzing the results of the Team server in Table 4.4, it is clear that the total execution time in comparison to one node scenario is much bigger. However, in the Zynq-7000 nodes, the overall execution time is reduced from 8.659 to 5.453 seconds.

Figure 4.1 shows event, time diagram, provides a good explanation for the behavior of the team-server version. From Table 4.1, Team server is an Intel x86_64 CPU with significantly more processing power than the application itself needs. The buffers latency in comparison to the time that every task takes to process those transmitted data buffers is too small which is why the overall time increases.

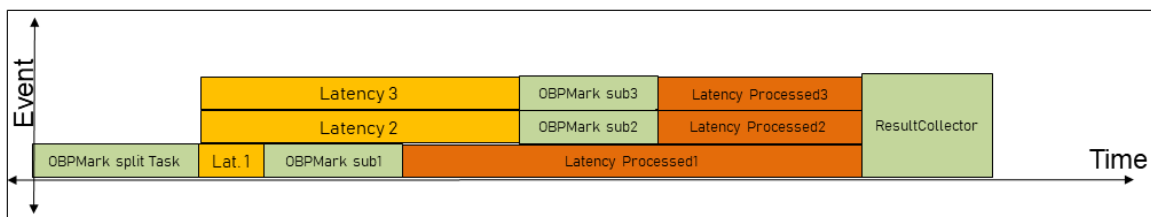


Figure 4.1: OBPMark three node Event, Time Diagram Team-server

The latency of processed buffer 1 does not depend on the size of the buffer since it is already

an internal transmission, but it depends more on the other two nodes and how they are going to send the data to the result collector. Also, the setup uses the network software layer for reader and writer tasks to transmit data. As it is clear, OBPMark subframe 1 is finished earlier than the other two subframes on the other nodes. The time synchronization of the nodes is accurate since it is the same CPU unit, which is why latency two and three are almost the same.

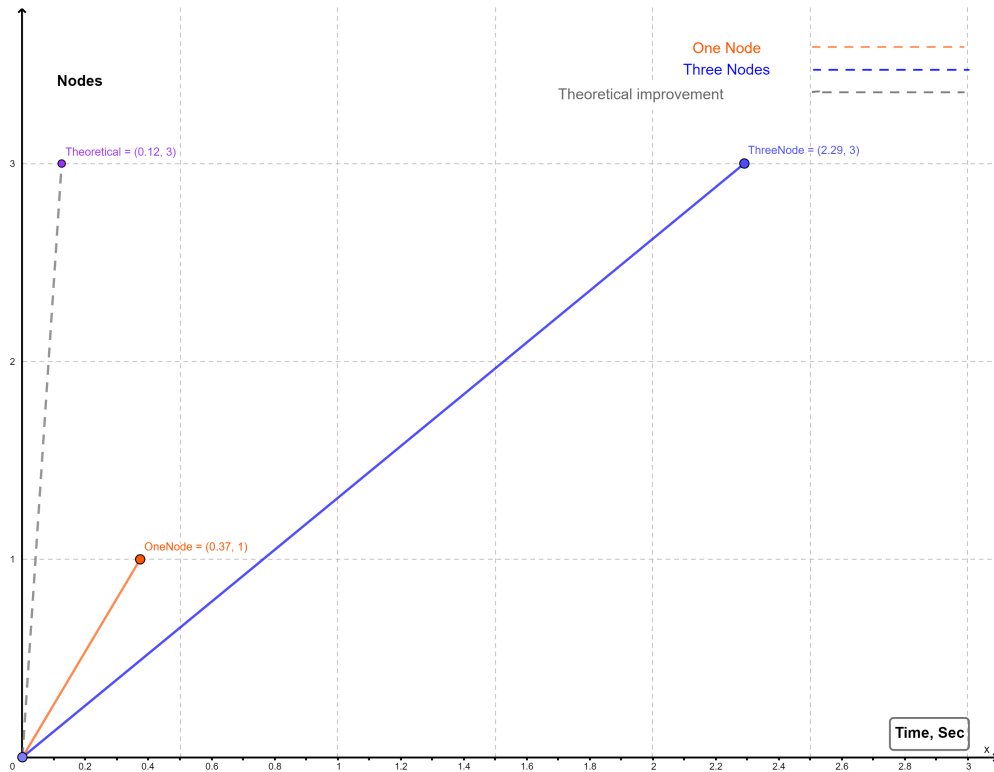


Figure 4.2: OBPMark Team server Performance

In Figure 4.4 the theoretical performance is simply the performance of the benchmark on one node before distribution divided by the new number of nodes the ScOSA will use. This theoretical performance is not attainable in distributed systems because of the overhead of data transmission and the configuration of data splitting. However, the closer we move to the theoretical, the better the overall performance. The total execution time of the Team server is dominated by the latency rather than the processing time, which increased the overall execution time. From the Figure 4.4, it is clear that the three nodes line are moving away from the

theoretical improvement line even farther than the one-node implementation.

4.2.4 Zynq-7000 Results Discussion

In the case of the three target nodes, Zynq-7000, the latency is not greater than the processing time, which results in improving the overall performance and reduces the execution time for the image processing benchmark from approximately 8.6 seconds to 5.4 seconds, including configuration and data transmission time.

Figure 4.3 shows that the tasks took longer in comparison to the latencies. The latencies at nodes two and three should have been nearly the same, but I faced the challenge of synchronizing the time at the three target nodes since they are not connected to the internet because of security issues. Nevertheless, I synchronized all of them manually, which is why there is a small milliseconds difference in latency three compared to latency two.

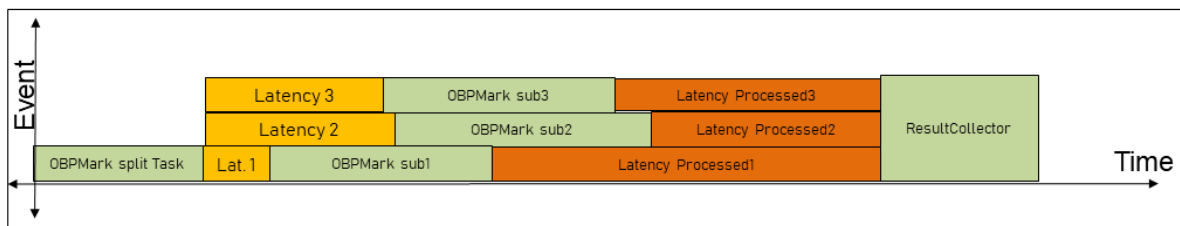


Figure 4.3: OBPMark three node Event, Time Diagram HPNs

Figure 4.4 shows the increase in performance from one node to three nodes, which clearly shows that the performance moved closer to the theoretical line of performance. As previously stated, this is accomplished by dividing the node time by the total number of nodes in the system.

When analyzing the results from the calculation of processed pixels per second,

Three nodes Zynq-7000:

$$\frac{8 * 1024 * 1024 * 10^{-6}}{5.45} = 1.538 \text{ Mpixels/s} \quad (4.3)$$

Which improved the performance from 0.968 Mpixels/s.

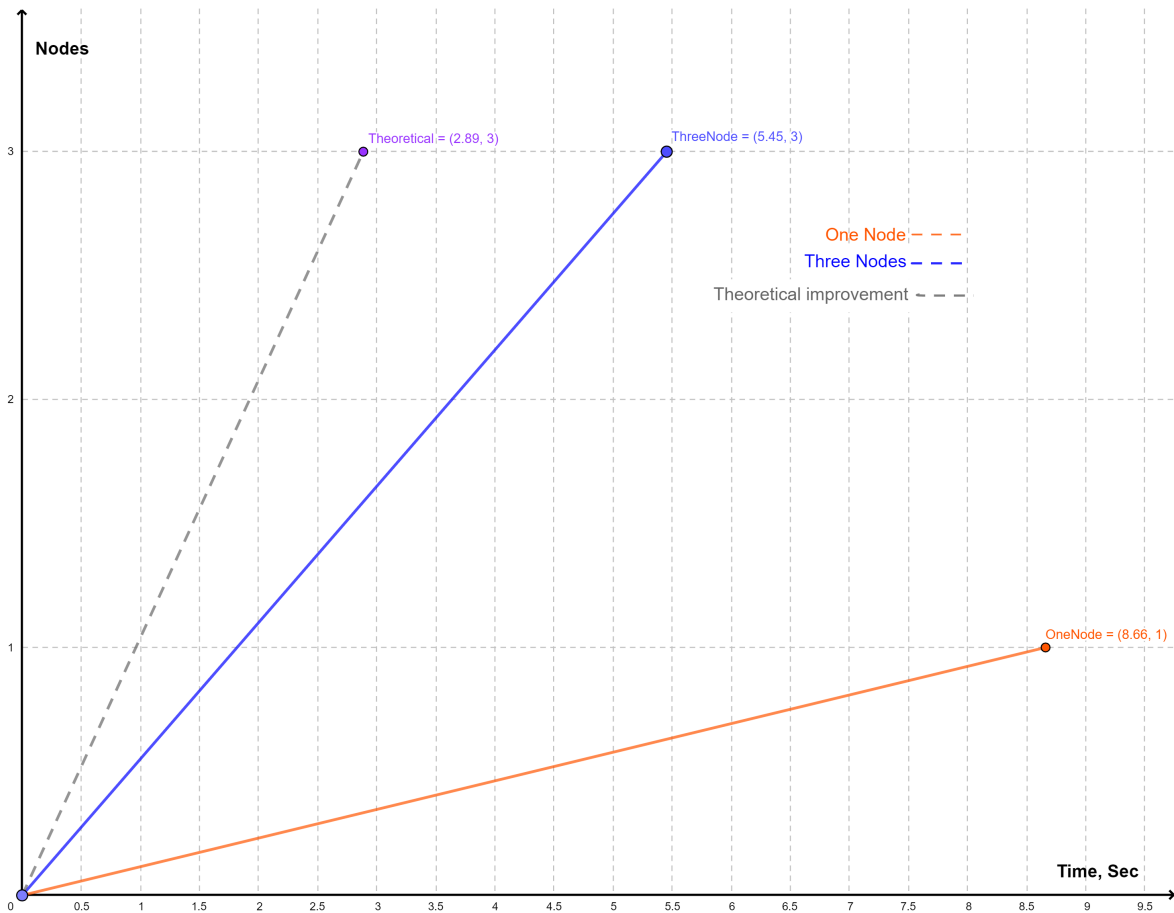


Figure 4.4: OBPMARK HPNs Performance

4.3 Verification

All the results of OBPMARK have been verified by comparing the output frame with the verification frame provided by the benchmark. I used the Linux utilities `diff` and `cmp` to compare those two files together, and they are identical.

Chapter 5

Conclusion

This thesis focused on selecting embedded benchmarks for the performance evaluation of on-board computers, specifically the ScOSA architecture for space applications, and possible ways to standardize the benchmarks in the space domain.

The first requirement of the thesis is fulfilled by selecting the LINPACK benchmark and OBPMARK, which are suitable benchmarks for evaluating the onboard computer in the space domain.

In response to the scientific question of which benchmarks are suitable for ScOSA, the LINPACK benchmark and OBPMARK have been utilized to evaluate the performance of ScOSA in a one-node CPU implementation for the development machine and target HPN nodes in different scenarios. The selection of LINPACK is based on its scientific popularity for benchmarking performance, which makes the results comparable with those of other systems. OBPMARK is based on compatibility with ScOSA and relevance to space applications, especially onboard processing.

The LINPACK benchmark was not internally distributed between the nodes since the output measuring unit is the float point operation. Instead, the same benchmark is repeated in all other nodes, and the result collector task is in charge of collecting the Mflops that each node is

capable of performing. The results show an increase in the flop rate almost three times when using three nodes, either in the development machine or in the target HPN nodes, which was expected.

In conclusion, LINPACK gives an idea of how many floating point operations that processing unit can perform, and it is well known in scientific communities. The results of LINPACK can also be compared easily with those of other processing units, but floating point operations are not necessarily the case for applications in the space domain, which is why there was a need for a space-specific benchmark (e.g., OBPMark).

OBPMark-Image Processing Benchmark distribution is done in a way to simulate the behavior of actual onboard applications by making the multiple nodes share the same big task to reduce execution time. The distribution divides the input frames into subframes, with each node processing one of them.

OBPMark execution time in one node at the Team server machine is the fastest execution time but when distributing for different nodes in the team server the execution time increased and the reason for that is the high transmission latency. The time is dominated by high latencies.

For the HPN nodes, OBPMark's distributed three-node setup reduces execution time and improves performance, resulting in a performance increase from 0.968 megapixels/s using one node to 1.538 megapixels/s using three nodes. The improvement is due to the small latency period in comparison to processing tasks. From the results, the scientific question of how much processing power we have and how many nodes to add to reach the required performance is answered for the scaling of the distributed system to a three-node implementation.

In general, if the system transmission latency rate is higher than the processing time for the application, it is not recommended to distribute the application, but if the transmission latency is lower than the processing time, it is recommended to distribute the application in the most efficient way the system designer sees fit.

Chapter 6

Future Work

For future work, I would recommend investigating scaling the benchmark to more than three nodes. An attempt to do so in this thesis work was an initial plan, but there was a challenge in sending big buffers between the nodes, and the thesis is constrained to only six months of work. The challenges encountered and implementation issues were reported to both engineers and developers in our team and ESA developers regarding issues with OBPMark, especially since it's a beta version.

I would also recommend investigating the rest of the benchmark suite of OBPMark by applying it to ScOSA. The benchmark suite is not fully completed yet. There are still some benchmarks inside that the developers are working on standardizing that would be worth implementing for ScOSA once released, especially the machine learning one.

I would recommend investigating other ways of distributing benchmarks. In OBPMark-Image processing, for example, instead of splitting the processed frames, we can split the pipeline and keep the frames intact. Theoretically, this will take a long time because the buffers will transfer all of the frames from one node to another, but it may be worth investigating in the future for the sake of benchmarking the system's reliability when sending big buffers.

Chapter 7

Appendix

Table 7.1: LINPACK One Node development machine detailed three-run results

LINPACK One Node	Run1 Mflops	Run2 Mflops	Run3 Mflops	Average Mflops
TeamServer x86_64 Single precision	4099.73	4482.46	4038.4	4206.863333
TeamServer x86_64 Double precision	3163.95	3376.73	3137.2	3225.96
Matrix order N = 1000				

Table 7.2: LINPACK One HPN Node detailed three-run results

LINPACK One Node	Run1 Mflops	Run2 Mflops	Run3 Mflops	Average Mflops
Zynq-7000 Cortex A9_Single precision	75.4387	74.3746	75.255	75.02276667
Zynq-7000 Cortex A9_Double precision	46.8493	46.6236	46.4432	46.6387
Matrix order N = 1000				

Table 7.3: LINPACK Three Nodes development machine detailed three-run results

LINPACK Three Nodes	Run1 Mflops L64	Run2 Mflops L65	Run3 Mflops L66	Average Mflops
TeamServer x86_64 Single precision	11078.6	11064.4	11049.8	11064.26667
TeamServer x86_64 Double precision	8036.29	8056.22	8093.84	8062.116667
Matrix order N = 1000				

Appendix

Table 7.4: LINPACK Three HPN Nodes detailed three-run results

LINPACK Three Nodes	Run1 Mflops L64	Run2 Mflops L65	Run3 Mflops L66	Average Mflops
Zynq-7000 Cortex A9_Single precision	303.075	300.153	297.101	300.1096667
Zynq-7000 Cortex A9_Double precision	192.423	190.01	187.663	190.032
Matrix order N = 1000				

Table 7.5: OBPMARK one Node results detailed three-run results

Number of processed pixels (frames*Height*Width) = 8*1024*1024	Run1 Time in ms	Run2 Time in ms	Run3 Time in ms	Average Time in ms
OBPMARK_ImageProcessing Team server x86_64 total time	374.693	365.542	380.684	373.6396667
OBPMARK_ImageProcessing Zynq-7000 Cortex A9 total time	8711.93	8632.68	8634.24	8659.616667

Table 7.6: OBPMARK distributed results for the development machine three runs

Team server x86_64	Run1 Time in ms	Run2 Time in ms	Run3 Time in ms	Average Time in ms
latency of BufferInput 1	2.360106	3.111839	2.393961	2.621968667
latency of BufferInput 2	1973.241091	1974.763155	1962.877035	1970.29376
latency of BufferInput 3	1978.310823	1976.907969	1960.783958	1972.000917
BufferInput size	12185648 bytes			
latency of BufferProcessed 1	2116.214037	2096.359968	2086.364031	2099.646012
latency of BufferProcessed 2	128.933907	117.656946	117.797136	121.462663
latency of BufferProcessed 3	117.508888	122.220993	127.15292	122.294267
BufferProcessed size	716832 bytes			
Split task time	64.95507	77.853424	74.533882	72.44745867
Sub-frames 1 processe time	116.593575	140.619644	121.05452	126.0892463
Sub-frames 2 processe time	130.692932	142.891983	125.848564	133.144493
Sub-frames 3 processe time	133.384796	131.616089	114.992393	126.664426
ResultCollector task time	4.436714	4.977554	5.420427	4.944898333
Number of processed pixels (frames*Height*Width)	8*1024*1024			
Total Execution time	2291	2306	2276	2291

Appendix

Table 7.7: OBPMark distributed results for the HPNs three runs

Zynq-7000 Cortex A9	Run1 Time in ms	Run2 Time in ms	Run3 Time in ms	Average Time in ms
latency of BufferInput 1	53.679943	53.457975	53.913832	53.68391667
latency of BufferInput 2	1880.186081	1880.32198	1870.844126	1877.117396
latency of BufferInput 3	1629.528999	1670.710087	1574.714184	1624.984423
BufferInput size	12185644 Bytes			
latency of BufferProcessed 1	2198.611975	2301.90897	2104.165077	2201.562007
latency of BufferProcessed 2	574.913025	660.80308	556.968212	597.561439
latency of BufferProcessed 3	821.245193	823.175907	849.457026	831.2927087
BufferProcessed size	716828 Bytes			
Split task time	664.123962	729.614929	649.576599	681.1051633
Sub-frames 1 processe time	2777.65332	2759.599365	2845.773193	2794.341959
Sub-frames 2 processe time	2463.584717	2464.05249	2464.516846	2464.051351
Sub-frames 3 processe time	2379.549561	2380.968262	2379.272217	2379.930013
ResultCollector task time	41.151066	41.399277	40.97683	41.17572433
Number of processed pixels (frames*Height*Width)	8*1024*1024			
Total Execution time	5433	5521	5407	5453.666667

Bibliography

- [1] Andreas Lund, Zain Alabedin Haj Hammadeh, Patrick Kenny, Vishav Vishav, Andrii Kovalov, Hannes Watolla, Andreas Gerndt, and Daniel Lüdtkke. ScOSA system software: the reliable and scalable middleware for a heterogeneous and distributed on-board computer architecture. *CEAS Space Journal*, 14(1):161–171, January 2022.
- [2] The edn embedded microprocessor benchmark consortium eembc. <https://www.eembc.org/>.
- [3] Obpmark (on-board processing benchmarks). Technical Report V0.3-DRAFT, European Space Agency, OBPMARK@esa.int, August 2022.
- [4] David Steenari, Leonidas Kosmidis, Ivan Rodriguez-Ferrandez, Alvaro Jover-Alvarez, and Kyra Förster. OBPMARK (On-Board Processing Benchmarks) – Open Source Computational Performance Benchmarks for Space Applications. June 2021. Publisher: Zenodo Version Number: 1.0.
- [5] Daniel Lüdtkke, Karsten Westerdorff, Kai Stohlmann, Anko Börner, Olaf Maibaum, Ting Peng, Benjamin Weps, Görschwin Fey, and Andreas Gerndt. Obc-ng: Towards a re-configurable on-board computing architecture for spacecraft. In *2014 IEEE Aerospace Conference*, pages 1–13. IEEE, 2014.
- [6] Carl Johann Treudler, Heike Benninghoff, Kai Borchers, Bernhard Brunner, Jan Cremer, Michael Dumke, Thomas Gärtner, Kilian Johann Höflinger, Daniel Lüdtkke, Ting Peng, et al. Scosa-scalable on-board computing for space avionics. In *Proceedings of the International Astronautical Congress, IAC*, 2018.

- [7] Jóakim v. Kistowski, Jeremy A. Arnold, Karl Huppler, Klaus-Dieter Lange, John L. Henning, and Paul Cao. How to Build a Benchmark. In *Proceedings of the 6th ACM/SPEC International Conference on Performance Engineering, ICPE '15*, pages 333–336, New York, NY, USA, January 2015. Association for Computing Machinery.
- [8] Marco Vieira, Henrique Madeira, Kai Sachs, and Samuel Kounev. Resilience Benchmarking. In Katinka Wolter, Alberto Avritzer, Marco Vieira, and Aad van Moorsel, editors, *Resilience Assessment and Evaluation of Computing Systems*, pages 283–301. Springer, Berlin, Heidelberg, 2012.
- [9] VargheseBlesson, WangNan, BermbachDavid, HongCheol-Ho, LaraEyal De, ShiWeisong, and StewartChristopher. A Survey on Edge Performance Benchmarking. *ACM Computing Surveys (CSUR)*, April 2021. Publisher: ACM PUB27 New York, NY, USA.
- [10] Lizy Kurian John and Lieven Eeckhout, editors. *Performance Evaluation and Benchmarking*. CRC Press, Boca Raton, January 2017.
- [11] R.P. Weicker. An overview of common benchmarks. *Computer*, 23(12):65–75, December 1990. Conference Name: Computer.
- [12] Reinhold P Weicker. An overview of common benchmarks. *Computer*, 23(12):65–75, 1990.
- [13] Standard performance evaluation corporation. <https://spec.org/benchmarks.html>.
- [14] Transactions processing council,. <https://www.tpc.org/>.
- [15] Christopher Michael Wyant, Christopher Robert Cullinan, and Timothy Richard Frattesi. Computing performance benchmarks among cpu, gpu, and fpga. *Computing*, 2012.
- [16] Byron C Lewis and Albert E Crews. The evolution of benchmarking as a computer performance evaluation technique. *MIS Quarterly*, pages 7–16, 1985.
- [17] Anup Patel, Mai Daftedar, Mohamed Shalan, and M Watheq El-Kharashi. Embedded hypervisor xvisor: A comparative analysis. In *2015 23rd Euromicro International Con-*

- ference on Parallel, Distributed, and Network-Based Processing*, pages 682–691. IEEE, 2015.
- [18] Jack J. Dongarra, Piotr Luszczek, and Antoine Petitet. The LINPACK Benchmark: past, present and future. *Concurrency and Computation: Practice and Experience*, 15(9):803–820, August 2003.
- [19] The linpack 1000x1000 benchmark program. See <http://www.netlib.org/benchmark/1000d> for source code.
- [20] Chuck L Lawson, Richard J. Hanson, David R Kincaid, and Fred T. Krogh. Basic linear algebra subprograms for fortran usage. *ACM Transactions on Mathematical Software (TOMS)*, 5(3):308–323, 1979.
- [21] Lapack—linear algebra package. <https://netlib.org/lapack/>. Accessed: 2022-08-21.
- [22] Lizy Kurian John and Lieven Eeckhout, editors. *Performance evaluation and benchmarking*. CRC Press, Boca Raton, FL, 2006.
- [23] Geekbench. <https://www.geekbench.com/index.html>.
- [24] Matthew R Guthaus, Jeffrey S Ringenberg, Dan Ernst, Todd M Austin, Trevor Mudge, and Richard B Brown. Mibench: A free, commercially representative embedded benchmark suite. In *Proceedings of the fourth annual IEEE international workshop on workload characterization. WWC-4 (Cat. No. 01EX538)*, pages 3–14. IEEE, 2001.
- [25] Mibench. <https://vhosts.eecs.umich.edu/mibench/>.
- [26] Chunho Lee, Miodrag Potkonjak, and William H Mangione-Smith. Mediabench: A tool for evaluating and synthesizing multimedia and communications systems. In *Proceedings of 30th Annual International Symposium on Microarchitecture*, pages 330–335. IEEE, 1997.
- [27] Berkeley design technology, inc. <https://www.bdti.com/services/benchmarking>.

- [28] N.I. Kamenoff. One approach for generalization of real-time distributed systems benchmarking. In *Proceedings of the 4th International Workshop on Parallel and Distributed Real-Time Systems*, pages 202–203, April 1996.
- [29] Hai Jin, Yunfa Li, Zongfen Han, Hao Wu, and Weizhong Qiang. Aeneas: real-time performance evaluation approach for distributed programs with reliability-constraints. *Cluster Computing*, 10(2):175–186, 2007.
- [30] Christoph Boden, Alexander Alexandrov, Andreas Kunft, Tilmann Rabl, and Volker Markl. PEEL: A Framework for Benchmarking Distributed Systems and Algorithms. In Raghunath Nambiar and Meikel Poess, editors, *Performance Evaluation and Benchmarking for the Analytics Era*, Lecture Notes in Computer Science, pages 9–24, Cham, 2018. Springer International Publishing.
- [31] R. Sharma. Flink vs. spark: Difference between flink and spark. <https://www.upgrad.com/blog/flink-vs-spark/> (Accessed: January 11, 2023).
- [32] Giovanni Falcone, Heinz Kredel, Sebastien Kreuter, Michael Krietemeyer, Dirk Merten, Martin Meuer, Matthias Merz, Franz-Josef Pfreundt, David Reinig, and Henry Ristau. *IPACS-benchmark : integrated performance analysis of computer systems (IPACS); benchmarks for distributed computer systems*. Logos-Verl., Berlin, 2006.
- [33] Tyler M. Lovelley, Donavon Bryan, Kevin Cheng, Rachel Kreynin, Alan D. George, Ann Gordon-Ross, and Gabriel Mounce. A framework to analyze processor architectures for next-generation on-board space computing. In *2014 IEEE Aerospace Conference*, pages 1–10, March 2014. ISSN: 1095-323X.
- [34] Justin Richardson, Steven Fingulin, Diwakar Raghunathan, Chris Massie, Alan George, and Herman Lam. Comparative analysis of hpc and accelerator devices: Computation, memory, i/o, and power. In *2010 Fourth International Workshop on High-Performance Reconfigurable Computing Technology and Applications (HPRCTA)*, pages 1–10. IEEE, 2010.

- [35] Leonidas Kosmidis, Ivan Rodriguez, Alvaro Jover-Alvarez, Sergi Alcaide, Jerome Lachaize, Olivier Notebaert, Antoine Certain, and David Steenari. GPU4S: Major Project Outcomes, Lessons Learnt and Way Forward. In *2021 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 1314–1319, Grenoble, France, February 2021. IEEE.
- [36] Rob VanderWijngaart and Bryan A Biegel. Nas parallel benchmarks. In *Supercomputing 2002*, 2002.
- [37] Roland Weigand and Luca Fossati. Dsp benchmark results of the gr740 rad-hard quad-core leon4ft.
- [38] AB Cobham Gaisler. Gr740 technical note on benchmarking and validation. *Doc. No GR740-VALT-0010, ESA Contract, 2000113922:15*, 2019.
- [39] Leonidas Kosmidis, Jerome Lachaize, Jaume Abella, Olivier Notebaert, Francisco J. Cazorla, and David Steenari. GPU4S: Embedded GPUs in Space. In *2019 22nd Euromicro Conference on Digital System Design (DSD)*, pages 399–405, Kallithea, Greece, August 2019. IEEE.
- [40] Zain A H Hammad, Tobias Franz, Olaf Maibaum, Andreas Gerndt, and Daniel Lüdtko. Event-driven multithreading execution platform for real-time on-board software systems. In *Proceedings of the 15th annual workshop on Operating Systems Platforms for Embedded Real-time Applications*, pages 29–34, 2019.
- [41] Deutsches Zentrum für Luft-und Raumfahrt-Sc. Tasking-framework. <https://github.com/DLR-SC/tasking-framework/wiki>.
- [42] Xilinx Inc.: Zynq-7000 SoC. Data sheet overview. <https://docs.xilinx.com/v/u/en-US/ds190-Zynq-7000-Overview>.
- [43] S Sinha. Fundamentals of mapreduce with mapreduce example. medium. retrieved january 8, 2023, from. <https://medium.com/edureka/mapreduce-tutorial-3d9535ddbe7c>.