Fault-Tolerant Satellite Computing with Modern Semiconductors

ISBN: 978-94-028-1766-9

Fault-Tolerant Satellite Computing with Modern Semiconductors

Proefschrift

ter verkrijging van
de graad van Doctor aan de Universiteit Leiden,
op gezag van Rector Magnificus prof.mr. C.J.J.M. Stolker,
volgens besluit van het College voor Promoties
te verdedigen op dinsdag, 17 december 2019
klokke 11:15 uur
door

Christian Martin Fuchs

Geboren te Linz, Oostenrijk in 1984

Promotor: Prof. Dr. A. Plaat

Promotiecommissie:

Dr. H. Quinn	Los Alamos National Laboratory, Los Alamos, USA
Prof. Dr. X. Wen	Kyushu Institute of Technology, Japan
Dr. M.S. Gorbunov	Scientific Research Institute of System Analysis, Russian Academy of Sciences, Moscow, Russia
Prof. Dr. J.J. Liou	National Tsing Hua University, Hsinchu, Taiwan
Prof. Dr. S. Wu	Shanghai Jiao Tong University, Shanghai, China
Dr. M. Kenworthy	
Prof. Dr. S. Manegold	
Dr. E. Bakker	
Prof. Dr. H. Wijshoff	

For taking these first steps into a new frontier.

Front & Back cover: Illustrations by Dr. Nadia M. Murillo Mejías. Image of Europa taken by the Galileo spacecraft during its second orbit around Jupiter.

Contents

Preface **1**
Space: The Final Frontier 1

1 Introduction **3**
1.1 Problem Statement 5
1.2 Research Questions 6
1.3 Thesis Organization 7

2 A Brief Introduction to Spaceflight and Fault Tolerance
Thesis Motivation and Legitimization **11**
2.1 Spacecraft and Satellite Miniaturization 12
2.2 Early CubeSat Reliability and Motivation 17
2.3 Nanosatellites Today and Legitimization 19
2.4 Fault-Tolerant Computer Architecture 21

3 The Space Environment
Physical Fault Profile and Operational Considerations **31**
3.1 The Impact of the Space Environment on Electronics 32
3.2 Technology Readiness and Standardization 39
3.3 Operational Constraints for Satellite Computers 41

4 A Fault Tolerance Architecture for Modern Semiconductors
Stage 1 & Architecture Overview **47**
4.1 Introduction 48
4.2 Related Work 49
4.3 Fault Tolerance through Software 51
4.4 Stage 1: Short-Term Fault Mitigation 54
4.5 Stage 2: MPSoC Reconfiguration & Repair 59
4.6 Stage 3: Applied Mixed Criticality 61
4.7 Platform Architecture 62
4.8 Discussions 67
4.9 Conclusions 68
4.10 Annex: Worst-Case Performance Estimation 69

5 MPSoC Management and Reconfiguration
Stage 2 **73**
5.1 Introduction 74

5.2 Debugging and Reliability . . . 75
5.3 Implementation Details . . . 76
5.4 Use Cases beyond Debugging . . . 82
5.5 Discussions . . . 87
5.6 Conclusions . . . 87

6 Mixed Criticality and Resource Pooling
Stage 3 **89**
6.1 Introduction . . . 90
6.2 Background . . . 90
6.3 Related Work . . . 91
6.4 System Overview & Requirements . . . 92
6.5 System Architecture Review . . . 94
6.6 Spare Resource Pooling . . . 97
6.7 Adapting to Varying Mission Requirements . . . 98
6.8 Discussions . . . 103
6.9 Conclusions . . . 104

7 Reliable Data Storage for Miniaturized Satellites
Memory Fault Tolerance **105**
7.1 Introduction . . . 106
7.2 Data Integrity as Foundation of Fault Tolerance . . . 107
7.3 Volatile Memory Consistency . . . 109
7.4 A Radiation-Robust Filesystem for Space Use . . . 115
7.5 High-Performance Flash Memory Integrity . . . 122
7.6 Conclusions . . . 131

8 Validating Software-Implemented Fault Tolerance
Systematic Fault Injection **133**
8.1 Introduction . . . 134
8.2 Related Work . . . 136
8.3 Target Implementation . . . 138
8.4 Obtaining a Practical Fault Model . . . 139
8.5 Suitable Fault-Injection Techniques . . . 140
8.6 Test Campaign Setup . . . 142
8.7 Executing a Test Campaign . . . 143
8.8 Results & Interpretation . . . 150
8.9 ArchC MPSoC vs. FIES Result Comparison . . . 153
8.10 Comparison to Literature . . . 154
8.11 Discussions . . . 155
8.12 Conclusions . . . 157

9 Combining Hardware and Software Fault Tolerance
High-Level System Design **159**
9.1 Introduction . . . 160
9.2 Background & Related Work . . . 160
9.3 A Hybrid Fault Tolerance Approach . . . 161
9.4 The MPSoC Architecture . . . 163
9.5 Subsystem Connectivity and Peripheral I/O . . . 167

9.6 Implementation Considerations 169
9.7 Conclusions 170

10 On-Board Computer Integration and MPSoC Implementation *Practical Design Verification on FPGA* **171**
10.1 Introduction 172
10.2 Related Work 173
10.3 A Reliable CubeSat On-Board Computer 175
10.4 Handling Chip-Level SEFIs and Failure 187
10.5 Utilization and Power Comparison 189
10.6 Experimental Results and Testing 192
10.7 Conclusions 192

11 Conclusions and Outlook **195**
11.1 Conclusions 195
11.2 Discussions 197
11.3 Outlook and Future Work 199

Bibliography **202**

Nederlandse Samenvatting **229**

中文摘要 **235**

中文摘要（繁體） **239**

日本語の要約 **243**

Resumen en Español **247**

Резюме на Русском Языке **253**

English Summary **259**

List of Selected Publications **263**

Curriculum Vitae **265**

Acknowledgments **269**

Preface

Space: The Final Frontier

Humankind has been fascinated by the stars, and planets of our solar system, probably since before our species developed complex language. Many cultures have considered them to be ancestors, spirits of nature, and deities guiding our life and influencing our world. As humankind developed, people chose to see their heroes in the constellations, and these curious objects in the sky sometimes even were considered gods. Knowing what these gods wanted or liked could help a society prosper, or could doom it. Even more were we intrigued by the Sun, our neighboring planets, the Moon.

Technology has always been critical in our quest to understand our environment, and our world. Today, we are dependent upon the availability and correct functioning of our technology. It has enabled us to transform nature, but also to damage it and most likely change it for generations. And we are using technology even in our attempts to repair some of that same damage we inflict through it. Without technology, modern societies and our every day life would be unthinkable.

Humans are curious, and using our technology, we began exploring space just recently, considering the timescale of human existence. We operate vast telescopes on the ground and in space, which help us answer the most fundamental questions about how we came to be and where we are going. A few decades ago, we began launching satellites into space, which we today use for science, commerce, and education. Two superpowers conducted a great race to the Moon just a few decades ago, arrived there, took pictures, and then returned home. Today, this race is being rerun with more participants, resulting maybe in an extension to Mars, or better and more productively, to the Galilean Moons of Jupiter.

Satellites allow us to communicate with any point on the surface of the Earth in real-time, and with Mars with more than 10 minutes delay. Weather forecasts, communication services, flight information, and geolocation systems today are possible only due to information transmitted, or relayed by satellites. In many aspects, our modern life would be unimaginable without them.

We have outgrown our homeworld and its limited pool of resource already in many aspects, and most likely we even have to go to space to survive, like a young bird leaving its nest. Within the next few generations, we will reach out into space, begin to understand whatever we may find there, and utilize the vast resources which we may find within our solar system for the benefit of all. To design, construct, test, and operate the spacecraft that we will require we depend upon modern computer technology and electronics.

Electronics and semiconductor technology are indispensable in spacecraft design,

and microprocessors can be found in all major satellite subsystems. Spacecraft and computers represent the peak of our technology, the application of all our skills in engineering, and the result of all the combined interdisciplinary scientific knowledge we have as a species. The reliability of these components is mission critical; and directly or indirectly, lives depend upon them, even in unmanned spaceflight. Scientists and engineers therefore seek to invent, develop, and utilize computer designs which can guarantee sufficient robustness and reliability for a space mission. The topic of this thesis is to enable the use of modern computer technology manufactured in fine technology nodes, which at the time of writing can not be used aboard spacecraft in a reliable manner.

Chapter 1

Introduction

Brief Abstract

Modern semiconductor technology has enabled the development of miniaturized satellites, which are cheap to launch, low-cost platforms for a broad variety of scientific and commercial instruments. Especially very small satellites (<100kg) can enable space missions which previously were technically infeasible, impractical or simply uneconomical. However, as discussed in Chapter 2, they suffer from low reliability. Especially the smallest such satellites are typically not considered suitable for critical and complex multi-phased missions, as well as for high-priority science missions for solar-system exploration and astronomical applications [1]. The on-board computer (OBC) and related electronics constitute a significant part of such spacecraft, and in related work, e.g., [2], were responsible for a majority of post-deployment failures, which are further discussed also in Chapter 3.

Indeed, the modern embedded and mobile-market semiconductors used aboard nanosatellites lack the fault tolerance (FT) capabilities of computer-architectures for larger spacecraft. Due to budget, energy, mass, and volume restrictions in miniaturized satellites, existing FT solutions developed for such larger spacecraft can not be adopted. Today, there exist no fault-tolerant computer architectures that could be used aboard nanosatellites powered by embedded and mobile-market semiconductors, without breaking the fundamental concept of a cheap, simple, energy-efficient, and light satellite that can be manufactured en-mass and launched at low cost [3].

To overcome this limitation, in this thesis, we develop a new approach to achieve fault tolerance for miniaturized satellite computers based upon modern semiconductors. The method we use to approach this challenge is to first consider protective measures proposed by science as theoretical concepts, as well as measures that are in use today in the space industry and other industries in Chapters 2, 3, and 4. We consider how these can be utilized to systematically protect each component of a spacecraft's OBC, as well as the software run on it.

A high-level schematic of the components making up a satellite on-board computer is depicted in Figure 1. For each OBC component indicated in this figure, we develop fault tolerance measures that can be used to protect them and describe them in the different chapters of this thesis. To assure that these concepts are effective, we develop them specifically considering the application constraints and requirements of a

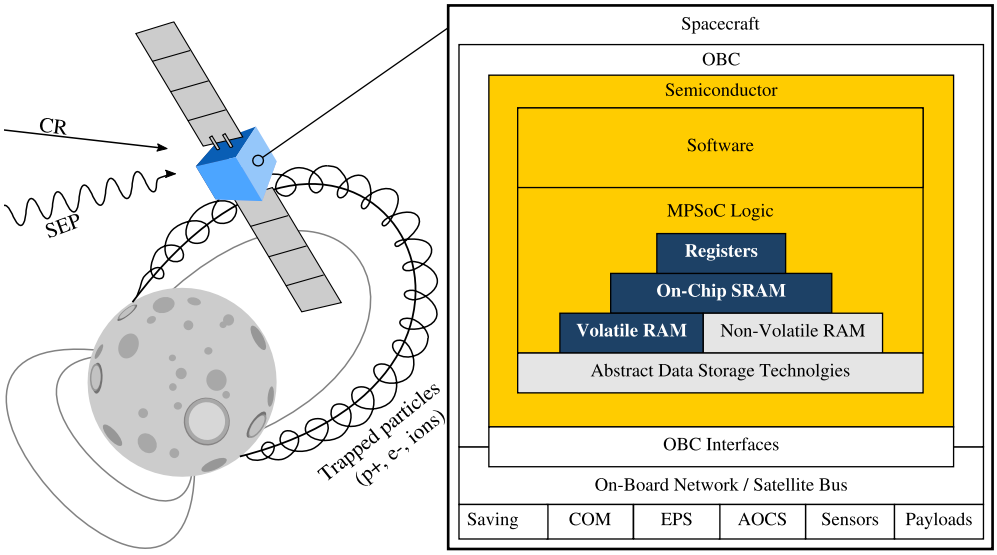


Figure 1: A high-level component model of an OBC, and the other subsystems within a satellite interacts with.

satellite operating in the space environment. Based on these concepts, we propose the hypothesis that fault tolerance can be achieved through hardware-software co-design, for which we produce a theoretical design in the form of a three-stage fault tolerance architecture.

We show that by systematically protecting critical key-component of the OBC using software measures, synergies between different fault tolerance measures can be achieved. These synergies enable us to protect the system as a whole more effectively, efficiently and in a way that is economical and feasible even for small-scale professional CubeSat developers and academic teams working on scientific spacecraft and instruments with a limited project budget. We test our hypothesis through fault-injection and provide statistics on the results, and implement a proof-of-concept for this system architecture in a reconfigurable logic device (FPGA).

Our ultimate objective is to allow a suitable miniaturized satellite design to reliably achieve a minimum of 2 years of on-orbit operation. At the time of writing, miniaturized satellite computer components do not include sophisticated fault tolerance capabilities, and may fail at any point in time during a space mission. In contrast to large spacecraft, they therefore can not be designed to achieve a specific mission lifetime, but designs function as long as no critical faults occur. Therefore, these missions are kept brief, as is further discussed in Chapters 2 and 3, thus implying risk acceptance instead of risk mitigation and risk handling.

We realize fault tolerance in software and assure an on-board computer's long-term robustness by exploiting partial FPGA-reconfiguration (see Chapter 5) and mixed criticality aspects (see Chapter 6), and develop a multiprocessor System-on-Chip (MPSoC) architecture through hardware-software co-design (see Chapter 4). Hence, this computer architecture also provides spacecraft designers with the capabilities necessary to achieve a given mission lifetime by adjusting our architecture's parameters, such as the necessary level of replication of software run on the system, provisioning

of spares, scrubbing periods, and error correction coding strength.

The MPSoC requires no custom-written IP-cores (library logic) and can be assembled from well tested commercial-off-the-shelf (COTS) components, and powerful embedded and mobile-market processor cores, yielding a non-proprietary, and open system architecture. The resulting computer architecture consists only of conventional consumer-grade hardware, commodity processor cores, standard parts, and openly available standard library IP.

In the final chapter of this thesis, we provide a proof-of-concept implementation of this MPSoC for three FPGAs, the Xilinx Kintex Ultrascale+ KU3P (the smallest of its class), KU11P, and the Xilinx Kintex Ultrascale KU60. Our implementation for KU3P requires only 1.94W total power consumption, which is well within the power budget range achievable aboard 2U CubeSats. To our understanding, this is the first scalable and COTS-based, widely reproducible OBC solution which can offer strong fault tolerance even for 2U CubeSats.

1.1 Problem Statement

Hardware-based fault tolerance measures for large satellites are effective for older, large-feature-size technology nodes which have fallen out of use in the mobile-market and the IT industry decades ago [4]. Modern mobile-market COTS processors depend upon manufacturing in low-feature size technology nodes, and can not be manufactured anymore using old technology nodes. Traditional hardware-implemented fault tolerance techniques diminish in effectiveness and efficiency with shrinking feature size [5]. This has left a protective gap due to a lack of fault-tolerant solutions, and the reliability of such miniaturized satellites is insufficient for critical missions, which is further discussed in Chapter 3.

Countless novel academic fault tolerance concepts have been proposed over the years, which, in theory, could be used to protect modern computer systems. But at the time of writing, there is a significant gap between fault tolerance research, and its applications to spacecraft of all classes, as discussed as part of related work in Chapters 4, 6, and 8. Many of the concepts mentioned there have low technological maturity and do not meet practical application constraints for a use within a real computer system, regardless of the intended operating environment [1]. Software-implemented fault tolerance concepts have thus until today been ignored by the space industry due to lacking maturity, perceived complexity, doubts about their effectiveness and testability [1].

In this thesis we therefore explore how fault tolerance can be achieved for computer systems manufactured in state-of-the-art technology nodes with low power-usage, and small feature-size through scientific means. We do this in collaboration with the European Space Agency, supported by a Networking Partnership Program grant. In this thesis we address the following problem:

RQ0 *Can a fault tolerance computer architecture be achieved with modern embedded and mobile-market technology, without breaking the mass, size, complexity, and budget constraints of miniaturized satellite applications?*

1.2 Research Questions

To show that it is indeed possible to address the problem stated in RQ0 in an affirmative way, we develop a fault-tolerant system architecture which can do exactly that. Systematically for each component in a satellite's on-board computer, we develop specific measures to address challenges regarding fault tolerance. These components are also depicted in Figure 1. However, we do not try to apply fault tolerance everywhere in the system as, as this would inflate system complexity and fault potential. Instead, we place fault tolerance measures strategically within the system to handle and cover faults where these can be addressed best at a system level.

In this thesis, we investigate the following research questions throughout the different chapters:

RQ1 Considering the design constraints of nanosatellites, can a fault-tolerant computer architecture be achieved with COTS components?
(Chapter 4)

RQ2 How can the correct functionality of a CubeSat's FPGA-based on-board computer be assured and verified, and its lifetime extended?
(Chapter 5)

RQ3 Can a satellite computer architecture enable novel functionality for a satellite computer, that improves satellite computing beyond just offering better fault tolerance and an increased lifetime?
(Chapter 6)

RQ4 Can commercial memories be retrofitted with error detection and correction in software, to substitute for hardware measures, and to what extent?
(Chapter 7)

RQ5 How can its software-implemented fault tolerance measures of a hardware- software hybrid architecture be tested and validated?
(Chapter 8)

RQ6 Can such a computer architecture be practically implemented within the size, energy, and budget constraints of nanosatellite applications?
(Chapters 9 & 10)

These questions are discussed in this thesis. To do so, we develop a fault-tolerant computer architecture for irradiated environments which can offer protection for on-board computer systems based upon modern semiconductors. Through implementation, testing via fault-injection, and the construction of a proof-of-concept implementation on FPGA, we show that this approach is technically feasible with contemporary technology.

The key contribution of this thesis is a computing concept that can allow future critical commercial and high-priority science missions to be done at low cost, to enable REAL progress in satellite miniaturization to take us as a species to the stars. My hope is that this thesis is the beginning of something new and significant, and in the coming years I plan to advance this technology from its current proof-of-concept state to maturity. To do so, radiation testing, long-term testing, as well as on-orbit demonstration aboard a CubeSat will be necessary.

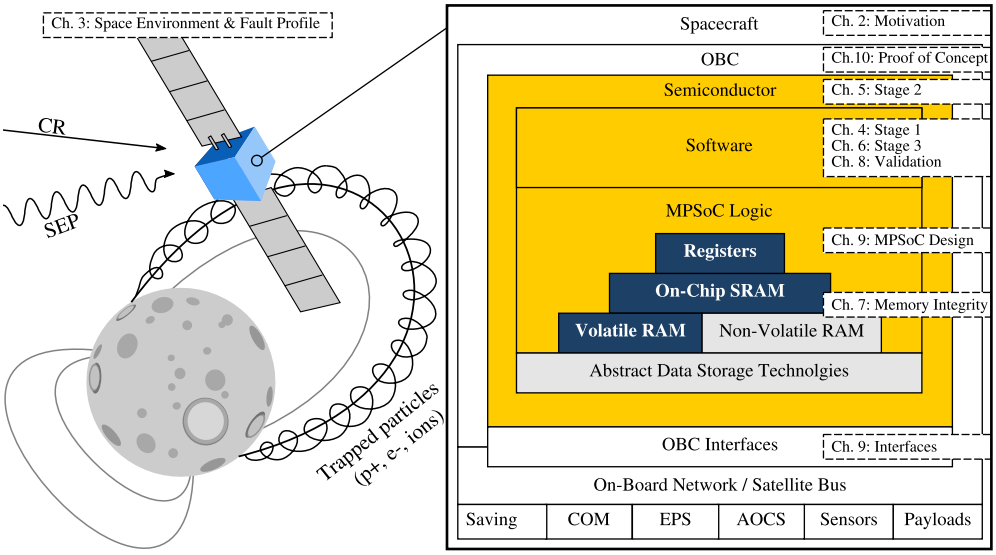


Figure 2: Chapter guide for this thesis.

1.3 Thesis Organization

A brief outline of the subsequent chapters follows, with a visual chapter guide depicted in Figure 2.

Chapter 2: A Brief Introduction to Spaceflight and Fault Tolerance

The research upon which this thesis is based is interdisciplinary. It relies upon concepts and results from several different fields, including computer engineering, nuclear science, electrical engineering, physics and astronomy, as well as space engineering. In this chapter, we provide a brief introduction to our application, its design constraints, as well as fault-tolerant computer architecture. We further provide an overview over the current status of small satellite space missions, as well as a review on satellite failures in the past and at the time of writing. This chapter therefore serves also as motivation and legitimization for our research, including mission success and failure statistics, which underline the lack of reliability of very small satellites today.

Chapter 3: The Space Environment

A satellite's on-board computer has to cope with unique challenges, requiring a general understanding of the physical effects of a spacecraft's operating environment. Hence, for the understanding of the fault profile and application constraints for this thesis, in this chapter we provide an in-depth discussion of the space environment and its effects. We discuss the physical design restrictions aboard spacecraft, and operational considerations. Most importantly we discuss the impact of radiation on semiconductors, and how it can be mitigated.

Chapter 4: A Fault Tolerance Architecture for Modern Semiconductors

In this chapter, we describe a non-intrusive, integral, flexible, hardware-software-hybrid approach which enable the use of modern MPSoCs for spaceflight meeting real-world constraints. Neither traditional hardware- nor software-based FT solutions can offer the functionality necessary to guarantee fault tolerance for state-of-the-art SoCs used in miniaturized satellite OBCs. We achieve fault-detection, isolation and recovery through the use of a co-designed fault tolerance architecture consisting of multiple interlinked protective measures. In combination, they form a fault tolerance architecture which can guarantee strong fault coverage even during space missions with a long duration, for which we provide an early proof-of-concept implementation. The research in this chapter was published in the proceedings of the *IEEE Asian Test Symposium (ATS)* [Fuchs9].

Chapter 5: MPSoC Management and Reconfiguration

In this chapter, we present the concept and proof-of-concept implementation of a subsystem for autonomous chip-level debugging within a CubeSat via JTAG [6]. This concept provides all the necessary functionality needed to implement Stage 2 of the fault tolerance architecture described in Chapter 4. In our multi-stage fault tolerance architecture, remote debugging is one of several tasks this subsystem performs: It is now used to control the coarse-grain lockstep implemented within an MPSoC, and referred to as supervisor in remainder of this thesis. It interacts with an on-chip configuration controller to control partial reconfiguration and error scrubbing for the FPGA's fabric via the internal configuration access port (Xilinx's ICAP). An early version of this chapter was presented in the proceedings of the *International Conference on Architecture of Computing Systems (ARCS)* [Fuchs11], and an extended paper [Fuchs10] was published in the proceedings of the *ESA/CNES Small Satellites, System & Services Symposium (4S)*.

Chapter 6: Mixed Criticality and Resource Pooling

In this chapter, we discuss Stage 3 of our multi-stage fault tolerance architecture, and the advantages it offers not just for miniaturized satellites, but for spacecraft of all weight classes. Our architecture allows a satellite to dynamically adjust the fault tolerance level, compute performance, and energy consumption to meet the varying performance requirements to a satellite computer during long and multi-phased space missions. The operator of a spacecraft can prioritize between processing performance, functionality, fault coverage, and energy consumption. The system can be autonomously adapted to the OBC's thread assignment to retain a functional system core by sacrificing performance or availability of less critical applications. This allows an OBC to to more efficiently handle accumulating permanent faults and to age gracefully. The research in this chapter was published [Fuchs7] in the proceedings of the *NASA/ESA Conference on Adaptive Hardware and Systems (AHS)*.

Chapter 7: Reliable Data Storage for Miniaturized Satellites

Reliable operation of an OBC can only be guaranteed if the integrity of the OBC's operating system, applications, as well as payload data can be safeguarded. Chapter 7 is therefore dedicated to discussing fault tolerance for the various volatile and non-volatile memories used aboard miniaturized satellites and within our architecture. The research presented in this chapter was published as finalist paper [Fuchs15] in the proceedings of the *AIAA/USU Conference on Small Satellites (SmallSat)*. It was awarded second place and a research grant in the *Annual Frank J. Redd Student Competition*. We describe the implementation of FTRFS, a fault-tolerant radiation-robust filesystem for space use. It was published [Fuchs18] in the proceedings of the *International Conference on Architecture of Computing Systems (ARCS)*. Furthermore, a protective concept for flash memory and phase change memory is described in the second part of this chapter. It was published [Fuchs16] in the proceedings of the *International Space System Engineering Conference Data Systems In Aerospace (DASIA)*.

Chapter 8: Validating Software-Implemented Fault Tolerance

In this chapter, we test and validate the software-mechanisms that are the foundation of our fault tolerance architecture by injecting faults into an RTEMS implementation of Stage 1. Traditional computer architectures for space applications are validated using system-level testing. This is viable for systems relying on hardware measures, but unsuitable for testing software due to a lack of test coverage and the expanded test-space. For testing software-based FT measures, a realistic test-setup is considered good practice and required to deliver representative fault-injection results. Therefore, a fault-injection campaign was conducted using system emulation through QEMU into a representative ARMv7a-SoC matching our architecture target, ARM's Cortex-A53, and into a RISC-V-based SystemC-model. Our results show that our lockstep implementation is effective and efficient, and we provide a direct comparison to related work. An early version of this chapter was published in the proceedings of the *IEEE Asian Test Symposium (ATS)* [Fuchs5].

Chapter 9: Combining Hardware and Software Fault Tolerance

As optimal platform for our architecture, we developed a compartmentalized MPSoC design for FPGA, where Stage 2's partial reconfiguration functionality can be utilized to recover defective parts of the MPSoC. This architecture is designed to satisfy the high performance requirements of current and future scientific and commercial space missions at very low cost, while offering the strong fault coverage guarantees necessary for missions with a long duration. We describe the topology of our multiprocessor System-on-Chip (MPSoC), and show how it can be assembled in its entirety from only well tested COTS components with commodity processor cores. The MPSoC can be implemented using only COTS hardware and extensively validated library IP, requiring no custom logic or space-proprietary processor cores. The research in this chapter was published [Fuchs6] in the proceedings of the *IEEE Conference on Radiation and Its Effects on Components and Systems (RADECS)*.

Chapter 10: On-Board Computer Integration and MPSoC Implementation

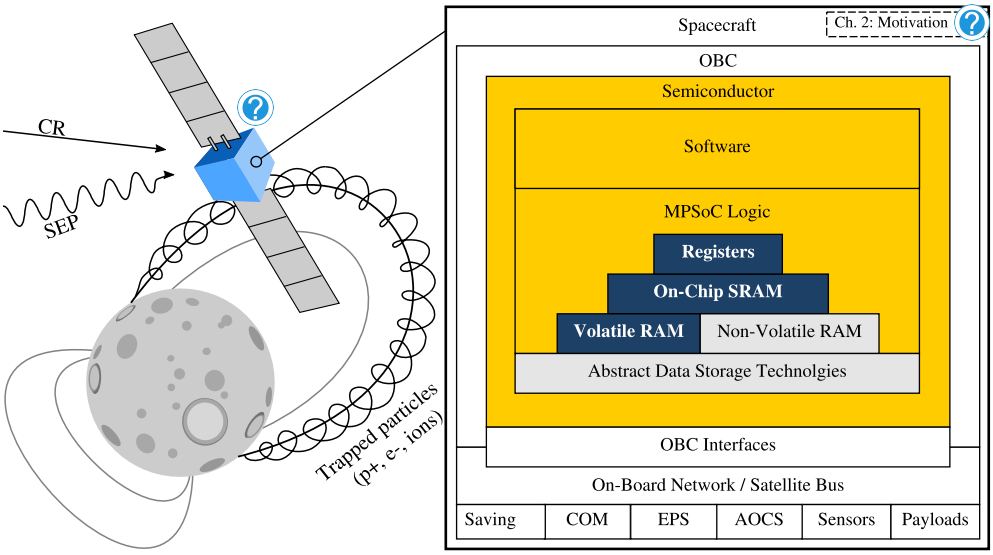
In the final research chapter of this thesis, we discuss practical implementation results for our MPSoC design. We provide detailed resource utilization results for this MPSoC for 3 different FPGAs: Xilinx Kintex Ultrascale+ KU3P (the smallest of its class), KU11P, and the Xilinx Kintex Ultrascale KU60, for which we are collaborating within the Xilinx Radiation Testing Consortium to achieve a suitable device-test platform for radiation testing in the future. We provide statistics on power consumption, and show that even between two FPGA generations power consumption can be reduced drastically through the use of more modern and efficient technology nodes. This serves as proof-of-concept for our architecture. This chapter is based on two publications [Fuchs1,Fuchs2] in the proceedings of to the *IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFT)* and the *AIAA/USU Conference on Small Satellites (SmallSat).*

Chapter 2

A Brief Introduction to Spaceflight and Fault Tolerance

Thesis Motivation and Legitimization

The research upon which this thesis is based does not come from one single field of science, but is interdisciplinary. It relies upon concepts and results from several different fields, including computer engineering, nuclear science, electrical engineering, physics and astronomy, as well as space engineering. In this chapter, we provide a brief and informal introduction to our application, its design constraints, as well as fault-tolerant computer architecture. We further provide an overview over the current status of small satellite space missions, as well as a review on satellite failures in the past and at the time of writing. This chapter therefore serves also as motivation and legitimization for our research, including mission success and failure statistics, which underline the lack of reliability of very small satellites today.



2.1 Spacecraft and Satellite Miniaturization

In this section, a brief introduction into the different kinds of satellites and satellite miniaturization itself is given, to provide general understanding for readers who are not familiar with this field. This section is meant as to give sufficient background information on the application for the research discussed in this thesis.

Satellites can be differentiated by mass in several classes.When thinking of space stations, satellites, and deep-space probes, we usually imagine large structures floating in space, weighing multiple tons, powered by vast solar panel arrays, radioisotope thermoelectric generators, or fission reactors [7]. Certainly, many early scientific, commercial, and military satellites were very large spacecraft. These are sometimes designed to operate for several decades in space. However, today, modern semiconductor technology, more efficient battery and photovoltaics, novel propulsion technologies, and robust lightweight materials enable the construction of much smaller, lighter, and cheaper spacecraft.

Spacecraft with a wet mass[1] of less than 500kg are therefore referred to as "miniaturized satellites", and can be constructed dramatically faster than large satellites. In Table 1, an overview over satellite classes and capabilities is given.

At the time of writing, several companies have achieved commercial success by operating large groups of miniaturized satellites in orbit. They have been successfully used to providing real-time earth observation data and help in disaster recovery [8], and in safety- and life-critical services [9] such as airplane traffic tracking and maritime shipping [10]. A broad variety of biological and chemical experiments [11] has been carried out using CubeSat platforms, which are also rather popular for testing and validating novel technologies in space [12, 13]. Several pico- and nanosatellite-based space-observatories [14, 15] have been launched, and nanosatellites were deployed by the Hayabusa 2 space probe at the asteroid *162173 Ryugu* [16]. In 2018, 2 interplanetary CubeSats traveled to the planet Mars as part of the MarCO mission [17],

[1]The mass of the spacecraft including payload and all consumables such as propellant.

	Weight		Minia-	Build as	Classical	Propulsion	Mission
Class	Max	Min	turized	CubeSat	Tech Usable	Available	Lengths
Large	-	1t	No	Absurd	Yes	Yes	Decades
Medium	1t	500kg	No	Absurd	Yes	Yes	Decades
Small	500kg	100kg	Yes	Limiting	Most	Yes	10 years
Micro	100kg	10kg	Yes	Common	Little	Yes	years
Nano	10kg	1kg	Yes	Standard	No	Yes	1 year
Picro	1kg	100g	Yes	Standard	No	Limited	months
Femto	100g	-	Yes	Inefficient	No	No	-

Table 1: Satellites can be classified in a variety of ways, with each type of spacecraft having different capabilities, technological limitations, and the capability to achieve different mission durations. In principle, almost any satellite could be manufactured to be a CubeSat, but only for some this makes sense due to the constraints of this form factor standard.

providing real-time telemetry during the arrival-phase of NASA's InSight Mars Lander. Several miniaturized satellite constellations for technology demonstration, and Earth observation, and positioning, and data relay purposes have been developed [18–21] and launched [8, 22, 23]. At the time of writing, scientists and engineers have even begun to develop CubeSat-based interferometers and composite space telescopes [13] that could outperform even the largest conventional space-observatories, and there are plan to use Nanosatellites even for gravitational-wave measurement [15].

2.1.1 Large Satellites based on Traditional Design Principles

Satellites with a wet mass above 500kg are at this point in time constructed in large projects with vast budgets quasi artisanally. Most "big-space" applications rely upon such satellites. Satellites of 500kg – 1000k are usually classified as *medium-sized satellites*, heavier spacecraft are designated as a *large satellites*. Development of such satellites is challenging, system architectures are complex, resulting in long development times, and the need to utilize well tested, proven technology, that is available over a very long period of time. This technology is usually space industry proprietary. Technology readiness, design maturity, and space heritage of a technology through prior use aboard other spacecraft are essential, and often seen a prerequisite for considering a technology for use within this satellite class.

Construction of these satellites in practice often takes many years [24], sometimes even decades [25]. To provide an example, the James Webb Space Telescope (JWST) is designed to have a wet mass of approximately 6620kg. It is a multinational project involving hundreds of stakeholders, and has been in construction for more than 25 years at the time of writing, and its precise date of completion and launch has not been announced yet. The cost of the electronics used aboard such a spacecraft is small compared to the funds required to meet legal requirements, for salaries, tooling, testing, management, certification, insurance, and launch. Spacecraft testing also requires access to specialized facilities [26, 27] including:

- thermal/vacuum chambers to analyze the behavior of the spacecraft in a space-like environment at high or low temperatures (often 173K and 373K) [28],
- radiation testing facilities using radiogenic sources or particle accelerator to simulate the radiation environment a satellite's components have to operate in, and to verify their correct behavior and, if available, effectiveness of fault tolerance measures, and
- a broad variety of other heavy machinery, e.g., to perform mechanical stress and vibration tests.

Most modern major launch vehicles can carry much heavier and bulkier loads than just one satellite [29, 30]. Often a substantial amount of volume and mass remains available which in the early days of spaceflight remained vacant to not endanger the primary payload [31]. To reduce costs, organizations often either sell this excess capacity, or hand the entire launch process over to a "launch broker", which then can combine multiple satellite launches into one "ride-share" launch [29]. An example of a ride-share launch with multiple satellites of various classes is depicted in Figure 3. The main spacecraft launched on a launch vehicle is then referred to as "primary payload", with other, often smaller satellites becoming "secondary payloads". Today

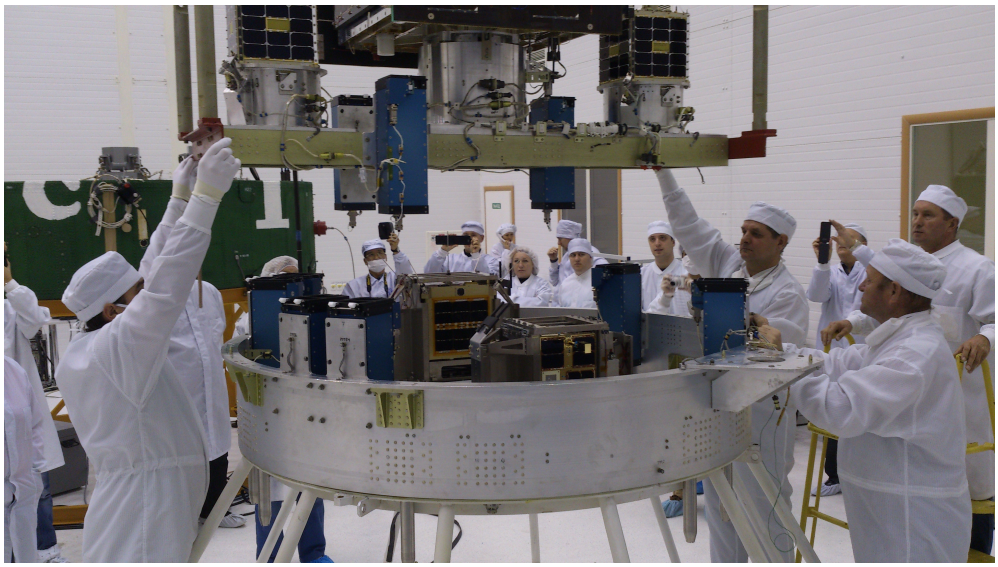


Figure 3: A ride-share satellite launch with the Earth observation SmallSat DubaiSat-2 (top center) being the primary payload. Secondary payloads were 4 microsatellites (top left and right, 2 bottom center) and 26 other nanosatellites which are located in the blue deployer boxes. The CubeSat First-MOVE (see Section 2.1.4) is located in the top right deployer.

even small start-up companies, and universities can bring their spacecraft into orbit at comparably low cost.

2.1.2 Small Satellites

SmallSats, or Minisatellites, weigh between 500 and 100kg, and traditionally were used for brief science and commercial missions. Historically, SmallSat missions used to be shorter than those realized with large satellites [32]. They can be constructed and launched at drastically lower cost, and in general also more quickly. The term SmallSat is colloquially also used to refer to *all* satellites lighter than 500kg in this field. Due to technological evolution in recent decades, the capabilities of the SmallSats have increased, and today they increasingly much replace larger satellites.

2.1.3 Microsatellites

MicroSats between 100kg and 10kg are today widely used for a variety of low cost commercial and novel scientific missions. The upper and lower boundaries between Nanosatellites, MicroSats, and SmallSats are fluent. MicroSats with a wet mass approaching 100kg differ little from lighter SmallSats, and usually carry fewer or lighter payloads and lighter components (e.g., smaller batteries, lighter and smaller solar cell array structures, ...) [33]. Light MicroSats become similar to a Nanosatellite and may even utilize Nanosatellite form factor standards, while larger ones can offer very similar capabilities to SmallSats. Many missions that a few decades ago required SmallSats can today be performed by MicroSats, which can be manufactured more rapidly and

launched at lower cost. Compare also [34] for a market assessment for a corporate view on this increasing down-scaling trend.

2.1.4 Nanosatellites and CubeSats

Nanosatellites weigh between 1 and 10kg and became popular for educational projects, especially due to the CubeSat standard. The CubeSat standard was originally intended to cheaply launch student projects into space at the beginning of the 21st century [35]. Today, it has become the standard form factor for Micro-, Nano-, and Picosatellites, and an example of a CubeSat is depicted in Figure 5. It requires a satellite to conform to certain design restrictions, e.g., banning the use of explosive substances within the satellite, and otherwise implies a stackable standard form-factor consisting of 10x10x10cm CubeSat units (U) and a maximum of 1.33 kg per 1U. CubeSats are designed to fit a standardized CubeSat *deployer*. Figure 4 depicts such a deployer consisting of a spring, and electric latch, which once the latch is released allows CubeSats to be safely be deployed by pushing them out of the box. This enables even heavy 12U or 24U designs (3x2x2 or 4x2x3U stacked) to be launched at reduced cost, and allows testing requirements to be reduced for launch qualification, as the failure of a CubeSat during launch will not interfere with the deployment of other satellites aboard the same launcher.

At the time of creation of the CubeSat standard, nanosatellites were intended to perform only simple and short missions in Low Earth Orbit (LEO), e.g., student education, or on-orbit concept validation. They rely on cheap commodity technologies and COTS components, such as lithium-polymer based batteries, and solar-cells intended for ground use. However, due to the rapidly increasing performance of embedded

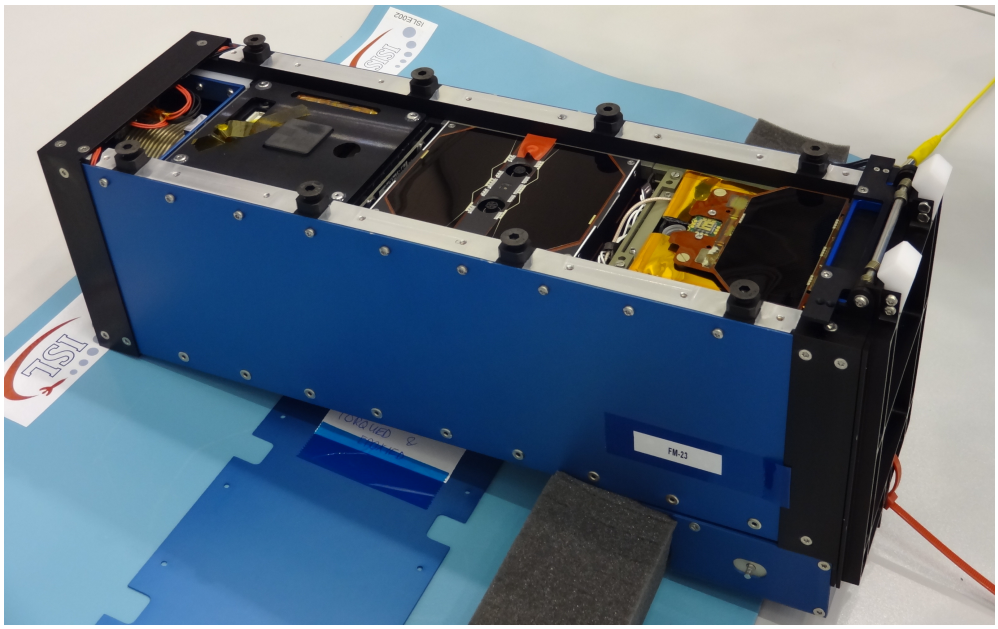


Figure 4: A 3U-CubeSat deployer holding First-MOVE (right), and two other 1U CubeSats.

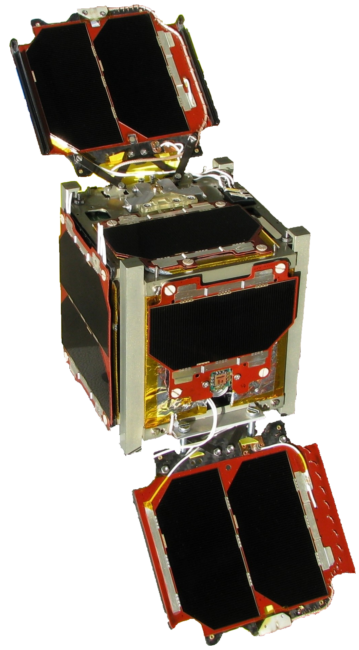


Figure 5: The 1U-CubeSat First-MOVE.

and mobile-market hardware since the early 2000s, the capabilities of nanosatellites have evolved considerably. At the time of writing, a diverse ecosystem of ready-to-use CubeSat components has developed. A variety of commercial companies of varying technical capabilities provide a customizable solutions of mixed quality, with ample launch opportunities into different orbits being available for 1–12U CubeSats.

The CubeSat First-MOVE (depicted in Figure 5) was one of these educational projects [36]. In 2013, I joined a research group developing this satellite at Technical University Munich, Germany, as a master student. Like many other first-generation educational CubeSats, First-MOVE was designed, constructed, and tested primarily by university students at the PhD, Master, and Bachelor levels. Planning of the First-MOVE mission began in 2006, a time when modern smartphones had just arrived in the consumer market, and construction in earnest began around 2010. It was launched into LEO on November 21st, 2013, and its malfunction, which is further described in Section 2.2, was the origin of the author's research on satellite fault tolerance.

2.1.5 Picosatellites and PocketQubes

PicoSats range in weight from between 0.1 to 1kg, and are today used for education or very brief proof-of-concepts. The PocketQube form factor and many 1U CubeSats fall into this category, and the electrical architecture of such PicoSats is often similar or even identical to that of light Nanosatellites. The main difference is lower mechanical complexity, and a further constrained power budget due to reduced solar cell surface (often ranging around or below 5W). In practice, this implies limitations especially for transceivers and payload, which are the main power consumers aboard modern miniaturized spacecraft.

2.1.6 Femtosatellites

FemtoSats are the smallest miniaturized satellite form factor and weigh less than 0.1kg. The concept of FemtoSats was theoretical until recently without allowing productive satellite designs that can take a productive role in a space mission. However, in the 2010s, first proof-of-concepts and practical applications have emerged [37]. FemtoSats usually consist of a single PCB using wireless energy harvesting or carrying a single solar cell on one side of the PCB, and electronics on the other [38]. With the emergence of more advanced energy harvesting and battery technologies in the future and an increasing level of semiconductor miniaturization, the basic character of FemtoSats could therefore change. Future FemtoSats will therefore find new niche use-cases, for which these lightest, cheapest, and expendable spacecraft will be optimal.

2.2 Early CubeSat Reliability and Motivation

Miniaturized satellite design is driven by the principle of designing a "good enough" spacecraft to do a job. Most Nanosatellites utilize COTS microcontrollers and application processor SoCs, FPGAs, and combinations thereof [39–41]. These components can offer one to two orders of magnitude more processing performance, are equipped with up to three orders of magnitude more memory, and an abundance of non-volatile storage capacity in comparison to classical space-proprietary components intended for larger satellites, while requiring less energy. Therefore, even a 5kg CubeSats can support a broad variety of commercial payloads and sophisticated scientific instruments, if these can be be fit into a smaller satellite chassis.

However, miniaturized satellites suffer from lower reliability, which discourages their use in long or critical missions, and for high-priority science. Most nanosatellites launched in the first two decades of the 21st Century (until the time of writing) still experience failure within the first months of their missions [39]. As depicted in Figure 6, even in late 2018 satellite malfunctions and early mission failures are widespread. The First-MOVE CubeSat is also representative in this regard, and we will use it as a case study to showcase the problems that still plaque this field.

First-MOVE: A Case Study

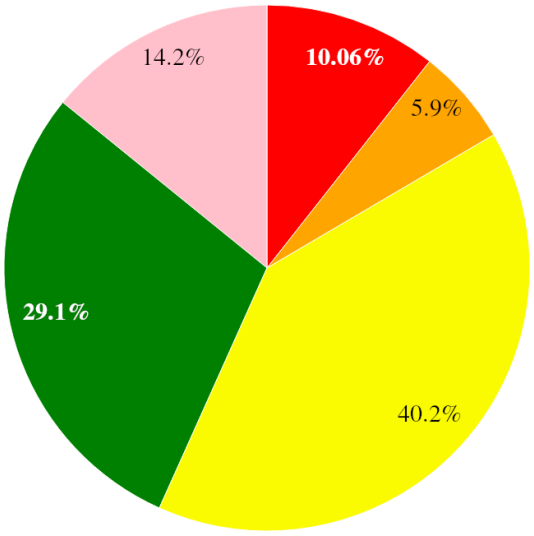
As a stereotypical late first-generation CubeSat, First-MOVE's design consisted of several microcontrollers. Its OBC was driven by a ARM926 based ATMEL microprocessor, utilized SDRAM, MRAM and NAND-flash memory, and is overall similar to a contemporary embedded device or smartphone. This fragile system architecture is representative for an entire generation of CubeSats built at that time.

At the time First-MOVE was designed little information was available on which components were expected to perform well in space, and which were likely to fail early on. During the actual construction phase, considerable information on these aspects became available continuously, and so its OBC was adjusted and retrofitted several times. E.g., the introduction MRAM was a retrofit to the original NAND-flash based design, as commercial MRAM was discovered to perform well aboard several earlier first-generation CubeSats. Further information on this First-MOVE's OBC is available in [Fuchs17].

First-MOVE successfully conducted its mission in LEO for two months after launch.

Towards the end of the mission, the OBC began to experience random reboots, which gradually increased over time. As of early 2014, the satellite could no longer be commandeered, and the mission was declared over. Both the funding organization (the german space agency DLR) and the CubeSat community considered the satellite performance and lifetime positive, and as the overall survival rates for CubeSat at that time were very low.

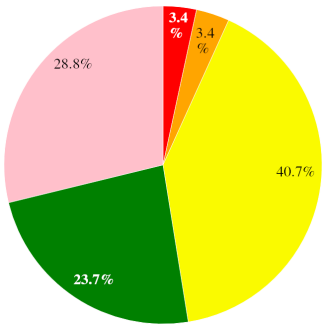
Subsequently, a team of three researchers, one of them being the author of this thesis, conducted a formal review of the First-MOVE project [Fuchs17]. This showed that if First-MOVE's system architecture had been fault-tolerant, the satellite could potentially have been recovered to a safe state. Otherwise, only minor organization issues related to the special setting of academic environments, which is a widespread prob-



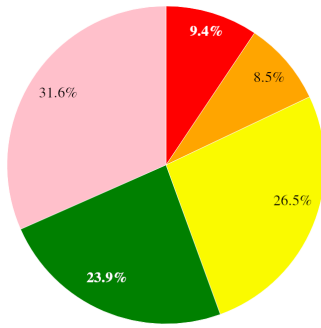
Full Mission Success
Partial Mission Success
Early Failure
Dead on Arrival
No Data/Unknown

Documented Launches	
Industry:	
Individual	59
Constellation	435
Professionals	234
University	223
Total	951

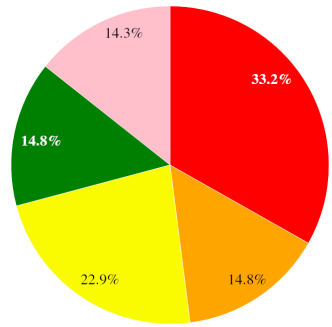
(a) CubeSat Mission Success



(b) Space Industry



(c) Professionals



(d) University & co.

Figure 6: CubeSat Mission success and failure for the time span 2000 to 2018. Bottom 3 charts show only data for individual CubeSats without satellites in constellations and swarms due to data quality reasons. It is reasonable to assume that developers of unsuccessful CubeSat missions also choose to not share information about the status of their satellites.

Image Credit: Charts produced through the CubeSat Database by Swartwout M. [42]. Military and other sensitive missions are often not publicly documented.

lem in academic satellite and instrumentation projects. A majority of first-generation Nanosatellite failures back then [43] could be attributed to design issues and manufacturing flaws due to developer inexperience (e.g., negative power budgets or dysfunctional communication channels) [39]. At the time of writing, failures caused by inexperience and design flaws have reduced drastically due to project professionalization and an increased staff of full-time developers in small-scale professional projects and academia.

2.3 Nanosatellites Today and Legitimization

Development on a second satellite, MOVE-II, began in late 2014 and the finished flight model is depicted in Figure 7. Since work on First-MOVE began in 2006, miniaturized satellite development has professionalized and fewer satellites fail due to practical design problems. Instead, the main source of failure aboard CubeSats today are environmental effects encountered in the space environment: radiation, thermal stress, and launch issues [2].

Mission result data shows that technological limitations are the main limiting factor regarding miniaturized satellite reliability at the end of 2018. Figure 6 shows that even experienced, traditional space industry actors who design such satellites "by the book" with quasi-infinite budgets struggle to reach 30% mission success. This lack of reliability and brief mission lifetimes curtails miniaturized satellite usage for critical and long-term space missions, as well as for high-priority science missions for solar system exploration, deep-space probes, and space observatories. During development

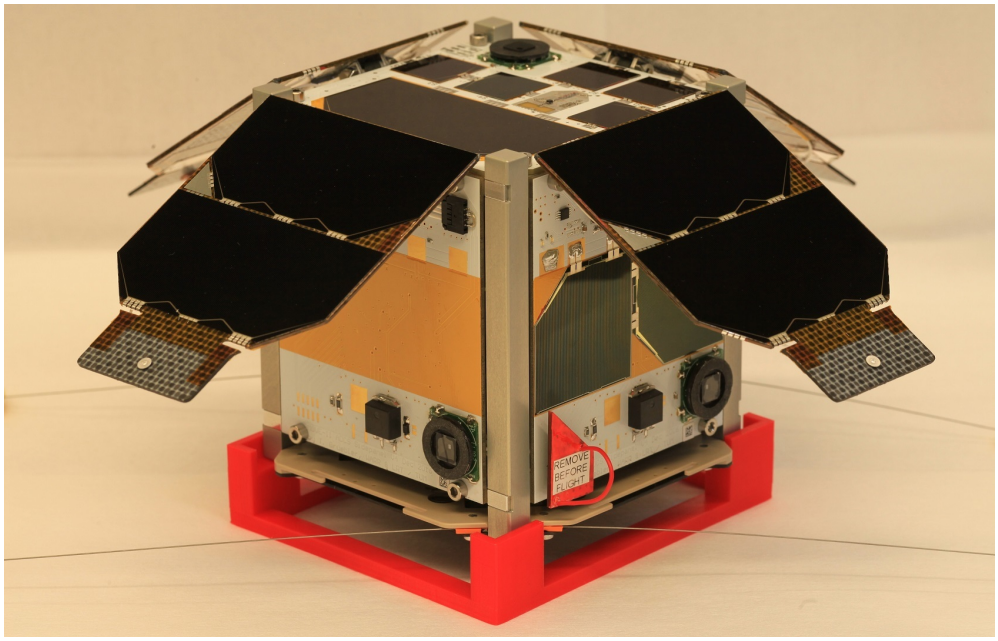


Figure 7: The MOVE-II CubeSat, which was part of the author's master thesis research and the design challenges faced during development initiated the research in this thesis.

of MOVE-II, it became clear to us as spacecraft designers that there were simply no fault-tolerant OBC solutions that could be used to achieve a more reliable satellite design within the constraints of a CubeSat.

Fault-tolerant computer design for spacecraft still relies upon radiation tolerant special purpose hardware These designs primarily rely upon proprietary fault-tolerant chip designs manufactured in technology nodes with a large feature size (radiation-hardening by design – RHBD) [44] and specialized manufacturing techniques and materials (radiation-hardening by manufacturing and process – RHBM/RHBP) [45]. Often, both of these techniques are combined and a RHBD chip design is manufactured in a RHBD process based with much more coarse feature size than commercial technology. Due to the lower energy efficiency and larger size of and greater distance between transistors, as well as less refined electrical properties, these components also require more energy, and offer less compute power compared to consumer hardware due to decreased clock frequencies and smaller memory sizes.

The use of traditional RHBM/RHBD components at the time of writing is limited to the civilian and military atmospheric aerospace industries, laboratory instrumentation for very large particle experiments run by well funded organizations (e.g., particle accelerators, radiation-testing sites) and traditional space-industry applications in long-term projects where cost considerations are not of primary concern. Especially in nanosatellites, the energy consumption, physical size, and cost of these components are prohibitive, making their use technically impossible and usually uneconomical. Therefore, nanosatellite computing has historically taken two paths: very simple onboard computers (OBCs) based on one single or few microcontrollers and very complex custom-tailored systems. This approach works to a certain extent, as there are a handful of COTS microcontrollers which are designed and manufactured in a way so that they unexpectedly turned out to be radiation hard (radiation-hard by serendipity – RHBS) [46].

At the time of writing, sophisticated fault tolerance capabilities are still absent in Nanosatellites. Instead CubeSat designers try to mitigate faults at the system level using custom mitigation circuitry [47], and thereby achieve "workarounds" to still somehow handle faults encountered in the space environment. The practical effect of this lack of viable fault tolerance techniques and the use of workarounds is reflected in the mission success statistics for miniaturized satellites depicted in Figure 6. However, a few CubeSats have also operated successfully in space for a decade or longer [48]. In practice, this shows that there is no hard technological limitation that would prevent the use of COTS technology in satellite missions with a much longer duration.

Many issues in other fields of spacecraft design can be overcome through engineering-based solutions. Such solutions work well, e.g., for addressing resonance issues, assuring a suitable thermal design and heat-distribution, and for deployable mechanical structures. Engineers therefore attempted to solve the lack of reliability of CubeSats similarly, by constructing custom fault tolerance computer design through component-level redundancy with commodity components. Practical flight results showed that such designs are fragile due to high complexity [39, 49], and tend to perform worse than much simpler designs without fault tolerance capabilities.

Today, nanosatellite designers have to forego fault tolerance in the hope of minimizing failure potential and thereby meeting satellite lifetime requirements for a given space missions by chance [50]. Designers are aware that such satellites may fail at any given point in time during a mission.

Figure 8: The launch of MOVE-II aboard SpaceX SSO-A: SmallSat Express on December 3rd, 2018 from Vandenberg Air Force Base, USA.

Image source: SpaceX SSO-A press material for public use.

MOVE-II was launched into LEO on December 3rd, 2018 with Space-X "SSO-A: SmallSat Express" (depicted in 8), where it operates successfully until at the time of writing this thesis. It utilizes only a few basic fault tolerance techniques that were available in commodity embedded components and COTS CubeSat subsystems. Its overall system architecture is still not fault-tolerant. Risk acceptance at this level is a viable approach only for educational, and uncritical, low-priority missions with brief duration. To construct future, more reliable miniaturized satellites, a robust, fault tolerance on-board computer architecture is needed. However, such an architecture do not exist yet, and with the research in this thesis I intend to change that.

2.4 Fault-Tolerant Computer Architecture

Fault tolerance in the most abstract sense, implies the capability of a system to overcome and gracefully handle failures. It is crucial for satellite computer design and a practical necessity to assure reliable operation of a satellite computer during space missions with an extended duration. As described in the previous section, the lack of such functionality within contemporary miniaturized satellites has become a major constraint to increase adoption of these spacecraft.

Fault-tolerant computer architecture, which is discussed briefly in this section, covers only a small part the entire field of fault tolerance and reliability engineering. Among others, systems can be designed to tolerate human error [51] and external attacks, which would require the discussion of aspects of psychology and human interface

design. In the remainder of this section, we discuss fault tolerance modes, measures, and testing from the perspective of computer architecture for spaceflight applications to provide the necessary background for this thesis. A more complete look on the different aspects and sub-fields of fault tolerance are available in literature, e.g., in [52].

Considering fault-tolerant computer architecture, the faults we must protect a system from depend on the application, the environment it operates in, as well as practical operating conditions (e.g., temperature and system load). Besides that, faults can occur due to technological wear and aging, and sometimes by chance. Many protective measures can be used to achieve fault tolerance for computer systems [53,54]. Often, the practical purpose for the application of these techniques is often not fault tolerance itself, but the need to increase scalability [55,56], manufacturing yield [57,58], higher clock frequencies and data throughput [59–61].

Different industries apply different fault tolerance techniques due to a variety of practical reasons, and today often maintain their own, proprietary implementations to tackle their domain-specific challenges. For proprietary fault tolerance implementations in different industrial applications, there is usually no immediate incentive to share and generalize such fault tolerance techniques by themselves, unless they can be patented, commercialized, and thereby protected [62]. This gap in turn is covered by scientists and researchers in industry and academia.

Today there is an entire field of science that tries to generalize application specific fault tolerance techniques, to produce new fault tolerance concepts through recombination. Unfortunately, this recombination is often done without considering the original application and its boundary conditions. As we show in Chapters 4 and 6, academic research and publications covering this topic are kept very abstract and do not consider a specific real-world application anymore. This works well for certain fields of science and even some fault tolerance topics[2]. However, for practical applications to system-architecture this is not the case, as generic solutions without proper boundary conditions and a realistic fault profile, can usually not be applied anymore to a real system. Today, academic fault tolerance research has produced a vast amount of publications and generated many theoretical concepts. But, only a handful of fault tolerance concepts envisioned by academic fault tolerance research have been implemented and tested in practice, and most have been ignored entirely by the industry. One could argue that this is the way science works, but knowingly publishing invalid and research without validation can also be seen as dishonest and only hinders publication of actually valuable research.

The path to validate such concepts is long, time-consuming, costly, and requires large amounts of engineering work [64–66]. The obtained validation results are often not considered publishable by academics, as they require a high degree of labor just to achieve one brief paper, while multiple theoretical journal publications could be produced in their stead. Industrial users are aware of such research [67], but are often skeptical. In the space industry, for example, concerns regarding validity, testability, verifyability and a perceived general lack of maturity of academic research has caused an entire industry to conservatively use very old technology [1].

When designing fault-tolerant systems, we must consider an application's operating environment, its fault profile, and system design constraints [68]. Generic fault tol-

[2]E.g.: erasure codes and performance overhead calculations to achieve quality of service under faults [63] can largely be discussed without a specific application in mind, as long as key parameters match.

erance concepts can serve as building blocks to design a comprehensive fault-tolerant architecture, assuming they are validated in a realistic manner.

2.4.1 Terminology and Fault Tolerance Objectives

Today, scientists and engineers use the terms ECC, EDAC, FDIR, and error correction almost interchangeably, while reliability, redundancy, fault tolerance, and robustness are surrounded by a shroud of marketing. In practice, error detection and correction (EDAC), fault-detection, isolation, and recovery (FDIR), redundancy, and failover all are distinct tools. They can be applied to achieve different kinds of fault tolerance, e.g., computational correctness, continuous non-stop operation, failover, and simple error correction.

Error detection and correction (EDAC) implementations usually utilize one or multiple erasure codes [69] to implement error correction coding (ECC), which allows errors in stored and transmitted data to be corrected. EDAC is efficient only for protecting the integrity of frequently access data, and may do so passively in the background without requiring a computer system to actively handle a fault in software. These limitations can be mitigated only in combination with other design measures such as error scrubbing and by generating error syndromes to notify the system about a fault [70].

FDIR instead assures that a fault-induced error is not just detected and corrected, but also that side-effects are isolated and resolved (e.g. discussed in [71] for space applications). In contrast, in case EDAC logic encounters errors when decoding data, it may inform the system about the result through an ECC syndrome and corrects data passing through. FDIR does not necessarily imply computation correctness, usually utilizes fault tolerance measures to achieve error detection and correction, but otherwise implies only that a fault is corrected and the system is restored to a working state.

Fail-over, in contrast, can be implemented as one-shot measure, e.g., with simple redundancy as discussed in [72], by falling from a primary to a secondary system instance and do not have to assess correctness, but only need be capable to detect faults. One of the most common applications for this approach is RAID1 with 2 memories or disks [72], but similar applications exist for avionics and network architecture in spaceflight and atmospheric aerospace applications [73].

2.4.2 Fault Detection and Correctness

To facilitate fault detection, we can exploit algorithmic measures as well as result comparison achieved through component replication (spatial redundancy) or repeat-execution (temporal redundancy). With algorithmic approaches detected errors can be reconstructed using parity data (informational redundancy) information, or by utilizing an alternative result generated through spatial or temporal redundancy. We refer to this type or error correction as forward error correction (FEC) [74]. Alternatively, backwards error correction (BEC) can be achieved with temporal redundancy and algorithmic measures, and implies message retransmission or re-execution of a failed operations [75].

Algorithmic Fault Detection and Informational Redundancy

The algorithmic approach exploits an inherent property of a system to detect faults. It can only be used if there is an inherent property in a system or protected data that can be used to judge the occurrence of a fault [76, 77]. Fault detection then does not imply the ability of the system to determine a correct result, but only the ability to asses if the protected data or system is faulty.

Algorithmic fault detection often exploits informational redundancy, but it may also use other inherent mathematical properties of data or logic-design properties of a system [77]. To a limited extent, algorithmic fault-detection can also be used to protect a program's data and control flow, e.g., by computing or modifying checksums for each executed instruction passing through a CPU's pipeline [78]. However, this requires a non-standard processor pipeline [79], a custom compiler toolchain [80], and therefore is feasible only for embedded software with a very specific structure.

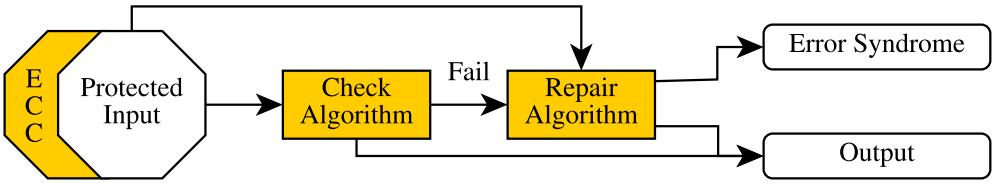


Figure 9: An example of algorithmic redundancy where extra algorithmic information is indicated separately as ECC. This extra information could also be an inherent property of the input data, instead of separate.

Spatial Redundancy

When utilizing spatial redundancy, we can realize fault-detection by comparing the output of multiple redundantly implemented system modules or equivalent but differently implemented variants of a subsystem run in parallel. Spatial redundancy can be implemented at all scales: for individual transistors and circuits, sets of logic, logic blocks, IP-cores, IP-core groups, ICs, components, to even an entire computer. At

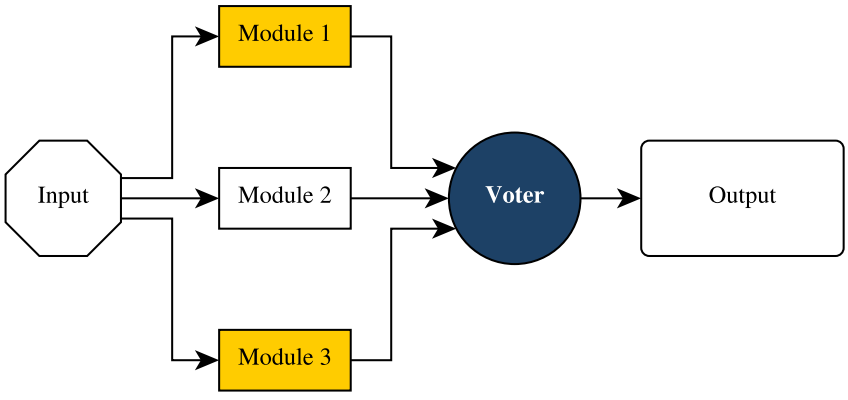


Figure 10: An example of spatial redundancy with 3 replicated modules in a TMR setup.

different scales spatial redundancy will offer different protective properties and different a level of scalability. A simple implementation of spatial redundancy requires replication of the actually protected modules as depicted in Figure 10.

We refer to systems consisting of N modules collectively as NMR systems. With 2 redundant modules, we can detect faults through supervision or in conjunction with a watchdog, to which we refer to as dual modular redundancy (DMR). We can determine correctness through a simple majority vote, for which at least 3 modules are needed (triple modular redundancy – TMR). These systems can be scaled up to realize $2k+1$ redundancy, as an odd-number of modules is needed to avoid a draw during voting. With more than 3 modules, more sophisticated voting concepts can be realized which then do no longer require a centralized and guaranteed-correct voting oracle [81] or allow a distributed majority decision [82]. An NMR systems can also be outfitted with spare modules to handle multiple subsequent transient faults or permanent faults.

Temporal Redundancy

Temporal redundancy implies re-execution of an operation multiple times in sequence, and example of which is depicted in Figure 11. Like in spatial redundancy, this is often done in $2k+1$ setups to assure error correction. This favors checkpoint implementation in software and the use of software diversity [83]. It is suitable for protecting applications where failed results can be discarded, individual operations can be repeated, or where an application as a whole can be restarted in a side-effect free manner. For most control systems and software running on general purpose computers, however, this is not the case.

The use of temporal redundancy introduces a degree non-determinism to an applications application, which can conflict with a requirements for real-time guarantees [84]. Thus, temporal redundancy concepts usually can only be applied to real-time systems unless the protected software implements a very specific structure [85]. Protection of applications at the scale of an operating systems, and programs with a complex program state or structure may incur a high performance overhead [86]. Due to the time-dependent nature of temporal redundancy, this form of redundancy is vulnerable to faults occurring in bursts or groups [87].

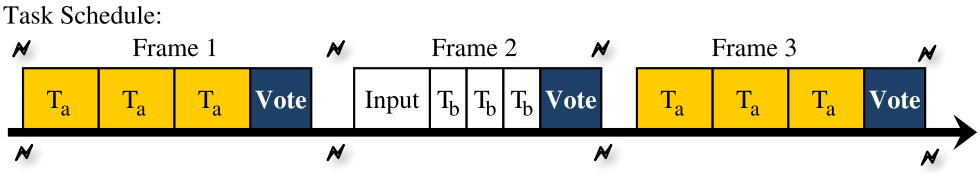


Figure 11: Task schedule of a temporal redundancy where every scheduled task (T_n) is executed 3 times with majority voting.

Fault Detection Granularity

The granularity and frequency for performing voting in spatial and temporal redundancy, as well as erasure coding parameters should be chosen based on the expected fault model and environmental conditions.

Most systems implementing spatial redundancy in use today implement instruction or clock-cycle bound lockstep for processor cores or larger system components [54]. This allows rapid error detection and correction without requiring the software or software to actively participate in fault handling [88]. Usually, the voter logic is combined with state-synchronization logic, to assure that all modules in a redundant set utilize the same input data. For more sophisticated computer designs, the level of complexity necessary to realize voting and state synchronization in hardware is non-trivial. Thus, such systems are limited to low clock frequencies than conventional designs [54].

As with temporal redundancy, we can also utilize software to realize lockstep functionality in spatial redundancy using checkpoints triggered through scheduling [89], or an external signal [90]. As we show in this thesis, lockstep-concepts implemented in software can enable more powerful dynamic, and runtime-configurable voting in conjunction with spatial redundancy to achieve FEC.

2.4.3 Effect Isolation

To achieve side-effect-freeness, the effect induced by a faults must be isolated, so they can not propagate within the rest of the system at large. However, the scope and way in which fault isolation can be implemented depends on the fault-detection measure, on the protected component, the high-level system architecture, as well as on the specific application scenario. For pure software-based measures utilizing temporal redundancy, this can be achieved by buffering results [91] and outputting a correct result after correctness has been assured.

Not all fault-tolerant systems require fault-isolation. The emission of incorrect data due to a fault can also be mitigated through a system architecture and instruction-set means [92], topological measures [93], or network-side [94]. Hence, a computer operating in such an environment does not have to be equipped with fault-isolation properties, as the overall system setup can already guarantee fault isolation.

2.4.4 Fault Recovery

In conjunction with or subsequent to effect isolation, the effects of a fault induced into a system should be resolved to prevent bit-rot and voter degradation due to transient faults [95]. It also reduces the need for over-provisioning redundant instances and parity data. For data storage, this can be achieved through parity in RAID- [72] or RAIF-like [96] systems, which can again be combined well with erasure coding [97]. It can make a system more robust especially if it has to operate for extended periods of time, or without maintenance.

Fault-recovery capabilities, thus, are not necessary for all applications, and may sometimes even be undesirable. For applications where maintenance can be performed frequently and the failure probability is low, simpler failover implementations can be of advantage since they are simpler, and therefore have a reduced failure potential. Examples include atmospheric aerospace applications for civilian use [98] or marine shipping [99]. This can allow a component to be implemented with lower complexity, thereby reducing overall failure potential, cost, and weight.

Depending on application requirements and if service interruption is acceptable, hot, cold, or warm [100] stand-by can be used to achieve failover [101]. Hot redundancy requires at least one redundant module executing in parallel to the primary module, to

allow the system to switch to failover without service interruption. Warm redundancy just implies a second module to be in standby mode, e.g., so it can rapidly take over operation by loading a correct application state. With cold redundancy, a redundant module is kept available but inactive, and has to be brought up when needed. This can allow energy saving and reduce wear in redundant module, but implies a time delay until regular operation can resume. In this thesis, we utilize warm standby when migrating applications from a permanently failed processor core to a new location.

Fault Recovery with Temporal Redundancy

In systems utilizing temporal redundancy to achieve backwards error correction, the generated incorrect application state of a failed operation has to be reverted. As temporal redundancy implementations usually require operations to be isolated or self-contained already, no further steps beyond discarding faulty data are necessary. By design, changes in the operating system state due to faults in temporal redundancy protected software will in practice be detected and subsequently not propagated.

Fault Recovery with Informational Redundancy

With informational redundancy, data containing a fault should be corrected and rewritten. In most memory-access based EDAC implementations, this step has to be performed independently from error correct, e.g., in software by an ECC syndrome or in hardware suitable error scrubber logic. In case of non-correctable erasure coding errors, or if backward error correction is used, data or a messages have to be retransmitted or rewritten. In memory-access based EDAC systems, non-correctable ECC errors can only be resolved with more redundancy and additional parity information, or through replacement and blacklisting.

Composite erasure coding systems combine multiple layers of erasure codes, to achieve the advantages of multiple different types of codes or parameter configurations [102]. These enable us to achieve overall stronger protection and mitigate weaknesses of individual erasure codes, e.g., symbol based block-codes are vulnerable to single bit-rot degrading their performance [103]. We describe the practical implementation of a composite erasure coding system combined with RAID-like features in Chapter 7.

Fault Recovery with Spatial Redundancy

In systems exploiting spatial redundancy, a fault may cause a failure of a redundant module, resulting in redundant system to become degraded.

To recover from transient faults, a failed module can be recovered using data from another module [104]. For voters replicating processor cores or larger system structures, this can be done with or without performing a reboot. For some cases, just copying the application or software state from a healthy module is insufficient, requiring a reboot to recover from a transient fault.

Conventional semiconductors affected by permanent faults can become dysfunctional, or may ceasing to function completely. To allow a system to tolerate additional, subsequent faults, additional spare modules are needed. We refer to this measure as over-provisioning. In practice, this can lead to large and very complex voter designs with high energy usage and large logic footprint [54]. With ASICs, the need for over-provisioning can only be alleviated through hardened manufacturing,

which is expensive [44]. This approach today is widely used in spaceflight applications to reduce the impact of transient and permanent faults. By design, such systems still become defunct once no further spare resources are available and a fault has occurred in system with only two intact modules.

Programmable logic devices such as FPGAs allow more refined permanent fault handling: permanent faults in the FPGA fabric can be mitigated by utilizing a configuration variant where no functionality-critical logic is placed in defective regions [105]. This can be used to restore a redundant module to a functional state. In practice, this approach can be exploited to allow a system to age gracefully by adapting to accumulating permanent faults over time, instead of failing spontaneously.

2.4.5 Fault Tolerance in the Real-World

Individual fault tolerance measures can be combined, allowing a vast amount of possible combinations. However, not all possible combinations are effective and efficient for protecting a system operating in a specific application environment and threat profile [106]. Certain combinations can even reduce reliability, or cause an increased failure potential [107]. However, if done right, fault tolerance measures deployed systematically in appropriate locations across a system [108], can allow for certain a defense-in-depth effect [109, 110].

Many fault-tolerant systems in use today are meant to isolate and recover from faults within the bounds of what their design constraints specified. However, this means that most fault-tolerant systems are not actually tolerant to faults, but that they are systems that can not fail so long as faults adhere to the specifications and "obey the rules set by the designer." In practical system design, these systems are then instead often treated not as robust and reliable, but as infallible systems that always work correctly and do not malfunction [111].

Validating Fault Tolerance Measures

To assess the effectiveness and strength of a fault tolerance architecture for a specific application, it must be validated in a realistic setup with a representative fault profile [112]. Such a profile is not just a statistical distribution over time, but should consider the impact of all relevant expected fault types (transient, intermittent, and permanent).

A variety of different test methods are available to analyze fault tolerance measures implemented at different scales and levels in hardware, in software, and both [52]. Historically, these methods included fault injection into hardware and software at different scales [65, 66], circuit simulation [64], mathematical correctness-proofs [113], statistical modeling [114], and even prototype experimentation for technology validation in a representative environment [115]. However, mathematical and logical proofs for modern processor based computer systems are non-trivial [116] and have been done only for individual algorithms, simple software, protocol state machines, and for simple circuits [113], but not for complex, OS-scale applications.

However, properly testing and validating software- and hardware-implemented fault tolerance measures is not trivial, requiring considerable time and development effort. Due to these challenges practical applications in industry tend to rely upon just a few widely used standard measures and combinations thereof, and disregard science.

Applied Fault Tolerance

Memory-access based EDAC through ECC is widely used in critical and always-on applications [117] due to its scalability, simplicity and low cost [118]. Due to technology scaling effects, technological reasons, and for the sake of yield enhancement, it has also become increasingly popular in consumer products [119]. All popular conventional high-speed interface and connector standards such as USB3 [120], SATA [121], Ethernet [122], and PCIexpress [123] rely upon powerful erasure coding systems to achieve high clock frequencies on serial channels [124]. Traditionally, ECC has been applied widely to protect non-volatile data storage solutions (e.g., nvRAM, memory cards) [125]. However, to increase yield in microfabrication, ECC has become common also to protect on-chip memories with a short data lifetime such as BlockRAM, caches, registers and the various scratchpad memories [126]. Designing systems for high-performance computing or critical applications without it would be impossible without erasure coding.

Today, most space-borne systems rely strongly upon spatial redundancy [54]. Most such systems rely upon hardware-voting, and only since the turn of the century has there been an increasing drive to realize FDIR functionality in software [127, 128] and using network topology and functionality [94]. This is an ongoing development, and this thesis should be read in context of this shift from traditional hardware to software and co-designed fault tolerance concepts [129]. Software-implemented fault tolerance concepts, however, have existed since the emergence of mainframes [130]. Even for space applications, they identified as promising already in the early days of microcomputers [131], but it was considered technically infeasible and inefficient until recently.

Technological Evolution and Heritage

The high stakes involved in operating critical systems in different fields, encourages the use of old and less efficient, but well understood architectures instead of more modern, and more powerful ones [54]. Hence, different industries progressed in developing fault tolerance concepts at different paces. While some innovated rapidly to achieve functional systems (e.g., the industrial and high-performance computing market, and the new space industry), others try to maintain a balance between old and new (e.g., automotive and medical embedded applications). Some chose to remain very conservative, preferring to re-use decades old concepts at extreme cost over using cheaper but more novel designs (e.g., the traditional space industry [54, 104, 132]).

Ultimately, however, all of industries are pressed hard to innovate, as technology progresses. An illustration of this need to innovate is the beginning adoption of the CAN bus standard [55], which was widely used by the automotive industry. The traditional space industry has just begun to adopt this standard few years ago and will benefit from its advantages over older standards considerably, though the interface and protocol are is currently being replaced in automotive industry by Flexray [56] and the use of high-speed computer network standards such as Ethernet [73].

However, the risky but fast-paced transfer of cutting edge technology from the embedded- and mobile market to spaceflight has resulted in the emergence of an entirely different, “new space industry”. Relevant industrial players try hard to utilize modern technology which can enable innovative space mission concepts that were completely unrealistic and often unimaginable just a few years ago. To do so, this industry

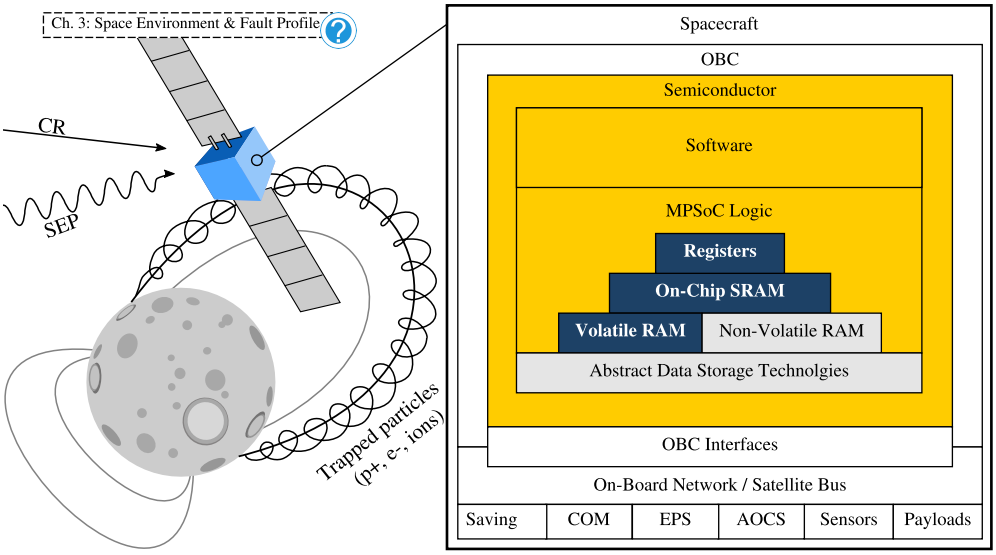
accepts an increased level of risk for failure. At the time of writing, the reduced cost of this engineering approach and the thereby produced designed spacecraft designs has succeeded and left a mark on the industry as a whole.

Chapter 3

The Space Environment

Physical Fault Profile and Operational Considerations

A satellite's on-board computer has to cope with unique challenges which on the ground are only encountered in irradiated environments such in proximity of a nuclear reactor. Hence, for the understanding of the fault profile and application constraints for this thesis, in this chapter we provide an in-depth discussion of our operating environment and its effects on a satellite's on-board computer. We discuss the physical design restrictions aboard spacecraft, and operational considerations. Most importantly we discuss the impact of radiation on semiconductors, and how it can be mitigated.



3.1 The Impact of the Space Environment on Electronics

Space is a challenging environment for electronics to operate in, and its fault profile differs from that of most ground-based applications. A system engineer and computer architect has to consider many different design challenges and a very special fault profile. Only then is it possible to develop a reliable computerized system suitable for operation in this environment for an extended period of time.

3.1.1 Radiation Effects

Radiation is the main cause of faults in electronics aboard a satellite due to defects caused by electro-static discharge effects (ESD) [133] and directly inflicted particle-damage. About 20% of all anomalies [134] aboard satellites can be attributed directly to high-energy particles, with the share of faults radiation-induced faults in electronics-heavy subsystems increasing drastically. This makes sense considering that the semiconductors used in electronics-heavy and computerized subsystems are more vulnerable to radiation-induced faults than, e.g., deployable structural elements (DE/STR) or a solar cell, whose performance will degrade slowly over time due to radiation.

A satellites on-board computer (OBC), communication transceivers (COM), and the electrical power system (EPS[1]), as well as its attitude determination and control or orbit control system (ADCS/AOCS) all consist of microntroller- or processor SoCs with a varying set of peripheral electronics attached. A majority of all faults aboard CubeSats can be traced back to the failure of these architecturally similar subsystems [2] even directly after launch. Statistics from [2] are depicted in Figure 12. The low

[1]Responsible for battery charging and health control, as well as power management and distribution across a spacecraft.

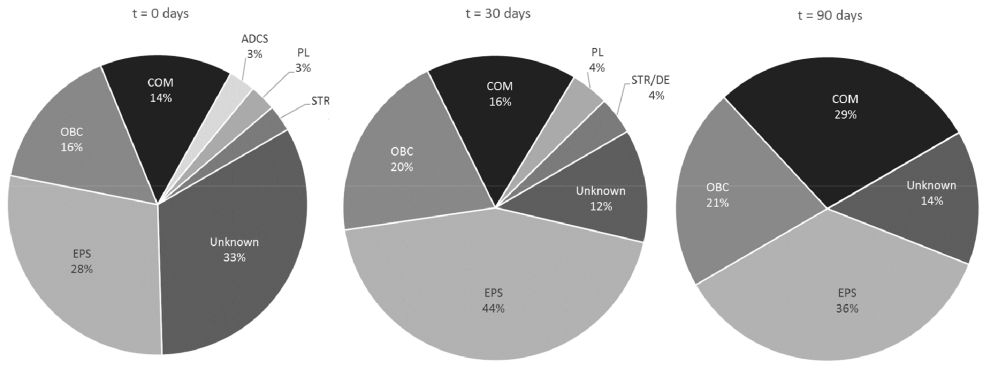


Figure 12: Failures sources aboard CubeSat in 2016 after deployment, and after 30 and 90 days from [2]. Upon deployment, 61% of failures can be traced to strongly computerized subsystems. From a computer architecture perspective, all these subsystems are based on the same kind of components: non-fault-tolerant microcontrollers and mobile-market SoCs. After 90 days, 86% of all failures of CubeSat can be attributed to failures in the indicated subsystems. The base of data used by Langer et al. only serves as tentative indicator, as not all CubeSat and especially commercial operators choose to share this information.

Image Credit: Langer et al. [2].

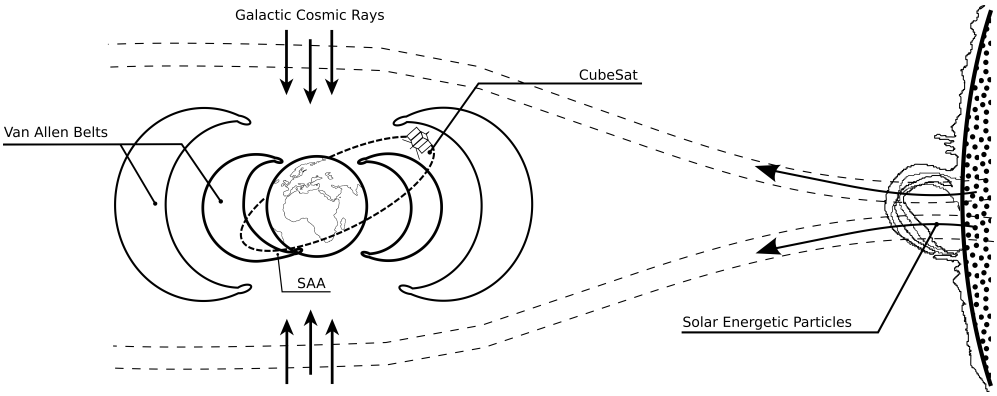


Figure 13: A visualization of the three main natural sources of radiation affecting spacecrafts.

number of satellite failures due to payload (PL), or ADCS malfunctions in Figure 12 can be attributed to the fact that the failure of this subsystem seldom causes a satellite be lost entirely. Instead ADCS or AOCS failure will prevent certain mission objectives from being accomplished, the effects of which fail in the *early failure* and *partial mission success* categories as defined in Swartwout's CubeSat Database [42]. See also Figure 6.

Highly charged particles originate from a variety of different sources, which are depicted in Figure 13. They travel spinning around the Earth's magnetic field-lines in the Van Allen belts, are ejected by the Sun during Solar Particle Events, or arrive as Cosmic Rays from beyond our solar system. Galactic cosmic rays from beyond our solar system are mostly protons [5, 135], whereas various other high-energy particles are ejected by the Sun during solar particle events (*proton storm*). The radiation environment near the Earth, as well as in the rest of the solar system changes dynamically over time. We refer to this as *space weather*.

Depending on the orbit of the spacecraft and the occurrence of solar particle events, an OBC will be penetrated by a mixture of high-energy protons, electrons and heavy ions. In LEO, the residual atmosphere and Earth's magnetic field provide some protection from radiation, but this absorption effect diminishes quickly with altitude. Hence, microelectronics are exposed to a mix of highly charged particles, with flux density depending on solar activity and the spacecraft's attitude.

These particles can corrupt logical operations, induce bit-flips within data-storage cells (Single Event Upset – SEU) and connecting circuitry, or induce a latch-up. They can also cause displacement damage (DD), molecular changes in a chip substrate's crystalline structure which can cause its electrical properties to change, potentially causing permanent malfunctions. The particle flux will be increased while transiting the South Atlantic Anomaly (SAA), which is also depicted in Figure 13 [136]. Earth's magnetic field experiences a local, height-dependent dip within the SAA, due to the offset of the spin axis from the magnetic axis. In this region, a satellite and its electronics will experience an increase of proton flux of up to 10^4 times (energies > 30 MeV) [5]. This flux increase results in a rapid growth of bit errors and other upsets in a satellite's OBC.

Radiation challenges OBC fault coverage constantly throughout a mission and

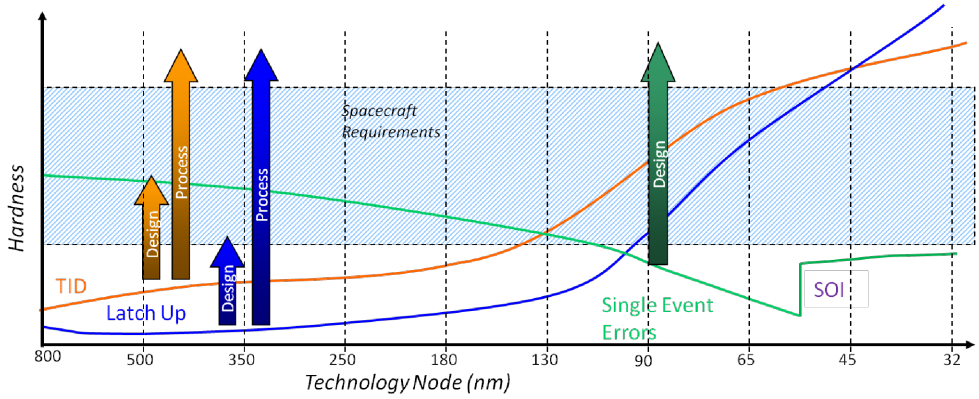


Figure 14: The impact of radiation on a semiconductor varies depending on the used manufacturing technology. Manufacturing in fine technology nodes such as FinFET reduce the overall likelihood to experience radiation faults that affect critical logic due to shrinking geometry and therefore a reduced footprint of vulnerable logic. COTS Techniques such as FD-SoI furthermore increase SEE performance. Therefore a combination of small feature size manufacturing and robust COTS manufacturing in conjunction with software measures can offer strong fault tolerance capacity.

Image Credit: [138], Boeing/US-DTRA, for public use.

affects all of an OBC's components. The impact of radiation on different microfabrication processes, substrates, and memory technologies varies, as depicted in Figure 14. In general, electronics with a large feature size are more resilient to radiation-induced single event effects (SEEs) than those manufactured in finer production nodes. Chips with a small feature size are more susceptible to multi-bit upsets (MBU), that can propagate within circuits corrupting larger circuits or memory cells. Radiation events can also cause Single Event Functional Interrupts (SEFIs). These can affect sets of circuits, individual interfaces, or even entire chips. The cumulative effect of charge trapping in the oxide of electronic devices (total ionizing dose – TID) further impacts the lifetime of an OBC. Other types of radiation-induced faults, the destructive ones being the most relevant, are well described in [137].

As depicted in Figure 14, the robustness of a semiconductors in regards to different types of radiation-induced faults varies as well. Devices manufactured in old technology nodes with a coarse feature size show low TID (yellow line) and latch-up performance (blue), are robust to SEEs (green). Non-fault-tolerant semiconductors manufactured with old technology nodes are thus robust to SEE, while TID and latch-up performance has to be increased through radiation hardening. CubeSat developers attempt to apply this same approach at the system level with modern semiconductors in the range well below the 50nm scale. However, SEE performance worsens with shrinking feature size, and drops below an acceptable level with modern technology nodes developed after the early 2000s. For comparison, commercial chip manufacturing using 130 nm technology nodes began in 2001, whereas at the time of writing smartphones-SoCs are manufactured with technology nodes between 16nm and 7nm. CubeSats seek to apply the latter kind of technology due to their much superior performance, lower cost, excellent availability, mature development tools, and reduced energy consumption.

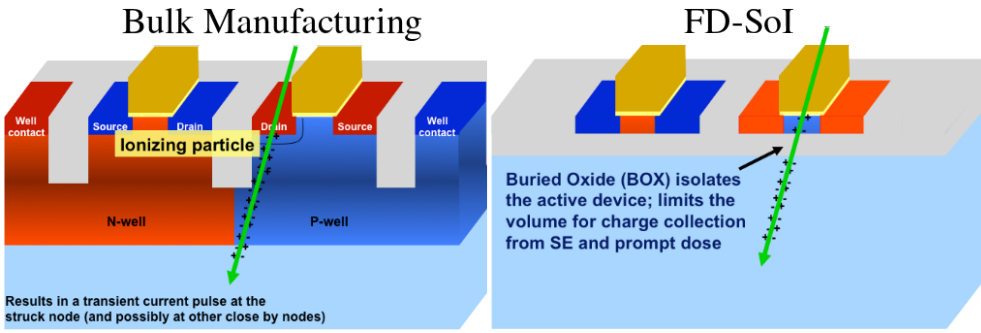


Figure 15: Modern manufacturing techniques such as FD-SoI show much better performance under radiation than traditionally processes. Originally, these were developed to reduce feature size and energy consumption to achieve increase semiconductor packing density. Regarding radiation, the reduced footprint and inherent isolating properties of these technology nodes implies a reduced likelihood for a particle to induce an effect. With FD-SoI specifically, the changed structural properties of thereby manufactured chips further reduce the impact of SEEs due to the introduced an isolating layer of oxide.

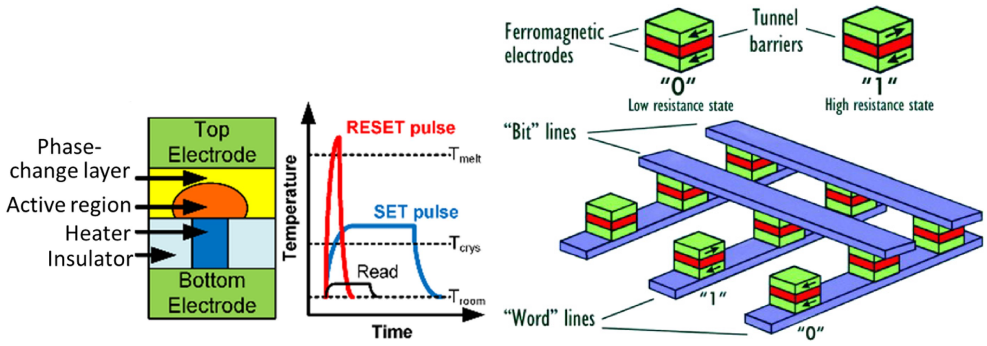
Image Credit: Alles et al. [141].

In general, the effects of SEEs and SEFIs can be both transient and permanent, while DD is always permanent [5]. In case permanent effects are induced, or faults occur in memory, radiation induced faults accumulate over time. The accumulative nature of permanent faults implies accelerated and often spontaneous ageing, which must be handled efficiently throughout the entire mission.

The increased impact of SEE on finer feature size chips also invalidates the naive approach of achieving better protection by adding more circuit-level protection. This prevents the continued application of traditional RHBD/RHBM concepts [104,132] to modern, high-performance embedded and mobile-market SoCs. The energy threshold above which SEEs induce transient faults in chips manufactured in fine technology nodes decreases, and the ratio of events inducing multi-bit upsets (MBU) or permanent faults increases.

Radiation tests with FinFET [139] and Fully Depleted Silicon On Insulator (FD-SoI) [140] based technology nodes also show improved SEE performance, contrary to projections based on technology scaling. As depicted in Figure 15, transistors in these technology nodes have a much reduced footprint as compared to bulk manufacturing. The smaller feature size there reduces the likelihood for a charged particle to interact with sensitive chip regions, which results in fewer but more severe upsets in such semiconductors [141]. FD-SoI introduces an additional layer of isolating oxide, which helps reduce reduce the impact of radiation effects on such a semiconductor. Hence, chips manufactured in certain new technology nodes, such as recent generation FPGAs [142] show better than expected TID [143] and latch-up performance [144], while also showing different SEE performance: fewer non-masked events with more severe impact.

In practice, radiation induced faults may corrupt computations of a computer, corrupt register contents, data stored in caches, main memory, and non-volatile memory. Memory mainly suffers from bit-rot and malfunctions in controller logic, and for volatile memory, these can well be compensated for using error correcting codes (ECC) combined with error scrubbing. Non-volatile memory also requires more powerful era-



(a) Phase-Change Memory (PCM – 3D-Xpoint) **(b)** Magnetoresistive RAM (MRAM)

Figure 16: The functional principles and structure of two of the currently most promising inherently radiation immune memories: PCM (a), and MRAM. Radiation immunity of these cells is based on that these memories do not store information as a charge, in contrast to radiation-susceptible DRAM, SRAM, or Flash.

Image Credit: (a) Hayat et al. [145] (b) Fert et al. [146].

sure coding systems, the basic notions of which also exist in latest-generation COTS flash memory based devices for ground use, as there galactic cosmic rays have become relevant sources for faults due to technology node scaling.

Functional interrupts can cause individual processor cores or other sub-units of a semiconductor to fail temporarily or permanently. Data can also be corrupted in transit, e.g. while being transferred or due to upsets in peripheral interface controllers. Hence, from a developer's perspective, to-be executed software and data can only be considered fault-free if it resides exclusively in radiation-hard memory and radiation-hard processing logic throughout. As this is not the case with all but trivial processing logic, no part of an OS can be relied upon to be fault-free, and concepts requiring such an entity do not offer effective fault coverage in the space environment.

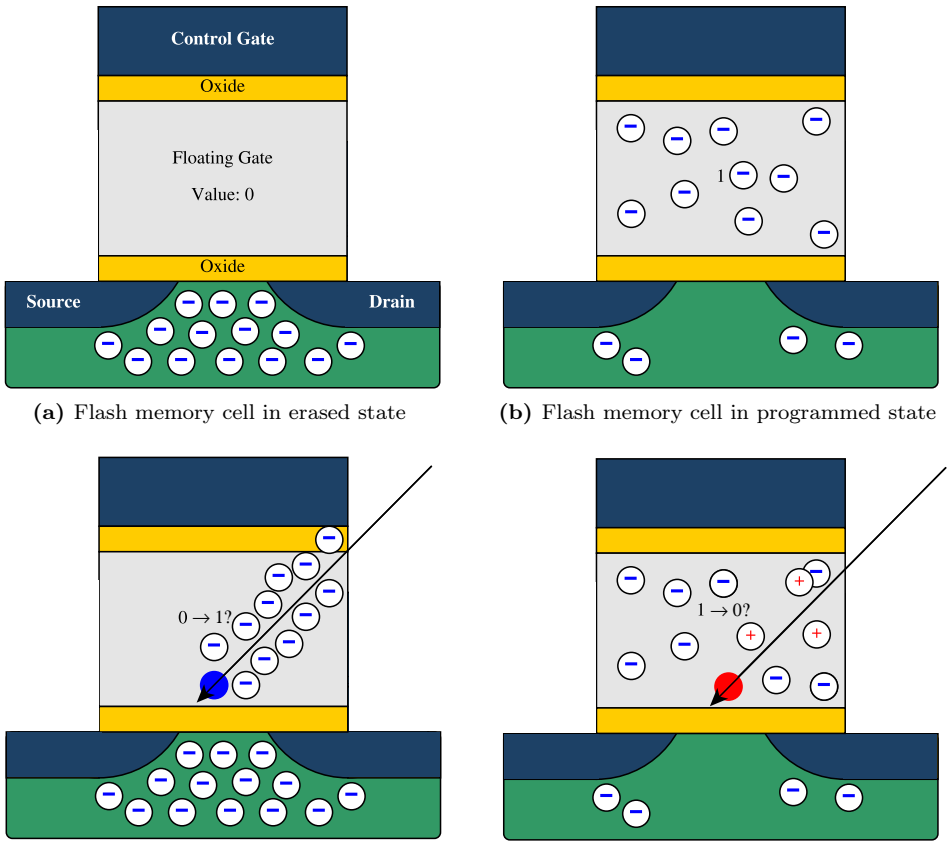
The memory cells of certain novel memory technologies (e.g., MRAM [147], and ReRAM [148], and PCM [149]) have been shown to be inherently immune. This is due to the data storage mechanism in these non-charge based memories [150, 151]. The memory cells of commercial MRAM and PCM ICs are largely immune to radiation-induced faults, and their structure and operating principle is depicted in Figure 16. However, connecting circuitry and controller logic of these parts if still vulnerable to radiation, and incorrect addressed memory can very well cause data corruption during read and write operations.

Flash memory, one of the most widely used charge-based memory technologies, has been shown to be rather susceptible to radiation effects [153]. Each flash memory cell contains a single field effect transistor with an additional floating gate, which is depicted in in Figure 17. Voltage applied between source and drain generates an electric field with a conductive channel through which electrons can flow into the floating gate. The state of a cell is thus dependent on whether or not a specific threshold voltage is exceeded (programmed, Figure 17b) or not (erased, Figure 17a).

Radiation can induce a variety of effects in charge-based memory such as flash [153]. In Figures 17c and 17d, we depict two opposing effects induced by particles with a positive and negative charge [154]. In Figure 17c a cell in erased state is hit by a

negatively charged particle. Such a particle can cause a storage cell to change its state by depositing electrons in the floating gate as it passes through the structure. Figure 17d depicts the inverse effect with a positively charged particle, which changes the net charge of the floating gate. The particle event may cause the charge in the floating gate to rise or drop one rise above or drop below a volatile threshold of the cell and thereby change the value represented by the storage cell.

Particles may also alter the structural integrity of different parts of the memory cell, e.g., draining the gate, or causing permanent damage [153]. Due to a shifting voltage threshold in floating gate cells caused by the total ionizing dose, flash memories become more susceptible to data degradation due to leakage. Modern multi-level cell flash memories manufactured in fine technology nodes are more prone to SEUs causing shifts in the threshold voltage profile of one or more storage cells [153]. Flash cells can also store more than a bit of data, and then also become susceptible to



(a) Flash memory cell in erased state

(b) Flash memory cell in programmed state

(c) Erased cell hit by a negatively charged particle

(d) Cell reset by a positively charged particle

Figure 17: The structure of a Flash memory cell in erased (a) and programmed state (b), inspired by a figure from Zandwijk et al. [152]. Data is stored as charge in a floating gate attached to a controlling field effect transistor. Radiation can induce a variety of different effects in charge-based memory [153], and in Figures (c) and (d) we depict two opposing effects induced by particles with a positive and negative charge [154].

MBUs: radiation may cause a state change across multiple voltage levels [155]. The semiconductor's temperature and particle events can also influence the leakage current of a these memory cells, thereby reducing the charge stored within the floating gate over time [156]. The radiation-induced effects depicted in Figure 17 are representative for the entire class of charge-based memories, even though other memory technologies store data as charge in electrically different ways [157].

Physical shielding using aluminum and other materials can reduce certain radiation effects [158]. The necessary shielding strength depends on the physical properties of the material used for shielding [159]. This approach has been used extensively in classical space applications in the early time of spaceflight. However, the level of shielding needed to protect modern semiconductors from radiation effects would require a miniaturized spacecraft to dedicate an unreasonable additional mass and volume to shielding [159]. For very large satellites, the use of strong shielding is still a viable (but costly and inefficient) option [160].

Weak shielding can introduce scattering effects, while offering nearly no added protection [161]. These can occur due to interaction of a highly charged particle with shielding material, which can cause a shower of charged secondary particles. This secondary particle radiation takes the shape of a cone from between the point of impact of the original particle and the underlying semiconductor [161]. Particle scattering can therefore cause multiple particles with lower charge to penetrate a semiconductor, instead of just one. Hence, very thin shielding such as aluminium-RF-cages commonly found in consumer electronics offer usually no radiation protection [159].

3.1.2 Design Constraints for Space Electronics

The success of a satellite missions depends on designer's ability to develop a system that can withstand operation in the space environment, and can cope with the design constraints that are in place aboard a satellite. In the remainder of this section, we therefore provide a brief overview of satellite design constraints.

Solar cells are the main power source aboard modern spacecraft in the inner regions of the solar system [7]. A spacecraft's orbit, location and orientation (attitude) relative to the Sun, and the solar array's temperature all influence the efficiency of its solar array. Miniaturized satellite's have small solar arrays with varying output, and their OBCs are limited to a few Watts of power-budget (power consumption averaged over time).

Operation in the space environment outside planetary atmospheres means that a satellite will operate in vacuum [162]. In turn, this implies the absence of the heat-transfer medium necessary for thermal convection, and hence also air cooling. Depending on the specific chip design implemented within a semiconductor, this can cause a chip and its packages to exhibit different or even anomalous thermal properties, potentially causing hot-spots and impact performance and lifetime [163]. Heat generated within a spacecraft therefore has to be transferred to the exterior and is then emitted as infrared radiation. A variety of engineering measures are available to help create a stable spacecraft-internal temperature environment [164].

Operation in vacuum and the low temperatures encountered in the space environment, can cause rapid material aging. The extreme temperature deltas when operating in a planetary orbit in direct sunlight and darkness can furthermore cause out-gassing, e.g., of chemical softeners present in materials such as plastics [165]. Gassed-out chem-

icals may interact with other components of a spacecraft, especially sensors, and may cause folded solar cell arrays to stick together, fold incorrectly, and fail to deploy from stowage [166]. This effect is a major problem for spacecraft equipped with optical payloads, e.g. astronomical observatories: out-gassed chemicals may then accumulate over time on sensors, mirrors, and lenses, and degrade an instruments performance. In large spacecraft projects, components are therefore often baked at high temperatures or exposed hot-cold cycles to reduce this effect in space as much as possible.

Upon launch, satellites have to withstand considerable physical stress and may experience vibration-induced resonance effects [167]. To a certain extent, these can be simulated through mechanical means (shakers) and acoustics on the ground, and then mitigated through engineering and a wise choice of materials. To design computer systems to better cope with launch stress and the extreme temperature changes that may be encountered in the space environment, electronics can be packaged in more suitable materials than the usual plastic packages used on the ground. However, electronics in ceramics and metal-based packages are at the time of writing significantly more expensive than conventional consumer parts, and usually non-options for CubeSat applications. Specialized materials can also be used in the different layers of a PCB, and can help optimize electrical, structural, and thermal properties, which today is also used aboard miniaturized spacecraft, e.g., aboard the MOVE-II CubeSat.

3.2 Technology Readiness and Standardization

Satellite missions can last from several months up to many decades, and therefore satellite designers may encounter hard technological barriers such as data retention [168]. Examples include, but are not limited to, issues with using electronics storage technologies due to limited data retention periods, solar cell degradation, and material degradation due to long-term thermal stress and out-gassing.

Traditional space companies and organizations are very cautious when considering new technology with little or no space heritage. Often, they modify and adapt existing, foreign industry standards to their own needs instead of reusing them, and develop their own standards [169]. Several sets of space related quality and design standards exist, which are administered by committees consisting of space agencies, governmental bodies, military and major industrial actors. Some of these standard libraries are published, while others remain proprietary (e.g., ARINC) or are even kept confidential (military standards). Currently, the most relevant publicly available and widely adopted standards are published by the *Consultative Committee for Space Data Systems* (CCSDS), the *European Cooperation on Space Standardization* (ECSS), and the *NASA Technical Standards Program*. Standards popular in the IT-industry in general do influence avionics design (e.g., Ethernet/IEEE 802.3 is today the technological base for AFDX [94], but adoption of this technology has taken more than 30 years), but mostly indirectly due to a technological lag between IT-industry and space-avionics that ranges from between 10 to 40 years [170].

Avionics (thus, Aerospace and Spaceflight electronics) development relies not just upon specialized and tested components. Instead, technological maturity has to be proven in practice to demonstrate that a component or technology is ready for application in the space environment. Thereby, the quality and heritage of a solution are assessed based on a standardized set of indicators resulting in a classification in technological readiness levels (TRLs) [171], see Figure 18. For some types of chips the

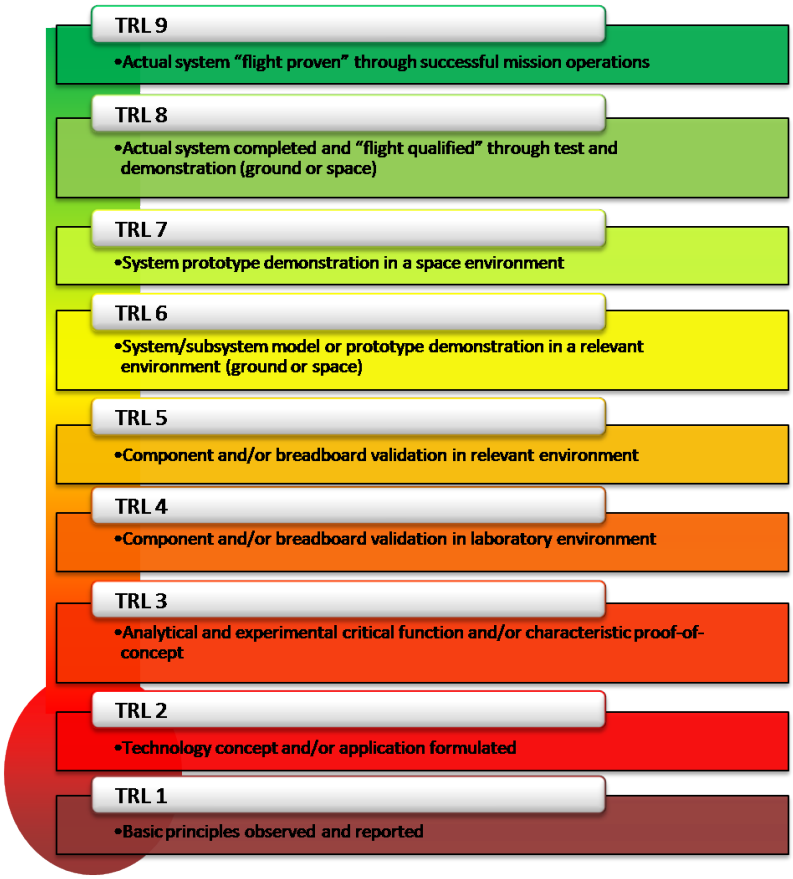


Figure 18: Technology readiness levels and the requirements to qualify a component for a certain level.

Image Credit: NASA, public domain.

global space industry may have annual demands for only several hundred or thousands of chips, resulting in extreme per-device development costs compared to common IT industry production quantities of millions of units. Due to limited alternatives and their requirement to rely upon proven and validated hardware, the space industry and their customers must afford high hardware costs and accept long development cycles [169]. The TRL required for a component may vary per usage scenario and subsystem, the highest level is thus not necessarily required and TRL9 components may even be replaced with less expensive or maybe more modern components with a lower level.

In this thesis, we propose an architecture which incorporates a set of theoretical fault tolerance concepts, which exist at TRL1. Based on these concepts, we formulate a conclusive architecture for our application, which initially exists at TRL2. We then proceed to conduct fault-injection experiments with a proof-of-concept of the architecture (TRL3), and produce a practical implementation based on development-board components in a bread-board setup (TRL4).

3.3 Operational Constraints for Satellite Computers

In contrast to most earth-bound computing, it is not possible to physically access a spacecraft in orbit [172] to diagnose or resolve faults. However, this does not mean that they can not be repaired, refueled, upgraded, or otherwise serviced during a mission. In fact, most spacecraft are designed to be service friendly, as this makes them easier to assemble and test on the ground. This is especially important as testing of a spacecraft as a whole and its individual subsystems is a complex and costly undertaking. Component-level as well as testing of a full avionics system makes up a significant share of the time needed for the design and construction process.

Hands-on maintenance or diagnostics on-orbit are uncommon today, and servicing missions have been conducted only on a few occasions. All of these spacecraft were large satellites and space-stations in LEO with outstanding significance to science, society, or driven by national interests. Prominent examples include the Hubble Space Telescope [173] and several space stations [172, 174, 175], where servicing was required to resolve faults. For most modern non-agency and non-governmental satellites, and especially smaller and cheaper spacecraft, hands on maintenance is not feasible, and usually also not economical [173]. Hence, an on-board computer has to operate and handle faults autonomously over the entire duration of a spacecraft's mission, which may last for several decades.

Diagnostics of computerized systems therefore have to be conducted remotely and in a scripted manner locally aboard a satellite. Considering the journey of Cassini/Huygens depicted in Figure 19, this implies differences in link behavior and communication bandwidth during a mission. Even in earth orbit, a satellite's telemetry and telecommand (TMTC) link is lossy, and offers very low bandwidth compared to ground-based communication (in the low kbps range). As depicted in Table 2, signal travel times in LEO and Geostationary Earth Orbit (GEO) still allow widely used network communication protocols for ground use to be utilized, if aspects such as Doppler-Shift are compensated for [178].

All CubeSats launched until 2018 operated in a LEO [17], and most utilize a combination of UHF and VHF frequency bands to realize their commandeering channel. LEO communication windows between a ground station and a satellite are limited to between 5 and 20 minutes in ideal weather conditions, and reduced by equipment dampening, environmental effects, and atmospheric conditions [179]. Only part of this communication window allows actual communication with a spacecraft due to link-quality issues. The actual duration varies depending on the satellite's orbit and the environment the ground station operates in: buildings, natural obstacles and fading signal quality with declining elevation angle when approaching the horizon all affect a link's signal-to-noise ratio [180]. For comparison, while commandeering the FirstMOVE CubeSat, actual link availability during communication windows never exceeded 12 minutes.

LEO-link availability can be increased through the use of satellite-relays (e.g., TDRS [181]) and ground-station networks [182]. However, these are currently largely unavailable to miniaturized satellites due to economical considerations on the operator's side and form-factor and cost constraints for miniaturized satellites. In practice, this curtails remote debugging capabilities of spacecraft. It prevents the direct re-use of, e.g., all low-level testing protocols which are today widely used on the ground application such as JTAG or ICE, and prevents remote-debugging using standard debugging

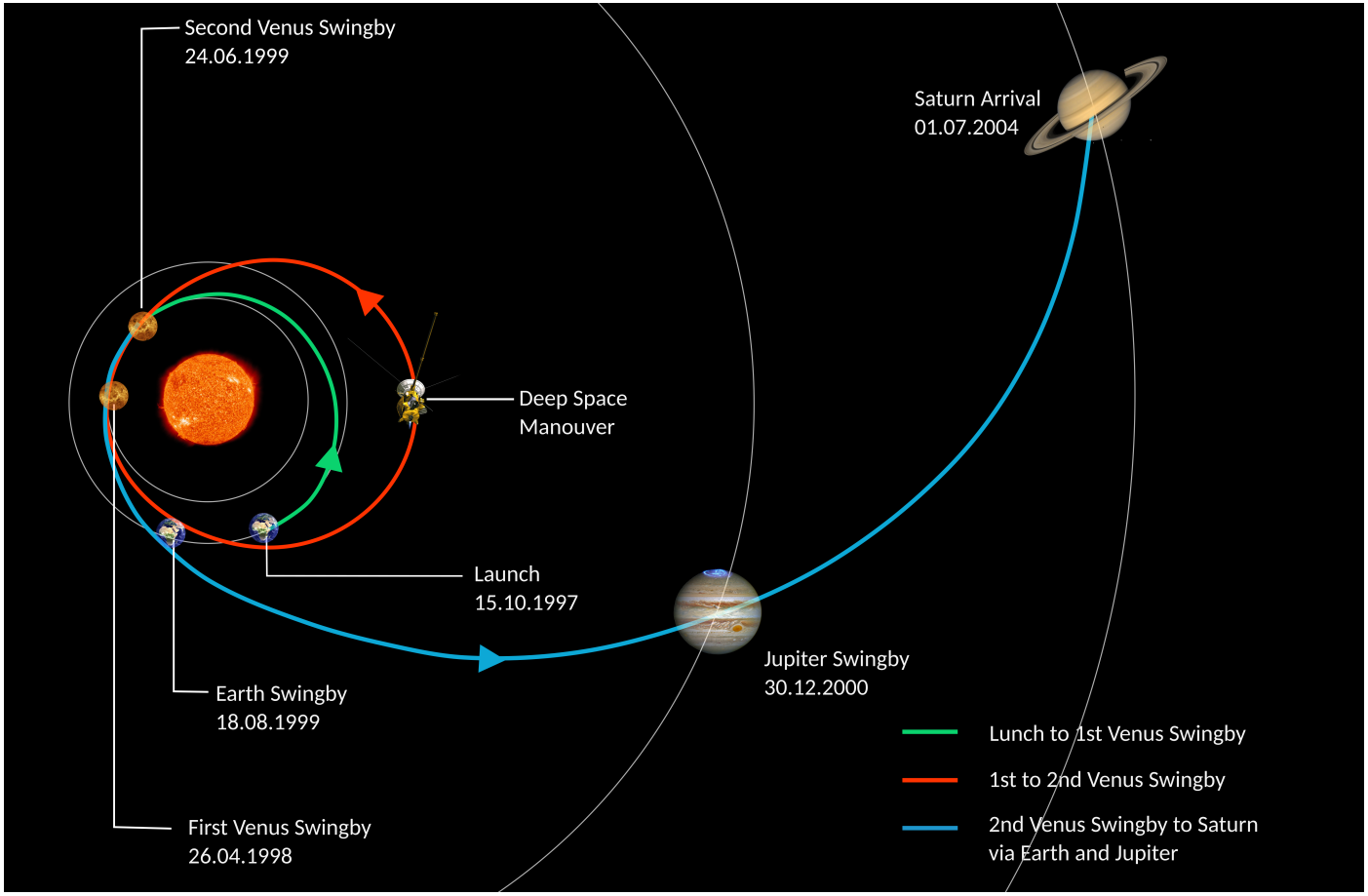


Figure 19: The flight path of Cassini/Huygens with different mission phases indicated in color. Cassini's mission ended after almost 20 years after a "Grand Finale" with several close flybys of Jupiter and its moons, when it burned up in Saturn's atmosphere on September 15$^{\text{th}}$, 2017.
Self-redrawn image based on [176] and [177], Image Credit: NASA/JPL, for Public Use

tools.

When communicating with spacecraft orbiting other planets in our solar system, signal travel times and thus link latency grow rapidly. With space probes traveling beyond the Earth/Moon system, the available link rates decrease sharply and often only few hundred bps can be achieved. Unidirectional signal travel times to neighboring planets make real-time bi-direction communication concepts as used on the Earth technically impossible. At the time of writing, the mars rover Curiosity can achieve a data rate of between 500 bps up to a theoretical maximum of 32 kbps and round-trip times of at least 8 minutes under ideal circumstances [183]. The TMTC link of the Voyager probes [184] can achieve a maximum of 160 bps at the edge of the solar system via the Deep-Space Network [185] with signal travel times approaching a duration of a day.

As depicted in Figure 19, a spacecraft may have to travel within our solar system for years, before actually arriving at its destination, where it can then begin to perform its actual mission. During such missions, the performance requirements to a satellite computer can vary. In Figure 20, we depict a simplified version of the orbit/work schedule of NASA's Enceladus Life Finder (ELF) probe, which will conduct science on Saturn's sixth largest moon. Travel to the Saturn system will take years, but once

Communication	**Distance from Earth**		**Signal Travel Time**	
Endpoint	Min.	Max.	Min.	Max.
LEO	400 km	2,000 km	3 ms	18 ms
GEO	35,786 km	-	~250 ms	-
Moon	356,400 km	406,700 km	2.4 s	2.7 s
Mercury	0.62 AU	1.39 AU	5 min	12 min
Venus	0.28 AU	1.72 AU	2 min	14 min
Mars	0.53 AU	2.52 AU	4 min	21 min
Jupiter	4.21 AU	6.21 AU	35 min	52 min
Saturn	8.54 AU	10.54 AU	1:11 h	1:28 h
Uranus	18.23 AU	20.23 AU	2:32 h	2:48 h
Neptune	29.06 AU	31.06 AU	4:02 h	4:18 h
Voyager 2	~121 AU	-	~16:50 h	-
Voyager 1	~147 AU	-	~20:22 h	-

Table 2: Unidirectional signal travel times for radio communication in vacuum between a ground station and a spacecraft at a particular location in the solar system. Distances between the Earth and different planets in the solar system vary due to celestial mechanics. In practice, the signal latency even for LEO communication is drastically larger than the theoretical signal travel speeds indicated here due to latency in the signal processing chain. Data for the Voyager probes based on `https://voyager.jpl.nasa.gov/mission/status`, accurate as of September 2019.

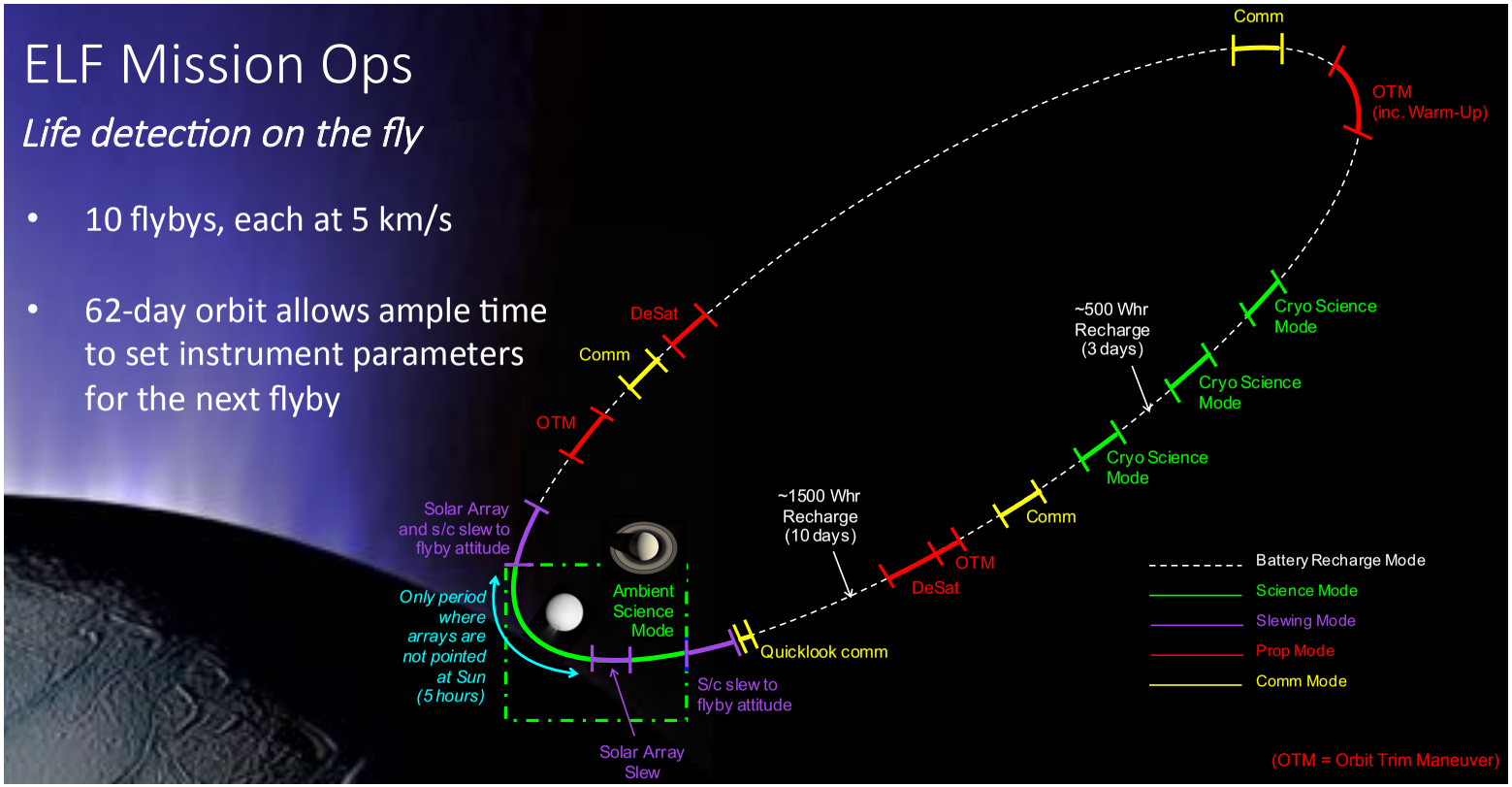


Figure 20: NASA's Enceladus Life Finder (ELF) is scheduled to make ten flybys of Saturn's moon Enceladus to investigate that its environmental properties. In each color-highlighted orbit-segment, ELF has to conduct a different task or operation, with different requirements towards the spacecraft's on-board computer.

Image Credit: Figure from public-use preview press material by the ELF Team/JPL/NASA, Based on a figure from [186]

ELF has entered orbit around Enceladus, it will have to handle a variety of different tasks with very different system requirements (indicated in color). We utilize this satellite's mission operations schedule to highlight how requirements to a satellite's on-board computer can shift during a mission.

During the yellow-outlined communication phases, reliability of the satellite computer is crucial, as communication windows are brief and the available link-rate is low. Any lost communication time could directly impact the satellite's mission and subsequently executed tasks. Ideally, during this time a satellite's computer should offer increased fault tolerance capabilities at the expense of other system parameters, if such capabilities were available.

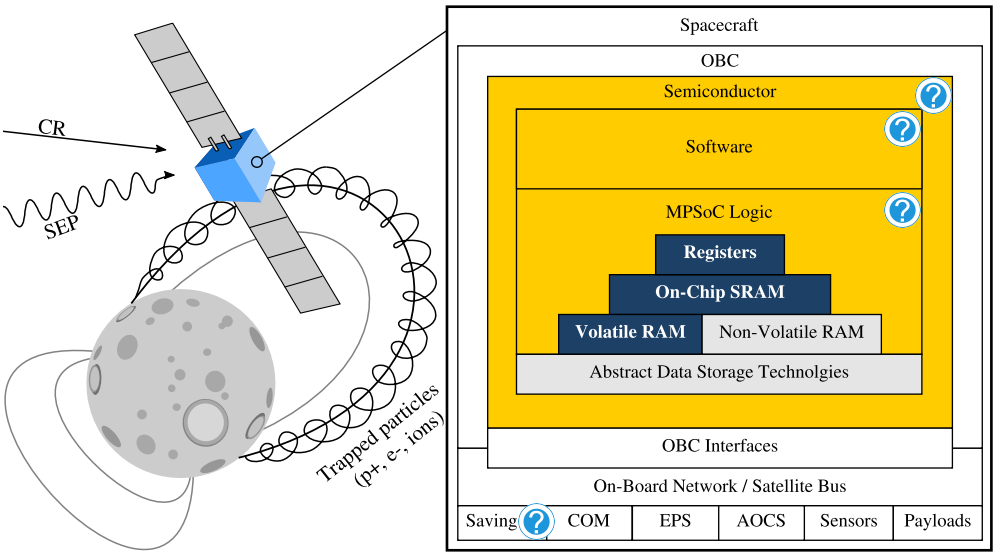
The red- and purple highlighted orbit segments indicate times when ELF will perform maneuvers through its propulsion subsystem and adjust the orientation of it's solar panel array. When performing maneuvers, precise timing and therefore the ability for real-time operations are crucial, while overall compute performance requirements will be comparably low. Finally, in the green-market science phase, performance is critical, and during this phase, spending extra energy to increase the satellite's overall compute and data-storage capacity may allow the spacecraft to conduct more and better science within its brief mission. With the computer architectures used aboard spacecraft today, little adaptivity is possible. However, future satellite computers based on modern mobile-market and embedded computer architectures could very well support such functionality if fault tolerance capabilities can be adjusted at runtime.

Chapter 4

A Fault Tolerance Architecture for Modern Semiconductors

Stage 1 & Architecture Overview

In this chapter, we describe a non-intrusive, integral, flexible, hardware-software-hybrid approach which enables the use of modern multiprocessor system-on-chips (MPSoCs) for spaceflight without violating application constraints. We introduce a co-designed system architecture utilizing three interlinked fault tolerance measures. To drive this architecture, we propose a coarse-grain thread-level lockstep implemented in software, and describe our implementation in detail in this chapter. We provide benchmark results for the lockstep, which allows very pessimistic worst-case performance overhead measurements. The technological feasibility of this architecture is demonstrated through implementation of a basic proof-of-feasibility MPSoC implementation.



4.1 Introduction

Modern embedded technology is a driving force in satellite miniaturization, contributing to a massive boom in satellite launches and a rapidly evolving new space industry. Micro- and nanosatellites (100-1kg) have become increasingly popular platforms for a variety of commercial and scientific applications, due to an excellent balance of performance and cost. However, this class of spacecraft suffers from low reliability, discouraging its use in long, complex, or high-priority missions. The OBC related electronics constitute a much larger share of a miniaturized satellite than they do in larger satellites. Thus, per component, they must deliver better performance and consume less energy. Therefore, due to cost considerations, miniaturized satellite OBCs are generally based upon processors manufactured in fine-feature-size technology nodes, such as those used in mobile embedded devices.

Traditional hardware-based fault tolerance (FT) concepts for general-purpose computing, however, are ineffective for modern, highly scaled systems-on-chip (SoCs), becoming a prime source of malfunctions aboard miniaturized satellites [2]. Larger satellites, too, are limited by the constraints of traditional ways to achieve fault tolerance for space applications, as these prevent larger satellites from harnessing the benefits of modern processor designs, and multiprocessor-SoCs (MPSoCs). Also, these hardware-based FT-measures can not handle varying performance requirements during multi-phased missions and mega-constellations [187]. Software-based FT measures rapidly evolved due to efforts of the scientific community, and are effective for modern embedded hardware. However, these advances have largely been ignored by the space industry, as well as closely related fields such as atmospheric aerospace, as they were researched only in theory, but rarely meant for implementation. While many of these concepts include innovative ideas, major implementation obstacles and fundamental issues remain unaddressed. Often, prior research makes impractical assumptions towards the platform or application environment, ignores fault detection, recovery from failover, or other real-world constraints. Many concepts also attempt to uphold safety and availability, e.g., for atmospheric aerospace use, but not computational correctness. To the best of our knowledge, no integral and practical solution to utilizing modern MPSoC-based systems within high-priority space missions has been developed to date.

There is a wide gap between academic research towards novel FT concepts and their practical application in spacecraft OBCs. Satellite computers for control purposes are still largely based upon architectures developed decades ago, while theoretical research has not achieved the level of maturity necessary to bridge this gap. Thus, neither traditional hardware- nor software-based FT solutions could offer all the functionality necessary to improve the reliability of state-of-the-art embedded SoCs in miniaturized satellite OBCs. Other concepts promise excellent FT guarantees in theory, but require complex architectures that often do not address the specific challenges of computers flying in space. Innovations are especially needed in general-purpose computing, as OBCs must execute a broad variety of applications efficiently.

This approach was developed for a 4-year European Space Agency (ESA) project with two industrial partners. Due to the interdisciplinary nature of this project, other aspects of this approach and its hardware implementation are described further in Chapters 5 – 10.

In the next section, we discuss related work, and how the design constraints and challenges outlined in Chapter 3 are up until the time of writing are addressed in fault-

tolerant OBC design. Section 4.3 contains a brief overview of the multi-stage approach, its limitations, terminology, as well as the application model and requirements. Each stage is described in the subsequent sections, with the supervision concept explained in Section 4.4.4. Section 4.7 then introduces briefly an MPSoC architecture specifically designed as a platform for this FT concept. Performance and checkpoint reliability are discussed in Section 4.8, followed by conclusions.

4.2 Related Work

Radiation challenges OBC fault coverage constantly and throughout a mission and affects all of an OBC's components depicted in Figure 21. Traditionally, FT is enabled through circuit-, RTL-, core-, and OBC-level voting, which is costly to develop, difficult to validate, maintain, and slow to evolve [88,104,132,188–190]. Software takes no active part in fault-mitigation, as faults are suppressed at the circuit level, preventing the effective assessment of a processor's health. Circuit- and RTL-voting are effective for microcontrollers and very small SoCs, while core-level voting requires logic unavailable in COTS systems. Modern embedded COTS MPSoCs consume very little energy. But to achieve FT using hardware-side measures, arrays of synchronized high-frequency voters or core-lockstep in hardware are necessary. As voting and core-level lockstep at GigaHertz clock rates are non-trivial, it has been implemented only at considerably lower frequencies with non-COTS hardware [88, 190–192].

In general, hardware-voting based MPSoC designs are static and non-adaptive, as the entire design's fault coverage properties are highly chip-specific [193]. All these components are single-vendor solutions, often with walled-garden ecosystems with vendor lock-in. FT MPSoCs for space use contain retrofitted TMRed single-core processors, e.g., [104], or are unique, experimental solutions for specific satellite missions [194,195]. In contrast to these solutions, modern MPSoCs also allow considerably more software design freedom due to the available compute resources, thereby reducing the required development time and complexity. For scientific instrumentation and low-priority CubeSat missions, COTS-based MPSoCs and FPGA-SoC-hybrids have been utilized, but these are not suitable for critical satellite control applications within miniaturized satellites [196]. Ground-based FT applications do not consider the specific threat-scenario and application environment, physical constraints, and thermal design constraints [5, 197]. Instead, we propose to use software-side functionality to assure FT for conventional, non-fault-tolerant processor cores.

First concepts involving coarse-grain lockstep are promising [198–200], but do not address the specific challenges to FT in space [201]. FT using thread-level very-long-instruction word architectures [202, 203] has also been explored, though the approach still requires pipeline-level voters in hardware. Most implement checkpoint & rollback or restart, which makes them unsuitable for spacecraft command & control applications [204], others ignore fault-detection [205, 206], or require external, infallible fault detection entities with deep knowledge about application-intrinsics [207] but no concept of how this could be obtained. Often, faults are assumed to be isolated, side-effect free and local to an application [208] and/or transient [199,200,205], which voids their effectiveness for space applications. Many prior concepts entail high performance- [209], resource-overhead [210,211], or impose severe design constraints on applications and the OS [198,199]. To be effective in the space environment, an FT approach must be based upon forward-error-correction and the implementation complexity must be

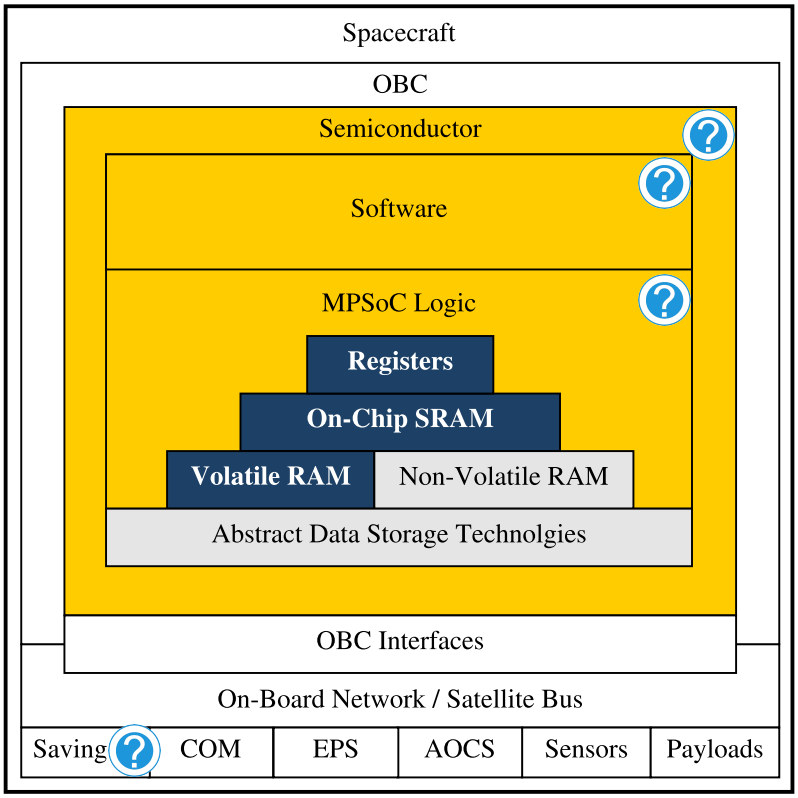


Figure 21: A component-level view of a satellite OBC. The multi-stage fault tolerance architecture proposed in this chapter covers faults affecting MPSoC, semiconductor infrastructure, logic as well as software (yellow). Volatile memory (blue) and non-volatile memory (gray) can well be protected using error correction coding and is described in Chapter 7.

low, and must be suitable for general-purpose computing and impose little or no constraints on the application software. Changes to the OS infrastructure must be platform portable, code-wise localized, and individually verifiable.

[199, 200, 208] implement voting through OS invasive measures, can not handle multi-threaded applications and consider the OS and stored program code to be fault-free. [201] requires no modifications to the application software whatsoever, but can only assure availability in a networked application architecture. An acceptance of these constraints does not allow for adequate FT in a space mission scenario, and thus we propose that application and OS instance must be able to fail arbitrarily without impacting the residual system. In this case, fault propagation between application instances also becomes a non-issue. Considerable research has been directed towards FT real-time scheduling and mixed critical software-FT systems, though only at a theoretical level [212–214]. As a consequence, no implementable, software-driven FT concept for modern embedded- and mobile-market MPSoCs in space exists, creating a gap between the described prior research on software- and hardware-FT based implementations.

4.3 Fault Tolerance through Software

This approach consists of three fault-mitigation stages:

Stage 1 is implemented entirely in software and provides fault-detection through coarse-grain lockstep to enable self-testing, and can be implemented in COTS MPSoCs.

Stage 2 improves medium-term reliability through FPGA reconfiguration, and enables long-term fault coverage using alternative configuration variants. It utilizes Stage 1's fault detection capabilities.

Stage 3 extends the lifetime of a degraded OBC by utilizing mixed criticality to assure fault coverage for high-criticality threads. It enables the OBC to automatically sacrifice performance or fault coverage of lower-criticality threads in favor of higher-critical applications, thereby maintaining a stable core system.

The presented concept is flexible and the individual stages are modular, as Stage 2 or 3 can be omitted depending on the OBC and mission. Our approach is designed for generic COTS MPSoCs, as these are readily available in a variety of performance classes at low cost. In the architecture described in Section 4.7, we place processor cores within isolated compartments. We consider it an ideal platform for our approach. In MPSoCs without a compartments, *compartment* can be substituted for *processor core*, and the differences in fault coverage are discussed in Section 4.7.

Terminology

Fault detection in our approach is based upon sets of compartments running two or more lockstepped copies of application threads. We refer to such a group of lockstepped threads as a *thread group*. Timing-compatible thread groups can be combined and executed on the same set of compartments, and are then referred to as a *compartment group*.

The relation between these is visualized in Figure 22. A thread group can realize a varying level of replication to achieve majority voting (thread 0 in the figure), error detection (thread 1), or even individual execution. One compartment may be host to

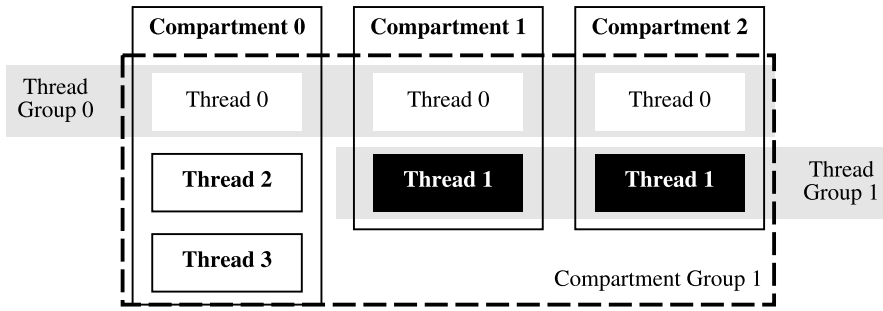


Figure 22: Schematic illustration of the relation between compartments running applications as threads, thread groups, replication, and timing-compatible compartment groups.

multiple thread groups threads may be unassigned from it, or newly assigned to it at runtime using conventional thread and process management functionality of the OS.

A compartment group periodically executes a *checkpoint routine*, which computes checksums for all active threads and compares them with the other compartments in the group (*siblings*), thereby enabling a majority decision or error detection. The time between checkpoints (the *checkpoint frequency*) is defined by the threads in a compartment group and can be modified at runtime. All lockstep-relevant information is stored in *state memory*, a compartment-dedicated memory segment which is read-only accessible by compartments.

Application Requirements

The OS only has to support interrupts, wake-up timers, and a multi-threading capable scheduler. To the best of our knowledge, such functionality is available in most widely-used RT- and general-purpose OS implementations. Virtual memory support is required to enable performance-efficient multi-threading. Furthermore virtual memory simplifies thread-management, context switching, and thread isolation, benefiting overall fault tolerance.

The only requirement for applications is interruptable at application-defined points in time, during which checkpoints can be executed. As there is no efficient, uniform approach to assess the health of threads, we rely upon applications assessing their own health-state. A thread can provide four callback routines to the OS, which are executed during compartment initialization and by the checkpoint handler:

- an *initialization routine*, to be executed on all compartments at bootup;
- a *checksum callback*, used to generate a checksum for comparison with siblings,
- a *expose state callback*, exposing all thread-state relevant data to synchronize a sibling with a compartment group; This data can either be placed directly in the compartment's state memory, or as a reference to structures in main memory.
- and an *update state callback*, which is executed on a compartment that needs to synchronize its state to a compartment group.

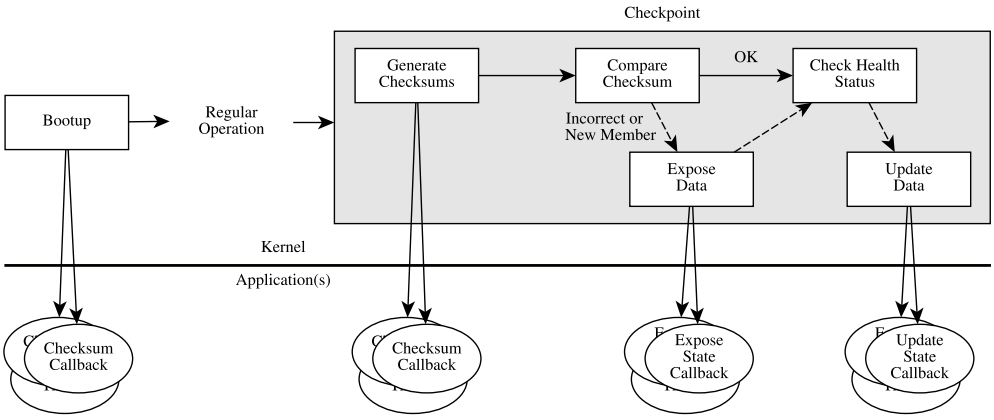


Figure 23: High-level time diagram for the execution of application provided callback functions during the operation on an on-board computer.

Figure 23 depicts where and how these callbacks are used during the regular operation of the lockstep. Some of the callbacks may be omitted, e.g., for applications not requiring bootstrapping or with an already exposed state. The checksum computation and state synchronization callbacks are intentionally placed within the domain of the application developer. This enables decisions about an application state to be taken by the entity with the best knowledge of the individual thread and the means to determine which data is relevant to the system and application state, and must be preserved.

Threads can be executed in an arbitrary order within a lockstep cycle as long as their state is equivalent during the next checkpoint. However, interrupting an active application at a random point in time is usually undesirable. We avoid thread-synchronization issues [198] by enabling the application developer to define comparison points where the application will yield control to the checkpoint handler. If an application requires real-time scheduling, the tightness of the RT guarantees depends upon the time required to execute these callbacks. Communication between thread-groups and compartment-groups is of course possible and will remain reliable, as long as the receiving application is aware that it will receive multiple message replicas. To prevent faults from propagating through IPC channels, a thread can compare the received messages.

Limitations

This approach guarantees system state consistency and control flow correctness after each checkpoint, and for all past checkpoint periods. It also assures computational correctness before the last checkpoint, but can not actively prevent faults from occurring during the ongoing checkpoint cycle. Thus, if one compartment experiences a fault, incorrect results may be propagated outside the system, even though the damage caused to the OBC will be corrected during the next checkpoint, and system state consistency will be asserted. This limitation is inherent to coarse-grain lock-stepping concepts, but could be elevated at the thread-level somewhat using finer-grain event hooking, e.g., system-call hooking [199]. However, this workaround requires in-depth modifications to the OS kernel and development toolchain, is thus non-portable and difficult to maintain, while still not solving the underlying conceptional limitation.

Related research, however, does show that a solution at the system-design level is much better suited to prevent fault-propagation of transient faults between checkpoints using simple I/O voting [201]. Traditional hardware-FT approaches used in space computing are strong for assuring non-propagation of faults across interfaces using hardware-side voting, but can not protect the control-flow and system-state consistency efficiently. While the system state and system-level fault tolerance are assured by Stage 1, and long-term system resilience are safeguarded in Stages 2 and 3, we can utilize simple I/O voting to prevent fault-propagation for compartment groups. Performing I/O voting on interface is already a common practice in satellite computing, as considerable effort is put into providing interface redundancy aboard larger satellites. Small satellites, especially CubeSats, usually can not spare the additional energy, space and mass required for interface replication. For such spacecraft, I/O voting can be implemented on-chip using library IP cores.

4.4 Stage 1: Short-Term Fault Mitigation

Stage 1 offers software-controlled, thread-level, distributed majority voting and fine-grain fault logging within any COTS MPSoC with three or more processor cores. The objective of Stage 1 is to detect and correct faults at each checkpoint to assure computational correctness, control-flow consistency, and a consistent system state after each checkpoint. To do so, Stage 1 requires a processor guaranteeing sequential consistency.

Instead of exerting direct control over the MPSoC, a supervisor can assure FT indirectly, as fault coverage and control are distributed and enforced by the compartments themselves. In consequence, the supervisor does not require any knowledge about the executed application threads, an individual compartment's state, or other OBC intrinsics. The thread group assignment within an MPSoC can be reconfigured freely at runtime to implement different voting configurations. Thus, the described approach can exploit parallelization to improve reliability, throughput, or minimize power consumption, thereby allowing the system to adapt to multi-phased missions with varying performance requirements.

4.4.1 Thread-Based Self-Testing

The program flow of this stage is depicted in Figure 24 and described subsequently. It can be implemented within an existing scheduler and an interrupt service routine (ISR). A practical example for compartment fault handling and recovery, and an overview over how the supervisor interacts with the system are provided at the end of this section.

Bootup & Initialization

After bootup, a compartment first executes basic self-test functionality to assure integrity of compartment-local IP-cores and memory. Each thread's initialization routine is executed on all compartments to allow faster state-update in case a new thread-group is added to a compartment. When being assigned to a compartment, a thread will register its desired checkpoint frequency and its checksum, expose/update callback routines. After the threads have been initialized, each compartment will set a periodic timer to initiate checkpoints. As depicted in Figure 24, a compartment will execute its first checkpoint immediately after the MPSoC has been fully rebooted, to assure that application and OS initialization were successful. If only this individual compartment was rebooted, it can thus return to the spare compartment pool to replace a faulty core in the future.

Checkpoint Start

A checkpoint is triggered by a timer interrupt or externally by the supervisor. A thread can delay a checkpoint until it has reached a viable state for checksum comparison by disabling interrupts, thereby deferring interrupt processing. The checkpoint ISR saves the existing system state, loads the actual checkpoint handler, performs a context switch to kernel mode, and invokes the checkpoint handler.

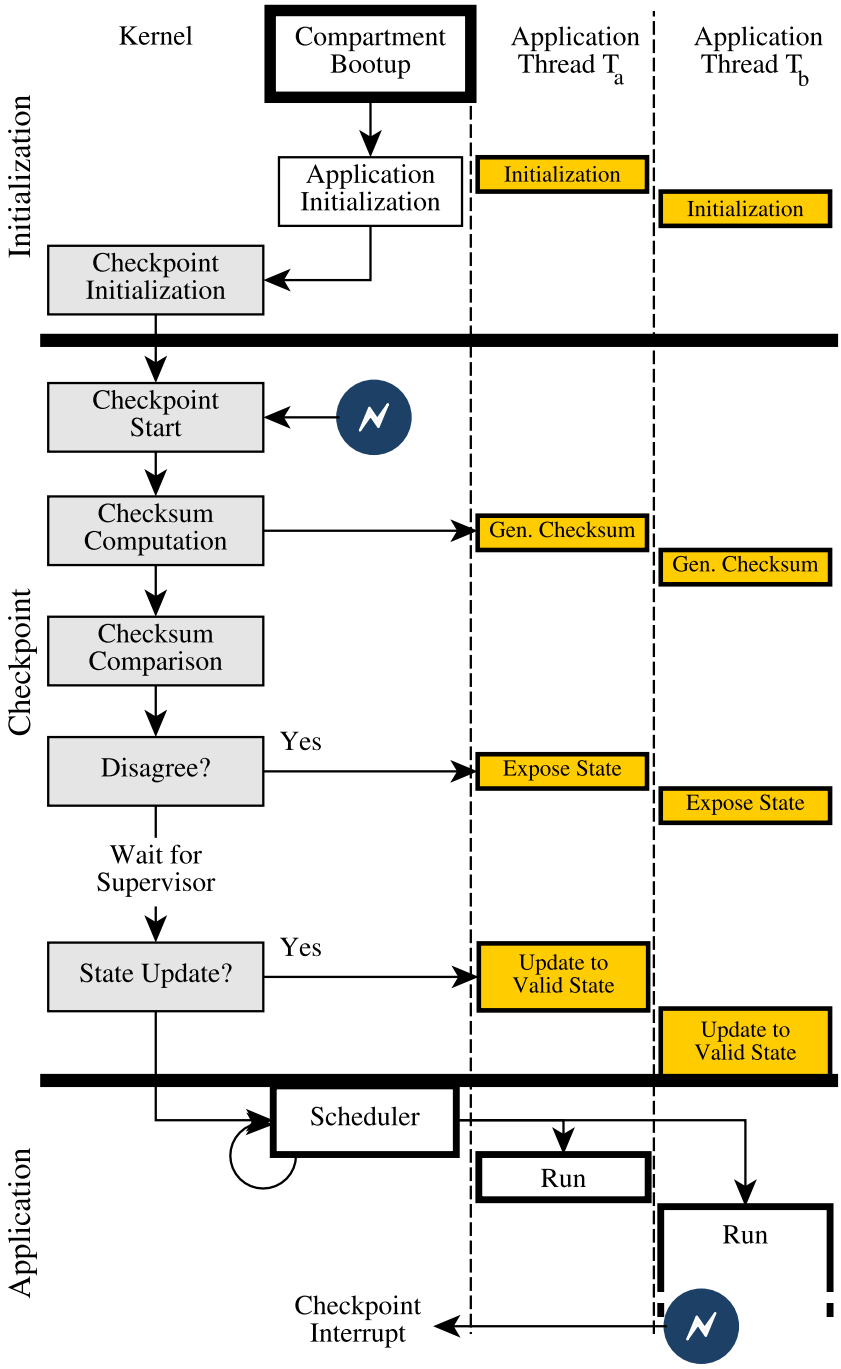


Figure 24: The execution cycle of a compartment during Stage 1. All code necessary for implementation is highlighted in gray, callbacks in yellow.

Checksum Computation

The checkpoint handler invokes each active thread's checksum callback scheduled for checking. As not all threads in a compartment group require the same checking frequencies, not all active threads will be validated during each checkpoint. This checksum callback returns a representation of the application thread's internal state as checksum or hash generated from thread-private variables and other internal application state. The checksum format is compile-time defined, and must be chosen based on FT needs. The algorithm used to generate this checksum is up to the application developer. Each checksum is stored in the compartment's local state memory and thereby exposed to the other compartments. If no checkpoint routine can be provided, a checksum is computed by the checkpoint handler for an application-defined memory range. This memory range can be utilized by the application to deposit state-relevant data passively, e.g., through linker scripts or pre-processor macros. A non-continuously running application can also deposit its results in state memory or return a checksum upon exit.

Prior concepts required deep modifications to the OS to allow a proprietary central health-management entity to retrieve this information directly [198,205], or utilized no application-internal information [200, 201, 211]. Instead, this approach enables us to utilize application-intrinsics to assess the health-state of the system, without requiring any knowledge on the applications. The time required to generate checksums can be minimized by adapting the application code, e.g., by retaining computational by-products which would usually be discarded.

Checksum Comparison

Once all checksum callbacks have been executed, a compartment will monitor its group members' state memory segments until another compartment is ready for comparison. It will do so until it has compared its checksums with all siblings, or the system designer's compartment-group deadline expired. Compartments will usually begin comparing its checksums with siblings immediately or wait only briefly, as delays are mainly induced due to varying memory latency or malfunctions. If it detects a checksum mismatch or a sibling violated the deadline, the compartment will stop comparing checksums and report disagreement with that compartment to the supervisor.

Thread Disagreement & State Propagation

If a compartment detected a checksum mismatch, it executes the expose state callback routine of all threads in the affected compartment group. This callback can be omitted if all state-relevant data is already in state memory, e.g., for non-continuous running applications. The checkpoint routine will adjust the checkpoint's timer if a new thread group was added to the compartment group, and return control to the scheduler.

State Update and Thread Execution

The scheduler will check three conditions during regular operation: if any thread-group is active, the compartment was newly added to a compartment group, or requires an update. Idle compartments sleep until the next checkpoint and can be woken up by the supervisor to reduce energy consumption and fault-potential. In case a compartment must update a thread-group's state from a sibling, the relevant update callback will be

executed for each thread. Compartments that have detected disagreement with one of their siblings will delay execution for a compartment-group-wide grace period, to allow a sibling to retrieve a state-copy from state memory. Once a compartment has updated its state using a sibling's data, application processing continues. The other compartment group members will also wake up after the grace period and continue executing threads. This concludes the lockstep cycle.

4.4.2 A Practical Example

Figure 25 depicts a quad-core MPSoC with a single compartment group and three members. A fault has occurred during the second lockstep cycle on compartment C_2, which is subsequently replaced with the idle compartment C_3. C_3 must retrieve a copy of the state of its threads T_a and T_b from another valid sibling. The replaced compartment, C_2, can subsequently be tested for permanent defects by the OS and the supervisor.

4.4.3 Checkpoint-Frequency & Real-Time Capabilities

The level of fault coverage is mainly dependent on the checkpoint frequency. During a checkpoint, the computationally most costly operations are the application checksum callbacks, the expose/update callbacks and a new compartment's update callback. Each of these operations involves a context switch and may imply a varying level of data being read or written. Thus, the performance overhead and fault tolerance capabilities are mainly based upon actual applications checked, as this actual checkpoint handler code is rather trivial. In general, a higher checkpoint frequency implies more time will be spent in checkpoints, finer grained fault-detection are possible, thus better fault coverage.

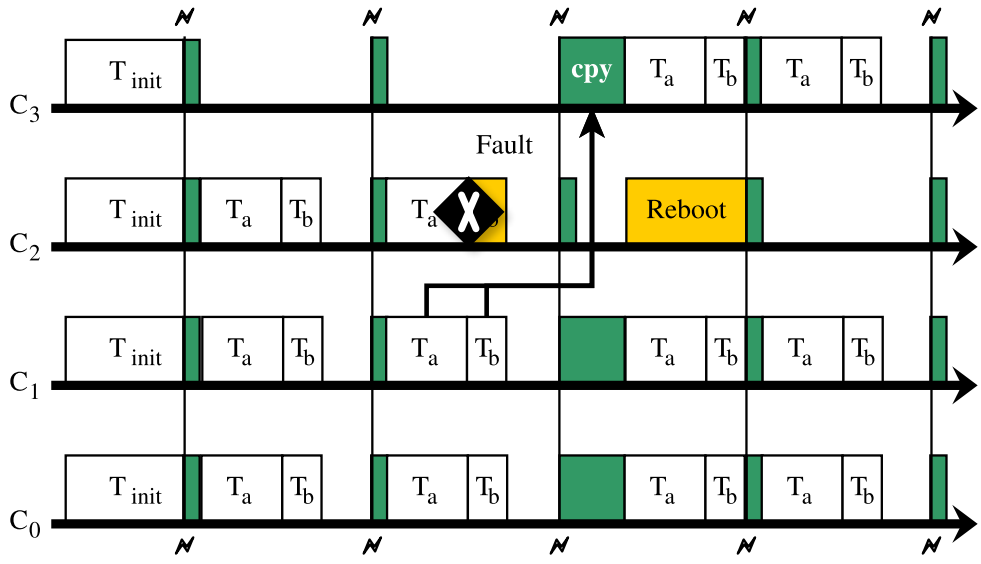


Figure 25: Compartment initialization and a complete Stage 1 lockstep cycle.

In our implementation, interrupts are deferred during a checkpoint, thus applications are not serviced and will not process I/O, thereby affecting the level of real-time capabilities the MPSoC can offer. However, though this can be worked around using a more elaborate interrupt handling concept, e.g., using interrupt prioritization or filtering. Real-time capabilities are thus directly dependent on the MPSoC, and application implementation characteristics, with the OS infrastructure playing a minor role. For complex applications with a large state, a lower checkpoint frequency, however, also implies a larger difference in state. Hence, more data must be copied between compartments to achieve thread-synchronization requiring additional time. Thus, a larger state also requires more time for execution, potentially more complex data structures, thereby implying longer expose- and update-callback.

Overall, the performance of OBCs executing less complex applications with little state will improve with lower checking frequencies. For such OBCs, more checkpoints imply more computational overhead. With more complex applications, there is considerable optimization potential to find a sweet-spot between checkpoint frequency and application-state size. However, performance is strongly dependent assuring that high-quality callback-routines are provided by the application developer.

4.4.4 Supervision

The supervisor is connected to the MPSoC through a multiplexed bus-interface, where each line signals agreement with another compartment. Fine grained disagreement reporting does not significantly improve fault coverage and constrains scalability of the MPSoC. As depicted in Figure 26, the supervisor only reacts to disagreement between compartments, otherwise remaining passive. It maintains a fault-counter for each compartment, and acts as a system-reset inducing watchdog timer for the MPSoC. To resolve transient faults within a compartment, it increments the fault counter and induces a state update through a low-level debug interface. After repeated faults, the supervisor will replace the compartment by adjusting the thread-mapping of a spare compartment, activating it, and rebooting the faulty compartment. In case a system developer indicated threshold is exceeded, the disagreeing compartment is assumed permanently defunct and not re-used as a spare. Stage 1 alone can not reclaim defective compartments beyond programmatically avoiding the use of defective peripherals, memory pages or processor functionality. Thus, Stage 2 will attempt to repair compartments to prevent resource exhaustion.

In contrast to existing FT solutions, faults can be reported by each compartment individually, because fault detection is decentralized. As this functionality is implemented at the kernel level, we can utilize the OS's powerful logging and diagnostics facilities, instead of relying upon the supervisor to provide a minimal useful level of logging. Diagnostics can thus be enriched with application-level information. Thereby, defect assessment accuracy can be improved compared to prior FT-approaches, enabling more sophisticated debugging without requiring live-interaction.

Our lockstep is effective with very low checkpoint frequencies, requiring few checks in second intervals. Hence the supervisor is no performance bottleneck for the system as a whole. Therefore, high-performance MPSoCs can be well supervised using pre-existing discrete COTS supervisors. COTS MPSoCs will utilize an external supervisor, while ASIC, FPGA and FPGA-SoC-hybrid based MPSoCs can implement this functionality in reconfigurable logic. An off-chip supervisor can be used for ac-

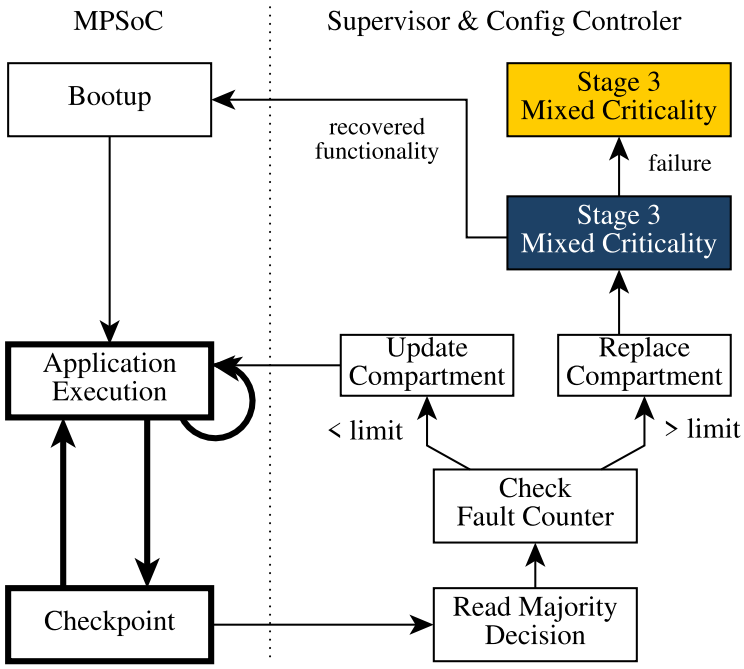


Figure 26: A compartment's and supervisor's program-flow and their interactions. Stage 1, 2 and 3 logic are indicated in white, blue and yellow respectively.

tive compartment health-management and FPGA reconfiguration, enabling the use of FPGA reconfiguration. See Chapter 10 for further details the supervisor interface.

4.5 Stage 2: MPSoC Reconfiguration & Repair

The previous stage can compensate faults as long as healthy compartments are available to replace defective compartments. In all existing hardware-side FT implementations, resource exhaustion is mitigated through over-provisioning (adding more spares). Over-provisioning of compartments naturally is inefficient and curtails system scalability, but is certain due to the static, unchangeable nature of existing ASIC based solutions. This will inevitably result in resource exhaustion, and has not been solved in prior work.

Stage 2 is designed to perform active compartment health management and test, repair, validate and recover faulty compartments, thereby tackling this fundamental limitation. In FPGA-based systems transient faults can corrupt the stored configuration of programmed logic, thus induce permanent effects within the running configuration [215, 216]. However, even if a logic cell is damaged permanently the residual highly-redundant FPGA fabric will remain intact and can be re-purposed [217]. It could be repaired with differently routed, functionally equivalent configurations.

The main issue preventing prior research from utilizing FPGA reconfiguration to increase FT of general purpose computing architectures is a lack of non-invasive, flexible circuit level fault detection. As efficient fault-detection for configurable logic is an

unresolved issue, Stage 2 relies upon fault-detection by Stage 1.

The functionality of Stage 2 is depicted in Figure 27. The supervisor will first attempt to recover a compartment using partial reconfiguration. Afterwards, the supervisor validates the relevant partitions to detect permanent damage to the FPGA (well described in, e.g., [218]), and executes self-test functionality on the compartment to detect faults in the compartment's main memory segment and peripherals. If unsuccessful, the supervisor will repeat this procedure with differently routed configuration variants, potentially avoiding or repurposing permanently defective logic.

Assuming a MPSoC architecture outfitted with compartments (see Section 4.7) is used, compartments are topologically isolated. Thus, reconfiguration of just one compartment will not impact the other compartments and allow the OBC to recover a compartment in the background. If reprogramming was unsuccessful or fabric-level faults persist, the supervisor will repeat the previous step with differently routed configuration variants. Partially defective logic cells can be re-purposed, while other cells can be avoided entirely, if no other usage is possible. Other elements of the FPGA fabric can be treated equivalently. The supervisor can also attempt full reconfiguration implying a full reboot of all compartments.

Stage 2 can also test different on-chip memories, the processor cores, and peripheral controllers through external interconnect access ports (e.g., an AXI-bridge). If the OBC is implemented on an ASIC or with a COTS MPSoC, a widely available low-level debug and testing interface such as JTAG can be utilized for the same purpose. Further details on reconfiguration and error scrubbing with a microcontroller-based

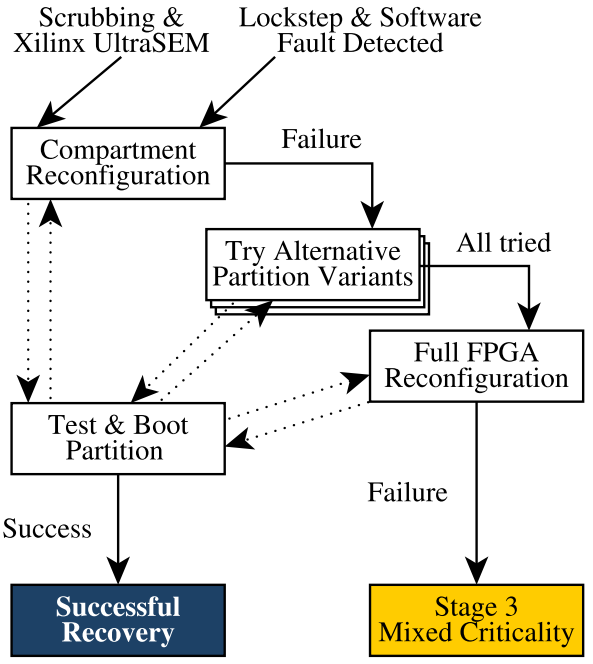


Figure 27: The objective of Stage 2 is to recover defective compartments and other logic through partial and full FPGA reconfiguration. If this is unsuccessful as well and no further spare processing capacity is available to handle future faults, Stage 3 is activated to find a more resource conserving application schedule, replenishing the spare resource pool.

proof-of-concept implementation for a nanosatellite are available in Chapter 5.

If a defunct compartment can not be repaired through automated reconfiguration, additional diagnostic information can be used for further analysis. The operator can utilize this information to conduct fault analysis on the ground, to craft a suitable replacement configuration to avoid these areas. Of course, this implies extreme development effort but for many higher-priority space missions, the loss of a spacecraft may be more costly than the engineering costs for saving the mission.

4.6 Stage 3: Applied Mixed Criticality

Stage 3 utilizes thread-level mixed criticality to extend an OBC's lifetime once the previous stages have depleted all spare resources. Its primary objective is to autonomously maintain system stability of an aged or degraded OBC at short notice to avert loss-of-mission and loss-of-subsystem, even if an OBC approaches the end of its lifetime. The operator can then define a more resource conserving satellite operations schedule, sacrifice link capacity, or on-board storage space. Thus, dependability for high-criticality threads can be maintained by reducing compute performance, throughput, or increasing latency of lower-criticality applications.

The criticality of applications executed on an OBC can be differentiated by the importance of the controlled subsystem or relevance for commandeering the spacecraft. Performance degradation or even a loss of lower-criticality tasks aboard a satellite is in general preferable to a loss of system stability for key applications. As thread groups can be added and removed from compartment groups, and multiple compartment groups can coexist in the same MPSoC, individual threads can also be migrated between compartment groups [206]. Furthermore, the checkpoint frequency of a compartment group can be reduced to increase a compartment's computational capacity, or it can cease servicing low-priority interfaces.

The supervision logic is extended to reallocate thread-groups across the system based upon the thread's priority. Hence, if Stage 2 failed to reconfigure the OBC, the supervisor can generate new compartment-group assignments for threads with high priority and will attempt to retain existing assignments. Eventually, all healthy compartments will be saturated with threads, and no further assignments will be possible. Then, it can either allocate more mappings, providing lower-priority threads with less processing time to maintain availability, reduce the checking frequency, or leave them inactive. The OBC developer can decide at design time, which applications would benefit most from continuous operation with reduced performance or reliability, and which can be forgone.

In practice a satellite operator can use this functionality also to dynamically adjust the performance of the MPSoC mid mission. This is achieved by adapting the distribution of applications across compartments, the level of replication of application threads, and the processing time allocated to individual application threads. The three properties, thus, are in competition to each other, as depicted in Figure 28. This capability is analogous to the powersaving capabilities present in today's mobile devices and consumer desktop computers, where performance and energy consumption objective compete. An optimal combination of these objectives exists only in theory, but in practice would be very costly to obtain. For practical use, a set of "good enough but non-optimal" can be achieved as at runtime autonomously using heuristics. Further information on Stage 3 including dynamic thread-mapping, as well as performance,

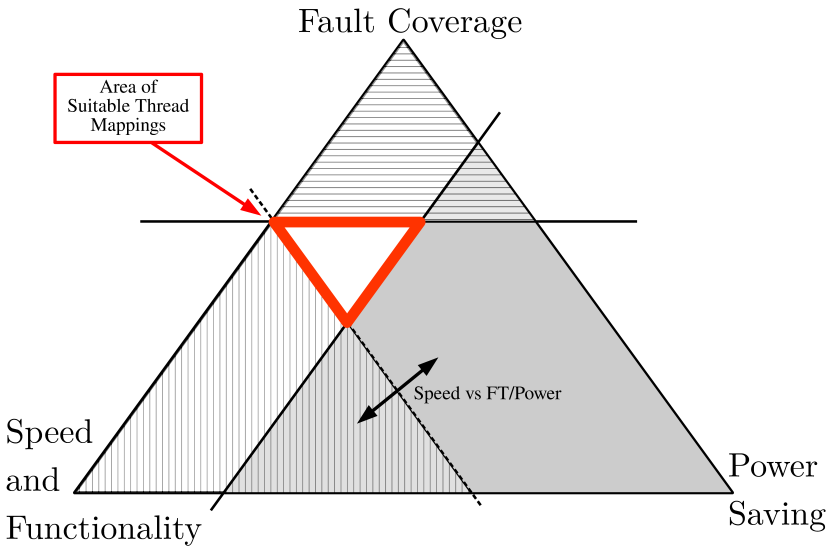


Figure 28: Our architecture allows the system properties of fault tolerance, performance, and energy consumption of an OBC to be adjusted at runtime. The spacecraft operator can prioritize one of these objectives, e.g., to achieve minimum energy consumption by sacrificing processing speed, while maintaining a given level of fault tolerance.

energy and robustness optimization at run-time is available in Chapter 6.

In Figure 29, initially two compartment groups are executed on one MPSoC with 6 compartments. The first group consists of T_a and T_b executed on C_0 – C_2, to perform highly-critical platform management and control tasks. The second group performs payload data handling tasks and is initially run on C_3 – C_5, and runs its lockstep at half the frequency as the higher critical group mentioned before. It consists of two threads, with T_c acting as payload subsystem driver task of medium criticality, and a computationally expensive low-criticality application T_d performing data compression. In the first checkpoint cycle, a fault occurs on C_5 which is detected after this group executes its first checkpoint. No spare processing capacity is left to replace the failed core with directly. C_2, however, still has sufficient spare capacity to accommodate T_c, but not T_d. T_c is migrated to a separate, new compartment group and executed on compartments 2 – 4, thereby maintaining strong FT. The lower-criticality task T_d remains degraded. Therefore, T_d will continue to run in DMR mode on the intact cores C_3 and C_4, which only allows fault-detection in the future.

4.7 Platform Architecture

Our multi-stage FT-approach is in principle platform independent and can be implemented within any multi-threading capable OS supporting interrupts and timers. For most COTS-MPSoC based nanosatellites in a LEO orbit, stage 1-3 alone offer sufficient fault coverage. Aboard such spacecraft, MPSoC interfaces are either unprotected or protected programmatically and outside the MPSoC (e.g., using EDAC chips or by resolving SEFIs through power cycling). Aboard larger, more critical spacecraft

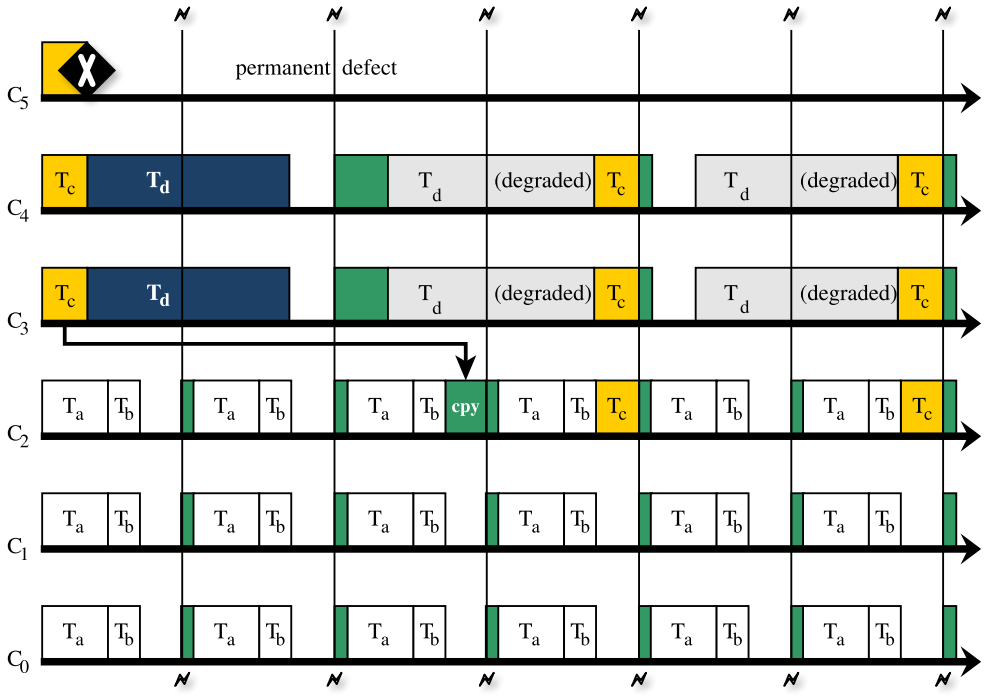


Figure 29: If no healthy spare compartments are available, the Stage 3 can split defunct compartment groups and uphold FT guarantees for high-criticality threads. The necessary adjustment to the checkpoint frequency on compartment 2 is omitted for simplicity.

such faults can not be accepted, and OBC interfaces are usually implemented redundantly at great effort. This redundancy is inherent to our approach with due to the compartment-based architecture, and we developed an MPSoC platform capable of surviving the loss of peripheral devices and permanent, non-resolvable defects in interfaces.

4.7.1 MPSoC Architecture Concept

This MPSoC can be implemented in full using library IP available with standard industry FPGA or ASIC design tools without custom FT components. We have implemented our MPSoC prototype with Xilinx Vivado standard IP, AXI Interconnects, for low-tier ARM Cortex-A processor cores to be provided by one of our industrial partners. For common space applications, size-optimized cores such as the Cortex-A32, -A35 and A5 offer an excellent balance between performance, universal platform support and logic utilization. The architecture minimizes shared logic, compartmentalizes compartments, and offers a clearly defined access channel between compartments for sharing checkpoint-results and application-state. We are aware that most miniaturized satellites do not require such a high degree of fault coverage, and often can not afford the added hardware complexity and development effort.

The MPSOC depicted in Figure 30 follows a compartmentalized architecture. The software run on the individual processor cores is strongly isolated from each other. It

is meant to be implemented within an FPGA to counter resource exhaustion when mitigating faults in Stage 1. It utilizes simple redundancy to compensate for SEFIs, but does not contain radiation-hard or FT processor cores or custom logic. Each compartment is equipped with a processor core, an interrupt controller (IRQ in the figure), a dedicated on-chip memory slice used as state memory, and several peripheral interfaces through the local interconnect. Compartments are connected through an I/O memory management unit (IOMMU) and a global interconnect to main- and non-volatile memory. They can not access the local interconnect of other compartments to prevent interference and minimize shared logic. This compartmentalized architecture benefits from partial reconfiguration, as compartments can be placed strategically on an FPGA's fabric along partition borders. Our approach and this architecture support multi-FPGA and -ASIC MPSoCs without adaptation, thereby improving scalability and resilience against FPGA-level SEFIs.

The ECC-protected dual-port state memory in each compartment holds the current compartment-status, thread assignments, as well as the checksums and state information. One interface is connected to the compartment's local interconnect, while the second port is read-only accessible via the global interconnect. The state memory is inherently redundant, as threads are executed on at least two compartments. The

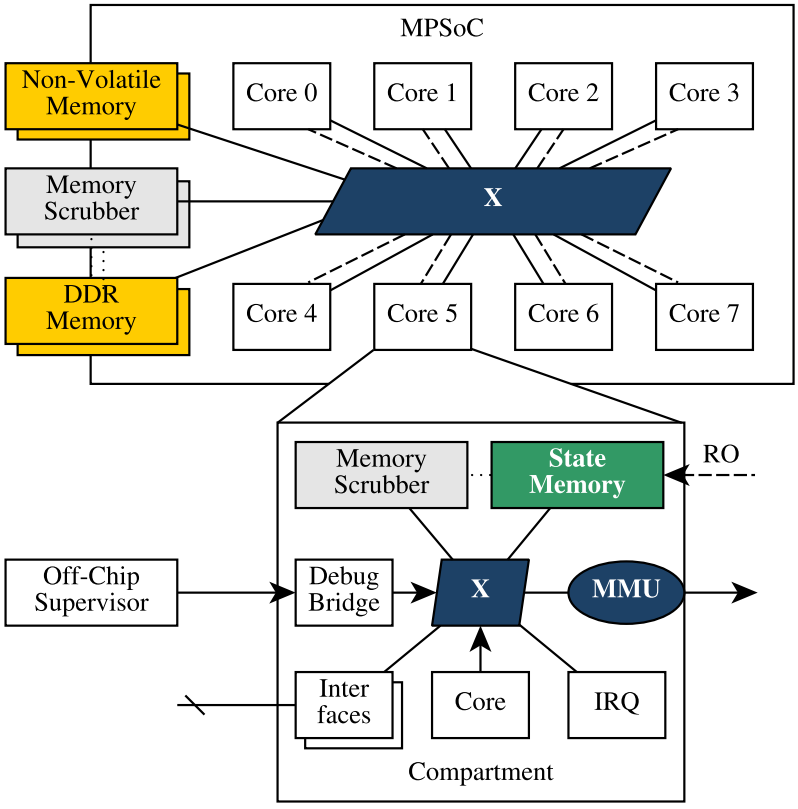


Figure 30: A simplified representation of the presented MPSoC with memory controllers highlighted in yellow, scrubbers in green, and interconnect in blue. A dedicated interface on each compartment allows supervisor access.

shared main memory is redundant to safeguard from SEFIs affecting the compartment-shared interface. Both instances are ECC protected and connected to the global interconnect. The main memory is split into several segments: each compartment has write-access to its own segment, and can read the global shared code segment. ECC-fault syndrome interrupts for main memory are handled by the supervisor. We perform error-scrubbing on these memories to avoid accumulating bit-flips due to transient and permanent faults. The scrubbing frequency should be set depending on the actually used memory technology, production node and mission parameters. Non-volatile memory is implemented redundantly as well. Our prototype is designed to utilize radiation immune MRAM and PCM [197] and we realize advanced FT for these memories as described in Chapter 7. Each compartment's main memory segment, state memory, and non-volatile memory are mapped to the same compartment-local address ranges. At the thread-level, the address-space in each compartment is thus identical, making application and OS code location independent and allowing compartments to share binaries. Further implementation details are available in Section 4.10.

4.7.2 Feasibility

We developed an early MPSoC design based on the this architecture utilizing exclusively library-IP. Instead of ARM cores, this quad-core demonstration design includes Xilinx MicroBlaze processor cores, as these are more available to the general public. It targets standard FPGA development boards and is equipped with a single shared DDR4 main memory controller, and 2MB on-chip BRAM program memory. This reduced design was implemented successfully using the Xilinx Vivado Design Suite and Stage 1 was implemented using FreeRTOS and using the Xilinx SDK toolchain.

Each compartment is outfitted with data and instruction caches, an interrupt controller, a UART interface, state memory and an additional local memory for storing compartment-private information, and a GPIO controller to signal agreement between compartments. All compartment-local memories are equipped with ECC, as this increases logic size of the relevant memory controllers, and includes two additional interrupts for each connected memory. We could achieved full timing closure at 250MHz core frequency on VCU118 and KCU116 development kits, though the clock frequency

Resource	Utilization	Available	**Utilization %**
LUT	68,705	1,182,240	**5.81%**
LUTRAM	9,235	591,840	**1.56%**
FF	92,536	2,364,480	**3.91%**
BRAM	810	2,160	**37.48%**
DSP	27	6,840	**0.40%**
IO	163	832	**19.59%**
BUFG	17	1,800	**0.94%**
MMCM	6	30	**20.00%**

Table 3: Resource utilization of the quad-core demonstration MPSoC on a Xilinx VCU118 development board. The on-chip program memory and DDR4 memory controller disproportionately inflate BRAM utilization.

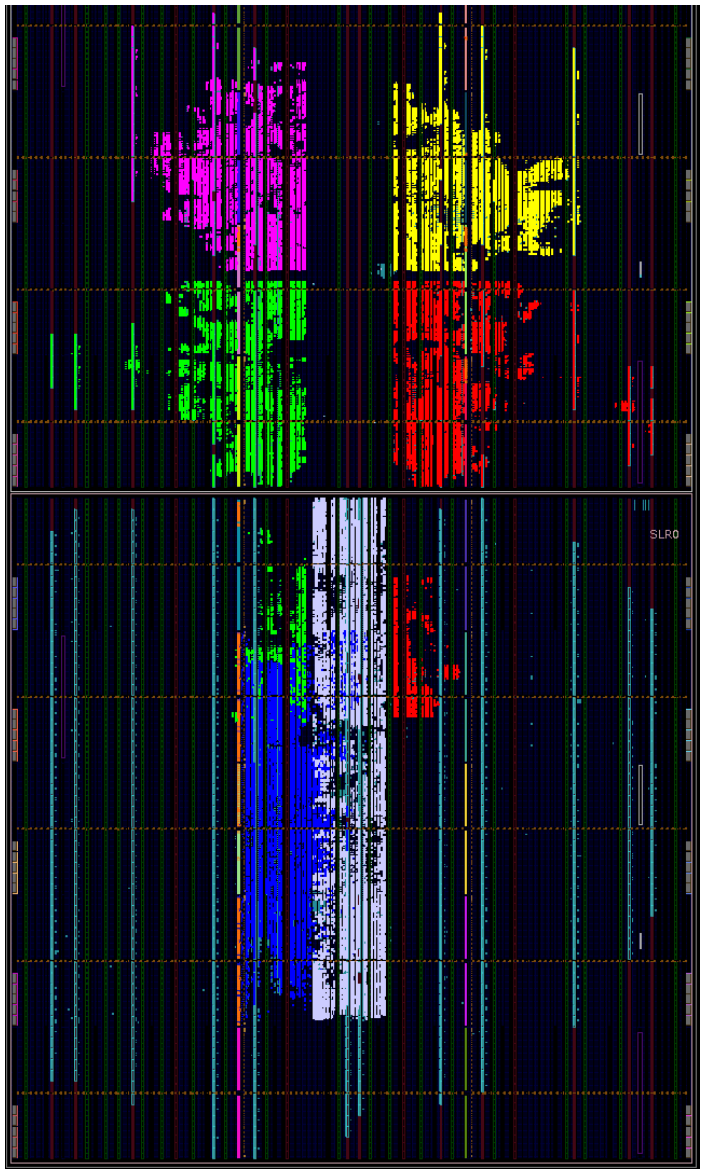


Figure 31: Logic placement of the demo-MPSoC on a VCU118 development board running 4 Compartments: green, red, yellow, pink; Global Interconnect: white; Xilinx DDR4 controller: blue; Program Memory: teal.

was selected to achieve a simple design, not an efficient or fast one. If additional time was invested into timing optimization and clocking, the clock speed can be drastically increased. Additional information regarding the compartment and SoC layout are available in Chapters 9 and 10.

Fabric utilization based upon the Xilinx Virtex VCU118 Development Kit is depicted in Figure 31. Due to the use of on-chip program memory and the DDR4 memory controller, BRAM utilization is inflated compared to the MPSoC described previously.

Resource utilization is indicated in Table 3, with more details given in Section 4.10. Stage 2 and 3 do not require additional FPGA logic.

This design's very low logic usage shows that the architecture itself can be scaled to 8 and more compartments comfortably, and most current-generation FPGAs offer an abundance of unused resources for Stage 2. With current-generation FPGA platforms, Stage 2 will thus not only be able to recover defective compartments using spare resources, but could even place multiple compartments as cold or hot spares. The Microblaze cores utilized here for demonstration purposes can directly be replaced with more powerful processor cores, assuming the necessary peripheral IP is added as well (e.g., an ARM GIC instead of the MicroBlaze Interrupt Controller).

4.8 Discussions

The reliability of each individual compartment's voting decision can be weak, and an individual compartment can report false (dis)agreement with its siblings. Our approach takes into account that any software or hardware component associated within a compartment can fail arbitrarily. Such failure is mitigated through a distributed decision, which is taken based on each compartment's perspective of its siblings. Thus, this approach does not require the checksum logic to compute correctly, and we assume that faults may occur at any time during the lifetime of a compartment. As compartment groups usually consist of three or more compartments, the likelihood of false-disagreements or non-reported disagreement is insignificant. To mask such a fault, multiple faults would have to coincide in a majority of compartments within the same compartment group during a single checking period and induce the same fault. The probability for such an event is extremely low, except at very high radiation levels. Even in such situations, such faults would be detected after the subsequent checkpoint with near certainty.

Prior research proves the conceptual effectiveness of thread-based FT [88,200] and software-based FT combined with simple I/O voting [201]. Also, the detailed FT capabilities of a platform utilizing our approach are influenced by the actually used FPGA, ASIC or COTS-MPSoC design. These imply mainly design decisions and a varying acceptance of single-points-of-failure. Schedulability, timing conformity, and deadlock-avoidance have been extensively researched in literature, e.g., in [210]. Thus, what remains to be shown is the runtime performance overhead induced by the presented approach, as the main objective of our research is to enable the efficient use of high-performance mobile-market COTS MPSoCs within satellite computers. To achieve worst-case performance estimations, we developed a naive, unoptimized implementation of the Stage 1 of our approach, as the others do not affect the runtime performance of the MPSoC. This naive implementation shows a median-best performance degradation of 9% and median-worst degradation of 26% on compartments with a single processor core. Further information on the conducted tests is available in Section 4.10, as well as performance measurements for 6 different application scenarios modeled after the NASA/James Webb Space Telescope's Mid-Infrared Instrument (MIRI) [219].

As prior thread-level FT implementations [199, 200, 208] are based upon fundamentally different concepts, only address transient faults within a very limited scope, and are deeply embedded into proprietary OS, their fault coverage and performance can not be directly compared. However, the measured performance overhead does fall

within the same range as measured in [199], and we also observe comparable average-case performance. To put these measurements into context, even a 50% slowdown on modern MPSoCs will offer a factor-of-5 performance increase over state-of-the-art radiation-hardened processor designs, thereby showing a favorable cost-vs-benefit trade-off.

4.9 Conclusions

In this chapter, we presented the first practical and integral multi-stage approach to fault-tolerant (FT) general purpose computing for spaceflight use. The approach explicitly does not utilize radiation-hardened or hardware-FT processor cores and utilizes no central MPSoC-internal voting logic. It can thus be implemented within COTS MPSoCs or alternatively entirely with non-FT, standard library IP-cores available in FPGA or ASIC design software. In contrast to prior research, the presented approach considers the full and realistic fault-model for space computing, and operates within real-world constraints. The approach does not require failure-free components within an MPSoC or in the OS, and does not leave conceptual gaps, e.g., regarding fault detection and recovery. It is not based upon traditional radiation-hardened processor cores and does not achieve fault tolerance through hardware-measures.

We showed that our approach is programmatically simple and requires little custom code, which can also be implemented in most pre-existing multi-threading capable OS. Faults can be detected and mitigated using application provided routines, enabling decisions about an application's integrity to be taken by the application developers themselves. As a consequence, the system designer no longer must struggle to assess the health of each individual application's state, and instead can focus on determining an optimal solution to problems at hand. It allows flexible fault-detection, mitigation and recovery within COTS MPSoCs, laying the foundations for FT computing aboard miniaturized satellites, and helping to bridge the gap between theoretical embedded research and practical implementation in the space industry. While remaining flexible, and inducing only a minimal performance overhead, the presented multi-stage approach offers time-bounded real-time guarantees.

The approach can be well complemented with several other reliability-improving measures which were integrated into the outlined reference MPSoC architecture. Preliminary benchmark results of an unoptimized implementation show a low performance overhead, suggesting a beyond factor-of-5 performance increase over state-of-the-art radiation-hardened processors for space use. Our approach allows the host platform to scale vertically (more powerful processor cores and more interfaces per compartment) as well as horizontally (more compartments), with virtually any modern processor core. Thereby, we aim to increase acceptance for software-side FT approaches in the space industry, building trust in hybrid hardware-software architectures. Thus, our approach is the first integral, real-world solution to enable the fault-tolerant application with modern MPSoC designs for critical satellite control applications, thereby enabling the use of such SoCs in future high-priority space missions.

4.10 Annex: Worst-Case Performance Estimation

To achieve worst-case performance estimations, we developed an unoptimized implementation of the first stage of our approach in C to be run in user-space. The provided benchmark results were generated based on code derived off a special CCD readout program used for space-based astronomical instrumentation. The application was executed with a varying amount of data processing runs in a compartment group at the indicated checking frequencies, and without protection for reference.

4.10.1 Implementation Outline

This implementation was written in approximately 800 lines of user-space C-code including benchmark facilities. It utilizes system calls and the POSIX threading library to simulate compartments and thread management. Thread-management at this level is computationally much more expensive than if performed bare-metal or in kernel-code. A bare-metal implementation within an operating system reduces this performance overhead drastically. This implementation therefore allows very pessimistic benchmarking, which can yield a baseline for the lockstep's performance cost. The implementation also serves as an excellent simulator to validate the correctness of the described logic, and allows better debugging than on the actual MPSoC implementation.

4.10.2 Test Application

Synthetic, widely used benchmark suites are unsuitable to benchmark OS-level functionality. Thus, we derived a demo-application off an astronomical instrumentation application. We chose to utilize the background scenario of scientific computing, as devices for scientific instrumentation are usually better documented. The program flow of our demo application is based on the NASA/James Webb Space Telescope's Mid-Infrared Instrument (MIRI) described in [219]. This program continuously reads three 16-bit 1024x1024 false-color sensor arrays, stores, and processes the results. It averages multiple captured frames to optimize the instruments exposure time and avoid pixel saturation, or to capture faint astronomical sources [219].

4.10.3 Methodology and Test Setup

The setup simulates an MPSoC three compartments executing the described demo application, and measures performance of the application executing within a compartment. For each plot in Figure 32, 100 measurements were taken of the real-time necessary to process 600 1-Megapixel frames with subsequent processing runs. Data heavy modes indicate a high amount of post-processing runs, whereas compute-heavy modes indicate lower per-thread workload.

- Very Compute Heavy: 60000 Postprocessing Runs
- Compute Heavy: 75000 Postprocessing Runs
- Balanced Compute Heavy: 90000 Postprocessing Runs
- Balanced Data Heavy: 105000 Postprocessing Runs

- Data Heavy: 135000 Postprocessing Runs
- Very Data Heavy: 150000 Postprocessing Runs

Benchmark results were generated on a Intel Core I7-2600K Sandy Bridge-based system with a host kernel's scheduling frequency of 1kHz (`CONFIG_HZ_1000`). Hyper-Threading and SpeedStep was disable to avoid interference between threads. Binaries were compiled with `GCC 6.3.1 (20161221)` without compiler optimization (`-O0`).

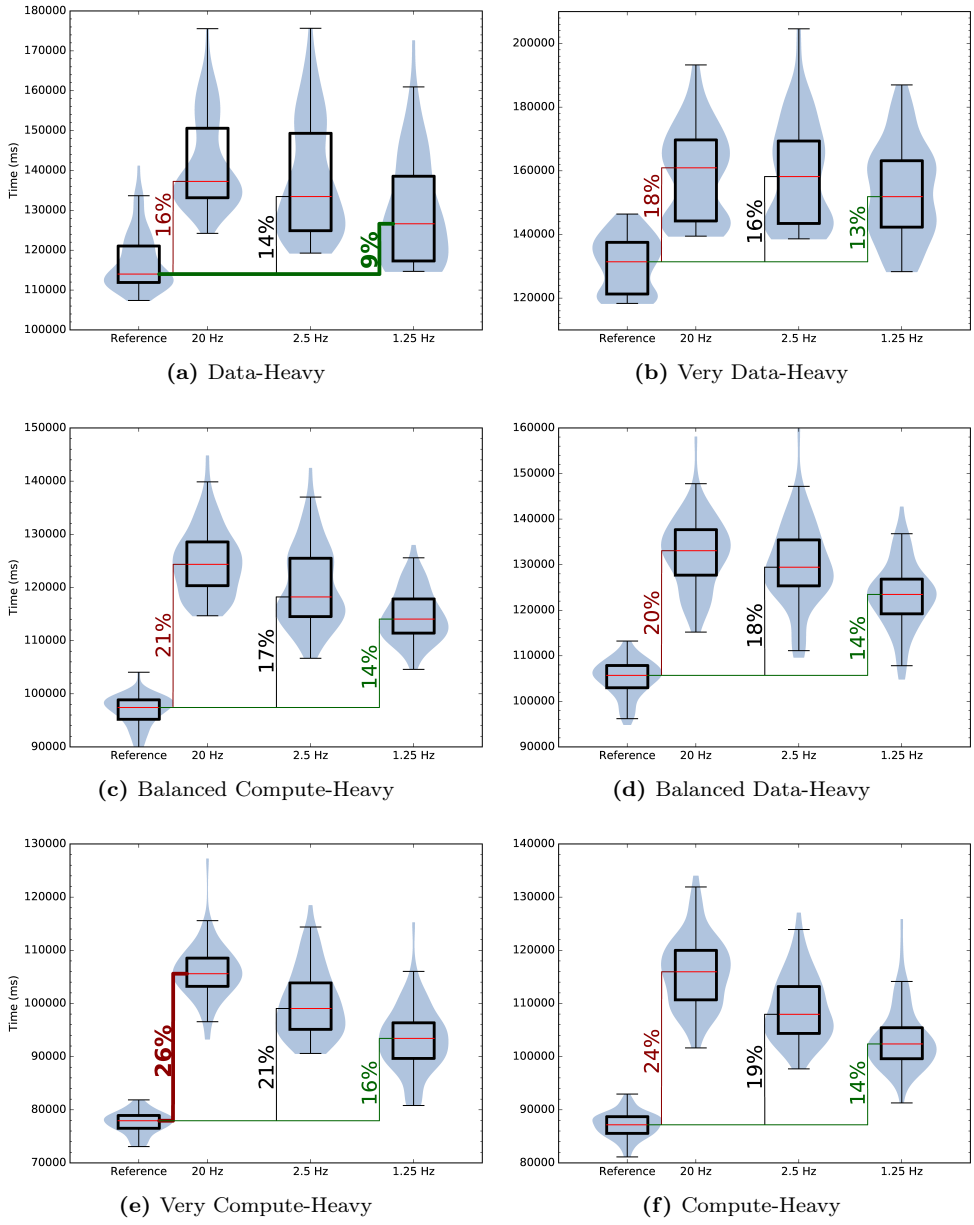
4.10.4 Results

This naive implementation of our approach at the application level on Linux shows median-best performance degradation of 9% and median-worst degradation of 26%, which are also indicated in Figure 32a and e in bold. Across all test runs, we measured on average 80% worst-case and 95% best-case performance compared to the unprotected reference runtime. The violin plots – shadows around the box-plots – indicate the distribution of the measurements to depict the accumulation of the individual measurements.

As expected, the performance varies depending on workload, with data-heavy tasks a-c showing better performance. This too was expected as Stage 1's code consists mainly of integer operations, binary comparisons, load/stores, and jumps. Better performance can be expected in a more optimized implementation at the kernel level due to a reduced computational cost of operations that in userland require system calls. To put these measurements into context, even a 50% performance degradation on modern MPSoCs will offer a factor-of-5 performance increase over state-of-the-art radiation-hardened processor designs.

Assuming an average performance degradation between 10% and 20% at such extreme checking frequencies, our approach can thus allow a modern MPSoC to perform better than comparable state-of-the-art hardware-voting based processor solutions, while requiring no proprietary processor design, offering full software-control at a fraction of the development effort and costs. And in contrast to existing hardware-based fault tolerance solutions, our architecture does not struggle against feature-size reduction, but scales up with technology and benefits from more modern production nodes.

The lockstep was run with very high checkpoint frequencies (20hz, 2.5hz and 1.25hz) which during normal operation will most likely never be used. For most LEO applications, we expect that checkpoints would be run only every 5 to 10 seconds. Furthermore, system calls and thread-management on high-performance mobile-market processor cores can be much less costly than when run on desktop hardware. Realistically, this would implying very little performance cost ranging from 0.5% to 2% overhead.



(a) Data-Heavy

(b) Very Data-Heavy

(c) Balanced Compute-Heavy

(d) Balanced Data-Heavy

(e) Very Compute-Heavy

(f) Compute-Heavy

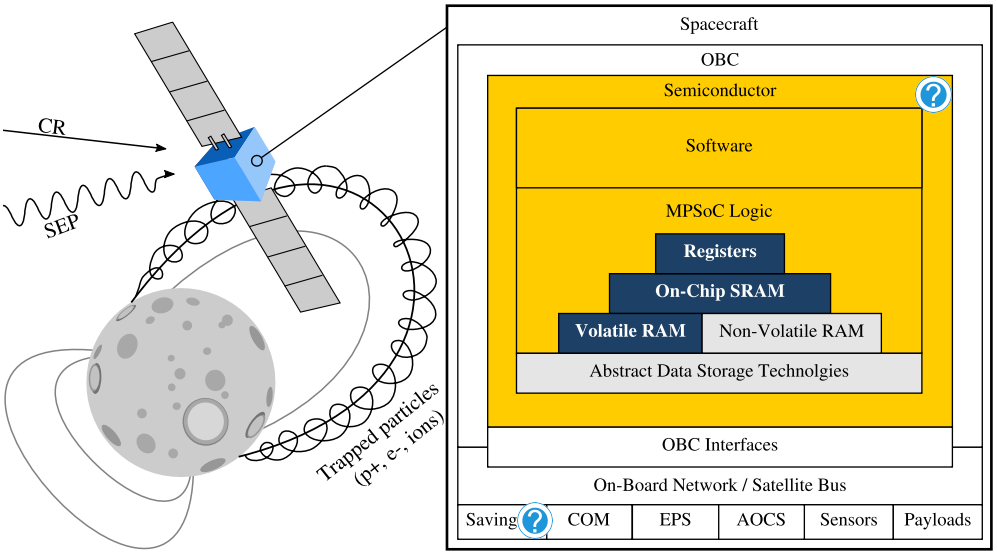
Figure 32: Performance measurements of 6000 runs for processing 100 1024x1024 pixel CCD frames with different checkpoint frequencies and workloads.

Chapter 5

MPSoC Management and Reconfiguration

Stage 2

In this chapter, we describe the functionality of the second fault-mitigation stage of our architecture and address RQ2. Stage 2's functionality was originally developed as a saving-subsystem for the MOVE-II CubeSat and was meant to perform autonomous chip-level debugging only. Within the system architecture described in this thesis, it now fulfills the role of the FPGA's supervisor*, but the concept itself predates the architecture described in the previous chapter. In the context of this thesis, remote debugging is one among several tasks this component performs: it controls the coarse-grain lockstep of Stage 1, conducts FPGA configuration management, and handles thread-allocation within the system for Stage 3. It safeguards the integrity of the FPGA-fabric, may repair defective processor cores through partial reconfiguration, and can offload tasks to the configuration controller implemented within the FPGA. Thereby, it can increase the long-term fault coverage of the system as a whole.*



5.1 Introduction

Nano- and microsatellites have evolved from purely educational projects to fit a diverse range of commercial and scientific use-cases. This class of satellites can do so by combining rapid development, reduced design complexity, low manpower requirements, and minimal cost through a reliance on commercial off-the-shelf components (COTS). Modern embedded technology enables a high level of compute performance at the cost of little energy. Miniaturized satellite development has begun to rely upon conventional application processor architectures as well as FPGAs. Hence these satellites can nowadays offer an abundance of storage capacity and compute performance [220].

CubeSats have proven to be both versatile and efficient for various use cases. They have also become platforms for an increasing variety of scientific payloads and commercial applications [32]. However, such missions require an increased level of dependability in all subsystems compared to educational vessels, especially to enable their use within critical missions and for such with prolonged lifetime requirements. Currently, miniaturized satellites are plagued by low dependability, and will be requiring failure tolerance and reliability enhancing measures in the future. Due to the limited budget, mass and volume restrictions within miniaturized satellite projects, such measures usually must be achieved using means beyond replication and redundancy.

Data storage and processing applications can be protected using architectural and software side approaches, combining them into hybrid solutions. However, even utilizing such hybrid concepts, component level failure tolerance remains limited using only COTS hardware. Acceptance of eventual failure of an on-board computer (OBC) due to issues beyond the control of the deployed flight software without a viable recovery strategy in place is a tolerable approach for educational satellites. However, especially when deployed in larger quantities (e.g., constellations), failure diagnostics and recovery measures that do not require the active cooperation of an OBC or its operating system should be available.

In contrast to larger vessels, the use of chip-level debug functionality aboard miniaturized satellites has up until now largely been restricted to the development and testing phases. During system development and testing on the ground, low-level debug interfaces are usually used for diagnostics, debugging and failure analysis, providing chip-level access to satellite hardware. However, such functionality often lays dormant once the satellite has been deployed or is not even activated in a satellite OBC's flight model. Thus, debugging functionality has rarely been utilized in-orbit aboard CubeSats, as the necessary protocols could not be implemented over the unreliable low-bandwidth links without major effort.

Few nanosatellite projects possess the manpower and time to implement sophisticated failover functionality and testing effort until a very late phase during development when facing non-trivial bugs. Many CubeSat developers also are unaware of the challenges of hardware development, and therefore ignore low-level debug functionality in satellite design altogether. In contrast to debugging capabilities, flight software reprogramming functionality is usually desired aboard nanosatellites. Hence, several CubeSats were equipped with simple proprietary update solutions [221–223]. Even though the capabilities of these concepts were limited with little re-use potential, they underlined the importance of software-independent chip-level debug functionality such as JTAG [6].

Hence, began exploring how a miniaturized satellite's saving subsystem could be

outfitted with chip-level debugging capabilities in late 2014, and developed a concise concept in early 2015 and implemented the prototype described in this chapter in late 2015. We designed this subsystem to enable extensive debugging and analysis support for the MOVE-II CubeSat [Fuchs13], as prior experiences in the field and especially in the FirstMOVE predecessor CubeSat showed that this functionality is critical [Fuchs17]. It is designed to support testing, verification, and debugging on the ground as well during a space mission. It offers scripting support through the use of STAPL [224] bytecode which is then translated into JTAG operations using a STAPL virtual machine, thereby offering near universal test-target support. Hence, the subsystem's software can remain static at run-time and does not need to be changed throughout a space mission. The multi-stage fault tolerance architecture described in Chapter 4 is a direct evolution of the concept described in this chapter. In the remainder of this thesis, this saving subsystem also takes on the role of the MPSoC's supervisor, integrating most of the usage concepts described in Section 5.4.

In the next section, we will analyze how and why debugging at chip level can help improve dependability. We outline why this functionality up until now is largely unavailable aboard miniaturized satellites, and what functionality is required to implement such a saving subsystem. Section 5.3 then contains a description of our work and offers insight into several key aspects of the developed concept. Afterwards, use cases beyond mid-mission debugging are presented in Section 5.4. We discuss plans for future work and present our conclusions in the final two sections.

5.2 Debugging and Reliability

Testing and error diagnostics are critical tasks during hardware development, and thus also when developing nanosatellites. While larger spacecrafts' OBCs have extensive debugging support, CubeSats usually offer no equivalent functionality and, if at all, resort to creative ad-hoc testing solutions. Most such solutions can not deliver equivalent functionality to the comprehensive set of testing and debugging features often encountered within COTS hardware or aboard larger spacecrafts. Besides functionality, the reliability and universal usability of these solutions is often insufficient, resulting in few CubeSats fielding any form of software-independent mid-mission capable fault analysis functionality. In consequence, few CubeSats nowadays offer sufficient fault detection, isolation and recovery functionality (FDIR) to reliably detect and recover from hard- or software malfunctions.

Most system-on-chip architectures, FPGAs, and many other ICs provide JTAG test access ports (TAPs) [6]. Originally developed for circuit testing, JTAG nowadays is the de-facto standard chip-level debugging interface and is widely used in electronics for larger satellites. Hence, JTAG is an ideal interface for sophisticated fault detection, isolation and recovery in case of component failure. In addition, it can be utilized to update an OBC's software, firmware, as well as to control and reconfigure the programmable logic of an FPGA. We argue that chip-level debugging is currently not widely used because there are no readily available CubeSat-compatible solutions that can be adapted to a wide variety of different designs.

The properties of the communication bands utilized for commandeering aboard contemporary CubeSats (usually UHF and VHF, see Chapter 3), the constrained up- and downlink availability, and the low bandwidth make mid-mission debugging challenging. As discussed in Chapter 3, these restrictions result in constrained data rates

around tens of kbps, even if strong error correction is utilized. As ground station networks and satellite relay systems at the time of writing are not accessible to ordinary nanosatellites, debugging and error diagnostics must be conducted fully remotely. JTAG requires bi-directional real-time communication and is sensitive to timing issues, aspects which are not suitable for satellite links in general and especially the links available aboard miniaturized satellites. Hence, the chip-level debugging must be decoupled from the satellite link, so that live-interaction during debug sessions only happens locally within the spacecraft.

STAPL scripts can be executed autonomously and perform all timing-critical operations locally within the space segment. Thereby, we can terminate the timing-critical aspects of chip-level debugging while minimizing link congestion. The saving subsystem described in this chapter can, thus, efficiently operate even via a lossy, unreliable very-low-bandwidth communication channel. It can operate even in environments with elevated radiation levels, requires little PCB space, low power and entails minimal cost.

5.3 Implementation Details

The main objective of the research described in this chapter is to improve overall reliability and survivability of a spacecraft. Hardware complexity has been a major issue in CubeSat projects, often resulting in oversimplified systems due to lack of experience and sometimes even in overly complex systems due to uncontrolled feature creep. Due to the absence of sophisticated FDIR functionality, even minor hardware and software may cause a CubeSat to become unrecoverable.

In the remainder of this section, we will discuss the MOVE-II CubeSat specific implementing of our saving subsystem using an Microchip/Atmel SAM7SE MCU. However, it should be noted that besides the hardware choices outlined in this chapter, there are numerous other MCUs which could be utilize instead. Originally, the this saving subsystem was intended to integrate into an existing Spartan 6 LX45 FPGA on MOVE-II's transceiver module. However, due to the densely populated transceiver board and insufficient FPGA resources on the LX45, a microcontroller (MCU) based implementation was developed instead.

In the context of this thesis, we instead chose to utilize a radiation-robust TI MSP430FR MCU, as we describe further in Chapters 9 and 10. A SAM7SE offers considerably more performance than an MSP430FR MCU. However, the tasks this saving subsystem is meant to perform within the architecture described in Chapter 4 require little performance, and MSP430FR MCUs have been shown to perform exceptionally well under radiation [225].

5.3.1 Hardware Requirements

The saving subsystem can be implemented with comparably basic hardware, however, we must also consider assuring integrity of the subsystem itself. MRAM [150] and phase-change memory (PCM) [226] both are ideal technologies for holding saving subsystem's code and stack segments, as their storage cells are radiation immune. At the time of this writing, no affordable highly-reliable nanosatellite-compatible hardware that could be used to implement the presented saving subsystem is available. Thus, we have to resort to utilizing COTS MCUs and minimize fault potential. This MCU must provide the following functionality:

- an external memory interface to attach a parallel magnetoresistive RAM (MRAM [150]) to contain the saving subsystem's code, or an MCU with internal MRAM. However, we are unaware of the existence of COTS MCUs equipped with sufficient MRAM.

- A second memory interface will be needed to access flash memory to store larger chunks of data such as FPGA configurations operating system updates. Once PCM or STT-MRAM with larger capacities [227] becomes widely available, the saving subsystem could also be implemented using just one large memory IC.

- The saving subsystem does not require a real-time clock, as we intended the saving subsystem to be as static and stateless as possible. However, we still must assure precise timing for certain operations requiring at least a counter/timer.

- We also must be able to interface with at least one JTAG chain which we can best achieve using a set of general-purpose I/O pins. The capability to access additional JTAG chains enables more advanced usage scenarios.

The program code of the saving subsystem resides in a write-protected MRAM region, whereas the stack segment will be kept within a separate writable region. Thus, faults in the running system's state can be resolved through a reboot in many cases. In consequence, it can then resolve or remove leftover information from the (corrupted) previous system state and thereby recover to a consistent system state. The saving subsystem's (runtime-static) firmware, in turn, can be protected from corruption through erasure coding as described in Chapter 7. Redundancies for MCU and memories can be added as necessary, and are omitted from this chapter for the sake of briefness.

5.3.2 STAPL Scripts and Commandeering Interface

The subsystem offers extensive scripting support through the use of the STAPL scripting language, which is then translated into JTAG operations using a STAPL virtual machine [6, 224]. Hence, the saving subsystem's program code can remain static at run-time requiring no modification to the virtual machine's code. As the STAPL scripting language is Turing-complete[1], it can be utilized to implement arbitrary sequences of JTAG operations in the form of STAPL scripts, achieving code separation and time triggered execution. By using STAPL scripts, we can thus avoid timing critical aspects of chip-level debugging aboard the satellite while minimizing link congestion. Thereby, the saving subsystem can be efficiently operated even over a unreliable very-low-bandwidth communication channel, which would otherwise make chip-level debugging infeasible.

We chose to utilize the STAPL bytecode format [224] to minimize script- and code-size while retaining flexibility. These scripts as well as all relevant program code and state information must reside within radiation tolerant MRAM. Even though STAPL bytecode is more compact than the text based equivalent, experiments have shown that more complex scripts can still become as large as 50kB.

Due to the limited memory capacity in MRAM, only few scripts can be uploaded to and stored permanently within the STAPL machine. For the sake of simplicity, we

[1] in our context it most importantly supports recursion and jumps

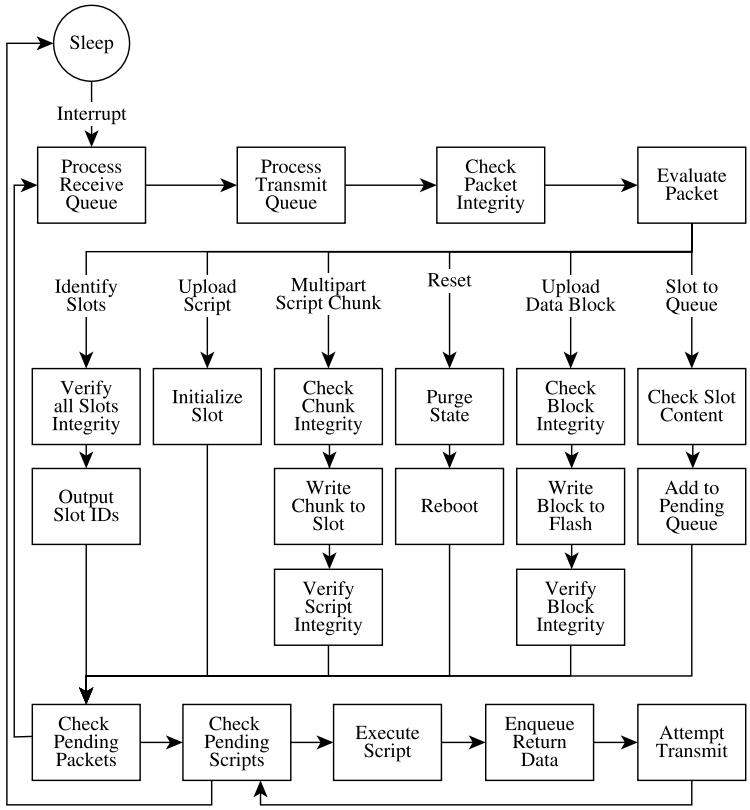


Figure 33: A visualization of the saving subsystem's program flow and commandeering protocol we developed around the Altera JAM player.

utilize a compile-time space distribution, creating a fixed number of identically sized script slots. Each slot can only hold one script, even if the script does not utilize entire entire capacity of a slot. The original implementation of this saving subsystem utilized 2MB of MRAM, and we implemented 10 x 50kB sized slots leaving 1.5MB of MRAM for the stack and code segments.

In the current implementation, slot allocation is managed at the ground segment by the satellite operator and we currently support only equally sized slots. A potential future optimization would be to utilize differently sized slots (e.g., 5×10kB slots, 5×50kB slots, 2×100kB slots), to achieve better resource utilization. We implemented static slot management to minimize code-complexity and failure potential.

Slots are identified by a CRC16 checksum used as reference for commandeering, and also for integrity checking of an individual script. This checksum is uploaded with each new script, and verified once the transfer of all script-parts has been concluded.

An additional identifier beyond this checksum is unnecessary. The low number of scripts minimizes the chance of checksum-collisions due to the birthday paradox [228], Operators can avoid collisions altogether through padding scripts on the ground.

Scripts are directly committed to a slot and then checked for integrity to minimize data duplication and resource usage. Hence, we can assure that only uniquely identified, correctly and completely uploaded scripts will be executed.

5.3.3 Transfer of Large Scripts and Data Housekeeping

The maximum frame size supported by the communication modules of most nanosatellites is considerably smaller than the script size, hence the saving subsystem supports multipart transfers for scripts and other data. A multipart script transfer initialization packet contains the intended slot ID to be overwritten, the expected script checksum and size, as well as the chunk size. The initialization packet also provides a null terminated array of checksums for each to be expected chunk.

For each active multipart transfer, the saving subsystem retains a list of missing frames. It notifies the ground station in case the final missing chunk has been received, or upon command. For slots, this information is stored within the slot header. Later packets indicate the chunk-offset, to facilitate simple retransmission.

FPGA configuration variants and software updates for the OBC can be as large as several megabytes. Hence, they must be stored in dedicated heap memory and multipart transfers of such data is conducted akin to multi-part scripts. We decided to perform allocation and data management on the ground, instead of implementing dynamic heap memory management. Again, this implementation decision was made to minimize software complexity and failure potential. As all operations executed by the saving subsystem must be pre-planned by the operator, more advanced allocation mechanisms do not result in operational advantages.

We utilize flash memory to store larger data volumes outside of the script-slots as neither PCM nor larger MRAM chips are currently widely available. As this data is not executed, we can utilize flash memory and store the data using erasure coding in software. However, in STAPL scripts all payload-data is usually encoded inline and cannot be omitted without modifications to the scripting language syntax.

For this purpose, we extended the STAPL syntax to also support references to external data. We replace inline data with a reference to data in flash, which can then be uploaded independently. Therefore, the STAPL Bytecode player was modified to make it capable of side-loading auxiliary data.

The results of scripts, e.g., kernel dumps, system state information and other diagnostics data, are thus also held in flash memory until they can be transmitted to the ground station. Script execution can be triggered in bulk, hence outgoing packets are being stored in a FIFO queue for transmission. A more detailed representation of the saving subsystem's program flow is provided in Figure 33.

To safeguard against data corruption due to space radiation effects (single- and multi-event upsets), coarse symbol level Reed-Solomon erasure coding [229] will be applied when writing to flash memory [230]. As flash memory with comparably low density is utilized, no additional layers of erasure coding are necessary but could be implemented, see Chapter 7. Reasons for utilizing higher-density flash memory may be the requirement for storing more partial reconfiguration partition variants to cover the increased number of permanent faults that can be expected in space missions with longer duration, or to provide feature-diversity as described in Section 5.4.3.

5.3.4 Integration into an On-Board Computer

Our current saving subsystem implementation consists of an ARM7TDMI MCU with an OBC-independent communication channel toward the CubeSats transceiver or saving subsystem as depicted in Figure 34. We chose to utilize an interrupt-driven bidirectional SPI-based interface to implement this channel due to its flexibility and

simplicity. Also, this interface is less prone to implementation issues than I^2C, however there are many other alternatives and the saving subsystem's concept does not foresee a specific interface. The saving subsystem is attached to a single four pinned JTAG chain, containing all to be debugged JTAG enabled devices. Due to abundantly available GPIO pins, additional JTAG chains could be attached with ease once the software has been adapted.

The Microchip/Atmel SAM7SE MCU is able to boot from memory attached to its external interface, has excellent toolchain support, documentation and minimal energy consumption. Attached to the external memory interface are an Everspin 2MB MRAM memory chip as well as 16MB of NAND Flash. The MRAM chip is connected to the 16-bit memory interface and used to store the program code, scripts, and also serves as main memory. The use of the SAM7SE's internal memories is avoided whenever possible since radiation hardness cannot be achieved here. Only the MRAM address ranges used as main memory and for STAPL scripts and the stack segment are writable by software, all the rest of the memory is set read-only through the ARM7TDMI's MPU.

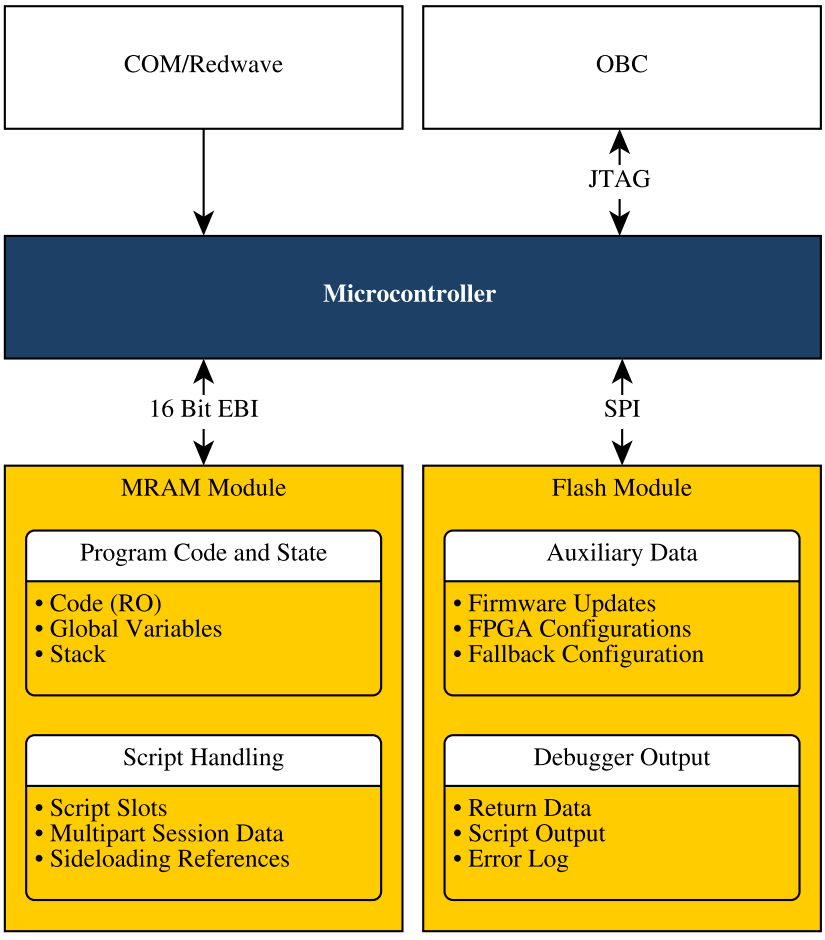


Figure 34: A component-level view of the saving subsystem.

Since the MCU reads its output data from different external memories at different clock rates (MRAM and flash), JTAG clock frequency is dynamically adapted. However, the JTAG clock speed is capped by the maximum support frequency of the debugging target. Dynamic clocking and the use of lower JTAG frequencies are common. Therefore, the duration of one clock cycle is variable, resulting in drastically varying clock speeds especially if access to flash memory is necessary.

Figure 35 shows the hardware setup used to port the saving subsystem from the original proof-of-concept implementation to embedded hardware. It includes a SAM7SE512 MCU and two external memories:

- 16MB SDRAM to simulate the MRAM, and
- 256MB flash memory.

All components and interfaces besides the SDRAM correspond to the originally intended design of the saving subsystem. The commandeering interface has been successfully tested with several self-contained scripts as well as such referencing external data. The saving subsystem currently implements the commandeering API depicted in Figure 33 directly. In a future version of this implementation, we plan to replace the SAM7SE512 MCU with a radiation-robust MSP430FR microcontroller, to reduce failure potential, and as this saving subsystem has very low performance requirements.

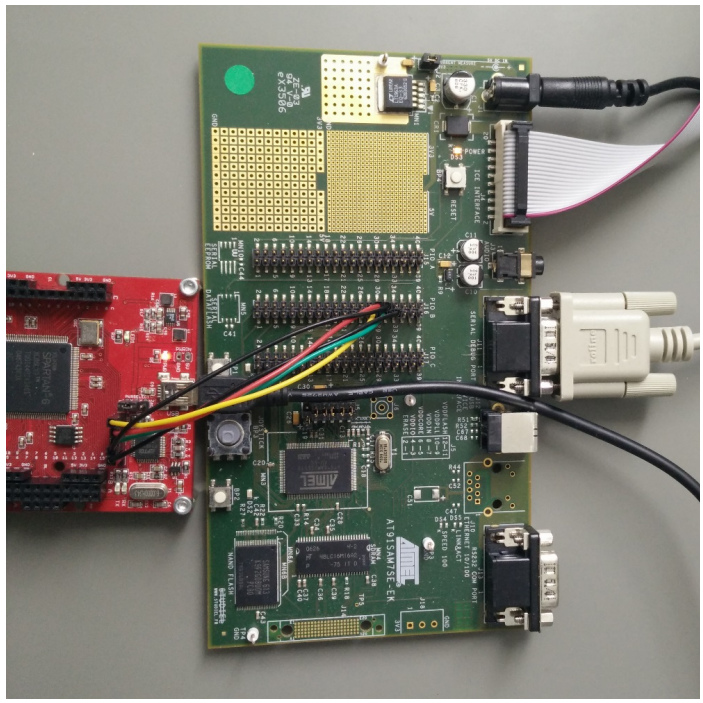


Figure 35: A saving subsystem demonstration setup utilizing the SAM7SE (green PCB) and external NAND-flash and external SDRAM. In this picture, the system was interfaced with a Xilinx Spartan 6 FPGA (red PCB to the left) and validated the FPGA configuration. Due to the concepts simplicity and flexibility, the saving subsystem can be implemented in full just a micro-controller development board.

5.4 Use Cases beyond Debugging

While the presented subsystem was developed primarily for FDIR reasons, there are several additional use-cases that were considered during design. The saving subsystem could be extended with additional functionality or may even be used outside of its originally intended usage scenario aboard a spacecraft. Hence, we dedicate this section to discuss other use cases for this saving subsystem beyond traditional LEO CubeSat applications.

The main limitation of the saving subsystem within a CubeSat application scenario is storage capacity and buffer size to return data via a satellite link. However, these limitations mainly affect the following capabilities:

- size and number of slots available within the saving subsystem,
- storage space for referenced data such as FPGA configurations and
- to-be-returned information and logs, and finally the
- total size of FPGA configurations.

For ground applications and even aboard vessels only slightly larger than 1U CubeSats, these restrictions can easily be lifted.

5.4.1 Watchdog Integration

The saving subsystem can be interfaced with a watchdog to achieve extended functionality. This watchdog could notify the saving subsystem about malfunctions within other components of the OBC. The saving subsystem could then begin recovery measures, enabling considerably better fault-recovery and logging possibilities than the usual reset triggered by CubeSat watchdogs. Instead of directly rebooting the OBC into a (presumably) safe mode, the saving subsystem can first collect relevant log information (i.e. retrieve register contents and a stack-trace). Once this information has been stored, it can then be directly reported to the ground station. Also, this functionality could be adapted, e.g., to take into account known permanent faults that may have occurred in a previous mission phase.

We have not yet implemented this functionality, as the described logic first would have to be written as STAPL script and is highly hardware and software dependent. To avoid the saving subsystem's return-buffer from being flooded with crash-logs in case of frequent or repeated crashes, additional logic must be implemented. A simple mitigation method would be a message queue implemented as a ring buffer. Then only a fixed number of diagnostics messages would be retained at any given time, assuring that only the most recent logs are retained and transmitted to the ground.

As watchdog functionality is usually rather simple, it could also be provided by the saving subsystem itself. Integrated watchdog functionality would only require minimal additional code and could be combined more efficiently with the script-driven state machine. However, such functionality is usually considered critical and malfunctions of the watchdog code within the saving subsystem could cripple the rest of the OBC. Hence, watchdog functionality should only be integrated if a suitable interface setup can be achieved, as described see Chapter 10).

5.4.2 FPGA Management and Validation

Radiation tests of several COTS FPGAs such as Xilinx' Spartan 6 FPGA family have yielded promising results. Recent radiation testing activities have shown the Spartan 6 devices to be outright latch-up immune and largely unaffected by event effects [231–233]. While these devices are not truly radiation-hard, they can offer a sufficient level of reliability to be used aboard spacecraft if scrubbing and other software-driven dependability measures are applied.

In contrast to using a discrete processor or a classical SOC design, an FPGA based Soft-SOC could provide drastically improved OBC flexibility. As an FPGA can be programmed with largely arbitrary logic, a broad variety of interfaces and processor architectures can be utilized. Such interface logic thus no longer has to be implemented in hardware using separate controller ICs, and can directly be attached to the FPGA. System parameters can be modified and interface assignment can be changed even mid-mission. Also, permanent faults in reconfigurable logic could be worked around by deploying a similar configuration avoid the use of a certain FPGA area. Hence, one of the main drivers for the saving subsystem's design was to control an FPGA and take full advantage of programmable logic devices.

As depicted in Figure 36, the saving subsystem can not only control and reprogram an FPGA, it can also be used to implement more advanced usage scenarios: A continuous read-verify-repair cycle could be scripted and executed in a timed manner to enable scrubbing and reduce the impact of transient errors [234]. As most radiation effects within FPGAs are transients, thus temporary errors, their impact on the system can be reduced even if radiation-soft SRAM FPGAs were used.

While access to the running configuration of an FPGA is comparably well documented, access to attached configuration memory requires slightly more effort:

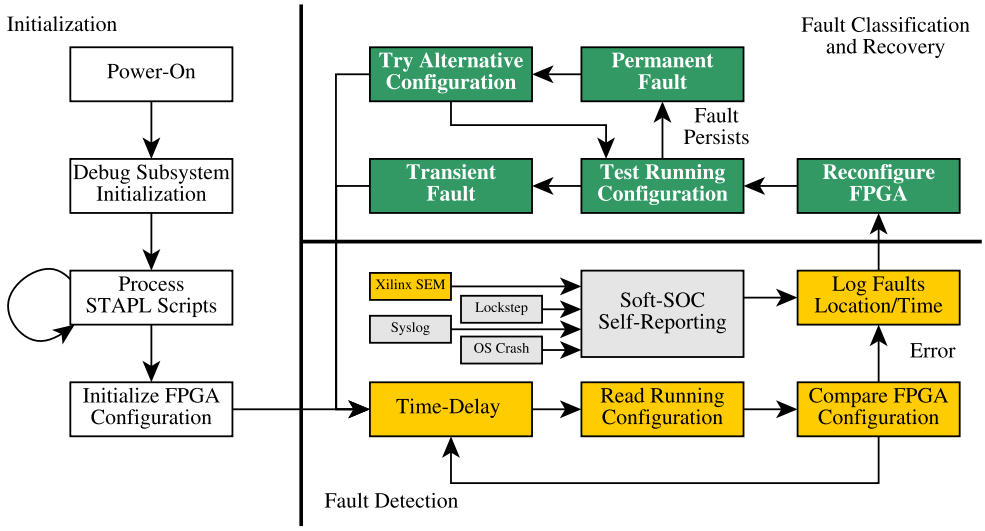


Figure 36: The saving subsystem can also be adapted for radiation testing and FPGA integrity assurance in space. In this case, the saving subsystem can implement all functionality necessary for MPSoC supervision as described in Chapters 4 and 6.

- In case the running configuration is still functional, the saving subsystem can access such memory via the system bus through a separate JTAG bridge.our Such bridges are standard IP-cores and readily available for many platforms (e.g., AMBA/AHB, AXI, ...) and often are even foreseen in the platform specification for system debugging (i.e. GRLIB). In Chapter 10 we realize this functionality through an SPI2AXI bridge.
- For simple interfaces such as SPI, a multi-master setup with both the FPGA and the saving subsystem driving configuration memory can be realized. Again, we utilize such a setup in Chapter 10.
- Otherwise, a separate FPGA configuration must be uploaded to function as a JTAG bridge.

On some FPGA platforms, the second approach is being performed using nested configurations (nested bit-files). An FPGA configuration implementing a JTAG to SPI interface is used to transfer the actual configuration bit file into the configuration memory. Even though this interface requires minimal logic and usually covers only few slices on an FPGA, the total size of an FPGA configuration is still determined by the size of the FPGA. Compression can be used to reduce this dead-space, thus the JAM player foresees ACA [224] compression. However, the saving subsystem then still has to store multiple bit-files.

5.4.3 Flexible OBC Provisioning for Advanced Missions

The saving subsystem can also reconfigure an OBC with several different FPGA configurations for reasons beyond FDIR. More complex space missions consist of several different phases with varying duration and requirements towards the OBC as depicted in color in Figure 37. Using traditional discrete processing components or write-once anti-fuse FPGAs, the properties of a system are static and can not be modified later on. An $n+1$-voting circuit can deliver a fixed amount of compute performance and a certain level of dependability. Thus, if the OBC must be able to handle an increased compute burden or provide stronger integrity assurance guarantees for a certain mission phase, the system design as a whole has to be adapted.

To fulfill varying requirements, systems engineers usually resort to over-provisioning to assure system performance and failover capabilities. Thus, if additional compute performance was required for a voted SOC setup, system properties such as clock frequency and the number of processing cores being part of the voter could be increased. If this is insufficient, then a second, identical setup would have to be added to allow the system to scale with these requirements. Of course, the resulting system's efficient will thereby be reduced.

Additional compute resources or redundancy thus remain unused throughout most of a mission, increasing overall power consumption and system complexity. Dynamic FPGA configuration management based on mission phase requirements could drastically improve overall performance and reliability of an OBC design. As shown in Figure 37, the saving subsystem could provision different SOC variants with a varying number of processing cores and TMR strength depending. Provisioning could be conducted automatically based on the requirements of different mission phases. Thereby, instead of over-provisioning, an OBC design could be adapted to deliver a near-optimal level of performance, reliability, latency and power saving for each mission phase.

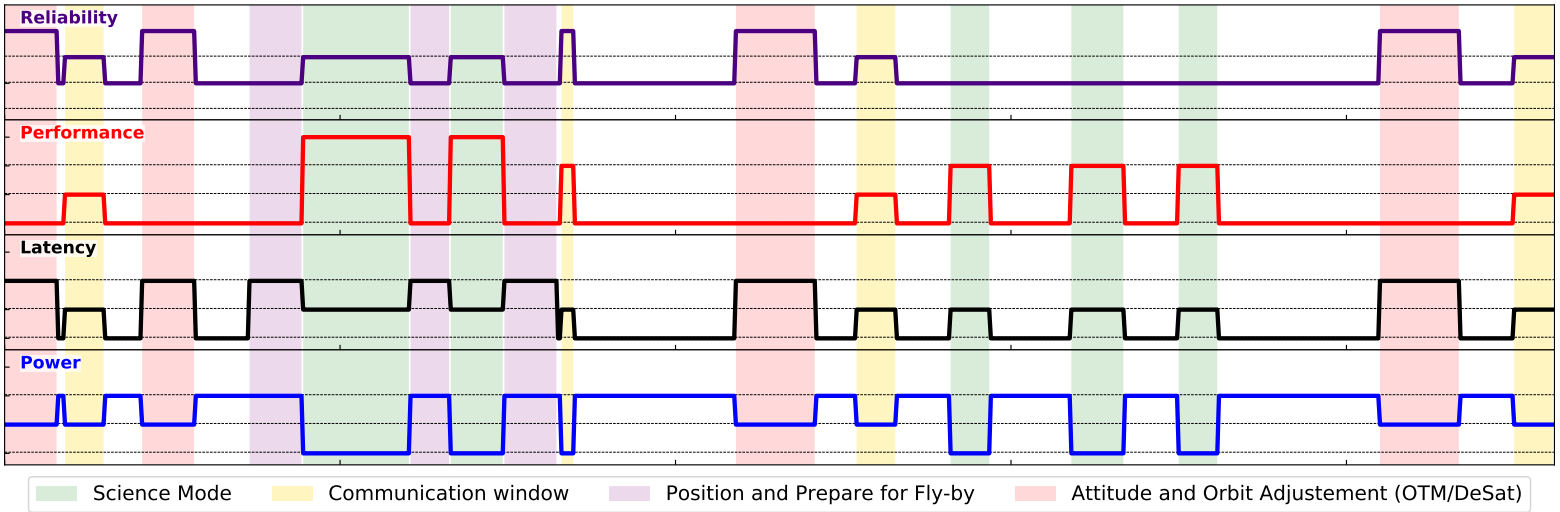


Figure 37: The different operational phases of an exemplary solar system exploration mission modeled after NASA's Enceladus Life Finder (ELF) mission. Each phase implies different requirements towards an OBC's reliability and robustness to faults (purple line), raw compute performance (red), power saving capabilities (blue) and latency to achieve a desired level real-time capability (black). The dashed lines in each plot indicate qualitative "high", "normal", and "reduced" levels of each properties that must be satisfied delivered by an OBC. E.g. in science mode, it will be beneficial to maximize system reliability and performance, and in turn accept an increased level of energy consumption for a this period of time. In communication or maneuvering modes, raw compute performance may be secondary to increased reliability (communications), or better real-time capabilities (maneuvering and orbit adjustment). Operational phases corresponding to those of Figure 20.

As an example, a regular TMRed system consisting of three active cores and one spare could be slit into two independent DMRed SOC pairs using a different SoftSoC configuration. As shown in the figure as well, during some phases of the mission, not all interfaces to other subsystems of the spacecraft are necessary. A separate FPGA configuration could be deployed which does not drive these interfaces to help conserve energy. Hence, the same chip on an unaltered OBC board could fulfill its role in a considerably more efficient way, resulting in efficiency improvements in all regards.

5.4.4 Radiation Testing and Profiling

There are also use cases for this concept on the ground, e.g., to substitute for equipment usually used for radiation testing and profiling of programmable logic or processor designs. To improve the quality of results on a device's behavior undergoing radiation testing, the subject device or FPGA should be continuously probed to log the type of radiation-induced errors when they occur. A post-mortem analysis hereby would only reduce the quality of information obtained and may even mask errors.

As outlined in Section 5.4.2, the saving subsystem can maintain a configuration scrubbing and reprogramming cycle. While the necessary hardware to do so has been developed in the past already, the saving subsystem allows improved flexibility while reducing the need for support equipment. To do so, the saving subsystem must be implemented using radiation hardened components, and the simple design and low performance requirements allow the use of primitive electrical components.

Instead of counteracting the effects of radiation events, the saving subsystem can log upsets within the running configuration of the subject device. Later on, this information can be forwarded to perform forensic analysis and look up which region of the configuration was affected and in what way. If combined with watchdog functionality as outlined in Section 5.4.1, the setup can also help assess the severity and impact of event upsets and can help to map critical logic. The saving subsystem can automatically determine information about which of the most recent upsets could trigger system failure within, e.g., Soft-SOC configurations. Of course, the saving subsystem can also make use of more advanced integrity control functionality and can therefore improve logging. It can directly utilize other information sources such as crash logs, information about software-handled errors, and faults detected by specialized IP (e.g., Xilinx Soft Error Mitigation [235]).

The saving subsystem can also perform scrubbing on an FPGA configuration, which allows further classification into transient and permanent errors, refining testing results. Hence, fault analysis can then be conducted using high-quality information and the results obtained can also be fed-back into the testing cycle, see Figure 36. This information could ultimately also be introduced into an FPGA design's testbench and can help simulate the impact of changes to design based on realistic information without performing additional radiation tests. Analysis suites such as SETA [236] could further help automate this process and may be used to obtain additional information from saving subsystem traces. The saving subsystem can thus drastically improve the quality of radiation testing results when working with FPGAs and can substitute a major part of the otherwise required testing infrastructure.

5.5 Discussions

Development of the saving subsystem currently is in the prototype stage and a successful proof-of-concept has been implemented. Therefore the next step is to integrate it with other components of a CubeSat on-board Computer. The protocol to interface with the communication module via SPI has to be implemented and tested thoroughly. Once the API has been adapted to this protocol, a custom hardware prototype with the respective memories can be implemented.

Also, the saving subsystem is currently based upon a set of development boards meant for rapid prototyping. It therefore must be condensed to a CubeSat compatible form factor. Testing in this case also requires a broad variety of STAPL scripts to be developed to assure code coverage during testing. These additional scripts will then also be utilized to support development of other subsystems and testing of the attached OBC. Performance measurements, including power consumption under load, execution speed of different debugging operations must be performed as well.

There are also several extensions to the current saving subsystem implementation that should be added, such as support for multiple JTAG chains. The current implementation relies on using only one JTAG chain for all devices connected to the debugger, subjecting it to the risk of failure. In case one of the JTAG chain members malfunctions and can not transport the test data signal, the chain is rendered useless and debug operations can not be performed. Support for more than one JTAG chain would allow access to, e.g., a SoftSOC to be implemented in parallel to controlling the FPGA itself. The to-be-executed script could then also select the correct JTAG chain, requiring only minimal modifications to the STAPL logic. This also opens up additional usage scenarios especially when combined with FPGA/SOC hybrids such as Xilinx's Zynq family and the more powerful FPGAs utilized to realize the proof-of-concept MPSoC described in Chapters 9 and 10.

5.6 Conclusions

In this chapter we presented a subsystem enabling autonomous chip-level debugging for nanosatellite OBCs. Until now, chip-level debug functionality had not been readily available aboard miniaturized satellites. If at all present aboard CubeSats, such functionality had largely been restricted to the development and testing phases. We are convinced that the low survivability of many earlier CubeSats can be attributed, among other causes, to low per system dependability and a lack of FDIR functionality. Hence, we developed this concept to provide a readily usable CubeSat compatible mid-mission FDIR solution for the nanosatellite audience.

We developed two prototype implementations up until now:

1. an initial proof-of-concept based upon a Raspberry-Pi to demonstrate the general feasibility of the saving subsystem and to determine requirements for further development.

2. An embedded implementation for an ARM7TDMI MCU in preparation to migrating the design to CubeSat compatible form factor.

The saving subsystem can be integrated into most CubeSat architectures requiring only a JTAG interface towards to-be-controlled devices. It is based upon a minimal set

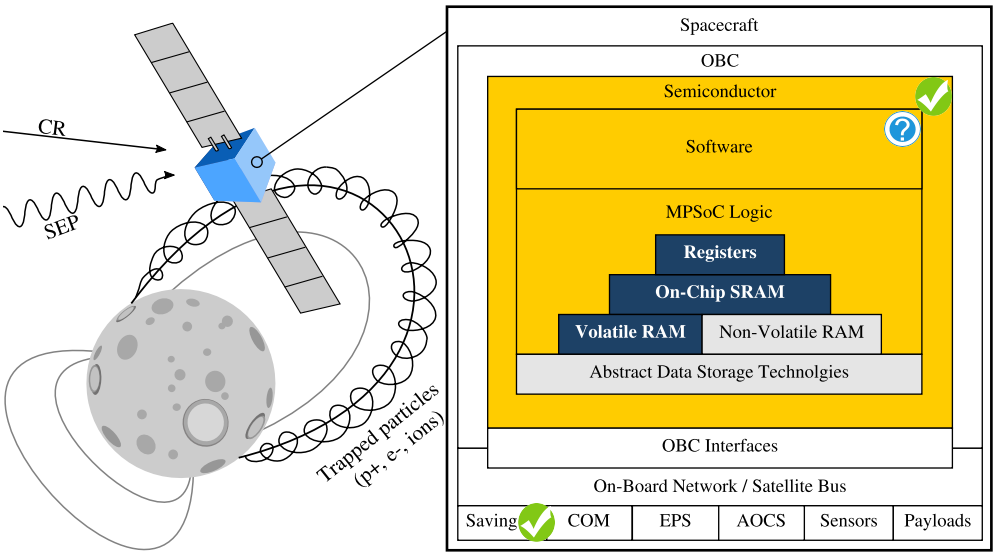
of components to retain simplicity, utilizing smart technological choices and erasure coding where necessary to achieve dependability using affordable COTS hardware. The presented design utilizes the STAPL scripting language and therefore can support a wide variety of devices. Due to its flexibility, several other use cases beyond debugging are imaginable, both in space and on the ground. The setup has been implemented successfully and thoroughly tested by controlling several ARM SoCs as well as FPGAs.

Chapter 6

Mixed Criticality and Resource Pooling

Stage 3

In this chapter, we discuss the third and final stage of our fault tolerance architecture. Stage 3 enables satellites of all weight classes to more efficiently handle accumulating permanent faults, and to age gracefully instead entering a degenerate state, thereby answering RQ3. We show how this functionality in conjunction with mixed criticality properties of a satellite's on-board computer can be exploited to improve robustness and efficiency. By modifying the application mapping within the MPSoC and adjusting thread-replication at runtime, the system can dynamically trade compute performance for functionality, robustness, and energy consumption at runtime. Considering our architecture as a whole, the the mechanisms discussed in this chapter exist in software and utilize extensively architectural properties of our MPSoC.



6.1 Introduction

Satellite miniaturization has enabled a broad variety of scientific and commercial space missions, which previously were technically infeasible, impractical or simply uneconomical. However, very small satellites such as nanosatellites and sometimes even microsatellites ($\leq$100kg) are currently not considered suitable for critical and complex multi-phased missions, as well as high-priority science applications, due to their low reliability. On-board computer (OBC) and related electronics constitute a large part of such a spacecraft's mass, yet these components lack often even basic fault tolerance (FT) functionality. Due to budget, energy, mass and volume restrictions, existing FT solutions originally developed for larger spacecraft can in general not be adopted. Nanosatellite OBCs also have to cope with drastically varying workload throughout a mission, which traditional FT solutions can not handle efficiently. Therefore, we developed a novel FT approach offering strong fault coverage, which was implemented fully using only a single FPGA with commodity processor designs, and library IP.

This architecture can protect generic applications with an arbitrary structure, can adapt to varying performance requirements in longer multi-phased missions, and can adapt to a shrinking pool of processing capacity similar to a biological system, efficiently handling aging effects and accumulating permanent faults. As major parts of our approach are implemented in or directly controlled by software, a spacecraft operator can configure the OBC to deliver the desired combination of performance, robustness, functionality, or to meet a specific power budget. To offer strong fault detection, isolation and recovery (FDIR), we combine software-side fault detection and mitigation and configuration scrubbing with various other FT measures across the embedded stack, enabling strong, low-cost FT with commodity hardware, while exploiting FPGA reconfiguration to mitigate permanent faults.

The next two sections contain background information, and a discussion of related work. In Section 6.4 a brief overview over the three stages of our approach is provided. Our proof-of-concept OBC-design is described in Section 6.5, with the functionality of each FT-stage outlined in the subsequent sections. How this approach can improve efficiency of OBC in spacecraft of all weight classes, spare resource utilization and fault coverage, is discussed in Section 6.6. Section 6.7, introduces *performance profiles* allowing a system-on-chips (SoC) to trade compute performance for energy efficiency, robustness, and functionality at runtime. Our approach provides advantages to spacecraft of all weight classes, and can be implemented also within distributed systems, for which further applications and improvements are discussed in Section 6.8.

6.2 Background

Tasks which would be handled by multiple dedicated payload and subsystem processing systems aboard a larger satellite, are usually handled by just one COTS-based command & data handling system in nanosatellites. These utilize mobile-market and embedded SoCs with one or more cores (MPSoCs), SDSoCs [40], or FPGAs [237]. Due to manufacturing in fine technology nodes, such chips offer superior efficiency and performance as compared to space-grade OBC designs, but are also non-FT[1]. These SoCs consist mostly of extensively tested and optimized standard logic, reused, supported,

[1]Exceptions to this rule received uncommonly abundant funding, are technology demonstration for FT concepts, or custom failover designs.

and evolved continuously by several industries and used daily by countless developers. In contrast, most radiation-hard-by-design (RHBD) processors cores, and SoCs manufactured in more robust manufacturing processed (RHBM) are crafted almost artisanally at high cost by few designers with little commercial stimulus for optimization. Their cost, energy consumption and mass often exceed such a spacecraft's global power budget, total mass, and almost always its overall project budget. Therefore, we developed a hybrid FT-approach based upon only COTS components, library IP, and existing software, instead of artisanal processor designs and proprietary instruction set architectures.

Existing hardware voting based FT solutions are design-time static and can tolerate a fixed number of failures within a voter setup, which can not be changed at runtime. Critical biological systems instead consist of independent, cooperating cells or clusters of similar functionality with a high degree of inherent redundancy and self-healing capabilities. Damage to a single cell is compensated by the remaining cells, and a complete breakdown of functionality occurs only due severe damage to the system at a broader scale. Our approach combines various FT techniques to mimic such behavior at the logic and SoC level, through FPGA reconfiguration and software-controlled thread migration within a globally share pool of processor cores, enabling graceful aging. The replication level, hence fault coverage capabilities, and various other parameters can be adjusted at runtime, while spare capacity can be reused to run background and lower-criticality applications instead of remaining idle.

In small feature-size chips, the energy threshold above which highly charged particles can induce faults in digital logic (single event effects - SEE) decreases, while the ratio of events inducing multi-bit upsets (MBU), and the likelihood of permanent faults in logic and memory increases. Increased fault coverage of hardware-FT based concepts on such chips through additional FT-circuitry therefore implies diminishing returns, preventing an application of traditional RHBD/RHBM concepts [104, 132] to mobile-market SoCs. Total ionizing dose, however, becomes less of a problem with finer technology nodes, and recent generation FPGAs also show decent latch-up performance [142,143]. FPGAs have drastically improved FDIR potential [238] despite being more vulnerable to transients, as radiation-induced upsets in the running configuration can be corrected via reconfiguration with alternative configuration variants [105].

6.3 Related Work

Fine-grained, non-invasive, and scalable fault detection in FPGA fabric is challenging, and subject of ongoing research [239, 240], and often is simply ignored in scientific publications [241]. Most FPGA-based FT-concepts rely on error scrubbing, which has scalability limitations for complex logic [239, 242], unless special-purpose offline testing is utilized [243]. In the future, memory-based reconfigurable logic devices (MRLDs) [244] may allow programmed logic to be protected like conventional memory, and thus would drastically simplify fault detection. If manufactured using phase/polarity-change memory instead of charge-based technologies, MRLDs could further increase robustness, but the memory technologies themselves are only emerging at the time of writing. In this chapter, we thus present an approach to general-purpose FT computing that compensates for faults across the embedded stack and through partial FPGA reconfiguration. We realize fine-grained fault detection at the software level, and perform scrubbing only as an auxiliary measure in the background

to increase robustness of our SRAM-based FPGA platform.

Hardware voting today is used exclusively for protecting simpler FT processor cores at the microcontroller level [88,104], and for accelerators [245] supporting application code with tightly constrained program structure. Hence, the application of this hardware-centered approach has become a technical dead-end for protecting widely used application processor designs intended for general-purpose computing, while accelerators by themselves would only assure FT for computation and data offloaded to such a device. In our research, however, we seek to deliver strong fault coverage for general purpose computing, and aim to efficiently protect even larger and more complex modern application processors, such as those widely used in mobile market and embedded devices.

Mobile market processors can run at gigahertz clock rates, for which hardware-side voting or instruction-level lockstep are non-trivial, hence, hardware voting approaches have been implemented only at lower clock rates [88,191,192]. For comparison, today's highly optimized COTS library IP achieves clock speeds comparable to traditional FT-processor designs on ASIC even on an FPGA, without requiring manual fine-tuning. We instead utilize software-driven coarse-grain lockstep to achieve fault detection, and maintain consistency between cores, requiring no vast arrays of synchronized voters, while utilizing COTS IP.

Thread migration has been shown to be a powerful tool for assuring FT, but prior research ignores fault detection, and imposed tight constraints on an application's type and structure (e.g., video streaming and image processing [241]). However, to implement sophisticated and efficient thread migration, fault-detection must be facilitated at the OS or application-level without falling back to design space exploration. Coarse-grain lockstep of weakly coupled cores can do just that, and in the past has already been used for high availability, non-stop service, and error resilience concepts. However, in prior research, faults are usually assumed to be isolated, side effect free and local to an individual application thread [208] or transient [199, 205], and entail high performance [209] or resource overhead [210, 211]. More advanced proof-of-concepts [198,199], however, attempt to address these limitations, and even show a modest performance overhead between 3% and 25%, but utilize checkpoint & rollback or restart mechanisms [199], which make them unsuitable for spacecraft command & control applications.

6.4 System Overview & Requirements

Coarse-grain lockstep is one among several measures used in our hybrid FT approach to facilitate forward-error-correction (FEC) and deliver strong fault coverage. Our approach consists of three fault mitigation stages:

Stage 1 utilizes coarse-grain lockstep for fault detection. It generate a distributed majority decision between processor cores.

Stage 1 utilizes time-triggered checkpoints to autonomously resolved faults corrupting the state of applications. It facilitates re-synchronization and thread migration in case of repeated faults, enabling strong **short-term fault coverage**.

Stage 2 assures the integrity of programmed logic by interfacing with Stage 1 and functionality such as Xilinx SEM. Its objective is to assure and recover

the integrity of processor cores and their immediate peripheral IP through FPGA reconfiguration, thereby **counteracting resource exhaustion**.

Stage 3 handles resource exhaustion and re-allocates processing time within the system to **maintain stability of critical applications and functionality in a degraded system**.

These Stages form a closed loop and implements FDIR in several steps as depicted in Figure 38. Additional information on Stage 1's thread-level coarse-grain lockstep, beyond what is briefly described in Section 6.5.1 are available in Chapters 4.

Stages 1 and 3 can be implemented separately on a generic MPSoC in low-end nanosatellites (e.g., 1U CubeSats). Then, they would provide a level of system-level robustness which otherwise would be only be achievable through proprietary hardware-FT solutions, without requiring the use of an FPGA.

For larger spacecraft, we complement this functionality with a compartmentalized MPSoC architecture for FPGA as outlined in the next section. It allows the system to recover defective compartments through reconfiguration, and enables it better handle permanent faults.

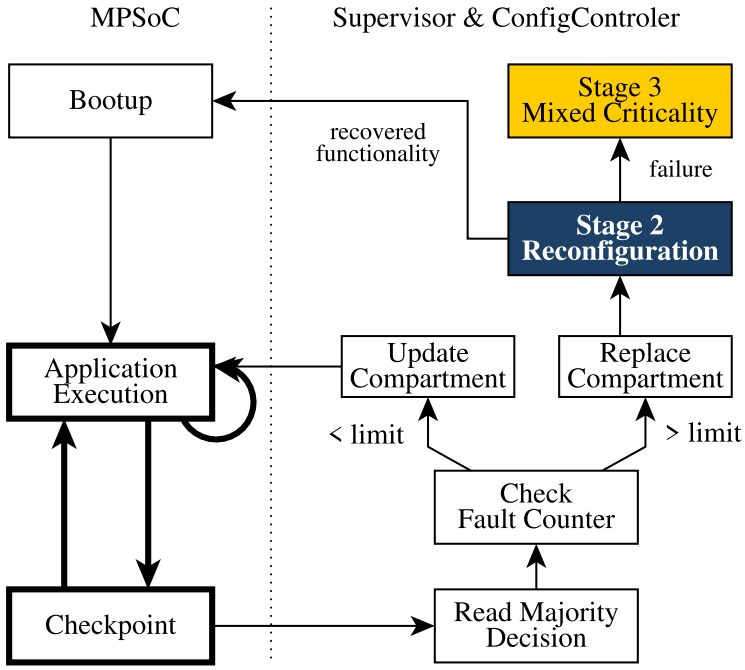


Figure 38: Stage 1 (white) implements a continuous checking loop, which facilitates fault coverage through thread-level synchronization and migration between compartments. Stage 2 (blue) can recover faulty compartments using reconfiguration. In case of resource exhaustion, Stage 3 (yellow) adapts the thread allocation to best utilize the remaining processing capacity.

6.5 System Architecture Review

Figure 39 depicts a simplified version of our MPSoC design. It follows a multi-core-like architecture with each compartment containing a processor core, local interconnect, and peripheral IP-cores and interfaces. A debug bridge allows supervisor access to each compartment, e.g., to perform introspection for testing purposes or to trigger a reset. The only globally shared resources are a set of redundant main memory controllers and non-volatile (nv) data storage. Code in nv-memory can be shared between compartments, while widely used DDR and SDRAM controllers are too large to instantiate for each compartment, and would require an excessive number of I/O-pins. Hence, our MPSoC architecture consists of isolated SoC-compartments accessing shared main memory and operating system code, in contrast to the conventional MPSoC designs, where cores share most infrastructure and peripherals.

Each compartment's checkpoint-related information is stored in a dedicated on-chip dual-port BRAM memory (*validation memory*) and exposed to other compartments, to allow low-latency information exchange between compartments without requiring inter-compartment cache-coherence or access to main memory. Validation memory is

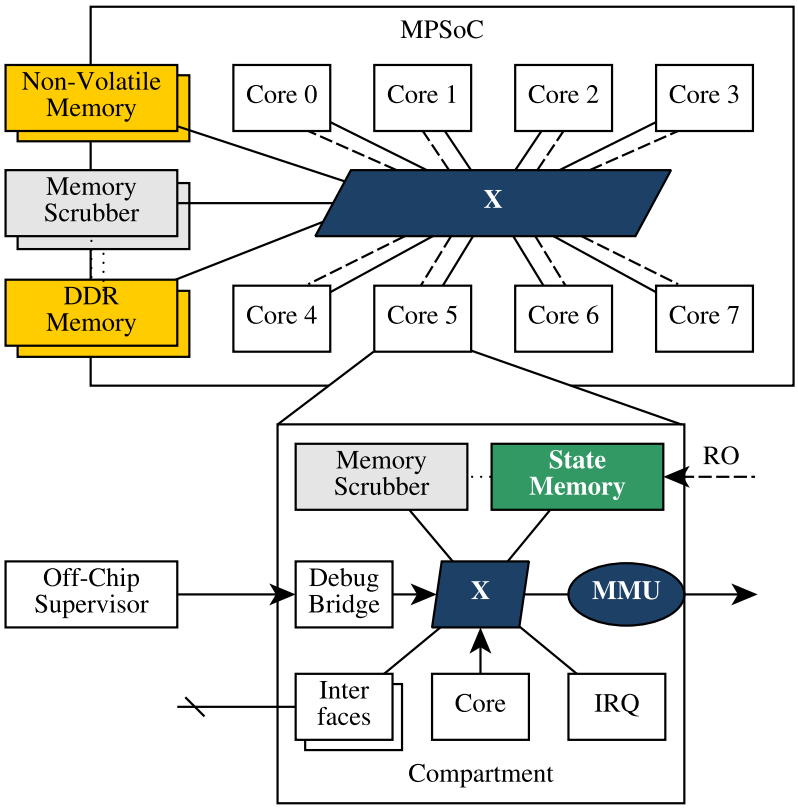


Figure 39: A high-level topology diagram of our compartmentalized MPSoC architecture with memory controllers highlighted in yellow, and interconnect-logic in blue. A debug-bridge on each compartment allows supervisor access. Access to each compartment's validation memory is possible read-only through the global interconnect.

writable through the compartment-local interconnect, and is read-only accessible by other compartments.

The address space layout on each compartment, including mapping of peripherals and interfaces within the address space are identical. Each compartment can access its own main memory address segment, which is mapped to the same address range on all compartments. Additionally, main memory in its entirety (all memory segments) is read-only accessible system wide, to simplify state synchronization between compartments.

During a checkpoint, the state of all threads mapped to a compartment is compared and synchronized with its siblings. To do so, the checkpoint handler executes an application-provided callback function for all pending threads, producing checksums generated from thread-private data structures. Checksums are stored in the compartment's local validation memory and thereby exposed to the other compartments, and then compared with the other compartments in the system. In case of disagreement, the compartment signals disagreement with that sibling and executes synchronization callbacks for all affected threads. If necessary, it then also executes relevant update callbacks and then resumes application execution. An more detailed description of these mechanisms as well as benchmark results for an astronomical application are described in Chapter 4.

6.5.1 Stage 1: Short-Term Fault Mitigation

The objective of Stage 1 is to detect and correct faults within a compartment, and assure a consistent system state through checkpoint-based FEC. It is implemented as sets of compartments running two or more copies of application threads (siblings) in lock step. Checkpoints interrupt execution, facilitating the lockstep and enforcing synchronization, allowing thread assignment within the system to be adjusted if required, as depicted in Figure 38.

This approach enables us to utilize application intrinsics to assess the health state of the system without requiring in-depth knowledge about the application code. The supervisor just reads out the results of the compartments' decentralized consistency decision. Threads can be scheduled and executed in an arbitrary order between two checkpoints, as long as their state is equivalent upon the next checkpoint.

We avoid thread synchronization issues due to invasive lockstep mechanisms [198] by merely reusing existing OS functionality without breaking existing ABI contracts. Therefore, we can continue relying upon pre-existing synchronization mechanics such as POSIX cancellation points[2] and their bare-metal equivalents (e.g., in RTEMS *RTEMS_NO_PREEMPT* or the POSIX API). Stage 1 can even deliver real-time guarantees, and the tightness of the RT guarantees depends upon the time required to execute application callbacks. In our RTEMS/POSIX-based implementation, we utilize priority-based, preemptive scheduling with timeslicing, allowing threads to delay checkpoints until they reach a viable state for checksum comparison.

Checkpoints are time triggered, but can also be induced by the supervisor through an interrupt, e.g., to signal that new threads have been assigned. Thus, the OS only has to support interrupts, timers, and a multi-threading capable scheduler. To the best of our knowledge, such functionality is available in all widely used RT- and general purpose OS implementations.

[2]E.g., sleep, yield, pause, for further details, see IEEE Std 1003.1-2017 p517

A fault resolved during a checkpoint may cause the affected compartment to emit incorrect data through I/O interfaces, an inherent limitation to coarse-grain lockstep [199]. For many very small nanosatellite missions this is acceptable, as the use of COTS components requires incorrect I/O to be sanitized anyway. In contrast, larger spacecraft already utilize interface replications or even voting, usually requiring considerable effort at the interface level to facilitate this replication. Our approach combined with the previously described MPSoC architecture inherently provides interface-level replications by design, no longer requiring extra measures to be taken. Additional protection is therefore only needed for space applications where non-propagation of incorrect I/O is required but interface replication is undesirable, i.e., due to PCB-space constraints aboard CubeSats or unchangeable subsystem requirements. For packet-based interfaces such as Spacewire, AFDX, CAN, or Ethernet, no hardware-side solution is necessary, as data duplication can be managed more efficiently at OSI layer 2+. This approach today is widely used as part of real-time capable FT-networking [94]. Other interfaces like I2C and SPI allow a simple majority decision per I/O line, which can be implemented on-chip through FIFO buffers, as the remaining on-compartment interfaces have low pin count and run at relatively low clock frequencies.

6.5.2 Stage 2: Tile Repair & Recovery

Stage 1 can not reclaim defective compartments, eventually resulting in resource exhaustion. Therefore, in Stage 2, we recover defective compartments through reconfiguration to counter transients in FPGA fabric. To do so, the supervisor will first attempt to recover a compartment using partial reconfiguration. Afterwards, the supervisor validates the relevant partitions to detect permanent damage to the FPGA (well described in, e.g., [218]), and executes self-test functionality on the compartment to detect faults in the compartment's main memory segment and peripherals. If unsuccessful, the supervisor can repeat this procedure with differently routed configuration variants, potentially avoiding or repurposing permanently defective logic.

As compartments are placed along partition borders in our MPSoC architecture, compartments can be recovered in the background without interrupting the rest of the system. The supervisor can also attempt full reconfiguration implying a full reboot of all compartments. Further details on reconfiguration and error scrubbing with a microcontroller-based proof-of-concept implementation for a nanosatellite are available in Chapter 5. If both partial- and full-reconfiguration are unsuccessful and all spare resources have been exhausted, Stage 3 is utilized to assure a stable system core to enable operator intervention.

6.5.3 Stage 3: Applied Mixed Criticality

Stage 3 autonomously maintains system stability of an aged or degraded OBC. When considering a miniaturized satellite's OBC, we can differentiate individual applications or parts of flight software by criticality. At the very least, we will find software essential to a satellite's operation, e.g., platform control and commandeering, as well as other applications of various levels of lower criticality. If the previous stages no longer have enough spare processing capacity or compartments to compensate the loss of a compartment, this stage utilizes thread-level mixed criticality to assure stability of core OBC functions. To do so, it can sacrifice lower criticality tasks in favor of providing compute resources to reach the desired replication level for critical threads.

Dependability for higher-criticality threads can efficiently be maintained by reducing compute performance or reliability of lower-criticality applications. Lower-criticality tasks may be executed less frequently or on fewer compartments, thereby reducing functionality or fault coverage for these tasks, retaining resources for higher-criticality threads. This decision is taken autonomously, and the operator can then define a more resource conserving satellite operation schedule at a spacecraft level, e.g., sacrifice link capacity, or on-board storage space, to make best use of the OBC in its degraded state.

6.6 Spare Resource Pooling

This FT approach enables FT even for very small satellites, but provides benefits for spacecraft of all weight classes. To increase fault coverage in traditional hardware voting FT systems, additional cores and spares must be provisioned, while compute performance can be increased by utilizing higher-performance processor cores and adding more hardware voting instances. This is done at design time, requiring over-provisioning, and can not be changed throughout a mission. Cores are hardwired to a specific instance, therefore, an instance will degrade once its spares are exhausted, even if idle spares were available elsewhere.

In contrast, our approach is not based on hardwired voting instances, as applications are mapped to a global pool of compartments with a given replication level. Our approach does utilize spare resources too, but spare compartments and conventional compartments are identical. Hence, spare compartments do not have to remain idle, and unused processor capacity becomes a spare resource that can be re-purposed. Thus, the fault coverage capabilities of the system are no longer dependent on the distribution and location of permanent faults within the system, increasing overall robustness.

As applications can be migrated between compartments, low criticality threads and background tasks can be assigned to utilize free spare capacity. These lower-criticality threads can be de-scheduled in favor of higher-criticality applications, if needed. Spare capacity can also be used to increase FT for threads, which usually would be executed without majority voting or separately due to resource constraints. We can distribute a defective compartment's workload to other compartments, to best take advantage of the remaining system resources.

The best target compartments and to-be-evicted threads are not determined ad-hoc, but before a fault actually occurs, to reduce the time spent in a checkpoint. We can maintain one replacement strategy for every compartment, due to the low compartment and thread counts common in space applications today[3]. Subsequent to a fault, these strategies are recomputed to consider the now reduced processing capacity of the system. As thread assignments are not controlled by the supervisor, but only adjusted, threads may exit, fork or create new child threads. Therefore, an update to adjust these strategies to the currently running threads is also triggered based on the fault counter of Stage 2. Even if a fault occurs immediately after the current

[3]The main application for our architecture is platform control. ManyCore-systems with hundreds of cores would allow too many combinations, but they will not be applied to satellite platform control in the foreseeable future. For dedicated payload data processing, this may be different, but our interest in this thesis is mainly platform control and unified satellite data handling aboard miniaturized satellites.

checkpoint, these strategies will only be needed at the next checkpoint. Therefore, this is a background operation which can be handled by the supervisor, allowing the OBC to resume processing immediately.

Figure 40 depicts a six compartment MPSoC running four applications of different criticality. A fault has occurred in compartment 3, which has been marked as permanently defective, and there are multiple recovery solutions:

- Affected threads could be relocated to a compartment running lower-criticality applications, replacing them as depicted in Figure 40a. For example, the threads previously run on compartment 3 can be migrated to compartment 6, replacing lower criticality thread-copies previously run there. This requires compartment 6 to copy the state of its newly assigned threads from compartment 1 or 2, at the cost of executing the lower-criticality applications redundantly instead of with majority voting.

- Instead of entirely de-scheduling one instance of each lower criticality threads, the clock frequency on two compartments could be increased, allowing one of each high-criticality thread to be migrated. In Figure 40b, this is depicted by moving the threads from the failed compartment to compartments 5 and 6 without de-scheduling instances of the low criticality threads. This is possible as coarse-grain lockstep only requires an equivalent state between siblings upon reaching a checkpoint and no cycle-accurate synchronization. Most modern embedded and mobile-market cores support frequency scaling.

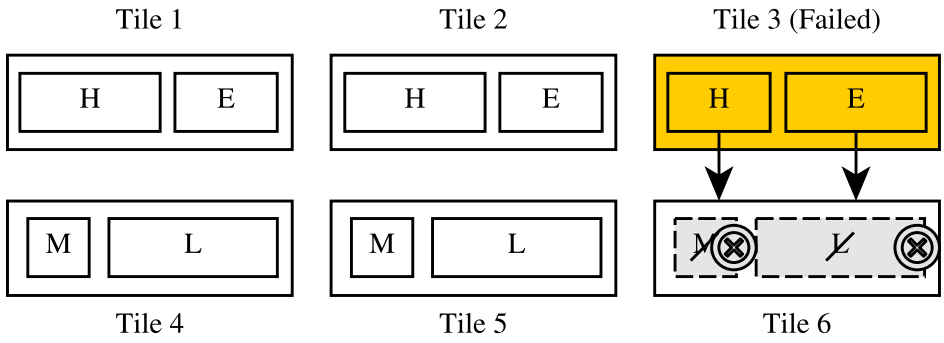
- Another possibility would be to instead increase the clock frequency of just one compartment, if sufficient additional processing capacity can be made available that way.

- Finally, in contrast to increasing the clock frequencies of individual compartments, compartment 4-6's schedulers could also assign less processing time to the lower-criticality tasks as shown in Figure 40c. Due to timing implications for real-time applications, this may only be possible for sporadic tasks, and background applications, which do not require a fixed amount of processing time. Also, to guarantee equivalent work is conducted for the medium and lower-criticality threads, the schedulers on 3 instead of just 2 compartments would require adjustment, wasting processing capacity in Tile 4 and 6. However, during this idle time, Tile 4 could be deactivated to reduce energy consumption.

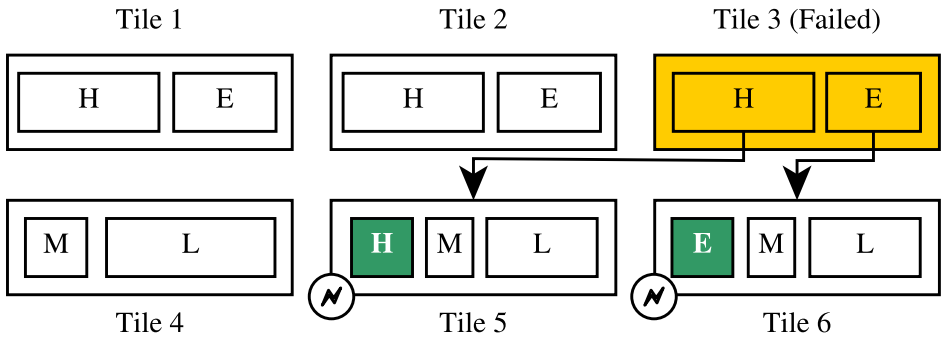
The ideal recovery strategy depends on the current performance requirements towards the OBC. Additional thoughts on this aspect are discussed, e.g., in [241], where different replacement strategies are described at a more mathematical level for video streaming applications. In the next section, we therefore discuss a heuristic approach to find near-best solutions to calculate this decision autonomously and rapidly, considering different performance requirements.

6.7 Adapting to Varying Mission Requirements

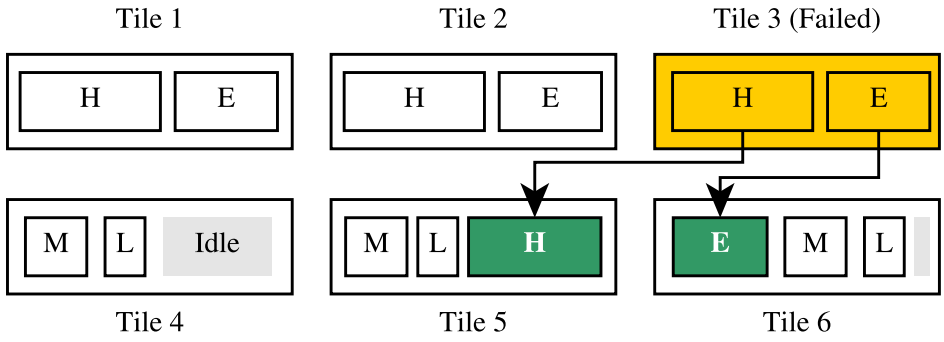
The approach described in the previous sections allows an OBC to meet a desired power budget, maximize fault coverage, processing power, or even functionality. Hence, the spacecraft can better fulfill its scientific or commercial mission, and increase the



(a) Migration by low-criticality thread pruning.



(b) Migration through clock-speed increase.



(c) Migration through processing time reduction.

Figure 40: A hexa-core MPSoC running 4 threads of mixed criticality (**E**ssential, **H**igh, **M**edium, and **L**ow), where compartment 3 (yellow) suffered a hard fault. To retain majority voting for the higher criticality threads, different recovery strategies can be facilitated through, without directly requiring spares.

spacecraft's lifetime. Theoretically, all we need to do is find the ideal set of thread mappings which fulfill our desired trade-off between processing capacity, FT, and minimal energy consumption. These three performance objectives can be visualized as depicted in Figure 41, and viable mappings can be found in the inner area outlined in red.

These three objectives oppose each other, and fully dynamic performance optimization at runtime is non-trivial and costly. Prior publications in computer science (e.g., [241, 246])approaches such issues with computationally expensive optimization algorithms to find the ideal solution, or design space exploration to find a large set of near-best and chose the optimal solution either at runtime [241] or design time [246]. The latter defeats the purpose of run-time flexibility and adjustment. While design space exploration at runtime is infeasible due to the limited processing capacity of a supervisor, unless tight constraints are placed upon applications regarding structure and functionality [241]. In practice, however, we do not have to find the singular "best possible" solution when recovering from a fault, instead we just need a "good enough" solutions yielded by a heuristic algorithm [247]. Once the system has been stabilized, ample time will be available to further optimize the thread mapping and usually this is done by the operator or flight software. The code of this algorithm is depicted in Algorithm Listing 1.

To facilitate a heuristic approach, we first reduce these three competing objectives to a set of *performance profiles*, examples of which are given in Table 42. In each

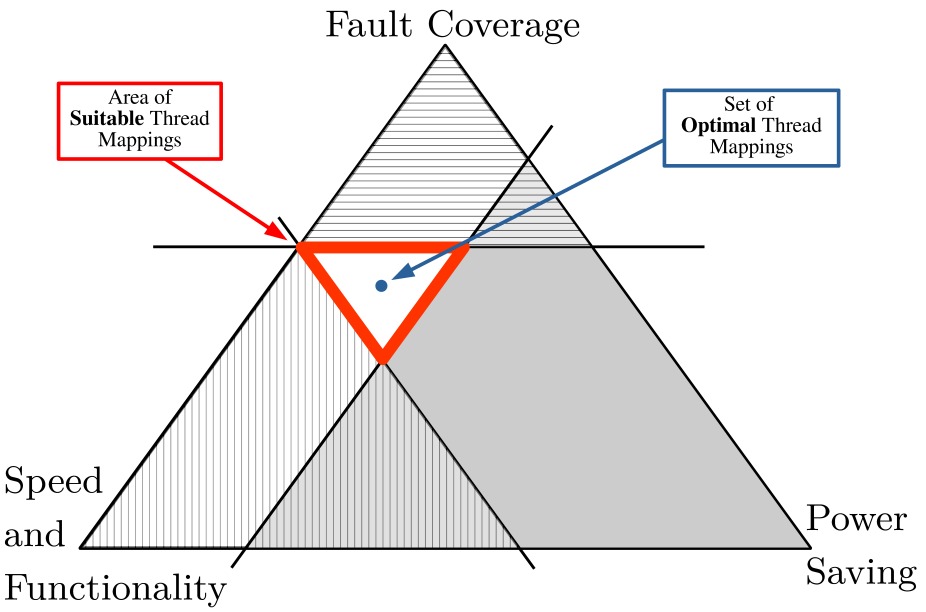


Figure 41: An MPSoC utilizing the presented approach can trade speed, energy efficiency, and fault coverage at run-time. We utilize *performance profiles* for each objective to facilitate a heuristic solution, which is located somewhere within the red highlighted area. This is an approximation of one or multiple "ideal/optimal" thread-mappings, which can be computed only with more processing time, through design-space exploration solution space (brute force).

profile, criticality classes (essential - low) are assigned one or multiple execution modes: separate execution with de-scheduling allowed, separate, redundant, majority voting, or with more cores, e.g., to enable Byzantine voting (referred to as NMR, TMR, DMR, separate, and de-schedule in Table 42). Duplicate assignments allow threads to be mapped in either mode, to enable mode reduction in case of resource constraints. For example, when running in the robustness profile, essential applications are always assigned the desired number of cores, while high-criticality applications are at least TMRed (depending on available resources). Other applications are preferably executed TMRed, but may be executed also DMR to retain fault detection, in case of resource exhaustion, instead of entirely de-scheduling lower criticality threads. Depending on mission requirements, the operator can then select the most suitable performance profile from a set of pre-generated at runtime, or could draft a new one.

To map threads, we build a new mapping for a task using the strongest desired execution mode. We evaluate if this exceeds the available power budget (energy profile) or processing capacity. If so, we begin reducing the execution mode of tasks beginning with the last mapped and therefore lowest-criticality thread. If successful, we append the mapped thread to a list and proceed with the next thread. To minimize the amount of de-scheduled and mode reduced threads, we can sort threads of same criticality based on required processing capacity. Thereby, computationally expensive threads are reduced in execution mode first, freeing up larger amounts of processing resources.

If not all threads could be mapped, we can de-schedule lower-threads exceeding the compute capacity, energy constraints, or allocate less processing time to specific applications system. Once no further mode or processing time reductions are possible due to real-time guarantees, we cease mapping new threads to uphold fault tolerance guarantees for this reduced core system. As final step, we traverse the list from the start and increasing execution mode to undoing mode reductions for as many threads as possible. The supervisor itself only has to execute the latter part of this algorithm and perform mode and processor time reduction, or de-schedule the lowest criticality threads. It does not have to actually generate all these mappings as it does not enforce

Mode	Performance	Power Saving	Robustness	Functionality
NMR	E - - - ↑	E - - - ↓	E H M L ↑	E - - - ↑
TMR	E H M L	E H M L	E H M L	E H M L
DMR	- H M L	- - M L	- H M L	E H M L
Separate	- - - L	- - - L	- - M L	E H M L
Deschedule	- - - -	- - - -	- - - -	- H M L

Figure 42: *Performance profiles* with threads of different criticality levels (**E**ssential, **H**igh, **M**edium, **L**ow) being assigned different replication levels to enable fault detection or different voting configuration through thread replication. Arrows indicate the strategy used for choosing mappings. E.g., In the Power Saving profile, all threads are first mapped in their highest desired replication level, and then reduced beginning with the lowest priority threads until the system's thread mapping allows a given energy consumption threshold to be surpassed. In the Performance or Robustness profiles, we instead attempt to achieve the highest level of thread-replication that is possible with the given available processor compartments. In the Functionality profile, we wish to retain a stable setup for essential application, even if this requires lower criticality threads to be de-scheduled.

ALGORITHM 1: Pseudo-Code of the Thread-Allocation Heuristics

Input: T_i: List of Threads, P: performance profile, C: Set healthy Cores
Output: M: List of mapped thread-groups

```
1  for T_i from T_0 to T_n do
       // Attempt to create a mapping for the thread
2      replication_level = getDesiredReplication(P, T_i)
3      thread_group = makeGroup(T_i, replication_level, C)
4      thread_mapping = getTargetCores(thread_group, C)
5      if isValid(thread_mapping) then
6          AppendGroup(M, thread_group, targets)
7      else
           // Failure, try to map with lower replication
8          lowest_replication = getLowestAllowedReplication(P, T_i)
9          while replication_level is not lowest_replication do
               // reduce replication level and retry
10             replication_level = getLowerReplication(P, T_i)
11             thread_group = makeGroup(T_i, replication_level, C)
12             thread_mapping = getTargetCores(thread_group, C)
13             if isValid(thread_mapping) then
14                 AppendGroup(M, thread_group, thread_mapping)
15                 goto line 1 // break out of nested loop and continue

           /* Insufficient compute capacity available in the system. E.g., too many
              compartments failed. Attept to reduce the replication level of an early
              mapped higher priority application to free compute capacity.          */
16         for M_i from M_i to M_0 do
17             t = getThread(M_i)
18             others_replication = getCurrentReplication(P, t)
19             lowest_replication = getLowestAllowedReplication(P, t)
20             while others_replication is not lowest_replication do
                   // Reduce replication for next higher priority group and retry
21                 tryReduceReplication(P, M, M_i, others_replication, C)
22                 thread_mapping = getTargetCores(thread_group, C)
23                 if isValid(thread_mapping) then
24                     AppendGroup(M, thread_group, targets)
25                     break
               // Can not reduce mapping, try to reduce earlier mapped thread
           // Too-few compute resources, de-schedule and try to map next thread
```

thread assignment in the system and only intervenes if necessary.

This algorithm also provides all mechanisms necessary to minimize the amount of active processor cores, and as threads can be concentrated to as few compartments as possible, maximizing the number of clock-gated cores. Individual tasks could also signal preference for reduced processing instead of a mode reduction as the approach itself is computationally inexpensive.

6.8 Discussions

We implemented the MPSoC architecture described in Section 6.5 using Xilinx Kintex and Virtex FPGAs as well as the Zynq SDSoC platform [40], as these are relevant for our target missions. However, for larger satellite platforms, this approach and architecture could very well be implemented on ASIC, and we see this as a “big-space" variant of our approach. An ASIC implementation would have lower energy consumption, and allow higher clock rates due to tighter timing and shorter paths, and be less susceptible to transient faults. If manufactured in an inherently radiation hardened technology such as FD-SoI [144], the system as a whole would be considerably more resistant to transient faults. Stage 2 would then be reduced to testing and validate compartments, while no longer being able to recover faulty compartments containing defective logic, but strong fault coverage of SEEs would be improved due to RHBM.

Overall, an FPGA implementation offers stronger FDIR capabilities, better coverage for permanent faults, and high flexibility at low cost, while the ASIC variant could offer better system performance and radiation tolerance due to RHBM. Custom ASIC development of course is expensive and time-consuming, thus, the resulting implementation would not be a viable solution for most miniaturized satellite applications, and therefore not in the scope of this technology development project.

The relaxed cost, energy, and size constraints aboard larger spacecraft allow an implementation of our approach spanning multiple FPGAs. Compared to a single-chip implementation, a multi-FPGA MPSoC variant offers better scalability due to easier routing, can tolerate chip-level defects, and SEFIs to the globally shared memory controllers, these can be distributed to different FPGAs. Replicated thread-instances could then also be distributed across FPGAs, offering non-stop operation while one of the FPGAs undergoes full reconfiguration. However, our proof-of-concept is focused on a single-FPGA based prototype for nanosatellite use.

Our project is focused on payload data handling and platform control for miniaturized spacecraft, and therefore accelerator cores supporting computational offloading are outside the scope of our research. Nonetheless, it is possible to also protect accelerator systems using this approach, yielding at least similar benefits. The structure and type of applications usually executed on accelerators is tightly constrained as compared to general purpose platform control, simplifying lockstep replication and thread-mapping. Especially synchronization for real-time applications and the impact of live-migration between compartments or state-updates on a faulty compartment, become much simpler if fully deterministic application behavior is assumed, as would be the case for computational offloading.

Our existing MPSoC design utilizes an AXI interconnect, but we plan to rework our MPSoC to instead use a NoC between compartments and shared memory controllers. The existing interconnect implementation allows low-latency communication, but has

a large footprint, and is difficult to route[4] for larger compartment counts (without optimization, we successfully placed 8 compartments). A NoC instead allows not only better scalability and easier routing, but also enables the implementation of a broad variety of FT concepts such as [93].

Tiles have direct read-only access to another compartment's memory segment to allow rapid thread migration and allow real-time capacity. However, direct access to shared main memory is not necessary to facilitate Stages 1-3. The data exchange required to facilitate thread migration could very well be implemented using IPC or through sockets, when considering complex networked architectures. In distributed systems, our approach could thus manage threads across multiple nodes sharing data when required, at the cost of higher latency.

We developed this approach to guarantee FT for opaque threaded applications on POSIX-compatible RTOS and general purpose operating systems such as RTEMS and Linux. However, the same functionality can also be applied to virtualized, voted systems and to runtime based platforms. It would be very well imaginable to implement Stage 1 within MicroPython or a hypervisor, and instead vote on Python scripts or virtual machines.

6.9 Conclusions

To the best of our knowledge, the on-board computer (OBC) design presented in this chapter is the first practical, non-proprietary, and affordable fault tolerance (FT) approach suitable even for very small spacecraft. It offers strong fault coverage, using just commercial-off-the-shelf hardware, library IP, and commodity processor cores, requiring only a single FPGA and a microcontroller based supervisor. The software-side FT approach outlined in Stage 1 is non-invasive to applications and the OS, therefore existing software can be reused and extended easily, while retaining real-time capabilities. The research presented in this chapter covers the entire FDIR loop, and does not ignore or make unrealistic assumptions regarding fault detection.

Our approach enables the re-use of existing development tools and IP designed for mass-produced mobile-market applications, taking an important step towards departing from the artisanal development approach in today's space computing. Instead of requiring new technologies to be re-invented constantly and maintained at high cost, the FT mechanisms presented in this chapter are flexible, which can adapt and grow with the development of computer and processor technology.

We do not just enable FT for a satellite class which so far has been considered unreliable, but also enhance the fault coverage capabilities of OBCs in larger spacecraft, and other applications with similar constraints and fault profile. Our approach facilitates majority voting through dynamic, replicated thread groups mapped to the available processor cores dynamically at runtime, instead of hardwiring them. Thus, all processing capacity, including spares, are part of a shared resource pool. Therefore, spare resources can be used more efficiently, and allowing idle compute capacity to be used productively until it is needed for fault coverage. An OBC running the presented hybrid hardware-software FT approach can adapt to varying mission requirements regarding adjusting the OBC transparently at run-time, trading processing capacity for reduced energy consumption or increased fault coverage.

[4]We can still achieve a functional implementation meeting timing constraints at several hundred megahertz, but the interconnect PBlock becomes disproportionately large.

Chapter 7

Reliable Data Storage for Miniaturized Satellites

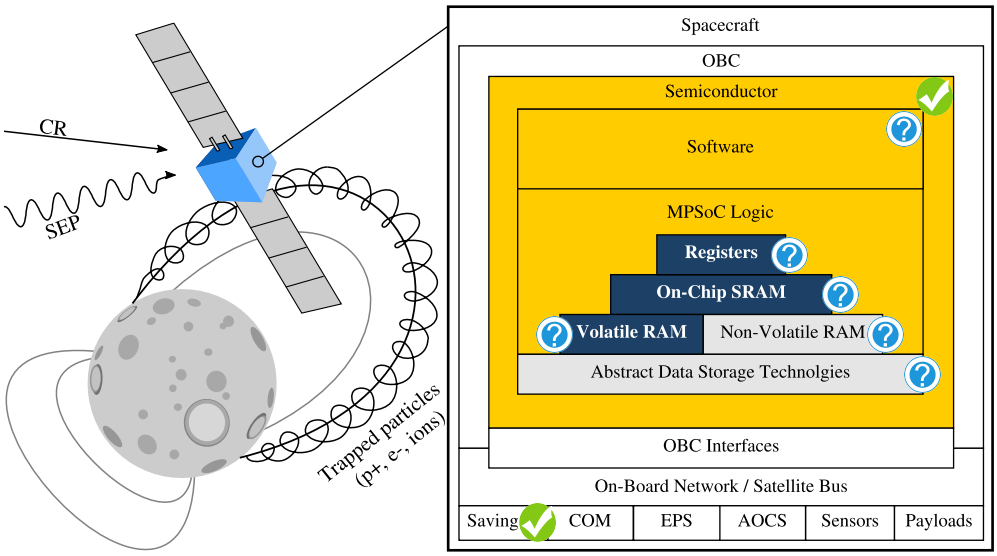
Memory Fault Tolerance

Reliable operation of an OBC can only be guaranteed if the integrity of the OBC's firmware, operating system, applications, as well as payload data can be safeguarded. Chapter 7 is therefore dedicated to discussing storage fault tolerance to answer RQ4. We discuss how the robustness of a nanosatellite's volatile memory components can be increased through software measures, as space-grade parts with strong erasure coding are not available to CubeSat designers. In the later parts of the chapter we cover integrity protection for data stored in commercial non-volatile memory ICs. The research presented in this chapter was published as finalist paper in the proceedings of the AIAA/USU Conference on Small Satellites (SmallSat) *[Fuchs15]. The sections related to MRAM and flash memory were published in the proceedings of the* International Conference on Architecture of Computing Systems (ARCS) *[Fuchs18] and the* International Space System Engineering Conference Data Systems In Aerospace (DASIA) *[Fuchs16].*

7.1 Introduction

Recent miniaturized satellite development shows a rapid increase in available compute performance and storage capacity, but also in system complexity. CubeSats have proven to be both versatile and efficient for various use-cases, thus have also become platforms for an increasing variety of scientific payloads and even commercial applications. Such satellites also require an increased level of reliability in all subsystems compared to educational satellites, due to prolonged mission duration and computing burden. Nanosatellite computing will therefore evolve away from federated clusters of microcontrollers towards more powerful, general purpose computers; a development that could also be observed with larger spacecraft in the past. Certainly, an increased computing burden also requires more sophisticated operating system (OS) or software, making software-reuse a crucial aspect in future nanosatellite design. In commercial and agency spaceflight, a concentration on few major OSs (e.g., RTEMS [248]) and processors (e.g., LEON3 and RAD750) has therefore occurred. A similar evolution, albeit much faster, can also be observed for miniaturized satellites.

To satisfy scientific and commercial objectives, miniaturized satellites will also require increased data storage capacity for scientific data. Thus, many such satellites have begun fielding a small but integrity-critical core system storage for software, and a dedicated mass-memory for pre-processing and caching payload-generated data. Unfortunately, traditional hardware-centered approaches to fault tolerance, also increase costs, weight, complexity and energy consumption while decreasing overall performance. Therefore, such solutions (shielding, simple- and triple-modular-redundancy – TMR) are often infeasible for miniaturized satellite design and unsuitable for nanosatellites. Also, hardware-based error detection and correction (EDAC) becomes increasingly less effective if applied to modern high-density electronics due to diminishing returns with fine structural widths. As a result of these concepts' limited applicability, nanosatellite design is challenged by ever increasing long-term fault coverage requirements.



7.1.1 Context and Application

Neither component level, nor hardware or software measures alone can guarantee sufficient system consistency. However, hybrid solutions can increase reliability drastically introducing negligible or no additional complexity. Software driven fault detection, isolation and recovery from (hardware) errors (FDIR) is a proven approach also within space-borne computing, though it is seldom implemented on nanosatellites. A broad variety of measures capable of enhancing or enabling FDIR for on-board electronics exists, especially for data storage. Combined hard- and software measures can strongly increase reliability.

This research was conducted as part of the MOVE-II CubeSat project based upon an ARM-Cortex processor as a platform for scientific payloads. To fulfill this role, the traditional CubeSat approach to reliability, risk acceptance, does not suffice. Hence, we designed MOVE-II's on-board computer (OBC) to guarantee data integrity using software side measures and affordable standard hardware where necessary. The capability to assure data integrity for program code and data is essential to then achieve fault-tolerance for data processing elements and at the system level.

After a detailed evaluation of potential OSs for use aboard MOVE-II, we chose the Linux kernel due to its adaptability, extensive soft-/hardware support and vast community. We decided against utilizing RTEMS mainly due to our limited software development manpower, the intended application aboard our nanosatellite MOVE-II, and the abundant compute power of recent OBCs.

7.1.2 Chapter Organization

Often, fault tolerance aboard spacecraft is only assured for processing components, while the integrity of program code is neglected. In the next section, we thus outline the importance of memory integrity as a foundation for fault-tolerant satellite computing and provide a view on the topic at a high level. To protect data stored in volatile memory, we present a minimalist yet efficient approach to combine error scrubbing, blacklisting, and error correction encoded (ECC) memory in Section 7.3. MOVE-II will utilize magnetoresistive random access memory (MRAM) [147] as firmware storage, hence, we developed a POSIX-compatible filesystem offering memory protection, checksumming and forward error correction. This filesystem is being presented in Section 7.4, can efficiently protect an OS- or firmware image and supports hardware acceleration. Finally, a high performance dependable storage concept combining block-level redundancy and composite erasure coding for highly scaled flash memory was implemented to assure payload data integrity, the resulting concept is outlined in Section 7.5. The final section of this chapter is used to discuss and wrap up the results obtained herein.

7.2 Data Integrity as Foundation of Fault Tolerance

The increasing professionalization, prolonged mission duration, and a broader spectrum of scientific and commercial applications have resulted in many different proprietary on-board computer concepts for miniaturized satellites. Therefore, miniaturized satellite development has not only seen a rapid increase in available compute power and storage capacity, but also in system complexity. However, while system sophis-

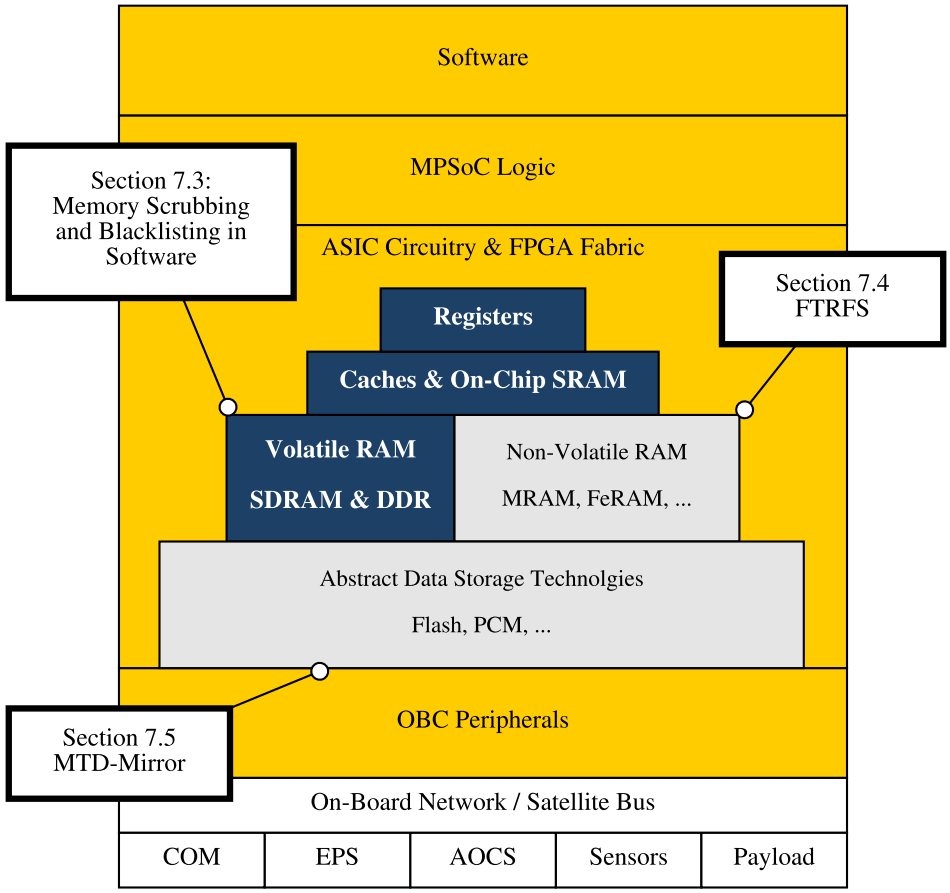


Figure 43: Data transiting or stored within components or system components shown in yellow and white may be corrupted due to radiation effects. Components depicted in blue can be safeguarded against data corruption using the concepts presented in the different sections of this chapter as indicated.

tication has continuously increased, re-usability, reliability remained quite low [249]. Recent studies of all previously launched CubeSats show an overall launch success rate of only 40% [41]. Such low reliability rates are unacceptable for missions with more refined or long-term objectives, especially with commercial interests involved.

As nanosatellites consist mainly of electronics, connected to and controlled by the OBC, achieving fault tolerance must begin with this component. Hence, an OBC's software and hardware must be designed to handle faults throughout a space mission, not if, but when they occur. fault tolerance can only be assured if program code and required supplementary data can be stored consistently and reliably aboard a spacecraft. Thus, data storage integrity must be assured first and foremost, without resorting to expensive, proprietary space-grade components that realize fault tolerance in hardware.

To enable meaningful fault tolerance, data consistency must be assured both within volatile and non-volatile memory, see Figure 43. Data is usually classified as either

system data or payload data stored in volatile or non-volatile memory. The storage capacity required for system data may vary from few kilobytes (firmware images stored within a microcontroller) to several megabytes (an OS kernel, its and accompanying software). Very large OS installations and applications are uncommon aboard spacecraft and thus not considered in this chapter. Payload data storage on the other hand requires much larger memory capacities ranging from several hundred megabytes to many terabytes depending on the spacecraft's mission, downlink bandwidth or link budget, and mission duration. In addition, data and code will temporarily reside in volatile system memory and of course the relevant memories within controllers and processors (i.e. caches and registers) which again must satisfy entirely different requirements to performance and size.

Due to these varying requirements, different memory technologies have become popular for system data storage, payload data storage and volatile memory. In the following sections, we will discuss and develop protective concepts to ensure memory integrity aboard spacecraft with a special focus on our nanosatellite use-case. All these concepts can be implemented at least as efficiently to larger satellites, as size and energy restrictions are much less pressing aboard these vessels.

7.3 Volatile Memory Consistency

Inevitably, data stored will at least temporarily reside within an OBC's volatile memory and all current widely used memory technologies (e.g., SRAM, SDRAM) are prone to radiation effects [250]. As a straightforwards solution, some OBCs were built to utilize only (non-volatile) MRAM as system memory which is inherently immune to SEUs and therefore allows OBC engineers to bypass additional integrity assurance guarantees for RAM. However, MRAM currently can not be scaled to capacities large enough to accommodate more complex OSs. Thus, while miniaturized satellites often utilize custom firmware optimized for very low RAM usage, larger spacecraft as well as most current and future nanosatellites do utilize DDR or SDRAM. For simplicity, we will refer to these technologies as RAM in this chapter. However, it is not to be confused with the use of the term RAM in Sections 7.4 and 7.5 of this chapter, as in MRAM.

Radiation induced errors alongside device failover is often assured using error correcting codes (ECC), which have been in use in space engineering for decades. However, a miniaturized satellite's OS must take an active role in volatile memory integrity assurance by reacting to ECC errors and testing the relevant memory areas for permanent faults. To avoid accumulating errors over time in less frequently accessed memory, an OS must periodically perform scrubbing. In case of permanent errors, software should cease utilizing such memory segments for future computation and blacklist them to reduce the strain on the used erasure code. Assuming these FDIR measures are implemented, a consistency regime based on memory validation, error scrubbing and blacklisting can be established.

7.3.1 DRAM Corruption and Countermeasures

The fault profile for DRAM aboard CubeSats mainly includes two types of gradually accumulating errors: soft-errors (bit-rot) and permanent (hard) errors. Depending on the amount of data residing in RAM, even few hard errors can cripple an on-board

computer: the likelihood for the corruption of critical instructions increases drastically over time. Therefore, to compensate for both hard and soft errors, ECC should be introduced [158].

Modern DRAM chips benefit strongly from feature size reduction and run at very high clock frequency, as a vast majority of a memory IC consists of memory cells. Soft errors there occur on the Earth as well as in orbit, due to electrical effects and highly charged particles originating from beyond our solar system. In case of such an error, data is corrupted temporarily but, and once the relevant memory has been re-written, consistency can be re-established. The likelihood of these events on the ground is usually negligible as the Earth's magnetic field and the atmosphere provide significant protection from these events, thus weak or no erasure coding at all is applied.

Hard errors generally occur due to manufacturing flaws, ESD, thermal- and aging effects. Thus, they may also occur or surface during an ongoing mission, further information on the causes for hard-faults in RAM is described in detail in [251].

By utilizing ECC, integrity of the memory can be assured starting at boot-up, though in contrast to other approaches ECC can not efficiently be applied in software [252]. Due to the high performance requirements towards RAM, weak but fast erasure codes such as single error correction Hamming codes with a word length of 8 bits are used [253, 254]. ECC modules for space-use usually offer two or more bit-errors-per-word correction. These codes require additional storage space, thereby reducing available net memory, and increase access latency due to the higher computational burden. Single-bit error correcting EDAC ASICs are available off-the-shelf at minimal cost, whereas multi-bit error correcting ones are somewhat less common and expensive. While such economical aspects are usually less pressing for miniaturized satellites beyond the 10kg range, nanosatellite budgets usually are much more constrained prompting for alternative, lightweight low-budget-compatible solutions. In the remainder of this section, we thus present a software driven approach to achieve a high level of RAM fault-coverage. We do so using commercial ECC paired with software measures, without expensive and comparably slow space-grade multi-bit-error correcting logic.

Ultimately, strong ECC is not a satisfying final solution to RAM consistency requirements due to inherent weaknesses of this approach to controller-faults, chip-level failure, and data-economical reasons in prolonged operation. Highly charged particles impacting the silicon of RAM chips can also permanently damage the circuitry of controller logic. In consequence, radiation can induce faults in control logic and other infrastructure elements of a memory IC, which there can causing SEFIs [255]. In contrast to hard and soft error in memory logic, SEFIs and permanent faults in controller logic can not be mitigated effectively through ECC. Instead, these should be mitigated at the system level, if this is possible. In Chapter 9, we show how this can be facilitated with commercial components. Otherwise, if no system-level mitigation is possible, the OBC remains prone to chip-level faults.

7.3.2 A Software-Driven Memory Consistency Concept

When utilizing ECC, memory consistency is only assured at access time, unless specialized self-checking RAM concepts are applied in hardware [256, 257]. Rarely used data and code residing within memory will over time accumulate errors without the OS being aware of this fact, unless scrubbing is performed regularly to detect and cor-

rect bit-errors before they can accumulate. The scrubbing frequency must be chosen based on the amount of memory attached to the OBC, the expected system load and the duration required for one full scrubbing-run [258]. Resource conserving scrubbing intervals for common memory sizes aboard nanosatellites range from several minutes up to an hour. Also, if a spacecraft were to pass through a region of space with elevated radiation levels (e.g., the SAA), scrubbing should be performed directly before and after passing through such regions.

As depicted in Figure 44, the DRAM integrity assurance measures usually realized in hardware in traditional space-grade components can be also be facilitated in software. We can construct a DRAM-integrity assurance regime using allocation-time memory testing, software-realized error scrubbing, and OS-side blacklisting of memory pages with defective blocks. All these elements can be realized in software using standard functionality, while a scrubbing tasks can be implemented within the OS's kernel, or even in userland. The specific implementation details therefore vary depending on what level this functionality is realized in.

Concept Overview

At a high level, this concept can be described as follows:

Bootup: During operating system bootup, the second stage bootloader or OS Kernel itself will execute platform bring-up code, may relocate the Kernel or RTOS code from storage into faster main memory. Subsequently, it will then prepare key OS data structure, and initialize core system functionality such as virtual memory, memory protection, a kernel console and logging, if available. All of these operations occur linearly, and require very little memory to be allocated. More memory intensive operations will occur past this point.

Blacklist: We add functionality to read a matrix of bad memory pages, where pages containing defective hard errors are marked. We can elegantly blacklist these memory pages by simply reserving them, thereby preventing them from being issued at a later stage. This is being done for performance and simplicity reasons, to avoid triggering ECC syndromes for known bad memory pages during operation, and performance costs. As the integrity of this bit-matrix is critical, it should reside in radiation-immune memory that does not suffer wear. Both FRAM and MRAM are viable technologies, due to small size of this memory, and simply redundancy for this memory can be realized as described in Chapter 9.

Operation: Once the bootup is completed, the Kernel will setup a suitable scrubbing task to periodically perform error scrubbing on main-memory associated memory regions. the OS will initialize flight software applications, and allocate memory for them.

Allocation: During operation of the flight software, whenever an application allocated memory, the OS will test the integrity of a memory page before issuing it to the consuming application. Should a memory page be discovered to be permanently defective, it will be left allocated but not issued. As we assume the availability of virtual memory, fragmentation of the memory map is a non issue. Memory allocation in most operating systems is an atomic operation, with interrupts being disabled during the operation. Hence, for the duration of memory allocation, no ECC syndromes will be

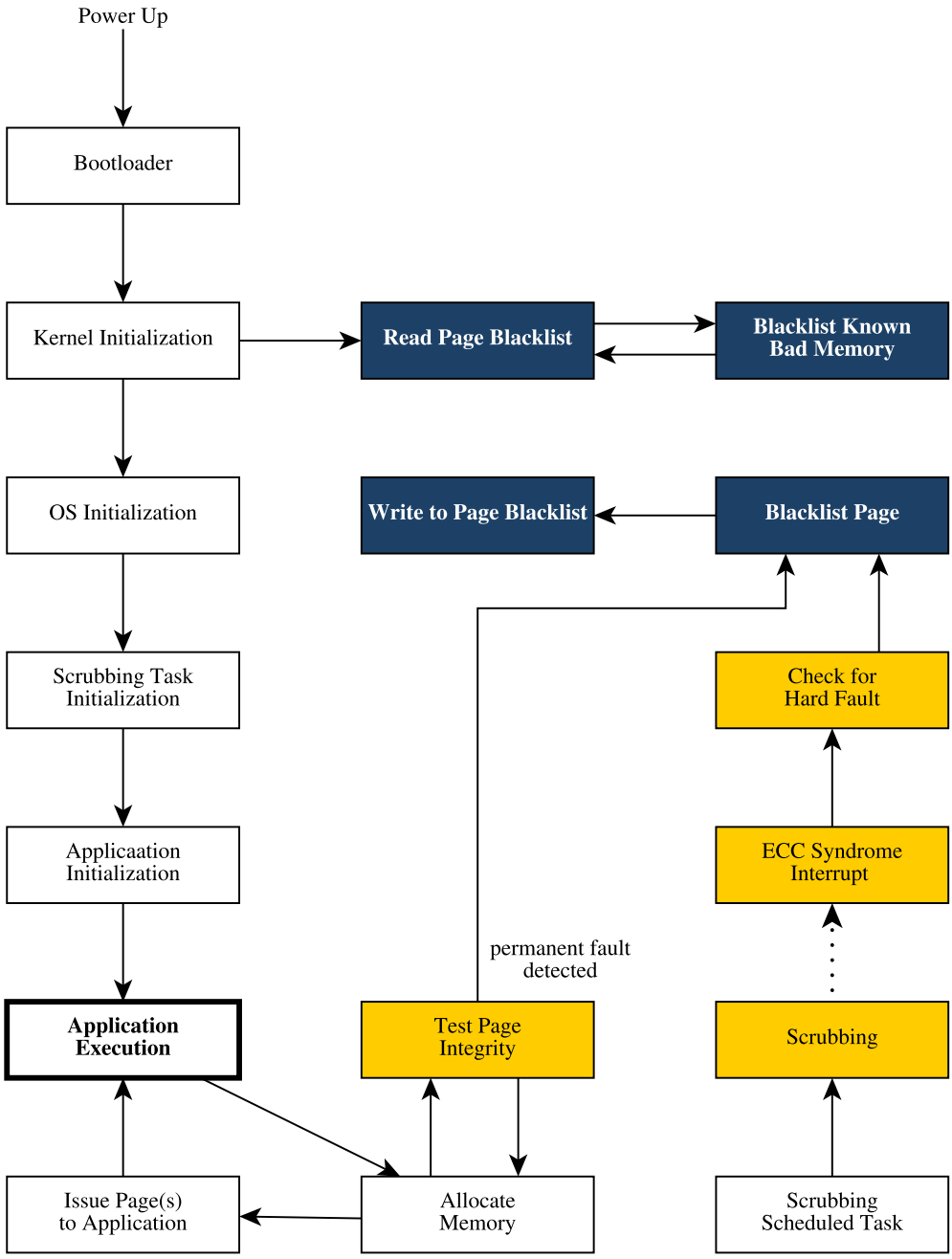


Figure 44: Integrity of volatile memory can be guaranteed if memory checking and ECC (yellow), as well as memory blacklisting (blue) are combined. Scrubbing must be performed periodically to avoid accumulating errors in rarely used code or data.

processed. At the end of allocation, in case an ECC syndrome interrupt is pending, syndromes for bad memory pages will be discarded.

Scrubbing: Periodically, the scrubbing task set up during OS initialization will read the entire DRAM address space, if hardware scrubbing is unavailable. This causes rarely accessed memory regions to be refreshed, preventing bit-upsets to accumulate there. The scrubbing application itself will not attempt to test if a page contains permanent faults, it just triggers ECC syndromes. It can be implemented in a variety of different ways, as described in Section 7.3.2.

Syndromes: We extend the functionality of the ECC syndrome handler, to not only determine if the ECC error was recoverable or not, and to respond to it in a suitable manner. Instead, we add functionality to test the relevant piece of memory to detect if the ECC error was caused by a soft or hard fault. In case of a hard fault, the relevant bit of the bad-memory matrix is flipped, and the page should no longer be used, as far as this is possible for already issued in-use memory. If desired, the syndrome handler can therefore consider ECC parameters in case multi-bit correcting ECC or Reed-Solomon block coding are used. Then, a minimum delta between hard errors in memory word and error correction capacity can be defined. This can help slow down the pace at which pages are discarded that contain faulty memory words.

Software-Implemented Scrubbing

In the case of a Linux Kernel and a GNU userland, a scrubbing task can most conveniently be implemented as a `cron`-job reading the OBC's physical memory. For this purpose, the device node */dev/mem* is offered by the Linux Kernel as a character device. */dev/mem* allows access to physical memory where scrubbing must begin at the device specific *SDRAM base address* to which the RAM is mapped. Technically, even common Unix programs like `dd(1)` could perform this task without requiring custom written application software.

Another possibility would be to implement a Linux kernel module using timers to perform the same task directly within kernel space. In this case, the scrubbing-module could also directly react to detected faults by manipulating page table mappings or initiating further checks to assure consistency. Execution within kernel mode would also increase scrubbing speed, allowing more precise and reliable timing.

Memory Checking and Blacklisting

Unless very strong multi-bit-error correcting ECC (> 2 bit error correction) and scrubbing are utilized, ECC can not sufficiently protect a spacecraft's RAM due to in-word-collisions of soft- and hard errors as depicted in Figure 45. To avoid such collisions, memory words containing hard faults should no longer be utilized, as any further bit-flip would make the word non-recoverable [228]. Even when using multi-bit ECC, memory should be blacklisted in case of grouped permanent defects which may be induced due to radiation effects or manufacturing flaws as well.

Memory must also be validated upon allocation before being issued to a process. Validation can be implemented either in hardware or software, with the hardware variant offering superior testing performance over the software approach. However, memory testing in hardware requires complex logic and circuitry, whereas the software

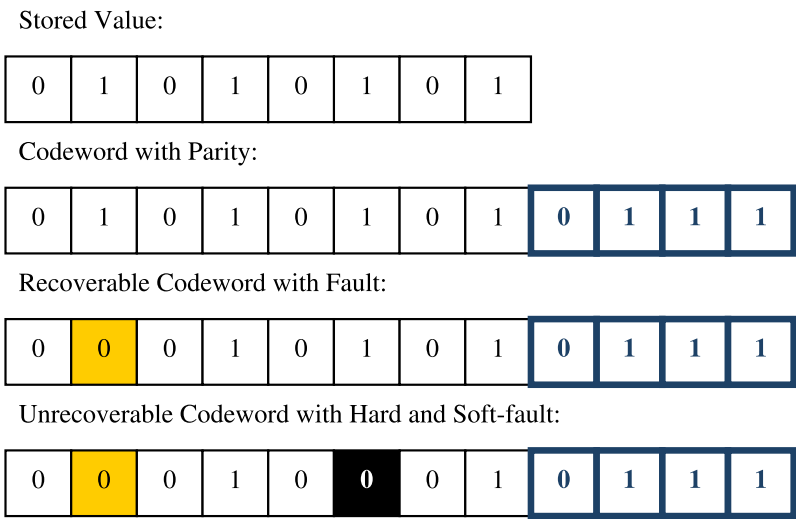


Figure 45: With single-bit correcting ECC-RAM, a word should no longer be used once a single hard-fault has been detected. Hard faults are depicted in black, soft faults in yellow, erasure code parity in green.

variant can be kept extremely simple. The Linux kernel offers the possibility to perform these steps within the memory management subsystem for newly allocated pages for ia32 processors already, and are currently porting this functionality to the ARMv7 MMU-code. In case the Linux kernel detects a fault in memory, the affected memory page is reserved, thereby blacklisted from future use, and another validated and healthy page is issued to the process. Therefore, we chose to rely upon this proven and much simpler software-side approach.

The ia32 implementation does not retain this list of blacklisted memory regions beyond a restart of the OS, though doing so is an important feature for use aboard a satellite. As memory checking takes place at a very low kernel-level (MMU code essentially works on registers directly and in part must be written in assembly), textual logging is impossible and persistent storage would have to be realized in hardware. An external logging facility implemented at this level would entail rather complex and thus slow and error prone logic, thus, a logging based implementation is infeasible. However, at this stage we can still utilize other functionality of the memory management subsystem to access directly mapped non-volatile RAM, in which we can retain this information beyond a reboot. Due to the small size required to store a page bitmap, it can be stored within a small dedicated FRAM/MRAM module, read by the bootloader and passed on to the kernel upon startup. This implementation can thus enable multi-bit-error correcting equivalent protection without requiring costly specialized hardware, while increasing system performance on strongly degraded systems.

7.4 A Radiation-Robust Filesystem for Space Use

The increased compute burden handled aboard modern nanosatellites also requires more sophisticated operating system (OS) software, which in turn results in increased code complexity and size [259].

For very simple computers, custom tailored OSs offer an excellent balance of size and functionality. However, development of proprietary OSs for unique custom computers has been abandoned in most of the IT industry, in favor of standard soft- and hardware reuse. This is still an ongoing process in spaceflight, though already producing a focus on a few types of radiation hardened processor platforms (e.g., LEON3, PPC750, RAD6000, see [260]) running common OSs [261, 262]. The same evolution has begun in nanosatellite computing, albeit much faster.

OSs popular in spaceflight such as RTEMS can consume less than 256KB of non-volatile (nv) memory [263], whereas Linux requires at least 2MB. If such a larger OS is used aboard a satellite, more sophisticated storage concepts are needed. Data must be stored permanently and consistently throughout the mission lifetime. Space missions often last between 5 and 10 years [264], but can reach 25 years or longer as discussed in Chapter 3. Thus a satellite's command and data handling (CDH), the on-board computer, must guarantee integrity and recover degraded or damaged data (error detection and correction – EDAC) over a prolonged period of time in a hostile environment. We consider a filesystem the most resource conserving and efficient approach, which also allows dynamically adjustable protection for the individual data structures. As Magnetoresistive Random-Access Memory (MRAM) [147] is widely used for radiation resistant data storage in nanosatellites, and therefore we developed FTRFS specifically for this technology.

7.4.1 Related Work and Preexisting File Systems

Filesystems often include performance optimizations such as disk head tracking, utilization of data locality and caching. However, most of these enhancements do not apply to storage technologies used in spaceflight. In fact, such optimizations add significant code overhead, possibly resulting in a more error prone filesystem and may even reduce performance.

Next-generation Filesystems, e.g., BTRFS, F2FS, and ZFS, are designed to handle many-terabyte sized devices and RAID-pools. Silent data corruption has become a practical issue with such large volumes [265]. Thus, these filesystems can maintain checksums for data blocks and metadata. Due to their intended use in large disk pools, they do also offer integrated multi-device functionality.

Multi-device functionality would certainly be advantageous, but neither ZFS nor BTRFS scale to small storage volumes. Minimum volume sizes are far beyond what current nanosatellite CDHs can offer. Technology scaling for the technologies strongly drives development of these file systems continuously towards larger volumes Hence, future development of these filesystems will require design decisions the conflict with the needs for spaceflight applications.

Filesystems for flash devices, similar to the memory technology itself, have evolved considerably over the past decade [266, 267]. Upcoming filesystems already handle challenges concerning potentially negative compression rates [268] or erase block abstraction, offer proper wear leveling and interact with device EDAC functionality (checksumming, spare handling and recovery). UFFS even offers integrity

protection for data and metadata using erasure codes.

Most new flash-filesystems interact directly with memory[1], thereby are incompatible with other memory technologies unless flash properties are emulated. This introduces further IO and may result in unnecessary data loss, as flash memory is of course block oriented.

RAM filesystems are usually optimized for throughput or simplicity, often resulting in a relatively slim codebase. If designed for volatile RAM, these filesystem are optimized for simplicity and do not necessarily require a nondestructive unmount procedure. Non-volatile RAM filesystems access data in memory directly avoiding many of the indirection and abstraction layers required for more abstract memory technologies [269], while some even utilize in-line compression to increase storage capacity [269].

Except for *PRAMFS* [270], none of these filesystems consider memory protection to increase dependability. *PRAMFS* offers execute-in-place (XIP) support [271] and is POSIX-compatible, but offers no data integrity protection.

In contrast to flash memories RAM filesystems are not block based, but benefit from the ability to access data arbitrarily. Thereby, no intermediate block management is required and read-erase-update cycles are unnecessary. While simple block-layer EDAC would certainly be possible, structures within a RAM filesystem can be protected individually allowing for stronger protection.

Open source space engineering and CDH research is directed mainly towards testing radiation related properties of memory technologies [272, 273] and on NAND-flash in particular [274, 275]. At the time of this writing, we are unaware of advanced software-side non-flash driven storage concepts for space use.

7.4.2 FTRFS

We designed FTRFS as Fault-Tolerant Radiation-robust Filesystem for Space use. It is intended to operate efficiently with small volumes($\leq$4MB) and assure data integrity for critical firmware-related data stored within COTS MRAM components. To fulfill its purpose for storing a firmware image, it was designed to be bootable, and also to allow for the capacity to scale much to larger volumes than can be achieved with toggle-MRAM at the time of writing.

As base for this filesystem's fault model, we assume that computational correctness within the OBC itself can be assured. Furthermore, we assume that within the OBC, ECC is applied to CPU-caches and RAM so that upsets in in-transit data can be detected and mitigated before they are written to memory. A CPU running FTRFS must be equipped with a memory management unit with its page-table residing in ECC protected volatile memory. All other elements (e.g., periphery and ALUs), other memories (e.g., registers and buffers) and in-transit data are considered potential error sources.

Memory protection has been largely ignored in RAM-filesystem design. In part, this can be attributed to a misconception of memory protection as a pure security-measure against malware. However, for directly mapped nv-memory, memory protection introduces the memory management unit as a safeguard against data corruption due to upsets in the system [276]. Thus, only in-use memory pages will be writable

[1] in the case of Linux through the memory technology device subsystem (MTD)

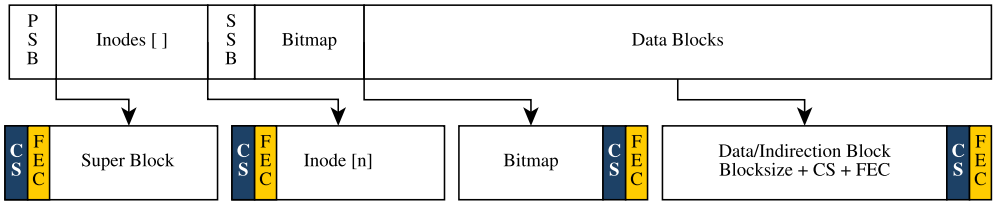


Figure 46: The basic layout of the presented filesystem. EDAC data is appended or prepended to each filesystem structure. PSB and SSB refer to the primary and secondary super blocks.

even from kernel space, whereas the vast majority of memory is kept read-only, protected from misdirected write access i.e. due to SEUs in a register used for addressing during a store operation.

FS-level data compression has been popular in size constrained filesystems. However, in our use case, well-compressible data, e.g., textual or binary log data, would reside in flash or PCM. Hence for a satellite's flight software firmware image will yield little gain, a and we therefore do not realize data compression as part of FTRFS, thereby allowing reduced code complexity and increasing performance.

After a detailed OS evaluation which was presented in [Fuchs12], we chose the Linux kernel as the base for our filesystem due to its adaptability, extensive soft/hardware support and vast community. We decided against utilizing RTEMS mainly due to our limited software development manpower. Further details on this evaluation including scoring data and a detailed description of the used criteria is available in [Fuchs12].

A loss of components has to be compensated at the software- or hardware level through voting or simple redundancy. Multi-device capability was considered for this filesystem, however it should rather be implemented below the filesystem level (e.g., via majority voting in hardware [277]) or as an overlay, e.g., RAIF [96].

The capability to detect and correct metadata and data errors was considered crucial during development. Based on the mission duration, destination or the orbit a spacecraft operates in, different levels of protection will be necessary. The protective guarantees offered can be adjusted at format time or later through the use of additional tools.

Due to the relatively restricted system resources aboard a nanosatellite, cryptographic checksums do not offer a significant benefit. Instead, CRC32 is utilized for performance reasons in tandem with Reed-Solomon encoding (RS) [229].

Metadata Integrity Protection

For proper protection at the filesystem level, in addition to the stored filesystem objects (inodes) and their data, all other metadata must be protected. Figure 46 depicts the basic layout. Although similar to *ext2* and *PRAMFS* [270], data addressing and bad block handling work fundamentally different. We adapt memory protection from the *wprotect* component of *PRAMFS*, as well as parts of the inode layout. *PRAMFS* is licensed under GPLv2 and based upon *ext2*.

The Super Block (SB) is kept redundantly, as depicted in Figure 46. An update to the SB always implies a refresh of the secondary SB, hence, hereafter no explicit reference of the secondary SB will be made. The SB also contains EDAC parameters

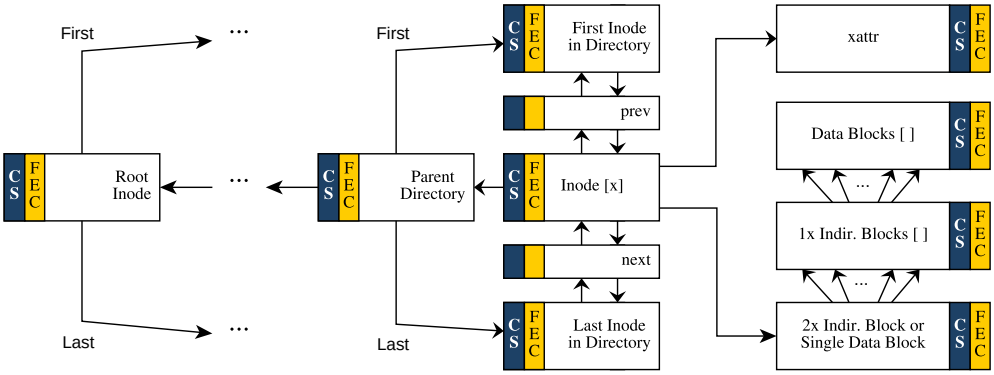


Figure 47: Each inode can either utilize direct addressing or double indirection. Extended attributes are always addressed directly.

for blocks, inodes and the bitmap.

The SB is the most critical structure within our filesystem, and is static after volume creation. Its content is copied to system memory at mount time, thus it is sufficient to assure SB consistency the first time it is accessed.

As the SB contains critical filesystem information, we avoid accumulating errors over time through scrubbing. Thereby, the CRC checksum is re-evaluated each time certain filesystem API functions (e.g., directory traversal) are performed.

A block-usage bitmap is dynamically allocated based on the overhead subtracted data-block count and is appended to the secondary SB. The bitmap EDAC is also dynamically sized and must be stored beyond the compile-time static SB, even though placing it there would be convenient. Thus, the protection data is located in the first block after the end of the bitmap, see Figure 46. In case the bitmap is extended, the new part of the bitmap is initialized and then the error correction data is recomputed at its new location. We refrain from re-computing and re-checking the EDAC data upon each access, instead FEC data is checked before and updated after each relevant operation has been concluded.

Inodes are kept as an array. Their consistency is of paramount importance as they define the logical structure of the filesystem. The array's length is determined upon filesystem initialization and can change only if the volume is resized. As each inode is an independent entity, an inode-table wide EDAC is unnecessary. Instead, we extend and protect each inode individually.

Data Consistency and Organization

To optimize the filesystem towards both larger (e.g., a kernel image, a database) and very small (e.g., scripts) files, direct and double indirect data addressing are supported, as depicted in Figure 47. The filesystem selects automatically which method is used. Data protection requirements vary depending on block size, and use case. Thus FTRFS allows the user to adjust the protection strength for data blocks, as will be described in the next section.

Data block size cannot be arbitrarily decreased, as some Linux kernel subsystems assume them to be sized to a power of two. Instead, the filesystem internally utilizes larger blocks to include EDAC data, see Figure 48.

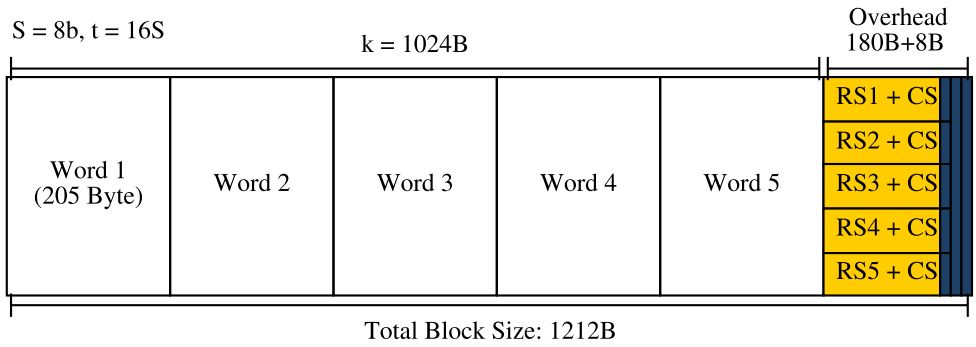


Figure 48: A data block subdivided into 5 subblocks. Separate checksums for the entire data block, EDAC data and each subblock are depicted in blue, which EDAC data is depicted in yellow.

Extended attributes (*xattr*) are deduplicated and referenced by one or more inodes, as depicted in Figure 47. Like in *PRAMFS*, *xattrs* are stored as data blocks, thereby we can treat these identically to regular data.

Nanosatellites, at least the non-classified ones, are not yet considered security critical devices. However, the application area of nanosatellites will expand considerably in the future [220]. An increasing professionalization will introduce enhanced requirements regarding dependability and security. Shared-satellite usage scenarios as well as technology testing satellites will certainly also require stronger security measures, which can be implemented using *xattrs*.

An *xattr* block's integrity is verified once its reference is resolved. Once all write access (in bulk) has been concluded, the EDAC data is updated.

Algorithm Details and Performance

Our primary design objective was to create a filesystem which could be used to store a full size-optimized Linux root FS including a kernel image safely over a long period of time within an 8MB volume. There are numerous erasure codes available that could be used to protect our filesystem, as discussed also by Wylie et al. in [102]. After careful consideration, RS was chosen due to the following reasons:

- The algorithm is well analyzed, and widely used in various embedded scenarios, including spacecraft.
- Highly optimized software implementations of RS encoder and decoder are available as part of standard libraries free of charge and are present in the Linux kernel.
- Open-source and commercial IP-cores are available to achieve hardware accelerations in an FPGA-based system, e.g. from opencores, from Xilinx, and via GRLIB.
- MRAM, while being SEU immune, is still prone to stray-writes, controller errors and in-transit data corruption. Misdirected access within a page evades memory protection and can then corrupt the filesystem, thus corrupted single-byte, 2, 4

and 8B runs will occur. RS relies upon symbol level error correction and can support symbols longer than 8 bit to then allow much larger codewords. This covers well the practical effects of faults will induce in commercial MRAM ICs.

RS decoding is computationally expensive, thus we split protected data into subblocks sized to 128B plus the user specified error number of correction-roots simplifying addressing and guaranteeing data alignment for power-of-two correction-root counts. Inodes and SBs can be fit into one single RS-code, while data block length does not result in extreme checking times. To skip the expensive RS decoding step during regular operation, a CRC32 checksum allows high-performance checking. The RS-code is only read in case the checksum is invalid.

Data blocks are divided into subblocks so the filesystem can make optimal use of the RS code length. For common block-sizes and error correction strengths, 5 to 19 RS codes are necessary, see Table 4 for information on expected overhead. The correction data is accumulated at the end of the data block. Checksums across the entire block's data, each subblock and the error correction data are also retained. The resulting data format is depicted in Figure 48. Protection can be enhanced further by performing symbol interleaving for the RS codes and the block data, at the cost of performance.

Filesystem traversal and data access will eventually slow down for strongly degraded storage volumes. As we immediately commit corrected data to memory, performance degradation is only temporary, assuming soft-faults.

Results and Current Status

FTRFS has been implemented for the Linux kernel. Due to its POSIX-compliance, it could easily be ported to other platforms. The memory protection functionality has been inherited from *PRAMFS*, the filesystem structure from *ext2*. We utilize the RS implementation of the Linux kernel, as its API also supports hardware acceleration.

Several components of the filesystem should undergo an optimization process, which will increase fault coverage capacity and read/write performance. Even though we have not yet conducted long-term benchmarking and performance analysis, the throughput degradation during regular operations is minimal: most modern mobile-market CPU cores can compute CRC32 within a few clock cycles due to hardware acceleration. We intend to publish additional performance and energy consumption

Data Structure	Size (B)	EC-Symbols per Word	Words per Block	Parity (B)	Overhead (B)	Overhead (%)
Super Block	128	32	1	32	68	53.13%
Inode	160	32	1	32	68	42.50%
Data Blocks	1024	4	5	20	68	5.86%
	1024	16	5	80	188	17.58%
	4096	4	17	68	212	4.98%
	4096	16	19	304	692	16.70%
Bitmap	1773	32	10	320	688	38.80%

Table 4: EDAC overhead for FS structures. 16MB volume size, 5% inodes, 1024B bock size

metrics, once testing has been concluded and basic optimizations have been applied and the OBC computer has been finalized.

Data is read and written once per access. It is good practice in critical scenarios and especially spaceflight to read and write data multiple times, or deploy more advanced consistency checking techniques [278]. These changes could be applied in bulk, through a macro, or compiler side.

The level of protection offered by FTRFS is adjustable during volume creation, or later by using a proprietary filesystem-tuning tool. RS has a long record of space use in CDH and communications. Thus, we know the algorithm offers efficient protection regarding our threat scenario. Once testing has been concluded, we will perform long-term performance analysis in a degraded environment. To benchmark the filesystem, data degradation can be introduced through fault injection.

Limitations and Advanced Applications

It is debatable whether journaling would increase FTRFS's reliability, as it usually helps safeguard filesystem consistency with slow storage media [279] due to power loss or disconnect. Spontaneous power loss for an OBC could also occur aboard a spacecraft due to EPS malfunction, but in most cases the practical effects of such an event can be handled differently at the design side. Spacecraft are battery backed and can utilize power electronics with a sufficient hold-back time to notify and gracefully shut down an OBC in case of EPS failure. All access in our filesystem happens synchronously, and MRAM still allows rapid access unlike classical mechanical disks. Hence, FTRFS can thus either conclude a pending write operation within the remaining active time, or the OS will have sufficient time to cancel pending writes in case the system has sufficient warning time. We therefore do not implement journaling.

Our filesystem implementation can currently not handle the failure of entire memory ICs holding the volume, or component-level SEFIs. However, FTRFS could be extended to support RAID-like features to compensate for device failure [277].

If data is stored with RS-symbol interleaving, an XIP mapping would technically be impossible. XIP could still perform mappings for non-interleaved data though, but thereby only the clear-text part of each RS code would be mapped and read. Via this memory mapping, integrity protection for stored file data would be ignored, unless we accept that a potential XIP mapping would allow program code to be loaded/executed without any integrity checking. Thereby, the integrity assumptions upon which FTRFS's concept is based would be violated and integrity could not be guaranteed for any executed program stored on the filesystem. Theoretically, data integrity could also be checked each time a mapping is established for a block. To perform these checks however, this data would have to be read in full, obsoleting the performance advantage and RAM conserving properties of XIP. XIP and filesystem-level data integrity protection can thus be considered mutually exclusive.

Permanent faults would cause fault effects to be corrected upon every access to a memory word, which is inefficient Fault in frequently accessed file system components (e.g., int the root inode), could therefore degrade the performance of FTRFS. In the current filesystem implementation, there is no functionality to avoid this behavior completely. Bad-block relocation is implemented within the filesystem, but only applied during file data write, truncate and allocation operations. This functionality could also be applied to file data read operations as well as for accessed inodes to increase robustness.

FTRFS could also operate on different memory technologies than MRAM, as long as data in this memory is directly addressable RAM or mapped through OS-kernel means (mmap). For more abstract memory technologies such as Flash, FTRFS is not an optimal solution and a block-based approach as described in the next section should be used.

7.5 High-Performance Flash Memory Integrity

Scientific and future commercial space missions as well as miniaturized satellites impose increasing demands on their on-board computer (OBC) systems, especially data storage devices [280]. They may require vast amounts of data to be stored, high throughput, and the possibility for concurrent access of multiple threads, programs or devices. While satisfying these requirements, storage systems must guarantee data integrity and the recovery of degraded or damaged data (error detection and correction – EDAC) over a prolonged period of time in a hostile environment. Consistent data storage becomes even more crucial for long-term missions (e.g., JUICE [281] and Euclid [282]) or in cases where highly scaled memory is used.

Legacy memory technologies can not be scaled for modern storage applications due to mass and energy restrictions or result in high complex storage systems. Thus, single-level cell NAND-flash memories (SLC), have become popular for high performance mass memory scenarios as they offer reasonably high packing density, and can be manufactured sufficiently radiation hardened. The chip-industry has moved on from SLC to multi-level cell flash memories (MLC) due to economical reasons. Therefore, SLC will become unavailable and will force future spacecraft storage concepts to rely upon MLC or entirely different memory technologies. While there are promising candidates [283] to fill this role in the long run, technological evolution does not yet allow, for example, non-volatile magnetoresistive RAM (MRAM) to be used as mass storage [272]. Phase change [284] or charge-trap based memory both would at present be usable as mass storage, but are not yet widely available in high density versions [157].

Traditionally, single-bit error correction, shielding, specialized manufacturing techniques, coarse structure width and redundancy are combined to enable radiation tolerant flash [285]. However, the protective level offered by such solutions is static and fixed at design-time and can result in high cost and low overall efficiency. For miniaturized satellites, cost and efficiency are crucial, thus, countermeasures must be implemented at a different level. With modern MLC-flash single bit error correction is insufficient and all-in-one solutions, such as file systems, tend to become very complex and difficult to debug. For future prolonged missions and larger storage arrays, more sophisticated and efficient EDAC concepts are required. Thus, we present an advanced high performance dependable storage concept based on composite erasure coding. As MLC-flash is also widely used aboard miniaturized satellites, and the authors are involved in developing such a satellite, MOVE-II, development was originally driven by nanosatellite requirements. However, the approach can be applied more efficiently to commercial applications where miniaturization imposed limitations do not apply. The concept could be implemented even more efficiently with very large volumes common in commercial spaceflight applications. It can be implemented entirely in software, with or without hardware acceleration, but also partially or fully in hardware.

Single- and Multi-Level Cell Flash

Each flash memory cell contains a single field effect transistor with an additional floating gate, the basic functionality of which is described further in Chapter 3. The state of a flash memory cell depends on whether the charge stored in the floating gate exceeds a specific threshold voltage (V_t). Hence, a flash memory cell is dependent on the capability of the memory cell structure to retain a charge. If the voltage exceeds the threshold, a cell can be read as programmed (0), else as erased (1), see Figure 49a. Single Level Cell Flash (SLC) cells can store one bit per cell.

The charge in an MLC cell can represent more than two states by introducing additional voltage thresholds. Assuming a four level cell, one can hold four states and represent two bits, as depicted in Figure 49b. The number of levels is not restricted to four, with 2^n states it is possible to encode n bits, but electrical complexity grows and the required read sensitivity and write specificity increase with the number of bits represented. Within nearly the same area of silicon, MLC flash memory thus allows a much higher packing density and the structure itself can be stacked and scaled well [286].

As the delta between voltage thresholds decreases due an increased number of state-levels, increased sensing accuracy is required for read operations, and more precise charge-placement on the floating gate is necessary. MLC memory is thus more dependent on its cells' ability to retain charge. In contrast to SLC, a state machine is required for addressing MLC memory which in turn increases latency and adds considerable overhead logic. Addressing in MLC flash can thus take multiple cycles and the state machine may hang or introduce arbitrary delays.

Due to a shifting voltage threshold in floating gate cells caused by the total ionizing dose, MLC flash memories are more susceptible to bit errors than SLC [153, 287]. Depending on the number of bits represented within a cell, a varying amount of data may thereby be corrupted by a single particle event, as depicted in see Figure 50. EDAC measures must thus compensate for more than single bit corruptions within a given word. Thus software or a filesystem must implement appropriate functionality to handle these effects in addition to erasure coding to safeguard from radiation.

Flash Memory Organization

NAND-flash memories are organized in blocks, consisting of multiple pages, in which cells are connected as NAND gates. In most NAND technologies, pages can be written and read individually, but only the block as a whole can be erased. The drawback

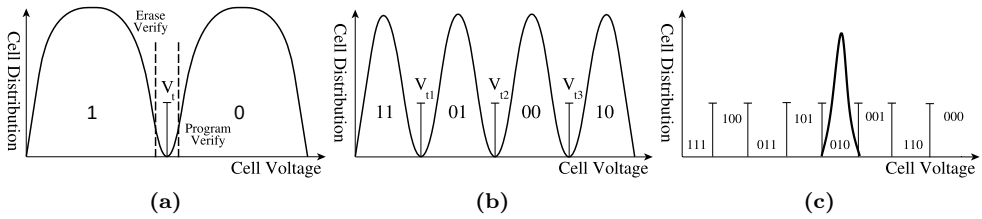


Figure 49: The voltage reference and threshold levels of SLC- flash cells (a) and MLC cells with 4 (b) and 8 voltage levels (c).

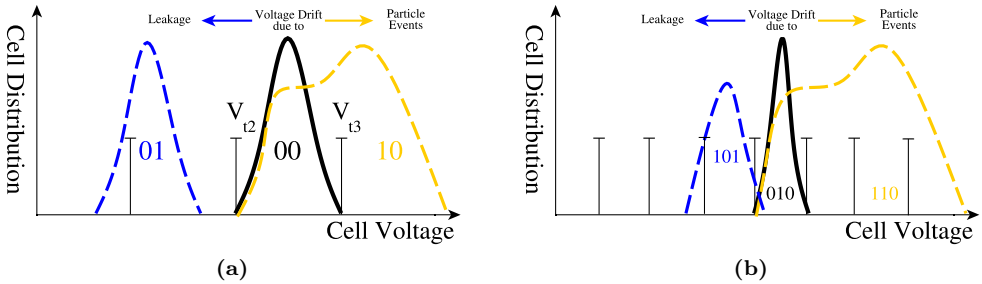


Figure 50: Radiation-induced bit upsets encountered in 4- (a) and 8-level (b) MLC cells.

here is that if a NAND-flash cell fails, the entire NAND block is affected. In NOR-flash, cells form NOR gates, which allow more fine grained read access at the cost of strongly increased wiring and controller overhead. Therefore, in order to appropriately handle NAND-flash block corruption, a filesystem must handle read/write and erase abstraction, as well as basic block FDIR. This is done through the introduction of an additional layer of functionality, the flash translation layer (FTL). When data is written to a flash block, partial erase operations are (usually) impossible and the entire block's previous content first has to be read and updated. Next, the block must be erased (by draining the block's cells' voltage) and may subsequently be programmed anew per page. Thus, read and write operations introduce different latency and make access to MLC flash much more complicated than to SLC due to the required addressing state machine.

To access data and handle special properties of flash efficiently, a filesystem has to interact with the memory device directly or via the OS's FTL. A flash filesystem must implement all functionality necessary to perform block wear leveling, read and erase block abstraction, bad-block relocation and garbage collection (depicted in blue in Figure 51) to prevent premature degradation and failure of a bank. The FTL acts as an interface between hardware specific device drivers and the filesystem, and can provide part of this FDIR functionality instead of the filesystem. In commercial SSD applications, this is handled by the SSD's controller and hidden from the OBC.

Over time, a flash memory bank will accumulate defective pages and blocks hve to and utilize spare pages and blocks to compensate. Traditionally, simple erasure coding (usually some form of cyclic block codes with large symbol sizes) is applied in software or by the controller to counter wear and charge leakage. Eventually, the pool of spares will be depleted, in which case the FTL or filesystem will begin recycling less defective blocks and compensate with erasure coding only, thereby sacrificing performance to a certain degree. For space use, the erasure codes' symbol size is usually reduced to support one or two bit correcting erasure coding, as corruption will mostly result from radiation effects. However, if this solution is applied for MLC-NAND-flash, block EDAC becomes very inefficient due to the occurrence of both single bit- and grouped errors, the latter being induced by SEUs affecting multiple cells in highly scaled memory.

Majority Voting for Flash Memories

While voting is technically still possible for MLC-flash, it is severely constrained by the additional circuitry, logic and strongly varying timing behavior. Voting would have to be implemented for the addressing state machine as well, otherwise it could stall the entire voting circuit or permanently disable its memory bank. Due to the varying timing behavior of NAND-flash and the more complex logic, the resulting voter-circuit thus becomes more error prone. The added logic also requires more energy and reduces overall performance. Of course the slowest memory bank or block also dictates performance of the voting circuit.

7.5.1 The MTD-mirror Middleware Layer

As outlined in the previous sections, error correction is crucial for current data storage based on NAND-flash. To enable future dependable MLC-NAND-flash based data storage solutions for space flight applications, existing EDAC functionality can be adapted and improvements added where necessary. Thus, we developed a storage system to satisfy the following requirements:

1. Efficient, fast data storage on MLC mass-memory.
2. Integrity protection and error correction with adjustable strength, to allow optimization according to mission duration, environment and type.
3. Efficient handling of direct and indirect radiation effects on the memory as well as the control logic.
4. Protection against device failure.

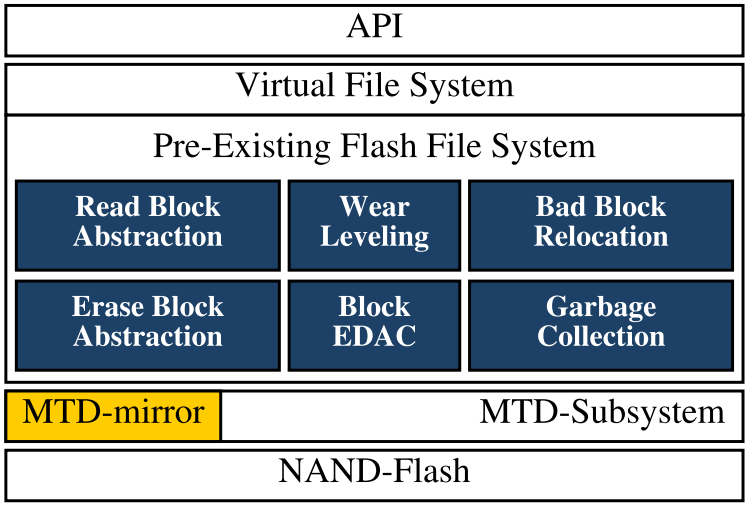


Figure 51: Memory access hierarchy for an MTD-Mirror set. Flash-memory specific logic is depicted in blue and partially resides within the FTL. Required modifications to enable the concept are depicted in yellow.

5. Low soft- and hardware complexity: While a certain level of complexity is acceptable for commercial spaceflight applications, it is crucial in microsatellite design.

6. Universal filesystem support and interactivity.

We consider these requirements to be met best through enhanced EDAC functionality as FTL-middleware. At this level, RAID-like features and checksumming can be combined most effectively with a composite erasure coding system. As our use case includes a Linux based OBC, we implemented MTD-mirror on the memory technology device (MTD) FTL subsystem of the Linux Kernel. The solution is depicted in Figure 51. Any unmodified flash-aware filesystem can be deployed on top of the MTD-mirror set. By utilizing mirroring (RAID1) and distributed parity (RAID5/6) we can therefore protect against device, bank and block failure. Within this section we focus on mirroring, as the basic concept is very similar to distributed parity sets.

To safeguard against permanent block defects, single event functional interrupts, radiation induced programmatic errors and logic related problems, we apply coarse symbol level erasure coding. As this is insufficient to compensate for radiation effects, silent data corruption and bit flips are compensated using bit-wise error correction. The solution was implemented in the FTL, as the required logic can still be kept abstract and device independent while it can profit significantly from hardware acceleration. The FTL-middleware also provides enhanced diagnostics, as no further abstraction is introduced.

Alternative Approaches

EDAC and device independence could also be provided by an filesystem directly, which we showed for MRAM with FTRFS in Section 7.4. A Flash filesystem such as UFFS could be extended to handle multiple memory devices and EDAC, or FTRFS could be modified to handle flash memory. Even though possible to implement, such an all-in-one filesystem would be complex and error prone.

Device independence could also be added on top of an existing flash filesystem as a separate layer of software [96], see Figure 52. Within a RAIF set, increased protective requirements could be satisfied with additional redundant copies of the filesystem content. The underlying individual filesystems would then have to handle all EDAC functionality and escalate fatal errors and unrecoverable file issues to the set, as RAIF by itself does not offer any integrity guarantees beyond filesystem or file failure.

Since RAIF only reads from underlying filesystems, it is prone to filesystem-metadata corruption which can result in single block errors failing entire filesystems. Additionally, Flash-filesystems usually rely upon parameter-fixed block based error correction and do not offer configurable protection for different filesystem structures, which is at best sub-optimal for space use.

A file damaged in different locations across the set's filesystems would become unrecoverable as RAIF would discard information regarding the location of damage to a file and in the best case would forward a defective copy to the application. It would therefore inhibit error correction and may even cripple recovery of larger files. While RAIF could be adapted to handle these issues, the resulting storage architecture would again become very complex, difficult to validate and debug. As RAIF implements filesystem redundancy, its storage efficiency will furthermore be inferior to distributed

parity concepts such as the more advanced variants of the presented concept. As a pure software layer without the possibility to interact with the devices, hardware acceleration of RAIF would be impossible.

Device Failure and SEFI Protection

In contrast to RAIF, RAID can been applied efficiently to storage architectures and has been used previously aboard spacecraft (e.g., in the GAIA mission) [288]. However, these were based on SLC (see Section 7.5) and only relied on RAID to achieve device failover through data mirroring (RAID1) and distributed parity (RAID5/6) [288,289]. As RAID itself does not offer any integrity guarantees beyond protection against read device failure, designs usually rely upon the block level hardware error correction provided by the flash memory or controller or implement simple parity only.

The main issue encountered with plain RAID setups is the absence of validation for a block or group of blocks. RAID merely retains redundant copies of data – parity – which can be used to restore lost data. RAID foresees that a data block is either unrecoverably lost (signaled by a read error) or fully intact; it is thereby prone to silent data corruption encountered in flash memory [72]. As the basic RAID concepts do not utilize checksumming to verify integrity, corrupted data will be read and used even if sufficient parity or valid copies were available. However, once checksumming and forward error correction is added to RAID levels, they can be utilized aboard spacecraft efficiently.

The even distribution of bit-errors would be troublesome for symbol based erasure codes traditionally applied for flash block EDAC. Utilizing RAID on top of block based erasure coding is thus insufficient for protecting MLC-NAND-flash.

RAID functionality usually would be implemented as a block layer. This is certainly

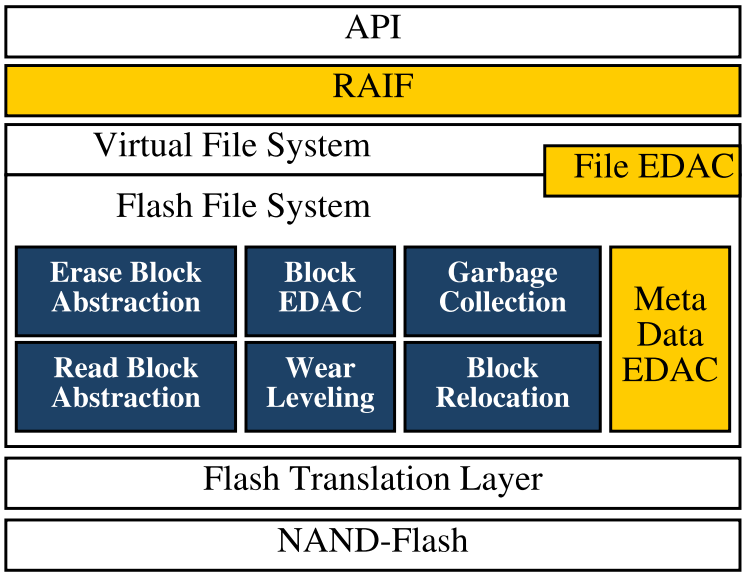


Figure 52: Memory access hierarchy for an enhanced RAIF based concept with added filesystem level error correction.

possible also for flash memory, however it would hinder the file system from performing block EDAC and wear leveling. While block abstraction would still be possible even on top of a block layer on-top of the FTL, other high-level filesystem functionality would be denied device access, depicted in red in Figure 53. These functions would then have to be implemented at a much lower level within the access hierarchy, introducing further code overhead and reducing EDAC efficiency.

RAID-like functionality could however also be implemented as a middleware within the FTL as depicted in Figure 51. As such, it can interact both with the underlying flash memory as well as the filesystem and the rest of the FTL, without requiring alterations to either. Such middleware can remain pervious to filesystem operations requiring direct interactivity with the underlying flash and at the same time allow device failure protection to be combined with enhanced erasure coding. RAID can therein be implemented with comparably little effort. Validation, testing and analysis can thus be simplified as all implementation work can be concentrated into an FTL middleware module.

Block-Level Consistency

MTD-mirror's block consistency protection is depicted in Figure 54 and includes two checksums and error encoding layers. Thus, it implements a concatenated/composite

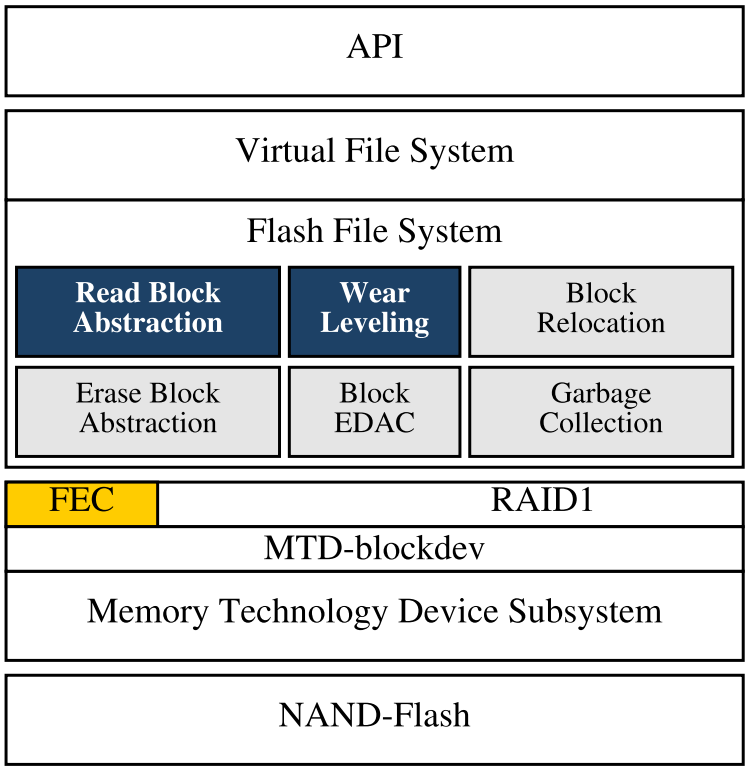


Figure 53: RAID prevents EDAC and wear leveling functionality withing a flash-filesystem from being implemented. Affected elements are colorized in gray.

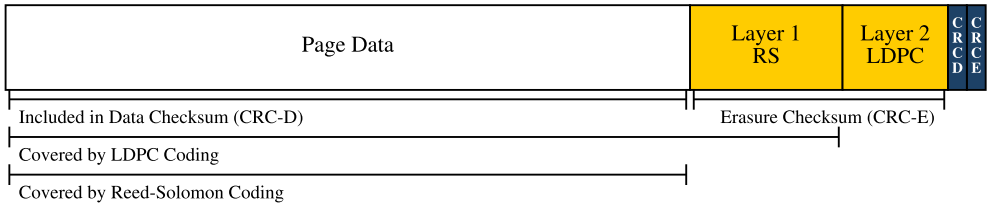


Figure 54: The layout of an MTD-mirror page. Added erasure code correction information is depicted in yellow, checksums in blue.

erasure code system. The data checksum allows bypassing decoding of intact data, which will often be the optimistic default case. The second checksum can be used for error-scrubbing of erasure data and prevents symbol drift of the RS-layer. Even though CRC16 could be considered sufficient for most common page and block sizes, we utilize a 32-bit checksum to further minimize collision probability at a minimal compute overhead.

Protection against Multi-Bit Upsets

The first layer of erasure coding is based on relatively coarse symbols and protects against data corruption induced by stray writes, controller issues and multi-bit errors. As data on NAND-flash is stored in pages and blocks of fixed length and the coding layer should protect against corruption up to 8 byte length (`int64_t`), Reed-Solomon (RS) erasure coding [103] was selected. We chose to rely on the RS block code as the algorithm is well analyzed, and widely used with NAND-flash memory and in various embedded scenarios, including spacecraft. Optimized software implementations, IP-cores and hardware acceleration are available.

Erasure coding with coarse symbols is efficient if symbols are largely or entirely corrupted, but shows weak performance when compensating radiation-induced bit-rot, to which MLC is comparably prone. SEUs will be evenly distributed across the memory and will thus equally degrade all data of a code word, corrupting multiple code symbols with comparably few bit errors. Therefore, RS is applied at the page level, instead of the block level to allow more efficient reads and avoid access to other pages within the same block to retrieve erasure coding parity. RS parity is therefore stored within each page, together with a checksum for the page and the parity. RS encoding and decoding can be should parallelized due to the small word sizes in hardware.

Bit-Level Erasure Coding

Previous radiation-tolerant OBC storage concepts often relied upon convolution codes as these allow efficient single-bit error correction. However, as error-models become more complex (2-bit errors as in MLC), codes complexity increases and efficiency diminishes. Therefore, a second level of erasure coding using Low-Density Parity Check Codes (LDPC) [290] was added to counter single or double bit-flips within individual code symbols of the first level RS code. LDPC was chosen as it is efficient with very small symbol sizes (1 or two bit), offers superior performance compared to convolution codes [291], and allows iterative decoding [292]. Only if RS decoding fails, the set resorts to LDPC. LCPC can then support recovery of slightly corrupted

RS-symbols and parity. Thereby otherwise unrecoverable data can be repaired by salvaging damaged symbols which can drastically increase recovery rates on radiation-degraded memories.

Although LDPC codes benefit from longer code word lengths, Morita et al. [293] show that the gain from a 4KB code to a 32KB code can be negligible. For systems where buffer memory is scarce, it may therefore be of advantage to use comparably small codes and sacrifice a bit of LDPC performance. Thus, an LDPC word size between 3 and 4KB offers solid LDPC performance without requiring very large words and thereby enable fast iterative decoding.

Joint Iterative Decoding using Soft-Output Shorter code words also enable joint iterative decoding [294] using soft-output for both LDPC and RS codes. LDPC can be adjusted to output not only plain copy of the expected original code word, but can also yield the decoder's certainty about each bit's value. Equally, an RS decoder could be extended to handle such soft-input. Then, it could attempt decoding to decode multiple variants of a corrupted word using different uncertain positional values from the LDPC soft-output.

In practice, this allows us to produce linear composite erasure coding system, which we depict in Figure 55. However, decoding does not have to happen linearly: A hardware-implemented LDPC decoder has a considerable logic footprint, while RS decoding can be parallelized. Hence, it may be desirable to construct such a composite system by paralellizing RS decoding.

As depicted in Figure 55, a closed feedback-loop that inputs the soft message output $R(Y)$ of the LDPC decoder into RS can be constructed. The system iterates between RS and LDPC decoding until either decoder can reconstruct a valid code word. To tackle the issue of the thereby variable timing behavior, the number of iterations can be limited or a timeout can be defined.

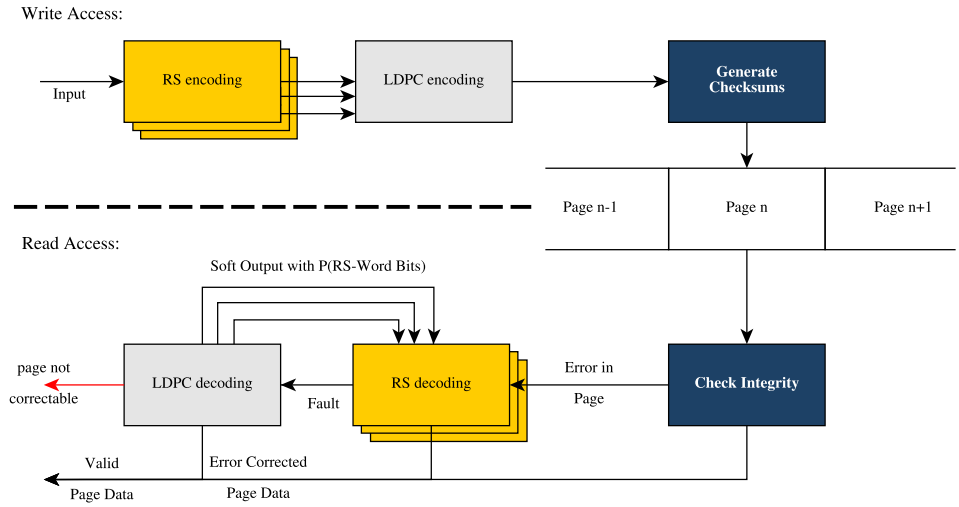


Figure 55: Joint iterative decoding using LDPC soft-output with added parallelization (triple-arrows). RS encoding can be parallelized to increase write-throughput. Speculative RS-decoding could be utilized to reduce LDPC iterations by performing multiple parallel RS-decoding attempts with different values for low-certainty bits.

Error Handling Runtime Behavior In case the checksum does not match the plain block data, an MTD-mirror set will first attempt to retrieve an intact copy of the data from another memory of the RAID-set. If this fails, or all other blocks are invalid as well, erasure decoding for the damaged block is attempted. As multiple copies of the erasure code parity data and checksums are available, the set can also attempt repair using fields of different blocks in the hope of obtaining a consistent combination of block-data. This behavior can allow recovery even of strongly degraded data or permanently defective blocks.

As RS hardware-acceleration is readily available in our use-case, we apply the two FEC layers in order (Figure 54). However, the sequence can be chosen based on the individual system design, the used algorithmic parameters, the available acceleration possibilities and phase of the mission. An important aspect for this decision is the expected level of degradation of the utilized flash memory due to radiation, thus the occurrence of single bit errors. If severe bit-rot is expected or higher order density MLC is used, the LDPC-layer should be applied prior to RS decoding. Thereby, the increased probability of the second FEC layer failing to recover data is accepted in the hope of achieving a sufficiently high amount of intact code symbols.

7.5.2 Advanced Applications

In this section, we focused on describing a storage solution based on RAID1 for simplicity reasons. While the logic required to implement this storage solution is relatively simple, more advanced distributed parity RAID concepts offer increased mass/cost/energy efficiency due to overhead reduction. Thus, we have been working to adapt and expand MTD-mirror to benefit from such more advanced architectures.

There has been prior research on adding checksumming support to RAID5 in [288, 289], though utilizing RAID5 directly would introduce certain problematic aspects. Error correction information in RAID5C can either be stored redundantly with each block, introducing unnecessary overhead, or as single copy within the parity-block. While this would increase the net storage capacity, a single point of failure would be introduced for each block group. If the parity block was lost, the integrity of data which was protected by this block could no longer be verified. Instead, RAID5 can be applied to data and error correction information independently, only requiring one extra checksum to be stored with each block.

RAID6, however, can be implemented almost as-is, with error correction data and checksums being stored directly on the two or more parity blocks associated with each group. There are also promising concepts for utilizing erasure coding for generating parity blocks by themselves, thereby obsoleting simple hamming-distance based parity coding [97,295]. Further research on this topic is required and may enable optimization for flash memory and radiation aspects similar to the ones described in this paper.

7.6 Conclusions

In this chapter we presented three software-driven concepts to assure storage consistency, each specifically designed towards protecting key OBC components: a system for volatile memory protection, FTRFS to protect firmware or OS images and MTD-mirror to safeguard payload data. All outlined solutions can be applied to different OBC designs and do not require the OBC to be specifically designed for them. They

can be used universally in miniaturized satellite architectures for both long and short-term missions, thereby laying the foundation to fault tolerance at the system level. In contrast to earlier concepts, none of the approaches requires or enforces design-time fixed protection parameters. Both can be implemented either completely in software, or as hardware accelerated hybrids. The protective guarantees offered are fully run-time configurable.

Assuring integrity of core system storage up to a size of several gigabytes, FTRFS enables a software-side protective scheme against data degradation. Thereby, we have demonstrated the feasibility of a simple bootable, POSIX-compatible filesystem which can efficiently protect a full OS image. The MTD-mirror middleware enables reliable high-performance MLC-NAND-flash usage with a minimal set of software and logic. MTD-mirror is independent of the particular memory devices and can be entirely based on nanosatellite-compatible flash chips by utilizing FEC enabled RAID1 and checksumming.

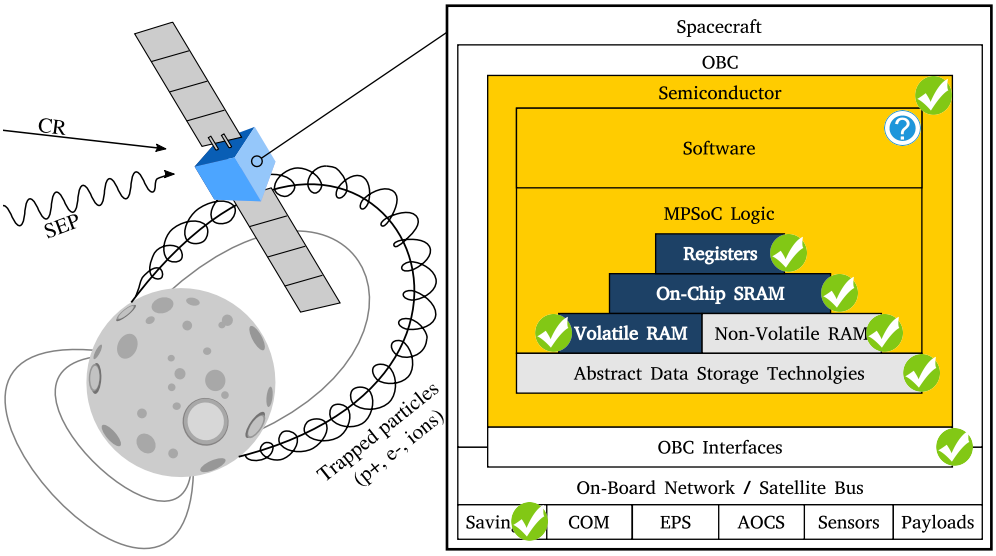
Neither traditional hardware nor pure software measures individually can guarantee sufficiently strong system consistency for long-term missions. Traditionally, stronger EDAC and component-redundancy are used to compensate for radiation effects in space systems, which does not scale for complex systems and results in increased energy consumption. While redundancy and hardware-side voting can protect well from device failure, data integrity protection is difficult at this level. A combination of hardware and software measures, as outlined in this chapter, thus can increase robustness, especially for missions with a very long duration. Thereby, a low-complexity satellite architecture can be maintained, thereby error sources reduced, while testability and throughput can be increased.

Chapter 8

Validating Software-Implemented Fault Tolerance

Systematic Fault Injection

In this chapter, we test and validate the software-mechanisms that are the foundation of our fault tolerance architecture to address RQ5. Therefore, we conducted a fault-injection campaign through system emulation with QEMU into a ARMv7a-SoC matching our architecture target ARM's Cortex-A53. Our results show that our lockstep implementation is effective and efficient for providing FDIR within our system, and the thread-level coarse grain lockstep's performance meets our requirements. To place our results into context, we compared them to literature and discuss lessons learned and knowledge obtained throughout our fault injection campaign.



8.1 Introduction

Modern embedded technology is a driving factor in satellite miniaturization, which today enables an entire class of smaller, lighter, and cheaper class of spacecraft. These micro- and nanosatellites (100kg-1kg mass) have become increasingly popular for a variety of commercial and scientific missions, which were considered infeasible in the past. They are drivers of a massive boom in satellite launches, new scientific and commercial space missions, laying the foundation for a rapidly evolving new space industry. However, these spacecraft suffer from low reliability, discouraging their use in long or critical missions, and for high-priority science.

For larger spacecraft, various protective concepts are available to assure fault tolerance (FT) through hardware measures. However, these concepts are effective only for traditional semiconductors manufactured in technology nodes with a large feature size. Such hardware can not be utilized aboard miniaturized spacecraft due to tight energy, mass, volume constraints, and high cost. Conventional embedded and mobile-market systems-on-chip (SoCs) are deployed in their stead, which only utilize error correction to handle wear and aging effects encountered on the ground. A significant share of post-deployment issues aboard nanosatellites can be attributed directly to the failure of these components and peripheral electronics [2], which caused usually by design failures and effects induced by the space environment, e.g., [296].

Therefore, we developed a non-intrusive, flexible, hybrid hardware/software architecture (see Chapter 4) to assure FT with commercial-off-the-shelf (COTS) mobile-market technology based on an FPGA-implemented MPSoC design. Our architecture utilizes multiple FT measures across the embedded stack, and runs software in coarse-grain thread-level lockstep to assure computation correctness through replication. It can offer strong fault coverage without relying upon any space-proprietary logic, custom processor cores, or other radiation-hardening measures in hardware.

The utilized lockstep concept facilitates state synchronization and forward error correction between otherwise independent processor cores. It also provides fault detection capabilities for other FT stages which otherwise would lack fault detection capabilities: FPGA reconfiguration and dynamic thread-replication and relocation based on mixed criticality. Therefore, it not only offers fault coverage, but also triggers other protective features of our architecture, requiring thorough validation before a custom-PCB based prototype can be constructed.

Validation of such FT measures requires systematic testing of the actual concept implementation, a realistic fault model, a consistent fault model definition, and a suitable test setup. As our lockstep is part of the operating system kernel, system-level fault injection and application-level testing do not offer a sufficient level of test-coverage, and instead a variety of fault injection techniques for software are available. While validation using fault injection using a realistic test-setup is best practice in fault tolerance research and space-hardware development, very few coarse-grain lockstep concepts have been implemented and validated in this way. Most concepts described in academic publications today, instead are validated only using mathematical models only, but were not actually implemented or practically validated.

At the time of writing the 2018 – 2019 period, careful study of journals and conference proceedings yields only a single coarse-grain lockstep concept [199] that was practically implemented, and validated based on a realistic fault profile. Practical implementation and the possibility to compare an implementation's performance to

literature, however, is seen as a prerequisite by industrial users to consider an FT concept mature enough for practical application. This situation has resulted in a gap between theory and application, with industry often dismissing software-implemented FT concepts due to a (perceived?) lack of maturity and an (assumed?) tendency to ignore practical implementation obstacles. The research results of an entire field of research, dependable computing through software measures, are thus practically barred from application for an entire industry segment even though there would be a pressing technological need and a lack of viable alternatives. For critical applications like in the space industry, practical concept validation is then just the first of many validation and testing steps: eventually system-level testing is conducted with a hardware/software prototype. For space application, this prototype is then subjected to radiation testing followed by on-orbit demonstration.

8.1.1 Contributions

In this chapter, we show how software-implemented FT concepts can be validated for space applications in a realistic and representative manner, and fields with a similar fault profile, e.g., critical and irradiated environments. We do so by example of a fault-injection campaign we conducted to validate a novel thread-level coarse grain lockstep concept we developed for space applications, described in detail in Chapter 4. We utilize ISA-level fault injection into an ARM Cortex-A system through virtualization, and fault injection into a 3-core SystemC-implemented MPSoC. This chapter includes not only concept validation but is meant as a template for other researchers who wish to validate their own software-implemented FT concepts. We provide a detailed description of the fault profile in the space environment, and a through description of the utilized tools and scripts, which have been made available to the public. Thereby, we hope to increase acceptance of software implemented FT concepts by industry, and the share of concepts which are validated in a practically meaningful way.

A single set of data points is insufficient to judge the performance and effectiveness of the entire coarse-grain lockstep concept class. Thus, it is of great importance to offer a second set of validation results to allow fellow researchers to compare their forthcoming results to more than just one single paper. We document a variety of lessons learned as part of this campaign, which have allowed us to develop a better understand the practical behavior and protective properties of coarse-grained lockstep in critical systems.

Few software-implemented FT concepts proposed today have been implemented, and only a handful have been validated in a realistic and meaningful way. Therefore this chapter serves as practical guide for fellow researchers that can be used as walk-through to make proper testing of fault tolerance techniques a less challenging and time consuming task in an academic environment. The strategy which we describe throughout the remainder of this chapter is depicted in Figure 56, and described briefly below.

8.1.2 Chapter Organization

In the next section, we discuss how the challenges of the space environment described in Chapter 3 are met today in the industry, outline which solutions currently are available, and how these are tested. We then derive a practical fault model for an RTOS implementation of this approach (Section 8.4), and analyze which testing techniques

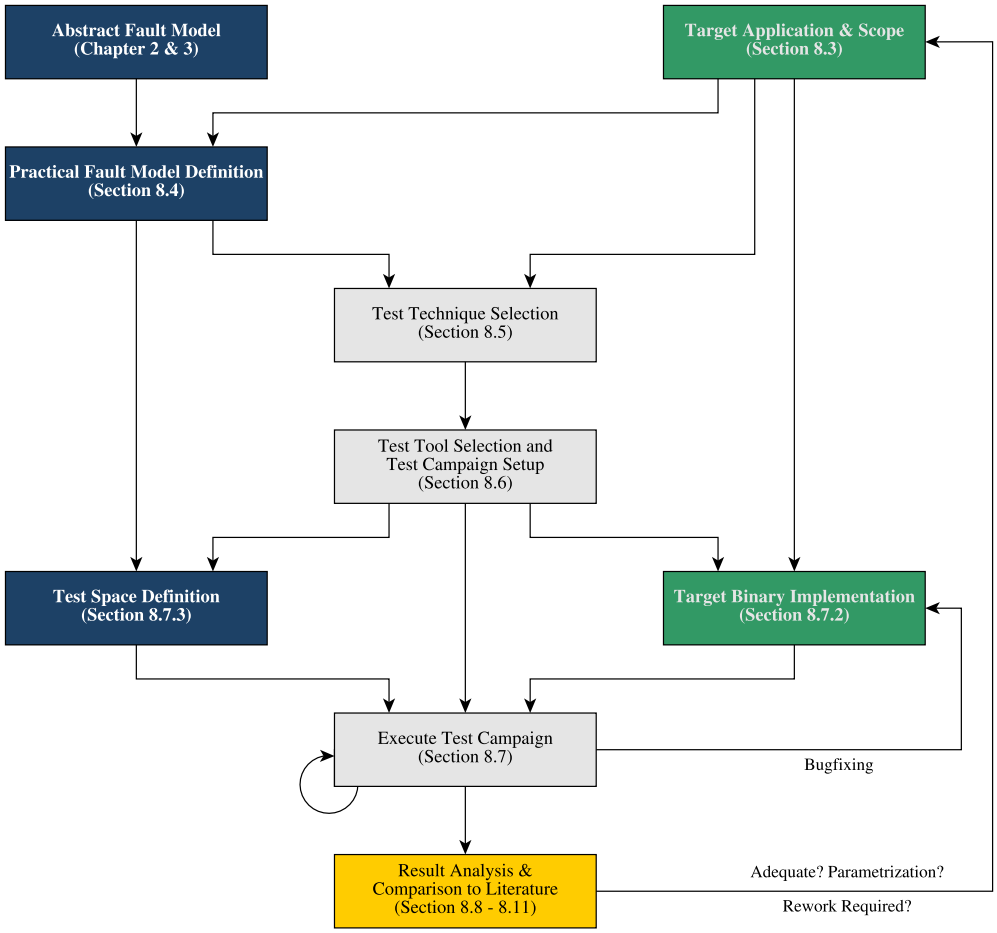


Figure 56: The top-down step-by-step testing strategy described in this chapter, with indications in which section each step is discussed.

are available to verify the lockstep in Section 8.5. Having chosen the most suitable fault injection techniques for our architecture, in Section 8.7 we describe the automated test toolchain we developed to systematically conduct our test campaign. We utilize a set of fault-templates to inject the different faults types described in Section 8.7.3, which we derive from our fault model. The results of our fault injection campaign are presented in Section 8.8, and we compare them to related work in 8.10. Before presenting conclusions, we document pitfalls encountered while preparing and conducting our campaign in Section 8.11, and describe changes made due to lessons learned during validation.

8.2 Related Work

Computer architectures for space-use usually undergo radiation testing or laser fault injection, as the state of the art in the field today is focused on hardware-level FT

measures or specialized manufacturing (RHBD and RHBM – radiation hardened by design/manufacturing). FT is traditionally implemented through circuit-, RTL-, core-, and OBC-level majority voting [104, 132, 188] using space-proprietary IP, which is difficult and costly to maintain and test. Circuit-, RTL-, and core-level voting are effective for small SoCs such as microcontrollers, but this does not scale for the more potent processor cores used in modern mobile-market MPSoCs [88, 191]. Software takes no active part in fault mitigation within such systems, as faults are suppressed at the circuit level and usually only indicated using hardware fault counters, without a direct feedback between fault-mitigation and software. Hence, testing is strongly focused on the pure hardware with software functionality during tests often being reduced to stub implementations to assert basic functionality.

The characterization of the effects induced by radiation within a semiconductor is of major concern when implementing traditional hardware-FT based systems. Today, radiation testing is the only practical way to evaluate them, with radiation models offering useful but tentative and often inaccurate high-level fault estimates. Radiation test results for different components including memory and watchdog/supervisor-μCs are available in databases such as ESCIES, NASA's NEPP[1] and the IEEE REDW Records. Relevant radiation tests have been conducted for the FPGAs utilized in our project, among others by Lee et al. in [297] and Berg et al. in [143], or are currently ongoing (Glorieux et al. [298, 299]).

Radiation testing can occur only at a very late stage in development, and the results may vary even for identical chip-designs manufactured in different fabs and fabrication lines. This form of testing effectively yields heritage and increases a system's technology readiness level, instead of verifying the effectiveness of a specific FT mechanism. For our architecture, radiation tests yield device-specific data, which enabling us to estimate fault frequencies, types, and effects on the FPGA on which our MPSoC is implemented. We require this information to choose an appropriate checkpoint frequency and frame times for our coarse-grain lockstep approach. By itself, however, radiation tests do not allow an assessment of the capabilities of software-implemented FT measures.

While transient random bit-flips are often considered in academic literature, the otherwise different fault model [5] prevents the re-use of many FT approaches developed for ground applications. Also, the form factor constraints aboard miniaturized satellites [197] prevent the re-use of most high-availability and failover concepts for critical terrestrial control applications. Even for atmospheric aerospace applications, dependable computing usually considers availability, non-stop operation, and safety, but rarely computational correctness in a fully isolated and autonomous system.

Prior research on software-implemented FT often considers faults to be isolated, side effect free and local to an individual application thread [208] or purely transient [199, 205]. Many practical application obstacles could be uncovered and resolved before publication by implementing these concepts [198]. However, implementation of a measure and fault injection are time consuming tasks [300]. They often require not only software to be implemented, but also suitable tools and hardware or a representative substitute, as outlined among others by Sangchoolie et al. in [301]. Especially fault injection for entire OS instances is non-trivial [302], as thorough preparation and careful test-tool selection is necessary to obtain representative results from a fault injection experiment [303]. Therefore, a sizable share of FT concepts exists at a theo-

[1] see `https://escies.org` and `https://nepp.nasa.gov`

retical level [212–214], instead of having undergone fault injection or hardware testing. To still achieve some degree of validation, many publications thus resort to statistical modeling using different fault distributions. This is a viable approach for validating FT concepts directed towards, e.g., yield maximization [58] and aging [304], but not for software-implemented FT measures for critical environments.

In this chapter, we conduct systematic validation of our coarse-grain lockstep approach using fault injection to verify the effectiveness and efficiency of our coarse-grain lockstep FDIR mechanisms under stress. Specifically, we must assure voter stability and a sufficient level of fault detection to avoid accumulating silent data corruption and excessively brief frame times, while helping assess the amount of spare resources needed. Together with FPGA-level fault-information obtained from radiation tests outlined earlier in this section, and information on the mission specific target environment, we can then calculate the appropriate fault-frequency for a specific mission and spacecraft.

8.3 Target Implementation

The high-level logic of our architecture is depicted in Figure 57, and consists of three interlinked fault mitigation stages implemented across the embedded stack. It is described in detail in Chapters 4 through 6. At the core of this architecture is a coarse-grain thread-level lockstep implemented within the kernel of an OS, which we refer to as Stage 1. It implements forward error correction and utilizes coarse-grain lockstep to generate a distributed majority decision for an operating system. The thread-level

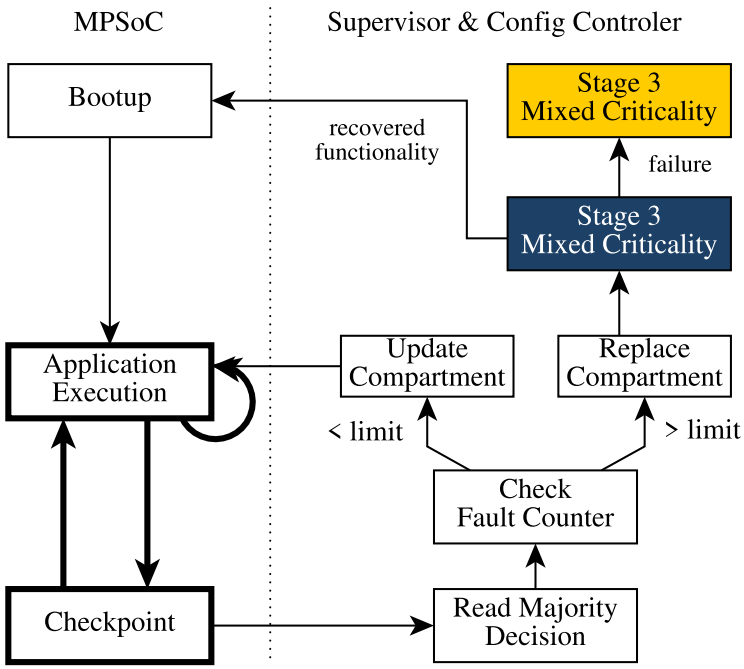


Figure 57: Stage 1 (white) assures fault detection (bold) and fault coverage, Stage 2 (blue) and 3 (yellow) counter resource exhaustion and adapt to reduced system resources.

lockstep assures the integrity of software replicas run on a set of otherwise isolated, weakly coupled processor cores. Fault detection is facilitated through application-provided callback functions, requiring no knowledge about application intrinsics and also no modifications to the application structure. Faults are resolved through state re-synchronization and thread migration to processors with spare processing capacity. Stage 1 is described in further in Chapter 4, where we also establish an upper bound for the performance cost of the lockstep. This coarse-grain lockstep is validated in this chapter, and provides fault-detection capacity for the subsequent stages and short-term fault-recovery.

8.4 Obtaining a Practical Fault Model

To properly validate software-implemented FT measures, information on the physical fault model is required. This information is necessary to choose a fault-injection technique and the right tools to inject the faults. In the remainder of this section, we show how to deduct a practical fault model from our operating environment. This enables us to subsequently determine the most suitable fault injection technique as well as to build a concrete test-space for our fault injection campaign.

To validate our lockstep implementation, we must specifically test how well our lockstep implementation can detect faults. We need to verify this not only at the system level, following a majority decision by all involved compartments, but also locally by an individual lockstepped compartment into which a fault has been injected. Besides fault detection and the possibility for recovery, it is necessary to determine how stable or unstable a lockstep will behave. For space applications, a software-implemented FT concept must be subjected to transient faults, permanent faults, faults that are neither (intermittent faults). The effect of a radiation induced fault depends on the particular effected chip region, logic, and microfabrication technology used [5].

Our coarse-grain lockstep exists as part of the scheduler and utilizes a set of application callbacks. Therefore, we must consider the actual effect and impact of faults on the system from a programmatic perspective. Radiation induced faults will, thus, have the following effects on the software executed within one of our MPSoC's compartments:

- Data corruption associated with access to main memory, caches, registers and scratchpad memory due to non-correctable ECC words caused by SEEs.

- Bit upsets, new-value, and zero-value faults due to SEEs and SEFIs in address and control logic of peripheral IP due.

- Incorrect or non-execution of instructions in the processor pipeline during the entire sequence of processing, i.e. from instruction fetch, execute to write-back, as well as incorrect decoding of instructions and execution of different instructions with the given parameters.

- Control-flow deviations and data corruption due to failure of interfaces and compartment I/O peripherals, due to faults in controller logic of FPGA's I/O components.

To properly represent these faults, we should inject both bit-flips and new-values. Random fuzzing or type-fault injection are widely used for finding exploits and vulnerabilities in software, as well as logic bugs, but are not useful for our purposes due to the different physical fault scenario. Proper validation for software must be systematic [305], which can not be achieved at the system-level when testing a physical hardware prototype. Software must be tested separately and systematically, so that then a prototype can be developed that can undergo system-level testing.

A broad variety of synthetic, theoretical failure types are well described in literature, e.g., in [303]. In practice these do emerge as one of the described fault types. As discussed among others in [306], most of these synthetic failure modes [303] actually emerge as one of the aforementioned effects. To validate the fault-detection and mitigation capabilities of our lockstep to radiation effects, we are only interested in the practical effects of a fault, not its theoretical origin, as discussed further by Sangchoolie et al. in [301].

Radiation can induce subtle effects into logic and may affect the OBC at a system level (e.g., full component failure or reset) [143]. Such faults emerge disguised as one of the aforementioned ones in case their effects are transient or intermittent. Furthermore, we also need to test the lockstep's behavior under permanent faults.

Faults with a permanent effect are either fatal to a compartment, therefore directly detectable by other compartments by majority decision, or affect the system as a whole. Our lockstep is not designed to recover the system from large-scale system-level permanent faults, and utilizes spare resources to cover the permanent failure of individual compartments. These are covered by Stage 2 and, if necessary, escalated to or detected by the on-board computer's external supervisor through time-out.

8.5 Suitable Fault-Injection Techniques

Fault injection into a live hardware-system or an FPGA (e.g., using JTAG or ICAP) would be most straight forward way of conducting fault injection. As research budgets are finite, this naive approach does not allow a meaningful level of test coverage from being achieved, as systematic test coverage is potentially destructive [115], time consuming, and would require a high degree of parallelization. [307]

As our architecture is designed for FPGA, fault injection using netlist simulation [64] or directly into the FPGA [115, 308] could be facilitated with comparably[2] little development effort, as we already utilize a development-board based MPSoC design implementation. This technique would grant precise control over the type and effect of faults and the simulation could be conducted with a system closely corresponding to the real one. Several proprietary partially [115, 308, 309] and fully automated test frameworks [310] as well as commercial applications [64] have been developed for this purpose. Unfortunately, netlist simulation of a full MPSoC is computationally disproportionately expensive. Therefore, netlist simulation, too, does not allow us to achieve meaningful level of test coverage.

Faults could also be injected via widely available standard software debug tools (e.g., GDB) into software running in userland. This is only representative for tests considering only the effects of transient faults in simple userland applications [199]. The effects of faults on a full OS implementation and permanent component damage

[2] as compared to developing a new FPGA design from scratch for the purpose of testing.

cannot be simulated [311]. Furthermore, validation of embedded software for low-power ARM or RISC-V SoCs using desktop-grade ia32/amd64 hosts may bias the outcome of a fault injection experiment, as the platforms and their ABIs are fundamentally different. Fault injection into kernel functionality emulated in userland may also result in a different run-time behavior than when running bare-metal. This technique can therefore only yield meaningful validation results for pure application level FT concepts [303]. Debugger-driven fault injection into a virtual machine can alleviate these constraints by allowing an actual OS to be tested. However, this technique is unable to correctly simulate permanent and intermittent faults in components other than memory and the current execution context. In consequence, the fault injection using debug tools is significantly constrained [303] and insufficient for validating our lockstep. This is an inherent limitation of that can only be alleviated through cooperation of a virtual machine monitor without hardware acceleration [302].

ISA-level binary instrumentation has been shown powerful and efficient for conducting black- and grey-box fault injection [301], and is today widely used for reverse engineering, security and malware analysis purposes. Though most of these tools are tuned towards reverse engineering, not fault injection. Fault-injection capable tools discussed today in relevant publications are mostly proprietary to individual research groups [301, 312]. Without exception, they are rather experimental and tuned towards single applications, and often also simply not publicly available [312]. To be comparable however, proprietary tools unavailable to all but a research group are not relevant.

Fault-injection into a virtual machine (VM), in contrast, allows considerable code and tool reuse: a VM can be constructed using pre-existing virtualized hardware available in widely used standard tools. Due to the considerable optimization effort invested into virtual machine monitors, this technique is computationally relatively cheap. Depending on the used VM technology, it no changes are to a victim application and the emulated machine be can resemble the actual intended target system rather closely. Several test frameworks implementing this approach have emerged in recent years, though most are still custom tailored for specific usecases or have not been released to the public [300, 305]. Notable exceptions here are the two open source frameworks FAIL [306] and FIES [313]. These are publicly and freely available as open source software and reasonably mature, and therefore we began to conduct our fault-injection campaign using this technique. However, these tools are only capable of injecting faults into a single core of an MPSoC, even though they can simulate a VM with multiple processor cores.

Fault injection using system simulation can combine many of the advantages of the aforementioned techniques. In prior research, actual MPSoC architectures were simulated using SystemC to demonstrate architectural features. This could also be used as compromise between the level of detail and extreme computational cost of fault injection using netlist simulation, and limitations of fault-injection using system emulation when targeting an multicore system. Until recently, however, modeling and implementation of an MPSoC capable of running real software software using SystemC required an excessive amount of development effort. With the emergence of modern architecture description languages such as ArchC and in combination with the emergence of more open processor core designs such as RISC-V, the development effort necessary to do so has been reduced to a more realistic level. We therefore conducted further testing of our implementation for with an ArchC implemented SystemC model

our our MPSoC to validate our lockstep in a true multi-core environment without the constraints of system-emulation-based fault injection.

8.6 Test Campaign Setup

Having determined a fault-injection techniques and knowing what kind of faults need to be injected, we must prepare a suitable test environment to properly To achieve systematic test coverage, manual fault injection or injection relying upon manual binary introspection are unsuitable. Instead, an automated campaign setup is needed. In this environment, we can then subject our lockstep implementation to fault injection in bulk. This process can then be paralleled to achieve the desired test coverage. In this section, we therefore describe how such a test setup can be realized with limited development manpower, and pre-existing standard software based on our own setup.

Our fault injection toolchain performs the following steps implemented as a set of python scripts:

1. Result harvesting: obtain the victim application's process state, results and correct lockstep checksums for each payload application. We run the emulation without fault injection and tracing, outputting the application and OS state for comparison during later steps. This allows us to e.g., include additional debug output or otherwise alter the victim-binary's code for our golden run. Thereby, we can obtain a correct victim OS state without distorting the actual golden-run.

2. Fault-free simulation: we execute a golden run of our target implementation and generate traces for executed instructions, register and memory access with the actual binary used for fault injection.

3. Filter the traces to constrain fault injection to application relevant code and data (e.g., omitting platform bring-up, OS, and shutdown code).

4. Remove duplicates, and annotate each trace-entry with the number of occurrence in the trace, generating the test-campaign input data.

5. For each address and occurrence, we generate a fault definition based on a template and launch an instance of our fault injection tool.

6. Based on a comparison to the known-correct results obtained in the first step, we determine the impact of the injected fault (e.g., OS crash, incorrect checksum, SDC, etc.) and log the result to an sqlite[3] database. Besides collecting and interpreting the results of a fault injection run, we also retain compartment state information to enable manual analysis in the future if necessary. This includes a compartment's human readable output to each compartments' serial port, CPU and qemu processor context dumps, as well as the logs generated by FIES during the fault injection, as well as its exit code.

Steps 1-3 are executed once at the beginning of a test campaign, whereas steps 4 and 5 are computationally comparably expensive but can be parallelized. As sqlite stores a run's database in an individual file, result databases from different systems

[3]Any database would work, but we want to keep the results portable so they can be combined later one.

can be merged, and each test record includes information about the precise injected fault.

Long fault injection campaigns place considerable strain on host a computer's filesystem. While running our test campaigns, we discovered that this can cause induce significant wear in SSD-based storage device. When replicating this setup, the avid reader may wish to instead conduct fault injection fully in memory to avoid damage the host computer's SSD. This can be achieved by running experiments in a ramdisk, e.g., by mounting tmpfs on the experiment directory.

8.7 Executing a Test Campaign

We conducted our fault-injection campaign using both system emulation with the FIES fault injection framework and through SystemC simulation with a 3-core MPSoC model.

8.7.1 Tool Selection

The available emulation-based FI tools which were available at the time of initiating validation for our lockstep were not functionally equivalent. They differ regarding the target environment, test setup and intended test subject scope, and the way in which they inject faults. The FAIL-framework utilizes a powerful C++ based test controller for thoroughly analyzing small binaries in a fully automated test campaign. While the test itself is therefore fully automatic, the development of a test-specific controller application requires deep knowledge of victim binary intrinsics and program structure. This information is target binary and concept dependent, and is hardcoded within a dedicated experiment controller binary [4]. The development of FAIL is mainly focused on the Intel platform. ARM support less mature and only available through GEM5 [314] or through into hard silicon, neither of which are viable for our purposes as discussed earlier.

FIES by Höller et al. [313] was developed specifically to validate ARM-based COTS-based critical systems. It is based upon the much faster and more mature virtual machine monitor QEMU, thereby supporting a broad variety of SoCs and virtual hardware. However, there is no not support for conducting fully automated test campaigns, but allows rule-based and systematic fault injection into opaque binaries during each run. Its fault injection engine utilizes a fault library which can be generated automatically using compiler-toolchain functionality and instruction and memory access traces. We can therefore efficiently test a full OS including its kernel, without requiring a test monitor with knowledge about application intrinsics. The test campaign described in the remainder of this section is thus carried out using an automated test toolchain incorporating FIES.

FIES does not guarantee timing and strict time determinism. Hence, when validating more timing-sensitive algorithms however, special care must be taken to assure the golden run and fault injection runs are equivalent [312,313]. However, our lockstep implementation also does not require strict time determinism during simulation runs. It only requires that a comparable level of work is conducted between checkpoints.

In the process of developing our test toolchain, we extended FIES' functionality to better support different tracing techniques and added functional improvements.

[4] See the src/experiments directory at https://github.com/danceos/fail

Initially, this began as bugfixing effort, but over the course of several months, we in practice rewrote most fault-injection triggering related code, as well as a major part of FIES' state machine. FIES originally was also based on QEMU 1.17, and therefore we rebased the heavily modified FIES code to QEMU-git 2.12 (qemu-head in December 2017). We also added support for the THUMB2 instruction set as FIES originally only could inject faults into ARM instructions, and only used those as fault-triggers, as most common software use both ARM and THUMB2 assembly intermixed. At this point, we had rewritten major parts of FIES, and we therefore made not just patches for FIES available, but released the entire tool as "FIESer – FIES Extended and Reworked" to the public. It is source code is available at `https://fieser.dependable.space` and on `https://github.com/dependableDOTspace/FIESer`.

To realized fault injection via SystemC, we first had to develop a suitable MPSoC implementation. Most SystemC MPSoC models described in literature, however, at close inspection turn out to only be capable of running brief instruction sequences to validate parts of, e.g., an instruction set, or a specific low-level functionality of an MPSoC. Hence, they are incapable and often not even intended to run run actual application software, which we require to test our lockstep implementation. This is no problem for emulation-based fault injection, where only the high-level behavior of a system is emulation, but challenging for more close-to-hardware SystemC-based simulation. Hence, as part of an ongoing international inter-university collaboration, we implemented a true multi-core model of our MPSoC. We implemented this MPSoC through the use of the open RISC-V platform, for which preexisting ArchC models were available. Each processor core existed in its own compartment with dedicated I/O capabilities as described in Chapter 4, and have access to a shared memory segment used to exchange and compare lockstep state information.

8.7.2 Target Implementation and Payload

When conducting fault injection it may seem obvious that these tests should be conducted against a realistic target implementation. However, this is only feasible if the right tools were chosen as described in the previous sections. A majority of publications today does not do so, and often researchers seemingly try to force-use unsuitable fault injection tools to validate their implementation. In the remainder of this section, we thus describe the fault injection target implementation of our lockstep, and outline how and why it is representative for our purposes.

A simplified function flow graph of our lockstep implementation is depicted in Figure 58 for reference, and in full described in Chapter 4. As payload application, we utilized two applications:

- The ESA Next Generation DSP benchmark[5] run as POSIX threads within RTEMS. This is a space-industry standard benchmark application used to measure and compare system performance.
- An application alike the NASA/James Webb Space Telescope Mid-Infrared Instrument readout software[6] [219].

While this choice represents satellite computing workloads reasonably well, test campaigns for other application should utilize representative software. If no specific target

[5]Source code publicly available at https://essr.esa.int
[6]See https://github.com/spacetelescope

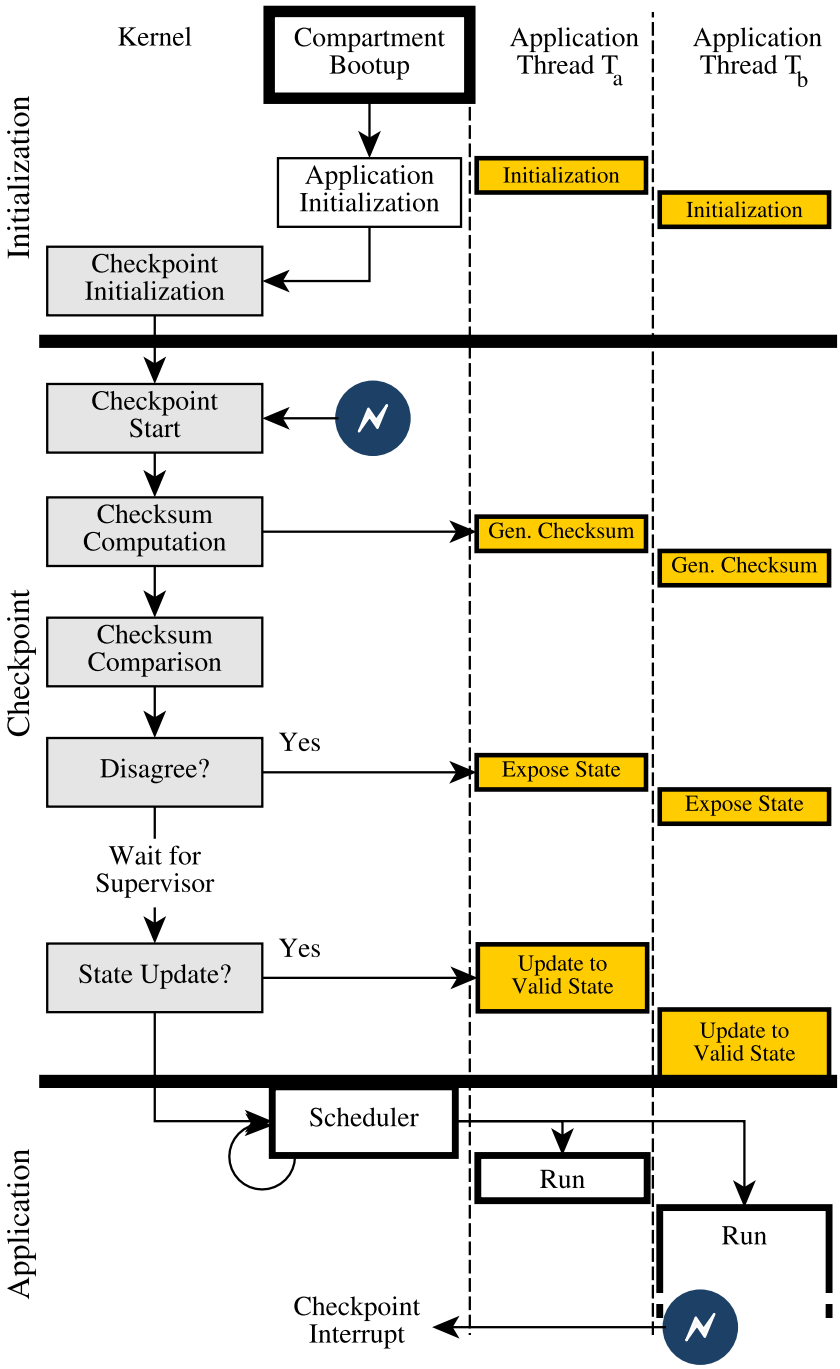


Figure 58: The execution cycle of our coarse-grain lockstep implementation on a compartment. Payload application callbacks are depicted in yellow, checkpoint trigger timers in blue. Faults are injected after initialization.

application code is available, synthetic algorithm suites such as the SPEC performance tests[7] can be utilized at a loss of realism due to the limited scope and low complexity.

Our fault injection experiments using system emulation were conducted against an implementation of our approach in RTEMS 4.11.2 using the ARMv7a-Zynq board-support-package, which closely resembles the compartments of our MPSoC. RTEMS is a real-time OS running bare-metal, and is used in a broad variety of space applications. We chose not to utilize the Linux kernel for our fault injection experiments to maximize the level of control over our experiment and reduce the test time overhead. We cross-compiled the kernel image from Fedora 28 x86_64 with standard compile flags (`-marm -mfpu=neon -mfloat-abi=hard -O2`) in RTEMS GCC 4.9.3. Note that RTEMS does not utilize privilege separation, enforces no separate between a userland and kernel code, and has no virtual memory support. All these features would make faults more easily detectable and the OS as a whole more robust. Hence, faults in application code can directly interfere with kernel data structures. However, the absence of such functionality is representative for today's space computing even aboard larger spacecraft.

For SystemC-based fault injection, the model used was implemented using SystemC version 2.3.1 and ArchC 2.4.1 with custom patches to enable fault injection. Instruction instrumentation was realized using nightly builds of AspectC++, as the latest released version of AspectC++ is outdated[8]. The excessive amount of compute time necessary for fault injection into the MPSoC prevented the re-use of the same lockstep implementation used as for emulation-based fault injection [315]. Initially, we attempted to re-use the same test application setup we developed for emulation-based fault injection, but a single fault-injection run with this application in our ArchC model on just one processor core would have taken more than 8 hours. Therefore, instead of running a full RTEMS implementation of our lockstep, we constrained our implementation to run bare-metal code without thread-management, interrupts, and timers. This implementation was cross-compiled using the RISC-V toolchain released and maintained by the Andes Technology Corporation at `https://github.com/andestech/riscv-llvm-toolchain` against the `ilp32` ABI of the `rv32ima` RISC-V architecture variant. At the time of writing and conducting these fault injection experiments, the toolchain uses GCC 7.1.1. Naturally, this curtails the fault tolerance capabilities this implementation can achieve, but it allows the test time to be reduced to approximately 1 minute of real-time per injected fault.

8.7.3 Test Space and Target Components

We prepare a set of fault definition templates, which our fault injection toolchain combines with information from the previously generated traces. These templates define the test-space of our campaign. However, choosing the right test-space for testing an OS-scale fault tolerance measure is non-trivial. A test-space as described in literature [316] as ideal for testing software in practice is usually not achievable [317], and stands in stark contrast to the best practices in system-level testing in industry [318, 319]. Even fault injection with state-of-the-art tools requires a carefully chosen compromise between realism and test-coverage to avoid runaway test-times and high cost.

[7] see `https://www.spec.org/cpu`

[8] At the time of writing AspectC++'s latest released 2.2 is more than 2 years out of date and its functionality is no longer comparable to those of the nightly development builds

Transient Fault Injection

Transients are injected as bit-flips and new-value errors into registers and the processor pipeline using the program counter as trigger. Simple time triggered injection is insufficient, as the available tools do not assure clock-cycle accurate timing. For instructions which are visited more than once, we trigger faults after the n-th occurrence, which is enabled by an extension of the FIES framework's fault definition language. Our SystemC implementation is designed to allow fault injection also with cycle accuracy in different parts of the processor pipeline, though we consider this functionality to be too unreliable to use it for fault-injection yet. With FIES, we inject faults also into memory access operations based on physical memory addresses. This allows us to approximate the effect of faults in caches and main memory, as well as faults in buffers. To better simulate non-correctable upsets in ECC words and faults in the address logic, we can also directly replace accessed data or replace the address of the operation.

Permanent Fault Injection

Permanent faults should be injected into accessed main memory and devices address space. However, they should not be injected into general purpose registers, special registers, and the CPU pipeline provided little added value for testing software-implemented fault tolerance measures. This is due to the fact that the effects of faults in these components are fatal at the latest after a few clock cycles. Hence, they will interrupt operation of a processor core, and this can be detected through our lockstep by other compartments in the MPSoC, as well as by the supervisor. While it is important to not ignore parts of our fault model, testing with faults with a predetermined and known result would needlessly inflate the test space and time.

Functional Interrupts and Intermittent Faults

Radiation may also cause fault-effects which are neither transient nor permanent. To simulate SEFIs with FIES, FIES' fault types of periodic and intermittent faults can be used. For these, fault effects persist for a user-described period of time and are resolved by the injection framework afterwards.

In our tests, we chose 100ns as fault-duration for SEFIs, the period-equivalent to 10 clock cycles at 100MHz, the frequency emulated by QEMU for the Zynq MPSoC. This represents the interruption effect and the reset-induced outage of specific circuit groups due to SEFIs reasonably well. However, we are not aware of radiation-test data further analyzing the actual timing and detailed interruption behavior SEFIs in processor logic and FPGA fabric.

Fault Placement during Execution

After executing bring-up code and OS initialization, our victim binary executes payload software for 3 lockstep cycles on FIES and 5 lockstep cycles on ArchC, and then terminates. The test sequence is depicted in Figure 59, and faults are injected during the first checkpoint cycle or frame of execution. This allows faults to propagate within the system, to corrupt the application state, without requiring excessive experiment time. During the first checkpoint executed after fault injection, corruption of the application state should be recovered. Upon reaching the second checkpoint after fault

injection, the application state should have fully recovered and thereby the system state should match the golden run's results. This allows us to verify the full FDIR cycle from fault injection to recovery.

For emulation-based fault injection we chose a frame time of 2 seconds as interval between checkpoints. This is a reasonable choice for operation in LEO when passing through increased radiation zones such as the South Atlantic Anomaly, based on radiation-testing data for Ultrascale [143, 297] and Ultrascale+ FPGAs [298]. For SystemC-based fault-injection, checkpoints are executed after each frame the `NIR HAWAII-2RG` algorithm has been processed.

For our RTEMS implementation, a golden run takes approximately 7 seconds of guest-virtual time, which on our test system is equivalent to approximately 30 seconds of host-time. In case the experiment does not terminate in time, e.g., due to control flow corruption, the experiment is terminated by the toolchain after 45 seconds (allowing one additional checkpoint to be processed). FIES can also be configured to end an injection run after executing given number of instructions (e.g., 10 times the number of instructions executed in the golden run). We are not relying upon this functionality as the value has to be hardcoded in FIES.

For our MPSoC, the execution time of a golden run for generating traces does not differ significantly from a run where faults are injected. However, even after much optimization a single run takes approximately 45 minutes of real-time on Core-i7 8700K system. We therefore reduced the NIR detector frame size from 2048x2048 pixels to 32x32 pixels, which then reduced the overall runtime to between 1 minute and 20 seconds, depending on the host system's performance. Naturally, this changes the ratio between code and data due to the much reduced size of the data structures used, but does not change the overall program structure of the executed application and the lockstep. As we already established an upper bound for the performance cost of our lockstep in Chapter 4, we consider this constraint acceptable.

After fault injection has terminated, we analyze if our lockstep could detect the effects induced by the injected fault (if any), and if they could be resolved through a

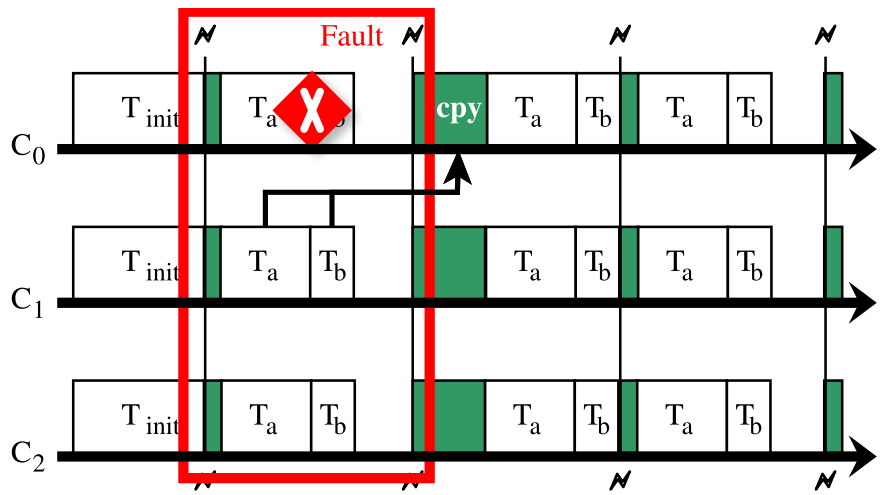


Figure 59: The experiment sequence and fault placement for a compartment. Fault are injected during the red-outlined time period on processor compartment C_0.

state update from another compartment. To reduce the test space, we do not inject faults into platform code, bring-up, an shutdown-related code.

Limitations

We chose the length of a fault injection run to allow our victim binary to exhibit the entire FDIR circle. As we are testing a full OS instead of just code snipplets or brief instruction sequences, this is necessary. In contrast to related work, the runtime of our fault injection campaign is therefore already excessively long, e.g., extended by more than an order of magnitude as compared to Amarnath et al. [305]. However, such a brief run still does not allow dormant or latent faults to be discovered, e.g., such affecting OS data structures and logic resulting time-delayed regressions. Only certain fault will produce immediate effects, and it is infeasible to extend our target binary's runtime even further. Therefore, it is impossible to observe or even determine if a fault results in no effect, silent data corruption, or time-delayed effects. The time allotted to each fault injection run therefore is a direct trade-off between achieving sufficient test-coverage to judge the fault-detection capacity of our lockstep, and to observe long-term effects.

In our ArchC system model, simulate RISC-V processor cores. This instruction set offers a large quantity of general purpose registers, which would inflate the test space as compared to our FIES ARM target (30 general-purpose registers as compared to 12 on the ARM platform). Therefore, we conduct an Architectural Vulnerability Factor (AVF) analysis [320] for the traces used in our fault injection campaign. AVF allows us to reduce the test space to avoid injecting faults into locations which would subsequently be overwritten, reducing masked faults and the overall test space. However, as discussed further by Maniaktakos et al. in [321] AVF overestimates vulnerability by more than 70%, and can not properly model the impact of multi-bit upsets in semiconductors manufactured in technology nodes less than 65nm feature size. In our campaign, we utilize AVF to constrain potential fault location (register address), but not to determine which bits are vulnerable and instead inject faults in each bit of a 32-bit word.

Our need for systematic testing also induces another limitation: Being constrained to running only a few lockstep cycles after fault injection, we also can not making more long-term observations regarding fault recovery. The fault recovery potential of coarse-grain lockstep also are heavily influenced by the protected applications and OS structure. Any fault-recovery statistics obtained for very short term fault recovery thus would be unreliable. Instead, this information should better be obtained through system-level testing with actual on-board data handling software on a prototype.

It would be feasible to inject faults in QEMU's emulated virtual hardware and into the infrastructure of our SystemC-MPSoC model. This would allow faults to be injection more realistically for each emulated or simulated device and MPSoC component. However, this is not supported in FIES and our SystemC-MPSoC model today. To our understanding FIES was also never developed with such functionality in mind. Hence, while technically possible, fault injection in qemu virtual devices would require considerable development effort even for only one set of virtual devices relevant for validating our target architecture. Due to a lack of tools, we can instead approximate the practical effects of radiation by injecting faults during access to memories and device address space, as well as into the CPSR on FIES.

For our SystemC-MPSoC, there is no structural limitation to fault injection as with

FIES, and in the coming months we plan to expand the fault-injection capabilities of this model. At this point in time, have begun adding cycle accurate fault injection support, instead of instruction-based fault injection which is possible with FIES and our ArchC model today. Once this has been accomplished, we plan to inject faults also into the MPSoC's interconnect, as well as CPU peripherals and interfaces that are part of a compartment.

8.8 Results & Interpretation

To test our toolchain and verify its correct functionality, we conducted manual fault injection into specific application structures using FIES. We injected such faults into interesting data and logic which could cause an incorrect application state, or could otherwise alter the run-time behavior of a compartment. This allows us to analyze the practical behavior of our lockstep under faults, and enabled us to directly compare the impact of a fault in a specific location when injected as transient, permanent and intermittent faults. Table 5 shows the behavior of our lockstep under faults, and we subsequently expanded our fault injection campaign in the described automatized way with FIES and our ArchC model. In Table 6, we provide statistics observed when conducting fault-injection with FIES and ArchC.

In payload-application code, a majority of the injected transient faults resulted in a corruption to the payload applications' state. With less than 20% of all faults, the application of the entire OS crashed or terminated prematurely (compartment resets were treated as early termination). Faults affecting the lockstep mechanisms (e.g., resulting in false comparison or incorrectly generated checksums from correct data) were rare due to the minimal time spent executing lockstep mechanisms, as its low code and data footprint.

A comparable share of bit-flips with permanent effects resulted in a corrupted thread state and thus checksum-comparison mismatch, as was the case with transient faults. However, this number alone is misleading, as the amount of masked upsets without noticeable effects plummeted to just 19%, while the share of thread- or OS-crashes increased. Therefore, we can deduct that a number of faults which due to transient faults would have resulted in just thread state corruption, now instead result

	Detection by		Recovery	Recovery Method	
Result	Victim	System	Trigger	State Update	Reboot
Corrupted State	yes	yes	lockstep	yes	yes
Thread Crash	yes	timing only	lockstep	yes	yes
Lockstep Failure	no	yes	supervisor	no	yes
Crash/Hangup	no	yes	victim core	no	yes
No Effect/SDC	no	no	supervisor	sometimes	yes

Table 5: Behavior of our RTOS implementation under faults, considering fault detection at the system level, as well when considering victim-processor core itself. Notice that our lockstep implementation can not detect silent data corruption with no immediate impact on the thread state.

	Effect by Injected Fault Type			
Result	FIES Transient	ArchC Transient	Permanent	Intermittent
Corrupted State	49%	32%	44%	53%
Thread Crash	8%	-	17%	10%
Lockstep Failure	1%	1%	2%	1%
Crash/Hangup	10%	14%	18%	15%
No Effect/SDC	32%	54%	19%	21%

Table 6: Fault injection experiment results to date with FIES and ArchC divided into transient, permanent, and intermittent faults. A share of all masked faults will cause silent data corruption, which can have long-term effects on OS data structures. These could be detected through erasure coding, while memory protection and virtual memory would allow us to detect misdirected memory access caused by faults. Neither measures is in place in our proof-of-concept.

in crashes. The total amount of detected faults in turn was increased again by faults which were previously masked. Intermittent faults have a similar effects to permanent ones, though with slightly fewer crashes and more faults affecting only the payload application.

Our coarse grain lockstep implementation contributed fault-detection to the system, whereas the state synchronization functionality serves to reduce the amount of reboots needed to restore the state of each compartment. In practice, its fault-detection strength depends on both the frequency at which checkpoints are execute (frame-time) and the likelihood that faults can be covered and corrected. Hence, we analyzed how rapidly a compartment itself can detect faults in Figure 60.

The fault injection campaign shows that there is indeed a measurable difference in behavior between transient and permanent faults, and between target applications of different complexity. As expected, permanent faults are more likely detectable than transients, due to their increased severity. However, we also expected permanent faults to be easier detectable by a compartment than SEFIs (see Figure 60a). This was not the case. The increased likelihood of permanent faults resulting in crashes and the higher percentage of non-fatal state corruption faults due to SEFIs made fault detection within the affected compartment more likely for SEFIs. For permanent faults a larger percentage of faults results in a crash, which can no longer be detected by the affected compartment. These results underline the importance of conducting validation not only using transient faults, but also with permanent and intermittent faults.

The effects of a fault will be detected through majority decision by the rest of the system. The fault detection rate increases sharply, as the MPSoC as a whole can also detect crashes of an entire compartment or lockstep mechanism failure, as shown in Figure 60b. In Figure 61, we therefore provide a direct comparison between self detection and majority decision for transients, permanent and intermittent faults. While the results for transient faults again match our expectations, for permanent faults and SEFIs, the initial fault detection capability for the full MPSoC even with only a single executed checkpoint is drastically better than for self-detection. Here, a

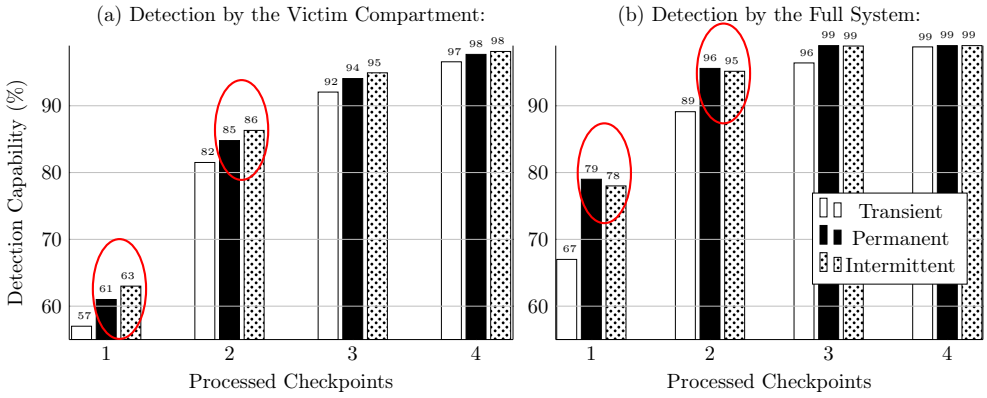


Figure 60: Payload application and state corrupting fault detection chance of a single compartment for different fault types after a given number of execute checkpoints. Notice that intermittent faults are more likely to be detected than permanent faults by the affected compartment itself, which is counter intuitive. This is due to the increased percentage of faults that are fatal for a compartment, and the system as a whole can detect permanent faults with higher likelihood.

fault detection chance of near 79% and 78% during the first checkpoints also implies a near certain fault detection likelihood during the second checkpoint; see Figure 61b and c. In contrast, for self detection, faults can be detected after with 57%, 61% and 63% during the first checkpoint after fault occurrence and near certain detection only being achieved after three checkpoints.

When designing our lockstep concept, we considered fluctuations in compartments thread assignment within the MPSoC to be critical. This is caused by crashes and reboots of individual compartments. Worst-case benchmark results showed that frequent crashes of compartments could degrade performance of the system by between 9% and 26% for high checkpoint frequencies and brief frame times. Based on our experiments, we find comparably few faults, between 11% and 20%, cause crashes and lockstep-failures. Even under the (unrealistic) assumptions that faults were to

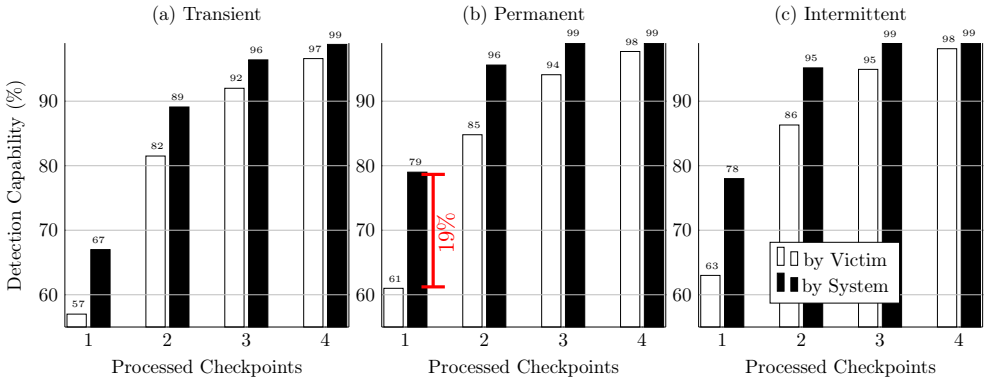


Figure 61: Comparison of the fault detection capabilities of an individual compartment and the by MPSoC through majority decision. The full system can also detect a crash of the OS instance running on a compartment, and malfunctions in the lockstep logic.

Effect	Number of Faults	%	**Thereof:**	Immediate Recovery	Lockstep Timeout	Reboot Required
Non-Masked	47526	46%		22004	10915	14607
				46%	23%	31%
Masked	57379	54%				
All	104905					

Table 7: Fault Recovery statistics for SystemC fault injection.

occur in each checkpoint period, many faults could still be resolved through a state update and do not require a reboot. Hence, our lockstep implementation can provides the necessary degree of voter stability to making application reassignments between compartments rare.

A majority of faults that resulted in no observable effect on our implementation may indeed be masked and require no measures to be taken, as they may have no impact on the application state [322]. This is a limitation of our fault injection toolchain, as faults are also injected into registers and memory which may be overwritten by subsequent instructions, or faults that cause self-masking control flow deviations. Such situations occur e.g., due to faults in branch or comparison instructions triggering the same iteration of a loop more than once. They have no practical impact on the application state while, and also cause only minor timing deviations which do not impact the work conducted until to the next checkpoint.

8.9 ArchC MPSoC vs. FIES Result Comparison

Comparing our transient results between ArchC and FIES, we notice that the results are mostly comparable. The share of faults without noticeable effect are increased by approximately 20%, which seems reasonable considering the different lockstep implementations tested: part of this difference can be attributed to the vulnerability overestimation remaining due to limitations of our AVF analysis. Furthermore, the lockstep implementation on ArchC can not exploit the powerful exception handling function available in a proper operating system implementation, as we are here running the test implementation bare-metal. Instead, our FIES implementation exists as part of RTEMS, which allows more precise fault analysis, and overall reduces the chance that a fault will crash the entire OS instead of just the test application thread.

To allow better comparison of the fault effect ratios between system emulation and SystemC fault injection, we have to normalize the results obtained with both techniques. To do so, we apply normalization to the 54% of masked faults to all effect ratios obtained with FIES, where we encountered just 32% masked faults. A comparison between normalized FIES fault effect ratios and ArchC is depicted in Table 8. As depicted, after normalizing the result data, we receive almost identical fault effect ratios with both techniques, with our RTOS implementation showing 6% higher data corruption likelihood than our bare-metal implementation. In our ArchC lockstep implementation, 15% of all faults cause a crash or hangup effect, while in our RTOS implementation 14% of cause such an effect. As our FIES implementation utilizes

threading 6.5% of all crashes remain isolated to the crashed application software, or the lockstep, while our ArchC implementation knows no such separation. In practice, this shows that the additional OS and application isolation functionality implemented within a modern OS also has a positive impact on suitability. In turn, the increased amount of code an data required for an OS-scale implementation also shows that the ratio of faults causing data corruption is slightly higher than when running the same application bare-metal.

In Figure 7, we provide fault effect and recovery statistics obtained from our ArchC MPSoC model. After observing 105905 fault injection runs into our ArchC MPSoC model using AVF-filtered golden run traces, we can observe that: in 46% of cases a corrupted thread-state could immediately be recovered through a state update, required no reboot of the faulty MPSoC core. In further 23% of cases, faults could have been recovered if the lockstep had allowed for more wait time during checkpoint voting, which was severely constrained in our test campaign to assure sufficient test coverage. Only in 31% of cases, fault resolution was unsuccessful, requiring a reboot of the affected processor core. Overall, these statistics are very positive, considering especially the much reduced fault-recovery potential that a bare-metal lockstep implementation has as compared to a full OS implementation.

Considering the different scale and detection capabilities of the two different lockstep implementations analyzed, this different is in line with our expectations: The target implementation we used for ArchC fault injection does not utilize a threaded scheduler, and therefore thread-management and scheduling is eliminated as potential failure source. Overall, injected faults in a threaded RTOS implementation should locally also impact OS-level control logic, and infrastructure data structures, and induce secondary fault effects there. At the same time, the this also means that faults which in an RTOS implementation caused a thread to crash, now would only cause data corruption in the protected application.

8.10 Comparison to Literature

To place these results in context with results from other lockstep concepts, we sought to compare our results to literature. Unfortunately, few coarse-grain lockstep concepts have been implemented in practice and tested using means beyond modeling. At the time of writing, we are aware of only one publicly released validation report by Dobel

	FIES			
	Ref.	@ 54% SDC	ArchC	Δ
Corrupted State	49%	38.22%	31.72%	-6.5%
Thread Crash	8%	6.24%	0%	-6.24%
Lockstep Failure	1%	1%	1%	0%
Crash/Hangup	10%	7.8%	14.54%	+7.66%
			Δ Total	**5.08%**

Table 8: Transient fault effect comparison between system emulation and SystemC fault injection, normalized to equivalent SDC ratios.

et al. [199] considering practical fault injection with real software and faults, instead of statistical estimation.

When directly comparing our results to Dobel et al.'s *transient* fault injection report [199], the share of faults causing application, thread, and OS crashes with our approach is noticably increased. For transient faults, this can at least in part be explained with the different capabilities of Dobel et al.'s proposed lockstep mechanisms. In their contribution, lockstep is facilitated through application intrusive function call hooking. Thereby, Dobel et al.'s lockstep can offer more fine-grained protection than our approach. However, it also require considerable code, deep and non-portable changes in the target OS, has a high performance overhead, and constrains the target OS and application structure. The measured detection differences are consistent across all effect categories: we measure a higher amount of masked faults, a decreased amount of detected state deviations, and an increased amount of crashes with our approach.

Dobel et al. consider their fault injection measurements overly optimistic, as they utilized payload applications *"of little complexity (leading to few potential candidates for fault injection)"* [199]. Their validation and lockstep implementation is constrained to handling transient faults, while SEFIs or permanent effects are not covered as these faults were injected into a user-land application of their approach through a debugger. Dobel et al. assume the OS, system libraries, and kernel to be fault-free, while we instead inject faults into a full OS including POSIX libraries with payload applications. In light of this bias, we consider our results are in line with Dobel et al.'s, and our lockstep implementation to function as desired.

The results we obtained with SystemC fault injection into our ArchC MPSoC confirms this further. There, we can in practice reproduce exactly this same scenario between the two lockstep implementations we have been utilizing for testing with FIES and for our ArchC MPSoC-model. The lockstep implementation there is overall simpler, has fewer calls to critical infrastructure functionality that could break, and therefore offers less overall failure potential than our full RTEMS-implementation. Furthermore, in this MPSoC we utilize RISC-V processor cores with a much simpler and less powerful instruction set than that offered by a full Cortex-A processor core implementing the ARMv7a instruction set, which not only supports one instruction set, but uses two instruction sets in combination (ARM and THUMB).

8.11 Discussions

Fault injection today can be conducted for different reasons, such as to detect security vulnerabilities in software, memory leaks, or to assure test coverage when testing for functional correctness. However, fault injection for validating the correction functionality of a fault-detection and lockstep technique is very different from, e.g., fault injection conducted for security purposes. Applying the same assumptions or test tools to both, while attractive, does not result allow for proper validation. The used fault injection techniques, target implementations, and payload software will influence the obtained results. Validation using an overly simplistic target implementation will bias the results obtained. Comparing our results to Dobel et al.'s underlines that it is important to conduct fault injection into a realistic implementation with non-trivial payload software, but also that more lockstep concepts must be validated.

Our coarse-grain lockstep can detect faults resulting in a crash or in corruption of

the thread state. However, it is unable to detect silent data corruption and latent faults in OS data structures and code. To better handle this, a compartment's checkpoint handler could generate a checksum for certain critical kernel data structures. However, the scope to which this is possible is limited and the computational cost may be high. It would be practically impossible to do this for a larger OS or, e.g., the Linux kernel.

Velasco et al. propose in [323] to apply erasure coding for critical OS data structures in software. The proposed concept is similar to code signing, and today widely used for tamper-proving of embedded devices and e.g., for secure boot. The availability of this functionality would allow our lockstep to also detect silent data corruption in rarely accessed OS structures and device drivers code and data.

When experimenting with different compiler flags, we found that faults injected in equivalent code segments of differently compiled binaries could result in varying fault effects. We determined through introspection of the relevant target binary parts, that the changed behavior was caused due to specific compiler flags. Especially loop unrolling (GCC's `-funroll-loops` flag) had a particularly positive effect when injecting permanent and intermittent faults. In practice then compiler then flattens the program structure, duplicating code segments instead of executing the same segment multiple times within a loop. Serrano Cases et al. in [324, 325] as well as Lins et al. in [326] have begun to explore these effects for improving reliability, but otherwise industry and literature today seem oblivious on this issue. Designers of software-FT measures in the future should consider the impact of a broad variety of behavior-altering flags and toolchain settings supported by modern compiler suites, as these have a direct impact on the utilized FT mechanisms as well as validation.

FIES originally offered no support for the THUMB instruction set. However, most OS kernels, many device drivers, and even standard library functions mix THUMB and ARM instructions. Therefore, we had to implement support for the THUMB and THUMB2 instruction sets for FIES, to assure consistent tracing and fault injection results.

A jump between instruction sets without compiler-interwork would yield an undefined instruction exception, as the opcode-encoding for ARM and THUMB instructions differs. This effectively prevents undetected, incorrect jumps in ARM/THUMB interwoven code segments. We argue that instruction set mixing could be exploited to improve fault detection. Critical code segments could intentionally be assembled with strong instruction-set interweaving to assure that an incorrect jump immediately results in an exception instead of silent data corruption or control-flow deviations. For C-code, this can be achieved per function using target attributes and prefixes, or more fine-grained using preprocessor definitions and pragma. This would reduce the likelihood of silent data corruption and introduce a level software diversity through compiler instrumentation or scripted, automated code transformation [327].

When designing our coarse grain lockstep measure, we were aware of two ways of inducing checkpoints: through timers on each compartment and externally through interrupts. If timers are used, checkpoints are triggered independently on each compartment. Interrupt induced checkpoints are centrally triggered by the off-chip supervisor, creating a potential single point of failure. At design time, we therefore considered timer driven lockstep to be better, as it avoids a central authority inducing checkpoints in favor of decentralized triggers. However, our fault injection campaign showed that interrupt induced checkpoints are considerably simpler. The timer-handling related logic requires more code and increases the OS state, and thus also more prone to faults

than a simple interrupt handler. Hence, in future work we decided to use interrupt driven checkpoints instead of timed checkpoints.

8.12 Conclusions

In this chapter, we presented an automated fault injection toolchain, and validation results of the software-implemented fault tolerance (FT) concept described in Chapter 4. Few software-implemented FT concepts proposed today have been validated, and therefore this chapter also serves as practical guide for fellow research, to make proper testing of fault tolerance techniques a less challenging and time consuming task. Today, a broad variety of fault injection techniques and tools are available for finding bugs or security vulnerabilities, to assure logical correctness of a concept, or to validate FT concepts. Validation of software-implemented FT concepts requires a realistic implementation, and in-depth knowledge on the tested mechanisms and tools. Hence, not all tools and techniques are suitable for all purposes, and validating FT concepts in the same way as fault injection is conducted for, e.g., software security purposes, does not work.

Proper validation thus is non-trivial, is time consuming and requires considerable research. In consequence, developers of coarse-grain lockstep concepts often forego the practical concept implementation and validation, resorting instead to modeling. Practical validation, however, is a prerequisite to even consider a concept for application in mission critical systems, which then can be subjected to system-level validation and prototype development. This has resulted in a large gap between academic theory and practical application, with researchers proposing powerful concepts but industrial users disregarding them out of hand due to a perceived lack of maturity and time pressure due deliver results.

The lockstep implementation validated in this publication and is the key element of a hardware-software-hybrid system architecture which combines different FT measures across the embedded stack within an FPGA-based MPSoC design. Validation of such concepts has to be conducted differently than for traditional hardware-voting based systems, and requires systematic fault injection. Hence, we developed an automated fault injection toolchain, which enables systematical testing using system emulation to validate the complete FDIR cycle. To place our results into context, we compared them to literature and discuss lessons learned and knowledge obtained throughout our fault injection campaign beyond analyzing raw numbers. The overall results of our fault injection campaign are positive and the thread-level coarse grain lockstep's performance meets our requirements.

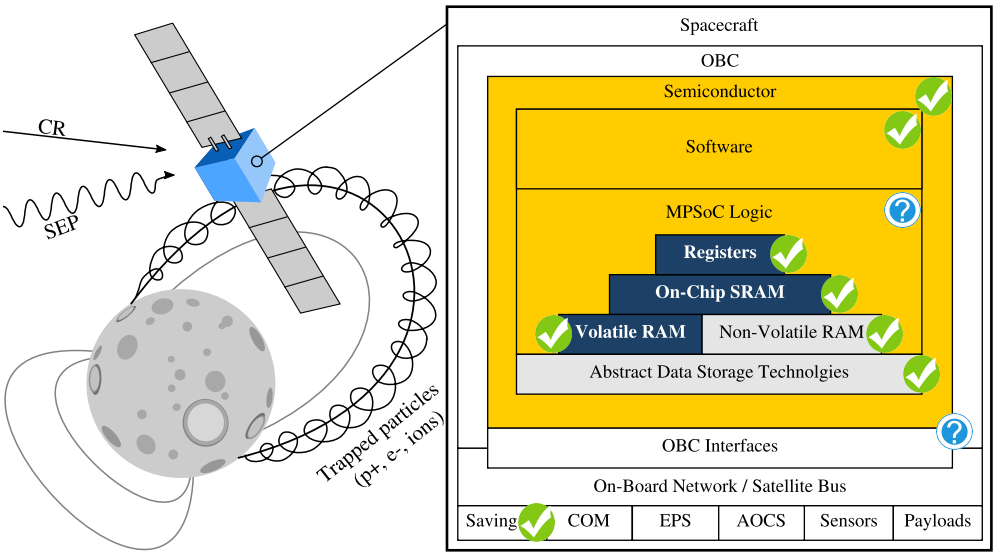
As the other parts of our architecture have been verified separately in related work, our test campaign represent the final step in validating our current development-board based proof-of-concept. In practice, through this testing, we have exhausted all technically feasible testing techniques for software that are possible today to validate a fault tolerance measure of the scale of our lockstep. The positive outcome of our test enables us to now produce a prototype OBC implementation, which then allows us to then subject it to laser fault injection, radiation testing, and trials on-orbit. Systematic validation of our coarse-grain lockstep implementation is therefore an intermediate step. To further test our architecture, a prototype system must be implemented to then conduct radiation testing.

Chapter 9

Combining Hardware and Software Fault Tolerance

High-Level System Design

In this chapter, we describe in detail the topology of our multiprocessor System-on-Chip (MPSoC) to address RQ6 by providing an ideal platform architecture for the lockstep described in Chapter 4. We show how it can be assembled in its entirely from well tested COTS components using commodity processor cores and library IP. The resulting MPSoC is the result of a true hardware-software co-design process, and utilizes the concepts presented in the previous chapters. It is designed as ideal platform for our architecture, where each design decision was taken to reinforce the fault tolerance properties of the system as a whole. This chapter therefore servers the final step in developing our fault-tolerant system architecture. In Chapter 10, we then present practical implementation results of this MPSoC.



9.1 Introduction

Satellite miniaturization has enabled a broad variety of scientific and commercial space missions, which previously were technically infeasible, impractical or simply uneconomical. However, due to their low reliability, nanosatellites, as well as light microsatellites, are typically not considered suitable for critical and complex multi-phased missions and high-priority science. The on-board computer (OBC) and related electronics constitute a large part of such spacecraft, and were shown to be responsible for a significant share of post-deployment failure [2]. Indeed, these components often lack even basic fault tolerance (FT) capabilities.

Due to budget, energy, mass, and volume restrictions, existing FT solutions originally developed for larger spacecraft can not be adopted. In this chapter we describe an multiprocessor System-on-Chip (MPSoC) that utilizes conventional hardware, providing FT for miniaturized satellites. The MPSoC is assembled from well tested COTS components, library logic (IP), and powerful embedded and mobile-market processor cores, yielding a non-proprietary, open architecture. Our key contribution is a fault-tolerant OBC architecture for CubeSat use that consists only of extensively validated standard parts, and can be reproduced with minimal manpower and financial resources.

9.2 Background & Related Work

Aboard nanosatellites, subsystems are controlled by just one command & data handling system, whereas aboard a larger satellite these tasks are distributed across multiple dedicated payload and subsystem computers. This implies a varying OBC workload throughout a nanosatellites mission, which traditional FT solutions only handle through over-provisioning. The MPSoC design presented in this chapter can efficiently handle faults through thread migration and partial reconfiguration. Major parts of our approach are implemented in software, allowing the OBC to deliver the desired combination of performance, robustness, functionality, or to meet a specific power budget. To enable strong FT with low-cost commodity hardware, we combine fault detection, isolation and recovery in software, FPGA configuration scrubbing with other fault detection, isolation and recovery (FDIR) measures across the embedded stack.

Nanosatellites today utilize almost exclusively COTS microcontrollers and application processors-SoCs, FPGAs, and combinations thereof [40, 237]. Due to manufacturing in fine technology nodes, and the use of extensively optimized standard IP, they offer superior efficiency and performance as compared to space-grade OBC designs. The energy threshold above which highly charged particles can induce faults (SEE – single event effects) in such components decreases, while the ratio of events inducing multi-bit upsets (MBU), and the likelihood of permanent faults, increase. To adapt such hardware-FT based concepts additional FT-circuitry is required, inflating logic size and producing diminishing returns, resulting in limited scalability and low clock frequencies [188, 190, 192]. We can observe that traditional FT-concepts applied to modern COTS hardware yield no nanosatellite compatible architectures.

While more sensitive to transient faults than ASICs [142, 143], FPGA-based Soft-SoCs have been shown to offer excellent FDIR potential for miniaturized satellites [238]. Transients in critical parts of the FPGA fabric can be scrubbed [242], while permanent faults may be compensated through reconfiguration with differently routed

configuration variants [105]. Fine-grained, non-invasive fault detection in FPGA fabric, however, is challenging, and subject of ongoing research [239, 240]. Relevant FT-concepts thus rely on error scrubbing, which has scalability limitations and cover only parts of the fabric [239, 242]. We overcome these limitations by implementing fault-detection in software through thread-replication and coarse-grain lockstep within an MPSoC using weakly coupled cores.

Tiled architectures [246,328] are often used for well paralellizable applications with many low-performance processor cores. Among others, [329] and [328] showed that this topology can also be exploited to achieve FT for image processing applications with a very specific structure. We combine a compartmentalized topology with a coarse-grained lockstep described in Chapter 4, enabling FDIR without constraining the application type or system architecture. Thus, the architecture presented in this chapter is well suited for platform control and can be used as a template, allowing a high level of OBC design freedom, and enabling a considerable amount of testing to be inherited from COTS components and logic.

Thread migration has been shown to be a powerful tool for assuring FT, but prior research ignores fault detection, and imposed tight constraints on an application's type and structure (e.g., video streaming and image processing [241]). Thread-level coarse-grain lockstep of weakly coupled cores instead supports general purpose computing, and in the past, has already been used for high availability, non-stop service, and error resilience concepts. However, in prior research, faults are usually assumed to be isolated, side effect free, and local to an individual application thread [208] or transient [199, 205], entailing high performance [209] or resource overhead [210, 211]. More advanced proof-of-concepts [198, 199], however, attempt to address these limitations, and even show a modest performance overhead between 3% and 25%, but utilize checkpoint & rollback or restart mechanisms [199], which make them unsuitable for spacecraft command & control applications.

Many of these limitations and obstacles ultimately can be attributed to low maturity, as a majority of software-FT concepts are published as a concept TRL1 but remain unvalidated. Hence, they could be uncovered, and in many cases, can be potentially resolved through implementation and practical validation [198], increasing maturity to TRL2 or TRL3. However, development of a testable proof-of-concept is a time consuming and costly undertaking [300], as outlined among others by Sangchoolie et al. [301] with limited immediate yield for academic publication. Fault injection for entire OS instances is especially non-trivial [302], as thorough preparation and careful tool-selection is necessary to obtain representative results from a fault injection experiment [303]. Therefore, a broad variety of TRL1 software-FT concepts exist today at a theoretical level [212–214], for which validation was only conducted statistically using modeling with different fault distributions or not a all. In this chapter, we therefore conduct validation of our coarse-grain lockstep approach using systematic fault-injection. Thereby we verify the effectiveness of our coarse-grain lockstep FDIR mechanisms under stress using a RTOS-based proof-of-concept implementation, increasing maturity to TRL3.

9.3 A Hybrid Fault Tolerance Approach

Conventional FT architectures require proprietary logic in hardware to facilitate fault detection and coverage. In contrast, the architecture described in this chapter can

offer strong FT using just COTS components and proven standard library logic. This is made possible through the use of the FT approach we presented in Chapter 4. The high-level functionality of this approach is depicted in Figure 62, and consists of three interlinked fault mitigation stages implemented across the embedded stack:

Stage 1 implements forward error correction and utilizes coarse-grain lockstep of weakly coupled cores to generate a distributed majority decision across compartments. Fault detection is facilitated through application callback functions, without requiring deep modifications to an application or knowledge about intrinsics.

Stage 2 recovers failed compartments through reconfiguration and self-testing. It assures the integrity of programmed logic and deploys configuration scrubbing, as well as Xilinx Soft-Error-Mitigation (SEM), to correct transients in FPGA fabric. Its objective is to assure and recover the integrity of processor cores and their immediate peripheral IP through FPGA reconfiguration and the use of differently routed and placed alternative configuration variants, thereby counteracting resource exhaustion.

Stage 3 engages when too few healthy compartments are available, and re-allocates processing time to maintain reliability. To do so, thread-level mixed criticality is exploited, assuring sufficient compute resources are available to high-criticality applications by sacrificing performance or availability of lower-criticality threads.

Further details including benchmark results are available in Chapter 4. The main target in our project is the ARM Cortex-A53 application processor, which is today widely used in embedded and mobile-market devices. However, this research is processor and ISA independent. In this chapter, we describe an MPSoC design and

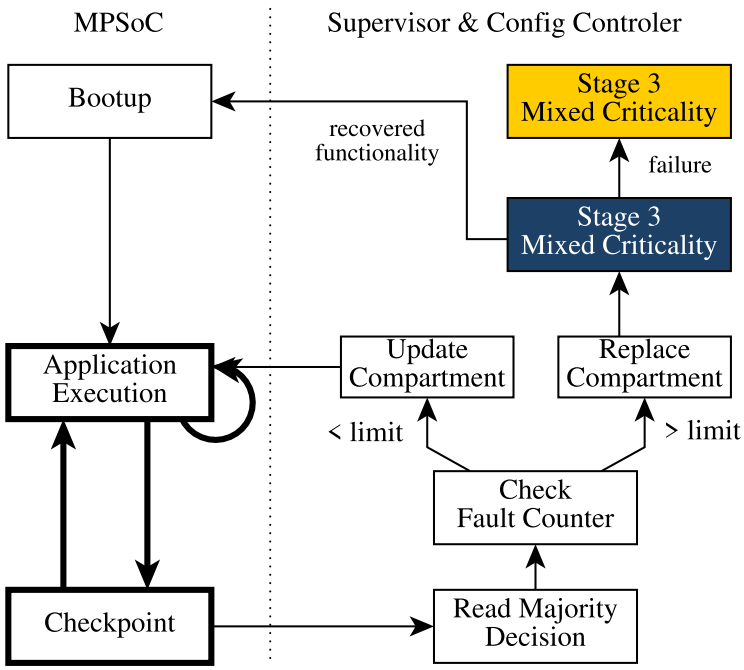


Figure 62: Stage 1 (white) assures fault detection (bold) and fault coverage. Stages 2 (blue) and 3 (yellow) counter resource exhaustion and adapt the on-board computer application schedule to reduced system resources.

architecture template, which is enabled by this approach and can be reproduced in Xilinx Vivado 2017.1 and later.

9.4 The MPSoC Architecture

We developed our software-FT architecture for use on top of an MPSoC consisting only of COTS technology. The main target in our project is the ARM Cortex-A53 application processor. For many size-optimized space applications, smaller cores such as the Cortex-A32, A35 and A5 may also offer a better balance between performance, universal platform support, and logic utilization. The Cortex-A53 core was chosen as it is today widely used in a variety of industrial and mobile-market devices, though our architecture is processor and instruction set architecture (ISA) independent.

In this section, we describe a publicly reproducible MPSoC design variant implementing our architecture, which can be designed in full using Xilinx library IP and Microblaze processor cores. The architecture minimizes shared logic, compartmentalizes compartments, and offers a clearly defined access channel between compartments and the supervisor, and is depicted in Figure 63.

9.4.1 Supervision & Reconfiguration

Stage 1 can be implemented on a single chip, but we utilize an off-chip supervisor to facilitate FPGA reconfiguration and transient fault scrubbing in the running configuration. The outlined multi-stage FT approach puts only minimal load on the supervisor, and it can thus be again implemented using a traditional radiation hardened or tolerant microcontroller. The FeRAM-based TI-MSP430FR family would be a solid somewhat radiation-tolerant but non-FT substitute, which is today widely used aboard a broad variety of CubeSats and low-performance COTS products designed for nanosatellite use. The level of performance offered by such microcontrollers is usually sufficient only for educational CubeSats and federated systems. However, a supervisor

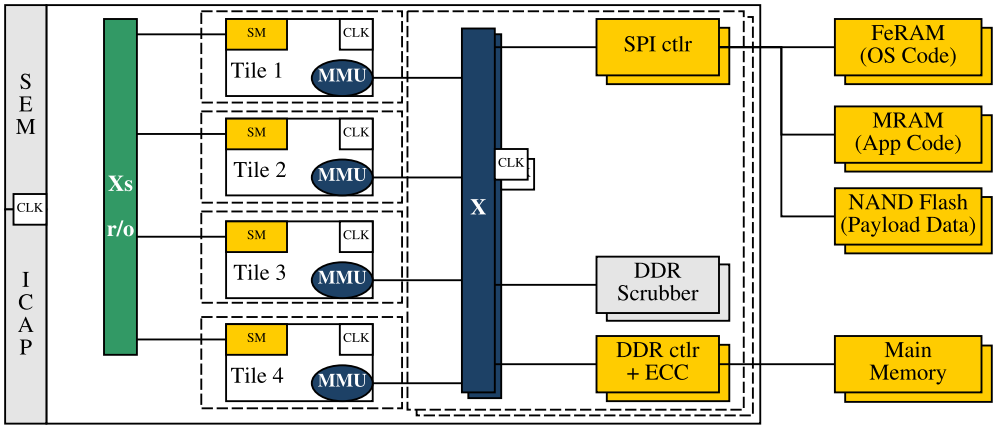


Figure 63: The topology of our compartmentd MPSoC design. Each compartment exists in its own reconfiguration partition and therefore also clock domain, simplifying routing and logic placement. Reconfiguration partitions are indicated with dashed lines.

in our architecture only receives the majority voting results from the coarse grain lockstep, controls the FPGA, and facilitates reconfiguration through an ICAP controller in static logic. Hence, the low level of performance of an MSP430FR, for example, is sufficient, and allows an ultra-low-cost implementation of our approach for academic CubeSat projects and scientific instrumentation.

We deployed configuration error mitigation through Xilinx SEM in combination with supervisor-side scrubbing to safeguard logic integrity. However, SEM and scrubbing only detect faults in specific components of the FPGA fabric (e.g., not in BRAM), leaving significant parts of the design unprotected unless logic-side ECC is used.

These measures alone do not provide sufficient protection for fine-feature size FPGAs. Thus, our software-FT functionality can locate faults in the partition of a specific compartment, allowing the supervisor to resolve them using reconfiguration. We place compartments in separate configuration partitions to enable partial reconfiguration of individual compartments, without affecting the rest of the system.

As depicted in Figure 62, the supervisor only reacts to disagreement between compartments, otherwise remaining passive. It maintains a fault-counter for each compartment and acts as a watchdog. When resolving transient faults within a compartment, it increments the fault-counter and induces a state update through a low-level debug interface. After repeated faults, the supervisor will replace the compartment by adjusting the thread-mapping of a spare compartment, activating it, and rebooting the faulty compartment. In case a system developer indicated threshold is exceeded, the disagreeing compartment is assumed permanently defunct and not re-used as a spare.

To allow supervisor access to a compartment and its address space, each compartment is equipped with an AXI debug-bridge (Figure 64). The supervisor can trigger execution of self-test functionality within a compartment to detect faults in peripherals. It can also trigger an adjustment of a compartment's thread allocation as part of Stages 1 and 3, making the MPSoC's computational performance, robustness and energy consumption adjustable at runtime.

Majority voting between compartments can be implemented as distributed majority decision [330], then requiring no direct intervention of the supervisor during regular operation. If this is not desired, or lockstep through interrupt triggered checkpoints is implemented, then the supervisor should also take care of receiving the voting results generated on each compartment. In that case, the supervisor can access each compartment's thread mapping via each compartment's debug interface, and if necessary induce a reset or otherwise manipulate a compartment without requiring its cooperation.

9.4.2 Tile Architecture

Our MPSoC design implements multiple isolated SoC-compartments accessing shared main memory and OS code. Even though the purpose and function of these compartments is different, the topology resembles a compartmentalized architecture instead of a conventional MPSoC design, in which cores share infrastructure and peripherals. This topology increases Stage 1's fault coverage capacity and allows task mapping for general-purpose software. Each such compartment contains a processor core, local interconnect, and peripheral IP-cores and interfaces as depicted in Figure 64, resides in its own clock domain, and can be reset independently. Allocating a clock domain to each compartment improves timing, and reduces logic-overlap and interdependence

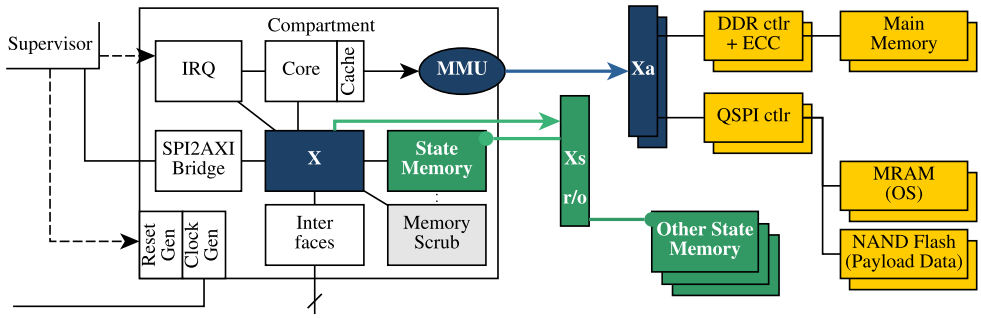


Figure 64: The logic-side architecture of a compartment. Access to local IP bypasses the cache, while access to global memory passes is cached for performance reasons.

between compartments. Furthermore, we can then also utilize partial reconfiguration and frequency scaling for each compartment, as well as clock gating.

A compartment executes a set of thread replicas, and its loss can be compensated by the rest of the system. To assure a failed compartment can not cause performance degradation in the rest of the system (e.g., by continuously accessing DDR or program memory), it can be disconnected off from the global interconnect by the supervisor. Non-masked faults (due to radiation, aging, and wear) disrupt the data or control flow of the software running on a compartment. Stage 1 builds upon this capability at the thread-level, as state differences can be detected by other compartments and often even by the malfunctioning compartment itself as described in Chapter 8.

All compartments are equipped with an identical set of peripheral interfaces, with controllers being mapped to identical locations and address ranges. The compartment address space layout is uniform across the system and compartments are indistinguishable for software. Hence, application code and data structures are portable between compartments, simplifying thread migration drastically. This allows us to reduce the computational cost and complexity of software-lockstepping.

Thread allocation and information relevant to the coarse-grain lockstep is stored in a dedicated dual-ported on-chip BRAM on each compartment. We refer to component is as state memory, and indicate it as SM in the figures. One port is accessible to the compartment's processor core, while the other is read-only accessible to the system. This allowing low-latency information exchange between compartments without requiring inter-compartment cache-coherence or main memory access. The state memory architecture is depicted in Figure 65. The supervisor can access and modify each compartment's state memory through its debug interface on each compartment.

9.4.3 Interconnect Topology and Shared Memory

Figure 63 depicts the MPSoC's high-level topology. Our MPSoC design utilizes an AXI interconnect in crossbar mode to allow compartments access to shared main and non-volatile memory controllers, though we are currently reworking our MPSoC to instead use a NoC [329].

Main memory is shared between compartments, as SD- and DDR memory controllers are too large and require too much I/O to instantiate for each compartment. Each compartment has full access to a segment of main memory, which is mapped to the same address range on all compartments (the MMU component in the figures).

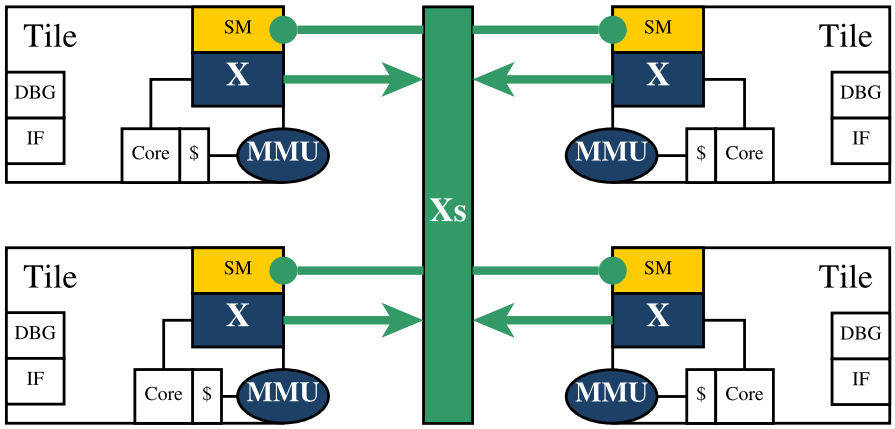


Figure 65: A compartment's state memory is accessible to all other compartments in the system. It provides a write protected, high-speed on-chip possibility to expose state-relevant data to the MPSoC as a while.

All compartments can access main memory read-only to simplify state synchronization and IPC. The supervisor can access each set of main memory controllers directly.

For nanosatellite missions to LEO, often only SECDED ECC support is required and readily available in library IP already [331], while basic error scrubbing can be facilitated in software. For critical, deep-space, and long-term missions, block coding should be used instead to compensate for the increased impact of SEEs and higher likelihood of MBUs in high-density SDRAM. Reed-Solomon ECC as well as error scrubbers are available commercially, or can be assembled from open-source IP. The main memory scrubbers are controlled by the supervisor to avoid potential interference by malfunctioning compartments. ARM Cortex-A53 as well as Microblaze caches and several local memories and buffers offer ECC support as basic functionality [331].

To safeguard main memory, FeRAM [332], MRAM [150], and mass memory from SEFIs, as well as permanent failure, these memories are implemented redundantly to enable failover. To allow non-stop operation during FPGA reconfiguration, we also implement their controllers, and the AXI interconnects they are attached to redundantly. This also enables further protective measures which we described in Chapter 7, and allows load distribution for timing critical main memory through segment interleaving. Thereby the available DDR memory bandwidth is increased and the overall latency for memory access can be reduced. This also enables us to recover an instance of a memory controller on short notice without requiring the full system to be halted[1].

Tiles compete for DDR memory access. As our architecture is implemented on FPGA, the clock frequency of each compartment's processor core is lower as on ASIC implemented MPSoCs. In consequence, the global interconnect as well as DDR memory controllers offer abundant throughput at drastically higher clock frequencies. Each processor core caches access to shared memory, drastically reducing the strain on the memory subsystem. Access to a compartment's state memory still bypasses the cache, but this is implemented directly in high-speed, low-latency on-chip BRAM. Hence,

[1]Note that depending on the used OS, a reboot of a compartment may be required. Linux supports modifications to the memory layout and relocation, while simpler OS, such as RTEMS, do not currently know such functionality.

while in principle competing for memory bandwidth, even an 8-compartment system can not saturate the two available DDR4 channels in our current MPSoC design. Ideally however, our architecture should be implemented using a NoC instead of a global AXI-interconnect crossbar, which would offer drastically better scalability, more effective caching and buffering, and also a degree of FT.

9.5 Subsystem Connectivity and Peripheral I/O

A fault resolved in Stage 1 may cause incorrect data to be emitted through I/O interfaces. This is an inherent limitation of coarse-grain lockstep concepts, and can only be slightly alleviated through additional application-intrusive work-around as described, for example, in [199]. Instead, this limitation is better solved at the logic level through interface-level voting, which is possible with minimal extra logic. For most CubeSats, most nanosatellites, and less critical microsatellite missions, however, this is usually foregone.

Larger spacecraft already utilize interface replication or even voting to assure full hardware TMR, usually requiring considerable effort in hardware or logic to facilitate this replication. Our MPSoC architecture inherently provides interface replications by design, requiring no extra measures to be taken, as the individual compartment-interfaces can be directly used for TMRed architecture. Further safeguards are necessary for very small CubeSats where interface replication is undesirable, for example, due to PCB-space constraints.

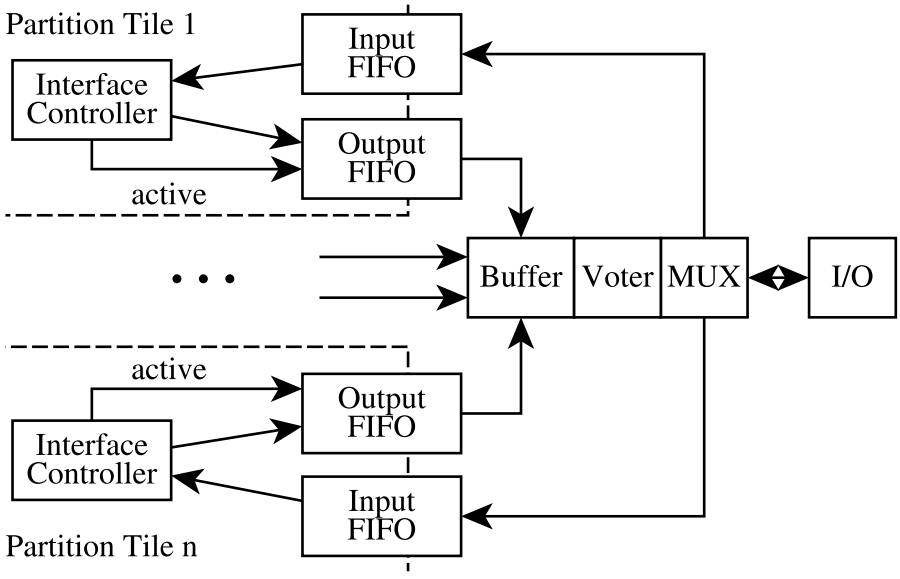


Figure 66: An activation-driven, buffered output voter with input de-multiplexer can be constructed for low-pin-count CubeSat interfaces. Note that an additional re-sampling step would be required in case of different thread scheduling on lock-stepped compartments.

9.5.1 Electrical- and Logic-level Interface Voting

For simple embedded interfaces like I2C and SPI connected to "dumb" sensors or actuators with no user configurable firmware, a simple majority decision per I/O line is possible. While hardware voting is challenging for large arrays of voters running synchronized at very high frequencies, the CubeSat-relevant interfaces are electrically simple, have a very low pin count, and run at relatively low clock frequencies. Hence, voting for these interfaces can efficiently be implemented on-chip through simple voters assuming compartments signals interface activity.

Our coarse grain lockstep mechanisms allow software to be executed with slight timing variations. These may be caused by clock-domain interactions, competition of compartments for global interconnect DDR4 and QSPI access, as well as differences in compartment partition routing and or I/O pin placement. In general, these variations will be limited to few clock cycle duration. I/O on these interfaces must be buffered, which can be done within the FPGA as discussed further also by Li et al. in [333]. For simplicity, compartments should also indicate that an interface is active, and we can double-use the chip-select pins present in almost all I2C and SPI implementations. The voter can use activity on these pins as indication that the interfaces is active, and delay voting for a given amount of clock cycles using a set of FIFO buffers. The depth of these FIFOs thereby determines the maximum delay compensated by the voter [334]. In our design we can utilize a combination of re-sampling majority voter and MUX as depicted in Figure 66.

Note that larger MPSoC variants with 6 or more compartments can host multiple independent lockstep sets as described in Chapter 6. In this case, simple buffered voting is insufficient, as compartments could then also run mixed lockstep groups where threads may be scheduled with much larger time differentials. This differential will always be shorter than the duration of a lockstep cycle or the frame time, but in LEO these may extend to up to several seconds. It would be uneconomical and, depending on the application, even technically infeasible to buffer I/O for long duration. However, we consider the design-combination of a low-end CubeSats that can not afford subsystem TMR, packet-based communication, with a high-performance 6-core MPSoC not very attractive and therefore a corner case. If this combination was still deemed necessary, a straight forward solution would be to maintain multiple isolated thread-assignment groups.

9.5.2 Simple Inter-Subsystem and Controller Networks

Many SPI and I2C implementations support multi-master shared bus operation, and it is possible to even create large and complex CAN-bus networks [335]. CubeSats often use these interface standards for low-speed inter-subsystem communication in simple CubeSat designs [39, 336]. While packet based interfaces offer far better scalability, reliability, and fault-mitigation properties for this purpose [337], in reality these concepts will remain in use aboard CubeSats for the foreseeable future. However, in contrast to interfacing with "dumb" endpoints ICs, these networks[2] usually consist of microcontrollers running satellite developer provided software. In this case, a better solution to de-replicating and obtain consensus within the system of our MPSoC's compartments is to make the subsystems aware of the replication.

[2]In CubeSat jargon often referred to as "buses".

A subsystem controller then can await receiving a second replica of a command sequence from a different master. Of course this does not solve the issue of a single compartment/master jamming or saturating the bus due to malfunction. However, most CubeSats using these interfaces as subsystem-bus currently usually also do not take actual meaningful countermeasures in this regard. This is technically possible, but requires entirely different network topologies [335, 337] than the simplistic single-level bus concepts used aboard CubeSats today [39].

9.5.3 Packet Switching and Routing On-Board Networks

For packet-based interfaces such as Spacewire [338], AFDX [94], CAN [55], or Ethernet [73], no hardware- or logic-side solution is necessary. There, packet duplication and integrity checking can be managed efficiently at the data link, network and transport layers (OSI layers 2 – 4 [339]). At the physical layer, Ethernet and thereof derived technologies such as AFDX [94] and TTEthernet [340] perform shared medium through collision detection and micro-segmentation with frame switching. Then, packet routing (L3) and de-duplication in software at the higher OSI layers can be deployed, e.g., in software. Today, this is common practice in relevant industrial applications such as AFDX and TTEthernet used in related fields such as atmospheric aerospace or safety critical automotive applications.

The FPGAs considered in our research provide an abundance of high-speed MGT transceivers. These are intended to support high-performance serial interfaces such as PCIe, or USB3 host interfaces [341], which may become attractive for CubeSat use in the future and have built in error correction support. Even the smallest XCKU3P part fields 16 such interfaces, and the location of these interfaces is in very attractive locations for using 2-3 of them isolated within each of our MPSoC's compartments [342]. In practice, this would allow for a very scalable, high-performance CubeSat inter-subsystem communication architecture [343] at little cost assuming a the satellite's high-level design takes this into account.

9.6 Implementation Considerations

The MPSoC architecture described in this chapter was developed for miniaturized satellite use, as an ideal platform for the software-FT approach described in Chapter 4. This architecture is not specifically dependent on utilizing ARM processor cores, but can be implemented with any FPGA-implementable soft-core. Our choice of the ARM platform was taken in part to allow thread migration between soft- and hard-cores (e.g., on Zynq Ultrascale+), maximum comparability to COTS mobile-market and embedded MPSoCs with secondary use aboard a major share of CubeSats. Especially for low-budget CubeSat users in research or university projects, standard vendor library cores such as Xilinx Microblaze may be an excellent alternative to our Cortex-A choice. These cores offer erasure coding and other basic fault tolerance features out of the box already, and performed rather well in radiation tests [331]. They are readily available and often even free of charge, especially to academics and non-commercial scientific research users.

We implemented a proof-of-concept on a Xilinx XCKU5P FPGA with modest resource utilization (28% LUTs, 33% BRAMs, 16% FFs, 5% DSPs) and 1.92W total

power consumption with Microblaze cores. In this 4-compartment design, each compartment was equipped each with one peripheral I2C master controller, one SPI master, as well as a dual-channel GPIO controller. Such an interface configuration is representative for most CubeSat applications, while AFDX, TTEthernet, and Spacewire are today not widely used aboard CubeSats.

This approach and architecture could very well be implemented on ASIC without reconfiguration and Stage 2, and we see this as a "big-space" variant of our approach. An ASIC implementation offers lower energy consumption, and allows higher clock rates due to reduced timing and shorter paths. If manufactured in an inherently radiation hard technology such as FD-SoI [144], it would be less susceptible to transients and more robust to permanent faults. Due to the drastically increased development cost and required manpower, the resulting OBC would not be viable for most miniaturized satellite applications (not anymore "on a budget").

9.7 Conclusions

The 3-stage FT approach combined with its MPSoC host system presented in this chapter is the first practical, non-proprietary, affordable architecture suitable for FT general-purpose computing aboard nanosatellites. It utilizes FT measures across the embedded stack, and combines topological with software functionality, utilizing only extensively validated standard parts. Thereby, we enable the use of nanosatellites in critical space missions, while the architecture allows trading processing capacity for reduced energy consumption or fault coverage.

An OBC relying upon this architecture can be facilitated with the minimal manpower and financial resources. The MPSoC can be implemented using only COTS hardware and extensively validated, and widely available library IP, requiring no proprietary logic or costly, custom space-grade processor cores. It offers a high level of resource isolation for each processor, utilizing architectural features originally conceived for ManyCore systems to achieve FT.

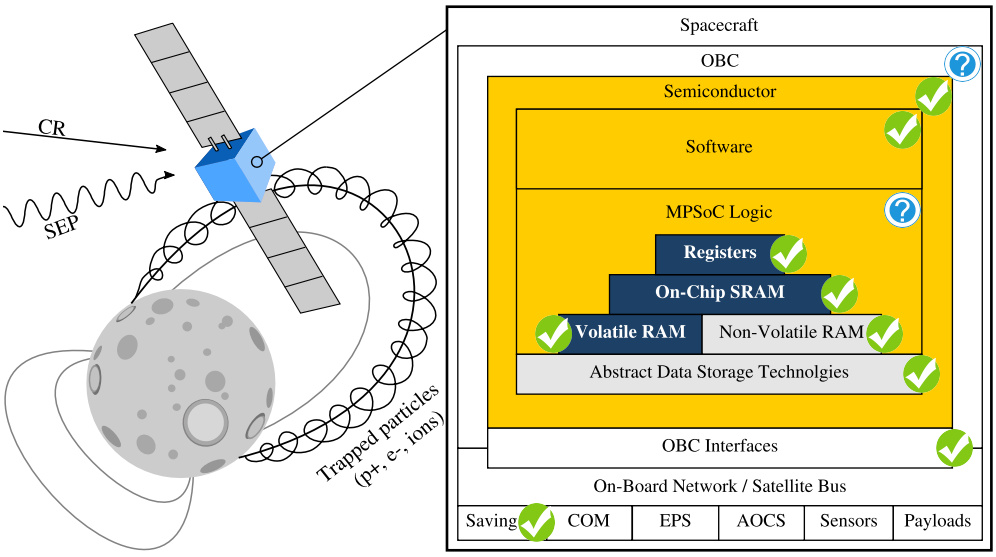
Each compartment functions as a stand-alone processing compartment with dedicated I/O, existing in its own clock domain and reconfiguration partition, thereby minimizing shared resources and reducing routing complexity. Compartments were purposefully designed to best support thread-level coarse-grain lockstep of weakly coupled cores, while allowing partial reconfiguration without stalling the rest of the system. The architecture was implemented successfully, and tested on current generation Xilinx Zynq/Kintex and Virtex FPGAs with 4, 6 and 8 compartments, and validated through fault-injection into RTEMS.

Chapter 10

On-Board Computer Integration and MPSoC Implementation

Practical Design Verification on FPGA

In this chapter, we present a practical implementation results for our MPSoC design, as just making up fault tolerance concepts would be insufficient to answer RQ6. We show that this on-board computer architecture in its full functionality can be implemented with a low component cost, and only with standard development tools and IP. We achieve 1.94W total power consumption, which is well within the power budget range achievable aboard 2U CubeSats and larger satellites. This serves as proof-of-concept for our architecture and answers RQ6, paving way to radiation testing and on-orbit demonstration in the future.



10.1 Introduction

Cheap, CTOS electronics designed for the embedded and mobile-markets are the foundation of modern nanosatellite design. They offer an excellent combination of low energy-consumption, minimal cost, and broad availability. However, such components are not designed for reliability, and include only rudimentary fault tolerance capabilities. Due to the elevated risk of loosing a satellite due to failure of these components, CubeSat missions today are kept brief or up-scaled to larger, more expensive satellite form factors.

Low-complexity, low-performance satellite on-board computer (OBC) designs have allowed a variety of successful CubeSat missions, with a few missions even operating successfully for as long as 10 years. This demonstrates that there is no fundamental, hard technological barrier that could prevent the use of modern semiconductors in space missions. However, these designs are sufficient only for missions with very low performance requirements, e.g., for educational missions and brief technology demonstration experiments.

Many sophisticated scientific and commercial applications can today also be fit into a CubeSat form factor, which make a much longer mission duration desirable. To fly these payloads, a CubeSat has to process and store drastically more data, and at all levels requires increased performance. Therefore, all advanced CubeSats today utilize industrial embedded and mobile-market derived systems-on-chip (SoC), which offer an abundance of performance. However, these SoCs in turn are manufactured in modern technology nodes with a fine feature size. They are drastically more susceptible to the effects of the space environment than simple but robust low-performance microcontrollers. Hence, proper fault tolerance capabilities are needed to ensure success for advanced long-term CubeSat missions, as gambling against time and radiation can be risky.

Radiation hardening for big-space applications can not be adopted, as this approach is only effective for very old or very proprietary and costly manufacturing processes. Budget, energy, and size constraints prevent the use of traditional space-grade components used aboard large satellites, while component-level fault tolerance significantly inflate CubeSat system complexity and failure potential. Today, no fault-tolerant computer architectures exist that could be used aboard nanosatellites powered by embedded and mobile-market semiconductors, without breaking the fundamental concept of a cheap, simple, energy-efficient, and light satellite that can be manufactured en-mass and launched at low cost. Hence, we developed a scalable, yet simple OBC architecture that allows high-performance MPSoCs to be used in space, and is suitable for even small 2U CubeSats.

Our proof-of-concept OBC utilizes Microblaze processors on a low-power FPGA, exploits partial reconfiguration and software-implemented fault tolerance to handle system failure. It is assembled only from COTS components available on the open market, standard vendor library IP, and runs standard operating system and software. To protect our system, we utilize a combination of runtime reconfigurable FPGA logic and software-implemented fault tolerance mechanisms, in addition to well understood and widely available EDAC measures. We facilitate fault tolerance in software, which enables our system to guarantee strong fault coverage without introducing the hard design limitations of traditional hardware-TMR based solutions.

Our OBC architectures can efficiently and effectively handle permanent faults in

the FPGA fabric by utilizing alternative FPGA configuration variants. It ages gracefully over time by adapting to an increasing level semiconductor degradation, instead of just failing spontaneously. The performance of the OBC itself is adjustable, allowing spacecraft operator to modify system parameters during the mission. An operator can trade processing-capacity and functionality to achieve increased fault coverage or reduced energy consumption, without interrupting satellite operations. Thereby, we can maintain strong fault coverage for missions with a long duration, while adjusting the OBC to best meet the requirements of complex multi-phased space missions.

To our understanding, this is the first scalable and COTS-based, widely reproducible OBC solution which can offer strong fault coverage even for 2U CubeSats. We provide an in-depth description of our proof-of-concept MPSoC, which requires only 1.94W total power consumption, which is well within the power budget range achievable aboard 2U CubeSats. In the next section, we provide a brief overview over the status-quo in fault-tolerant computer system design for large spacecraft, CubeSats, and ground use. Subsequently in Section 10.3, we describe our OBC's component-level architecture, the MPSoC used, as well as the interplay between the different components of the OBC. Before providing conclusions, we present our implementation results and details about how this MPSoC was tested and validated in Section 10.5. Finally, we discuss advanced applications of our proof-of-concept with multiple FPGAs, Network-on-Chip usage and resistance to full-chip SEFIs in Section 10.4. All components required to re-implement this OBC design are available at low cost to scientists and engineers in an academic environment. The necessary IP and standard design are available free of charge from the relevant vendors, e.g., through Xilinx's university program for academics and scientific users.

10.2 Related Work

In contrast to the initial generation of educational CubeSats, today fewer satellites fail due to practical design problems caused by inexperience [39]. Instead, Langer et al. in [2] showed that a majority of these failures can be attributed to electronics heavy subsystems. Even experienced, traditional space industry actors with years of experience in large satellite design, who develop CubeSats satellites "by the traditional book" with quasi-infinite budgets today struggle to reach just 30% mission success [42].

The main source of failure are environmental effects encountered in the space environment: radiation, thermal stress, and corruption of critical software components that can not be recovered from the ground, and failures caused by power electronics. Considering again Langer et al., [2], with increasing age mission duration, a broad majority of documented failures aboard CubeSats originate from OBCs, transceivers, and the electrical power subsystem. While functionally disjunct, these subsystems all have in common that they are heavily computerized and architecturally rather similar, built around one or multiple microcontrollers and memories.

Fault tolerance concepts targeting generic commercial ground-based computing applications usually cover only a small subset of our fault model: transient faults, material aging, and occasionally gradual wear. Such assumptions are valid for critical applications for ground applications, but not for space applications. Often, the introduction of permanent faults breaks fault tolerance concepts for ground applications, weaken their protective capabilities strongly, or limit their protection to only a brief period of time. Most ground-based and atmospheric aerospace fault tolerance

concepts also aim to guarantee reliable operation from the point in time a fault occurs until maintenance can be performed. This is a problematic assumption for CubeSat use, as servicing missions have only been performed on rare occasions for spacecraft of outstanding scientific, national, and international significance such as the International Space Station or the Hubble Space Telescope. But certainly not for low-cost CubeSats.

These limitations, however, by using a combination of different additional fault tolerance measures across the embedded stack. Fault tolerance concepts for ground and atmospheric aerospace applications can therefor serve as building blocks to design a fault-tolerant architecture for space applications.

10.2.1 Fault Tolerance for Large Spacecraft

Traditional OBCs for large satellites realize fault tolerance using circuit-, RTL- [344], IP-block- [104, 132], and OBC-level TMR [90] through costly, space-proprietary IP. They make heavy use of over-provisioning and tries to include idle spare resources (processor cores, components, memory, ...) where necessary. Naturally, this is done at the cost of performance and storage capacity, increases system complexity, and power consumption. Circuit-, RTL-, and core-level measures are effective for small microcontroller-SoCs [88,345], if they are manufactured in large feature-size technology nodes. More and more error correction and voting circuitry is needed to compensate for the increased severity of radiation effects with modern technology nodes [345]. This in turn again inflates the fault-potential, requiring even more protective circuitry, making this approach ineffective for modern semiconductors.

Processor lockstep implemented in hardware lacks flexibility, limits scalability, and is feasible only for very small MSoCs with few cores [88, 346]. Timing and logic placement becomes increasingly difficult for more sophisticated processor designs, and becomes infeasible for SoCs running at higher clock frequencies. Practical applications run at very low clock frequencies [347] with two or three very simple processor cores, even for ASIC implementations [88, 132]. Common to all these solutions is that they are proprietary to a single vendor, implying a hefty price tag and tight functional constraints. Especially the space-proprietary single-vendor solutions available are often difficult to develop for, have in many cases no publicly available developer documentation, have no open-source software communities which could provide support in development, and usually imply vendor lock-in into a walled garden ecosystem.

To design nanosatellites, we instead utilize the energy efficient, cheap modern electronics [41], for which traditional radiation-hardening concepts become ineffective. Specifically, CubeSats utilize COTS microcontrollers and application processor SoCs, FPGAs, and combinations thereof [40, 41]. Some of these were shown to performing well in space, and others poorly. On-orbit flight experiences varying drastically even between different controller models of the same family and brand [39]. Specifically, components that were discovered to perform well are very simple microcontrollers with a minimal logic footprint and low complexity. These are manufactured in coarse feature-size technology nodes, and were by coincidence designed to be rather tolerant to radiation (radiation-hard by serendipity) [46]. Examples of such parts are the PIC controller family, which are logically extremely simple, and controllers that include inherently radiation-tolerant functionality such as the Ferroelectric RAM (FeRAM) [332] based MSP430FR family [225]. Unfortunately, these “well behaved” components also

offer very limited performance, which is sufficient only for simple educational missions, technology demonstration, and short low-data rate science missions.

Computer designs for nanosatellites utilized about 10 years ago began to heavily utilize redundancy at the component level to achieve failover, to provide at least some protection from failure. However, practical flight results show that such designs are complex and fragile, as compared to entirely unprotected ones [39, 41]. Entirely unprotected OBC designs, in turn, may fail at any given point in time. However, today satellite designers are usually forced to simply accept this risk, leaving the hope that a satellite will by chance not experience critical faults before its mission is concluded. Risk acceptance is viable only for educational, and uncritical, low-priority missions with a very brief duration.

10.2.2 Fault Tolerance Concepts for COTS Technology

FPGAs have become popular for miniaturized satellite applications as they allow a reduction of custom logic and component complexity. FPGA-based SoCs can offer increased FDIR potential in space over ASICs manufactured in the same technology nodes [40] due to the possibility to recover from faults through reconfiguration. Transients in configuration memory (CRAM) can usually be recovered right away through reconfiguration [105], while permanent faults may be mitigated using alternative configuration variants. However, fine-grained, non-invasive fault detection in FPGA fabric is challenging [345], and is a subject of ongoing research [239, 240]. Applications thus rely on error scrubbing, which has scalability limitations and covers only parts of the fabric.

Software implemented fault tolerance concepts for multi-core systems were identified as promising already in the early days of microcomputers [131], but was technically unfeasible and inefficient until few years ago. Modern semiconductor technology allows us to overcome these limitations and recent research [348,349] shows that modern MultiCore-MPSoC architectures can theoretically be exploited to achieve fault tolerance. However, these are incapable of general-purpose computing, and instead cover deeply embedded applications with a very specific software structure [241, 350]. They require custom processor designs [348], or programming models which are suitable for accelerator applications [349]. The fundamental concept of software-implemented coarse-grain lockstep, however, is flexible and can be applied, e.g., to MPSoCs for safety-critical applications [348, 351], networked, distributed, and virtualized systems [201].

10.3 A Reliable CubeSat On-Board Computer

A system designed for robustness must avoid single-points of failure and assist in fault-detection. It should also support non-stop operation. Ideally, it should be capable of tolerating the failure of entire block and individual attached component. The OBC architecture presented in this chapter consists of an FPGA and a microcontroller in tandem, which is used for test and diagnostic purposes. Within the FPGA, we implement an MPSoC architecture, which is then made fault-tolerant using software measures, while its robustness is increased using memory EDAC and FPGA reconfiguration.

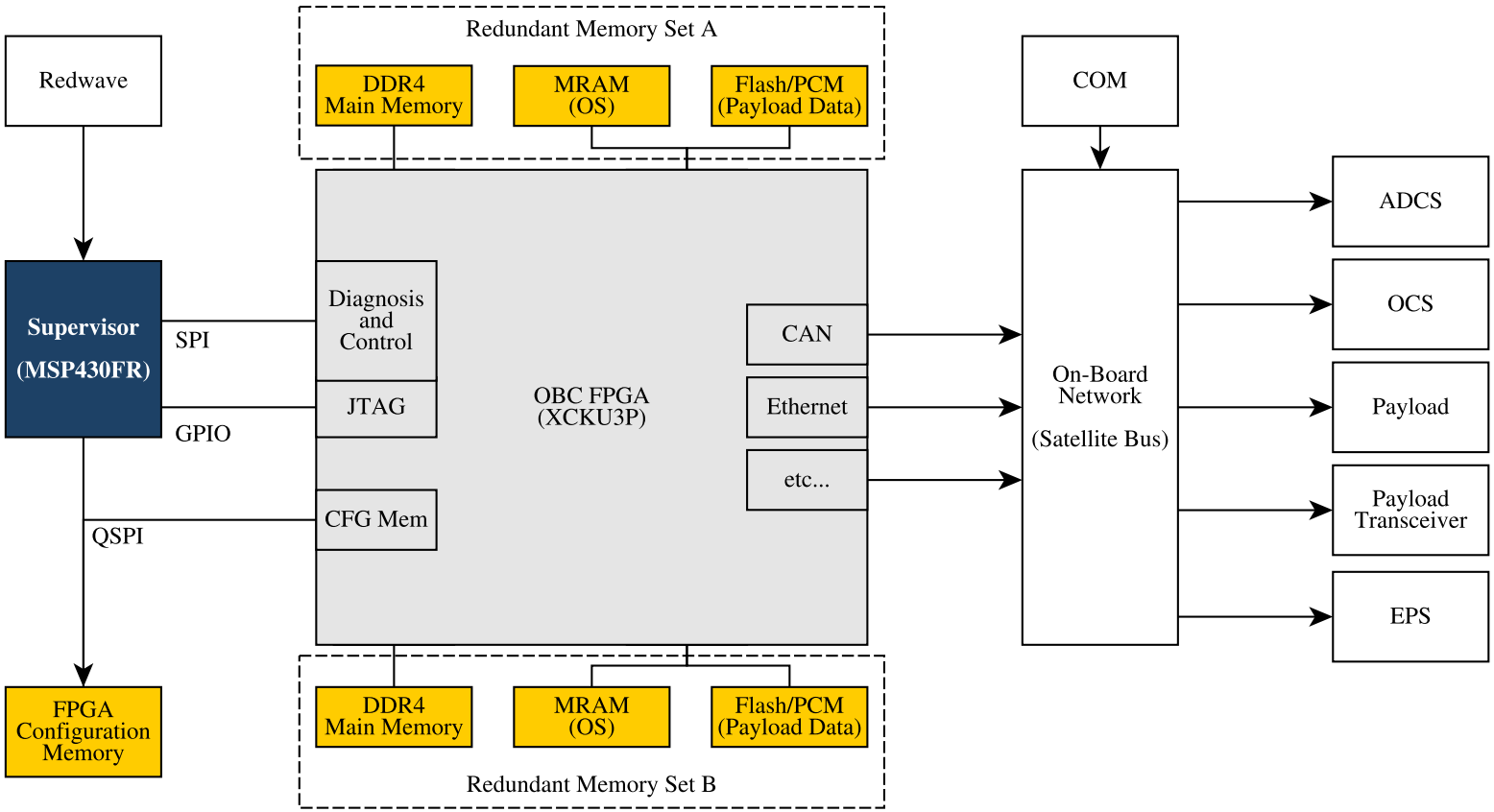


Figure 67: A component-level diagram of our OBC architecture. This architecture is intended as an in-place substitute for a conventional ASIC-based System-on-Chip, and only adds a second set of memory ICs to counter component-level failure.

However, conventional MPSoCs follow a centralist architecture with processor cores sharing functionality where possible to minimize footprint, optimize access delays, improve routing [238]. There, processor cores share memory in full, and have full access to all controllers operating within this address space, to maximize system functionality and code portability. In consequence, conventional high-performance computer designs offer only weak isolation for application running on different processor cores for the sake of performance. Faults in one core may therefore compromise the functionality of other cores and the MPSoC as a whole. This increases the overall failure-potential sharply as compared to very small microcontroller SoCs, as an MPSoC's logic does not have only a larger footprint, but also more components that can independently cause such a system to fail.

From a fault tolerance perspective this is undesirable, and in our OBC we follow a different approach. Designers of fault-tolerant processors for traditional space applications handle this issue by utilizing custom fault-tolerant processor cores, to assure that faults occurring within a core are mitigated and covered before they could propagate. For miniaturized satellite use, this is not feasible, and instead we must achieve fault-isolation and non-propagation through system-, software-, and design-level measures. In the remainder of this section, we show how this can be done with only commodity COTS components and tools that are available to academic CubeSat designers.

10.3.1 System- and Component-Level Architecture

We designed out architecture as in-place replacement for a conventional MPSoC-driven OBC design and utilize a commodity FPGA. The component-level topology of our OBC design is depicted in Figure 67.

We utilize an FPGA to realize an MPSoC that offers strong isolation between the individual processor cores, and to enable recovery from permanent faults. This FPGA serves as main processing platform for our OBC, and capable of running a full general-purpose OS such as Linux. We implemented a proof-of-concept of our OBC architecture using Xilinx Kintex and Virtex Ultrascale+ FPGAs, as well as the earlier generation Kintex Ultrascale FPGAs. For CubeSat use, only Kintex Ultrascale+ FPGAs are relevant at this point due to drastically reduced power consumption as compared to older generation and Virtex FPGAs. We provide further details on this MPSoC in the second to next subsection.

To store the FPGA's configuration memory is attached to the FPGA via SPI. The FPGA by default acts as SPI-master for this memory and automatically loads its configuration from there. In our proof-of-concept implementation, we utilize conventional NOR-flash [153] for this purpose, which also is included on most commercial FPGA development platforms. However, NOR-flash is inherently prone to radiation [153], and phase-change memory (PCM [284]) is much better suited for this task as its memory cells are inherently radiation-immune. Thus, in future applications and in our prototype, we will utilize a PCM IC instead of serial-NOR-flash.

Like most CubeSat OBCs, our OBC includes an additional microcontroller which acts as watchdog, and performs debug and diagnostic tasks. However, as we are utilizing an FPGA as the main processing platform, it only controls the FPGA and the MPSoC implemented within it. Hence, it acts as a saving subsystem (redwave/hard-command-unit), and can resolve failures within the MPSoC its peripheral ICs for diagnostics purposes in case the MPSoC became dysfunctional. To reflect this role,

we refer to it as "supervisor".

As depicted in Figure 71, the supervisor is connected to the FPGA through GPIO and SPI. The SPI interface allows low level diagnostic access to different parts of the MPSoC, as well as facilitate low-level test access to FPGA-attached components. Through the GPIO interface, the supervisor controls the FPGA's JTAG interface and can reset the FPGA as well as different parts of the MPSoC. The FPGA also has access to the FPGA's configuration memory, and shares this SPI bus with the FPGA in a multi-master, so that in case of failure, it can independently reconfigure the FPGA.

The supervisor itself is not connected to other satellite subsystems, and can not control other parts of the satellite beyond the OBC itself. During regular operation, it takes no part in the normal data processing operations of the OBC and only receives correctness information from the MPSoC, which is further described in Chapter 4. However, for failure diagnostics the supervisor can be used to reprogram the OBC FPGA to access the rest of the satellite through its interfaces for debug purposes. Therefore, the supervisor requires very little processing power, and we utilize a robust low-performance MSP430FR5969 microcontroller. The MSP430FR controller family is manufactured with inherently radiation-tolerant FeRAM instead of flash, and has become popular in low-performance COTS CubeSat products due to its good performance under radiation and in space [225]. A space-grade substitute is available in the form of the MSP430FR5969-SP.

10.3.2 Memory Components

Besides the FPGA, configuration memory, the supervisor, and the usual power electronics, our OBC architecture includes two redundant sets of memory ICs for use by the MPSoC implemented on the FPGA. Each memory set includes DDR memory used as main working memory by the MPSOC, magnetoresistive-RAM [150] (MRAM) used to store the operating system and flight software, as well as PCM for holding payload data. In our development-board based proof-of-concept, we are constrained to substituting MRAM and PCM with NAND-flash due to hardware constraints.

DDR-SDRAM is prone to radiation-induced faults [250], though with modern high-density components manufactured in fine technology nodes, the likelihood to experience bit-upsets is low [255, 352]. Hence, for most nanosatellite missions single-bit correcting error correction coding (ECC) [254] is sufficient to protect the integrity of data stored [251] as long as error scrubbing is implemented [353]. In LEO, scrubbing intervals can be kept very low, e.g., once per orbit, as the particle flux and likelihood to receive bit-flips with modern DDR memory is minimal. This can be realized using software-measures as we showed in Chapter 7. ECC can be implemented using standard Xilinx Library IP [331], as well as free open-source cores from OpenCores, and the GPL version of GRLIB. Specifically, standard Xilinx design software out-of-the-box includes the necessary library IP for Hsiao and Hamming coding.

For CubeSats venturing to areas in the solar system with more intensive radiation bombardment, continuous memory scrubbing can be implemented in logic within the MPSoC. Then, stronger EDAC with longer code-words and larger code-symbols should be used, instead of the weaker coding that can be assembled using Xilinx library IP. Symbol-based ECC can compensate better for the effects of radiation in modern DDR-SDRAM: despite occurring less frequently overall, highly charged particles have an increased likelihood to cause multi-bit upsets instead of changing the state of just

a single DRAM cell. EDAC using Reed-Solomon ECC as well as interconnect error scrubber IP cores are available commercially, e.g., via Xilinx or from the commercial GRLIB library. Alternatively, they can be assembled from open-source IP, available from OpenCores, and a broad variety of other open-source code repositories. However, the quality of such cores is often uncertain, and even a good part of the IP available through the curated OpenCores catalog is known to be defunct. Memory scrubbing can be assembled on the FPGA from standard library IP, while ready-made scrubbers are available commercially (e.g., the "memscrub" IP core from commercial GRLIB).

To store the OBC's OS and its data, COTS MRAM ICs are available at low cost on the open market today and flight experience with the parts inside earlier CubeSats has been overwhelmingly positive. However, only the memory cells of these memories are radiation immune. Without further measures, they are still susceptible to misdirected read- or write access, and SEFIs. We showed in Chapter 7 that these issues can be mitigated in software, through ECC, and redundancy. We also showed that this can be achieved with minimal overhead through the use of a bootable file-system with Reed-Solomon erasure coding. FeRAM would be more power efficient than MRAM, and is also inherently radiation tolerant, but its low storage density makes it insufficient for our use-case.

For storing applications and payload data, memory technologies with a much higher storage density than MRAM are necessary. In practice, this limits us to use NAND-flash and PCM, of which only the latter is radiation-immune. The storage cells of both have a limited lifetime, and therefore are subject to wear. However, high-density PCM has not become widely available on the open market, and so we currently have to resort to using NAND-flash. Fault tolerance for these memories can again be realized in software. As both these memories suffer from use-induced wear, the necessary functionality to handle wear is needed to efficiently safeguard their long-term use. Therefore in Chapter 7, we presented MTD-mirror, which combines LDPC and Reed-Solomon erasure coding into a composite erasure coding system.

One of the main causes for failures in commercial memory ICs of all memory technologies are faults in control logic and other infrastructure elements, causing SEFIs [255]. These may cause temporary or permanent failure of memory ICs, regardless of the memory technology used, which can not efficiently be mitigated through erasure coding. Instead, redundancy for these devices is needed, which we can realize by placing two memory sets. However, we do not implement failover in hardware, but merely connect the two memory sets to the FPGA. All failover functionality is realized through the topology of our MPSoC and in software.

10.3.3 The OBC Multiprocessor System-on-Chip

To realize fault tolerance for our OBC architecture, we isolate software run within our OBC as much as possible and without constraining software design. To do so, we co-designed an MPSoC as platform for the software functionality described in Chapter 4. Its logic placement is depicted in Figure 68, and we will describe its composition here.

We place each processor core within a separate *compartment*. Applications and the environment in which they are executed are strongly isolated through the topology of the MPSoC. The MPSoC version described in this chapter has 4 Xilinx Microblaze processor cores, and therefore 4 compartments, which are depicted in brown, green,

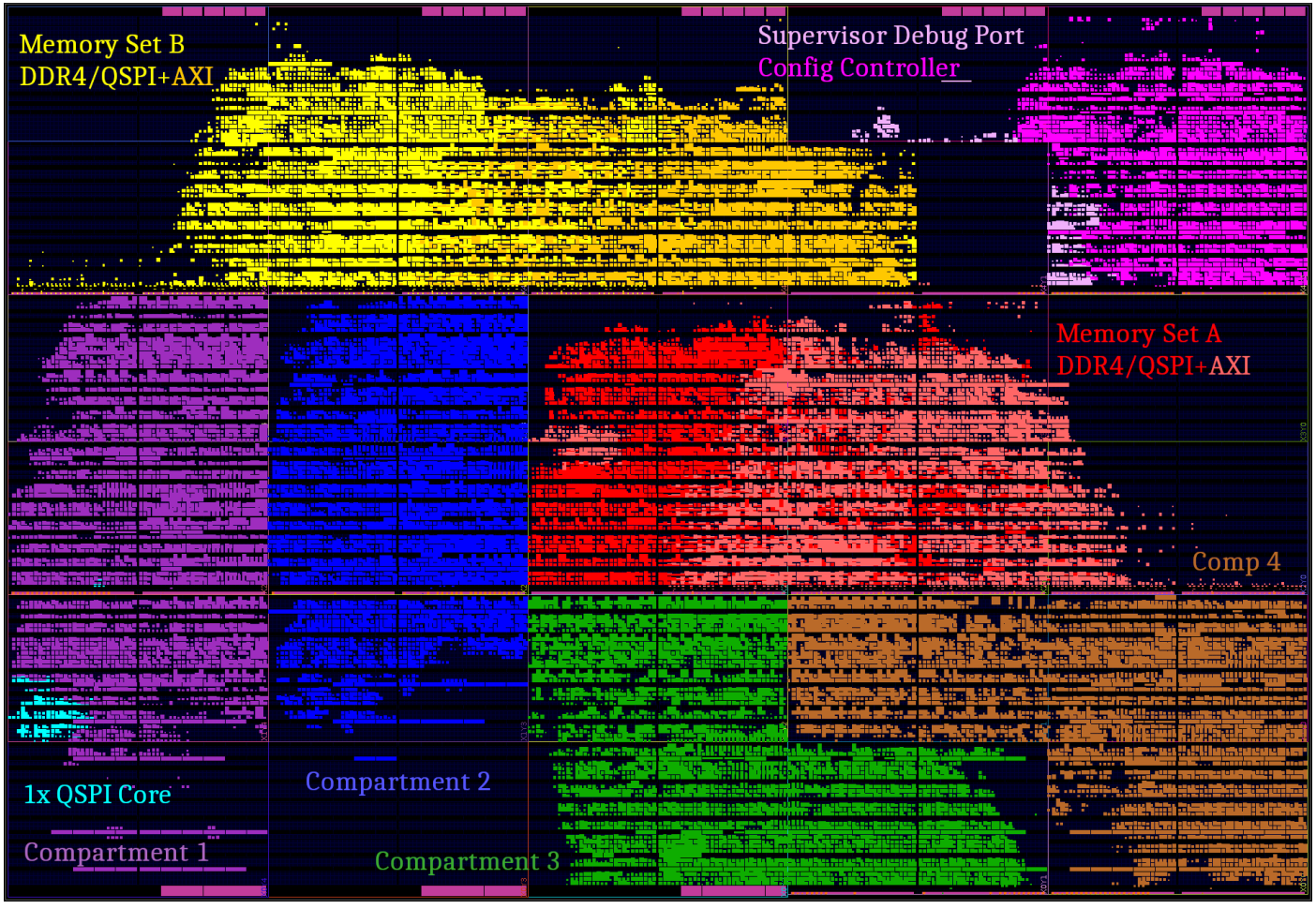


Figure 68: Logic placement of our proof-of-concept quad-core MPSoC for the upcoming XRTC Kintex Ultrascale KU60 device-test board. A QSPI controller is highlighted in teal for size-comparison between an interface core and a compartment's total size.

blue and purple. Compartments have access to two independent memory controller sets through an FPGA-internal high-speed interconnect. The two memory controller sets are depicted in the Figure in red and yellow.

The final, pink-colorized logic segment contains infrastructure IP responsible for FPGA housekeeping, as well as an on-chip configuration controller with access to the FPGA's internal configuration access port (ICAP). As depicted in Figure 69, several MPSoC components related to FPGA housekeeping are placed in static logic:

- the configuration controller makes up only a minor part of the pink-indicated logic,
- the supervisor's debug interface (further described in Section 10.3.4),
- as well as a library IP core facilitating CRAM-frame ECC for the detection and correction errors in the FPGA's running configuration (Xilinx Soft Error Mitigation IP – SEM [354]).

Researchers showed in related work [355, 356] that faults within an FPGA can effectively be resolved through reconfiguration, or mitigated using alternatively routed and placed configuration variants [105]. Usually, full FPGA reconfiguration would interrupt the operation of the MPSoC, and depending on the configuration memory used, can require considerable time. By using partial reconfiguration, we can instead split the MPSoC into separate partitions, which can then be independently reconfigured. The use of an on-chip reconfiguration controller drastically improves the reconfiguration speed, but also allows fine-grained fault analysis and configuration error scrubbing. Multiple alternative partition designs can be provided for each compartment and memory controller set, which can then be reconfigured independently. This not only allows non-stop operation, but also increases the likelihood that a suitable combination of partition variants can be found to mitigate permanent faults present in the FPGA fabric [105].

Compartments and memory controller sets are placed in dedicated partial reconfiguration partitions. Partial reconfiguration allows us to test and repair individual compartments, and to reprogram one memory controller set transparently in the background, without affecting the remaining system. We have implemented this concept in prior research in Chapter 5 for the MOVE-II CubeSat.

Placement in static logic instead of a partition implies that infrastructure logic is not part of any partial reconfiguration partition, which is required both for SEM and logic utilizing ICAP. In practice approximately 90% of the fabric's area is part of the reconfiguration partitions, of which 75% is quadruple-redundant and part of a compartment supporting TMR operation through software. The other 25% of the logic holds the shared memory controllers, which offers simple redundancy and can be recovered transparently using partial reconfiguration. Only 10% of the fabric holds static logic, which can be still be recovered through reconfiguration.

Large clock trees and reset networks are known to be problematic in space applications [357]. The logic in each compartment resides in a separate clock domain, and a memory controller set in 3 – one each for DDR4 backend, memory controller front-ends, and AXI-interconnects. Therefore, clock trees are isolated from each other and are de-coupled on the AXI interconnects of the memory controller sets. This minimizes clock skew and its impact, as well as temperature-related effects, while improving timing and logic routing.

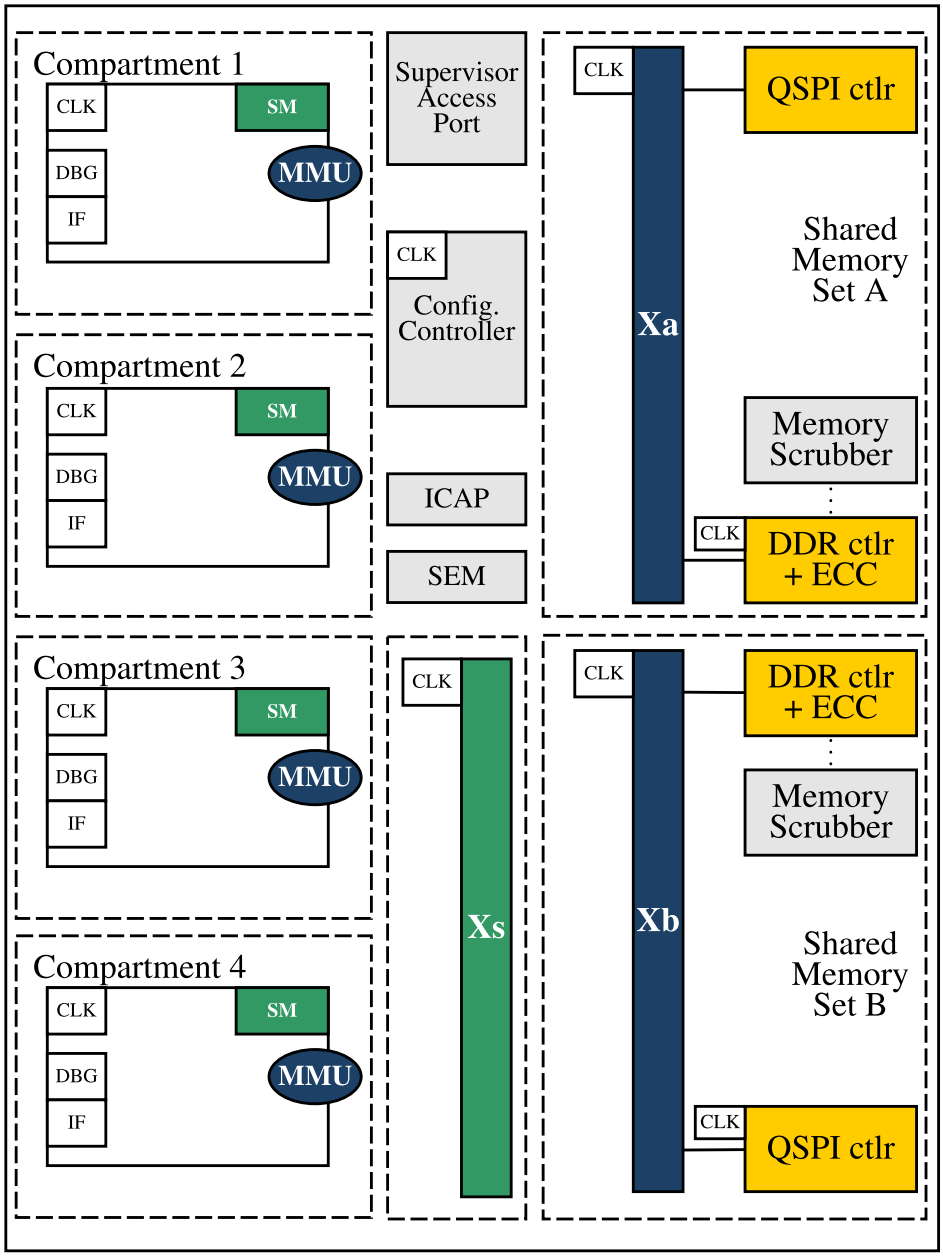


Figure 69: Block-level layout in our MPSoC including clock-placement. Partial reconfiguration partitions are indicated with dashed lines. Compartment and memory controller sets ($X_{a/b}$) can be reconfigured without interruption. The state-exchange interconnect (X_s) resides in a dedicated configuration partition, but during reconfiguration compartments can no access state information. In practice, this results in an interruption of the MPSoC, which can be avoided using a NoC instead of a AXI interconnect.

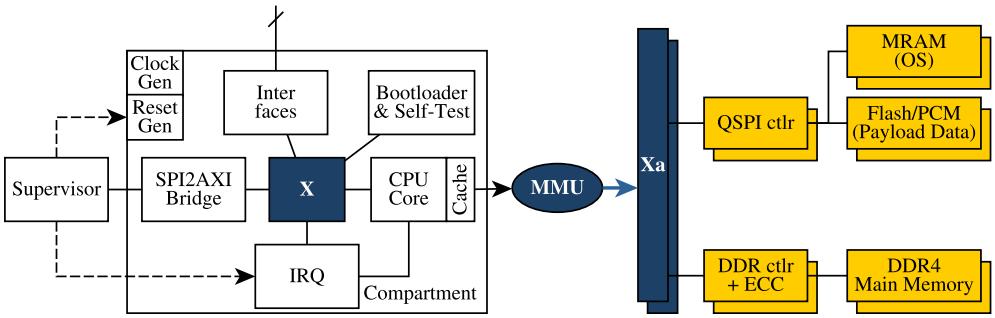


Figure 70: The memory and logical topology of a compartment in a quad-core MPSoC. The compartment local and the global memory controller interconnects are logically isolated. A compartment's processor core has access to the memory controller sets and to compartment-local controllers. Access to compartment-local controllers bypasses the cache.

Compartments are comprised by the minimum set of IP-blocks required for a conventional single-core SoC, including interrupt controller, peripheral controllers, I/O, and bring-up software. A compartment is conceptually similar to a tile in a ManyCore architecture, which are today widely used for compute acceleration and payload data processing [205]. However, their functionality is different, as a ManyCore compute-tile usually is constrained to run simple software, without supporting interrupts, inter-process communication, and I/O. A compartment instead runs a full copy of a general-purpose OS with rich software, has access to hardware timers, interrupts, may preform inter-process communication freely, and can handle I/O autonomously. Besides an on-chip memory holding the bootloader, it is also outfitted with a dedicated dual-port state-memory used to exchange lockstep information. The topology of a compartment is depicted in Figure 70. Each compartment is outfitted with a diagnostic access port, which enables low-level access to a compartment's internal logic through an SPI2AXI bridge. This facility is further described in Section 10.3.4.

In general, for the sake of reliability, the use of SPI or I2C based satellite bus architectures is in general discouraged. However, in Chapter 9, we showed how the interfaces of multiple compartments can be concentrated to emit only a correct result to the satellite bus. Indeally, a network-based satellite-bus should be implemented, which has been shown to be more robust to failures aboard CubeSats of all sizes. If an on-board network is available, no interface-concentration measures are needed, as the network can take care of data de-duplication and can assure that data from a faulty compartment is not propagated. See also [94], for an excellent example of how this can be done while providing real-time guarantees.

On-chip memory controllers used across our MPSoC are implemented in BRAM, which in turn consists of SRAM. Xilinx library IP offers ECC for caches and on-chip memories to detect and correct faults. We utilize Hsiao ECC to protect the data stored in these memories due to its lower logic footprint and otherwise comparable performance as compared to Hamming coding. Due to the brief lifetime of data in caches and buffers, no scrubbing is necessary and the overhead induced through ECC would be detrimental to the overall robustness of the system. Instead, faults in these components are mitigated in software, as described in Chapter 4. To avoid accumulating errors in a compartment's bootloader, we can attach an error scrubber to each

compartment's local interconnect, which is managed by each compartment.

To protect the running configuration of our SRAM-based FPGA, we implement CRAM-frame ECC using the Xilinx Soft Error Mitigation IP (SEM [354]). However, configuration-level erasure coding and scrubbing can still only detect faults in specific components of the FPGA fabric (e.g., not in BlockRAM). We address this limitation at the system level: Our coarse grain lockstep functionality enables us to detect faults in the fabric with compartment granularity within 1-3 lockstep cycles, which is further discussed in Chapters 4 and 5. In practice, this closes the fault-detection gap left by scrubbing and configuration erasure coding.

Each memory controller set consists of a DDR4 memory controller, a QSPI controller, a set of clock and reset generators, as well as an optional memory scrubber core and the top-level AXI crossbar. The optional memory scrubber cores can be controlled by the supervisor to avoid potential interference by malfunctioning compartments.

Each compartment has full write access to a segment DDR memory, while it can access the DDR memory in its entirety read-only. We construct the interconnect used by compartments to access a controller set from an AXI crossbar and four AXI switches, one for each compartment. The top-level crossbar is connected to the area-optimized AXI interconnect attached to each compartment, which makes up the second level of the MPSoC's interconnect. In each interconnect, we realize memory protection for the address space of the relevant compartment to avoid a single point of failure causing misdirected write access. Thereby, we create a topology that strongly isolates compartments from each other, and assures non-interference between compartments.

The address space of all compartments is uniform, enabling memory structures to be migrated between compartments and re-used. Through the MMU component indicated in Figures 70 and 69, we perform the necessary address translation operations.

In case one memory controller set fails, MPSoC compartments that were using this set will switch to failover through a reboot. Compartments that are already utilizing the secondary set can continue executing correctly and provide non-stop operation. Hence, it is desirable to run two of the MPSoC's compartments off the A-controller set, and the rest off the B-set. This allows the software-implemented fault tolerance functionality to guarantee non-stop operation even if an entire memory set would fail. In our proof-of-concept, we realize this functionality by outfitting compartments to be able to use two kernel variants, of which one booting into with main memory in the A set, and the second one into the B set. However, there are more elegant ways to accomplish this, e.g., using position-independent firmware images [358].

To efficiently perform lockstep state comparison and synchronization between compartments, an MPSoC has to provide adequate means of exchanging state-data, as discussed also in Chapters 4 and 9. For small MPSoCs with less than 6 cores, this is realized in DDR/SDRAM memory. For larger designs, a dedicated state-exchange network improves performance and offers stronger isolation. These components are depicted in green in the figures. Access to state memory then takes place entirely on-chip without passing through caches, and the global interconnect.

10.3.4 The Supervisor-FPGA Interface

The supervisor can access the FPGA through the FPGA's JTAG interface. JTAG in principle is powerful which can be used as a universal tool to interact with the FPGA and its MPSoC, and manipulate it in a variety of ways. However, JTAG TAPs can be

very complex, and the protocol does not assure the integrity of transferred data, while binary data transfer via JTAG can be very slow. Hence, we only use it to reconfigure the FPGA in case the on-chip configuration controller fails.

The supervisor can trigger an interrupt or permanently disable a compartment, and can induce a reset in compartments, memory controller sets, for the configuration controller, and for the FPGA itself. This is realized through a set of GPIO pins attached to the supervisor. The supervisor can conduct low-level diagnostics and has access to each compartment's address space, without having to rely upon a compartment's processor core.

We realize high-speed interconnect access through SPI, as the CubeSat community is already familiar with this type of interface. As we just required a direct point-to-point between the FPGA and the supervisor without chip select, this interface setup on the PCB-side is very simple. We attach an SPI2AXI bridge to each compartment's local interconnect, and additionally to each memory controller set. This SPI-bridge can be assembled entirely from well tested, free, open-source IP available in the GPL version of GRLIB, using the SPI2AHB and AHB2AXI IP cores. Alternatively, a variety of open-source SPI2AXI cores are available, e.g., on gitlab, but the quality of these cores is uncertain. Xilinx and other vendors offer a selection of commercial IP cores.

The supervisor also communicates with the FPGA-internal configuration controller, which is outfitted with a conventional SPI-slave interface. In contrast to the SPI-diagnostics setup used for accessing the interconnect of compartments and memory controller sets, the configuration controller actively collaborates with the supervisor. The configuration controller communicates with SEM and can be deactivated by the supervisor in case of failure. During normal operation, it will notify the supervisor about faults in the FPGA fabric. It can then perform reconfiguration via ICAP. The satellite developer can therefore deposit multiple differently placed designs for each partition in configuration memory, which the configuration controller can attempt to use to resolve a fault. Finally, the configuration controller will report outcome of the repair attempt to the supervisor.

Architecturally, the configuration controller resembles a stripped-down compartment design, but is constrained to a minimal logic footprint in the following way:

- It can run only baremetal code or an RTOS, not a general-purpose OS, thereby reducing the controller's logic footprint.
- This software is stored directly in on-chip BRAM which is part of the reconfigurable fabric.
- It has no access to the memory controller sets, to prevent interdependence between static logic and partial-reconfiguration partitions.
- Besides its SPI master connected to configuration memory, the configuration controller has no other external interfaces.

In case of failure, the supervisor can substitute the full set of the configuration controller's functionality through JTAG, and can recover it through full-FPGA reconfiguration.

As depicted in Figure 71, the supervisor can utilize it's SPI interface to access the different components of the MPSoC in a controlled and performance-efficient manner.

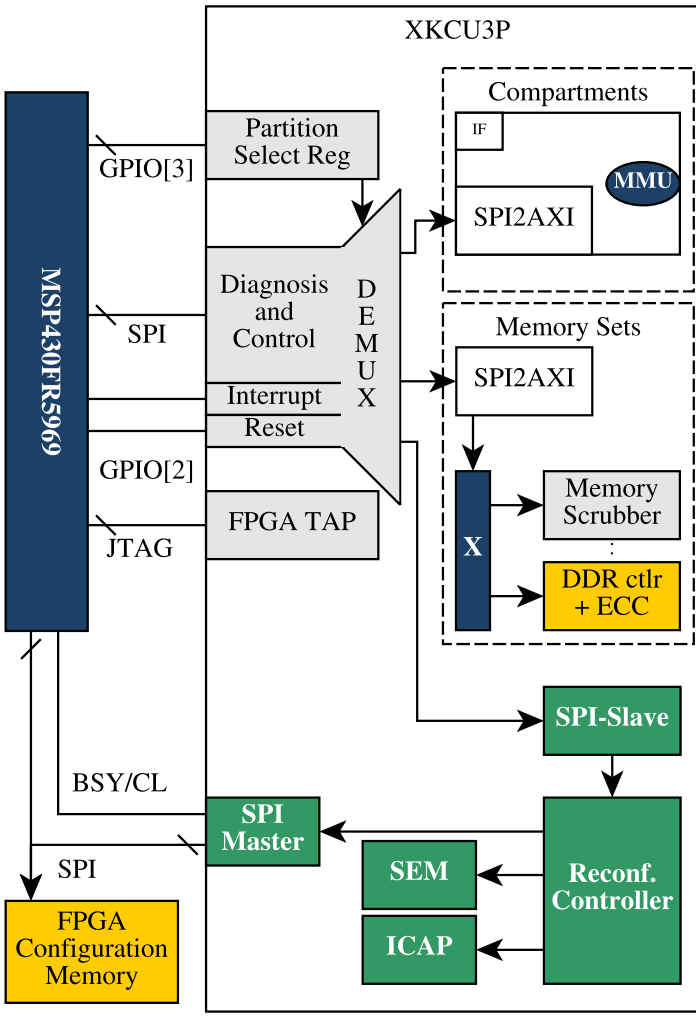


Figure 71: The design of our supervisor-FPGA control and diagnostic interface including the debug-facilities used by the supervisor to access different compartments of the MPSoC.

It can disable individual compartments in case of failure by using existing circuitry required for partial reconfiguration, as indicated in Figure 70. However, instantiating the combination of SPI, reset, and interrupt lines for each compartment, memory set, and the reconfiguration controller would require a large amount of IO-pins. In practice, the supervisor will only communicate one MPSoC component at any given time, and never with multiple concurrently. Hence, we de-multiplex (DEMUX) this interface, thereby reducing the need for I/O resources to just an SPI interface and 5 GPIO lines.

10.4 Handling Chip-Level SEFIs and Failure

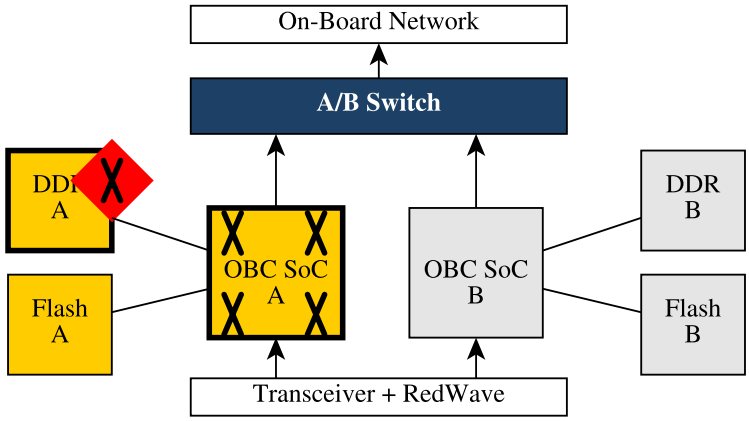
Our proof-of-concept MPSoC design spans only of a single FPGA and is not designed to withstand component-wide SEFIs affecting the entire FPGA. However, it can be implemented to tolerate such faults and even full component failure.

Figure 72a depicts an idealized traditional A/B-failover system with I/O switching. Such a system can tolerate the failure of components in either the A or the B side, but fails if an additional fault occurs elsewhere in the system. The B-side of the system remains inactive until a fault has been detected and isolated, and can be used productively without further design measures in hardware. Due to failover being implemented at the component level in hardware, additional glue logic required for switching between the A and B-system. It is usually not possible to test the failed side without further design measures, and tests can only be conducted if the system is taken offline. These limitations can be worked around with more glue logic and a more complex failover implementation, but even then the relevant logic can usually not just be be turned off and bypassed. Instead, it remains a potential failure source.

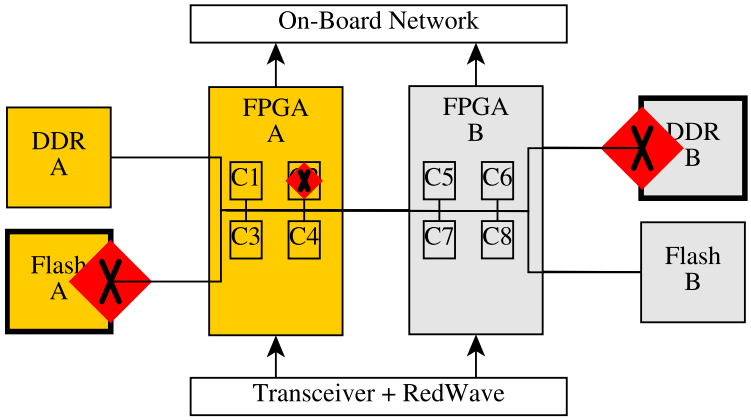
The system depicted in Figure 72b implements our architecture on two FPGAs and does not suffer these limitations: Instead of implementing all compartments and shared memory controller sets on a single FPGA, they can be distributed across multiple FPGAs. The chip-to-chip AXI IP used to connect two or more FPGAs is available in the Vivado IP library. The failure of, e.g., a memory component connected to one FPGA, does not cause the failure of an entire redundant system side. Compartments on one FPGA connected to a failed component can still access components on the B-side. The supervisor and platform controller on the faulty side can then reconfigure the relevant FPGA partitions, and conduct further analysis on the failed components. The system can thus continue thus support non-stop operation in case of severe component failure, if threads-replicas are distributed so that not all replicas of a thread are executed on the same FPGA. In a TMR setup, this enables non-stop operating, e.g., with the A-side running 2 replicas on one FPPGA and the B side running the third replica on the other. In NMR setups, two replicas can be assigned to each side, allowing fault-detection even if one of the FPGAs has failed during the same lockstep cycle. For diagnostic purposes, thread-replication and therefore fault tolerance can also be constrained temporarily or even fully disabled. Even a severely degraded system implementing our architecture that has suffered multiple component failures can thus still operate correctly and support non-stop operation. In contrast to a traditional OBC based on component-redundancy, our architecture thus can delivers stronger fault tolerance capabilities than traditional OBCs. As compartments on different FPGA can share resources, this allows for increased efficiency and performance as compared to traditional systems.

To support larger MPSoCs with more than 8 compartments efficiently, a more

scalable interface between compartments and memory controller sets should be used. This can be achieved with a Network-on-Chip (NoC). A NoC allows drastically larger MPSoC designs [329] due to improved scalability, but also enables fault-tolerant routing [349], backwards error correction (re-transmission), and quality-of-service support [359]. When implementing our architecture with a NoC, the shared memory controller sets would be implemented as one NoC layer, while the state-exchange network forms a second layer. In contrast to conventional interconnects typologies, a NoC can also utilize error correction for NoC routers [93].



(a) A traditional redundant system where there A-side failed due to malfunction in one memory components, which will fail once a fault occurs on the B side.



(b) Our architecture, which is still functional and not degraded, even though multiple components have failed on both sides.)

Figure 72: Fault tolerance examples of a traditional OBC and our architecture, which shows that our architecture can tolerate a much increased number of faults than a traditional system.

10.5 Utilization and Power Comparison

The quad-core MPSoC architecture described in this chapter was implemented on a set of Kintex Ultrascale and Ultrascale+ devices using Xilinx Microblaze soft-cores running at 300MHz, and DDR4 controllers. In our proof-of-concept, we utilize a FeRAM-based MSP430FR5969 controller for our proof-of-concept, for which a low-cost space-grade substitute is available. The MPSoC is reproducible in Xilinx Vivado 2017.1 and later. The necessary IP is included in the Vivado IP library, and can be obtained free of charge through Xilinx's university program by academics and non-commercial scientific users. This serves as proof-of-concept for our architecture, with resource utilization indicated in Table 9.

For this Microblaze-based MPSoC implementation, the added logic footprint for instantiating a compartment as compared to just an application-processor without any peripherals is low. For size comparison between an interface IP-core and a compartment, a QSPI controller core is highlighted in Figure 68 in teal. It makes up only 2.5% of a compartment's LUT and 6% BRAM utilization, with other commonly used cores aboard CubeSat such as I2C or UART showing a similar or even lower footprint. The larger size of ARM Cortex-A53 processor cores reduce this ratio even further.

Our initial proof-of-concept was implemented on the Xilinx Virtex Ultrascale+ VCU118 Evaluation Kit with DDR4 controllers running at 1600MHz. This FPGA family was ideal for design space exploration as the kit has two DDR4 memory channels and a large fabric. Within the Xilinx Radiation Test Consortium we are currently working on a Kintex Ultrascale KU60/XQRKU060 test board for radiation testing, to which we ported our design. Logic and partition placement are depicted in Figure 68. FPGA utilization and power consumption tables are indicated in Tables 9 and 10. On KU60, DDR4 memory controllers run at 1000MHz due to generational constraints.

We ported our MPSoC also to smaller Kintex Ultrascale+ devices, the KU60's closest equivalent part KU11P and the smallest FPGA in the family and generation,

	KCU3P		KCU11P		KCU60	**(XRTC)**
Resource	Used	**% Total**	Used	**% Total**	Used	**% Total**
LUT	85505	**52.55%**	87187	**29.20%**	132359	**39.91%**
LUTRAM	9319	**9.33%**	9632	**6.49%**	19536	**13.30%**
FF	93766	**28.81%**	96043	**16.08%**	158617	**23.91%**
BRAM	**303.5**	**84.31%**	303.5	**50.58%**	316	**29.26%**
DSP	30	**2.19%**	30	**1.02%**	30	**1.09%**
IO	224	**73.68%**	224	**43.75%**	378	**60.58%**
BUFG	21	**8.20%**	22	**3.20%**	26	**4.17%**
MMCM	**2**	**50.00%**	2	**25.00%**	2	**16.67%**
PLL	**7**	**87.50%**	**9**	**56.25%**	13	**54.17%**

Table 9: Resource utilization our MPSoC on different Xilinx Kintex FPGAs. The XRTC variant's DDR4 memory controllers has a larger data-width due to package constraints. Design constraining fabric-resources are marked in bold.

FPGA	XKCU3P	XKCU11P	XKCU60
FPGA Generation	Ultrascale+	Ultrascale+	Ultrascale
Technology Node	16nm FinFET	16nm FinFET	20nm Planar
Part Package	SFVB784-I	FFVE1517-I	FFVA1517-I
Clocks	0.23W	0.29W	0.71W
Signals	0.11W	0.15W	0.30W
Logic	0.11W	0.15W	0.42W
BRAM	0.19W	0.19W	0.41W
DSP	<0.01W	<0.01W	<0.01W
PLL	0.37W	0.46W	0.72W
MMCM	0.23W	0.23W	0.21W
I/O	0.27W	0.34W	1.50W
Dynamic Power	**1.51W**	**1.81W**	**4.26W**
Static Power	**0.44W**	**0.70W**	**0.67W**
Total Power	**1.94W**	**2.51W**	**4.93W**

Table 10: Power consumption of the 3 quad-core MPSoC implementations. Data generated by Xilinx Vivado 2018.3's Implementation Power Report.

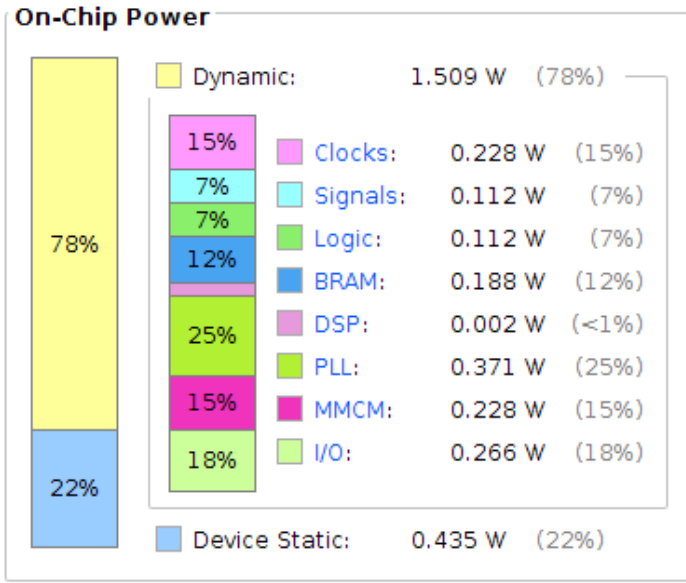


Figure 73: Power consumption of the 4-core MPSoC powering our MPSoC implemented on XCKU3P. Figure generated by Xilinx Vivado 2018.3.

the KU3P. The port required minor adjustments to the utilization of clocking resources, as both KU11P and KU3P have fewer clocking-resource (MMCM and PLL tiles) than the KU60. On KU11P, it was sufficient to switch several clock-generators used in the shared memory controller sets from PLL to MMCM tiles, without changing other parameters. The main constraint of the KU3P, however, required a reduction of clock generators in memory controller sets to 1 clock domain instead of 3 as described in Section 10.3.3. Due to the much smaller fabric of the KU3P, clock-domain sizes and routing distances decrease, resulting better timing of the design.

Despite much lower dynamic power consumption across the board in Ultrascale+, the KU11P variant shows slightly higher static power consumption than the KU60, which is counterintuitive. After discussion within the Xilinx Radiation Testing Consortium, the most plausible explanation for this anomaly is the different IO-bank placement within the fabric between these devices. On KU60, IO-banks are placed in more favorable locations considering MPSoC design than on KU11P. This increases

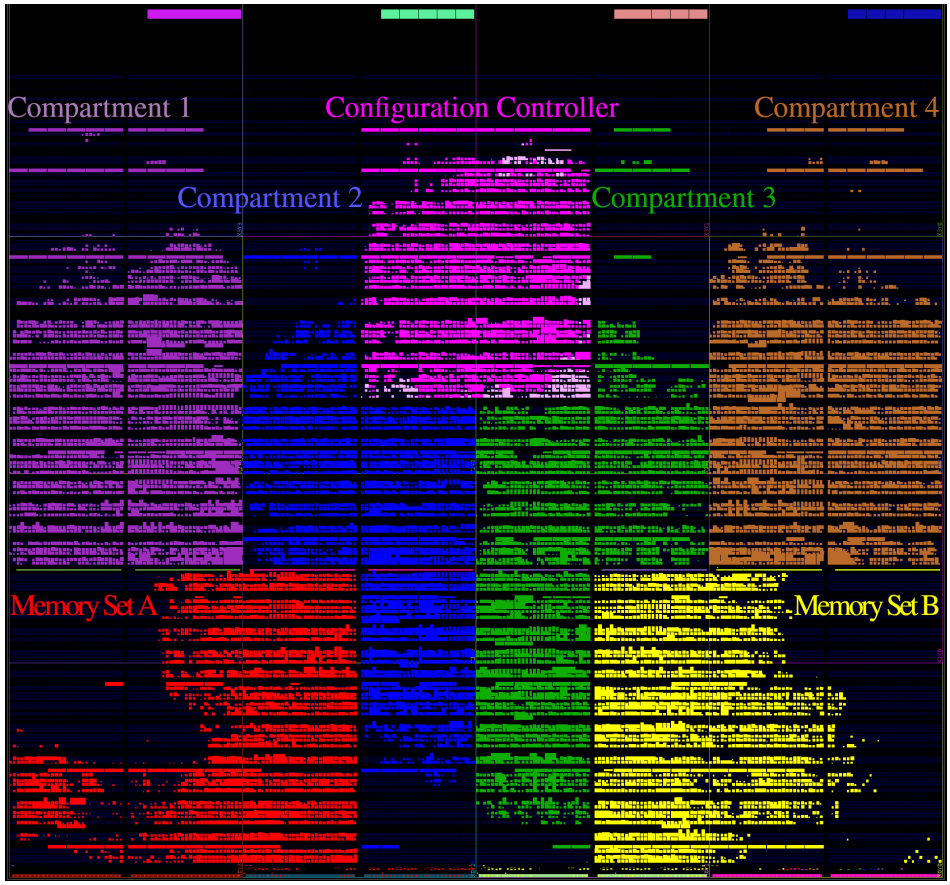


Figure 74: Logic placement of our proof-of-concept MPSoC on a Xilinx Kintex Ultrascale+ KU3P with 4 compartments (purple, blue, green, and brown), two shared memory controller sets (red & yellow) and static logic (pink). In contrast to the KU60 implementation, DDR controllers of this designs have reduced data width.

logic-spread, leaving less fully inactive fabric sections, which could explain an increase in static power consumption due to infrastructure on KU60.

The resulting Ultrascale+ MPSoC implementations, while functionally equivalent, show a 50% lower power consumption than the previous generation. This is due to manufacturing in a 16nm FinFET technology node instead of 20nm planar. Power savings mainly come from a reduced dynamic power consumption of this design, due to an increased degree of logic concentration in a smaller of FPGA-fabric area. For CubeSat-use, the Kintex Ultrascale+ family is therefore more attractive, despite the potential risk of IO-pin latch-up is acceptable [299] which today is mitigated in this field through the system-level measures [39]. On the the smallest Ultrascale+ part and most compact BGA package xcku3p-sfvb784 available at the time of writing, we achieved 1.94W total power consumption. This is well within the power budget range of 2U CubeSats. Vivado's power report for this design is depicted in Figure 73.

Synthesis was run in "Alternative Routability" mode, while implementation was with the "Performance-Explore" strategy with post-route placement & power optimization, as the resulting implementations showed consistently better timing and power utilization.

10.6 Experimental Results and Testing

We have tested our proof-of-concept OBC on Xilinx VCU118 (with 2 DDR memory channels) and KCU116 boards (with 1 channel due to board constraints), and constructed a breadboard setup in conjunction with an MSP430FR development board. Further information on this designs is available in Chapter 9, with an MPSoC implementation paper currently undergoing peer review. The actual platform for our research has been the ARM Cortex-A53 application processor, which is today widely used in a variety of mobile-market devices and certain COTS CubeSat OBCs. The architecture we presented in this chapter is processor and platform independent, with the MPSoC presented here implemented using Xilinx Microblaze processor cores.

To test our implementation, we have conducted fault injection through system emulation into an RTEMS implementation of Stage 1 running on a Cortex-A processor. In 2019, we also constructed a multi-core model of our MPSoC also in ArchC/SystemC on RISC-V to conduct further fault-injection close-to-hardware. The results of this fault-injection campaign are documented in Chapter 8. They show that with near statistical certainty, a fault affecting a compartment can be detected within 1–3 lockstep cycles, demonstrating that Stage 1 is effective and works efficiently.

10.7 Conclusions

In this chapter, we presented a CubeSat compatible on-board computer (OBC) architecture that offers strong fault tolerance to enable the use of such spacecraft in critical and long-term missions. It is the result of a hardware-software co-design process, and utilizes fault tolerance measures across the embedded stack. We described in detail the design of our OBC's breadboard layout, describing its composition from the component-level, to the MPSoC implementation used, all the way down to the software level. We implement fault tolerance not through radiation hardening of the hardware, but realize it in software and exploit partial FPGA-reconfiguration and

mixed criticality. To implement and reproduce this OBC architecture, no custom-written, proprietary, or protected IP is needed. All COTS components required to construct this architecture can be purchased on the open market, and are affordable even for academic and scientific CubeSat developers. The needed designs are available in standard FPGA-vendor library logic (IP), which in most cases is available to academic developers free of charge through university donation programs.

Overall, our OBC architecture is non-proprietary, easily extendable, and scales well to larger satellites where slightly more abundant power budget is available. We successfully implemented a proof-of-concept of our MPSoC for a variety of Xilinx Kintex and Virtex Ultrascale and Ultrascale+ FPGA. This MPSoC was implementable even for the smallest Kintex Ultrascale+ FPGA, KU3P, and we achieved 1.94W total power consumption. This puts it well within the power budget range available aboard current 2U CubeSats, which currently offer no strong fault tolerance.

A comparison to existing traditional space-grade solutions as well as those available to CubeSat developers seems unfair. Today, miniaturized satellite computing can use only low-performance microcontrollers and unreliable MPSoCs in ASIC or FPGA without proper fault tolerance capabilities. Using the same type of commercial technology, our OBC can assure long-term fault coverage through a multi-stage fault tolerance architecture, without requiring fragile and complex component-level replication. Considering the few more robust, low-performance CubeSat compatible microcontrollers, our implementation can offer beyond a factor-of-10 performance improvement even today. Considering traditional space-grade fault-tolerant OBC architectures for larger spacecraft, our current breadboard proof-of-concept implemented on FPGA exceeds the single-core performance of the latest generation of space-grade SoC-ASICS such as an GR740. However, it does so at a fraction of the cost of such components, and without the tight technological constraints of traditional or ITAR protected space-grade solutions.

Traditional fault-tolerant computer architectures intended for space applications struggle against technology, and are ineffective for embedded and mobile-market components. Instead, we designed a software-based fault tolerance architecture and this MPSoC specifically to enable the use of commercial modern semiconductors in space applications. We do not require any space-grade components, fault-tolerant processor designs, other custom, or proprietary logic. It can be replicated with just standard design tools and library IP, which is available free of charge to many designers in academic and research organizations.

Our architecture scales with technology, instead of struggling against it. It benefits from performance and energy efficiency improvements that can be achieved with modern mobile-market hardware, and can be scaled up to include more, and more powerful processor cores. At the time of writing, Xilinx has begun to introduce a new generation of FPGA-equipped devices manufactured in a 7nm FinFET+ technology node, in which the design issue causing latch-up in Ultrascale+ could also have been mitigated [299]. Xilinx's foundry TSMC expects this manufacturing process to offer approximately 65% reduction power consumption as compared to the 16nm FinFET node used for Ultrascale+ FPGAs [360]. Even if only half of this expected power reduction would manifests, in combination with FPGA-fabric optimizations, we can expect to achieve approximately 1W power consumption with our MPSoC implemented on a next-gen Xilinx FPGA. While these expectations based on experiences with the current 20nm Planar and 16nm FinFET manufactured Xilinx FPGAs, future FPGA

generations released within the next decade will, with near certainty [361], allow our architecture to even become usable aboard 1U CubeSats.

At the time of writing, each component of our OBC architecture has been implemented and validated experimentally to TRL3 in a 1-person PhD student project. From each individual component, we have assembled a development-board based breadboard setup. As next step in validating this new OBC architecture, we will construct a prototype for radiation testing. Since 2018, we have therefore contributed to the Xilinx Radiation Testing Consortium to develop a suitable Kintex Ultrascale-equipped device-test board. This will bring our architecture to TRL4, and is an intermediate step before developing a custom-PCB based prototype for on-orbit demonstration. Once this has been achieved, we intend to perform the final step in validation of this technology aboard a CubeSat.

Chapter 11

Conclusions and Outlook

11.1 Conclusions

RQ1 In this thesis, we presented a satellite on-board computer (OBC) architecture that can offer strong fault tolerance with conventional, low-cost, modern semiconductors manufactured in small feature-size technology nodes. The correct functionality of this architecture is safeguarded through a set of inter-linked software-implemented fault tolerance measures combined with FPGA reconfiguration, which we described in Chapter 4. These concepts allow us to assure fault tolerance even for satellites with a very small form factor, which today can only utilize primitive or no fault tolerance measures at all, as traditional radiation-hardened satellite computer solutions can not be utilized due to volume, mass and power restrictions. We showed that through lockstep implemented in software, we can efficiently protect a system consisting of embedded and mobile-market components, and should ideally be implemented within an FPGA to exploit reconfiguration. We demonstrate that the performance cost of this lockstep mechanics is economical, and that its implementation is possible in a non-invasive manner. Its protective guarantees are run-time configurable, and fault tolerance can even be entirely deactivated at runtime if so desired.

RQ2 In Chapters 4 and 5, we showed that the logic of an FPGA-implemented MPSoC can be protected well from radiation effects through smart configuration management and off-chip diagnostics. We closed the fault-detection gap which prior research struggles to close through the multi-stage fault tolerance architecture described in Chapter 4. To safeguard an FPGA from transient faults, we showed that error scrubbing and FPGA reconfiguration can be used to detect and correct bit-upsets in the CRAM of an FPGA. As described in Chapter 4, permanent faults can then be mitigated through reconfiguration with alternative partition variants. This not only increases the capability to cover permanent faults, but as we show in Chapter 5, it also allows an OBC to adapted to the specific requirements during each phase of complex, multi-phased space missions. This allows a reduction of overall system complexity, reduces the need for spare processor cores and MPSoC infrastructure logic, and can drastically extend the lifetime of a COTS FPGA-based OBC.

RQ3 In space missions with a very long duration, parts of an FPGA's fabric will eventually no longer be recoverable through reconfiguration. This is due to accumulating permanent faults in the semiconductor the FPGA, and thus also the

MPSoC, are implemented in. Over time, this will result in an increasing number of the MPSoC's processor cores becoming unusable, gradually reducing the amount of processing time available to the lockstep, and the level of replication it can achieve for all applications. In Chapter 6, we showed that the run-time configurable nature of software-implemented fault tolerance enables an OBC to respond to this behavior in a way that can best be described as "graceful aging". By exploiting mixed criticality, it is possible to autonomously reallocate processing time between the different applications that are part of an OBC's flight software, allowing us to safeguard fault-tolerant operation for the flight software's core functionality. We showed that stability and availability of critical applications can be maintained by sacrificing performance of less important applications. In practice, this allows an OBC to age gracefully and adapt to a shrinking set of intact processor cores, instead of failing spontaneously as traditional systems do. A satellite operator can use this functionality to prioritize and dynamically trade system performance for increased fault coverage, power saving, or to maximize an OBC's functionality. Spare processor cores in traditional hardware-voting based systems remain idle until a fault occurs, but our lockstep can use them actively to run less critical parts of the flight software, until they are needed in practice to replace a failed processor core. This allow spare processor cores available throughout an MPSoC to be pooled and used more efficiently, thereby to overcoming the static nature of traditional static hardware-implemented fault tolerance measures. This allows an OBC to offer stronger fault coverage, and to more efficiently meet the changing performance requirements throughout complex multi-phased solar system exploration missions with much reduced over-provisioning and without requiring idle spares.

RQ5 All these operational and system-design improvements are possible due to the coarse-grain lockstep concept described in Chapter 4, which we utilize to achieve forward error correction. We implement this lockstep within the OS kernel of an operating system (RTEMS, FreeRTOS, and experimentally also on Linux) or as part of baremetal software, where it assures synchronization between multiple thread-replicas run on the processor cores of an MPSoC. To test and validate our architecture, in Chapter 8, we conduct fault-injection into an emulated system and into a SystemC-implemented MPSoC model. In this chapter we describe the two fault injection campaigns we conducted against implementations of our lockstep: In the first campaign, we utilized the QEMU-based FIES fault injection framework to inject faults into an RTEMS implemented variant of our lockstep run on a Cortex-A system. In the second campaign, we modeled a triple-core model of our MPSoC using RISC-V cores in ArchC, and injected faults using SystemC simulation. Few software-implemented fault tolerance concepts described in literature have been practically implemented and validated. Therefore this chapter is also intended as practical guide for fellow researchers, to make proper testing of software-implemented fault tolerance measures less challenging and time consuming.

RQ4 Relying on software-implemented fault tolerance measures also require special care to be taken to assure the integrity of the flight-software in which they are implemented. Hence, in Chapter 7, we explored how unprotected volatile and non-volatile COTS memory can be retrofitted with strong error correction and protected from bit-upsets and SEFIs in control logic. We showed that error scrubbing for volatile memory can be combined with allocation-time integrity checking and blacklisting for defective pages in widely-used operating systems such as Linux. To safeguard the logic

of our lockstep and a full firmware image, we showed that a file system can be equipped symbol-based erasure coding and can use memory protection to mitigate the impact of faults in control logic. To protect payload data, we described that a composite erasure coding system can be combined with RAID-like functionality to efficiently protect data stored within high-density NAND-flash and phase change memory. We showed that software measures can guarantee strong fault tolerance, the NAND-flash industry has in even begun to adopt the same erasure coding systems we proposed in this paper as part of a solid-state drives embedded software-stack, e.g., in [286]. Simple erasure coding for caches and other on-chip memories at the time of writing is a standard feature in Xilinx library IP, and supported in all currently available model-market devices [119]. Security vulnerabilities such as Rowhammer and an increased need for yield enhancement have prompted the adoption of ECC also for protecting main memory [362], and in combined with software-implemented memory testing and scrubbing described in this chapter, sufficient protection can be assured even for LEO CubeSat missions with an extended duration of 2-5 years.

RQ6 Much of today's fault tolerance research proposes interesting and novel concepts. But in practice, the majority of these concepts can not be applied to protect a critical system as it exists in the real world. To show that our architecture is effective in practice, in Chapter 9 we developed an MPSoC design which provides an ideal platform for the software-mechanics used to assure fault tolerance. It is the result of a hardware-software co-design process and assures a high-degree of logic and data isolation for software run on the individual processor cores of the OBC within compartments. It can be implemented with just currently available COTS hardware and extensively validated FPGA-vendor library IP, requiring no proprietary logic or costly, custom space-grade processor cores. This design demonstrates that our architecture can not just protect a satellite OBC in theory, but also that a suitable computer architecture is feasible, and that no space-proprietary logic or IP is required.

In Chapter 10, we described the practical implementation of this MPSoC for a variety of Xilinx Ultrascale and Ultrascale+ FPGAs as proof-of-concept. To show how a practical OBC implementation for this MPSoC can look like, we developed a series of MPSoC implementations and a breadboard proof-of-concept of this architecture on Xilinx VCU118 (with 2 DDR memory channels) and KCU116 boards (with 1 channels due to board constraints) in conjunction with TI-MSP430FR development boards. We described the component-level setup of this architecture for CubeSat-use, for which an MPSoC implementation on a KU3P FPGA is possible with just 1.94W total power consumption. This demonstrates that a practical implementation of our architecture can be achieved, which stays well within the power budget range available aboard current 2U CubeSats.

11.2 Discussions

Traditional fault-tolerant computer architectures intended for space applications struggle against technology, and are ineffective for embedded and mobile-market components manufactured in technology nodes with a fine feature size. In this thesis we showed that the solution to this limitation is the use of software-implemented fault tolerance measures, which can be utilized to systematically protect each component of an OBC as depicted in Figure 75. Through the architecture we developed originally as OBC for the MOVE-II satellite, we show that it is possible to efficiently

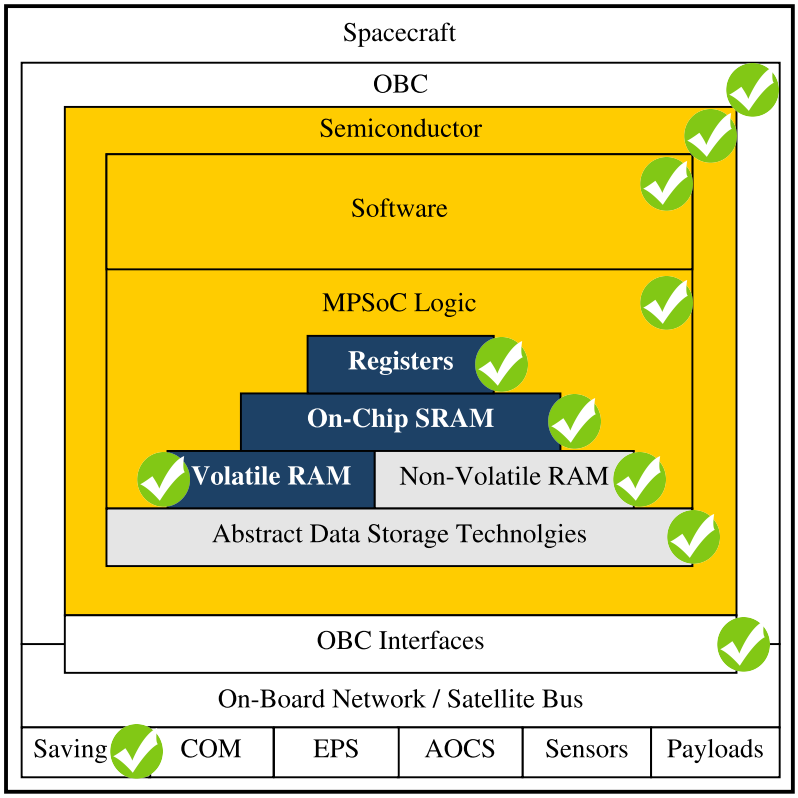


Figure 75: A component-level model of a satellite OBC, components for which the research presented in this thesis offers protection are indicated with checkmarks.

protect modern COTS semiconductors effectively, and make them usable for critical space applications. To realize such an architecture, we do not require any space-grade components, fault-tolerant processor designs, or other custom and proprietary logic. The OBC architecture we developed from this approach can be replicated with just standard design tools and library IP, which are available commercially and even free-of-charge to designers in academic environments. Our architecture scales with technology, instead of struggling against it. It benefits from performance and energy efficiency improvements that can be achieved with modern mobile-market hardware, and can be scaled up to include more, and more powerful processor cores.

In Chapter 10, we showed as practical example that our architecture can achieve beyond 50% power saving even between two generations of Xilinx FPGAs, one being manufactured in 16nm FinFET and the prior generation in a 20nm planar technology node. In this regard, we eagerly await the release of the next generation of FPGAs manufactured in EUV-based technology nodes with 7nm or 5nm feature size. Compared to 16nm FinFET and 20nm planar manufactured devices, we expect that next generation FPGAs manufactured in these technology nodes will offer further power saving, will allow much higher clock frequencies to be achieved for an MPSoC implemented in configurable logic, while the reduced feature size of the semiconductor logic would further reduced the likelihood for radiation to affect.

A comparison of our OBC architecture to traditional space-grade solutions and contemporary CubeSat computing seems unfair. Today, miniaturized satellite developers are limited to use low-performance microcontrollers and MPSoCs implemented in ASIC or FPGA. Considering the few CubeSat compatible low-performance microcontrollers that have been shown robust under radiation, our implementation can offer drastically more performance. At the time of writing Chapter 4, we estimated that our architecture run on modern MPSoC and FPGAs can offer a beyond factor-of-5 performance improvement as compared to these microcontrollers. Since 2017, within a time-span of just two years, mobile market MPSoCs have advanced drastically, and a beyond factor-of-10 improvement seems more realistic. At the time of writing in mid-2019, most mobile-market devices can offer almost twice the clock speed and a better performance per clock cycle as compared to their counterparts in 2017. Same applies to the upcoming generation of FPGA which will benefit greatly from technology scaling.

Mobile-market MPSoCs used aboard CubeSats today seldom include any fault tolerance capabilities. Only sometimes to CubeSat designers implement custom homebrew component-level failover concepts, which has been shown to inflate complexity and failure potential. Our OBC architecture is based upon the same type of commercial technology, but through software-measures and a smart MPSoC design, we assure long-term fault coverage with a component-wise simple setup. Comparing this OBC architecture with traditional solutions for larger spacecraft, even our current FPGA-based proof-of-concept exceeds the single-core performance of the latest generation of space-grade ASICS-SoCs such as an GR740 (250MHz vs 300MHz+). On top of that, our architecture can offer fault tolerance at a fraction of the cost. It can do so without suffering from the tight technological constraints of this classical technology and the archaic development tools used there. All this is possible while still using COTS hardware, without being impacted by the legal constraints of components that are subject to ITAR or other export control laws.

11.3 Outlook and Future Work

As of early 2019, Xilinx has began to introduce a new generation of FPGA-equipped devices manufactured in a 7nm FinFET+ technology node, in which the design issue causing latch-up in Ultrascale+ should be mitigated [299]. With this node, Xilinx's foundry TSMC expects an around 65% reduction power consumption as compared to the 16nm FinFET node used for Ultrascale+ FPGAs [360]. Even if only half of this expected power reduction would manifests, in combination with FPGA-fabric optimizations, we can expect to achieve approximately 1W power consumption with our MPSoC implemented on a next-gen Xilinx FPGA. While these expectations based on experiences with the current 20nm Planar and 16nm FinFET manufactured Xilinx FPGAs, future FPGA generations released within the next decade will, with near certainty, allow our architecture to even become usable aboard 1U CubeSats.

At this point in time, I have validated this OBC architecture to the extent that this is possible for a single researcher in an academic environment. As next step to validate it, I therefore plan to develop a prototype implementation. Since 2018, I have therefore collaborated with and contributed to the Xilinx Radiation Testing Consortium in the creation of a Kintex Ultrascale KU60 device-test card to reduce the cost and time required for constructing this prototype. As of 12.09.2019, we, the

XRTC infrastructure team, have finalized the KU60 card's design and schematics, and after routing and a final review pass, the KU60 DuT-card will go into production later this year.

Once the XRTC KU60 DuT-card becomes available, I plan to implement a matching daughterboard carrying DDR-SDRAM, MRAM, and PCM components as well as a supervisor MSP430FR, to then conduct radiation testing. Radiation testing will then increase the maturity of this architecture to TRL4, and also serves as intermediate step to then realize a full custom-PCB based prototype. This prototype can then for the first time be used to demonstrate the full capabilities of this architecture at TRL5, without the constraints present in a development-based breadboard setup.

There is considerable potential for improvements considering the proof-of-concept that I have developed before and during my PhD: The relaxed cost, energy, and size constraints aboard microsatellites and larger spacecraft would allow an implementation of this OBC architecture spanning multiple FPGAs and with a drastically higher number of compartments. Such an OBC would not only offer better scalability and fault-isolation than a single-FPGA system, but can then also tolerate chip-level defects and SEFIs. Application replicas in lockstep could then be distributed across multiple FPGAs, allowing non-stop operation even if an individual FPGA would have to be reset, if or full reconfiguration is necessary.

To support larger MPSoCs with more than 8 compartments efficiently, a more scalable interface between compartments and memory controller sets should be used. This can be achieved by replacing the 2-level AXI crossbar the MPSoC is built around today with a Network-on-Chip (NoC). A NoC offers improved scalability [329], can also be used to enable fault-tolerant routing [349], backwards error correction through re-transmission, and quality-of-service support [359]. When implementing this architecture with a NoC, the shared memory controller sets would be implemented on one NoC layer, while the state-exchange network described in Chapter 9 would exist as second layer. NoC routers can also be outfitted with error correction themselves [93]. Unfortunately, the few NoC-specialized experts I encountered while conducting this research had little interest in implementing their research practically. Hence I hope incorporate NoC into this MPSoC design in the future in collaboration with those who are willing to do so.

I designed this OBC architecture specifically to utilize and exploit the powerful fault-recovery capabilities of modern FPGAs. However, this OBC architecture could very well be realized also on ASICs manufactured in radiation-robust COTS manufacturing processes such as FD-SoI [144]. This would allow much reduced energy consumption, and drastically higher clock speeds to be achieved. An ASIC variant would be less susceptible to transients and more robust to permanent faults, while loosing the capability to mitigate permanent faults through FPGA reconfiguration. However, due to the drastically increased development costs of an ASIC implementation, the resulting OBC would not be viable for miniaturized satellite applications anymore. We see this as a "big-space" variant of this approach with its own advantages, but it would no longer offer fault tolerance "on a budget".

This research began as a one-person project, but towards the end of my PhD, it has become clear that it has today outgrown the capacity of just a single researcher. In all regards, the end of my PhD is actually the beginning of something new, and more important. I know that in the coming years, I must gather a research group to advance this research and develop it further in a suitable environment. Where I will do this

remains yet to be seen. At the end of the second year and the beginning of the final year of my time as PhD researcher, I therefore began to explore ways for conducting long-term testing for this OBC architecture to appropriately consider the time-component that is introduced in testing hardware-software-hybrid systems. In this processes, I have had the pleasure collaborate with several international experts in the fields of radiation testing, space engineering, and semiconductor testing. Promising test environments for long-term testing include the close proximity of a radiation source, the Exposed Facility aboard the ISS (JEM-EF), or the vicinity of the Fukushima Daiichi site. Naturally, all these test setups require considerable preparation time, and preparing a prototype for deployed, e.g., aboard ISS is a highly competitive and certification-heavy undertaking. Therefore, I aim to conduct in parallel to long-term testing also on-orbit validation aboard a CubeSat, which is possible more rapidly and at reduced cost than e.g., through an ISS experiment. After all, on-orbit technology demonstration and validation is one of the prime use-cases for CubeSats today, and also one of their most successful applications.

On-orbit validation aboard a CubeSat also closes a circle that began with the early failure of the FirstMOVE CubeSat, and that initiated my satellite fault tolerance research. I started this research, searching for a way to realize a better, fault-tolerant satellite bus architecture for the MOVE-II CubeSat project. Back then, it became clear that there were simply no fault-tolerant OBC architectures or products in existence that could even theoretically be used to assure fault tolerance and guarantee reliable operation for long-term CubeSat mission. At the start of this thesis, we raised the question:

RQ0 *Can a fault tolerance computer architecture be achieved with modern embedded and mobile-market technology, without breaking the mass, size, complexity, and budget constraints of miniaturized satellite applications?*

This hard question arrose at the beginning of the development process of the MOVE-II CubeSat. I approached this research without a specific architecture or solution in mind, and even briefly considered a highly experimental, academic VLIW platform. Three years, many published research papers, and several catastrophes later, it is now possible to answer this question in the following way:

RQ0 Yes. A fault-tolerant computer architecture for miniaturized satellites is technically feasible with contemporary COTS technology. Once fully implemented as a prototype, it can be used to expand the reliable lifetime of modern day CubeSats drastically, thereby enabling their use in critical and long-term space missions. With contemporary COTS components, this OBC architecture can be applied to satellites as small as 2U CubeSats. Advances in semiconductor manufacturing in the upcoming generation of FPGAs will make this approach also usable for smaller spacecraft, and even more appealing as it scales with technology. It can improve efficiency and scalability when implemented aboard heavier spacecraft that we use today for high-priority science and solar system exploration. And maybe in the future, hopefully, we can explore even what lies beyond its boundaries.

Bibliography

[1] Directorate of Technical and Quality Management, *ESA/NPI 497-2016: Efficient Dependable Space-Borne Computing through Advanced Reconfigurability Concepts.* ESA, December 2016.

[2] M. Langer and J. Bouwmeester, "Reliability of CubeSats-statistical data, developers' beliefs and the way forward," in *AIAA/USU Conference on Small Satellites (SmallSat)*, 2016.

[3] M. Swartwout, "The first one hundred CubeSats: A statistical look," *Journal of Small Satellites*, vol. 2, no. 2, pp. 213–233, 2013.

[4] E. Stassinopoulos and J. P. Raymond, "The space radiation environment for electronics," *Proceedings of the IEEE*, vol. 76, no. 11, pp. 1423–1442, 1988.

[5] J. R. Schwank, M. R. Shaneyfelt, and P. E. Dodd, "Radiation Hardness Assurance Testing of Microelectronic Devices and Integrated Circuits," *IEEE Transactions on Nuclear Science*, 2013.

[6] L. Whetsel, "An IEEE 1149.1-based test access architecture for ICs with embedded cores," in *International Test Conference (ITC)*. IEEE, 1997.

[7] M. R. Patel, *Spacecraft power systems.* CRC press, 2004.

[8] C. Boshuizen, J. Mason, P. Klupar, and S. Spanhake, "Results from the planet labs flock constellation," in *AIAA/USU Conference on Small Satellites (SmallSat)*, 2014.

[9] A. Poghosyan and A. Golkar, "CubeSat evolution: Analyzing CubeSat capabilities for conducting science missions," *Progress in Aerospace Sciences, Elsevier*, vol. 88, pp. 59–83, 2017.

[10] T. Wahl, G. K. Høye, A. Lyngvi, and B. T. Narheim, "New possible roles of small satellites in maritime surveillance," *Acta Astronautica, Elsevier*, vol. 56, no. 1-2, pp. 273–277, 2005.

[11] M. Parra, A. J. Ricco, B. Yost, M. R. McGinnis, and J. W. Hines, "Studying space effects on microorganisms autonomously: genesat, pharmasat and the future of bio-nanosatellites," *Gravitational and Space Biology Bulletin*, vol. 21, pp. 9–17, 2008.

[12] Q. Schiller, D. Gerhardt, L. Blum, X. Li, and S. Palo, "Design and scientific return of a miniaturized particle telescope onboard the colorado student space weather experiment (CSSWE) CubeSat," in *IEEE Aerospace Conference*. IEEE, 2014.

[13] C. Underwood, S. Pellegrino, V. J. Lappas, C. P. Bridges, and J. Baker, "Using CubeSat/micro-satellite technology to demonstrate the autonomous assembly of a reconfigurable space telescope (AAReST)," *Acta Astronautica, Elsevier*, vol. 114, pp. 112–122, 2015.

[14] W. Weiss, S. Rucinski, A. Moffat, A. Schwarzenberg-Czerny, O. Koudelka, C. Grant, R. Zee, R. Kuschnig, J. Matthews, P. Orleanski *et al.*, "Brite-constellation: nanosatellites for precision photometry of bright stars," *Publications of the Astronomical Society of the Pacific*, vol. 126, no. 940, p. 573, 2014.

[15] S. Lacour, M. Nowak, P. Bourget, F. Vincent, A. Kellerer, V. Lapeyrère, L. David, A. Le Tiec, O. Straub, and J. Woillez, "Sage: using CubeSats for gravitational wave detection," in *Space Telescopes and Instrumentation 2018: Ultraviolet to Gamma Ray*, vol. 10699. International Society for Optics and Photonics, 2018, p. 106992R.

[16] S.-i. Watanabe, Y. Tsuda, M. Yoshikawa, S. Tanaka, T. Saiki, and S. Nakazawa, "Hayabusa2 mission overview," *Space Science Reviews, Springer*, vol. 208, no. 1-4, pp. 3–16, 2017.

[17] J. Schoolcraft, A. T. Klesh, and T. Werne, "Marco: interplanetary mission development on a CubeSat scale," in *Space Operations: Contributions from the Global Community*. Springer, 2017.

[18] I. F. Akyildiz and A. Kak, "The internet of space things/cubesats: A ubiquitous cyber-physical system for the connected world," *Computer Networks, Elsevier*, vol. 150, pp. 134–149, 2019.

[19] M. Cappella, "The principle of equitable access in the age of mega-constellations," in *Legal Aspects Around Satellite Constellations*. Springer, 2019, pp. 11–23.

[20] L. Wang, R. Chen, B. Xu, X. Zhang, T. Li, and C. Wu, "The challenges of LEO based navigation augmentation system–lessons learned from Luojia-1a satellite," in *China Satellite Navigation Conference*. Springer, 2019, pp. 298–310.

[21] M. Harris, "Tech giants race to build orbital internet [news]," *IEEE Spectrum*, vol. 55, no. 6, pp. 10–11, 2018.

[22] H. Bedon, C. Negron, J. Llantoy, C. M. Nieto, and C. O. Asma, "Preliminary internetworking simulation of the qb50 CubeSat constellation," in *IEEE Latin-American Conference on Communications*. IEEE, 2010, pp. 1–6.

[23] V. L. Foreman, A. Siddiqi, and O. De Weck, "Large satellite constellation orbital debris impacts: case studies of oneweb and spacex proposals," in *AIAA SPACE and Astronautics Forum and Exposition*, 2017, p. 5200.

[24] T. Hiriart and J. H. Saleh, "Observations on the evolution of satellite launch volume and cyclicality in the space industry," *Space Policy, Elsevier*, vol. 26, no. 1, pp. 53–60, 2010.

[25] L. D. Feinberg, "Engineering history of the james webb space telescope (JWST) optical telescope element," 2018, nASA Goddard Space Flight Center.

[26] F. Lura and D. Hagelschuer, "System conditioning-our ways and testing tools for the development of reliability for spaceborne components and small satellites," in *Digest of the First International Symposium of the International Academy of Astronautics (IAA), Berlin, November*, 1999, pp. 4–8.

[27] S. Vinod *et al.*, "Satellite ground testing-objectives and implementation," *Ground Testing of Aerospace Vehicles Including Engines, Allied Publishers*, p. 223, 1994.

[28] R. Haefer, "Vacuum and cryotechniques in space research," *Vacuum, Elsevier*, vol. 22, no. 8, pp. 303–314, 1972.

[29] D. Koelle, "Specific transportation costs to GEO – past, present and future," *Acta Astronautica, Elsevier*, vol. 53, no. 4, pp. 797–803, 2003.

[30] J. N. Pelton, "Launch vehicles and launch sites," in *Handbook of Satellite Applications*. Springer, 2013, pp. 1131–1144.

[31] A. L. Weigel and D. E. Hastings, "Evaluating the cost and risk impacts of launch choices," *Journal of Spacecraft and Rockets, AIAA*, vol. 41, no. 1, pp. 103–110, 2004.

[32] H. Helvajian and S. Janson, *Small satellites: past, present, and future*. AIAA, 2009.

[33] J. Depasquale, A. Charania, H. Kanamaya, and S. Matsuda, "Analysis of the earth-to-orbit launch market for nano and microsatellites," in *AIAA SPACE 2010 Conference & Exposition*, 2010, p. 8602.

[34] D. DePasquale and J. Bradford, "Nano/microsatellite market assessment," *Public Release, Revision A, SpaceWorks*, 2013.

[35] B. Twiggs, S. Lee, A. Hutputanasin, A. Toorian, W. Lan, R. Munakata, J. Carnahan, D. Pignatelli, A. Mehrparvar *et al.*, "CubeSat design specification rev. 13," Cal Poly SLO, Standard, 2015.

[36] M. Czech, A. Fleischner, and U. Walter, "A first-move in satellite development at the tu-münchen," in *Small Satellite Missions for Earth Observation*. Springer, 2010, pp. 235–245.

[37] D. J. Barnhart, T. Vladimirova, A. M. Baker, and M. N. Sweeting, "A low-cost femtosatellite to enable distributed space missions," *Acta Astronautica, Elsevier*, vol. 64, no. 11-12, pp. 1123–1143, 2009.

[38] J. Tristancho and J. Gutierrez-Cabello, "A probe of concept for femto-satellites based on commercial-of-the-shelf," in *IEEE/AIAA Digital Avionics Systems Conference*. IEEE, 2011, pp. 8A2–1.

[39] J. Bouwmeester, M. Langer, and E. Gill, "Survey on the implementation and reliability of CubeSat electrical bus interfaces," *CEAS Space Journal, Springer*, vol. 9, no. 2, pp. 163–173, 2017.

[40] R. Carlson, K. Hand, and E. Ozer, "On the use of System-on-Chip technology in next-generation instruments avionics for space exploration," in *IFIP/IEEE International Conference on Very Large Scale Integration-System on a Chip (VLSI-SoC), revised paper.* Springer, 2016.

[41] M. Swartwout, "The first one hundred CubeSats: A statistical look," *Journal of Small Satellites*, 2014.

[42] M. Swartwout, "You say "PicoSat", i say "CubeSat": Developing a better taxonomy for secondary spacecraft," in *IEEE Aerospace Conference*, 2018.

[43] M. Swartwout, "Cubesats and mission success: A look at the numbers," in *CubeSat Developers Workshop.* CalPoly, 2016.

[44] R. Trivedi and U. S. Mehta, "A survey of radiation hardening by design (RHBD) techniques for electronic systems for space application," *International Journal of Electronics and Communication Engineering & Technology (IJECET)*, vol. 7, no. 1, p. 75, 2016.

[45] P. Roche, J.-L. Autran, G. Gasiot, and D. Munteanu, "Technology downscaling worsening radiation effects in bulk: SOI to the rescue," in *IEEE International Electron Devices Meeting.* IEEE, 2013, pp. 31–1.

[46] S. M. Guertin, M. Amrbar, and S. Vartanian, "Radiation test results for common CubeSat microcontrollers and microprocessors," in *IEEE Radiation Effects Data Workshop (REDW).* IEEE, 2015, pp. 1–9.

[47] D. Selčan, G. Kirbiš, and I. Kramberger, "Low level radiation and fault protection techniques suitable for nanosatellite missions," in *Conference on Radiation and its Effects on Components and Systems (RADECS).* IEEE, 2017.

[48] M. Swartwout, "Secondary spacecraft in 2016: Why some succeed (and too many do not)," in *IEEE Aerospace Conference.* IEEE, 2016, pp. 1–13.

[49] M. Williamson, "Commercial space risks, spacecraft insurance, and the fragile frontier," in *Frontiers of Space Risk.* CRC Press, 2018, pp. 143–163.

[50] J. R. Samson, "Update on dependable multiprocessor CubeSat technology development," in *2012 IEEE Aerospace Conference.* IEEE, 2012, pp. 1–12.

[51] B. S. Dhillon, *Human reliability: with human factors.* Elsevier, 2013.

[52] R. Isermann, *Fault-diagnosis systems: an introduction from fault detection to fault tolerance.* Springer Science & Business Media, 2006.

[53] I. P. Egwutuoha, D. Levy, B. Selic, and S. Chen, "A survey of fault tolerance mechanisms and checkpoint/restart implementations for high performance computing systems," *Journal of Supercomputing*, vol. 65, no. 3, pp. 1302–1326, 2013.

[54] R. Ginosar, "Survey of processors for space," *Eurospace Data Systems In Aerospace (DASIA)*, 2012.

[55] K. Tindell, H. Hanssmon, and A. J. Wellings, "Analysing real-time communications: Controller area network (CAN)." in *Real Time System Symposium (RTSS).* IEEE, 1994, pp. 259–263.

[56] R. Makowitz and C. Temple, "Flexray–a communication network for automotive control systems," in *IEEE International Workshop on Factory Communication Systems*. IEEE, 2006, pp. 207–212.

[57] W. R. Moore, "A review of fault-tolerant techniques for the enhancement of integrated circuit yield," *Proceedings of the IEEE*, vol. 74, no. 5, pp. 684–698, 1986.

[58] L. Jiang, R. Ye, and Q. Xu, "Yield enhancement for 3d-stacked memory by redundancy sharing across dies," in *IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. IEEE, 2010, pp. 230–234.

[59] P. A. Buckland, J. R. Herring, G. M. Nordstrom, and W. A. Thompson, "Cable redundancy and failover for multi-lane pci express io interconnections," Mar. 18 2014, US Patent 8,677,176.

[60] B. Vucetic and J. Yuan, *Turbo codes: principles and applications*. Springer Science & Business Media, 2012, vol. 559.

[61] Z. Zhang, V. Anantharam, M. J. Wainwright, and B. Nikolic, "An efficient 10gbase-t ethernet ldpc decoder design with low error floors," *IEEE Journal of Solid-State Circuits*, vol. 45, no. 4, pp. 843–855, 2010.

[62] Y. Furukawa, "Intellectual property protection and innovation: An inverted-u relationship," *Economics Letters, Elsevier*, vol. 109, no. 2, pp. 99–101, 2010.

[63] M. Fidler and A. Rizk, "A guide to the stochastic network calculus," *IEEE Communications Surveys & Tutorials*, vol. 17, no. 1, pp. 92–105, 2014.

[64] K. Suresh, C. W. Selvidge, S. Gupta, and A. Jain, "Debug environment for a multi user hardware assisted verification system," Feb. 1 2018, US Patent App. 15/646,003.

[65] M. Kooli and G. Di Natale, "A survey on simulation-based fault injection tools for complex systems," in *International Conference on Design & Technology of Integrated Systems in Nanoscale Era (DTIS)*. IEEE, 2014.

[66] H. Ziade, R. A. Ayoubi, R. Velazco *et al.*, "A survey on fault injection techniques," *Int. Arab J. Inf. Technol.*, vol. 1, no. 2, pp. 171–186, 2004.

[67] M. M. Hassan, W. Afzal, M. Blom, B. Lindström, S. F. Andler, and S. Eldh, "Testability and software robustness: A systematic literature review," in *Euromicro Conference on Software Engineering and Advanced Applications*. IEEE, 2015.

[68] J. W. Bennett, G. J. Atkinson, B. C. Mecrow, and D. J. Atkinson, "Fault-tolerant design considerations and control strategies for aerospace drives," *IEEE Transactions on Industrial Electronics*, vol. 59, no. 5, pp. 2049–2058, 2011.

[69] G. C. Clark Jr and J. B. Cain, *Error-correction coding for digital communications*. Springer Science & Business Media, 2013.

[70] P. W. Coteus, H. C. Hunter, C. A. Kilmer, K.-h. Kim, L. A. Lastras-Montano, W. E. Maule, and V. Patel, "Error feedback and logging with memory on-chip error checking and correcting (ECC)," Jan. 30 2018, US Patent 9,880,896.

[71] M. Tipaldi and B. Bruenjes, "Survey on fault detection, isolation, and recovery strategies in the space domain," *Journal of Aerospace Information Systems*, vol. 12, no. 2, pp. 235–256, 2015.

[72] D. A. Patterson, G. Gibson, and R. H. Katz, *A case for redundant arrays of inexpensive disks (RAID)*. ACM, 1988, vol. 17, no. 3.

[73] P. R. Grams, "Ethernet for aerospace applications-ethernet heads for the skies," in *IEEE Ethernet Technology Summit*. NASA, 2015.

[74] M. Luby, L. Vicisano, J. Gemmell, L. Rizzo, M. Handley, and J. Crowcroft, "RFC 5052: Forward error correction (FEC) building block," IETF, Tech. Rep., 2007.

[75] D. Boley, G. H. Golub, S. Makar, N. Saxena, and E. J. McCluskey, "Floating point fault tolerance with backward error assertions," *IEEE Transactions on Computers*, vol. 44, no. 2, pp. 302–311, 1995.

[76] R. C. Aitken, "Modeling the unmodelable: Algorithmic fault diagnosis," *IEEE Design & Test of Computers*, vol. 14, no. 3, pp. 98–103, 1997.

[77] J. Sloan, R. Kumar, and G. Bronevetsky, "Algorithmic approaches to low overhead fault detection for sparse linear algebra," in *Conference on Dependable Systems and Networks (DSN)*. IEEE, 2012.

[78] N. R. Saxena and E. J. McCluskey, "Control-flow checking using watchdog assists and extended-precision checksums," *IEEE Transactions on Computers*, vol. 39, no. 4, pp. 554–559, 1990.

[79] S. Z. Shazli and M. B. Tahoori, "Transient error detection and recovery in processor pipelines," in *IEEE International Symposium on Defect and Fault Tolerance in VLSI Systems (DFT)*. IEEE, 2009, pp. 304–312.

[80] J. Gaisler, "Concurrent error-detection and modular fault-tolerance in a 32-bit processing core for embedded space flight applications," in *IEEE International Symposium on Fault-Tolerant Computing (FTCS)*. IEEE, 1994, pp. 128–130.

[81] S. Agrawal and K. Daudjee, "A performance comparison of algorithms for byzantine agreement in distributed systems," in *European Dependable Computing Conference (EDCC)*. IEEE, 2016.

[82] M. Barborak, A. Dahbura, and M. Malek, "The consensus problem in fault-tolerant computing," *ACM Computing Surveys (CSur)*, vol. 25, no. 2, pp. 171–220, 1993.

[83] T. Jackson, B. Salamat, A. Homescu, K. Manivannan, G. Wagner, A. Gal, S. Brunthaler, C. Wimmer, and M. Franz, "Compiler-generated software diversity," in *Moving Target Defense*. Springer, 2011, pp. 77–98.

[84] S. Punnekkat, A. Burns, and R. Davis, "Analysis of checkpointing for real-time systems," *Real-Time Systems, Springer*, vol. 20, no. 1, pp. 83–102, 2001.

[85] A. Thekkilakattil, R. Dobrin, and S. Punnekkat, "Fault tolerant scheduling of mixed criticality real-time tasks under error bursts," *Procedia Computer Science, Elsevier*, vol. 46, pp. 1148–1155, 2015.

[86] ——, "Bounding the effectiveness of temporal redundancy in fault-tolerant real-time scheduling under error bursts," in *IEEE Emerging Technology and Factory Automation (ETFA)*. IEEE, 2014, pp. 1–8.

[87] M. Short and J. Proenza, "Towards efficient probabilistic scheduling guarantees for real-time systems subject to random errors and random bursts of errors," in *Euromicro Conference on Real-Time Systems*. IEEE, 2013, pp. 259–268.

[88] X. Iturbe, B. Venu, E. Ozer, and S. Das, "A triple core lock-step (TCLS) ARM Cortex-R5 processor for safety-critical and ultra-reliable applications," in *Conference on Dependable Systems and Networks Workshop (DSN-W)*. IEEE, 2016.

[89] J. Arm, Z. Bradac, and R. Stohl, "Increasing safety and reliability of roll-back and roll-forward lockstep technique for use in real-time systems," *IFAC Conference on Programmable Devices and Embedded Systems (PDES), Elsevier*, vol. 49, no. 25, pp. 413–418, 2016.

[90] K. D. Safford, D. C. Soltis Jr, and E. R. Delano, "Off-chip lockstep checking," Jun. 26 2007, US Patent 7,237,144.

[91] B. H. Meyer, B. H. Calhoun, J. Lach, and K. Skadron, "Cost-effective safety and fault localization using distributed temporal redundancy," in *International Conference on Compilers, architectures and synthesis for embedded systems*. ACM, 2011, pp. 125–134.

[92] Y. Zhou, X. Wang, Y. Chen, and Z. Wang, "Armlock: Hardware-based fault isolation for arm," in *ACM SIGSAC conference on computer and communications security*. ACM, 2014, pp. 558–569.

[93] J. Zhou, H. Li, T. Wang, and X. Li, "Loft: A low-overhead fault-tolerant routing scheme for 3D NoCs," *Integration, the VLSI Journal*, 2016.

[94] Aeronautical Radio, INC, *ARINC Specification 664: Avionics Full Duplex Switched Ethernet (AFDX)*, 2005.

[95] Y. Li, E. L. Miller, and D. D. Long, "Understanding data survivability in archival storage systems," in *International Systems and Storage Conference*. ACM, 2012.

[96] N. Joukov, A. M. Krishnakumar, C. Patti, A. Rai, S. Satnur, A. Traeger, and E. Zadok, "RAIF: Redundant array of independent filesystems," in *Conference on Mass Storage Systems and Technologies (MSST)*. IEEE, 2007, pp. 199–214.

[97] M.-A. Song, S.-Y. Kuo, and I.-F. Lan, "A low complexity design of Reed-Solomon code algorithm for advanced RAID system," *IEEE Transactions on Consumer Electronics (TCE)*, vol. 53, 2007.

[98] R. L. Alena, J. P. Ossenfort, K. I. Laws, A. Goforth, and F. Figueroa, "Communications for integrated modular avionics," in *IEEE Aerospace Conference*. IEEE, 2007, pp. 1–18.

[99] M. Roa, W. Cantrell, D. Cartes, and M. Nelson, "Requirements for deterministic control systems," in *IEEE Electric Ship Technologies Symposium*. IEEE, 2011, pp. 439–445.

[100] S. V. Amari and G. Dill, "Redundancy optimization problem with warm-standby redundancy," in *Reliability and Maintainability Symposium (RAMS)*. IEEE, 2010.

[101] ——, "A new method for reliability analysis of standby systems," in *Reliability and Maintainability Symposium (RAMS)*. IEEE, 2009.

[102] J. J. Wylie and R. Swaminathan, "Selecting erasure codes for a fault tolerant system," Aug. 21 2012, US Patent 8,250,427.

[103] J. S. Plank, "A tutorial on reed–solomon coding for fault-tolerance in raid-like systems," *Software: Practice and Experience*, vol. 27, no. 9, pp. 995–1012, 1997.

[104] M. Hijorth, M. Aberg, N.-J. Wessman, J. Andersson, R. Chevallier, R. Forsyth, R. Weigand, and L. Fossati, "GR740: Rad-hard quad-core LEON4FT system-on-chip," in *Eurospace Data Systems In Aerospace (DASIA)*, 2015.

[105] L. Bozzoli and L. Sterpone, "Self rerouting of dynamically reconfigurable SRAM-based FPGAs," in *NASA/ESA Conference on Adaptive Hardware and Systems (AHS)*. IEEE, 2017.

[106] R. Baheti and H. Gill, "Cyber-physical systems," *The impact of control technology, IEEE Control Systems Society*, vol. 12, no. 1, pp. 161–166, 2011.

[107] M. D. Berg, H. S. Kim, A. M. Phan, C. M. Seidleck, K. A. LaBel, J. A. Pellish, and M. J. Campola, "The effects of race conditions when implementing single-source redundant clock trees in triple modular redundant synchronous architectures," in *Conference on Radiation and its Effects on Components and Systems (RADECS)*. IEEE, 2016.

[108] M. Berg, K. LaBel, M. Campola, and M. Xapsos, "Analyzing system on a chip single event upset responses using single event upset data, classical reliability models, and space environment data," in *Conference on Radiation and its Effects on Components and Systems (RADECS)*. IEEE, 2017.

[109] G. G. Preckshot, *Method for performing diversity and defense-in-depth analyses of reactor protection systems.* Division of Reactor Controls and Human Factors, Office of Nuclear Reactor Regulation, US Nuclear Regulatory Commission, 1994.

[110] D. K. Nilsson and U. Larson, "A defense-in-depth approach to securing the wireless vehicle infrastructure," *Journal of Networks (JNW)*, vol. 4, no. 7, pp. 552–564, 2009.

[111] T. G. Rauscher, "Raid system with multiple controllers and proof against any single point of failure," Mar. 29 2005, US Patent 6,874,100.

[112] H. Madeira, R. R. Some, F. Moreira, D. Costa, and D. Rennels, "Experimental evaluation of a cots system for space applications," in *Proceedings International Conference on Dependable Systems and Networks.* IEEE, 2002, pp. 325–330.

[113] J. Hammarberg and S. Nadjm-Tehrani, "Formal verification of fault tolerance in safety-critical reconfigurable modules," *Journal on Software Tools for Technology Transfer, Springer*, vol. 7, no. 3, pp. 268–279, 2005.

[114] X. Cai and M. R. Lyu, "Software reliability modeling with test coverage: Experimentation and measurement with a fault-tolerant software project," in *IEEE International Symposium on Software Reliability (ISSRE)*. IEEE, 2007, pp. 17–26.

[115] J. L. Nunes, T. Pecserke, J. C. Cunha, and M. Zenha-Rela, "FIRED–fault injector for reconfigurable embedded devices," in *IEEE Pacific Rim International Symposium on Dependable Computing (PRDC)*. IEEE, 2015.

[116] I. Sommerville, "Software engineering (10th edition)," *ISBN-10*, vol. 0133943038, 2015.

[117] B. Schroeder, E. Pinheiro, and W.-D. Weber, "DRAM errors in the wild: a large-scale field study," *Communications of the ACM*, vol. 54, no. 2, pp. 100–107, 2011.

[118] A. Mukati, "A survey of memory error correcting techniques for improved reliability," *Journal of Network and Computer Applications, Elsevier*, vol. 34, no. 2, pp. 517–522, 2011.

[119] T. Lanier, "Exploring the design of the cortex-a15 processor," *https://www.arm.com/files/pdf/AT-Exploring_the_Design_of_the_Cortex-A15.pdf*, 2011.

[120] USB-IF, "Universal serial bus revision 3.1 specification," 2011.

[121] K. Deyring *et al.*, "Serial ATA: High speed serialized at attachment," *Jan*, vol. 7, pp. 1–22, 2003.

[122] P. Savio, A. Nespola, S. Straullu, S. Abrate, and R. Gaudino, "A physical coding sublayer for gigabit ethernet over POF," in *International Conference on Plastic Optical Fibers (POF)*. International Cooperative of Plastic Optical Fibers, 2010.

[123] A. Goldhammer and J. Ayer Jr, "Understanding performance of pci express systems," *Xilinx WP350, Sept*, vol. 4, 2008.

[124] H. Zhang, S. Krooswyk, and J. Ou, *High Speed Digital Design: Design of High Speed Interconnects and Signaling*. Elsevier, 2015.

[125] R. Micheloni, A. Marelli, and K. Eshghi, *Inside solid state drives (SSDs)*. Springer, 2013.

[126] C. W. Slayman, "Cache and memory error detection, correction, and reduction techniques for terrestrial servers and workstations," *IEEE Transactions on Device and Materials Reliability*, vol. 5, no. 3, pp. 397–404, 2005.

[127] L. L. Pullum, *Software fault tolerance techniques and implementation*. Artech House, 2001.

[128] N. Diniz and J. Rufino, "ARINC 653 in space," in *Eurospace Data Systems In Aerospace (DASIA)*, 2005.

[129] J. Teich, "Hardware/software codesign: The past, the present, and predicting the future," *Proceedings of the IEEE*, vol. 100, no. Special Centennial Issue, pp. 1411–1430, 2012.

[130] W. Bartlett and L. Spainhower, "Commercial fault tolerance: A tale of two systems," *IEEE Transactions on dependable and secure computing*, vol. 1, no. 1, pp. 87–96, 2004.

[131] T. Slivinski, C. Broglio, C. Wild, J. Goldberg, K. Levitt, E. Hitt, and J. Webb, "Study of fault-tolerant software technology," NASA, Technical Report, 1984.

[132] K. Reick, P. N. Sanda, S. Swaney, J. W. Kellington, M. Mack, M. Floyd, and D. Henderson, "Fault-tolerant design of the ibm power6 microprocessor," *IEEE micro*, vol. 28, no. 2, pp. 30–38, 2008.

[133] C. C. Reed, R. Briët, M. Begert, and T. Newbauer, "Esd detection, location and mitigation, and why they are important for satellite development," in *Spacecraft Charging Technology Conference*, 2014.

[134] S. Bourdarie and M. Xapsos, "The Near-Earth Space Radiation Environment," *IEEE Transactions on Nuclear Science*, 2008.

[135] Xapsos, O'Neill, and T. P. O'Brien, "Near-Earth Space Radiation Models," *IEEE Transactions on Nuclear Science*, 2013.

[136] J. Heirtzler, "The future of the south atlantic anomaly and implications for radiation damage in space," *Journal of Atmospheric and Solar-Terrestrial Physics, Elsevier*, 2002.

[137] ECSS, "Calculation of radiation and its effects and margin policy handbook," 2010.

[138] T. Amort, "Radiation-hardening by design phase 3," in *Microectronics Reliability and Qualification Workshop (MRQW)*. The Aerospace Corporation, 2013.

[139] P. Mishra, A. Muttreja, and N. K. Jha, "FinFET circuit design," in *Nanoelectronic Circuit Design*. Springer, 2011, pp. 23–54.

[140] S. A. Vitale, P. W. Wyatt, N. Checka, J. Kedzierski, and C. L. Keast, "FDSOI process technology for subthreshold-operation ultralow-power electronics," *Proceedings of the IEEE*, vol. 98, no. 2, pp. 333–342, 2010.

[141] M. Alles, R. Schrimpf, R. Reed, L. Massengill, R. Weller, M. Mendenhall, D. Ball, K. Warren, T. Loveless, J. Kauppila *et al.*, "Radiation hardness of FDSOI and FinFET technologies," in *IEEE International SOI Conference*. IEEE, 2011.

[142] L. A. Tambara, F. L. Kastensmidt, N. H. Medina, N. Added, V. A. Aguiar, F. Aguirre, E. L. Macchione, and M. A. Silveira, "Heavy ions induced single event upsets testing of the 28 nm Xilinx Zynq-7000 all programmable SoC," in *IEEE Radiation Effects Data Workshop (REDW)*, 2015.

[143] M. D. Berg, K. A. LaBel, and J. Pellish, "Single event effects in FPGA devices 2014-2015," in *NASA NEPP Electronics Technology Workshop*, 2015.

[144] M. Kochiyama, T. Sega, K. Hara, Y. Arai, T. Miyoshi, Y. Ikegami, S. Terada, Y. Unno, K. Fukuda, and M. Okihara, "Radiation effects in Silicon-on-Insulator transistors with back-gate control method fabricated with OKI semiconductor 0.20 μm FD-SOI technology," *Nuclear Instruments and Methods in Physics Research, Elsevier*, 2011.

[145] H. Hayat, K. Kohary, and C. D. Wright, "Can conventional phase-change memory devices be scaled down to single-nanometre dimensions?" *Nanotechnology, IOP Publishing*, 2016.

[146] A. Fert, J.-M. George, H. Jaffrès, R. Mattana, and P. Seneor, "The new era of spintronics," *Europhysics news, EDP Sciences*, vol. 34, no. 6, pp. 227–229, 2003.

[147] J.-C. Wu, H. L. Stadler, and R. R. Katti, "High speed magneto-resistive random access memory," Dec. 22 1992, US Patent 5,173,873.

[148] D. Chen, H. Kim, A. Phan, E. Wilcox, K. LaBel, S. Buchner, A. Khachatrian, and N. Roche, "Single-event effect performance of a commercial embedded reram," *IEEE Transactions on Nuclear Science*, vol. 61, no. 6, pp. 3088–3094, 2014.

[149] F. Chen, "Phase-change memory," Feb. 26 2014, US Patent App. 14/191,016.

[150] G. Tsiligiannis, L. Dilillo, A. Bosio, P. Girard, A. Todri, A. Virazel, S. McClure, A. Touboul, F. Wrobel, and F. Saigné, "Testing a Commercial MRAM Under Neutron and Alpha Radiation in Dynamic Mode," *IEEE Transactions on Nuclear Science*, 2013.

[151] J. Maimon, K. Hunt, J. Rodgers, L. Burcin, and K. Knowles, "Results of radiation effects on a chalcogenide non-volatile memory array," in *IEEE Aerospace Conference*, 2004.

[152] J. P. van Zandwijk and A. Fukami, "NAND flash memory forensic analysis and the growing challenge of bit errors," *IEEE Security & Privacy*, vol. 15, no. 6, pp. 82–87, 2017.

[153] S. Gerardin, M. Bagatin, A. Paccagnella, K. Grürmann, F. Gliem, T. Oldham, F. Irom, and D. N. Nguyen, "Radiation Effects in Flash Memories," *IEEE Transactions on Nuclear Science*, 2013.

[154] C. Poivey, "Total ionizing dose (TID) and total non ionizing dose (TNID) effects in electronic parts," Lecture Notes of the School on the Effects of Radiation on Embedded Systems for Space Applications (SERESSA), 2018.

[155] T. Oldham, M. Suhail, M. Friendlich, M. Carts, R. Ladbury, H. Kim, M. Berg, C. Poivey, S. Buchner, A. Sanders *et al.*, "TID and SEE response of advanced 4g NAND flash memories," in *IEEE Radiation Effects Data Workshop (REDW)*. IEEE, 2008, pp. 31–37.

[156] K. Young *et al.*, "SLC vs. MLC: An analysis of flash memory," *Whitepaper, Super Talent Technology, Inc.*, 3 2008.

[157] S. Gerardin and A. Paccagnella, "Present and future non-volatile memories for space," *IEEE Transactions on Nuclear Science*, vol. 57, 2010.

[158] K. Gupta and K. Kirby, "Mitigation of high altitude and low earth orbit radiation effects on microelectronics via shielding or error detection and correction systems," NASA, Tech. Rep., 2004.

[159] B. Klamm, "Passive space radiation shielding: Mass and volume optimization of tungsten-doped polyphenolic and polyethylene resins," in *AIAA/USU Conference on Small Satellites (SmallSat)*, 2015.

[160] E. Benton and E. Benton, "A survey of radiation measurements made aboard russian spacecraft in low-earth orbit," NASA, Tech. Rep., 1999.

[161] M. Poizat, M. Sauvagnac, A. Samaras, Y. Padie, P. Garcia, B. Renaud, L. Gouyet, J. P. Abadi, F. Widmeer, E. Le Goulven *et al.*, "Compendium of total ionizing dose, displacement damage and single event transient test data of various optocouplers for esa," in *IEEE Radiation Effects Data Workshop (REDW)*. IEEE, 2013, pp. 1–6.

[162] A. E. Bergles, "Evolution of cooling technology for electrical, electronic, and microelectronic equipment," *IEEE Transactions on Components and Packaging Technologies*, 2003.

[163] K. Puttaswamy and G. H. Loh, "Thermal herding: Microarchitecture techniques for controlling hotspots in high-performance 3d-integrated processors," in *International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2007.

[164] D. G. Gilmore and M. Donabedian, *Spacecraft thermal control handbook: cryogenics*. AIAA, 2003, vol. 2.

[165] B. Wood, W. Bertrand, R. Bryson, B. Seiber, and P. M. FALCO, "Surface effects of satellite material outgassing products," *Journal of Thermophysics and Heat Transfer, AIAA*, vol. 2, no. 4, pp. 289–295, 1988.

[166] B. R. Spence, S. White, M. LaPointe, S. Kiefer, P. LaCorte, J. Banik, D. Chapman, and J. Merrill, "International space station (ISS) roll-out solar array (ROSA) spaceflight experiment mission and results," in *IEEE World Conference on Photovoltaic Energy Conversion (WCPEC)*. IEEE, 2018, pp. 3522–3529.

[167] R. Gubby and J. Evans, "Space environment effects and satellite design," *Journal of Atmospheric and Solar-Terrestrial Physics, Elsevier*, vol. 64, no. 16, pp. 1723–1733, 2002.

[168] A. Driskill-Smith, D. Apalkov, V. Nikitin, X. Tang, S. Watts, D. Lottis, K. Moon, A. Khvalkovskiy, R. Kawakami, X. Luo *et al.*, "Latest advances and roadmap for in-plane and perpendicular STT-RAM," in *IEEE International Memory Workshop (IMW)*. IEEE, 2011, pp. 1–3.

[169] V. Dos Santos Paulino, "Influence of risk on technology adoption: inertia strategy in the space industry," *European Journal of Innovation Management*, vol. 17, no. 1, pp. 41–60, 2014.

[170] C. Boshuizen, W. Marshall, C. Bridges, S. Kenyon, and P. Klupar, "Learning to follow: Embracing commercial technologies and open source for space missions," in *International Astronautical Congress (IAC'11)*, no. IAC-11, 2011.

[171] G. Dubos, J. Saleh, and R. Braun, "Technology readiness level, schedule risk and slippage in spacecraft design: Data analysis and modeling," in *AIAA SPACE 2007 conference & exposition*, 2007, p. 6020.

[172] D. M. Waltz, *On-orbit servicing of space systems.* Krieger Pub Co, 1993.

[173] D. E. Hastings and C. Joppin, "On-orbit upgrade and repair: The hubble space telescope example," *Journal of spacecraft and rockets, AIAA*, vol. 43, no. 3, pp. 614–625, 2006.

[174] A. Long, M. Richards, and D. E. Hastings, "On-orbit servicing: a new value proposition for satellite design and operation," *Journal of Spacecraft and Rockets*, vol. 44, no. 4, pp. 964–976, 2007.

[175] L. Crane, "Crunch time in orbit," 2018, elsevier.

[176] J. L. Webster, "Cassini spacecraft engineering tutorial," 2006, nASA/Jet Propulsion Lab.

[177] R. D. Lange, "Cassini-huygens mission overview and recent science results," in *IEEE Aerospace Conference.* IEEE, 2008, pp. 1–10.

[178] G. Maral and M. Bousquet, *Satellite communications systems: systems, techniques and technology.* John Wiley & Sons, 2011.

[179] W. Larson, J. Wertz, and B. D'Souza, *SMAD III: Space Mission Analysis and Design, 3rd Edition: Workbook*, ser. Space technology library. Microcosm Press, 2005.

[180] S. Cakaj, W. Keim, and K. Malarić, "Communications duration with low earth orbiting satellites," in *IASTED International Conference on Antennas, Radar and Wave Propagation (ARP).* ACTA Press, 2007.

[181] S. H. Schaire, S. Altunc, G. Bussey, H. Shaw, B. Horne, and J. Schier, "NASA near earth network (NEN), deep space network (DSN) and space network (SN) support of CubeSat communications," in *NASA SpaceOps Workshop.* NASA, 2015.

[182] Y. Nakamura, S. Nakasuka, and Y. Oda, "Low-cost and reliable ground station network to improve operation efficiency for micro/nano-satellites," in *International Astronautical Congress (IAC)*, 2005.

[183] R. Welch, D. Limonadi, and R. Manning, "Systems engineering the curiosity rover: A retrospective," in *International Conference on System of Systems Engineering.* IEEE, 2013, pp. 70–75.

[184] R. Ludwig and J. Taylor, *Voyager telecommunications.* John Wiley and Sons, Inc, 2016.

[185] J. Taylor, *Deep Space Communications.* John Wiley & Sons, 2016.

[186] K. Reh, L. Spilker, J. I. Lunine, J. H. Waite, M. L. Cable, F. Postberg, and K. Clark, "Enceladus life finder: the search for life in a habitable moon," in *IEEE Aerospace Conference.* IEEE, 2016, pp. 1–8.

[187] B. Bastida Virgili and H. Krag, "Mega-constellations issues," in *COSPAR Scientific Assembly*, 2016.

[188] A. S. Jackson, "Implementation of the configurable fault tolerant system experiment on NPSAT-1," Ph.D. dissertation, Naval Postgraduate School Monterey, 2016.

[189] D. Lüdtke, K. Westerdorff, K. Stohlmann, A. Börner, O. Maibaum, T. Peng, B. Weps, G. Fey, and A. Gerndt, "OBC-NG: towards a reconfigurable on-board computing architecture for spacecraft," in *IEEE Aerospace*, 2014.

[190] S. Gupta, N. Gala, G. Madhusudan, and V. Kamakoti, "SHAKTI-F: A fault tolerant microprocessor architecture," in *IEEE Asian Test Symposium (ATS)*, 2015.

[191] R. DeCoursey, R. Melton, and R. R. Estes, "Non-radiation hardened microprocessors in space-based remote sensing systems," in *Sensors, Systems, and Next-Generation Satellites X*, vol. 6361. International Society for Optics and Photonics, 2006, p. 63611M.

[192] M. Pigno *et al.*, "A testbench for validation of DST fault-tolerant architectures on PowerPC G4 COTS microprocessors," in *Eurospace Data Systems In Aerospace (DASIA)*, 2011.

[193] M. Pignol, "DMT and DT2," in *IEEE International Symposium on On-Line Testing and Robust System Design (IOLTS)*, 2006.

[194] C. A. Hulme, H. H. Loomis, A. A. Ross, and R. Yuan, "Configurable fault-tolerant processor (CFTP) for spacecraft onboard processing," in *IEEE Aerospace Conference*, 2004.

[195] J. R. Samson, "Implementation of a dependable multiprocessor CubeSat," in *IEEE Aerospace*, 2011.

[196] X. Iturbe, D. Keymeulen, P. Yiu, D. Berisford, R. Carlson, K. Hand, and E. Ozer, "On the use of system-on-chip technology in next-generation instruments avionics for space exploration," in *IFIP/IEEE International Conference on Very Large Scale Integration-System on a Chip (VLSI-SoC)*. Springer, 2015, pp. 1–22.

[197] M. Marinella and H. Barnaby, "Total ionizing dose and displacement damage effects in embedded memory technologies," Sandia National Laboratories, Tech. Rep., 2013.

[198] U. Kretzschmar, J. Gomez-Cornejo, A. Astarloa, U. Bidarte, and J. Del Ser, "Synchronization of faulty processors in coarse-grained TMR protected partially reconfigurable FPGA designs," *Reliability Engineering & System Safety, Elsevier*, vol. 151, pp. 1–9, 2016.

[199] B. Döbel, "Operating system support for redundant multithreading," Ph.D. dissertation, Dresden University, 2014.

[200] A. Shye, T. Moseley, V. J. Reddi, J. Blomstedt, and D. A. Connors, "Using process-level redundancy to exploit multiple cores for transient fault tolerance," in *Conference on Dependable Systems and Networks (DSN)*. IEEE, 2007.

[201] Y. Dong, W. Ye, Y. Jiang, I. Pratt, S. Ma, J. Li, and H. Guan, "COLO: COarse-grained LOck-stepping virtual machines for non-stop service," in *Symposium on Cloud Computing (SoCC)*. ACM, 2013.

[202] A. L. Sartor, A. F. Lorenzon, L. Carro, F. Kastensmidt, S. Wong, and A. Beck, "Exploiting idle hardware to provide low overhead fault tolerance for VLIW processors," *ACM Journal on Emerging Technologies in Computing Systems (JETC)*, 2017.

[203] F. Anjam and S. Wong, "Configurable fault-tolerance for a configurable VLIW processor," in *International Symposium on Applied Reconfigurable Computing (ARC), Springer*, 2013.

[204] J. Hursey, J. M. Squyres, T. I. Mattox, and A. Lumsdaine, "The design and implementation of checkpoint/restart process fault tolerance for Open MPI," in *IEEE International Parallel and Distributed Processing Symposium.* IEEE, 2007, pp. 1–8.

[205] P. Munk, M. S. Alhakeem, R. Lisicki, H. Parzyjegla, J. Richling, and H.-U. Heiss, "Toward a fault-tolerance framework for COTS many-core systems," in *European Dependable Computing Conference (EDCC).* IEEE, 2015.

[206] L. Zeng, P. Huang, and L. Thiele, "Towards the design of fault-tolerant mixed-criticality systems on multicores," in *International Conference on Compilers, Architectures and Synthesis for Embedded Systems (CASES), ACM*, 2016.

[207] S. P. Azad, B. Niazmand, J. Raik, G. Jervan, and T. Hollstein, "Holistic approach for fault-tolerant network-on-chip based many-core systems," *HiPEAC DREAM-Cloud, ACM*, 2016.

[208] A. Höller, T. Rauter, J. Iber, G. Macher, and C. Kreiner, "Software-based fault recovery via adaptive diversity for COTS multi-core processors," 2015, arXiv:1511.03528.

[209] A. D. Santangelo, "An open source space hypervisor for small satellites," in *AIAA SPACE*, 2013.

[210] E. Missimer, R. West, and Y. Li, "Distributed real-time fault tolerance on a virtualized multi-core system," *Euromicro Conference on Real-Time Systems (ECRTS/OSPERT)*, 2014.

[211] Z. Al-bayati, B. H. Meyer, and H. Zeng, "Fault-tolerant scheduling of multi-core mixed-criticality systems under permanent failures," in *IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFT)*, 2016.

[212] S. Malik and F. Huet, "Adaptive fault tolerance in real time cloud computing," in *IEEE World Congress on Services (SERVICES)*, 2011.

[213] K. Smiri, S. Bekri, and H. Smei, "Fault-tolerant in embedded systems (MPSoC): Performance estimation and dynamic migration tasks," in *IEEE International Design & Test Symposium (IDT)*, 2016.

[214] Z. Al-bayati, J. Caplan, B. H. Meyer, and H. Zeng, "A four-mode model for efficient fault-tolerant mixed-criticality systems," in *Conference on Design, Automation and Test in Europe (DATE)*, 2016.

[215] S. Azimi, B. Du, and L. Sterpone, "On the prediction of radiation-induced SETs in flash-based FPGAs," *Microelectronics Reliability, Elsevier*, 2016.

[216] H. Zhang, L. Bauer, M. A. Kochte, E. Schneider, H.-J. Wunderlich, and J. Henkel, "Aging resilience and fault tolerance in runtime reconfigurable architectures," *IEEE Transactions on Computers*, 2016.

[217] F. Siegle, T. Vladimirova, J. Ilstad, and O. Emam, "Mitigation of radiation effects in SRAM-based FPGAs for space applications," *ACM Computing Surveys*, 2015.

[218] N. T. H. Nguyen, "Repairing FPGA configuration memory errors using dynamic partial reconfiguration," Ph.D. dissertation, The University of New South Wales, 2017.

[219] G. Rieke, M. Ressler, J. E. Morrison, L. Bergeron, P. Bouchet, M. García-Marín, T. Greene, M. Regan, K. Sukhatme, and H. Walker, "The mid-infrared instrument for the james webb space telescope, VII: the MIRI detectors," *Publications of the Astronomical Society of the Pacific*, vol. 127, no. 953, p. 665, 2015.

[220] D. Evans and M. Merri, "OPS-SAT: An ESA nanosatellite for accelerating innovation in satellite control," in *SpaceOps Conference*. ESA, 2014.

[221] H. Kayal, F. Baumann, K. Briess, and S. Montenegro, "Beesat: A pico satellite for the on orbit verification of micro wheels," in *IEEE International Conference on Recent Advances in Space Technologies (RAST)*. IEEE, 2007, pp. 497–502.

[222] S. Fitzsimmons, "Reliable software updates for on-orbit CubeSat satellites," Ph.D. dissertation, Master Thesis, California Polytechnic State University, 2012, 2012.

[223] S. Busch and K. Schilling, "UWE-3: a modular system design for the next generation of very small satellites," in *Small Satellites Systems and Services–The 4S Symposium, ESA Press*, 2012.

[224] A. Corporation, "Arria GX device handbook, volume 2: Jam STAPL," 2008.

[225] S. M. Guertin, "CubeSat and mobile processors," in *NASA Electronics Technology Workshop*, 2015, pp. 23–26.

[226] A. Pirovano, A. Lacaita, A. Benvenuti, F. Pellizzer, S. Hudgens, and R. Bez, "Scaling analysis of phase-change memory technology," in *International Electron Devices Meeting (IEDM)*. IEEE, 2003, pp. 29–6.

[227] Y. Huai, "Spin-transfer torque MRAM (STT-MRAM): Challenges and prospects," *AAPPS Bulletin, Association of Asia Pacific Physical Societies*, vol. 18, no. 6, pp. 33–40, 2008.

[228] K. Suzuki, D. Tonien, K. Kurosawa, and K. Toyota, "Birthday paradox for multi-collisions," in *International Conference on Information Security and Cryptology (ICISC), Springer*, 2006.

[229] S. Wicker and V. Bhargava, *Reed-Solomon codes and their applications*. John Wiley & Sons, 1999.

[230] F. Irom *et al.*, "SEEs and TID results of highly scaled flash memories," in *IEEE Radiation Effects Data Workshop (REDW)*, 2013.

[231] CNES, "Utilisation DSP FPGA Xilinx Spartan 6 pour application spatiale," 2013, dCT/AQ/EC-2012/0019591.

[232] ——, "Fiabilité d'un module de processing haute performance à base de FPGA CMP Xilinx Spartan 6," 2014, dCT/AQ/EC-2014/01646.

[233] ——, "Spécification technique de besoin pour évaluation en dose cumulée d'un FPGA CMP en Co60," 2015, dCT/AQ/EC-2015/01158.

[234] J. Heiner, B. Sellers, M. Wirthlin, and J. Kalb, "FPGA partial reconfiguration via configuration scrubbing," in *International Conference on Field Programmable Logic and Applications (FPL)*. IEEE, 2009, pp. 99–104.

[235] J. D. Corbett, "The xilinx isolation design flow for fault-tolerant systems," *Xilinx White Paper WP412*, vol. 53, 2012.

[236] L. Sterpone and B. Du, "SET-PAR: place and route tools for the mitigation of single event transients on flash-based FPGAs," in *Applied Reconfigurable Computing*. Springer, 2015, pp. 129–140.

[237] F. Kastensmidt and P. Rech, *FPGAs and Parallel Architectures for Aerospace Applications: Soft Errors and Fault-Tolerant Design*. Springer, 2016.

[238] M. Wirthlin, "High-reliability FPGA-based systems: space, high-energy physics, and beyond," *Proceedings of the IEEE*, vol. 103, no. 3, 2015.

[239] M. Ebrahimi, P. M. B. Rao, R. Seyyedi, and M. B. Tahoori, "Low-cost multiple bit upset correction in SRAM-based FPGA configuration frames," *IEEE Transactions on VLSI Systems*, 2016.

[240] F. Rittner, M. Ristic, R. Glein, and A. Heuberger, "Automated test procedure to detect permanent faults inside SRAM-based FPGAs," in *NASA/ESA Conference on Adaptive Hardware and Systems (AHS)*. IEEE, 2017.

[241] U. Martinez-Corral and K. Basterretxea, "A fully configurable and scalable neural coprocessor ip for soc implementations of machine learning applications," in *NASA/ESA Conference on Adaptive Hardware and Systems (AHS)*. IEEE, 2017.

[242] A. Stoddard, A. Gruwell, P. Zabriskie, and M. J. Wirthlin, "A hybrid approach to FPGA configuration scrubbing," *IEEE Transactions on Nuclear Science*, 2017.

[243] F. Siegle, T. Vladimirova, J. Ilstad, and O. Emam, "Availability analysis for satellite data processing systems based on SRAM FPGAs," *IEEE Transactions on Aerospace and Electronic Systems*, vol. 52, no. 3, pp. 977–989, 2016.

[244] S. Wang, Y. Higami, H. Takahashi, M. Sato, M. Katsu, and S. Sekiguchi, "Testing of interconnect defects in memory based reconfigurable logic device (MRLD)," in *IEEE Asian Test Symposium (ATS)*, 2017.

[245] A. Guerrieri, B. Belhadj, P. Lombardi, P. Ienne, and S. Kashani Akhavan, "FPGA based multithreading for on-board processing," in *SpacE FPGA Users Workshop*, 2018, ESA/CNES.

[246] A. K. Singh, M. Shafique, A. Kumar, and J. Henkel, "Mapping on multi/many-core systems: survey of current and emerging trends," in *ACM/EDAC/IEEE Design Automation Conference (DAC)*. ACM, 2013.

[247] E. Carvalho, N. Calazans, and F. Moraes, "Heuristics for dynamic task mapping in NoC-based heterogeneous MPSoCs," in *IEEE/IFIP International Workshop on Rapid System Prototyping (RSP)*. IEEE, 2007.

[248] RTEMS Development Team, "The real-time executive for multiprocessor systems RTOS," project website: www.rtems.org.

[249] J. Bouwmeester and J. Guo, "Survey of worldwide pico-and nanosatellite missions, distributions and subsystem technology," *Acta Astronautica, Elsevier*, vol. 67, no. 7, 2010.

[250] L. Z. Scheick, S. M. Guertin, and G. M. Swift, "Analysis of radiation effects on individual DRAM cells," *IEEE Transactions on Nuclear Science*, 2000.

[251] A. A. Hwang, I. A. Stefanovici, and B. Schroeder, "Cosmic rays don't strike twice: understanding the nature of DRAM errors and the implications for system design," *ACM SIGPLAN Notices*, 2012.

[252] D. Dopson, "SoftECC: A system for software memory integrity checking," Ph.D. dissertation, Massachusetts Institute of Technology, 2005.

[253] R. Goodman and M. Sayano, "On-chip ECC for multi-level random access memories," in *IEEE/CAM Information Theory Workshop at Cornell*. IEEE, 1989.

[254] D. Bhattacharryya and S. Nandi, "An efficient class of SEC-DED-AUED codes," in *International Symposium on Parallel Architectures, Algorithms and Networks (I-SPAN)*. IEEE, 1997.

[255] A. Samaras, F. Bezerra, E. Lorfevre, and R. Ecoffet, "Carmen-2: In flight observation of non destructive single event phenomena on memories," in *Conference on Radiation and its Effects on Components and Systems (RADECS)*. IEEE, 2011.

[256] Y. You and J. Hayes, "A self-testing dynamic RAM chip," *IEEE Transactions on Electron Devices*, 1985.

[257] D. Callaghan, "Self-testing RAM system and method," 2008, US Patent 7,334,159.

[258] J. Foley, "Adaptive memory scrub rate," 2012, US Patent 8,255,772.

[259] M. Stringfellow, N. Leveson, and B. Owens, "Safety-driven design for software-intensive aerospace and automotive systems," *Proceedings of the IEEE*, vol. 98, no. 4, pp. 515–525, 2010.

[260] K. Ryu, E. Shin, and V. Mooney, "A comparison of five different multiprocessor soc bus architectures," in *Euromicro Symposium on Digital Systems Design*. IEEE, 2001.

[261] D. McComas, "NASA/GSFC's flight software core flight system," *NASA*, 2012.

[262] J. Williams and N. Bergmann, "Reconfigurable linux for spaceflight applications," *Single Event Effects Symposium (SEE) & Military and Aerospace Programmable Logic Devices (MAPLD)*, 2004.

[263] D. Atienza, J. Mendias, S. Mamagkakis, D. Soudris, and F. Catthoor, "Systematic dynamic memory management design methodology for reduced memory footprint," *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, vol. 11, no. 2, pp. 465–489, 2006.

[264] J. Saleh, D. Hastings, and D. Newman, "Weaving time into system architecture: satellite cost per operational day and optimal design lifetime," *Acta Astronautica, Elsevier*, vol. 54, no. 6, pp. 413–431, 2004.

[265] M. Baker, M. Shah, D. Rosenthal, M. Roussopoulos, P. Maniatis, T. Giuli, and P. Bungale, "A fresh look at the reliability of long-term digital storage," in *ACM SIGOPS Operating Systems Review*, vol. 40. ACM, 2006, pp. 221–234.

[266] J. Engel and R. Mertens, "LogFS-finally a scalable flash file system," in *International Linux System Technology Conference (LinuxCon)*. Linux Foundation, 2005.

[267] S. Qiu and N. Reddy, "NVMFS: A hybrid file system for improving random write in NAND-flash SSD," in *Symposium on Mass Storage Systems and Technologies (MSST)*. IEEE, 2013.

[268] W. Liangzhu, "The investigation of JFFS2 storage," *Microcomputer Information*, vol. 8, p. 030, 2008.

[269] N. K. Edel, D. Tuteja, E. L. Miller, and S. A. Brandt, "MRAMFS: A compressing file system for non-volatile RAM," in *Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunications Systems (MASCOTS)*. IEEE, 2004, pp. 596–603.

[270] M. Stornelli, "Protected and persistent RAM filesystem," pramfs.sourceforge.net.

[271] J. Hulbert, "The Advanced XIP file system," in *Linux Symposium (OLS)*. Linux Foundation, 2008, p. 211.

[272] D. Nguyen and F. Irom, "Radiation effects on MRAM," in *Conference on Radiation and its Effects on Components and Systems (RADECS)*. IEEE, 2007.

[273] M. Elghefari and S. McClure, "Radiation effects assessment of MRAM devices," NASA/JPL, Tech. Rep., 2008.

[274] M. Cassel, D. Walter, H. Schmidt, F. Gliem, H. Michalik, M. Stähle, K. Vögele, and P. Roos, "NAND-flash memory technology in mass memory systems for space applications," in *Eurospace Data Systems In Aerospace (DASIA)*, 2008.

[275] H. Herpel, M. Stähle, U. Lonsdorfer, and N. Binzer, "Next generation mass memory architecture," in *Eurospace Data Systems In Aerospace (DASIA)*, 2010.

[276] S. Suzuki and K. Shin, "On memory protection in real-time os for small embedded systems," in *Workshop Real-Time Computing Systems and Applications (RTCSA)*. IEEE, 1997.

[277] S. Su and E. DuCasse, "A hardware redundancy reconfiguration scheme for tolerating multiple module failures," *IEEE Transactions on Computers*, vol. 100, no. 3, pp. 254–258, 1980.

[278] B. Cagno, J. Elliott, R. Kubo, and G. Lucas, "Verifying data integrity of a non-volatile memory system during data caching process," US Patent 8,037,380.

[279] V. Prabhakaran, A. Arpaci-Dusseau, and R. Arpaci-Dusseau, "Analysis and evolution of journaling file systems." in *USENIX Annual Technical Conference*, 2005.

[280] M. Cropper *et al.*, "VIS: the visible imager for Euclid," in *SPIE Astronomical Telescopes + Instrumentation*, 2012.

[281] ESA/SRE, *JUICE Definition Study Report.* ESA, September 2014.

[282] ——, *EUCLID Definition Study Report.* ESA, July 2011.

[283] K. F. Strauss and T. Daud, "Overview of radiation tolerant unlimited write cycle non-volatile memory," in *IEEE Aerospace Conference*, 2000.

[284] A. P. Ferreira, B. Childers, R. Melhem, D. Mossé, and M. Yousif, "Using PCM in next-generation embedded space applications," in *RTAS.* IEEE, 2010.

[285] N. Gupta, B. Vermeire, H. Barnaby, M. Goksel, E. Li, and D. Czajkowski, "Design of a 1 Gb radiation hardened NAND flash memory," in *Non-Volatile Memory Technology Symposium.* IEEE, 2007.

[286] S. Suzuki, Y. Deguchi, T. Nakamura, K. Mizoguchi, and K. Takeuchi, "Error elimination ECC by horizontal error detection and vertical-LDPC ECC to increase data-retention time by 230% and acceptable bit-error rate by 90% for 3D-NAND flash SSDs," in *IEEE International Reliability Physics Symposium (IRPS).* IEEE, 2018, pp. P–MY.

[287] F. Irom, D. N. Nguyen, M. L. Underwood, and A. Virtanen, "Effects of scaling in SEE and TID response of high density NAND flash memories," *IEEE Transactions on Nuclear Science*, vol. 57, no. 6, pp. 3329–3335, 2010.

[288] S. Zertal, "A reliability enhancing mechanism for a large flash embedded satellite storage system," in *IEEE International Conference on Systems (ICONS)*, 2008.

[289] B. Kroth and S. Yang, "Checksumming RAID," 2010, university of Wisconsin-Madison, unpublished manuscript.

[290] E. M. Kurtas, A. V. Kuznetsov, and I. Djurdjevic, "System perspectives for the application of structured LDPC codes to data storage," *IEEE Transactions on Magnetics*, vol. 42, 2006.

[291] K. S. Andrews, D. Divsalar, S. Dolinar, J. Hamkins, C. R. Jones, and F. Pollara, "The development of Turbo and LDPC codes for deep-space applications," *Proceedings of the IEEE*, vol. 95, no. 11, 2007.

[292] M. Lentmaier, A. Sridharan, D. J. Costello, and K. S. Zigangirov, "Iterative decoding threshold analysis for LDPC convolutional codes," *IEEE Transactions on Information Theory*, 2010.

[293] T. Morita, M. Ohta, and T. Sugawara, "Efficiency of short LDPC codes combined with long reed-solomon codes for magnetic recording channels," *IEEE Transactions on Magnetics*, vol. 40, no. 4, pp. 3078–3080, 2004.

[294] Z. Shi, C. Fu, and S. Li, "Serial concatenation and joint iterative decoding of LDPC codes and Reed-Solomon codes," *National Laboratory of Communication, UESTC, Chengdu, China*, vol. 610054, 2006.

[295] P. Sobe, "Reliability modeling of fault-tolerant storage system-covering MDS-codes and regenerating codes," in *International Conference on Architecture of Computing Systems (ARCS)*, 2013.

[296] M. Nowak, S. Lacour, A. Crouzier, L. David, V. Lapeyrère, and G. Schworer, "Short life and abrupt death of picsat, a small 3u CubeSat dreaming of exoplanet detection," in *Space Telescopes and Instrumentation 2018: Optical, Infrared, and Millimeter Wave*, vol. 10698. International Society for Optics and Photonics, 2018, p. 1069821.

[297] D. S. Lee, G. R. Allen, G. Swift, M. Cannon, M. Wirthlin, J. S. George, R. Koga, and K. Huey, "Single-event characterization of the 20 nm Xilinx Kintex Ultrascale field-programmable gate array under heavy ion irradiation," in *IEEE Radiation Effects Data Workshop (REDW)*. IEEE, 2015.

[298] M. Glorieux, A. Evans, T. Lange, A.-D. In, D. Alexandrescu, C. Boatella-Polo, R. G. Alía, M. Tali, C. U. Ortega, M. Kastriotou *et al.*, "Single-event characterization of Xilinx UltraScale+ MPSoC under standard and ultra-high energy heavy-ion irradiation," in *IEEE Nuclear & Space Radiation Effects Conference (NSREC)*. IEEE, 2018, pp. 1–5.

[299] D. S. Lee, M. King, W. Evans, M. Cannon, A. Pérez-Celis, J. Anderson, M. Wirthlin, and W. Rice, "Single-event characterization of 16 nm FinFET Xilinx UltraScale+ devices with heavy ion and neutron irradiation," in *IEEE Nuclear & Space Radiation Effects Conference (NSREC)*. IEEE, 2018.

[300] R. Natella, D. Cotroneo, and H. S. Madeira, "Assessing dependability with software fault injection: A survey," *ACM Computing Surveys*, 2016.

[301] B. Sangchoolie, R. Johansson, and J. Karlsson, "Light-weight techniques for improving the controllability and efficiency of isa-level fault injection tools," in *IEEE Pacific Rim International Symposium on Dependable Computing (PRDC)*. IEEE, 2017.

[302] D. Cotroneo, A. Lanzaro, R. Natella, and R. Barbosa, "Experimental analysis of binary-level software fault injection in complex software," in *European Dependable Computing Conference (EDCC)*. IEEE, 2012.

[303] R. Natella, D. Cotroneo, J. A. Duraes, and H. S. Madeira, "On fault representativeness of software fault injection," *IEEE Transactions on Software Engineering*, vol. 39, no. 1, pp. 80–96, 2013.

[304] L. Leem, H. Cho, J. Bau, Q. A. Jacobson, and S. Mitra, "ERSA: Error resilient system architecture for probabilistic applications," in *Conference on Design, Automation and Test in Europe (DATE)*. EDAA, 2010.

[305] R. Amarnath, S. N. Bhat, P. Munk, and E. Thaden, "A fault injection approach to evaluate soft-error dependability of system calls," in *IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*. IEEE, 2018, pp. 71–76.

[306] H. Schirmeier, M. Hoffmann, C. Dietrich, M. Lenz, D. Lohmann, and O. Spinczyk, "FAIL: An open and versatile fault-injection framework for the assessment of software-implemented hardware fault tolerance," in *European Dependable Computing Conference (EDCC)*. IEEE, 2015.

[307] J. Isaza-González, A. Serrano-Cases, F. Restrepo-Calle, S. Cuenca-Asensi, and A. Martínez-Álvarez, "Dependability evaluation of cots microprocessors via on-chip debugging facilities," in *IEEE Latin American Test Symposium (LATS)*, 2016.

[308] D. Cozzi, "Run-time reconfigurable, fault-tolerant FPGA systems for space applications," Ph.D. dissertation, Universität Bielefeld, 2016.

[309] M. Alderighi, F. Casini, S. D'Angelo, S. Pastore, G. Sechi, and R. Weigand, "Evaluation of single event upset mitigation schemes for SRAM based FPGAs using the FLIPPER fault injection platform," in *IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFT)*. IEEE, 2007.

[310] W. Mansour and R. Velazco, "An automated SEU fault-injection method and tool for HDL-based designs," *IEEE Transactions on Nuclear Science*, 2013.

[311] D. Cotroneo and R. Natella, "Software fault injection for software certification," *IEEE Security & Privacy*, 2013.

[312] M. Kooli, P. Benoit, G. Di Natale, L. Torres, and V. Sieh, "Fault injection tools based on virtual machines," in *Reconfigurable and Communication-Centric Systems-on-Chip (ReCoSoC)*. IEEE, 2014.

[313] A. Höller, G. Schönfelder, N. Kajtazovic, T. Rauter, and C. Kreiner, "FIES: a fault injection framework for the evaluation of self-tests for COTS-based safety-critical systems," in *International Microprocessor Test and Verification Workshop (MTV)*. IEEE, 2014.

[314] J. Power, J. Hestness, M. S. Orr, M. D. Hill, and D. A. Wood, "gem5-gpu: A heterogeneous cpu-gpu simulator," *IEEE Computer Architecture Letters*, vol. 14, no. 1, pp. 34–36, 2015.

[315] P. Lisherness and K.-T. T. Cheng, "SCEMIT: A SystemC error and mutation injection tool," in *Design Automation Conference (DAC)*. ACM, 2010.

[316] R. Natella, S. Winter, D. Cotroneo, and N. Suri, "Analyzing the effects of bugs on software interfaces," *IEEE Transactions on Software Engineering*, 2018.

[317] D. Sinclair and J. Dyer, "Radiation effects and cots parts in smallsats," in *AIAA/USU Conference on Small Satellites (SmallSat)*, 2013.

[318] R. L. Pease, A. H. Johnston, and J. L. Azarewicz, "Radiation testing of semiconductor devices for space electronics," *Proceedings of the IEEE*, vol. 76, no. 11, pp. 1510–1526, 1988.

[319] M. A. McMahan, E. Blackmore, E. W. Cascio, C. Castaneda, B. von Przewoski, and H. Eisen, "Standard practice for dosimetry of proton beams for use in radiation effects testing of electronics," in *IEEE Radiation Effects Data Workshop (REDW)*. IEEE, 2008, pp. 135–141.

[320] V. Sridharan and D. R. Kaeli, "Using hardware vulnerability factors to enhance avf analysis," in *ACM SIGARCH Computer Architecture News*, vol. 38, no. 3. ACM, 2010, pp. 461–472.

[321] M. Maniatakos, M. K. Michael, and Y. Makris, "Investigating the limits of avf analysis in the presence of multiple bit errors," in *IEEE International Symposium on On-Line Testing and Robust System Design (IOLTS)*. IEEE, 2013, pp. 49–54.

[322] X. Li and D. Yeung, "Application-level correctness and its impact on fault tolerance," in *International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2007.

[323] A. o. Velasco, "A hardening approach for the scheduler's kernel data structures," in *International Conference on Architecture of Computing Systems (ARCS)*, 2017.

[324] A. Serrano-Cases, Y. Morilla, P. Martın-Holgado, S. Cuenca-Asensi, and A. Martınez-Álvarez, "Automatic compiler-guided reliability improvement of embedded processors under proton irradiation," in *Conference on Radiation and its Effects on Components and Systems (RADECS)*. IEEE, 2018.

[325] A. Serrano-Cases, J. Isaza-González, S. Cuenca-Asensi, and A. Martínez-Álvarez, "On the influence of compiler optimizations in the fault tolerance of embedded systems," in *IEEE International Symposium on On-Line Testing and Robust System Design (IOLTS)*. IEEE, 2016.

[326] F. M. Lins, L. A. Tambara, F. L. Kastensmidt, and P. Rech, "Register file criticality and compiler optimization effects on embedded microprocessor reliability," *IEEE Transactions on Nuclear Science*, 2017.

[327] P. Larsen, A. Homescu, S. Brunthaler, and M. Franz, "Sok: Automated software diversity," in *IEEE Symposium on Security and Privacy (SP)*. IEEE, 2014, pp. 276–291.

[328] P. Meloni *et al.*, "System adaptivity and fault-tolerance in NoC-based MPSoCs: the MADNESS project approach," in *IEEE DSD*, 2012.

[329] N. K. R. Beechu, V. M. Harishchandra, and N. K. Y. Balachandra, "Hardware implementation of fault tolerance NoC core mapping," *Springer Telecommunication Systems*, 2017.

[330] N. Katta, H. Zhang, M. Freedman, and J. Rexford, "Ravana: Controller fault-tolerance in software-defined networking," in *ACM SIGCOMM*. ACM, 2015.

[331] Z. K. Baker and H. M. Quinn, "Design and test of Xilinx embedded ECC for MicroBlaze processors," in *IEEE Radiation Effects Data Workshop (REDW)*. IEEE, 2016, pp. 1–7.

[332] Z. Zhang, Z. Lei, Z. Yang, X. Wang, B. Wang, J. Liu, Y. En, H. Chen, and B. Li, "Single event effects in COTS ferroelectric RAM technologies," in *IEEE Radiation Effects Data Workshop (REDW)*. IEEE, 2015.

[333] Y. Li, B. Nelson, and M. Wirthlin, "Synchronization techniques for crossing multiple clock domains in FPGA-based TMR circuits," *IEEE Transactions on Nuclear Science*, vol. 57, no. 6, pp. 3506–3514, 2010.

[334] J. Standeven, M. J. Colley, and D. Lyons, "Hardware voter for fault-tolerant transputer systems," *Microprocessors and Microsystems, Elsevier*, vol. 13, no. 9, pp. 588–596, 1989.

[335] A. T. Tai, S. N. Chau, and L. Alkalai, "COTS-based fault tolerance in deep space: Qualitative and quantitative analyses of a bus network architecture," in *International Symposium on High-Assurance Systems Engineering (HASE)*. IEEE, 1999.

[336] H. Kimm and M. Jarrell, "Controller area network for fault tolerant small satellite system design," in *IEEE International Symposium on Industrial Electronics (ISIE)*. IEEE, 2014, pp. 81–86.

[337] C. Wilson, J. MacKinnon, P. Gauvin, S. Sabogal, A. D. George, G. Crum, and T. Flatley, "μcsp: A diminutive, hybrid, space processor for smart modules and CubeSats," in *AIAA/USU Conference on Small Satellites (SmallSat)*, 2016.

[338] S. Parkes and P. Armbruster, "SpaceWire: a spacecraft onboard network for real-time communications," in *IEEE-NPSS Real Time Conference (RT)*. IEEE, 2005.

[339] H. Zimmermann, "OSI reference model–the ISO model of architecture for open systems interconnection," *IEEE Transactions on communications*, 1980.

[340] V. Gavrilut, B. Zarrin, P. Pop, and S. Samii, "Fault-tolerant topology and routing synthesis for IEEE time-sensitive networking," in *International Conference on Real-Time Networks and Systems (RTNS)*. ACM, 2017.

[341] G. J. Brebner, "Reconfigurable computing for high performance networking applications." *ARC*, vol. 1, 2011.

[342] J. Anderson, K. Bauer, A. Borga, H. Boterenbrood, H. Chen, K. Chen, G. Drake, M. Dönszelmann, D. Francis, D. Guest *et al.*, "Felix: a pcie based high-throughput approach for interfacing front-end and trigger electronics in the atlas upgrade framework," *Journal of Instrumentation, IOP Publishing*, 2016.

[343] M. Dreschmann, J. Heisswolf, M. Geiger, J. Becker, and M. HauBecker, "A framework for multi-FPGA interconnection using multi gigabit transceivers," in *Symposium on Integrated Circuits and Systems Design (SBCCI)*. IEEE, 2015.

[344] C. Carmichael, "Triple module redundancy design techniques for Virtex FPGAs," *Xilinx Application Note XAPP197*, 2001.

[345] A. Fedi, M. Ottavi, G. Furano, A. Bruno, R. Senesi, C. Andreani, and C. Cazzaniga, "High-energy neutrons characterization of a safety critical computing system," in *IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFT)*. IEEE, 2017, pp. 1–4.

[346] Á. B. de Oliveira, L. A. Tambara, and F. L. Kastensmidt, "Applying lockstep in dual-core ARM Cortex-A9 to mitigate radiation-induced soft errors," in *IEEE Latin American Symposium on Circuits & Systems (LASCAS)*. IEEE, 2017.

[347] R. V. Kshirsagar and R. M. Patrikar, "Design of a novel fault-tolerant voter circuit for TMR implementation to improve reliability in digital circuits," *Microelectronics Reliability, Elsevier*, 2009.

[348] M. Liu and B. H. Meyer, "Bounding error detection latency in safety critical systems with enhanced execution fingerprinting," in *IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFT)*. IEEE, 2016.

[349] E. Wachter, V. Fochi, F. Barreto, A. Amory, and F. Moraes, "A hierarchical and distributed fault tolerant proposal for NoC-based MPSoCs," *IEEE Transactions on Emerging Topics in Computing*, 2016.

[350] W. Liu, W. Zhang, X. Wang, and J. Xu, "Distributed sensor network-on-chip for performance optimization of soft-error-tolerant multiprocessor system-on-chip," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 24, no. 4, pp. 1546–1559, 2016.

[351] S. S. Sahoo, B. Veeravalli, and A. Kumar, "Cross-layer fault-tolerant design of real-time systems," in *IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFT)*. IEEE, 2016.

[352] E. Benton and E. Benton, "Space radiation dosimetry in low-earth orbit and beyond," *Nuclear Instruments and Methods in Physics Research, Elsevier*, 2001.

[353] V. Sridharan and D. Liberty, "A study of DRAM failures in the field," in *Conference on High Performance Computing, Networking, Storage and Analysis (SC)*. IEEE, 2012.

[354] P. Maillard, M. Hart, J. Barton, P. Chang, M. Welter, R. Le, R. Ismail, and E. Crabill, "Single-event upsets characterization & evaluation of Xilinx UltraScale soft error mitigation (SEM IP) tool," in *IEEE Radiation Effects Data Workshop (REDW)*. IEEE, 2016, pp. 1–4.

[355] C. Bolchini, A. Miele, and M. D. Santambrogio, "TMR and partial dynamic reconfiguration to mitigate SEU faults in FPGAs," in *IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFT)*. IEEE, 2007, pp. 87–95.

[356] G. Durrieu, G. Fohler, G. Gala, S. Girbal, D. G. Pérez, E. Noulard, C. Pagetti, and S. Pérez, "Dreams about reconfiguration and adaptation in avionics," *Embedded Real Time Software and Systems Congress (ERTS)*, 2016.

[357] M. Darvishi, Y. Audet, Y. Blaquière, C. Thibeault, and S. Pichette, "On the susceptibility of SRAM-based FPGA routing network to delay changes induced by ionizing radiation," *IEEE Transactions on Nuclear Science*, 2019.

[358] M. Payer, "Too much PIE is bad for performance," *ETH Zurich Technical Report*, vol. 766, 2012.

[359] J. W. Lee, M. C. Ng, and K. Asanovic, "Globally-synchronized frames for guaranteed quality-of-service in on-chip networks," in *ACM SIGARCH Computer Architecture News*, vol. 36, no. 3. IEEE Computer Society, 2008, pp. 89–100.

[360] TSMC's industry-first and leading 7nm technology enters volume production. [Online]. Available: https://www.tsmc.com/csr/en/update/innovationAndService/caseStudy/9/index.html

[361] S.-D. Kim, M. Guillorn, I. Lauer, P. Oldiges, T. Hook, and M.-H. Na, "Performance trade-offs in FinFET and gate-all-around device architectures for 7nm-node and beyond," in *IEEE SOI-3D-Subthreshold Microelectronics Technology Unified Conference (S3S)*. IEEE, 2015, pp. 1–3.

[362] Z. Zhang, Z. Zhan, D. Balasubramanian, X. Koutsoukos, and G. Karsai, "Triggering rowhammer hardware faults on arm: A revisit," in *Workshop on Attacks and Solutions in Hardware Security*. ACM, 2018, pp. 24–33.

Nederlandse Samenvatting

Moderne semiconductortechnologie maakt het mogelijk om geminiaturiseerde satellieten te bouwen, die goedkoop zijn om te lanceren en een betaalbaar platform bieden voor vele verschillende wetenschappelijke en commerciële instrumenten. Vooral de kleinste en lichtste satellieten maken ruimtemissies mogelijk die voorheen technisch niet haalbaar, onpraktisch of simpelweg oneconomisch te duur waren. Vooral zogenaamde CubeSats kunnen snel tegen lage kosten gebouwd worden, met beperkte hulpmiddelen in een academische omgeving. Dit soort satellieten heeft echter te maken met een lage betrouwbaarheid. Daarom zijn ze tot op heden voornamelijk gebruikt voor minder kritieke missies en missies met een laag budget, waar risico's genomen kunnen worden.

Vele geavanceerde wetenschappelijke en commerciële toepassingen passen tegenwoordig in een geminiaturiseerde satelliet, waarvoor een lange missieduur wenselijk is. Theoretisch kunnen zulke ruimtevaartuigen gebruikt worden voor verscheidene kritieke en complexe missies, waaronder wetenschappelijke hoge-prioriteitsmissies binnen het Zonnestelsel of voor astronomische toepassingen. Echter, door hun lage betrouwbaarheid zijn dit soort ruimtevaartuigen tot nu toe alleen gebruikt op ruimtemissies voor secundaire taken.

Miniatuursatellieten bestaan voor een groot gedeelte uit elektronica welke zijn verantwoordelijk voor een groot gedeelte van de kritieke subsystemen. Gezien het lage gewicht van de satelliet, moet deze elektronica lichter, kleiner en energiezuiniger zijn dan traditionele ruimtevaartcomponenten. Daarom gebruiken alle geavanceerde CubeSats tegenwoordig hoogwaardige computerontwerpen die gebaseerd zijn op commerciëel verkrijgbare ontwerpen voor de ingebed systemen en de mobiele markt. Tegen minimale kosten geeft dit soort elektronica hoge prestaties, verbruikt het minder energie en is het makkelijker om mee te werken dan hun tegenhangers die historisch voor ruimtemissies zijn ontwikkeld.

Conventionele computers die gebaseerd zijn op het System-on-chip-principe, missen echter de fouttolerantie van computerontwerpen op grotere ruimtevaartuigen. Subsystemen die draaien op dit soort componenten zijn verantwoordelijk voor de meeste storingen nadat miniatuur satellieten gelanceerd zijn en ingezet zijn in de ruimte. Door budget-, energie-, massa- en volumebeperkingen van miniatuursatellieten kunnen fouttolerantiesystemen van bestaande grotere ruimtevaartuigen niet toegepast worden.

Op het moment van schrijven bestaat er geen fouttolerante computerarchitectuur, bestaand uit halfgeleiders uit de mobiele markt, die aan boord van miniatuursatellieten gebruikt kan worden zonder te breken met het fundamentele concept van goedkope, simpele, energiezuinige en lichtgewicht satellieten die op grote schaal geproduceerd en tegen lage kosten gelanceerd kunnen worden. Ontwerpers van miniatuursatellieten

hebben daarom de volgende opties:

Vergroten: Gebruik maken van traditionele ruimtevaartcomponenten. Dit gaat meestal samen met het vergroten van het ontwerp van het ruimtevaartuig, aangezien zulke componenten meer energie gebruiken en minder functionaliteit, flexibiliteit en rekenkracht bieden. In de praktijk verhoogt dit de kosten, benodigde mankracht en ontwikkelingstijd drastisch. Daarom is deze aanpak niet praktisch voor de meeste nieuwe ruimtemissieconcepten, die juist gebruik willen maken van ruimtevaartuigen die snel ontwikkeld kunnen worden of die klein of goedkoop moeten zijn.

SpareSats: Het risico van vroege uitval verminderen door één of meerdere SpareSats te bouwen die een CubeSat vervangen zodra deze een storing heeft. In de praktijk verhoogt dit niet alleen de kosten, maar worden storingen ook waarschijnlijker aangezien het aantal componenten dat wordt gelanceerd nam drastisch toe. Daarom wordt deze aanpak alleen rendabel zodra systemen robuust genoeg zijn. Op dit moment heeft deze aanpak alleen nut voor satellietnetwerken, waarbij satellieten in hoog tempo en continue vervangen worden (bijv. Planet Lab) en individuele satellieten die werken met een uitzonderlijk groot budget (bijv. MarCo).

Accepteren: Het gebrek aan betrouwbaarheid accepteren. De ruimtemissie is bewust van korte duur, in de hoop dat alle missiedoelen gehaald worden voordat de satelliet uitvalt. Voor toekomstige langdurige missies met miniatuursatellieten, kan geluk geen factor zijn waarop het systeem is gebaseerd.

Op het moment van schrijven van deze thesis, zijn de meeste ontwikkelaars van miniatuursatellieten gedwongen om de derde optie te volgen. Voor simpele en korte CubeSatmissies resulteert deze aanpak meestal in succes, maar ook in vele vroege storingen. Gokken tegen tijd en hopen dat een satelliet niet beïnvloed zal worden door processen in de ruimte is echter onacceptabel en wordt steeds minder getolereerd door overheden, ruimtevaartorganisaties en investeerders. Om success te garanderen voor geavanceerde en langdurige CubeSat missies, zijn betere en betrouwbaardere systeemontwerpen nodig. Daarom zijn fouttolerante concepten nodig die geschikt zijn voor computers, gebaseerd op moderne commerciële halfgeleiders, in CubeSats.

De resultaten van deze thesis

Om de technologische tekortkomingen van kleine satellieten te overwinnen wordt in deze thesis een nieuw fouttolerante computer architectuur gepresenteerd. Deze architectuur is geschikt om in lichte wetenschappelijke CubeSats ingebouwd te worden, die gebruik maken van moderne commerciële halfgeleiders.

Om de architectuur die gepresenteerd wordt in deze thesis te ontwikkelen, worden resultaten en concepten uit verschillende wetenschappelijke vakgebieden en het ingenieurswezen gebruikt, waarbij de ontwikkeling van deze architectuur beide vakgebieden overstijgt. Daarom combineren wij het beste van twee werelden: we integreren

wetenschappelijke vooruitgang, conceptuele en theoretische kennis met een praktische implementatie en de grondige tests die standaard zijn in het vakgebied van ruimte- en elektronische bouwkunde.

Om de resultaten van dit onderzoek toegankelijk te maken voor zowel wetenschappers en ingenieurs, zijn Hoofdstukken 2 en 3 bedoeld als informele introductie en definitie van het foutmodel dat gepresenteerd wordt in deze thesis. Hoofdstuk 2 bevat een kort overzicht van essentiële aspecten van hedendaagse ruimtevaart, voor lezers die onbekend zijn met dit onderwerp. Dit hoofdstuk fungeert als motivatie voor deze thesis, en introduceert ook concepten die gerelateerd zijn aan fouttolerante computerontwerpen. Om een effectief en efficient fouttolerant computersysteem te ontwerpen en ontwikkelen, is het essentieel om het effect van de ruimte op een computer te begrijpen. Hoofdstuk 3 behandelt deze effecten in detail, beperkingen voor ruimteelektronica en overwegingen gedurende de ruimtemissie, zoals communicatietijd en hemelmechanica.

Gebaseerd op de voorgaande hoofdstukken, presenteren we in Hoofdstuk 4 een fouttolerante computer architectuur, die softwarematige fouttolerantieconcepten combineert met FPGA[1] herconfiguratie en mixed criticality. Dit wordt verder aangevuld met verscheidene conventionele fouttolerantietechnieken en correctiemaatregelen. Fouttolerantie in deze architectuur wordt geïmplementeerd in verschillende, onderling gelinkte, stappen, die een lange levensduur van computers aan boord van kleine satellieten garanderen.

Voor deze functionaliteit gebruiken we een softwarematige coarse grain lockstep, die in detail wordt beschreven in Hoofdstuk 4. Deze functionaliteit alleen biedt uitstekende fouttolerantie, maar niet genoeg voor langdurige ruimtemissies. Daarom beschrijven we in Hoofdstuk 5 hoe herconfigureerbare logica gebruikt kan worden om vele verschillende systeemfouten te herstellen. Wij gebruiken FPGA herconfiguratie om de integriteit van een system-on-chipontwerp te garanderen, zodat de bruikbare levensduur van de computer verlengd kan worden. Op den duur zullen tijdens een langdurige ruimtemissie defecte onderdelen van een FPGA niet meer te herstellen zijn. Daarom zal de hoeveelheid intacte programmeerbare logica die beschikbaar is in een computer met de tijd afnemen. In Hoofdstuk 6 tonen we hoe mixed criticality een computer kan helpen zich aan te passen aan systeemfouten, in plaats van spontaan uit te vallen zoals traditionele systemen doen. Wij kunnen deze functionaliteit gebruiken om autonoom systeemprestatie in te ruilen voor energiezuinigheid en robuustheid tijdens de runtime. Dit zorgt er voor dat de kernfunctionaliteit van de computer bewaard blijft en het maximaliseert de overlevingskansen van de satelliet.

Al deze functionaliteit bestaat als software die draait op een multiprocessor system-on-chip die geïmplementeerd is in FPGA. Software, ladinginformatie en logica geprogrammeerd in een FPGA zijn data waarvan de integriteit bewaard moet blijven gedurende de gehele ruimtemissie. In Hoofdstuk 7 worden beschermende concepten voor verschillende soorten geheugentechnologiën aan boord van moderne satellieten beschreven.

De voorgaande, op software gebaseerde, fouttolerantieconcepten die toegepast kunnen worden op moderne halfgeleiders klinken vaak goed in theorie, maar blijken in realiteit onpraktisch te zijn. Op het moment van schrijven is er nog geen fouttolerante architectuur geïmplementeerd en getest, terwijl dit wel een kritieke stap is. In Hoofdstuk 8 tot 10 van deze thesis nemen wij deze kritieke stap.

De lockstepfunctionaliteit die gebruikt wordt in onze architectuur wordt getest

[1] field-programmable gate array

door middel van foutinjectie in Hoofdstuk 8. In Hoofdstuk 9 beschrijven wij een praktisch multiprocessor system-on-chipontwerp dat geïmplementeerd kan worden in een FPGA, wat een ideaal platform is voor deze architectuur. Hoofdstuk 10 is gewijd aan de praktische implementatie van de concepten en ontwerpen die beschreven worden in voorgaande hoofdstukken. Hierdoor kunnen we laten zien hoe een computer met deze architectuur aan boord van een miniatuursatelliet er in het echt uitziet, door een test opstelling te bouwen met ontwikkelingskit. Dit werdt gedaan met de volgende zes Xilinx FPGAs:

- Kintex UltraScale KU60,
- Kintex UltraScale+ KU11p, KU3p, de KU5p van een Xilinx KCU116 ontwikkelingskit en het
- Virtex UltraScale+ VU9P of a Xilinx VCU118 ontwikkelingskit.

Voor drie van deze FPGAs, de KU60, KU11p en KU3p, geven we gedetailleerde gebruiks- en stroomverbruikdata.

Conclusies

Aan het begin van deze thesis begonnen we met de volgende vraag:

Kan een fouttolerante computerarchitectuur gemaakt worden met moderne, mobiele-markttechnologie, zonder over de massa-, ruimte-, complexiteit- en budgetbeperkingen van miniatuursatellieten te gaan?

En PhD, vele gepubliceerde onderzoeksartikelen en verschillende ongelukken later, is het nu mogelijk om deze vraag op de volgende manier te beantwoorden:

Ja. *Een fouttolerante computerarchitectuur voor miniatuursatellieten is technisch mogelijk met bestaande commerciële en industriële technologie. Zodra alle componenten samengevoegd zijn tot een prototype, kan deze architectuur gebruikt worden om de levensduur van moderne CubeSats drastisch te verlengen, waardoor ze bruikbaar worden voor kritieke en langetermijn ruimtemissies.*

De softwarecomponenten van de architectuur die gepresenteerd wordt in deze thesis kunnen zonder grote ingrepen geïmplementeerd worden. Ze bieden bescherming voor bestaande toepassingen, zonder dat deze programma's herschreven moeten worden. Met bestaande software tonen wij aan dat deze mechanismes fouten snel kunnen detecteren en met hoge zekerheid en nauwkeurigheid en dat succesvol van fouten hersteld kan worden, tegen in de meeste gevallen lage computationele kosten. We demonstreren dat de prestaties van deze architectuur economisch zijn en effectief blijven, zelfs wanneer ze gebruikt worden in gebieden van de ruimte met uitzonderlijk hoge hoeveelheden straling.

Met bestaande commerciële componenten kan een system-on-chipontwerp, dat geldt als ideaal platform voor deze architectuur, zelfs geïmplementeerd worden in de kleinste Ultrascale+ FPGA die slechts 1.94W aan energie verbruikt. Daarom kan deze computerarchitectuur toegepast worden op satellieten ter grootte van 2U CubeSats.

Aangezien de grootte van een systeem bepaald wordt door technologie, zal vooruitgang in halfgeleiderproductie in de volgende generatie FPGAs deze aanpak nog aantrekkelijker maken en bruikbaar voor nog kleinere ruimtevaartuigen. Ook kan het de efficiëntie en schaalbaarheid aan boord van zwaardere ruimtevaartuigen verbeteren. Hopelijk kunnen we op deze manier in de toekomst de gebieden ver buiten de grenzen van het zonnestelsel onderzoeken.

中文摘要（简体）

现代半导体科技让小型卫星的建造不再是梦想。 其低廉的发射成本，能够在有限的预算情况下，实现多样的科学及商业途用。 也就是说，这些小而且轻的卫星可以实现过去技术无法完成，或过于昂贵的太空任务。 这些小型卫星，尤其是立方卫星（CubeSats），其的造价便宜，而且可以快速生产，这使得即使在有限的学术资源环境下也能够进行卫星的研究。 但是由于其可靠性较低，当今这种小型卫星只能够应用于对安全系数要求不高且预算较低的任务。

现代许多成熟的科学及商业应用可以适用于这种小型卫星。 这就使得小型卫星的持久续性越来越受到重视。 理论上来说，这种小型卫星也可以完成许多极关键且复杂的多面任务，比如太阳系的探勘以及天文学的应用。 然而正如前文所言，由于低可靠性，这种小型卫星只能够执行一些次要任务。

小型卫星的关键子系统的建造依赖于电子制造工业，因此电子工业在这种小型卫星中扮演极其重要的角色。 由于小型卫星的整体必须轻巧，所以相关电子零件必须更轻、更小，还要具有比传统太空等级零件更好的效能功耗比。 因此，所有先进的立方卫星都使用了最尖端的工业级嵌入式以及商业通信系统。 这些零件不仅价格低廉，而且能够提供充足的效能，消耗更少的能源，同时相对于传统的太空等级元件更容易使用。

然而，传统基于片上系统（System-on-Chip-based）的计算机并不具备较大型的太空船所另有的容错能力。 相关的研究表明，使用这种基于片上系统的组件是太空船在发射和部署过程中主要的故障起原因。 而且，由于受到预算、能耗、重量及空间的限制，当今应用于较大型太空船的电脑容错技术，仍无法应用小型卫星。

截至2019年，尚没有任何一个可容错的计算机架构，可以在不破坏低价、简单、轻盈、节能的原则下，将嵌入式及商业手机中的半导体应用在这类可被大量生产且易于发射的小型卫星上。 因此，这些小型卫星的设计者有以下三个选择：

尺度提升： 采用传统的太空零件。 这通常需要将整个太空船的设计变大，而且这种零件的功耗较高、功能较少，还缺少设计弹性，甚至计算效能也较低。 实际上，这会大幅增加开发成本、人力开销以及卫星的开发时间。 因此，这种方法对于那些要求开发迅速、太空船体积要小、价格便宜或可支付的起的先进设计理念来说，是不可行的。

备用卫星： 部署一或多个备用卫星（SpareSats）以缓解立方卫星在早期发生错误的风险。 实际上，随着发射上去的零件数量的增加，不仅增加成本，还会让错误发生几率升高。 因此，这个方法只能在系统达到一定的稳定性之后才能使用。 现在这个方法只能用在卫星世代频繁更替的星座计划（Constellation Missions），例如Planet Lab，以及具有超多预算的卫星，例如Marco。

接受： 接受其不可靠的事实。 让任务保持精简，希望它可以在太空船发生故障

> 前完成所有主要工作。 而对于未来那些需要较长持续时间的小型卫星，它们的系统工程不应该基于任何不切实际的期望和侥幸。

在本论文中，大部分小型卫星任务的开发人员都只能遵照第三个选项。 对于那些立方卫星任务，这个方法通常都会成功，但有时也会发生早期的错误。 然而，去赌何时卫星会出错或者迷信器件不会受环境的影响是不能被接受的，更不用说政府、太空局或是投资者了。 为了保证可长期执行任务的立方卫星任务成功，必须要有更好的、更可靠的系统架构。 因此，适用于基于现代商业半导体的机载电脑的容错概念就显得尤为重要。

本论文书及其结果

为解决现代科技对于超小卫星的技术缺陷，本论文提出了一个全新的容错计算机架构。 这个架构甚至可以整合进使用现代半导体产业技术制造的立方卫星。

为了开发本论文所提出的架构，我们使用了来自各种科学和工程领域的方法和概念，而且我们所涉及到的专业技术超越了现有的科学和工程技术。 我们利用实际可行的操作方式与电机工程领域最严谨的测试方式将两个不同领域最先进的技术、概念与理论结合在一起。

为了使科学家和工程师更容易理解本文的工作，第二章和第三章简单介绍论文所涉及到的容错定义与技术。 第二章会为不熟悉该领域的读者介绍与当代航太技术相关的要点以及容错计算机设计的概念，阐述本论文的研究动机。 为在航天装置上设计有效的容错架构，我们必须要清楚宇宙环境对于电脑装置的影响，所以第三章会介绍这些宇宙射线的影响、太空电路设计的限制以及太空任务中时常考量的的因素等(如：天体力学以及通讯时间)。

第四章将介绍本论文所提出的容错机载计算机架构。 我们的架构结合了FPGA可重新组态的容错概念以及混合关键系统的技术，这进一步完善了传统错误容错检测与错误恢复的方法。 我们提出的方法被设计成多个不同但相关的过程，这使得我们的机载计算机能更稳定的老化。

在第四章中，为了实现以上功能，我们利用软件模拟出的一个简化版的锁步技术。 仅使用这个模式就可以有非常强大的容错能力，但它不能满足宇宙任务中长期运行的需求。 因此，第五章介绍可重新组态的逻辑如何协助修复各式各样的错误。我们利用FPGA的可重新组态特性来确保系统单晶片的完整性。 这将帮助我们延长整台计算机的寿命，并且尽可能地妥善利用备用资源。 然而，对于长期的太空任务，那些损坏的FPGA部件终究无法再次利用重新组态来修复，可以被可重新编写的程序逻辑将会随着时间越来越少。 因此在第六章中，我们将展示如何混合关键技术使得一台计算机慢慢被降级而不是像传统的计算机立即无法使用。 我们可利用这样的技术使得计算机自动将效能的损失转换为电力上的节约以及整体完善性。 这使得即便有错误的发生，我们在航天上的核心功能还是可以安全地被维持住，达到整体寿命的延长以及备用资源利用的最大化。

我们将上述全部的功能安装在一个FPGA多核的单片系统上。 软件、负载通信以及逻辑程序对于FPGA来说都是重要的数据。 在整个太空任务的过程中，这些信息都必须要保持完整性。 为此在第七章中，我们将介绍在现代的卫星中，如何保护各种不同存储介质中的资料。

在现有基于软件方面的容错技术，理论上应用于现代的半导体制程技术也都很合适。 然而事实上，这些技术对于现实中的应用需求是不切实际的。 目前为止还没有人成功把那些技术实现并验证。 为此，在本研究中，第八章至第十章阐述我们的实现方法。

第八章中，我们利用错误注入（Fault Injection）的方式来验证我们所提出的锁步系统模式。 在第九章，针对前面提出的架构，我们提出一个实际可行的多核的单晶片系统设计。 而在第十章中，我们说明前面章节所提到的观念以及设计和实现方法。 更进一步，我们用开发板以及概念验证的方式展现出采用这种架构的机载计算机在现实生活当中可能的样子。 我们利用以下六种Xilinx的FPGA来展示我们设计:

- Kintex UltraScale KU60,
- Kintex UltraScale+ KU11p, KU3p, KU5p 开发板 Xilinx KCU116, 和
- Virtex UltraScale+ VU9P 开发板 Xilinx VCU118.

对于 KU60、 KU11p 与 KU3p 这三种FPGAs，我们提供完整详细的功耗与利用率数据。

结论

在本论文的开始，我们提出过这样的疑问:

”我们能否在不打破小型卫星应用所需要的质量、大小、预算与复杂度的前提下，利用现代的嵌入式技术与移动装置技术完成具容错功能的架构？”

依照近三年的研究文献以及一些重大事故的纪录，我们或许可以使用下面的说法来回答这个问题:

“是的，利用现代一般用户级或是工业级技术确实可以达到这样的容错计算机结构。 一旦能完成雏形，我们就可以大幅度延长现代立方卫星的寿命，从而使其可被用来完成重大或长期的太空任务。”

本论文提出的架构能用非侵入的方式运行完成。 我们的架构支持目前已经存在的应用，并不需要针对那些服务重新设计来符合这个架构。

我们提出的机制可以快速且准确的检测出实际应用软件的故障问题，并且在大多处状况下，仅需要很低的计算量就能将错误更正。 我们展示了这种架构的高成本效益，且即使长期运行在高太阳直射的太空区域，也能维持正常的工作。

利用现代的元件，我们提出的架构甚至可以运行在耗电仅有1.94瓦的Ultrascale+ FPGA上。 因此，这样的机载电脑架构可以被应用在许多小型卫星上 (如: 2U 立方卫星)。

随着科技的发展，下一代FPGAs的半导体制造程技术会使得我们的方法更具有应用前景。 利用新的制成技术，我们可以更有效地制造那些被用来协助科学与太阳系探索的太空飞船。 在未来，我们将有机会探索更广阔、未知的宇宙。

中文摘要（繁體）

現代的半導體科技讓小型衛星的建造不再是夢想，其發射之成本低廉，能夠在預算有限的情況之下作為多樣化的科學以及商業的用途。 也就是說，這些小且輕的衛星可以達到過去科技無法完成或是過於昂貴的太空任務。 尤其是一些衛星像立方衛星（CubeSats），它們的造價便宜，而且還可以被快速地生產，這使得資源有限的學術環境也能夠跨足衛星的研究。 然而，現在它們的可靠性還是太低。 因此，截至目前為止，它們主要還是被用在較無安全疑慮且低預算的任務。

如今，許多複雜的科學以及商業應用也適合在這種小型衛星上面運作，這使得太空任務的持續性越來越受到重視。 理論上來說，這種衛星也可以用來完成許多極關鍵與複雜的多面向任務，像是太陽系的探勘以及一些天文學的應用。 然而，正如前文所言，它們的可靠性尚且不足，所以現在只能執行一些次要任務。

現代電子工業在這種小型衛星中扮演極其重要的角色，造就了小型衛星內的一些關鍵子系統。 由於整體重量必須要輕巧，所以相關的電子零件必須更輕、更小，還要能夠比傳統的太空等級零件具有更好的效能功耗比。 是故，所有先進的立方衛星都使用了最尖端的工業級嵌入式及行動通訊市場導向的電腦設計技術。 在造價十分低廉的情況下，這些零件不僅提供足夠的效能，耗費更少的能源，同時比其已經有長期使用歷史之太空等級的對應元件更容易使用。

然而，傳統基於單晶片系統（SoCs）的電腦也缺少較大型的太空船所需要的容錯能力。 在相關的研究報告成果中，使用這種單晶片的子系統被認為是太空船發射並部署在太空中發生故障的主因。 由於小型衛星具有預算、能量、重量及空間的限制，當今被設計用來符合較大型太空船的電腦容錯技術，仍無法被採納。

時至西元2019年，還是沒有任何一個可容錯的計算機架構，在不破壞低價、簡單、輕盈、節能的原則之下，成功地將嵌入式及行動市場中的半導體應用在可以被大量生產且易於發射的小型衛星上。 因此，這些小型衛星的設計者有以下三個選項：

元件提升： 採用傳統的太空零件。 這通常會需要把整個太空船的設計變大，而且這種零件的功耗較高、功能較少，還缺少設計彈性，甚至連計算效能也較為低落。 在實務上來說，這會大幅地增加成本、人力以及衛星的開發時間。 因此，這種方法對於那些要求開發迅速、太空船要小、便宜或可支付的起的先進設計理念來說，是沒有建設性的。

備用衛星： 部署一或多個備用衛星（SpareSats）以緩解立方衛星在早期發生錯誤的風險。 實際上，這不僅增加成本，還會讓錯誤更可能發生，因為發射上去的零件數量倍增了。 因此，這個方法只能在系統達到一定的穩固性之後才能使用。 現在這個方法只能用在衛星世代頻繁更替的星座任務，例如Planet Lab，以及一顆具有超多預算的衛星，例如MarCo。

接受： 接受其不可靠的事實。 讓任務保持精簡，期望它可以在太空船意外故障前完成所有主要工作。 而對於未來那些需要較長持續性的小型衛星任務，它們的系統工程不應該基於任何不切實際的期望、信仰以及僥倖。

當這篇論文在撰寫的時候，大部分小型衛星任務的開發人員都只能遵照第三個選項。 對於那些簡單扼要的立方衛星任務，這個方法通常都會成功，但有時候也會發生早期的錯誤。 然而，去賭衛星不會在錯誤的時間被環境影響理應是不能被接受的，更不用說是政府、太空局或是投資者了。 為了保證進階且長期的立方衛星任務可以成功，更好的、更可靠的系統架構是必須存在的。 因此，適合現代商業半導體所組成之機載電腦的容錯概念，是一定要有的。

本書及其結果

為解決現代科技對於超小衛星的技術缺陷，本論文提出了一個全新的容錯計算機架構。 這個架構甚至可以被整合進使用現代半導體產業技術製造的立方衛星。

為了開發出本論文所提出的架構，我們使用了來自各種科學和工程領域的方法和概念，而且 我們所涉及到的專業技術分別超越了現在的科學和工程技術。 我們利用實務上可行的實作方式與電機工程領域最嚴謹的測試方式將兩個不同領域最先進的技術、概念與理論結合在一起。

此外，我們希望能讓這篇研究同時也可以輕易地被科學與工程人員理解，章節二與三的重點會簡單介紹這篇文章所涉及到的容錯定義與技術。 為了讓那些不熟悉這個領域的讀者可以更完整地理解本文的核心理念與動機，章節二大致會介紹與當代航太技術相關的要點以及容錯計算機設計的概念。 而如果要能在航太裝置上設計有效的容錯架構，我們必須要很清楚知道宇宙環境對於電腦裝置的影響。 所以章節三會介紹這些宇宙射線的影響、太空電路設計的限制以及太空任務中時常被考量的的因素等(如：天體力學以及通訊時間)。

第四章將介紹我們所提出的容錯機載計算機架構。 我們的架構結合了FPGA可重新組態的容錯概念以及混合關鍵系統的技術，這進一步的協助了其他傳統的錯誤容錯與錯誤更正的方法。 我們提出的方法被設計成了許多不同且環環相扣的過程，這使得我們的機載計算機能更穩定的老化。

在第四章中也會提到為了實現以上功能，我們利用軟體模擬出的一個簡化版鎖步系統模式。 僅使用這個模式就可以有非常強大的錯誤容忍能力，但它不能滿足宇宙任務中需要長期運行的需求。 因此，我們在第五章中介紹可重新組態的邏輯如何協助我們將各式各樣的錯誤修復。 我們利用FPGA的可重新組態特性來確保系統單晶片的完整性。 這將幫助我們延長整台計算機的壽命，並且盡可能地妥善利用備用資源。 然而，在十分長久的太空任務中，那些損壞的FPGA區塊終究無法再次利用重新組態來修復。 因此，可以被重新編寫的程式邏輯將會隨著時間越來越少。 在第六章中，我們將展示如何利用混合關鍵技術來使得一台計算機慢慢被降級而不是像傳統的計算機一瞬間就整台無法使用。 我們可利用這樣的技術使得計算機自動地將效能的損失轉換為電力上的節約以及整體的完善性。 這使得即便有錯誤的發生，我們在航太上的核心功能還是可以安全地被維持住，達到整體壽命的延長以及備用資源利用的最大化。

我們將上述全部的功能，使用軟體實作在一個FPGA多執行序的單晶片系統上。軟體、負載資訊以及邏輯程式對於FPGA來說都是重要的資料，在一整個太空任務的過程之中，這些東西都必須要保持其完整性。 在第七章中，我們介紹在現代的衛星之中，如何保護各種不同記憶體中資料。

在以往基於軟體方面的容錯技術，理論上應用於現代的半導體製程技術也都很適合。 然而這些技術事實上對於真實世界中的應用需求都是不切實際的。 到現在為止還沒有人成功地把那些技術實作出來並驗證，因此這將會是很困難的一步。 於本篇研究中，我們接受這個挑戰，並在第八章節至第十章節說明我們是如何達成的。

第八章中，我們利用錯誤注入的方式來驗證我們所提出的鎖步系統模式。 於第九章中，我們針對前面提出的架構提出一個實務上可行的多處理器的單晶片系統設計。

而於第十章節中，我們專注在說明前面章節所提到的觀念以及設計是如何實作的。更進一步，我們用開發板以及概念驗證的方式展現出採用這種架構的機載計算機在現實生活當中可能會長成甚麼樣子。 我們利用以下六種Xilinx的FPGA來展示我們設計:

- Kintex UltraScale KU60,
- Kintex UltraScale+ KU11p, KU3p, KU5p 開發板 Xilinx KCU116, 和
- Virtex UltraScale+ VU9P 開發板 Xilinx VCU118.

對於 KU60、 KU11p 與 KU3p 這三種FPGAs，我們提供完整詳細的功耗與利用狀況數據。

結論

於文章開頭，我們提出過這樣的疑問:

「我們能否在不打破小型衛星應用所需要的質量、大小、預算與複雜度的前提下，利用現代的嵌入式技術與行動裝置技術完成具容錯功能的架構？」

依照近三年的研究文獻以及一些重大的災害紀錄，我們或許可以使用下面的說法來回答這個問題：

「是**的**。 利用現代一般用戶級或是工業級技術確實可以達到這樣的容錯計算機結構。 一旦能完成雛形，我們就可以用來大幅度延長現代立方衛星的壽命，從而使其可被用來完成重大或長期的太空任務。」

本文提出的架構能用非侵入的方式實作完成。 我們的架構都有支援那些已經存在的應用服務，完全不需要針對那些服務重新設計來符合這個架構。

我們提出的機制可以快速且準確的檢測出實際應用軟體的故障問題，並且在大多處的狀況下，僅需要很低的計算量就能將錯誤更正。 我們也展示出這種架構的成本效益很高，且即使長期運行在高太陽直射的太空區域，也能維持正常的工作。

利用現代的元件，我們提出的架構甚至可以實作在耗電僅有1.94瓦的Ultrascale+ FPGA之上。 因此，這樣的機載電腦架構肯定可以被應用在許多小型衛星上 (如: 2U立方衛星)。

隨著科技的發展，下一代FPGAs的半導體製程技術會使得我們的方法更加受到重視。 利用新的製成技術，我們可以更有效地製造那些被用來協助科學與太陽系探索的太空梭。 在未來，我們將有機會可以探索更廣大且未知的宇宙。

日本語の要約

現代の半導体技術により、衛星の小型化が可能になっている。 安価な打ち上げを特徴とする小型衛星は、様々な科学・商業機器を搭載できる低コストなプラットフォームである。 特に、最も小さくて軽い衛星は、今までは技術的に実行不可能、非実用的、または単に不経済であった宇宙ミッションを可能にしている。 特に、CubeSatとして作られた衛星は、限られたリソースしかない学術環境でも、低コストで迅速に製造できる。 しかし今、そのような宇宙船は低い信頼性という問題に直面している。 そのため、これまでは主に、リスクを許容できるような、重要性の低い低予算ミッションに利用されてきた。

今日、多くの洗練された科学・商用アプリケーションを目的として、小型衛星を利用する事も可能である。この様な場合、ミッション期間をできるだけ長くすることが望まれる。 理論的には、このような宇宙船は今日、様々な重要かつ複雑な多段階ミッションや、太陽系内探査や天文学の観測への応用といった高優先度の科学ミッションにも利用できる。 しかし、これらの宇宙船は信頼性が低いため、これまで副次的なタスクを達成するための助けとしてのみ利用されてきた。

現代の電子機器はそのような宇宙船の重要な部分や、最も重要なサブシステムのいくつかを構成している。 これらの電子機器は、宇宙船自体が軽量であることを考慮すると、従来の宇宙用コンポーネントよりも軽く、小さく、ワットあたりの性能が優れている必要がある。 従って、今日の全ての高度なCubeSatは、産業用組込み機器やモバイル機器にも使われる最先端のコンピューター設計を利用している。 これによって、最小限のコストで豊富なパフォーマンスを提供できる他、消費エネルギーが少なく、長年使用されてきた宇宙級同等品よりも操作が簡単である。

しかし、従来のSoCを使用したコンピューターには、大型宇宙船に搭載されているコンピューターアーキテクチャのフォールトトレランス機能がない。 従来研究では、これらの部品を使用したサブシステムは、宇宙船が打ち上げられて配備された後の大部分の障害の原因であると判断されている。 小型衛星の予算、エネルギー、重量、及び体積の制限により、大型宇宙船用に開発された既存のフォールトトレラントコンピュータソリューションは採用できない。

2019年現在、産業用組込み機器やモバイル機器にも使われる半導体を搭載したナノサテライトで利用できるフォールトトレランス機能を備えたコンピューターアーキテクチャは存在していない。 従って、小型衛星開発者には、次のような選択肢が残されている。

アップスケーリング： 従来の宇宙級部品を利用する。 これには通常、宇宙船の設計をより大きなフォームファクターにアップスケールする必要がある。 そのような部品はより多くのエネルギーを必要とし、機能性、柔軟性、処理性能が低いためである。 実際には、これにより、コスト、人件費、および衛星開発時間が大幅に増える。 従って、このアプローチは、短開発期間化、小型化、拡張可能化、低維持費を特徴とする宇宙船の利用を中心としたほとんどの新しいミッション

構想に対して建設的ではない。

予備衛星利用： 1つまたは複数の予備衛星を投入して、障害が発生したCubeSatを代替することにより、早期障害のリスクを軽減する。 実際には、これによりコストが増加するだけでなく、使用した部品の総数が倍増するため、障害発生の可能性が高くなる。 従って、このアプローチは、十分なレベルの堅牢性が達成された後にのみ実行可能になる。 現在、このアプローチは、衛星世代が急速なペースで継続的に交換される星座ミッション（例えば、Planet Lab）、及び非常に豊富な予算を持つ個別の衛星プログラム（例えば、MarCo）でのみ実行可能である。

受け入れ： 信頼性の欠如を受け入れる。 宇宙船が最終的に偶然に失敗する前に、すべての主要な目的を達成することを期待して、ミッションの簡潔化を図る。 しかし、将来の長運用期間の小型衛星ミッションの場合、希望、信仰、幸運をシステムエンジニアリングの基盤とすることは避けるべきである。

この論文が書かれたとき、ほとんどの小型衛星ミッションの開発者はこの3番目の選択肢に従うことを余儀なくされた。 非常にシンプルで運用期間の短いCubeSatミッションの場合、このアプローチは多くの場合成功したが、多くの初期の失敗ももたらした。 しかし、時間に賭けて、悪いタイミングで環境効果の影響を受けない様との希望に固執することは、政府、宇宙機関、及び投資家から益々容認されなくなってきている。 それは、より優れた、より信頼性の高いシステムアーキテクチャが必要とされる高度な長期CubeSatミッションの成功を確実にするためである。 従って、現代の商用半導体に基づいたオン・ボード・コンピューターに適したフォールトトレラントの概念が必要です。

本論文とその成果

本論文では、今日の小型衛星の利用に影響を与える技術的欠陥を克服するために、新しいフォールトトレランス機能を備えたコンピュータアーキテクチャについて詳述する。 これは最新の市販半導体を用いた科学用途の軽量CubeSatにも適用できる。

本論文で詳述されるアーキテクチャを開発するには、幅広い科学および工学分野の成果と概念が利用された。 また、このアーキテクチャの開発に関わる専門知識は、科学と工学の両方を個別に超えている。 代わりに、これらの両方の長所を組み合わせて、科学の進歩、概念的知識、理論的概念を、宇宙および電気工学の分野で実用的な実装と徹底的なテストを通じて統合している。

本論文の研究内容を科学者と技術者の両方にとって分かりやすいものにするために、第２章と第３章では、この論文で対象とされる故障モデルの紹介と定義について述べる。 第２章は、このトピックに精通していない読者のために、今日の宇宙飛行の重要な側面について概述している内容が含まれており、本論文の動機づけとなっている。 第２章では、フォールトトレラントコンピューターの設計に関連する概念についても紹介する。 効果的かつ効率的なフォールトトレラントオンボードコンピューターアーキテクチャの設計および開発を行うために、コンピューターの宇宙環境の影響に関して把握することが重要である。 従って、第3章では、これらの効果、宇宙電子機器の設計上の制約、通信時間や天体力学などの宇宙ミッション中の運用上の考慮事項について詳しく説明する。

第４章では、ソフトウェアで実装されたフォールトトレランスの概念とFPGAの再構成および混合重要度を組み合わせたフォールトトレラントオンボードコンピューターアーキテクチャについて述べる。 これは、他のいくつかの従来のフォールトトレランスおよびエラー修正手法でさらに補完される。 このアーキテクチャのフォールトトレランスは、オンボードコンピューターの無害劣化を可能にするいくつかの相互リンクされたステージとして実装される。

これらの全ての機能を有効にするために、ソフトウェアで実装された粗粒度lockstepロックを利用する。 これについては、第４章で詳述する。 この機能だけでも強力なフォールトトレランス機能を提供できるが、長期的なミッションには不十分である。 従って、第５章では、様々な障害から欠陥のあるシステムを回復するために再構成可能なロジックを使用する方法について述べる。 FPGAの再構成を利用して、システムオンチップ設計の整合性を確保し、オンボードコンピューターの耐用年数を延ばし、スペアリソースのフォールトカバレッジの可能性を最大化する。 非常に長期の宇宙ミッションでは、FPGAの欠陥部分は最終的には再構成によって回復できなくなる。 従って、オンボードコンピューター内で利用可能な正常プログラマブルロジックの量は、時間の経過とともに減少する。 第５章では、従来のシステムのように自然に失敗するのではなく、混合重要度によりコンピュータが劣化に適応させる方法を示す。 この機能を使用して、実行時に性能を節電と堅牢性と自律的にトレードオフすることができる。 これにより、障害が発生したときにフライトソフトウェアのコア機能を保護し、無害劣化を実現し、予備リソースをプールして、存続可能性を最大化できる。

この機能はすべてソフトウェアとして存在する。 FPGA内に実装されているマルチプロセッサシステムオンチップで実行される。 ソフトウェア、ペイロード情報、及びFPGAにプログラムされたロジックはデータであり、宇宙ミッション全体を通してその整合性を保護する必要がある。 第７章では、最新の衛星に搭載されている様々なメモリテクノロジー保護の概念について説明する。 現代の半導体に適用可能な以前のソフトウェアベースのフォールトトレラントの概念は、多くの場合理論的には良さそうである。 しかし、これらは実際のアプリケーションでは実用的ではない。 これまで、このようなフォールトトレランスアーキテクチャは実際に実装および検証されていないが、そうすることは重要である。 本論文の第８章から第１０章の内容はこのような実装と検証に関するものである。

アーキテクチャで使用されるlockstep機能は、第８章の故障挿入を用いて検証する。 第９章では、FPGAに実装するための実用的なマルチプロセッサシステムオンチップ設計について説明する。 この設計は、上記のアーキテクチャの理想的なプラットフォームとして機能する。 第９章は、前章で説明した概念と設計の実用的な実装について述べる。 これにより、開発ボードから構築されたブレッドボードベースの概念実証を使用して、このアーキテクチャを備えたオンボードコンピューターが実際にどのように見えるかを示す。 これは、次の6つのXilinx FPGAに対して行われた。

- Kintex UltraScale KU60、
- Kintex UltraScale + KU11p、KU3p、Xilinx KCU116開発ボードのKU5p、
- Xilinx VCU118開発ボードのVirtex UltraScale + VU9P。

これらのFPGAのうち、KU60、KU11p、およびKU3pの3つについて、詳細な電力および使用率データを提供する。

結論

本研究が始まったとき、私は次の質問を提起した。

「小型衛星アプリケーションの重さ、大きさ、複雑さ、および予算の制約を解消することなく、最新の組込みおよびモバイル市場向けの半導体技術でフォールトトレランスコンピューターアーキテクチャを実現できるか？」

その後の3年間、多くの研究論文が発表され、またいくつかの大惨事が発生したが、次のようにこの質問に答えることができた。

「**はい。** 小型衛星用のフォールトトレラントコンピュータアーキテクチャは、現代の消費者や産業向けのテクノロジで技術的に実現可能である。 プロトタイプとして完全に実装されると、現代のCubeSatの寿命を大幅に延長するために使用できるため、重要かつ長期的な宇宙ミッションでの使用が可能になる。」

本論文で提案されたアーキテクチャのソフトウェアコンポーネントは、非侵襲的な方法で実装できる。 これらは、既存のアプリケーションを保護し、このアーキテクチャをサポートするためにアプリケーションをカスタム作成する必要はない。 実際のソフトウェアを使用して、これらのメカニズムが障害を迅速かつ高い確率で検出でき、ほとんどの場合、低計算コストで障害から正常に回復できることが示された。 このアーキテクチャのパフォーマンスコストは経済的であり、非常に放射線量の高い空間領域で動作する場合でも効果的であることも実証されている。

最新の商用部品を使用すると、このアーキテクチャの理想的なプラットフォームとして機能するシステムオンチップ設計を、わずか1.94Wの消費電力で最小のUltrascale + FPGAに実装することができる。 従って、このオンボードコンピューターアーキテクチャは、2U CubeSatほどの小さい衛星に適用できる。

技術に合わせて拡張できるため、次世代FPGAの半導体製造の進歩により、このアプローチはさらに魅力的になり、小型宇宙船の保護にも使用できるようになるだろう。 現在、私たちが高優先度の科学と太陽系の探査に使用しているより重い宇宙船に実装されると、効率とスケーラビリティを改善できる。 そして、おそらく将来的には、その境界を越えて何が存在するのかを探ることが期待できる。

Resumen en Español

La tecnología de semiconductores modernos permite la construcción de satélites miniaturizados, los cuales son económicos para lanzar y sirven como plataformas de bajo costo para una amplia variedad de instrumentos científicos y comerciales. Los satélites más pequeños y livianos están especialmente situados para realizar misiones espaciales que previamente eran técnicamente imposibles, imprácticas, o simplemente costosas. Particularmente, los satélites construidos como CubeSats pueden ser fabricados rápidamente a bajo costo con los limitados recursos en ámbitos académicos. Sin embargo, en la actualidad estas naves espaciales presentan baja fiabilidad. Por ello se han utilizado principalmente para misiones de bajo presupuesto y menos críticas en donde los riesgos son aceptables.

Muchas aplicaciones sofisticadas, tanto de tipo científicas como comerciales, se prestan para el formato de los satélites miniaturizados, lo cual hace misiones de más larga duración deseables. Teoréticamente, dichas naves espaciales pueden ser utilizadas actualmente en una variedad de misiones críticas y polifacéticas complejas, al igual que para misiones científicas de alta prioridad como para la exploración del sistema solar y aplicaciones astronómicas. Sin embargo, debido a su baja fiabilidad, estas naves espaciales han sido utilizadas hasta ahora para realizar tareas secundarias.

Los electrónicos modernos constituyen una parte significativa de dichas naves espaciales, y componen varias partes de los subsistemas más críticos de la nave. Tomando en cuenta el restringido peso de estas, los electrónicos deben ser más livianos, pequeños, y además deben ofrecer mejor rendimiento por watt que los tradicionales componentes con resistencia a radiación. Por ende, los CubeSats avanzados en la actualidad utilizan arquitecturas de computadoras derivadas de tecnologias móviles e industriales innovadoras. Con un costo mínimo, estos ofrecen alto rendimiento, requieren menos energía, y son más fáciles de trabajar que sus contrapartes con resistencia a radiación, las cuales tienen un largo legado de uso en el espacio.

Sin embargo, las computadoras basadas en sistemas en chip convencionales también carecen de la capacidad para la tolerancia a fallos de las arquitecturas de computadoras a bordo de naves espaciales grandes. El análisis de naves espaciales lanzadas y desplegadas en el espacio determinaron que los sistemas en chip eran los responsables de la mayoria de fallas en las misiones. Debido a restricciones de presupuesto, energía, masa y volumen en satélites miniaturizados, las actuales técnicas de tolerancia a fallos, originalmente desarrolladas para naves espaciales grandes, no pueden ser adoptadas y aplicadas.

Hasta la fecha de esta tesis, no existen arquitecturas de computadoras con tolerancia a fallos que se puedan utilizar a bordo de nanosatélites con semiconductores integrados y móviles sin que se quiebre con el concepto de satélites de bajo costo, simples, energéticamente eficientes y livianos que puedan ser fabricados en masa y lanzados a bajo costo. Por consiguiente, los siguientes métodos existen para desarro-

llar satélites miniaturizados:

Escalamiento: Utilización de componentes tradicionales para uso espacial. Esto requiere incrementar las dimensiones del diseño de la nave espacial, ya que estos componentes requieren más energía y ofrecen menos funcionalidad, flexibilidad, y rendimiento. En práctica, esta opción incrementa drásticamente el costo, mano de obra, y tiempo de desarrollo requerido. Como tal, esta opción no es constructiva para la mayoria de misiones con el objetivo de mantener las naves espaciales pequeñas, con bajo presupuesto, y de rápido desarrollo.

SpareSats: Reducir y mitigar el riesgo de fallos tempranos mediante el desplegamiento de SpareSats para reemplazar un CubeSat que ha fallado. En práctica, este método no solo incrementa el presupuesto necesario, pero también incrementa la posibilidad de fallos ya que el número de componentes lanzados y desplegados se duplica. Debido a las limitaciones mencionadas, este método se convierte en una solución viable solo cuando se ha logrado suficiente robustez. Por ello, SpareSats es principalmente viable para misiones de constelación donde generaciones de satélites son reemplazados continuamente a un paso acelerado (por ejemplo, Planet Lab), y para satélites individuales con un prosupuesto abundante (por ejemplo, MarCo).

Aceptación: Aceptar el riesgo de baja fiabilidad. Este método se basa en que la misión sea de corta duración con la esperanza de alcanzar los objetivos principales antes que la nave espacial eventualmente falle. Para futuras misiones de satélites miniaturizados con una larga duración, esperanza, fe y suerte no deberian ser factores sobre los cuales este basada la ingeniería.

Cuando se escribio esta tesis, la mayoria de satélites miniaturizados tuvieron que seguir la tercera opción, aceptación, durante el desarrollo de la misión. Para misiones de CubeSats sencillas y breves, este método resultó en exito más amenudo de lo esperado, pero también llevo a fallos en etapas tempranas de la misión. No obstante, jugarse contra el tiempo y aferrarse a la esperanza que los efectos ambientales en el espacio no impacten la misión en el momento equivocado es inaceptable, y, cada vez más, menos tolerado por gobiernos, agencias espaciales e inversionistas. Para asegurar que las misiones avanzadas de larga duración con CubeSats sean exitosas, mejores arquitecturas de sistemas con alta fiabilidad son indispensables. Por ello son necesarios conceptos de tolerancia a fallos que sean adecuados para las computadoras a bordo de satélites basadas en semiconductores comerciales modernos.

Esta Tesis y sus Resultados

Para superar los déficits tecnológicos que impactan el uso de satélites muy pequeños en la actualidad, esta tesis detalla una novedosa arquitectura de computadoras con tolerancia a fallos. El método y enfoque presentado en esta tesis es adecuado para integración en satélites de todo tamaño, incluyendo los CubeSats livianos para misiones científicas, los cuales estan basados en semiconductores comerciales modernos.

Para desarrollar la arquitectura presentada en esta tesis, resultados y conceptos de varias areas de ciencias e ingienerias son utilizados. La experiencia necesaria para desarrollar esta arquitectura trasciende la ciencia e ingieneria individualmente. Lo mejor de ambos campos es combinado: avances científicos, conocimiento conceptual y nociones teoréticas son combinadas con la implementación práctica y pruebas minuciosas que son estándar en el ámbito de ingeniería espacial y eléctrica.

Con el objetivo de hacer esta tesis más accesible para ambos cientéficos e ingenieros, el segundo y tercer capítulo introducen el tema a tratar y definen el modelo de tolerancia a fallos tratado en esta tesis. El segundo capítulo sirve como motivación de la tesis, y presenta un resumen breve sobre aspectos claves del vuelo espacial y conceptos relacionados a la arquitectura de computadoras con tolerancia a fallos. Para poder diseñar y desarrollar efectivas y eficientes computadoras a bordo de satélites con tolerancia a fallos, se necesita entender los efectos del ambiente espacial en computadoras. Por ello, el tercer capitulo detalla estos efectos, las restricciones en el diseño de dispositivos electrónicos para el espacio, y las consideraciones necesarias durante misiones espaciales, tales como tiempos de comunicación y mecánica celeste.

Con base en los capítulos anteriores, el cuarto capítulo presenta una arquitectura de computadora que combina conceptos de tolerancia a fallos implementados via software junto con reconfiguración de arreglo de compuertas programables en el campo, o FPGA[2] y criticalidad mixta. A esto se le agrega otras medidas más convencionales de tolerancia a fallos y corrección de errores. Tolerancia a fallos en esta arquitectura es implementada mediante varias etapas entrelazadas que permiten una computadora a bordo de una nave espacial envejecer con elegancia.

Para hacer posible toda esta funcionalidad, se utiliza la ejecución sincronizada periódicamente (coarse-grained lockstep) implementada mediante software, lo cual está descrito en detalle en el cuarto capítulo. Esta funcionalidad por si sola ofrece una fuerte capacidad para tolerancia a fallos, pero sería insuficiente para recuperar misiones de larga duración. Por ello, en el quinto capítulo, se describe como la lógica reconfigurable puede ser utilizada para reuperar un sistema defectuoso causado por una variedad de fallas. Se utiliza un FPGA reconfigurable para asegurar la integridad del diseño del sistema en chip, con el objetivo de extender la vida útil de una computadora a bordo de una nave espacial, y maximizar la cobertura contra fallos de los recursos de repuesto. En misiones espaciales de larga duración, partes defectivas de un FPGA eventualmente no podrán ser recuperables mediante reconfiguración. Por ende, la cantidad disponible de lógica programable intacta dentro de los sistemas a bordo disminuye con el tiempo. En el sexto capítulo, se demuestra como la criticalidad mixta permite a una computadora adaptarse a la degradación, en lugar de fallar espontáneamente como lo hacen sistemas tradicionales. Esta funcionalidad se puede utilizar para intercambiar rendimiento con ahorro de energía y robustez autónoma durante el tiempo de ejecución. Esto permite que la funcionalidad central del software de vuelo sea protegida cuando fallos ocurren, logrando envejecimiento con elegancia y reuniendo recursos de repuesto para maximizar supervivencia.

Toda esta funcionalidad existe como software, y es ejecutada en un sistema en chip con un multiprocesador implementado dentro de un FPGA. El software, información sobre la carga útil, y la lógica programada dentro de un FPGA son datos, la integridad de los cuales debe ser protegida durante la duración de la misión. El sétimo capítulo describe conceptos para la protección de las diferentes tecnologias de memoria

[2]Arreglo de compuertas programable en el campo, o FPGA por sus siglas en inglés.

presentes a bordo de un satélite moderno.

Conceptos previos de tolerancia a fallos basados en software que se pueden aplicar a semiconductores modernos parecen funcionar en teoría. Sin embargo, estos resultan ser imprácticos para aplicaciones reales. Hasta la fecha de esta tesis no se ha implementado y validado tales conceptos en práctica, pero esto es un paso crítico y necesario. Los capítulos del ocho al diez detallan la implementación y validació del método de la arquitectura presentada en esta tesis.

La funcionalidad de lockstep utilizada en la arquitectura de esta tesis es validada mediante el método de injección de fallas en el octavo capítulo. En el noveno capítulo, se describe un diseño de un sistema en chip con multi-procesador implementado en un FPGA que sirve como plataforma ideal para la arquitectura presentada en esta tesis. El décimo capítulo se dedica a la implementación práctica de los conceptos y diseños descritos en los capítulos anteriores. De esta manera, se demuestra como una computadora a bordo de una nave espacial con esta arquitectura puede ser en la realidad, con la prueba de concepto construida a base de placas de desarrollo. Esto fue hecho con seis FPGA de Xilinx:

- Kintex UltraScale KU60,
- Kintex UltraScale+ KU11p, KU3p, el KU5p de la placa de desarrollo Xilinx KCU116, y el
- Virtex UltraScale+ VU9P de la placa de desarrollo Xilinx VCU118.

Para tres de estos FPGA, KU60, KU11p, y KU3p, datos detallados sobre utilización y consumo de energía son proporcionados.

Conclusiones

La pregunta principal de esta tesis es:

¿Se puede lograr una arquitectura de computadora con tolerancia a fallos utilizando tecnologias modernas integradas y móviles, sin quebrar las restricciones de masa, dimensiones, complejidad y presupuesto para aplicaciones con satélites miniaturizados?

Un doctorado, varios artículos publicados, y varias catástrofes después, ahora es posible responder esta pregunta de la siguiente manera:

Sí. *Una arquitectura de computadora con tolerancia a fallos para satélites miniaturizados es técnicamente factible con tecnología contemporánea de nivel industrial y para el consumidor. Cuando el prototipo esté completamente implementado, se podrá utilizar para extender drásticamente la vida de CubeSats modernos, y así permitir su uso en misiones espaciales críticas y de larga duración.*

Los componentes de software para la arquitectura presentada en esta tesis pueden ser implementados de manera no invasiva. Estos proveen protección para las aplicaciones pre-existentes sin la necesidad de escribir software específicamente para esta arquitectura. Utilizando software se demuestra que estos mecanismos pueden detectar fallos rápidamente y con alta probabilidad, y que se puede recuperar exitósamente de fallos con bajos costos computacionales en la mayoria de casos. Se demuestra que el

costo de rendimiento de esta arquitectura es económico, y permanence efectivo aún cuando opera en ambientes espaciales con fuerte irradiación.

Con componentes comerciales contemporáneos, un diseño de sistema en chip que funciona como plataforma ideal para esta arquitectura puede ser implementado aún en el FPGA Ultrascale+ más pequeño, con solo un consumo de 1.94 W de energía. Por ello, esta arquitectura de computadora a bordo de una nave espacial puede ser aplicada a CubeSats con dimensiones mínimas de 2U.

A medida que escala con la tecnología, avances en fabricación de semiconductores en la siguiente generación de FPGA hara el método presentado en esta tesis aún más atractivo, y también podra proteger naves espaciales aún más pequeñas que 2U. La eficacia y escalabilidad pueden ser mejoradas cuando se implementa a bordo de naves espaciales más grandes y pesadas que se utilizan en la actualidad para ciencia y exploración espacial. En un futuro, quizás podamos explorar más alla de los límites del sistema solar.

Резюме на Русском Языке

Современные полупроводниковые технологии позволяют создавать миниатюризированные спутники, запуск которых дёшев, и недорогие платформы для широкого круга научных и коммерческих инструментов. В особенности это касается наименьших и легчайших спутников, позволяющих организовать космические миссии, которые ранее были технически невозможны, непрактичны и просто неэкономичны. Спутники, сконструированные как Кубсат, могут создаваться быстро и дёшево, в условиях ограниченных ресурсов, характерных для академической среды. Однако такие спутники в настоящее время характеризуются низкой надёжностью. Следовательно, вплоть до последнего времени их использовали в основном для некритичных и малобюджетных миссий, где такие риски приемлемы.

Сегодня многие сложные научные и коммерческие применения могут быть реализованы в форм-факторе миниатюризированных спутников, желательно с намного большей длительностью активного существования. Теоретически, в настоящее время такой спутник мог бы быть использован в критических и сложных многофазных миссиях, а также в высокоприоритетных научных проектах по изучению Солнечной системы и астрономических применениях. Однако из-за своей низкой надёжности эти аппараты до сих пор использовались только как сопутствующие системы для выполнения вторичных задач.

Современная электроника составляет значительную часть таких космических аппаратов и определяет несколько их наиболее критических подсистем. Учитывая их меньший вес, электроника должна быть легче, меньше и предоставлять лучшее соотношение производительности на 1 Ватт мощности, по сравнению с традиционными компонентами космического класса. Таким образом, все наиболее сложные спутники Кубсат сегодня используют передовые промышленные разработки, пришедшие с рынков встроенных систем и мобильных устройств. При минимальной стоимости обеспечивается избыток производительности, малое энергопотребление и лёгкость использования, по сравнению с электроникой космического класса, имеющей длительную историю применения.

Однако для обычных вычислителей на базе систем на кристалле также требуется сбое- и отказоустойчивость, как и для бортовых систем больших космических аппаратов. В соответствующих работах подсистемы, использовавшие эти компоненты признаны ответственными за большинство отказов после того, как аппарат был запущен и выведен на заданную орбиту. Из-за требований ограниченных бюджета миссии, массы, энергии и объёма миниатюризированных спутников существующие решения со сбоеустойчивыми вычислителями для больших аппаратов не могут быть приняты.

По состоянию на 2019 год, не существует архитектур сбоеустойчивых вычислителей, которые могли бы быть использованы в наноспутниках, использующих электронную компонентную базу из применений на мобильных рынках и рынках встроенных систем без нарушений фундаментальной концепции дешёвого, простого, энергоэффективного и лёгкого спутника, который может серийно производится и имеет низкую стоимость запуска. Разработчики малых аппаратов, таким образом, имеют только следующие варианты:

Апскейлинг: (повышение качества) использование традиционных компонентов космического класса. Обычно это приводит к созданию спутника большего форм-фактора, т.к. компонентам нужно больше энергии и они обеспечивают меньшую функциональность, гибкость и производительность. На практике такой подход резко повышает стоимость, требования к рабочей силе и время разработки спутника. Таким образом, этот подход является неконструктивным для большинства концепций новых миссий, концентрирующихся на спутниках, которые разрабатываются быстро, имеют малый размер, способность к расширению и низкую стоимость.

SpareSats: (Спаренные спутники) Уменьшение риска раннего отказа с помощью выведения одного или нескольких SpareSat для замены Кубсат, как только тот отказал. На практике это не только увеличивает стоимость, но и также увеличивает вероятность отказа, поскольку общее количество запущенных компонентов удваивается. Таким образом, данный подход становится реализуемым только если достигнут достаточный уровень надёжности. На сегодняшний день подход может быть реализован только для спутниковых созвездий, где поколения спутников постоянно заменяются в быстром темпе (например, Planet Lab), и для индивидуальных спутников с исключительно большим бюджетом (например, MarCo).

Принятие: Принять недостаток надёжности. Оставить миссию скоротечной в надежде достичь всех главных задач до того, как космический аппарат в произвольный момент откажет. Для будущих миниатюризированных космических миссий с большими сроками активного существования такие факторы как надежда, вера и удача не должны использоваться в качестве инженерной базы.

Когда была написана эта диссертация, разработчики большинства миниатюризированных спутников были вынуждены следовать этому третьему варианту. Для очень простых и быстрых Кубсат миссий этот подход приводил к успеху чаще, чем к неудаче, но тем не менее – к большому числу ранних отказов. Однако, игры со временем и попытки зацепиться за надежду «авось в этот раз пронесёт» — неприемлемы и вызывают всё меньше понимания у правительств, космических агентств и инвесторов. Для обеспечения успеха современных долгоиграющих Кубсат миссий требуются лучшие и более надёжные системные архитектуры. Таким образом, нужны те сбое- и отказоустойчивые концепции, которые подходят для бортовых компьютеров на основе современных полупроводниковых приборов.

Настоящая диссертация и её результаты

В данной диссертации представлена в деталях новая архитектура сбоеустойчивого вычислителя, призванная преодолеть технологический дефицит, который сегодня влияет на использование очень маленьких спутников. Эта технология подходит для интеграции даже в лёгкие научные Кубсат, базирующиеся на современной коммерческой электронной компонентной базе.

Для развития архитектуры, представленной в этой диссертации, использованы результаты и концепции из широкого круга научных и инженерных областей, и потребовавшийся опыт лежит за пределами только науки или только инженерии. Вместо этого мы объединяем лучшее из обоих этих миров: мы интегрируем научные достижения, концептуальное знание и теоретические изыскания с практической реализацией и тщательным тестированием, являющимся стандартом для областей космоса и электронного машиностроения.

Чтобы сделать материалы диссертации доступными для учёных и инженеров, Главы 2 и 3 посвящены неформальному введению и определению рассматриваемой модели сбоев. В Главе 2 содержится краткий обзор сегодняшних ключевых аспектов космического полёта для читателей, незнакомых с данной темой. Он служит в качестве мотивации для данной диссертации. Глава также представляет концепции, относящиеся к проектированию сбоеустойчивого компьютера. Для разработки и развития действительно эффективной и действенной архитектуры сбоеустойчивого бортового компьютера необходимо понимать, как космическое пространство влияет на вычислитель. Глава 3 детализирует эти эффекты, ограничения для разработчика космической электроники, операционные вопросы космических миссий, такие как времена коммуникации, и небесная механика.

В Главе 4, основываясь на материале предыдущих глав, мы представляем архитектуру сбоеустойчивого бортового компьютера, которая включает программно-реализованные концепции на ПЛИС с реконфигурацией и смешанной критичностью. Далее это объединяется с несколькими другими, более традиционными способами обеспечения сбоеустойчивости и исправления ошибок. Сбоеустойчивость в этой архитектуре реализована как несколько взаимосвязанных стадий, позволяющих бортовому компьютеру «стареть изящно».

Для обеспечения этой функциональности мы используем программно-реализованное синхронизированное пошаговое выполнение, описанное в Главе 4. Эта функциональность сама по себе предоставляет широкие возможности обеспечения сбоеустойчивости, но может быть недостаточной для долгих миссий, поэтому в Главе 5 мы описываем, как реконфигурируемая логика может использоваться для восстановления дефективной системы из широкого круга возможных сбоев. Мы используем реконфигурацию ПЛИС для гарантии целостности проекта системы на кристалле, чтобы увеличить сроки функционирования бортового компьютера и максимизировать потенциальное покрытие сбоев и совместно используемые ресурсы.

В космических миссиях с очень долгим сроком выполнения дефективные блоки ПЛИС в конечном счёте перестанут восстанавливаться с помощью реконфигурации, т.е. количество доступной неиспорченной программируемой логики в бортовом компьютере со временем уменьшается. В Главе 6 мы показываем, как смешанная критичность может помочь вычислителю адаптироваться к деградации, вместо того, чтобы внезапно отказывать, как это происходит в традицион-

ных системах. Мы можем использовать эту функциональность для того, чтобы выторговать производительность за энергосбережение и надёжность автономно во время работы. Это позволяет сберечь ядро бортовой программной функциональности при возникновении сбоя, достигая «изящного старения» и используя совместные ресурсы для максимизации выживаемости.

Вся эта функциональность присутствует в виде программного обеспечения. Оно исполняется на мультипроцессорной системе на кристалле, реализованной на ПЛИС. Программное обеспечение, информация о полезной нагрузке и логика, программируемая в ПЛИС, — это данные, целостность которых должна быть обеспечена в течение всей космической миссии. В Главе 7 представлены концепции защиты для различных технологий памяти, используемой на борту современных спутников. Ранее предложенные программные концепции обеспечения сбоеустойчивости, применимые к современным полупроводниковым технологиям, часто звучат привлекательно в теории, однако оказываются непрактичными для реализации в реальном мире. На сегодняшний день не существует такой сбоеустойчивой архитектуры, реализованной и верифицированной на практике, хотя это критический шаг. Мы делаем этот критический шаг в Главах с 8 по 10 данной диссертации.

Функциональность синхронизированного пошагового выполнения, используемого в нашей архитектуре, верифицирована с помощью инжекции (внесения) сбоев в Главе 8. В Главе 9 мы описываем проект мультипроцессорной системы на кристалле, реализованный в ПЛИС, которая служит идеальной платформой для данной архитектуры. Глава 10 посвящена практической реализации концепций и проектов, описанных в предыдущих главах. Таким образом, мы показываем, как может выглядеть бортовой компьютер с этой архитектурой в реальном мире, используя для проверки концепции макеты, сконструированные на основе отладочных плат. Это было сделано для следующих 6-ти ПЛИС фирмы Xilinx:

- Kintex UltraScale KU60,
- Kintex UltraScale+ KU11p, KU3p, KU5p из отладочной платы KCU116 и
- Virtex UltraScale+ VU9P из отладочной платы VCU118.

Для трёх из этих ПЛИС: KU60, KU11p и KU3p – мы представили детальные данные по мощности и утилизации.

Заключение

В начале этой диссертации мы поставили вопрос:

«Может ли архитектура сбое- и отказоустойчивого компьютера основываться на технологиях современного рынка встроенных и мобильных применений, без нарушения ограничений массы, размера, сложности и бюджета, характерных для миниатюризированных спутников?»

Спустя три года, множество опубликованных исследовательских статей и несколько катастроф, можно ответить на этот вопрос следующим образом:

*«**Да.** Сбое- и отказоустойчивая архитектура для миниатюризированных спутников технически реализуема с помощью современных технологий потребительского и промышленного уровня. Будучи однажды полностью реализована в качестве прототипа, она может быть использована для значительного увеличения сроков активного существования современных Кубсат, позволяя тем самым использовать их для длительных космических миссий.»*

Программные компоненты архитектуры, представленные в настоящей диссертации, могут быть реализованы «неинвазивным» способом. Они предоставляют защиту существующих применений без необходимости специализированных изменений в них для поддержки этой архитектуры. Используя обычное программное обеспечение, мы показываем, что эти механизмы могут детектировать сбои и отказы быстро и с большой вероятностью и что мы можем успешно восстанавливаться после сбоев, в большинстве случаев – при малых вычислительных потерях. Мы демонстрируем, что вычислительная стоимость этой архитектуры экономична и остаётся эффективной даже при работе в исключительно жёстких радиационных условиях космоса.

С современными коммерческими электронными компонентами проект системы на кристалле, который служит идеальной платформой для этой архитектуры, может быть реализован даже на наименьшей Ultrascale+ ПЛИС с потреблением всего лишь 1,94 Вт. Следовательно, архитектура бортового компьютера может быть применена к спутникам размером с 2U Кубсат.

При технологическом масштабировании, прогресс в полупроводниковой технологии в следующем поколении ПЛИС сделает этот подход даже более желательным и удобным для защиты меньших космических аппаратов. Он может повысить эффективность и масштабируемость при применении на борту более тяжёлого космического аппарата, который мы используем сегодня для высокоприоритетных научных задач и для исследования Солнечной системы. И можно выразить надежду на то, что когда-нибудь в будущем мы сможем исследовать и то, что находится за её пределами.

English Summary

Modern semiconductor technology allows the construction of miniaturized satellites, which are cheap to launch, low-cost platforms for a broad variety of scientific and commercial instruments. Especially the smallest and lightest satellites can enable space missions which previously were technically infeasible, impractical or simply uneconomical. In particular satellites constructed as CubeSats can be manufactured rapidly at low cost, with the limited resources available in academic environments. However, today such spacecraft suffers from low reliability. Hence, they have up until now mainly been used for less critical and low-budget missions, where risks can be taken.

Many sophisticated scientific and commercial applications can today also be fit into a miniaturized satellite form factor, which make a much longer mission duration desirable. Theoretically, such spacecraft could also be used in a variety of critical and complex multi-phased missions, as well as for high-priority science missions for solar system exploration and astronomical applications. However, due to their low reliability, these spacecraft have until now been used only as companions to accomplish secondary tasks.

Modern electronics constitute a significant part of such spacecraft, and make up several of their most critical subsystems. Considering their lower weight, these electronics must be lighter, smaller, and offer a better performance-per-watt ratio than traditional space-grade components. Thus, all advanced CubeSats today utilize cutting-edge industrial embedded and mobile-market derived computer designs. At minimal cost, these offer an abundance of performance, require less energy, and are easier to work with than their space-grade counterparts that have a long legacy of use.

However, conventional systems-on-chip-based computers also lack the fault tolerance capabilities of computer-architectures aboard larger spacecraft. In related work, subsystems using these components were determined responsible for a majority of failures after spacecraft were launched and deployed in space. Due to budget, energy, mass, and volume restrictions in miniaturized satellites, existing fault-tolerant computer solutions developed for such larger spacecraft can not be adopted.

As of 2019, there exists no fault-tolerant computer architectures that could be used aboard nanosatellites powered by embedded and mobile-market semiconductors, without breaking the fundamental concept of a cheap, simple, energy-efficient, and light satellite that can be manufactured en-mass and launched at low cost. Miniaturized satellite developers are, thus, left with the following options:

Upscaling: Resort to utilize traditional space-grade components. This usually requires upscaling of the spacecraft design to a larger form factor, as such components require more energy and offer less functionality, flexibility, and processing performance. In practice, this drastically increases cost, manpower requirements, and satellite development times. Hence, this approach is not constructive for most novel mission concepts centered

around utilizing specifically spacecraft that can be developed rapidly, or which have to be kept small, expendable, or cheap.

SpareSats: Mitigate the risk of early failure by deploying one or multiple SpareSats to replace a CubeSat once it has failed. In practice, this not only increases costs, but also makes failures more likely as the total number of components launched is increased. Hence, this approach only becomes viable after a sufficient level of robustness can be achieved. Today this approach is only viable for constellation missions where satellite generations are replaced continuously at a rapid pace (e.g., Planet Lab), and individual satellites with an exceptionally abundant budget (e.g., MarCo).

Acceptance: Accept the lack of reliability. Keep the mission brief in the hope of achieving all main objectives, before the spacecraft eventually fails by chance. For future miniaturized satellite missions with a longer duration, hope, faith, and luck should not be factors upon which systems engineering is based.

When this thesis was written, developers of most miniaturized satellite missions were forced to follow this third option. For very simple and brief CubeSat missions, this approach resulted in success more often than not, but also in many early failures. However, gambling against time and clinging to hope to not be impacted by environmental effects in the wrong moment is unacceptable, and increasingly less tolerated by governments, space agencies, and investors. To ensure success for advanced long-term CubeSat missions, better, more reliable system architectures are required. Hence, fault-tolerant concepts are needed that are suitable for on-board computers based on modern commercial semiconductors.

This Thesis and its Results

To overcome the technological deficits that impact the use of very small satellites today, in this thesis a new fault-tolerant computer architecture is detailed. It is suitable for integration even into light scientific CubeSats, which are based on modern commercial semiconductors.

To develop the architecture presented in this thesis, results and concepts from a wide range of science and engineering fields are used. The expertise involved in developing this architecture transcends both science and engineering individually. Instead, we combine the best of both of these worlds: we integrate scientific advances, conceptual knowledge, and theoretical notions, with the practical implementation and thorough testing that is standard in the fields of space and electrical engineering.

To make the research contained within this thesis accessible to both scientists and engineers, Chapters 2 and 3 are intended as an informal introduction and definition of the fault-model considered in this thesis. Chapter 2 contains a brief overview over key aspects of spaceflight today, for readers who are unfamiliar with this topic. It serves as motivation for this thesis. The chapter also introduces concepts related to fault-tolerant computer design. In order to design and develop a fault-tolerant on-board computer architecture that is actually effective and efficient, it is crucial to understand the effects of the space environment on a computer. Chapter 3 thus details

these effects, design constraints for space electronics, and operational considerations during space missions, such as communication times, and celestial mechanics.

Based on the preceding chapters, in Chapter 4 we present a fault-tolerant on-board computer architecture which combines software implemented fault tolerance concepts with FPGA reconfiguration and mixed criticality. This is further complemented with several other, more conventional fault tolerance and error correction measures. Fault tolerance in this architecture is implemented as several interlinked stages that allow an on-board computer to age gracefully.

To enable all this functionality, we utilize a software-implemented coarse grain lockstep, which is described in detail in Chapter 4. This functionality alone offers strong fault tolerance capabilities, but would be insufficient for long term missions. Therefore, in Chapter 5, we describe how reconfigurable logic can be used to recover a defective system from a broad variety of faults. We utilize FPGA reconfiguration to assure the integrity of a system-on-chip design, in order to extend the useful lifespan of an on-board computer, and to maximize the fault coverage potential of spare resources. In space missions with a very long duration, defective parts of an FPGA will eventually no longer be recoverable through reconfiguration. Hence, the amount of intact programmable logic available within an on-board computer diminishes overtime. In Chapter 6, we show how mixed criticality can enable a computer to adapt to degradation, instead of failing spontaneously as traditional systems do. We can use this functionality to trade performance for power-saving and robustness autonomously at runtime. This allows the flight software core functionality to be safeguarded as faults occur, achieving graceful aging and pooling spare resources to maximize survivability.

All of this functionality exists as software. It is run on a multi-processor system-on-chip that is implemented within an FPGA. Software, payload information, and the logic programmed into an FPGA are data, the integrity of which must be safeguarded during the entirety of a space mission. In Chapter 7, protective concepts for the different memory technologies present aboard a modern satellite are described.

Previous software-based fault-tolerant concepts applicable to modern semiconductors often sound nice in theory. However, these turn out to be impractical for real-world application. To date no such fault tolerance architecture has been practically implemented and validated, but doing so is a critical step. We take this critical step in Chapters 8 through 10 of this thesis.

The lockstep functionality used in our architecture is validated using Fault Injection in Chapter 8. In Chapter 9, we describe a practical multi-processor system-on-chip design for implementation on an FPGA that serves as an ideal platform for said architecture. We then dedicate Chapter 10 to the practical implementation of the concepts and designs described in the previous chapters. Thereby, we show how an on-board computer with this architecture can look like in the real-world, using a breadboard-based proof-of-concept constructed from development boards. This was done for the following 6 Xilinx FPGAs:

- Kintex UltraScale KU60,
- Kintex UltraScale+ KU11p, KU3p, the KU5p of a Xilinx KCU116 development board, and the
- Virtex UltraScale+ VU9P of a Xilinx VCU118 development board.

For three of these FPGAs, KU60, KU11p, and KU3p, we provide detailed power and utilization data.

Conclusions

At the start of this thesis, we raised the question:

Can a fault tolerance computer architecture be achieved with modern embedded and mobile-market technology, without breaking the mass, size, complexity, and budget constraints of miniaturized satellite applications?

A PhD, many published research papers, and several catastrophes later, it is now possible to answer this question in the following way:

Yes. *A fault-tolerant computer architecture for miniaturized satellites is technically feasible with contemporary consumer- and industrial-grade technology. Once fully implemented as a prototype, it can be used to expand the lifetime of modern day CubeSats drastically, thereby enabling their use in critical and long-term space missions.*

The software-components of the architecture presented in this thesis can be implemented in a non-invasive manner. They provide protection for preexisting applications, without the need to custom-write them to support this architecture. Using real-world software, we show that these mechanisms can detect faults rapidly and with a high probability, and that we can successfully recover from faults at low computational cost in most cases. We demonstrate that the performance cost of this architecture is economical, and remains effective even when operating in exceptionally heavily irradiated regions of space.

With contemporary commercial components, a system-on-chip design that serves as ideal platform for this architecture can be implemented even on the smallest Ultrascale+ FPGA with just 1.94W power consumption. Hence, this on-board computer architecture can be applied to satellites as small as 2U CubeSats.

As the architecture scales with technology, advances in semiconductor manufacturing in the next generation of FPGAs will make this approach even more appealing, and also usable to protect smaller spacecraft. It can improve efficiency and scalability when implemented aboard heavier spacecraft that we use today for high-priority science and solar system exploration. And maybe in the future, hopefully, we can explore even what lies beyond its boundaries.

List of Selected Publications

[Fuchs1] C. M. Fuchs, P. Chou, X. Wen, N. M. Murillo, G. Furano, S. Holst, A. Tavoularis, S.-K. Lu, A. Plaat, and K. Marinis. **A Fault-Tolerant MPSoC For CubeSats**. In *IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems*. IEEE, 2019.

[Fuchs2] C. M. Fuchs, N. M. Murillo, P. Chou, J.-J. Liou, Y.-M. Cheng, X. Wen, S. Holst, A. Tavoularis, G. Furano, G. Magistrati, K. Marinis, S.-K. Lu, and A. Plaat. **Fault Tolerant Nanosatellite Computing on a Budget**. In *AIAA/USU Conference on Small Satellites*. AIAA, 2019.

[Fuchs3] C. M. Fuchs, N. M. Murillo, A. Plaat, E. van der Kouwe, D. Harsono, and P. Wang. **Software-Defined Dependable Computing for Spacecraft**. In *IEEE Pacific Rim International Symposium on Dependable Computing*. IEEE, 2018.

[Fuchs4] R. Perea-Tamayo, Fuchs, C. M., E. Ergetu, and B.-X. Li. **Design and Evaluation of a Low-Cost CubeSat Communication Relay Constellation**. In *IEEE Microwave Theory and Techniques Society Latin America Microwave Conference*. IEEE, 2018.

[Fuchs5] C. M. Fuchs, Nadia M Murillo, A. Plaat, E. van der Kouwe, and P. Wang. **Towards Affordable Fault-Tolerant Nanosatellite Computing with Commodity Hardware**. In *IEEE Asian Test Symposium*. IEEE, 2018.

[Fuchs6] C. M. Fuchs, N. M. Murillo, A. Plaat, E. van der Kouwe, D. Harsono, and T. P. Stefanov. **Fault-Tolerant Nanosatellite Computing on a Budget**. In *Conference on Radiation and its Effects on Components and Systems*. IEEE, 2018.

[Fuchs7] C. M. Fuchs, N. M. Murillo, A. Plaat, E. van der Kouwe, and T. P. Stefanov. **Dynamic Fault Tolerance Through Resource Pooling**. In *NASA/ESA Conference on Adaptive Hardware and Systems*. IEEE, 2018.

[Fuchs8] C. M. Fuchs, T. P. Stefanov, N. M. Murillo, and A. Plaat. **Boosting Fault-Tolerance in High-Performance COTS-based Miniaturized Satellite Computers**. In *COSPAR Symposium*. ISC, 2017.

[Fuchs9] C. M. Fuchs, T. P. Stefanov, N. M. Murillo, and A. Plaat. **Bringing Fault-Tolerant Gigahertz-Computing to Space**. In *IEEE Asian Test Symposium*. IEEE, 2017.

[Fuchs10] C. M. Fuchs, N. Dafinger, M. Langer, and C. Trinitis. **Enhancing Nanosatellite Dependability Through Autonomous Chip-Level Debug Capabilities**. In *ESA/CNES 4S: Small Satellites, System & Services Symposium*. ESA Press, 2016.

[Fuchs11] C. M. Fuchs, N. Dafinger, M. Langer, and C. Trinitis. **Enhancing Nanosatellite Dependability Through Autonomous Chip-Level Debug Capabilities**. In *International Conference on Architecture of Computing Systems*. Springer, 2016.

[Fuchs12] C. M. Fuchs. **Dependable Computer Architectures and Software Concepts for Next-Generation Nanosatellites**. Master's thesis, Technical University Munich, 2015.

[Fuchs13] M. Langer, N. Appel, M. Dziura, Fuchs, C. M., P. Günzel, J. Gutsmiedl, M. Losekamm, D. Meßmann, T. Pöschl, and C. Trinitis. **MOVE-II - der zweite Nanosatellit der Technischen Universität München**. In *German Aerospace Congress*. Deutsche Gesellschaft für Luft-und Raumfahrt-Lilienthal-Oberth eV, 2015.

[Fuchs14] N. M. Murillo, S. Bruderer, E. F. van Dishoeck, C. Walsh, D. Harsono, S.-P. Lai, and Fuchs, C. M. **A low-mass protostar's disk-envelope interface: disk-shadowing evidence from ALMA DCO+ observations of VLA1623**. *Astronomy & Astrophysics*, 579, 2015.

[Fuchs15] C. M. Fuchs. **Enabling Dependable Data Storage for Miniaturized Satellites**. In *AIAA/USU Conference on Small Satellites*. AIAA, 2015.

[Fuchs16] C. M. Fuchs, C. Trinitis, N. Appel, and M. Langer. **A fault-tolerant radiation-robust mass storage concept for highly scaled flash memory**. In *Data Systems In Aerospace*. Eurospace, 2015.

[Fuchs17] M. Langer, C. Olthoff, J. Harder, Fuchs, C. M., M. Dziura, A. Hoehn, and U. Walter. **Results and lessons learned from the CubeSat mission First-MOVE**. In *Symposium on Small Satellites for Earth Observation*. IAA, 2015.

[Fuchs18] C. M. Fuchs, M. Langer, and C. Trinitis. **FTRFS: A fault-tolerant radiation-robust filesystem for space use**. In *International Conference on Architecture of Computing Systems*. Springer, 2015.

[Fuchs19] C. M. Fuchs. **The evolution of avionics networks from ARINC429 to AFDX**. In *Innovative Internet Technologies, Mobile Communications, and Aerospace Networks*, volume 65, 2012.

[Fuchs20] M. Brunner, Fuchs, C. M., and S. Todt. **Integrated Honeypot Based Malware Collection and Analysis**. In P. Schoo, M. Zeilinger, and E. Herrmann, editors, *Advances in IT Early Warning*. Fraunhofer IRB Verlag, Germany, 2013.

Curriculum Vitae

I was born on May 22nd 1984 as first child of a computer engineer and a teacher in Linz, Austria. I remember my early childhood full of adventures out in nature, exploring forests, climbing mountains, mobile-home trips all across Western Europe, and mysterious castles and fortresses. As Austria was a nonaligned country during the Cold War, we also undertook frequent trips across the border to the Eastern Block during the 1980s. During these good times, I for the first time witnessed physics simulations running on my dad's computer, which fascinated me and made me curious about how computers can do such a thing.

In primary school, I spent countless hours reading science books, drawing spacecraft and rockets, and later constructing ship-, aircraft- and spacecraft models from plastic and wood. And I passionately watched science fiction TV-series and movies, especially Star Trek, appreciating the values of curiosity, exploration, collaboration, and peaceful cooperation it conveyed.

In middle school, I discovered that the repetitive, memorization-based teaching style used in Austrian schools was not conductive for me. However, I did very well in science, history, and geography, enjoying lengthy discussions beyond what was taught in class with my teachers. With these teachers, I enjoyed doing advanced physics- and chemistry lab experiments, and learned a lot about the time periods on which my history teachers were specialized.

I finally came into the possession of a my first hand-me-down computer in 1994, which enabled me to ... do difficult calculations! I certainly conducted those, but mostly experimented, learned how computers work, traded software with neighbors, and I played computer games. At that time, one of my mother's students became my first mentor, passing on some of his computer-science knowledge. I began to spend countless after-class hours in my middle school's computer lab, exploring "The Internet", a newfangled curiosity that had just recently arrived in mid-1990's Austria. I witnessed the beginning of the Dot-com era, and became an avid user of Internet Relay Chat (IRC) networks and various Bulletin Board Systems (BBS). In 1997, I gained access to "The Internet" also at home. Subsequently, I got in touch with a group of people in my region to organize LAN parties, share knowledge, modify computer hardware and software, build special purpose servers, and experiment with new technology.

By the time my school career came to an end, I had achieved an advanced level of understanding of computer architecture, operating systems, and network security. I began to read scientific papers, and surrounded myself with people working in the tech industry, academia, and the open source community, but pursuing academic studies was impossible. Subsequently, I briefly worked on a google-maps like web service,

and in 2002 began doing consulting as a freelancer next to working as consultant for a computer company, formally completing apprenticeship in computer engineering as well.

The tech sector evolved rapidly during the 2000's, and so did my consulting job. Supporting corporate clients, governmental organizations, and hospitals on computer and network security, failure analysis, and technical advise, I became department leader in 2005. In retrospect, it was a busy and exciting time, and I learned on the job how important good systems engineering and management can be.

In 2007, I joined Ars Electronica as computer and network security expert. Industrial R&D took up more and more of my time, as I gradually replaced the aging corporate servers and network architecture with a modern failure- and fault-tolerant one. I helped organize large-scale events for tens of thousands of participants. In this truly international and interdisciplinary environment I worked with artists on realizing experimental "cyberarts" showcases, for who hard limitations of technology were just minor obstacles that had to be overcome for the sake of art, science, and public outreach. Working there showed me that interdisciplinary collaborations between scientists, engineers, and artists can achieve much. Many Ars Electronica members actually were scientists, and in part this motivated me to finally pursue academic studies.

After obtaining university qualification through evening school, I began to study at the University of Applied Science Upper Austria, Hagenberg, for my Bachelors degree, which felt like holidays compared to evening school. As part of a research project in the curriculum, I began to work at the Fraunhofer Institute for Secure Information Technology (SIT) and Applied and Integrated Security (AISEC) in Germany. I continued my research there for several years on industrial-scale malware analysis, reverse engineering, and classification. Upon receiving my Bachelor's degree in Austria, I moved back to Germany and pursued a Master's degree at the Technical University Munich (TUM). Between 2010 and 2012, I and my colleagues at AISEC established a fully automated malware collection, analysis, and classification environment, exploiting virtualization and machine learning. Several research papers and a book chapter were published on this research.

I began to work for the GNU project and therefore moved on from AISEC, and then had the opportunity to conduct a research project with and for Airbus (EADS at that time). In this project, I conducted research on Airbus' then newly standardized fault-tolerant avionics network technology ARINC664/AFDX, and provided feedback on potential future improvements. This project exposed me to avionics for the first time, and I became interested in their spaceflight applications. I learned about a student-run CubeSat project ongoing at the Institute for Astronautics of Prof. Ulrich Walter (DLR/STS-55). Hence, I left the open source project I was professionally involved in at that time, and began to work on the FirstMOVE satellite.

After launching FirstMOVE into space in late 2013, we conducted on-orbit operations and solar cell validation for Airbus Space & Defense for two months. Then we lost our ability to control the satellite. The next half year, we conducted a truly rigorous post-mortem analysis for the funding agency DLR, the German space agency, which many considered complete overkill, but for us was incredibly valuable. We reviewed and analyzed all available documentation generated throughout the years, checked all hardware designs used in FirstMOVE, including historical ones and alternations made to them. We spent many hours conducting face-to-face and remote interviews with

current and prior project members, most of who had at that time begun to work in the industry and were dispersed all across the globe. This analysis indicated that the on-board computer was the cause for FirstMOVE's failure, and a lack of suitable diagnostics functionality prevented recovery of the spacecraft. We published a redacted version of these results and lessons-learned.

By the end of the post-mortem, a group of students had formed to begin working on a successor satellite – MOVE-II. I took on the supervisor role of the on-board computing team, which initially included all computerized subsystems, including also COM, and payload data handling. As one of the main designers, I began to develop a fault-tolerant system architecture for this satellite, just to discover that there exists no suitable technology to enable it. This astonished me and my colleagues, and I sought advice from the European Space Agency's technical directorate (ESA TEC-EDD), looking for clues to solutions that, I assumed, surely had to exist. It became clear that there was simply no technology which could enable robust and reliable on-board computer consisting of CubeSat-style hardware, and no suitable protective concepts, or ready made solutions existed. On the positive side, I henceforth was in contact with the right people at ESA, who initially gave me many requirements to work with. This kickstarted my on-board computer fault tolerance research, the results of which are described in this PhD thesis.

I have been pursuing research on satellite computer architecture and fault tolerance ever since we began working on MOVE-II. And together with my colleagues in Munich, Nadia, and her colleagues in astronomy in Leiden, we published several research papers and journal articles in the different fields my research is connected to. During some of these conferences, my contacts at ESA encouraged me to expand my research and offered support in pursuing research grants. In 2015, my research on fault-tolerance and computer architecture had outgrown the MOVE satellite program, and I established ties and made preparations for proposing for funding. At the end of 2015, my first proposal was awarded funding through the Networking and Partnership Program of the European Space Agency.

My Master's thesis essentially was a summary of my work within the MOVE-II satellite project at that time, and the main challenge was to compress 3 published scientific papers and two additional research projects reports into a single Master's thesis. For this work, I was awarded the first prize of the ZARM Award for Young Scientists, as well as monetary grant from the Center of Applied Space Technology and Microgravity in Bremen, Germany. I also presented my results at the Conference on Small Satellites of the American Institute of Aeronautics and Astronautics organized at Utah State University (AIAA/USU SmallSat), where I participated in the Frank J. Redd Student Competition of the AIAA and won the second prize. SmallSat truly has been a remarkable and inspiring experience every since I attended it the first time back then.

In early 2016, I moved to The Netherlands, into close proximity to ESTEC, ESA's technical research center and satellite testing facility. My PhD research began in July 2016 at Leiden University, and in November of the same year I received another ESA/NPI grant for my research. While deepening my research, I also sought to expand the scope of my research through collaboration. Together with a group of researchers from Singapore, Peru and Ethiopia through the Committee on Space Research (COSPAR), we developed an ultra-low cost satellite relay constellation. We won the COSPAR Small Satellite Design Competition in 2017 with this concept, and

published a paper on it. In the second half of my PhD I had the privilege of giving talks and lectures at institutions in East Asia, the Americas, and in Europe. Finally, I spent a good part of 2019 as guest researcher at National Tsing Hua University, in Taiwan. Most of my successful and productive collaborations today are international.

Acknowledgments

First, I would like to express my gratitude to Ewine van Dishoeck and Tim de Zeeuw for their help, support, and occasional advise throughout the past years.

Dear Daniel, Hello! Phyllis and Stanley, thank you! Irene, I understand how you felt after your PhD.

Thanks to Robert Perea Tamayo, Eyoas Ergetu, Li BingXuan, and Percy Castro Mejia for our collaboration! Thank you, Pai Chou, for the opportunity to learn from you. Lai Shih-Ping, thank you for all the times you have been our host. Chang Hsiang-Kuang, thank you for your kind advise! King Chung-Ta, thank you very much for the guest researcher stay and the introductions you made.

Best regards to the PADS students, especially to Cheng Yu-Min and Ho Chikai for the Mandarin translation! Lu Shyue-Kung, thank you for your advise, and proofreading! Thanks to Stefan Holst and Unknown for the Japanese translation, and to Notsu Shota for editing. Thanks to Dazhi and Peng for the Chinese summary. Credits to Niels and Sierk for the dutch translation!

Jelena & Di, thanks for warning me back in 2016 the only way you could.

Gary Swift, it has been a truly remarkable experience to learn from you. Best regards to the XRTC community. To Melanie Berg, a kindred spirit.

Thanks to Prof. Walter at TUM for being introduced into the field of spaceflight, to Prof. Schmucker for his excellent rocket-technology lecture, to Martin for asking the right first questions in 2012, and to Carsten. Nico, it has been a pleasure to work with you on MOVE-II, do not let the huge egos dishearten you.

Gianluca, I will always remember your first email. Giorgio, thank you for taking an interest in my work, and for listening at the right time in 2016. Thank you for getting me started on a journey that could not be more exciting. I hope that these results proof useful to ESA in the coming decades.

Thank you, Niki, for being my first Mentor! Prof. Gil & Phyllis Moore, you are an inspiration to me. Thanks Marianne Sidwell, Stan Kennedy, and those involved in the SmallSat student competition.

To the makers of Star Trek: TNG and of The Expanse: you have created the most outstanding science outreach ever.

Finally, Nadia, thank you for being with me all these years! Thank you for standing with me through our time in Holland, and for encouraging me.