Fault Tolerant Nanosatellite Computing on a Budget

Christian M. Fuchs, Nadia M. Murillo
Leiden University & Leiden Observatory
Niels Bohrweg 1&2, 2333CA Leiden, Netherlands
christian.fuchs@dependable.space

Pai Chou, Jing-Jia Liou, Yu-Min Cheng
National Tsing Hua University
No.101, KuangFu Rd., Hsinchu 30013, Taiwan
{phchou@cs, jjliou@ee, s107062541@m107}.nthu.edu.tw

Xiaoqing Wen, Stefan Holst
Kyushu Institute of Technology
Kawazu 680-4, Iizuka, Fukuoka 820-8502, Japan
{wen, holst}@cse.kyutech.ac.jp

Antonios Tavoularis, Gianluca Furano, Giorgio Magistrati, Kostas Marinis
European Space Agency, ESTEC, Noordwijk, Netherlands
{antonios.tavoularis, gianluca.furano, giorgio.magistrati, kostas.marinis}@esa.int

Shyue-Kung Lu
National Taiwan University of Science and Technology
No.43, Keelung Rd., Sec.4, Da'an Dist., Taipei City 10607, Taiwan
sklu@ee.ntust.edu.tw

Aske Plaat
Leiden Institute for Advanced Computer Science
Niels Bohrweg 1, 2333CA Leiden, Netherlands
a.plaat@liacs.leidenuniv.nl

ABSTRACT

In this contribution, we present 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. We describe in detail the design of our OBC's breadboard setup, and document its composition from the component-level, all the way down to the software level. Fault tolerance in this OBC is achieved without resorting to radiation hardening, just intelligent through software. The OBC ages graceful, and makes use of FPGA-reconfiguration and mixed criticality. It can dynamically adapt to changing performance requirements throughout a space mission.

We developed a proof-of-concept with several Xilinx Ultrascale and Ultrascale+ FPGAs. With the smallest Kintex Ultrascale+ KU3P device, we achieve 1.94W total power consumption at 300Mhz, well within the power budget range of current 2U CubeSats. To our knowledge, this is the first scalable and COTS-based, widely reproducible OBC solution which can offer strong fault coverage even for small CubeSats. To reproduce this OBC architecture, no custom-written, proprietary, or protected IP is needed, and the needed design tools are available free-of-charge to academics. 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.

INTRODUCTION

Cheap, embedded and mobile-market electronics 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 mechanics, 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 a summary of our proof-of-concept, 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 the third section, we describe our OBC's component-level architecture, the MPSoC used, as well as the interplay between the different components of the OBC. In the fourth section, we describe the software-implemented fault tolerance measures used. Before providing conclusions, we present our implementation results and details about the validation of this OBC architecture. 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.

BACKGROUND AND RELATED WORK

In contrast to the initial generation of educational CubeSats, today fewer satellites fail due to practical design problems caused by inexperience [1]. Instead, Langer et al. in [2] showed that the 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 [3].

The main source of failure there 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-Profile and Radiation

A satellite's OBC must cope with challenging design constraints, and a fault-profile otherwise only found in irradiated environments. The main source for faults within electronics in the space environment are highly charged particles. These arrive as Cosmic Rays from beyond our solar system, and are ejected by the Sun constantly, and at an increasing rate during solar particle events [4]. Particles interact with a spacecraft's electronics, and can induce different effects in a semiconductor depending on the type of particle and its charge. Radiation can corrupt logical operations or induce bit-flips within semiconductor logic and memory (single event effects - SEE), and may cause displacement damage (DD) at the molecular level, or induce a latch-up. The cumulative effect of charge trapping in the oxide of electronic devices (total ionizing dose – TID) further impacts the lifetime of an OBC. Radiation events can also cause functional interrupt in circuits, interfaces, or even entire chips (single event functional interrupts – SEFIs).

All these effects in practice can result in spontaneous or drastically accelerated aging compared to ground applications, which must be handled efficiently throughout an entire space mission. To do so, traditional space-grade hardware makes 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. The energy threshold above a which particles can induce transient faults in chips manufactured in finer technology nodes decreases, which overall can be seen as beneficial from a CubeSat perspective. However, recent generation semiconductor manufactured in certain modern technology nodes (e.g. FinFET and FD-SoI), show better performance under radiation than predicted by models [5]. However, the ratio of multi-bit upsets or permanent faults is also increased, causing experienced faults to in general be more severe IF a fault occurs.

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.

Such limitations, however, can often be overcome in combination with additional fault tolerance measures. Fault tolerance concepts for ground and atmospheric aerospace applications can therefor serve as building blocks to design a fault tolerant architecture for space applications.

Fault-Tolerance for Large Spacecraft

Traditional OBCs for large satellites realize fault tolerance using circuit-, RTL- [6], IP-block- [7], [8], and OBC-level TMR [9] through costly, space-proprietary IP. Circuit-, RTL-, and core-level measures are effective for small microcontroller-SoCs [10], [11], 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 [10]. 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 [12], [11]. 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 [13] with two or three very simple processor cores, even for ASIC implementations [11], [7]. 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 [14], for which traditional radiation-hardening concepts become ineffective. Specifically, CubeSats utilize COTS microcontrollers and application processor SoCs, FPGAs, and combinations thereof [15], [14]. 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 [1]. 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) [16]. 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) [17] based MSP430FR family [18]. 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 fail-over, 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 [1], [14]. 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.

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 [15] due to the possibility to recover from faults through reconfiguration. Transients in configuration memory (CRAM) can usually be recovered right away through reconfiguration [19], while permanent faults may be mitigated using alternative configuration variants. However, fine-grained, non-invasive fault detection in FPGA fabric is challenging [10], and is a subject of ongoing research [20], [21]. 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 [22], but was technically unfeasible and inefficient until few years ago. Modern semiconductor technology allows us to overcome these limitations and recent research [23], [24] 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 [25], [26]. They require custom processor designs [23], or programming models which are suitable for accelerator applications [24]. The fundamental concept of software-implemented coarse-grain lockstep, however, is flexible and can be applied, e.g., to MPSoCs for safety-critical applications [27], [23], networked, distributed, and virtualized systems [28].

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 presented in this contribution 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.

However, conventional MPSoCs follow a centralist architecture with processor cores sharing functionality where possible to minimize footprint, optimize access delays, improve routing. 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.

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 1.

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 [29] for this purpose, which also is included on most commercial FPGA development platforms. However, NOR-flash is inherently prone to radiation [29], and phase-change memory (PCM [30]) 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 6, 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

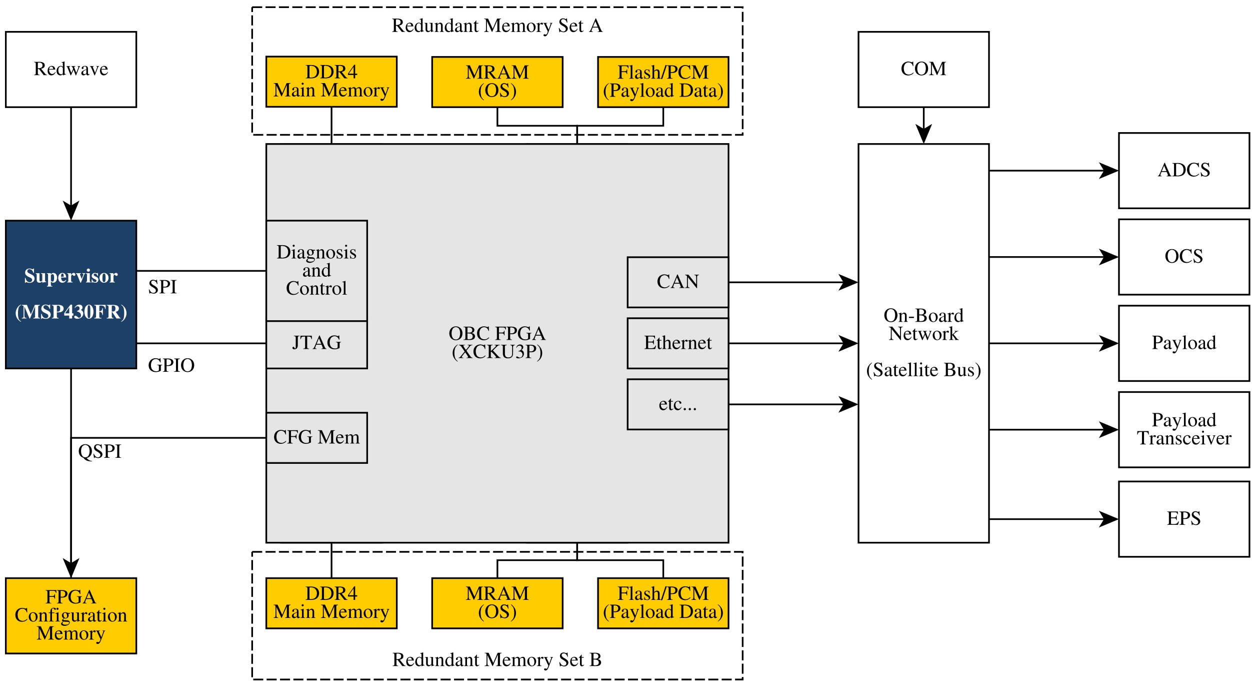


Figure 1. A component-level diagram of our OBC architecture. This architecture is intended as an in-place substitute for a conventional ASIC-based System-on-Chip, and only adds a second set of memory ICs to counter component-level failure.

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 the fourth section. 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 [18]. A space-grade substitute is available in the form of the MSP430FR5969-SP.

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 [31] (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, though with modern high-density components manufactured in fine technology nodes, the likelihood to experience bit-upsets is low [32], [33]. Hence, for most nanosatellite missions simple error correction coding (ECC) is sufficient to protect the integrity of data stored as long as error scrubbing is implemented [34]. 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 prior research [35] ECC can be implemented using standard Xilinx Library IP [36], 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 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 [37] 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 prior work, we presented MTD-mirror [38], 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 [33]. 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 fail-over in hardware, but merely connect the two memory sets to the FPGA. All fail-over functionality is realized through the topology of our MPSoC and in software.

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 the fourth section of this paper. Its logic placement is depicted in Figure 2, 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 paper has 4 Xilinx Microblaze processor cores, and therefore 4 compartments, which are depicted in brown, green, 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 3, several MPSoC components related to FPGA housekeeping are placed in static logic:

- the mentioned configuration controller makes up only a minor part of the pink-indicated logic,
- the supervisor's debug interface (further described at the end of this section),
- 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 [39]).

Researchers showed in related work [40], [41] that faults within an FPGA can effectively be resolved through reconfiguration, or mitigated using alternatively routed and placed configuration variants. 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

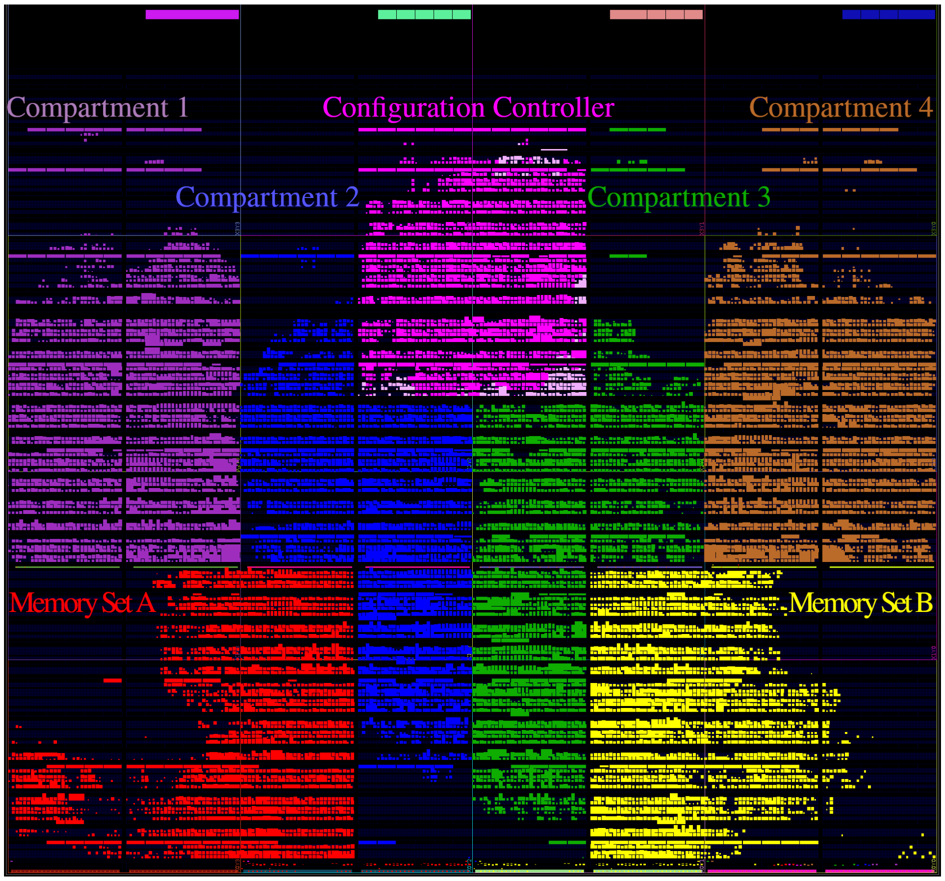


Figure 2. Logic placement of our proof-of-concept MPSoC on a Xilinx Kintex Ultrascale+ KU3P with 4 compartments (purple, blue, green, and brown), two shared memory controller sets (red & yellow) and static logic (pink).

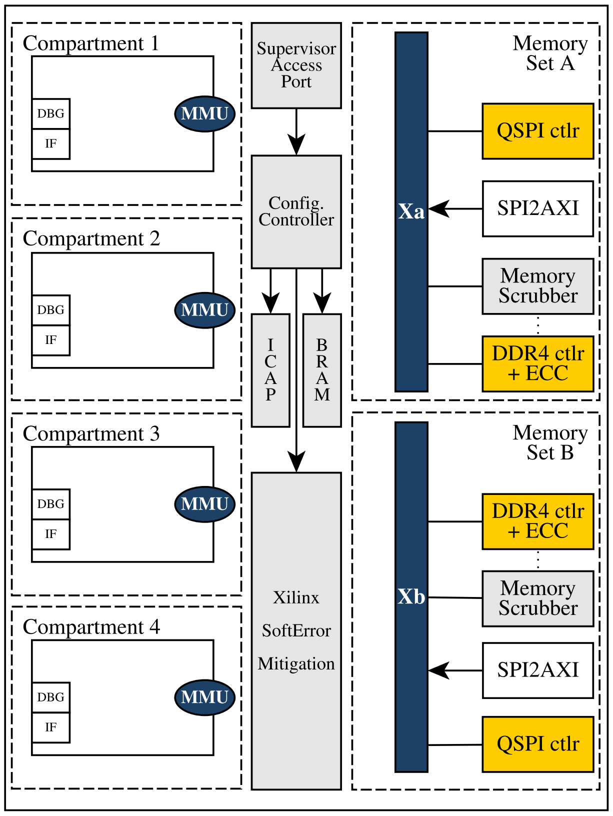


Figure 3. Block-level layout in our MPSoC. Partial reconfiguration partitions are indicated with dashed lines. Compartment and memory controller sets ($X_{a/b}$) can be reconfigured without interruption.

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 [19].

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 [42] 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.

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 [43]. 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. The topology of a compartment is depicted in Figure 4. 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 the final part of this section.

In general, for the sake of reliability, the use of SPI or I2C based satellite bus architectures is in general discouraged. However, in [44], 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 [45], 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 the next section. 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.

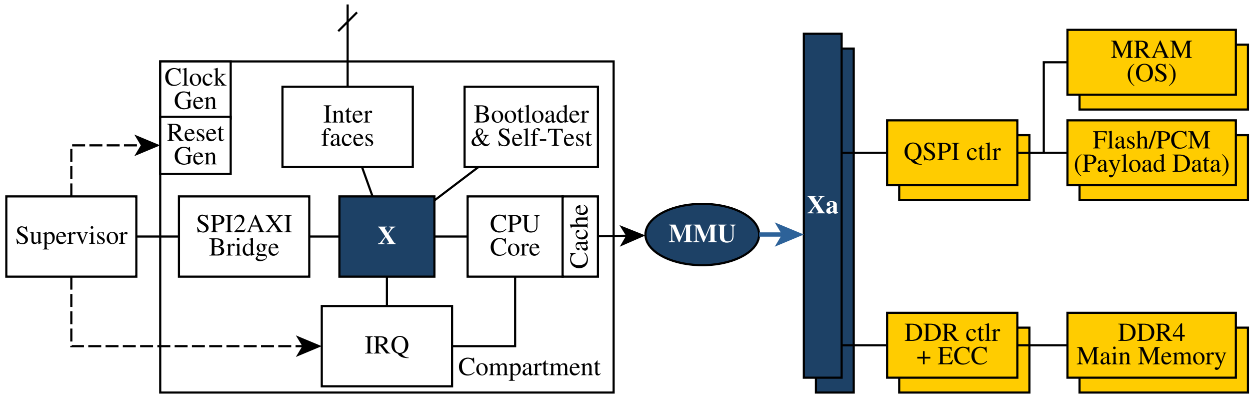


Figure 4. The memory and logical topology of a compartment. The compartment local and the global memory controller interconnects are logically isolated. A compartment's processor core has access to the memory controller sets and to compartment-local controllers. Access to compartment-local controllers bypasses the cache.

Beyond memory EDAC, the logic of a compartment does not offer hardware-implemented fault tolerance capabilities. Instead, we address this limitation at the system level. Our coarse grain lockstep functionality enables us to detect faults in the fabric with compartment granularity [46]. In practice, this closes the fault-detection gap.

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.

In case one memory controller set fails, MPSoC compartments that were using this set will switch to fail-over 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 [47].

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

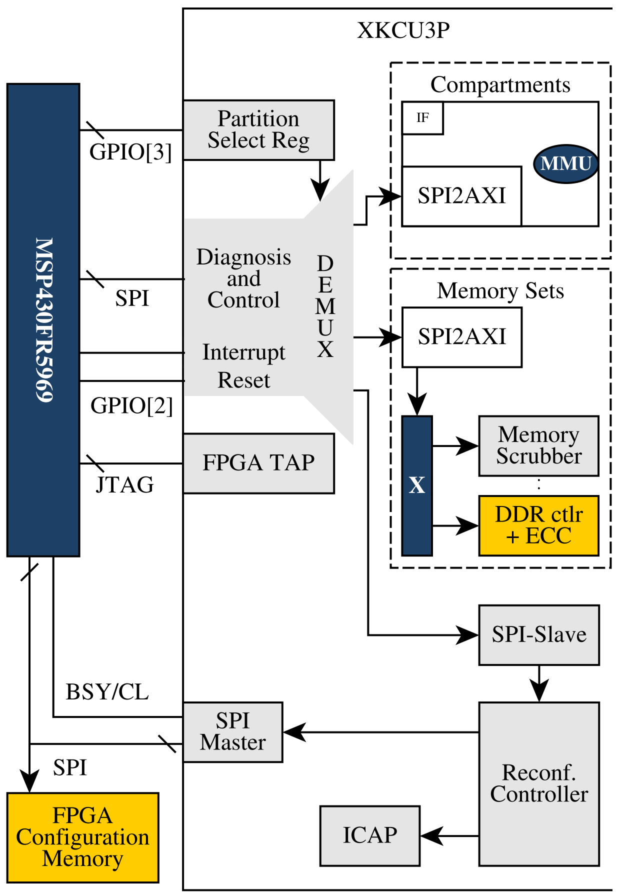


Figure 5. The design of our supervisor-FPGA control and diagnostic interface including the debug-facilities used by the supervisor to access different compartments of the MPSoC.

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 5, the supervisor can utilize it's SPI interface to access the different components of the MPSoC in a controlled and performance-efficient manner. It can disable individual compartments in case of failure by using existing circuitry required for partial reconfiguration, as indicated in Figure 4. 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.

Fault Tolerance through Software

We enable fault tolerance for our MPSoC through software functionality, while using just COTS components and proven standard library logic at the hardware level. To assure fault tolerance for the flight software run on our OBC MPSoC, we utilize a multi-stage fault tolerance mechanic. The high-level functionality of this approach is depicted in Fig. 6, and consists of three interlinked fault mitigation stages:

Stage 1 utilizes a coarse-grain lockstep at the operating system level to link together multiple compartments of our MPSoC. A fault within one compartment in practice causes a malfunction, which alters the state of a satellite's flight-software run on that compartment. We then use the functionality of the lockstep to compare this state, and generate a distributed majority decision across compartments. This enables us to assure that compartments operate correctly, enables us to localize a fault with compartment granularity. To recover from a fault, the lockstep enables us to copy the state of a correct compartment to recover a faulty compartment to a correct state, which in practice yields forward error correction. This functionality is described further in the next first part of this section.

Stage 2 recovers failed compartments through FPGA reconfiguration and CRAM error scrubbing to recover functional compartments and restore access to memory sets. Permanent faults are mitigated through reconfiguration with differently placed alternative partition variants. A compartment affected by multiple permanent defects may eventually no longer be able to support the same functionality as a completely intact one. However, related work showed that it is often still possible to utilize a simpler partition configuration, running at lower clock frequencies or with fewer system features implemented in logic. Individual parts of a partition may therefore fail, but this does not mean that it can not be used anymore at all. In practice, this enables graceful aging and prevents the system from failing spontaneously due to permanent faults.

Stage 3 engages when too few functional compartments are available due to accumulating permanent defects in an aged FPGA. It re-allocates processing time between the different applications run on the OBC, to maintain stability for the most critical parts of the flight software. This is done in an automated manner by exploiting mixed criticality, which is inherent to a satellite's flight software. It can, for example, sacrifice performance of payload processing tasks to maintain robustness for critical applications, e.g., commandeering- and power-management related tasks. This enables an MPSoC to dynamically trade OBC performance to save energy, increase fault tolerance for critical applications, or assure that as much of functionality of the flight-software can be run as possible.

Stage 1: Coarse-Grain Lockstep for Flight Software

The objective of Stage 1 is to detect and correct faults within a compartment, and assure a consistent flight software state through checkpoint-based FEC. It is implemented by running flight software application in coarse-grain lockstep on two or more compartments. A functionality of this stage is depicted in white in Figure 6, and can be implemented within an operating system kernel, an application, or in bare-metal software.

Each compartment will compare the state of the application it runs with other compartments that run additional application copies. The supervisor reads out the results of the decentralized voting decision, and maintains a fault-counter for each compartment. This counter is used to determine if a fault in one compartment could be correctly resolved by copying a correct state from another one, or if a reboot is required. In the latter case, the supervisor will replace the compartment to allow non-stop operation, so that the rest of the OBC does not have to wait for the reboot to finish. If a compartment then still behaves incorrectly, the supervisor can perform low-level diagnostics, reconfiguring it in Stage 2, with the expectation of repairing defective logic.

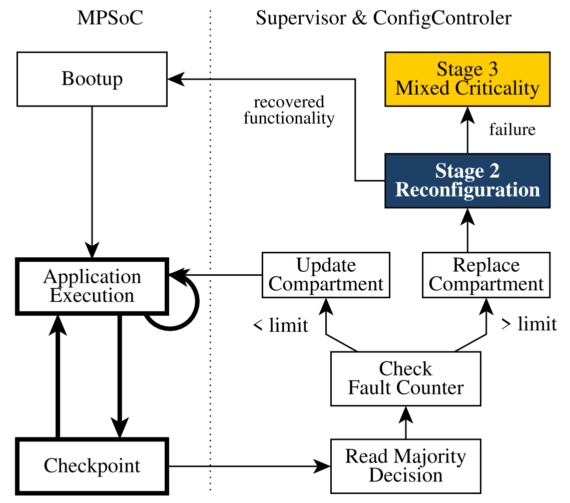


Figure 6. Stage 1 (white) assures fault detection (bold) and fault coverage. Stage 2 (blue) recovers defective compartments and repairs memory controller sets. Stage 3 (yellow) adapts the OBC's application schedule to retain a stable system in a strongly degraded OBC with many permanent faults in the FPGA fabric.

The OS will in regular intervals execute checkpoints, during which the state of the different applications is compared to facilitate the lockstep. Flight software applications can provide four callback routines to minimize the computational cost of this comparison and fault recovery. These routines are executed by the operating system's kernel: leftmargin=.35cm

- an optional *initialization routine*, to be executed on all compartments at bootup to prepare the application's environment, regardless of if it is active there or not;
- a *checksum callback*, which will generate a checksum from variables and data structures used in the specific application which make up its state. This checksum is subsequently used for state comparison during checkpoints;
- a *synchronization callback*, which exposes all application state relevant data;
- an *update callback*, which is executed by a compartment to retrieve or restore the correct state for its applications.

These callbacks enable our lockstep implementation to utilize application intrinsic information to assess the health state of a compartment, without actually requiring knowledge about the applications themselves. Besides the addition of these callbacks, no alterations to an application's logic is necessary.

Stage 1 can deliver real-time guarantees if required, and the tightness of the RT guarantees depends upon the time required to execute application callbacks. The only requirement towards the protected applications is the possibility to interrupt operation to run checkpoints periodically. These checkpoints are triggered by the supervisor. In our RTEMS/POSIX-based implementation applications can delay checkpoints slightly, to assure that an application has reached a suitable point for checksum comparison. We also implemented Stage 1 successfully in simple baremetal software running on a RISC-V MPSoC, where checkpoints are part of the application software. To support implementation of our coarse-grain lockstep, the OS only has to support interrupts, or be time-deterministic. To the best of our knowledge, such functionality is available in all widely used RT- and general purpose OS implementations, as well as in most scientific instrumentation processing systems.

The required development effort for implementing these callbacks depends on the flight software to be protected, but is in general very limited. For flight software based on state machine driven event loops, the relevant state data is usually limited to a few numeric variables, several status-flag variables, as well as data buffers, and 2D or 3D arrays storing partially received data. To date, we adapted several sophisticated astronomical instrumentation applications used aboard JWST and Hubble to support our architecture, which required no more than 3 weeks of development time in each case. For these applications, the relevant callback routines could be implemented with just 10-20 lines of C-code each, most of which being memcopy and CRC32 library function calls, as well as value assignments for variables and data structures in heap and stack.

Flight software often includes also applications that are not executed continuously, but instead run briefly for few seconds. For these, we can forego providing lockstep callback routines, as it can be much simpler to just allow such an application to exit correctly and directly return checksum to the OS. Often however, these applications also have no persistent state, store their state directly, e.g., in a file or database. Such information can then be copied by the checkpoint routine directly, without requiring cooperation by the application.

We prove a more in-depth description of the functionality of Stage 1 for a software-engineering audience in [48]. In Figure 7 we provide a brief practical example of fault detection and recovery process using this lockstep with the OBC design presented in this paper. In this example, a fault has occurred while executing the third checkpoint (blue) on compartment C_2. The failed compartment is then replaced with the spare C_3, and the flight software formerly run on the defective compartment are reassigned to C_3. C_3 will then copy the correct flight software state from one of the other compartments, in this case from C_1. The replaced compartment, C_2, can subsequently be tested for defects, and may subsequently be recovered through reconfiguration in Stage 2.

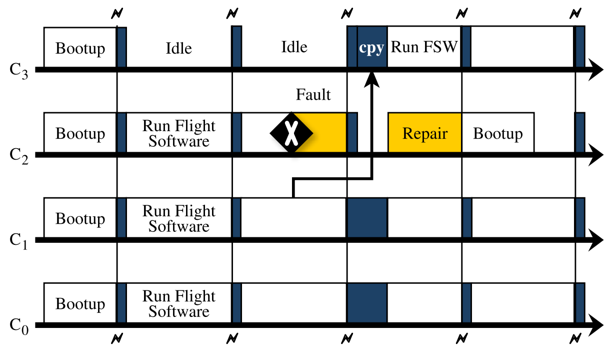
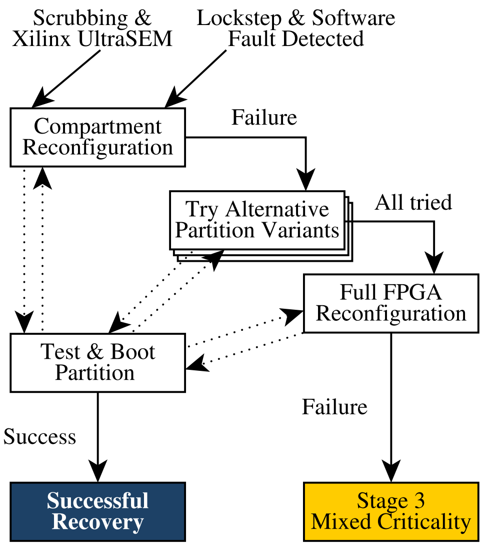


Figure 7. An example of fault detection and recovery mechanics of Stage 1. Note that all these operations only happen in software, requiring no hardware-implemented fault tolerance measures and no custom-written logic.

Stage 2: MPSoC Repair & Recovery

The previous stage can detect and correct faults as long as a sufficient number of functional compartments are available. Two intact compartments are required to realize fault detection, and three to achieve majority voting. Stage 2's task is to assure that sufficient intact compartments are available at all times.

Our software-implemented fault tolerance measures still require spare compartments to handle the permanent failure of a compartment. Over-provisioning of these spares naturally is inefficient and the OBC may eventually run out of spare compartments. Stage 2 is designed to test, repair, validate and recover defective



Scrubbing & Xilinx UltraSEM
Lockstep & Software Fault Detected
Compartment Reconfiguration
Failure
Try Alternative Partition Variants
All tried
Full FPGA Reconfiguration
Test & Boot Partition
Failure
Success
Successful Recovery
Stage 3 Mixed Criticality

Figure 8. The objective of Stage 2 is to recover defective compartments and other logic through partial and full FPGA reconfiguration via ICAP. If this is unsuccessful and too few functional compartments are available, Stage 3 is activated to find a more resource conserving application schedule, to retain system stability.

compartments, by using CRAM-frame error detection and partial reconfiguration. However, this only protects part of the FPGA fabric, and would be insufficient for detecting faults in compartments. The coarse grain lockstep functionality of Stage 1 enables us to detect faults in the fabric with compartment granularity. In practice, this closes the fault-detection gap left by scrubbing and configuration erasure coding.

Transient faults can corrupt the running configuration of the FPGA, thereby breaking the functionality of a compartment or redundant memory set [49], [50]. Even if parts of the FPGA fabric are damaged permanently, the residual highly-redundant FPGA fabric will remain intact and can be re-purposed [51]. Researchers in related work [40], [41] showed that faults within an FPGA can effectively be resolved through reconfiguration. Even permanent faults can in most cases be covered using alternatively routed and placed configuration variants that do not place critical logic in damaged regions of the FPGA fabric. We can provide multiple such configuration variants for each partition, as well as for static logic, allowing even a severely degraded FPGA to be recovered.

As depicted in Figure 6, if a compartment has repeatedly experienced failures which were detected by the lockstep, it will be replaced by a spare. The supervisor will then attempt to test and recover the compartments using partial reconfiguration, as depicted in Figure 8. The supervisor validates the relevant partial reconfiguration partition to detect permanent damage to the FPGA fabric. If reprogramming was unsuccessful and the faults persist, the supervisor will repeat this step with differently placed partition variants. Finally, faults within shared logic can be resolved using full reconfiguration, which implies an OBC reboot. For further details on how this functionality can be implemented, see [52].

All these steps are performed autonomously by the supervisor and the configuration controller. However, if a compartment can not be repaired through automated reconfiguration, Stage 2 can be used to generate additional diagnostic information for further analysis. The satellite operator can then conduct a detailed fault analysis on the ground, and craft a suitable replacement configuration to avoid utilizing defective areas of the FPGA.

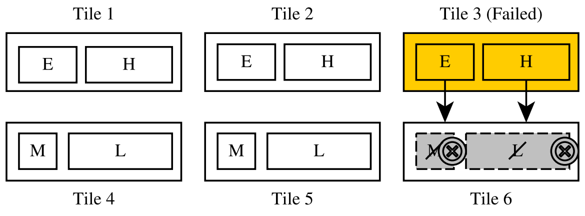
Stage 3: Graceful Aging through Mixed Criticality

Stage 3 engages if both partial- and full reconfiguration are unsuccessful and insufficient intact compartments are available. Its primary objective is to autonomously maintain system stability of an aged or degraded OBC to bring it into a safe state. The operator can then define a more resource conserving satellite operation schedule, or sacrifice link capacity, and OBC storage space.

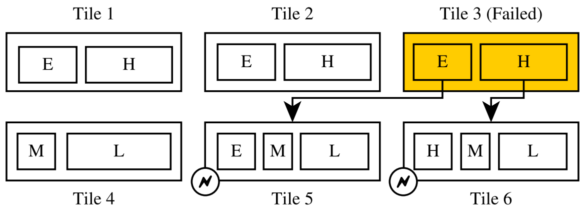
The criticality of the different applications making up a satellite flight software can be differentiated based on its relevance: ranging from essential commandeered-related parts of the flight software to uncritical payload data processing tasks. Performance degradation, or even a loss of less important parts of the flight software, is usually preferable over an unstable system.

As part of the functionality of Stage 1, applications can be migrated between different compartments of the OBC, and the level of replication for each application can be adjusted at runtime. Multiple such replicated application groups can coexist. In case of failure, low criticality applications can be discarded or assigned less compute performance to take over functionality from failed compartments that were running more important applications.

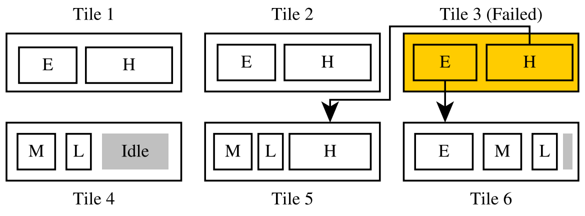
To visualize how Stage 3 can maintain system stability in a larger MPSoC, Figure 9 depicts an OBC with six compartments that are running four applications with different criticality. The third compartment in this example has failed but no idle spare compartments are available. Stage 3 allows us to resolve this failure in the following ways:



(a) Migration by low-criticality thread pruning.



(b) Migration through clock-speed increase.



(c) Migration through processing time reduction.

Figure 9. A 6-compartment MPSoC running 4 applications of mixed criticality (**E**ssential, **H**igh, **M**edium, and **L**ow), where compartment 3 (yellow) has failed. To assure computational correctness for the higher criticality threads, different recovery strategies are possible, which can avoid keeping around idle spares.

- The applications on the failed compartment can be relocated to one running less critical applications, and replace them as depicted in Figure 9a.
- Instead of entirely de-scheduling one copy of the low and medium priority applications, the clock frequency on two compartments could be increased, allowing one of each high-criticality application to be migrated. In Figure 9b, this is depicted by moving the applications replicas to compartments 5 and 6 without de-scheduling instances of the less critical applications. Most modern embedded and mobile-market processor cores support frequency scaling and multi-clocking.
- Finally, in contrast to increasing the clock frequencies of individual compartments, the less important applications could also just be assigned less processing time as shown in Figure 9c.

The ideal recovery strategy, depends on the current performance requirements towards the OBC. Additional thoughts on this aspect are discussed, e.g., in [26], where different replacement strategies are described at a more mathematical level.

A satellite operator can use this functionality to dynamically adjust the performance of an OBC implementing this architecture during a space mission. This is achieved by adjusting the distribution of applications across the OBC, the level of replication of each application, and the processing time allocated to each of them. The objectives to achieve maximum performance, power-saving capacity, and fault tolerance strength compete with each other, and one can be prioritized over the others. This is visualized in Figure 10. This functionality is analogous to the powersaving capabilities present in today's mobile devices, and consumer mobile devices and laptop computers. Further information on Stage 3 including dynamic application remapping, as well as performance, energy and robustness prioritization at run-time is available in [53].

Proof-of-Concept Implementation Results

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 these designs is available in [44], 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 paper is processor and platform independent, with the MPSoC presented here implemented using Xilinx Microblaze processor cores. It was implemented for a Xilinx Kintex Ultrascale+ XKU3P FPGA, where we achieve 1.94W total power consumption at 300Mhz. Vivado's power report is depicted in Figure 11. This MPSoC design can be reproduced with the Xilinx Vivado Design Suite version 2018.3, and we have created earlier MPSoC versions with 2017.1 and later.

To test our implementation, we have conducted fault injection through system emulation into an RTEMS implementation of Stage 1. Early fault injection results for this test campaign were published in [46] at the end of 2018. A full fault-injection result report is pending publication. We also constructed a multi-core model of our MPSoC also in ArchC/SystemC to conduct further fault-injection close-to-hardware. The results show that

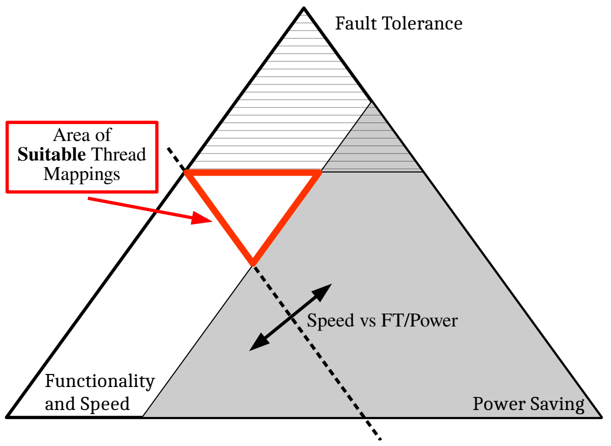
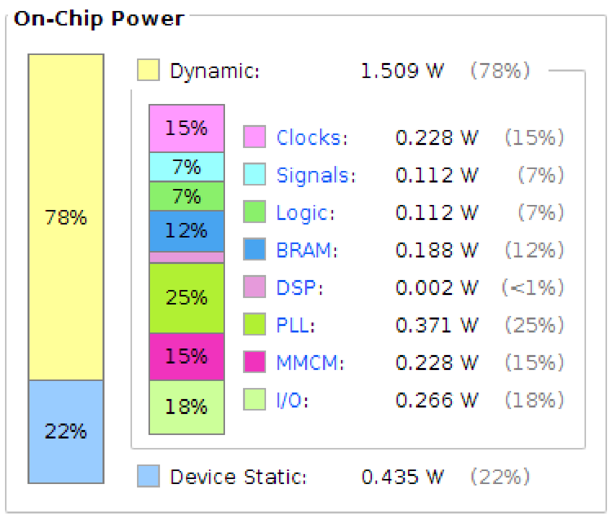


Figure 10. Our OBC allows the system properties of fault tolerance, performance, and energy consumption to be adjusted at runtime. The spacecraft operator can prioritize one of these objectives, e.g., to achieve minimum energy consumption by sacrificing processing speed, while maintaining a given level of fault tolerance.


On-Chip Power

78%	Dynamic:	1.509 W	(78%)
15%	Clocks:	0.228 W	(15%)
7%	Signals:	0.112 W	(7%)
7%	Logic:	0.112 W	(7%)
12%	BRAM:	0.188 W	(12%)
	DSP:	0.002 W	(<1%)
25%	PLL:	0.371 W	(25%)
15%	MMCM:	0.228 W	(15%)
18%	I/O:	0.266 W	(18%)
22%	Device Static:	0.435 W	(22%)

Figure 11. Power consumption of the 4-core MPSoC powering our OBC. Figure generated by Xilinx Vivado 2018.3.

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.

To achieve worst-case performance estimations, we measured the worst-case performance cost of the coarse-grain lockstep of Stage 1. The results of this test campaign are indicated in Figure 12. These benchmark results were generated based on code derived off a CCD readout program used for astronomical instrumentation. The application was executed with a varying amount of data processing runs in a tile group at the indicated checking frequencies, and without protection for reference. Note that to achieve worst-case performance overhead measurements, Stage 1 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, implying negligible with very little performance overhead ranging from 0.5% to 2% performance overhead. Note also that these benchmarks were also run in user-space, where thread-management is drastically more costly than in kernel-space. As expected, the performance cost also varies depending on workload, with data-heavy tasks (a),(b), and (c) showing better performance due to the increased cost of thread-management in user-space.

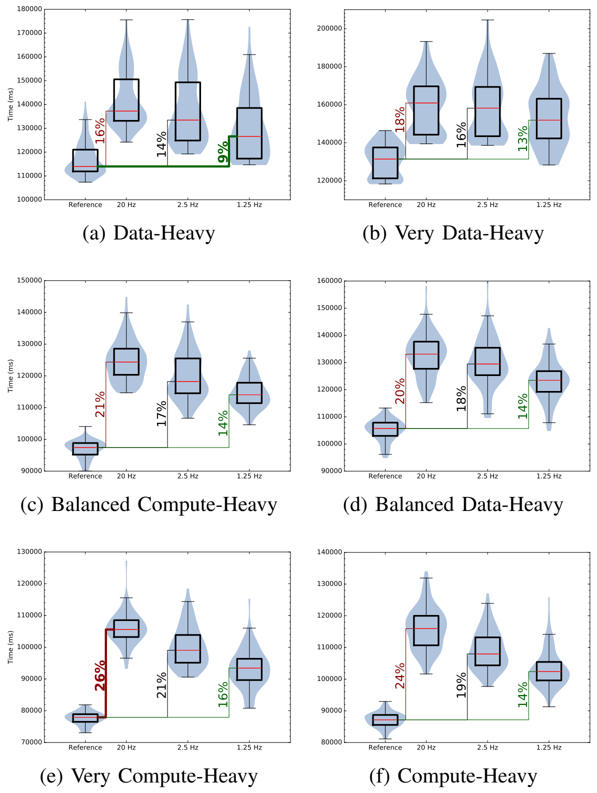


Figure 12. Performance measurements of 6000 runs for processing 100 2048x2048 CCD frames with different checkpoint frequencies and workloads.

Conclusions & Future Work

In this contribution, 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. We described in detail the design of our OBC's breadboard implementation, describing its composition from the component-level, to the MPSoC design 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 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 developed a proof-of-concept of our architecture for the smallest Kintex Ultrascale+ FPGA, KU3P, and achieve 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. Therefore, 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.

Today, 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.

REFERENCES

[1] J. Bouwmeester, M. Langer, and E. Gill, "Survey on the implementation and reliability of CubeSat electrical bus interfaces," *CEAS Space Journal, Springer*, 2017.

[2] M. Langer and J. Bouwmeester, "Reliability of CubeSats – statistical data, developers' beliefs and the way forward," in *AIAA/USU SmallSat*, 2016.

[3] M. Swartwout, "You say "Picosat", i say "CubeSat": Developing a better taxonomy for secondary spacecraft," in *2018 IEEE Aerospace Conference*, 2018.

[4] J. Schwank *et al.*, "Radiation Hardness Assurance Testing of Microelectronic Devices and Integrated Circuits," *IEEE Transactions on Nuclear Science*, 2013.

[5] M. D. Berg, K. A. LaBel, and J. Pellish, "Single event effects in FPGA devices 2014-2015," in *NASA NEPP/ETW*, 2015.

[6] C. Carmichael, "Triple module redundancy design techniques for Virtex FPGAs," *Xilinx Application Note XAPP197*, 2001.

[7] K. Reick *et al.*, "FT design of the IBM Power6 microprocessor," *IEEE micro*, 2008.

[8] M. Hijorth *et al.*, "GR740: Rad-hard quad-core LEON4FT system-on-chip," in *Eurospace DASIA*, 2015.

[9] K. D. Safford *et al.*, "Off-chip lockstep checking," Jun. 26 2007, uS Patent 7,237,144.

[10] A. Fedi *et al.*, "High-energy neutrons characterization of a safety critical computing system," in *IEEE DFT*. IEEE, 2017.

[11] X. Iturbe *et al.*, "A triple core lock-step ARM Cortex-R5 processor for safety-critical and ultra-reliable applications," in *IEEE DSN-W*, 2016.

[12] Á. B. de Oliveira *et al.*, "Applying lockstep in dual-core ARM Cortex-A9 to mitigate radiation-induced soft errors," in *LASCAS*. IEEE, 2017.

[13] 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.

[14] M. Swartwout, "The first one hundred CubeSats: A statistical look," *Journal of Small Satellites*, 2014.

[15] R. Carlson, K. Hand, and E. Ozer, "On the use of system-on-chip technology in next-generation instruments avionics for space exploration," in *IEEE VLSI-SoC, revised paper*. Springer, 2016.

[16] S. M. Guertin, M. Amrbar, and S. Vartanian, "Radiation test results for common cubesat microcontrollers and microprocessors," in *Radiation Effects Data Workshop (REDW)*. IEEE, 2015.

[17] Z. Zhang *et al.*, "Single event effects in COTS ferroelectric RAM technologies," in *Radiation Effects Data Workshop (REDW)*. IEEE, 2015.

[18] S. M. Guertin, "CubeSat and mobile processors," in *NASA Electronics Technology Workshop*, 2015, pp. 23–26.

[19] L. Bozzoli and L. Sterpone, "Self rerouting of dynamically reconfigurable SRAM-based FPGAs," in *NASA/ESA AHS*. IEEE, 2017.

[20] M. Ebrahimi *et al.*, "Low-cost multiple bit upset correction in SRAM-based FPGA configuration frames," *IEEE Transactions on VLSI Systems*, 2016.

[21] F. Rittner *et al.*, "Automated test procedure to detect permanent faults inside SRAM-based FPGAs," in *NASA/ESA AHS*. IEEE, 2017.

[22] T. Slivinski *et al.*, "Study of fault-tolerant software technology," 1984.

[23] M. Liu and B. H. Meyer, "Bounding error detection latency in safety critical systems with enhanced execution fingerprinting," in *DFT*. IEEE, 2016.

[24] E. Wachter *et al.*, "A hierarchical and distributed fault tolerant proposal for noc-based mpsocs," *IEEE Transactions on Emerging Topics in Computing*, 2016.

[25] 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 VLSI Systems*, 2016.

[26] U. Martinez-Corral and K. Basterretxea, "A fully configurable and scalable neural coprocessor ip for soc implementations of machine learning applications," in *NASA/ESA AHS*. IEEE, 2017.

[27] S. S. Sahoo, B. Veeravalli, and A. Kumar, "Cross-layer fault-tolerant design of real-time systems," in *DFT*. IEEE, 2016.

[28] Y. Dong *et al.*, "COLO: Coarse-grained lock-stepping virtual machines for non-stop service," in *ACM Symposium on Cloud Computing*, 2013.

[29] S. Gerardin *et al.*, "Radiation Effects in Flash Memories," *IEEE Transactions on Nuclear Science*, 2013.

[30] A. P. Ferreira *et al.*, "Using pcm in next-generation embedded space applications," in *RTAS*. IEEE, 2010.

[31] G. Tsiligiannis *et al.*, "Testing a commercial MRAM under neutron and alpha radiation in dynamic mode," *IEEE Transactions on Nuclear Science*, 2013.

[32] E. Benton and E. Benton, "Space radiation dosimetry in low-earth orbit and beyond," *Nuclear Instruments and Methods in Physics Research, Elsevier*, 2001.

[33] A. Samaras, F. Bezerra, E. Lorfevre, and R. Ecoffet, "Carmen-2: In flight observation of non destructive single event phenomena on memories," in *RADECS*.

[34] 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.

[35] C. M. Fuchs, "Enabling dependable data storage for miniaturized satellites," in *AIAA/USU SmallSat*, 2015.

[36] Z. K. Baker and H. M. Quinn, "Design and test of xilinx embedded ecc for microblaze processors," in *Radiation Effects Data Workshop (REDW)*. IEEE, 2016.

[37] C. M. Fuchs *et al.*, "FTRFS: A fault-tolerant radiation-robust filesystem for space use," in *Springer ARCS*, 2015.

[38] ——, "A fault-tolerant radiation-robust mass storage concept for highly scaled flash memory," in *Eurospace DASIA*, 2015.

[39] P. Maillard *et al.*, "Single-event upsets characterization & evaluation of xilinx ultrascale$^{\text{TM}}$ soft error mitigation (sem ip) tool," in *Radiation Effects Data Workshop (REDW)*. IEEE, 2016.

[40] C. Bolchini, A. Miele, and M. D. Santambrogio, "Tmr and partial dynamic reconfiguration to mitigate seu faults in FPGAs," in *DFT*. IEEE, 2007.

[41] G. Durrieu *et al.*, "DREAMS about reconfiguration and adaptation in avionics," *European Congress on Embedded Real Time Software and Systems (ERTS)*, 2016.

[42] C. M. Fuchs *et al.*, "Enhancing nanosatellite dependability through autonomous chip-level debug capabilities," in *Small Satellites, System & Services Symposium 2015 (4S)*. ESA, 2016.

[43] P. Munk *et al.*, "Toward a fault-tolerance framework for COTS many-core systems," in *IEEE EDCC*, 2015.

[44] C. M. Fuchs, N. M. Murillo, A. Plaat, E. van der Kouwe, and T. P. Stefanov, "Fault-tolerant nanosatellite computing on a budget," in *RADECS*. IEEE, 2018.

[45] Aeronautical Radio, INC, *ARINC Specification 664: Avionics Full Duplex Switched Ethernet (AFDX)*, 2005.

[46] C. M. Fuchs *et al.*, "Towards affordable fault-tolerant nanosatellite computing with commodity hardware," in *IEEE ATS*, 2018.

[47] M. Payer, "Too much PIE is bad for performance," *ETH Zurich Technical Report*, vol. 766, 2012.

[48] C. M. Fuchs *et al.*, "Bringing fault-tolerant gigahertz-computing to space," in *IEEE ATS*, 2017.

[49] S. Azimi, B. Du, and L. Sterpone, "On the prediction of radiation-induced SETs in flash-based FPGAs," *Elsevier Microelectronics Reliability*, 2016.

[50] H. Zhang *et al.*, "Aging resilience and fault tolerance in runtime reconfigurable architectures," *IEEE Transactions on Computers*, 2016.

[51] F. Siegle *et al.*, "Mitigation of radiation effects in SRAM-based FPGAs for space applications," *ACM Computing Surveys*, 2015.

[52] C. M. Fuchs *et al.*, "Enhancing nanosatellite dependability through autonomous chip-level debug capabilities," in *Springer ARCS*, 2016.

[53] ——, "Dynamic fault tolerance through resource pooling," in *NASA/ESA AHS*. IEEE, 2018.