Bringing Fault-Tolerant GigaHertz-Computing to Space

Bridging the Gap between Fault-Tolerance Theory and Spaceflight

Anonymous Author(s)

ABSTRACT

Modern embedded technology is a driving factor in satellite miniaturization, contributing to a massive boom in satellite launches and a rapidly evolving new space industry. Miniaturized satellites however suffer from low reliability, as traditional hardware-based fault-tolerance (FT) concepts are ineffective for on-board computers (OBCs) utilizing modern systems-on-a-chip (SoC), and larger satellites continue to rely on proven processors with large feature sizes. Software-based concepts have largely been ignored by the space industry as they were researched only in theory, and have not yet reached the level of maturity necessary for implementation. Therefore, we present the first integral, real-world solution to enable fault-tolerant general-purpose computing with modern multiprocessor-SoCs (MPSoCs) for spaceflight, thereby enabling their use in future high-priority space missions. In this contribution, we focus on the first stage of a multi-stage approach, offering short- and medium term FT using coarse-grained thread-level lockstepping with only a minimal performance degradation. We provide practical benchmark results indicating a beyond factor-of-5 performance increase over state-of-the-art radiation-hard OBC designs. On overview over the other stages which utilize FPGA reconfiguration and mixed criticality to extend the lifetime of spacecraft with aged or degraded OBCs is also provided. The presented approach was developed for a 4-year European Space Agency (ESA) project, and we are developing a tiled MPSoC prototype system together with two industrial partners.

1 INTRODUCTION

Modern embedded technology is a driving factor in satellite miniaturization, significantly contributing to a massive boom in satellite launches and a rapidly evolving new space industry. Especially 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 on-board computer (OBC) related electronics constitute a much larger share of a miniaturized satellite than they do in larger satellites, thus per component they must deliver drastically better performance and consume less energy. Therefore and due to cost considerations, miniaturized satellite OBCs are generally based upon processors with considerably finer feature size, such as those developed for mobile embedded devices. Traditional hardware-based fault-tolerance (FT) concepts for general-purpose computing are however ineffective for scaled technology node systems-on-chip (SoCs), becoming a prime source of malfunctions aboard miniaturized satellites [1, 2]. Larger satellites, too, are limited by the measures traditionally used to assure FT for space applications, as these prevent larger satellites from harnessing the benefits of modern processor- and multiprocessor-SoC (MPSoC) designs with a fine feature size. Also, these hardware-based FT-measures can not handle varying performance requirements during multi-phased missions and mega-constellations [3].

Software-based FT measures are effective for modern embedded hardware and have rapidly evolved due to efforts of the scientific community. However, these advances have largely been ignored by the space industry as they were researched only mathematically or in theory, but rarely intended for implementation. While many of these concepts include innovative ideas, they leave major implementation obstacles unaddressed, and often make impractical assumptions towards the platform or application environment. To the best of our knowledge, research has not yielded an integral and practical solution to FT in modern MPSoC based systems, which enables their use in high-priority space missions. Prior work often covers fault mitigation, but ignores fault detection, recovery from fail-over, or real-world constraints. Also, most concepts attempt to uphold safety and availability, e.g. for atmospheric aerospace use, but not computational correctness.

Today, there is a wide gap between academic research towards novel FT concepts and their practical application in spacecraft OBCs. Satellite command & data handling computers (CDH) 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. The presented research addresses these challenges and our main contributions are:

- to our best knowledge, the first practical, MPSoC based approach to FT general purpose computing for spaceflight use that operates within real-world constraints, leaving no conceptual gaps,
- is not based upon a custom or modified instruction set functionality or processor designs, and
- can be implemented in full with pre-existing standard platform toolchains, and minimal development effort;
- an in-depth analysis of the first stage of our multi-stage approach, which provides software-defined and controlled fault-detection and short- and medium term fault-coverage;
- performance measurements which show a low performance overhead and a beyond factor-of-5 performance increase over the current state-of-the-art in FT space-based computing,
- and an outline of the other stages which combine mixed criticality with reconfigurable logic to enable long-term fault coverage within a tiled MPSoC architecture.

We developed this approach for a 4-year European Space Agency (ESA) project, and designed an optimized, FPGA-based MPSoC architecture as platform for the presented approach with two industrial partners. Due to the interdisciplinary nature of this project, each stage, the MPSoC architecture, and the actual radiation-tested prototype OBC will be presented in separate publications.

In the next section, we will outline related work and its limitations which have resulted in the outlined gap between research and space computing. Section 3 describes the challenges and constraints of the space environment in which our approach must guarantee FT operation, our optimized MPSoC architecture, as well as terminology. Section 4 - 7 contain a high-level overview of the multi-stage FT approach, and an in-depth analysis of the first stage. Section 8, contains benchmark results. Discussions on aspects beyond the scope of this paper, a brief outlook on the prototype implementation schedule, and conclusions are given in Sections 9 and 10.

2 RELATED WORK

Radiation-hardened processors for space applications are based on proven manufacturing processes with a large feature size and hardware-FT, at the cost of efficiency and energy consumption. Traditionally, dual- and triple-modular redundancy (DMR and TMR) is implemented at the circuit- [4], RTL- and pipeline- [5, 6, 7, 8], core- [9, 10], and OBC-levels. Especially at the RTL- and circuit-level, the required voter-logic can not adequately report faults and instead suppresses them, making it difficult or impossible to assess the health-state of such a component. Thus software is unaware that it is being executed redundantly and takes no active part in fault-mitigation except for monitoring fault-counters and status registers. At core level and above, vast arrays of synchronized high-frequency voters are necessary to implement such solutions, and these are impractical beyond few hundred MegaHertz [9]. At GigaHertz clock frequencies, core lock-stepping has thus only been implemented with specialized processor cores and costly, custom IP-based SoCs [9]. The space industry has considerable experience designing FT solutions this way, but it results in single-vendor walled-garden solutions which proved to be costly to develop, maintain, and validate, as well as slow to evolve. Hence, RTL- and pipeline-level voting are only effective for comparably simple microcontroller designs, whereas coarser voting levels are costly and infeasible aboard miniaturized satellites. FT MPSoCs for space use are still largely at an experimental stage [11, 12, 13], and the only commercial implementations contain retrofitted single-core processors [6, 14]. To avoid the above mentioned technical constraints, our approach does not rely on hardware-TMR, but instead utilizes software-side functionality.

Coarse-grained lockstepping can be a powerful software-side FT measure, if applied to space computing. First practical non-space applications are promising but at an exploratory stage [15], or do not address the specific challenges to FT in space [16]. Unfortunately, most implement checkpoint & rollback or restart approaches, which makes their use in CDH applications difficult or impossible [17]. Most research which does not rollback faults is at an early theoretical stage [18], or entails an extreme performance-, or resource overhead [19]. Outstanding, promising concepts such as [15] still impose unreasonable design constraints on software and the operating system (OS), and ultimately do not implement software-side but only hardware-assisted, software-controlled FT. Another notable exception is COLO [16] which can only assure availability, but requires no modifications to the application software, lives entirely in software and requires little custom code. Thus, we consider coarse-grained lockstepping the way forward for space-based FT, but our approach does not utilize it to roll back operations and requires modifications only in two, distinct OS components. We allow the application developer to specify comparison points and utilize coarse-grained lockstepping to implement forward error-correction, utilizing known-correct results from a majority decision for recovery.

Other software-based FT approaches have advanced rapidly over the past decades. Though even the most complete FT concepts exist only in theory [20, 21, 22, 23], and do not address fundamental practical issues [24, 25, 26, 27, 28] such as fault detection [23]. While these contributions do not provide practical or implementable approaches, they do show how issues such as real-time scheduling in mixed critical software-FT systems can be solved efficiently [22, 29]. Some works also address peripheral concerns when deploying software-side FT, e.g. performance estimation [30], or QoS [31]. None of this prior research offers a single, viable and practical solution to assure FT of modern low feature-size MPSoCs, but each publication does treat individual issues which must be addressed if software-side FT is used. As such, our approach incorporates several of these ideas to assure scalability, compliance with timing-constraints, and enable task migration in mixed critical systems.

3 PLATFORM AND ENVIRONMENT

Here, we describe the application environment and platform architecture for our multi-stage FT approach. The threat-scenario for MPSoCs in satellite OBCs is drastically different than in most other application fields for embedded FT. Even in atmospheric aerospace, FT primarily implies fail-over and availability instead of computational correctness. Thus, the outlined gap between theory and application can in part be attributed to unawareness of the challenges to FT in space. We briefly present the main challenges to OBC dependability and conceptional constraints below. Then we will provide an overview of the architecture we have developed as a platform for our approach, and finally introduce the terminology used throughout the rest of the paper.

3.1 The Space Environment

Scientific and commercial space missions in the past often required large, custom designed spacecraft to accommodate the individual mission payload and maximize lifetime. Due to innovations in chip and electronics design, material science, and cost considerations, many modern instruments have modest performance requirements. These can be better satisfied by cheaper, less complex and (comparably) quick to build miniaturized satellites, for which pre-flight testing and the actual launch is possible at a fraction of the costs of a larger satellite. OBCs for such satellites are tightly constrained in size and mass, limiting design flexibility and minimizing slack-space. Due to the emergence of formation flying and mega constellations consisting of hundreds of satellites, the reliability constraints of more affordable modern embedded hardware have also become critical for the space industry.

Solar cells are the main power source aboard modern spacecraft. The 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. The average sustainable power consumption over time (power-budget) aboard a satellite is therefore very limited, and engineers are pressed hard to minimize each individual subsystem's energy consumption. Miniaturized satellite OBCs are usually limited to a few Watts of power-budget, thus most FT approaches for ground application are not viable for spacecraft.

Mid-mission physical access to a spacecraft is impossible, and historically satellite servicing missions were conducted only on rare occasions for satellites of outstanding importance in low-Earth orbit (LEO). Also, signal-travel times, limited communication windows, and scarce bandwidth make live-interaction with a spacecraft impractical. Thus, faults detected during a satellite mission must be resolved unattended, remotely, and fully autonomously.

About 20% of all anomalies [32] aboard satellites can directly be attributed to high-energy particles, and they are the predominant cause for faults within OBCs. These particles travel along 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. In LEO, the residual atmosphere and the Earth's magnetic field provides some protection from radiation, but this absorption effect diminishes quickly with altitude. Therefore, an OBC will be penetrated by a mix of highly charged particles, with flux density depending on solar activity and the spacecraft's attitude. They can induce a variety of electrical phenomena in OBC components and supporting electronics:

- Single Event Effects (SEE) - particle deposited electrical charges and local ionization can result in incorrect logical operations, bit-flips within data-storage cells and connecting circuitry.
- Displacement Damage (DD) - very high energy particles can also induce permanent lattice displacement within a chip's crystalline components, hence processor logic.
- Total Ionizing Dose (TID), is a cumulative charge trapping effect in the oxide of electronic devices, eventually resulting in a spontaneous electric discharge between components, which can be mitigated through power-cycling.

The impact of these effects on different microfabrication processes, substrates, and memory technologies varies. In general, electronics with a large feature size are more resilient to radiation effects than those manufactured in finer production nodes. Highly scaled chips are susceptible to multi-event upsets, SEEs propagating within circuits corrupting entire memory blocks or larger circuits. The increased impact of radiation effects on finer feature size chips also prevents better protection through more circuit-level protection.

Radiation events can also cause Single Event Functional Interrupts (SEFIs), affecting sets of circuits, individual interfaces, or even entire chips. In general, the effects of SEEs and SEFIs can be transient, while DD is permanent and accumulative [33]. Physical shielding can reduce certain radiation effects, but is infeasible even aboard medium sized spacecraft due to mass constraints.

3.2 Platform Architecture

As depicted in Figure 1, our MPSoC implements an asymmetric tiled architecture. We developed this architecture to be an ideal

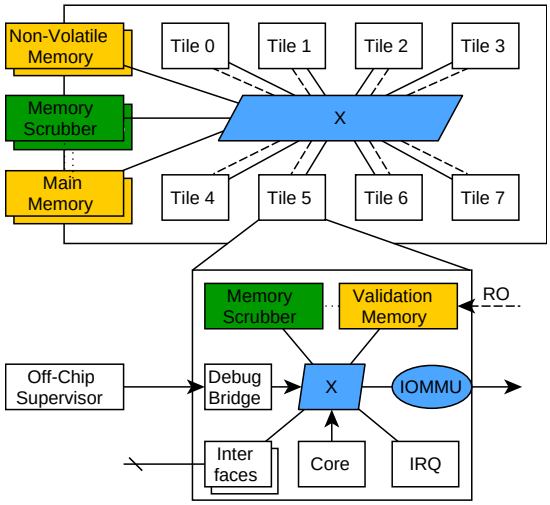


Figure 1: A simplified representation of our 8-core tiled MPSoC architecture with memory controllers highlighted in yellow, memory scrubbers in green, and the interconnect in blue. A dedicated interface on each tile allows the external supervisor access to the local interconnect.

platform for our approach and designed it for FPGA use. However, the architecture and our approach can also be realized on an ASIC, and one of our industrial partners will focus on an ASIC based implementation. Each tile is equipped with a processor core, an interrupt controller (IRQ in the figure), a dedicated on-chip memory slice, and several peripheral interfaces through the local interconnect. We refer to this dedicated on-chip memory slice as *validation memory*. Tiles 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 tiles to prevent interference and minimize shared logic.

The main memory is split into several segments: each tile has write-access to its own segment, and can read the global shared code segment. Each tile's main memory segment, validation memory, and its interfaces are mapped to the same tile-local address ranges. At the thread-level, the address-space in each tile is identical. The application and OS code is thus tile independent, allowing all tiles to share the same code segment. Data integrity in the architecture is assured using several error correction measures:

- To reduce the strain on the memory subsystem and increase overall system performance, processor cores are equipped with caches. As cached data has an extremely short lifetime, SECDED codes (error-correcting codes - ECC) offer sufficient protection from radiation induced upsets.
- The dual-port validation memory in each tile holds the current tile-status (active, needs update ,...), as well as the thread-checksums and state information. As this data is critical to the system's reliability, we utilize stronger Reed-Solomon based ECC, as this erasure coding system has shown superior performance in protecting highly-scaled SDRAM memory compared to SECDED codes [34]. ECC-fault syndrome interrupts are handled by each tile's processor. One interface is connected to the tile's local interconnect, while the second port is read-only accessible

via the global interconnect. The validation memory is inherently redundant, as threads are executed on at least two tiles.
- The shared main memory is redundant. Both instances are connected to the global interconnect, and ECC protected like the validation memory. ECC-fault syndrome interrupts for main memory are handled by the supervisor.
- Non-volatile memory is implemented redundantly as well. Our full prototype is designed to utilize MRAM [35] and PCM [36], both inherently immune to gradual data degradation caused by radiation. However, data can still be corrupted due to software issues and faults in the interface logic. Several solutions to address these issues have been developed, which usually are implemented at the block- or filesystem level and rely upon a system of checksums and composite erasure coding [37, 38, 39]. Through these solutions we can enable efficient and computationally inexpensive FT data storage for application data and program code even during long-term missions.

All these memories are susceptible to SEFIs and DD, but such faults are covered through redundancy. We perform error-scrubbing to avoid accumulating bit-flips in rarely used memory.

Faults detected by our approach are relayed to an off-chip supervisor, which is connected to each tile. The supervisor can then restore a tile's state or replace faulty tiles. In modern MPSoCs where processor cores run at GigaHertz clock frequencies, supervision is non-trivial as radiation-hard microcontrollers can not reach such high speeds. However, the first stage of our approach enables a reduction of the necessary lockstepping frequencies far below the KiloHertz range. Thus, high-performance MPSoCs utilizing our approach can very well be controlled using COTS supervisors. Hence, we can utilize a pre-existing, proven low-performance radiation-hardened microcontroller as supervisor.

Supervisor access to memory and other devices is possible via a MUXed AXI bridge in each tile to reduce the necessary pin count. To prevent malfunctioning tiles from interfering with the shared memories, the supervisor can detach tiles from the global interconnect by disabling their IOMMU. The supervisor also controls the other stages of our approach.

3.3 Application Model and Terminology

Fault detection in our approach is based upon sets of tiles 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 tiles, and are then referred to as a *tile group*. A tile group periodically executes a *checkpoint routine*, which computes checksums for all active threads and compares them with the other tiles in the group (*siblings*), thereby enabling a majority decision. Their relation is depicted in Figure 2.

A thread group can be executed on two tiles, enabling consistency checking as supervised DMR group. For many space applications this is sufficient, as they will first attempt to switch to a secondary CDH instance. Three tiles in a tile group assure computational correctness through TMR. More advanced setups are possible with 4 or more tiles, e.g. Byzantine agreement [40]. No alterations to the MPSoC or the software are required to support these advanced voting scenarios, as only the supervisor has to be aware of the actually used algorithm.

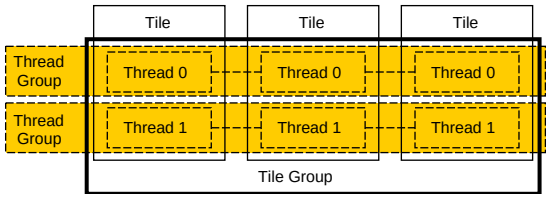


Figure 2: Timing-compatible thread groups (yellow) can be combined into tile groups, and multiple tile groups can co-exist in the same MPSoC.

A thread can provide the system with four callback routines, which are executed during tile initialization or by the checkpoint handler:

- an initialization routine, to be executed on all tiles at bootup, even on tiles where these threads are initially inactive.
- a checksum callback, used to generate a checksum for comparison with siblings,
- a synchronization callback, exposing all thread-state relevant data to synchronize a sibling with a tile group;
- and an update callback, which is executed on a tile that needs to synchronize its state to a tile group.

Besides the checksum generated by each checksum callback, the checkpoint handler also generates an additional checksum for global operating system metadata.

A lockstep cycle ends with a checkpoint, and the length of a cycle (the *checkpoint frequency*) is defined by the threads in a tile group. It can be modified at runtime, and must be updated if a thread is added or removed from a group. Schedulability, timing conformity, and deadlock-avoidance have been extensively researched in literature, e.g. in [22].

Our approach is designed for generic COTS MPSoCs, as these are readily available in a variety of performance classes at low cost. This is particularly important for miniaturized satellite applications, as these spacecraft are usually equipped with only one mobile-market MPSoC, for CDH and payload processing. The tiled architecture described above is thus optional but we consider it an ideal platform for this approach. In MPSoCs without a tiled architecture, the word *tile* can be substituted and read as processor core.

4 BRIDGING THE GAP: OUR APPROACH

To the best of our knowledge, this is the first integral and practical approach which can enable the usage of modern MPSoCs within an environment as hostile and isolated as space, and is not dependent on custom processor designs or circuit level FT. Our multi-stage FT approach provides the fault-coverage necessary for current and future CDH applications, while also offering sufficient performance and scalability to handle challenging payload tasks. This approach consists of three fault-mitigation stages:

- The fist stage is implemented entirely in software and provides fault-detection through coarse-grained lockstepping. The necessary code can be implemented in commercial off the shelf (CTOS) MPSoCs.
- The second stage improves medium-term reliability, and enables long-term fault-coverage through FPGA reconfiguration and the use of alternative configuration variants.

- The third stage extends the lifetime of a degraded OBC beyond the point where it would usually become unusable by utilizing mixed criticality.

Our approach delivers high performance and strong fault-coverage for modern MPSoCs which up until now are considered unsuitable for high-priority space missions due to their low reliability.

The first fault-mitigation stage is implemented in the scheduler, and lockstepping is instruction set and processor design agnostic. It offers software-controlled, thread-level, distributed majority voting and fine-grained fault logging. The checkpoint frequency necessary to assure a desired level of fault-coverage can be adjusted at runtime. The supervisor only receives information about agreements between tiles from each processor's perspective, without centrally assessing the system state itself. Instead of exerting direct control over the MPSoC, it can assure FT indirectly, as fault-coverage and control are distributed and enforced by the tiles themselves. In consequence, the supervisor does not require any knowledge about the executed application threads, an individual tile's state, or other OBC intrinsics. The thread group assignment within an MPSoC can be reconfigured freely at runtime to implement different voting configurations, while sufficient intact tiles are available. Thus, the described approach can exploit parallelization to improve reliability, throughput, or minimize power consumption. It can also take advantage of frequency-voltage scaling and various power management features to adapt to different operating conditions. Therefore, the system's FT guarantees and performance can be adjusted per thread to meet varying latency, energy consumption, throughput requirements or enable specific interfaces when needed, thereby allowing the system to adapt to multi-phased missions. This stage relies on the presence of spare tiles and tiles with spare processing capacity to replace faulty tiles.

The second stage is designed to recover faulty tiles by reprogramming corrupted SRAM- or flash memory components (BRAM and LUTs) and re-purposing permanently defective logic cells within an FPGA. As FPGA-fabric is inherently redundant, we can utilize it to improve the survivability of an OBC. Bit-flips can affect the content of logic cell LUTs, which can usually be repaired through reconfiguration [41, 42]. Even if a logic cell is damaged permanently, the residual FPGA-fabric will remain intact and can be re-purposed [43], reprogramming it with differently routed, functionally equivalent configurations. As efficient fault-detection at the FPGA-logic level is an unsolved issue, the second stage instead utilizes the extensive fault-detection capabilities of the first stage.

The third stage utilizes thread-level mixed criticality to stabilize the system and extend the OBC's lifetime, as the other stages can only offer fault coverage if enough healthy tiles are available. In strongly iradiated FPGAs, the system will thus eventually be unable to provide sufficient FT processing power for all applications. Performance degradation or even a loss of lower-criticality tasks aboard a satellite is in general preferable to a loss of system stability or degradation of the most important satellite control applications. Applications in a satellite's CDH system are thus clearly prioritized in criticality. This stage can uphold fault-coverage for high-criticality applications by sacrificing compute performance, increasing latency, or decreasing the checkpoint frequency of lower-criticality tasks to uphold reliability for high-criticality threads.

5 STAGE 1: SHORT-TERM FAULT HANDLING

The program flow of this stage is depicted in Figure 3, and can be implemented within the scheduler of an existing OS. In fact this figure is directly derived from our proof-of-concept implementation, and we will explain each element of the figure individually below. As depicted, the execution flow can be subdivided into three parts: tile and application initialization, checkpoint execution, and application processing. The only necessary modifications to an OS to implement this code are highlighted in blue. The additional checks during application processing are can be added to the scheduler, while the rest can be implemented as interrupt service routine (ISR). Thus, code additions are necessary only in two locations in the OS.

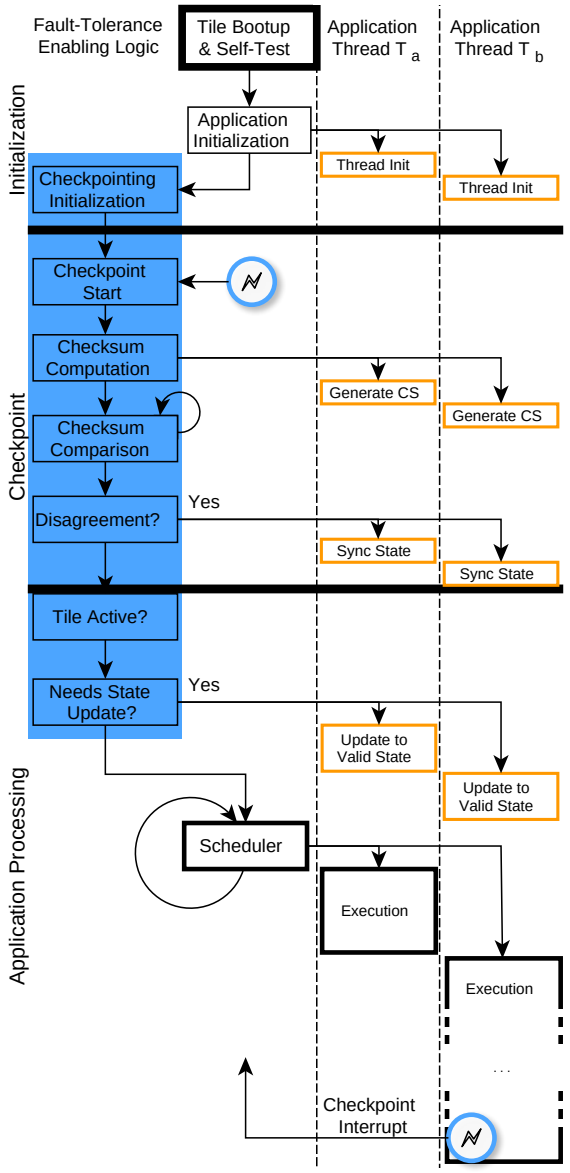


Figure 3: The execution cycle of a tile during the first stage of our multi-stage FT approach. All code necessary for implementation is highlighted in blue, callbacks in yellow.

Below, we will step through the depicted program flow and provide formal timing definitions for all recurring operations. The time required for an action is indicated as τ_{action}. As tile bootup and application initialization are one-time operations, they have no direct impact on the runtime performance of our approach. A practical example for tile fault handling and recovery is provided at the end of this section to illustrate the depicted logic's behaviour.

5.1 Bootup & Initialization

After bootup, a tile first executes basic self-test functionality to assure integrity of tile-local IP-cores, its interfaces, and the integrity of the validation memory. Each thread can provide an initialization routine to be executed when the OBC is booted, which is labelled as *Thread Init* in Figure 3 and denoted as T_{init} in Figure 4. Most applications will spend a significant amount of time with bootstrap and initialization. Re-executing these operations each time a new tile is added to its tile group would be inefficient and needlessly inflate an application's state data. Each thread's initialization routine is executed on all tiles, even on tiles where these threads are initially inactive. This behaviour increases the overall OBC memory consumption, but allows a drastic reduction of the time and data necessary for synchronization.

During application initialization, each thread will register its checkpoint frequency requirements and the callback routines which are needed in the subsequent steps. After these callbacks have been executed, the tile will set a periodic timer to initiate checkpoints. As depicted in Figure 3, each tile will execute its first checkpoint, if the entire MPSoC has just been booted to assure that application and OS initialization were successful. At this point, all lockstepping-relevant information has been initialized and stored in validation memory, and the scheduler can be left in control.

5.2 Checkpoint Start

A checkpoint can be induced by the described timer interrupt or by the supervisor. The checkpoint ISR will invoke the actual checkpoint routine, as the operations possible within an ISR are constrained in most popular instruction set architectures. The ISR is therefore only responsible for performing a context switch to system- or kernel mode and entering the checkpoint handler.

Threads can be executed in an arbitrary order within a lockstep cycle. However, interrupting an active application at a random point in time is usually undesirable, as outlined in [15]. We avoid this and the entailing thread-synchronization issues worked around in [15], as in our approach the application developer can define comparison points, where the application will yield control to the checkpoint handler. A thread can thus delay interrupt processing until it has reached a safe state for checksum comparison. Thereby introduced time variances are handled in Section 5.4.

5.3 Checksum Computation

The checkpoint handler will then iterate through all siblings' validation memories, and invoke each active thread's checksum callback, requiring τ_{CS} time to generate each checksum. As there is no efficient, uniform approach to assess the health of a thread, we chose to rely on the application to assess its own health-state or expose state-relevant information to the system for comparison. The generated checksums are re-computed upon each checkpoint, according to the thread group's lockstepping frequency. Their format and algorithm can be defined by the system designer to optimize for performance or fault-detection. Four options are offered for generating a checksum:

(1) The checkpoint handler by default invokes an application provided callback routine which returns a checksum. The time required to generate such a checksum can be minimized with slight modifications to the application's code, e.g. by retaining computational by-products which would usually be discarded. Thus, the application developer's cooperation is necessary, as the application code must be adapted to provide the necessary checking functionality. This approach can yield the strongest fault-detection guarantees and best performance.
(2) Sometimes, applications written for a spacecraft's OBC are proprietary and the vendor may be unable or unwilling to extend the functionality of their software. Thus, if no checkpoint routine was provided, a checksum is computed automatically for an application-defined memory segment. The application developer can place state-relevant data within this segment without altering the actual application logic, through linker scripts or pre-processor macros. The main limitation of this approach is the constrained space in validation memory that can be made available to an application.
(3) Alternatively, the system developer can also specify individual memory addresses of variables and data structures for byte-wise comparison, a viable approach if only few individual variables must be compared and source-code modifications are impossible.
(4) Non-continuously running applications can also deposit state-relevant data within a dedicated buffer and return a checksum when exiting. This option supports emulating algorithmic diversity and can efficiently handle sparse, time-triggered tasks.

Usually, only a subset n of all thread groups will be active on a tile, thus only $n+1$ checksums will be generated and stored in the tile's local validation memory (+1 as we generate an extra checksum for OS metadata). Each newly generated checksum will be stored in validation memory, and the time necessary to execute this part of the checkpoint is $\tau_{genCS} = \sum^{n+1} \left(\tau_{CS}\right)$.

5.4 Checksum Comparison

Once all checksum callbacks have been executed, a tile will monitor the other tile group members' validation memories for valid checksums until the system designer's global, tile-wide deadline passes ($\tau_{deadline}$). A tile will usually begin comparing its checksums ($\tau_{comparison} = (n+1) * \upsilon_{cmpCS}$) with other siblings early on, and only wait briefly ($\tau_{comparison} \leq \tau_{deadline}$). If a tile detects a checksum mismatch or a sibling did not provide checksums before the deadline expires, it will report disagreement with that tile to the supervisor and stop comparing checksums with this sibling. Technically, thread-level disagreement reporting is also possible, though it does not significantly improve voting reliability and restricts the system's scalability as it penalizes higher tile numbers. Thus, we chose to implement tile-level voting. Checksum comparison with a tile's siblings is thus $\tau_{vote} \leq \tau_{comparison} + \tau_{loadVM_n} + \tau_{cmpCS}$

It is important to note that the reliability of each individual tile's voting decision can be weak, and an individual tile can report false-agreement or disagreement with its siblings. Our approach takes this into account, and mitigates it through a distributed decision

based on each tile's perspective of it's siblings' correctness. As a tile group will usually consist of three or more tiles, and the likelihood of false-disagreements or non-reported disagreement is very low, as multiple concurrent faults would have to occur on several tiles within the same tile group during a single checking period. The probability for such an event is extremely low at all but exceptionally high radiation levels, and thus statistically negligible unless a tile group already consisted of tiles with permanent defects. As the second stage of our approach specifically addresses the issue of permanently defective tile-logic and its recovery, any remaining concern regarding this corner-case can be well addressed by expanding a tile group to additional tiles.

5.5 Thread Disagreement & State Propagation

If a tile detected disagreement with any of its siblings, it will execute the synchronization routines for all d threads in the affected tile group. These routines will copy all data necessary to synchronize a sibling within its tile group to validation memory. They can do so by either placing the relevant data directly in the tile's validation memory, or through references to data structures in the tile's main memory segment. Hence, an application requires no prior knowledge about another tile's memory contents, as it can utilize the provided references in validation memory. Like with checksum callbacks, the state synchronization routines should be provided by the application developer, and the time necessary for their executions is $\sum^{d} \tau_{sync}$. A faulty tile will execute these callbacks as well, as from its perspective the other tiles are faulty. This callback can thus be omitted, if all state-relevant data is already in validation memory. If a new thread group was added to the tile group, the checkpoint routine will update the checkpoint's timer and return control to the scheduler.

The supervisor will react to disagreement between tiles according to the system designer's recovery algorithm, but faults will actually be reported by each tile individually through the OS's logging facility (e.g. syslog, kernel-buffer, tile-local UART, ...). Diagnostics can thus be enriched with application-level information, which can drastically improve debugging and defect assessment accuracy. Thereby, we can provide the application developer with the functionality necessary to make better decisions regarding the spacecraft's health. Spacecraft operators and system designers no longer have to assess the consistency and correctness of an application's state, and can instead concentrate on finding an optimal FT strategy.

5.6 State Update and Thread Execution

When resuming processing, the scheduler will check three conditions: if the tile is member of a tile group and thus active, if it was newly added to a tile group, and which threads are active. Idle tiles will immediately enter into a sleep state and are woken up at the next checkpoint to reduce energy consumption and fault-potential.

In case the tile must update a thread's state from a sibling, the relevant application-provided update routine will be executed. The faulty tile or its replacement will utilize the data stored by the synchronization routine to update its own state to the tile group's known correct state. The other siblings will wait for this tile to update its state for an application developer or system designer defined grace period (τ_{grace}). Regardless if a tile had to synchronize with its siblings or wait for one, the time necessary for recovery is $\sum^{d} \tau_{update} \leq \tau_{grace}$.

Once the tile has updated its state using its sibling's data, its scheduler can continue to execute applications, and its siblings wake up and do the same. This concludes the lockstep cycle.

5.7 Stage 1 Summary

Figure 4 depicts a quad-core MPSoC running a single tile group on three tiles. In this example, a fault has occurred during the second lockstep cycle on tile C_2, which is subsequently replaced with the idle tile C_3. C_3 must then retrieve a copy of the state of its threads T_a and T_b from another valid sibling. The replaced tile can subsequently be rebooted to determine if it is permanently defective.

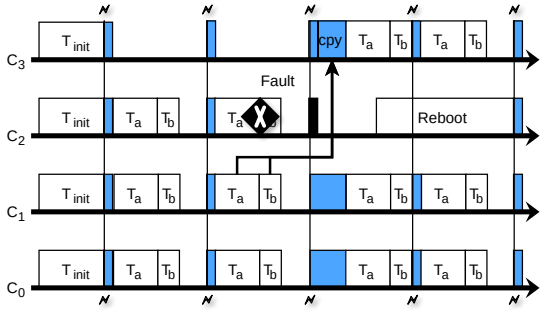


Figure 4: Tile initialization and a complete lockstep cycle in a quad-core MPSoC implementing the presented logic.

LEO operation will require checkpoint frequencies in the low Hertz range, and even in strongly irradiated environments below the Earth's magnetosphere 1-20 Hertz lockstepping will be sufficient for most applications [32]. Hence, the performance impact of our approach compared to unprotected application execution is low even in aged or degraded OBCs, and far below the constraints of traditional hardware-FT. The time necessary to execute one lockstep cycle including all described functionality can be formally defined as $\tau_{total} = \tau_{initCP} + \tau_{genCS} + \tau_{vote} + \sum^{d} \tau_{sync} + \sum^{d} \tau_{update}$.

6 STAGE 2: TILE REPAIR & RECOVERY

Over-provisioning of tiles naturally is inefficient and can curtail system scalability especially on FPGA based systems, where transient faults in the FPGA-fabric can corrupt the programmed logic. There, transients can induce permanent logic-level faults which can be repaired through reconfiguration. The second stage of our FT approach utilizes the fault-detection capabilities of our first stage, as no effecting and scalable approach to fault detection at the FPGA-fabric level beyond static TMR has been published to date. Our tiled architecture can especially benefit from partial reconfiguration, as tiles can be placed strategically on an FPGA's fabric within different partition borders. Details on the recovery of faulty tiles, as well as other features possible at this stage will be discussed in a separate publication.

7 STAGE 3: APPLIED MIXED CRITICALITY

This stage is designed to extend the lifetime of an OBC implementing our approach, in case the previous stages have depleted all idle

tiles and insufficient spare processing capacity to migrate a tile group is available in the MPSoC. Aboard a satellite, upholding FT guarantees for a minimal set of mission critical control applications is usually preferable to running a larger set of applications on an unreliable system. The criticality of applications executed on an OBC can be differentiated by the importance of the controlled subsystems. Also, thread groups can be added and removed from tile groups, and multiple tile groups can coexist in the same MPSoC. Hence, threads can also be migrated between tile groups [23]. We can then utilize mixed criticality to split degraded tile groups and evacuate high-criticality threads, thereby maintaining strong FT guarantees for high-criticality applications even if an OBC approaches the end of its lifetime. The operators can then define a more resource conserving schedule for all the tasks executed on a satellite without rewriting parts of the satellite's software, the traditional approach to solving such issues. They can then, for example, reduce on-orbit data pre-processing, and sacrifice link capacity or on-board storage space for system stability.

In practice, if a fault was detected and no further spare processing resources are available, other tile groups may be able to execute individual thread groups of a degraded tile group. The remaining, lower-criticality thread groups can then continue execution in a degraded tile group or be suspended. Thereby, this third FT stage can extend an OBC's lifetime and guarantee fault-coverage for high criticality threads, even if insufficient healthy tiles are available within a system and no tiles could be repaired by the second stage. Thus, it also relies upon the fault-detection functionality provided by the first stage to enable the practical use of mixed criticality in a strongly resource constrained environment.

In the example depicted in Figure 5, initially two tile groups are executed on one MPSoC with 6 tiles. The green tile group consisting of a computationally expensive low-criticality payload data processing application T_d and a shorter but more critical payload control thread T_c. The white group consists of mission critical CDH tasks. Such setups are extremely common in many space applications and inevitable aboard miniaturized satellites with only one OBC. Tiles 0, 1 and 2 would still have sufficient spare capacity to accommodate T_c, but not T_d (Tiles 0 and 1 are not depicted for simplicity). A fault occurred in tile 5 and there are no idle tiles available. The lower-criticality task T_d thus remains in a degraded tile group on tiles 3 and 4, which can only detect but not correct future faults. T_c is migrated to a separate, new tile group and executed on tiles 2, 3 and 4, maintaining computational correctness for all high-criticality applications. In our example we omitted that the checkpoint frequencies must be adapted to accommodate the timing requirements for the CDH group. The program code necessary to do so is not particularly complex, and only the timing properties for the new tile group and the defunct one must be updated.

8 PERFORMANCE ESTIMATION

To achieve worst-case performance estimations, we developed a naive and unoptimized implementation of the first stage of our approach in C. 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 tiles

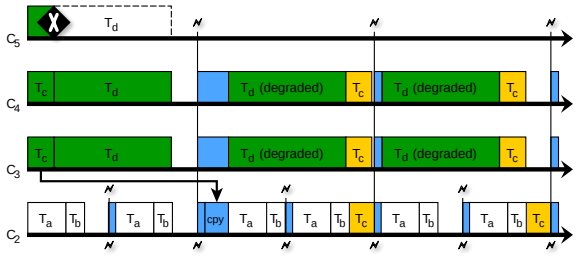


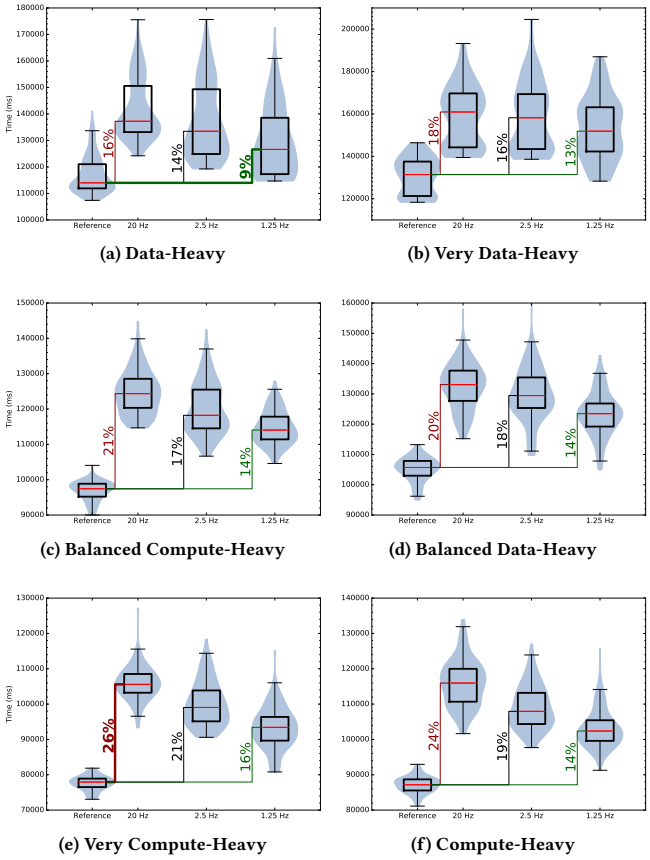
Figure 5: If no additional intact tiles are available, the third stage of our approach can split defunct tile groups and uphold FT guarantees for high-criticality threads.

and thread management. Thread-management at this level is computationally expensive, and is drastically faster in a kernel-level MPSoC based implementation. Besides enabling very pessimistic benchmarking, this 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. Also, this allows to assess the performance of our approach pessimistically without requiring a full prototype in hardware, as simulating such a large FPGA design is only viable for debugging purposes but not for benchmarking.

The provided benchmark results were generated based on code derived off a special CCD readout program used for space-based astronomical instrumentation. Its specifications and program flow is based on the NASA/James Webb Space Telescope's Mid-Infrared Instrument (MIRI) described in [44]. This program continuously reads three 16-bit false-color sensor arrays and stores the results in a buffer. It then averages multiple captured frames to optimize the instruments exposure time and avoid saturated pixels or capture faint astronomical sources [44]. For each plot, 100 measurements were taken of the real-time necessary to process 600 1-Megapixel frames with subsequent processing runs. 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. Data heavy modes indicate a high amount of post-processing runs, whereas compute-heavy modes indicate lower per-thread workload. Benchmark results were generated on a Intel Core I7 Sandy Bridge-based system with a host kernel's scheduling frequency of 1kHz (`CONFIG_HZ_1000`). Binaries compiled with `GCC 6.3.1 (20161221)` without compiler optimization (`-O0`).

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 6a 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 quality of the measurements taken. As expected, the performance varies depending on workload, with data-heavy tasks a-c showing better performance. This too was expected as the first stage's code consists mainly of function calls, integer operations, binary comparisons, and jumps. Drastically better performance can be expected in a more optimized implementation at the kernel level. 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%, our approach can thus allow a modern MPSoC to perform drastically better than comparable state-of-the-art solutions, while requiring no proprietary processor design, offering full software-control at a fraction of the development effort and costs.



(a) Data-Heavy (b) Very Data-Heavy

(c) Balanced Compute-Heavy (d) Balanced Data-Heavy

(e) Very Compute-Heavy (f) Compute-Heavy

Figure 6: Performance measurements for processing 600 frames with different checking frequencies and workloads.

9 DISCUSSION & OUTLOOK

In the current design, the checkpoint handler must assure the integrity of kernel structures and collect relevant OS metrics directly. This is a workaround, as error detection for operating system metadata is not available in any OS to date, and currently being developed [45]. The approach outlined in [46] could automatically generate EDAC statistics for validation during an OS's execution, and would therefore assess the health-state of the OS better and faster.

Our approach assures consistency for each lockstep cycle, but does not guarantee consistency of data emitted by each member of a tile group. However, peripheral output consistency is a non-issue aboard all but the smallest miniaturized satellites. OBC interfaces in most spacecraft are implemented redundantly at great effort for FT reasons, and our architecture inherently provides this redundancy already. Only for very small nanosatellites, interface redundancy is often impossible due to mass and PCB-space constraints. In such OBCs, I/O voting can be implemented for peripheral interfaces.

Over the past years, several publications on virtualization-based FT concepts were published, though these results are at an early stage [47, 48]. The miniaturized satellite community, too, is beginning to discover the advantages of virtualization for fault-tolerance, but relevant research projects have just been initiated [49]. As our approach is implemented in the OS scheduler, it would also be possible to implement it within a hypervisor. An implementation within a hypervisor would enable the described functionality at the virtual machine-level instead of for threads, thereby enabling an entirely new application area. Most functionality necessary to implement our approach at that level is similar or identical to the logic described in this paper.

Due to the basic design of our approach, the supervisor can also handle MPSoCs distributed across multiple FPGAs or ASICs. A multi-chip setup can drastically increase an OBCs resistance against SEFIs, permanent device failure, and improve scalability. It allows considerably higher tile counts and tighter real-time guarantees, if needed. In such an OBC, one instance of each redundant memory controller can be placed on each chip, which enables it to tolerate even complete chip-level failure. It would also allow one half of an FPGA-based OBC to continue processing unaffected while the other chip was reprogrammed, thereby enabling seamless operation during a full FPGA-reconfiguration. We are aware that additional logic is necessary for such MPSoC designs, and the global interconnect would have to be split between multiple chips. A three-level interconnect (tile-local, chip-local and global) would also be advisable due to overall system performance considerations. Of course, multi-chip MPSoCs require more PCB space and energy, which makes them suitable for larger satellites.

We are currently porting the presented FT stage to the ARM platform, as our MPSoC is based upon ARM Cortex-A/R cores. We plan to make this port available to the public under an open license at a later point during the project. Once the port has been completed we will gradually improve system performance, increase tile count, and reduce tile group volatility. In the third quarter of the project, we will migrate our MPSoC from an FPGA development board to prototype hardware with miniaturized satellite and space-industry relevant interfaces. Once this prototype hardware is available, it will undergo radiation testing to assess its performance in a space-like environment.

10 CONCLUSIONS

In this contribution, we presented to our knowledge the first practical and integral multi-stage approach to fault-tolerant general purpose computing for spaceflight use, which operates within real-world constraints and does not leave conceptual gaps. We showed that our approach is programmatically simple and requires a minimal amount of custom code, which can also be implemented in most pre-existing multi-threading capable operating systems. 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. In 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 fault-tolerant 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 can uphold 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 tile) as well as horizontally (more tiles), with virtually any modern processor core. Thereby, we aim to increase acceptance for these measures in the space industry, building trust in hybrid HW-SW architectures and present an example to develop realistic FT concepts for space applications. Thus, our approach is the first integral, real-world solution to enabling fault-tolerance with modern MPSoC designs, thereby enabling the use of such architectures in future high-priority space mission.

REFERENCES

[1] Martin Langer and Jasper Bouwmeester. Reliability of cubesats-statistical data, developers' beliefs and the way forward. In *AIAA/USU Conference on Small Satellites*, 2016.

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

[3] Benjamin Bastida Virgili and Holger Krag. Mega-constellations issues. In *41st COSPAR Scientific Assembly*, 2016.

[4] Carl Carmichael. Triple module redundancy design techniques for Virtex FPGAs. *Xilinx Application Note XAPP197*, 2001.

[5] Kevin Reick et al. Fault-tolerant design of the ibm power6 microprocessor. *IEEE micro*, 2008.

[6] Magnus Hijorth et al. GR740: Rad-hard quad-core LEON4FT system-on-chip. In *DAta Systems in Aerospace (DASIA)*. ASD-Eurospace, 2015.

[7] Andrew S Jackson. *Implementation of the configurable fault tolerant system experiment on NPSAT-1*. PhD thesis, Naval Postgraduate School Monterey, 2016.

[8] STMicroelectronics. *RPC56EL60L5: 32-bit Power Architecture microcontroller for Aerospace and Defense*, 2015.

[9] Xabier Iturbe, Balaji Venu, Emre Ozer, and Shidhartha Das. A triple core lock-step (TCLS) ARM Cortex-R5 processor for safety-critical and ultra-reliable applications. In *Dependable Systems and Networks Workshop*. IEEE, 2016.

[10] Daniel Ludtke et al. OBC-NG: towards a reconfigurable on-board computing architecture for spacecraft. In *IEEE Aerospace Conference*, 2014.

[11] Charles A Hulme et al. Configurable fault-tolerant processor (CFTP) for spacecraft onboard processing. In *IEEE Aerospace Conference*, 2004.

[12] John R Samson. Implementation of a dependable multiprocessor cubesat. In *IEEE Aerospace Conference*, 2011.

[13] Sukrat Gupta, Neel Gala, GS Madhusudan, and V Kamakoti. SHAKTI-F: A fault tolerant microprocessor architecture. In *IEEE Asian Test Symposium (ATS)*, 2015.

[14] Xabier Iturbe et al. On the use of system-on-chip technology in next-generation instruments avionics for space exploration. In *Very Large Scale Integration-System on a Chip (VLSI-SoC)*. Springer, 2015.

[15] Uli Kretzschmar et al. Synchronization of faulty processors in coarse-grained TMR protected partially reconfigurable FPGA designs. *Reliability Engineering & System Safety*, 2016.

[16] YaoZu Dong et al. COLO: Coarse-grained lock-stepping virtual machines for non-stop service. In *Symposium on Cloud Computing*. ACM, 2013.

[17] Joshua Hursey, Jeffrey M Squyres, Timothy I Mattox, and Andrew Lumsdaine. The design and implementation of checkpoint/restart process fault tolerance for open MPI. In *Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 2007.

[18] Sebastian Weis. *Fault-Tolerant Coarse-Grained Data-Flow Execution*. PhD thesis, Dissertation, Augsburg, Universität Augsburg, 2015, 2016.

[19] Brendan Cully et al. Remus: High availability via asynchronous virtual machine replication. In *Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation*. San Francisco, 2008.

[20] Sunondo Ghosh, Rami Melhem, and Daniel Mossé. Fault-tolerance through scheduling of aperiodic tasks in hard real-time multiprocessor systems. *Transactions on Parallel and Distributed Systems*, 1997.

[21] Zaid Al-bayati, Brett H Meyer, and Haibo Zeng. Fault-tolerant scheduling of multicore mixed-criticality systems under permanent failures. In *Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFT)*. IEEE, 2016.

[22] Yu Liu, Han Liang, and Kaijie Wu. Scheduling for energy efficiency and fault tolerance in hard real-time systems. In *Design, Automation and Test in Europe (DATE)*, 2010.

[23] Luyuan Zeng, Pengcheng Huang, and Lothar Thiele. Towards the design of fault-tolerant mixed-criticality systems on multicores. In *Compilers, Architectures and Synthesis for Embedded Systems (CASES)*. ACM, 2016.

[24] Peter Munk et al. Toward a fault-tolerance framework for COTS many-core systems. In *European Dependable Computing Conference (EDCC)*. IEEE, 2015.

[25] Andrea Höller et al. Software-based fault recovery via adaptive diversity for cots multi-core processors. *unpublished, arXiv preprint 1511.03528*, 2015.

[26] Siavoosh Payandeh Azad et al. Holistic approach for fault-tolerant network-on-chip based many-core systems. *unpublished, arXiv preprint 1601.07089*, 2016.

[27] Anderson L Sartor et al. Exploiting idle hardware to provide low overhead fault tolerance for VLIW processors. *ACM Journal on Emerging Technologies in Computing Systems (JETC)*, 2017.

[28] Fakhar Anjam and Stephan Wong. Configurable fault-tolerance for a configurable VLIW processor. In *International Symposium on Applied Reconfigurable Computing*. Springer, 2013.

[29] Sheheryar Malik and Fabrice Huet. Adaptive fault tolerance in real time cloud computing. In *World Congress on Services*. IEEE, 2011.

[30] Kamel Smiri, Safa Bekri, and Habib Smei. Fault-tolerant in embedded systems (MPSoC): Performance estimation and dynamic migration tasks. In *Design & Test Symposium (IDT), 2016 11th International*. IEEE, 2016.

[31] Zaid Al-bayati, Jonah Caplan, Brett H Meyer, and Haibo Zeng. A four-mode model for efficient fault-tolerant mixed-criticality systems. In *Design, Automation & Test in Europe (DATE), 2016*. IEEE, 2016.

[32] S. Bourdarie and M. Xapsos. The Near-Earth Space Radiation Environment. *IEEE Transactions on Nuclear Science*, 2008.

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

[34] European Cooperation for Space Standardization. ECSS-Q-ST-30C - dependability, March 2017.

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

[36] J. Maimon et al. Results of radiation effects on a chalcogenide non-volatile memory array. In *IEEE Aerospace Conference*, 2004.

[37] Christian M Fuchs, Martin Langer, and Carsten Trinitis. FTRFS: A fault-tolerant radiation-robust filesystem for space use. In *Architecture of Computing Systems (ARCS)*. Springer, 2015.

[38] S. Zertal. A reliability enhancing mechanism for a large flash embedded satellite storage system. In *IEEE ICONS*, 2008.

[39] Alexandre Peixoto Ferreira et al. Using pcm in next-generation embedded space applications. In *Real-Time and Embedded Technology and Applications Symposium (RTAS)*. IEEE, 2010.

[40] Shreya Agrawal and Khuzaima Daudjee. A performance comparison of algorithms for byzantine agreement in distributed systems. In *European Dependable Computing Conference (EDCC)*. IEEE, 2016.

[41] Sarah Azimi, Boyang Du, and Luca Sterpone. On the prediction of radiation-induced SETs in flash-based FPGAs. *MICROELECTRONICS RELIABILITY*, 2016.

[42] Cinzia Bernardeschi, Luca Cassano, Andrea Domenici, and Luca Sterpone. UA2TPG: An untestability analyzer and test pattern generator for SEUs in the configuration memory of SRAM-based FPGAs. *Integration*, 2016.

[43] Felix Siegle, Tanya Vladimirova, Jørgen Ilstad, and Omar Emam. Mitigation of radiation effects in SRAM-based FPGAs for space applications. *ACM Computing Surveys (CSUR)*, 2015.

[44] ME Ressler et al. The Mid-Infrared instrument for the James Webb Space Telescope: The miri focal plane system. *Astronomical Society of the Pacific*, 2015.

[45] A.D. Velasco, B. Montruccio, and M. Rebaudengo. A hardening approach for the schedulerfis kernel data structures. In *1st Workshop on Computer Architectures in Space (CompSpace) at ARCS2017*, 2017.

[46] Tobias Stumpf. How to protect the protector? In *Dependable Systems and Networks (DSN 2015) - Student Forum*. IEEE Computer Society, June 2015.

[47] Frederico Cerveira et al. Recovery for virtualized environments. In *European Dependable Computing Conference (EDCC)*. IEEE, 2015.

[48] Ganesh Venkitachalam et al. Virtual machine fault tolerance, June 12 2012. US Patent 8,201,169.

[49] Andrew D Santangelo. An open source space hypervisor for small satellites. In *AIAA SPACE 2013 Conference*, 2013.