Fault-Tolerant Nanosatellite Computing on a Budget

Christian M. Fuchs, *Member, IEEE,* Nadia M. Murillo, Aske Plaat, Erik van der Kouwe, Daniel Harsono, and Todor P. Stefanov, *Member, IEEE*

***Abstract*—We present an on-board computer architecture designed for small satellites (<50kg), which exploits software-fault-tolerance to achieve strong fault coverage with commodity hardware. Micro- and nanosatellites have become popular platforms for a variety of commercial and scientific applications, but today are considered suitable mainly for short and low-priority space missions due to their low reliability. In part, this can be attributed to their reliance upon cheap, low-feature size, COTS components originally designed for embedded and mobile-market applications, for which traditional hardware-voting concepts are ineffective. Software-fault-tolerance has been shown to be effective for such systems, but have largely been ignored by the space industry due to low maturity, as most have only been researched in theory. In practice, designers of payload instruments and miniaturized satellites are usually forced to sacrifice reliability in favor of delivering the level of performance necessary for cutting-edge science and innovative commercial applications. Thus, we developed a set of software measures facilitating fault tolerance based upon thread-level coarse-grain lockstep, which we validated through fault-injection. To offer strong long-term fault coverage, our architecture is implemented as tiled MPSoC on an FPGA, utilizing partial reconfiguration, as well as mixed criticality. This architecture can satisfy the high performance requirements of current and future scientific and commercial space missions at very low cost, while offering the strong fault-coverage guarantees necessary for platform control even for missions with a long duration. This architecture was developed for a 4-year ESA project. Together with two industrial partners, we are developing a prototype to then undergo radiation testing.**

***Index Terms*—CubeSat, SmallSat, Nanosatellite, Satellite, System-on-chip, RTOS, FPGA, ARM, Cortex-A53, Microblaze, Xilinx, COTS, partial reconfiguration, forward error correction, fault tolerant systems, fault tolerance, integrated circuit reliability, fault injection, reliability, robustness, software defined fault tolerance**

I. Introduction

Satellite miniaturization has enabled a broad variety of scientific and commercial space missions, which previously were technically infeasible, impractical or simply uneconomical. However, due to their low reliability, nanosatellites, as well as light microsatellites, are typically not considered suitable for critical and complex multi-phased missions and high-priority science. The on-board computer (OBC) and related electronics constitute a large part of such spacecraft, and were shown to be responsible for a significant share of post-deployment failure [1]. Indeed, these components often lack even basic fault tolerance (FT) capabilities.

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

In the next section, we describe our architecture's intended application, design constraints, and the physical fault model encountered. An overview of our multi-stage FT architecture and our coarse-grain lockstep approach is provided in Section III. In Section IV we outline a publicly reproducible variant of our architecture's MPSoC design based on Xilinx Microblaze processor cores. Our architecture utilizes software defined FT, and Section V-E contains validation results of the coarse-grain lock-step approach, which were obtained using fault injection. We begin by discussing the available test techniques, and then describe our test setup, the target application as well as the types of fault injection. In order to place our results into context, we compare them to the Dobel at al. [2] test-campaign, the only other openly published practical test-campaign conducted with a comparable scope and for software-FT mechanics. Subsequently, we present knowledge obtained from conducting our fault injection experiment, discuss the different applications of our architecture, and then finally present our conclusions.

II. Background & Related Work

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

Nanosatellites today utilize almost exclusively COTS microcontrollers and application processors-SoCs, FPGAs, and combinations thereof [3], [4]. Due to manufacturing in fine technology nodes, and the use of extensively optimized standard IP, they offer superior efficiency and performance as compared to space-grade OBC designs. The energy threshold above which highly charged particles can induce faults (SEE – single event effects) in such components decreases, while the ratio of events inducing multi-bit upsets (MBU), and the likelihood of permanent faults, increase. To adapt such hardware-FT based concepts additional FT-circuitry is required, inflating logic size and producing diminishing returns, resulting in limited scalability and low clock frequencies [5]–[7]. We can observe that traditional FT-concepts applied to modern COTS hardware yield no nanosatellite compatible architectures.

While more sensitive to transient faults than ASICs [8], [9], FPGA-based Soft-SoCs have been shown to offer excellent FDIR potential for miniaturized satellites [10]. Transients in critical parts of the FPGA fabric can be scrubbed [11], while permanent faults may be compensated through reconfiguration with differently routed configuration variants [12]. Fine-grained, non-invasive fault detection in FPGA fabric, however, is challenging, and subject of ongoing research [13],

C.M. Fuchs was with the Leiden Institute of Advanced Computer Science and Leiden Observatory at Leiden University, 2333 CA, The Netherlands, e-mail: christian.fuchs@dependable.space

A. Plaat, E.v.d. Kouwe, and T.P. Stefanov were with the Leiden Institute of Advanced Computer Science

N.M. Murillo and D. Harsono were with Leiden Observatory

This approach was developed for a 4-year European Space Agency (ESA) NPI project supported by two industrial partners. N.M. Murillo and D. Harsono acknowledge funding through the European Union A-ERC grant 291141 CHEMPLAN, by the Netherlands Research School for Astronomy (NOVA), and the Royal Netherlands Academy of Arts and Sciences (KNAW) professor prize.

Manuscript submitted to RADECS2018 on April 17th, 2018, revised September 16th, 2018 and reworked and extended on October 4th, 2018

[14]. Relevant FT-concepts thus rely on error scrubbing, which has scalability limitations and cover only parts of the fabric [11], [13]. We overcome these limitations by implementing fault-detection in software through thread-replication and coarse-grain lockstep within an MPSoC using weakly coupled cores.

Tiled architectures [15], [16] are often used for well paralellizable applications with many low-performance processor cores. Among others, [17] and [16] showed that such typologies can also be exploited to achieve FT for image processing applications with a very specific structure. We combine a tiled architecture with coarse-grained lockstep [18], enabling FDIR without constraining the application type or system architecture. Thus, the architecture presented in this paper is well suited for platform control and can be used as a template, allowing a high level of OBC design freedom, and enabling a considerable amount of testing to be inherited from COTS components and logic.

Thread migration has been shown to be a powerful tool for assuring FT, but prior research ignores fault detection, and imposed tight constraints on an application's type and structure (e.g., video streaming and image processing [19]). Thread-level coarse-grain lockstep of weakly coupled cores instead supports general purpose computing, and in the past, has already been used for high availability, non-stop service, and error resilience concepts. However, in prior research, faults are usually assumed to be isolated, side effect free, and local to an individual application thread [20] or transient [2], [21], entailing high performance [22] or resource overhead [23], [24]. More advanced proof-of-concepts [2], [25], however, attempt to address these limitations, and even show a modest performance overhead between 3% and 25%, but utilize checkpoint & rollback or restart mechanics [2], which make them unsuitable for spacecraft command & control applications.

Many of these limitations and obstacles ultimately can be attributed to low maturity, as a majority of software-FT concepts are published as a concept TRL1 but remain unvalidated. Hence, they could be uncovered, and in many cases, can be potentially resolved through implementation and practical validation [25], increasing maturity to TRL2 or TRL3. However, development of a testable proof-of-concept is a time consuming and costly undertaking [26], as outlined among others by Sangchoolie et al. [27] with limited immediate yield for academic publication. Fault injection for entire OS instances is especially non-trivial [28], as thorough preparation and careful tool-selection is necessary to obtain representative results from a fault injection experiment [29]. Therefore, a broad variety of TRL1 software-FT concepts exist today at a theoretical level [30]–[32], for which validation was only conducted statistically using modeling with different fault distributions or not a all. In this contribution, we therefore conduct validation of our coarse-grain lockstep approach using systematic fault-injection. Thereby we verify the effectiveness of our coarse-grain lockstep FDIR mechanics under stress using a RTOS-based proof-of-concept implementation, increasing maturity to TRL3.

III. A Hybrid Fault-Tolerance Approach

Conventional FT architectures require proprietary logic in hardware to facilitate fault detection and coverage. In contrast, the architecture described in this paper can offer strong FT using just COTS components and proven standard library logic. This is made possible through the use of the FT approach we presented in [18]. The high-level functionality of this approach is depicted in Fig. 1, and consists of three interlinked fault mitigation stages implemented across the embedded stack:

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

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

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

In the remainder of this section, we outline the functionality and purpose of each stage and provide an example of the coarse-grain lockstep's functionality.

Stage 1: Short-Term Fault Mitigation

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

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

We avoid thread synchronization issues as encountered by Kretzschmar et al. in [25] by merely reusing existing OS functionality without breaking or ABI/API guarantees. Therefore, we can continue relying upon pre-existing synchronization mechanics such as

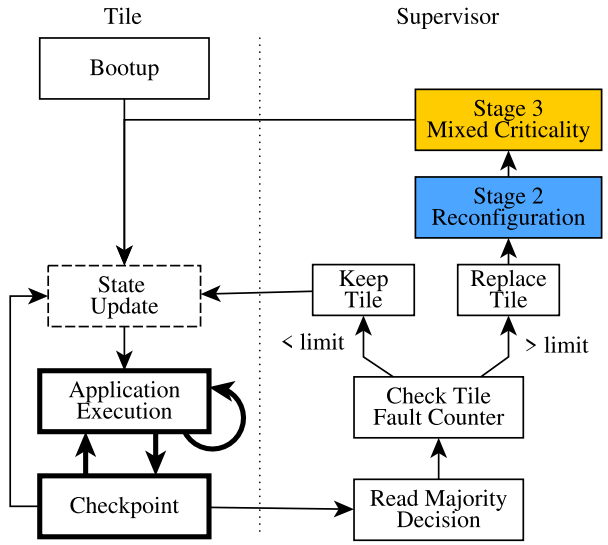


Fig. 1: Stage 1 (white) assures fault detection (bold) and fault coverage. Stages 2 (blue) and 3 (yellow) counter resource exhaustion and adapt the on-board computer application schedule to reduced system resources.

POSIX cancellation points[1] and their bare-metal equivalents (e.g., *RTEMS_NO_PREEMPT* in RTEMS's Classic API if used instead of *newlib* or the POSIX API).

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. In our RTEMS/POSIX-based implementation, we utilize priority-based, preemptive scheduling with timeslicing, allowing threads to delay checkpoints until they reach a viable state for checksum comparison.

An application should provide four callback routines to the OS, which are executed during tile boot by the OS or as part of a checkpoint routine:

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

Besides the addition of these callbacks, no alterations to an application's logic are necessary, except a viable way to assure it can be interrupted by a callback routine periodically. The required development effort for implementing these features in general is comparably low, but depends on the structure of an application. For the astronomical instrumentation applications utilized in our proof-of-concept, these routines could be implemented with 10-20 lines of C-code each. For example, the checksum callback consists almost exclusively of CRC library calls for generating a checksum from a set of state relevant variables and data structures in heap and stack.

Callbacks may be omitted due to practical reasons. For applications which require little code and time for initialization, the initialization routine can be omitted. Applications which are not executed continuously could return a pre-generated checksum to the OS, instead of providing checksum, synchronization and callback handlers, for example, by providing the OS with a signature or checksum before program termination. Applications without a persistent state, or in which the state is continuously re-generated based on input data, no update callback would be necessary.

Checkpoints were designed to be time triggered on each tile independently, but can also be induced by the supervisor through an interrupt, for example, to signal that new threads have been assigned (see also Section VI for additional information on time-vs-interrupt driven checkpoint triggering). Thus, the OS only has to support interrupts, timers, and a multi-threading-capable scheduler. To the best of our knowledge, such functionality is available in all widely used RT- and general purpose OS implementations.

Stage 2: Tile Repair & Recovery

Stage 1 can not reclaim defective tiles, eventually resulting in resource exhaustion. Therefore, in this stage, we recover defective tiles through reconfiguration to counter transients in FPGA fabric. To do so, the supervisor will first attempt to recover a tile using partial reconfiguration. Afterwards, the supervisor validates the relevant partitions to detect permanent damage to the FPGA (well described in, e.g., [33]), and executes self-test functionality on the tile to detect faults in the tile's main memory segment and peripherals. If unsuccessful, the supervisor will repeat this procedure with differently routed configuration variants, potentially avoiding or repurposing permanently defective logic.

[1]For example, sleep, yield, pause; for further details, see IEEE Std 1003.1-2017 p517

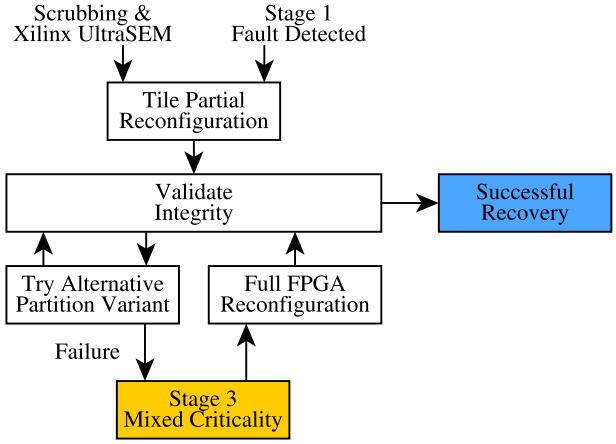


Fig. 2: The objective of Stage 2 is to recover defective tiles and other logic through partial and full FPGA reconfiguration via ICAP. If this is unsuccessful as well and no further spare processing capacity is available to handle future faults, Stage 3 is activated to find a more resource conserving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

The supervisor can also attempt full reconfiguration implying a full reboot of all tiles. Further details on reconfiguration and error scrubbing with a microcontroller-based proof-of-concept implementation for a nanosatellite are available in [34]. If both partial- and full-reconfiguration are unsuccessful and all spare resources have been exhausted, Stage 3 is utilized to assure a stable system core to enable operator intervention.

Stage 3: Applied Mixed Criticality

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

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

Further information on Stage 3 including dynamic thread-mapping, as well as performance, energy and robustness optimization at run-time is available in [35].

A Practical Example

Figure 3 depicts the four cores (C_n) of a quad-core MPSoC running two applications (T_n) on in TMR mode with a single idle spare core

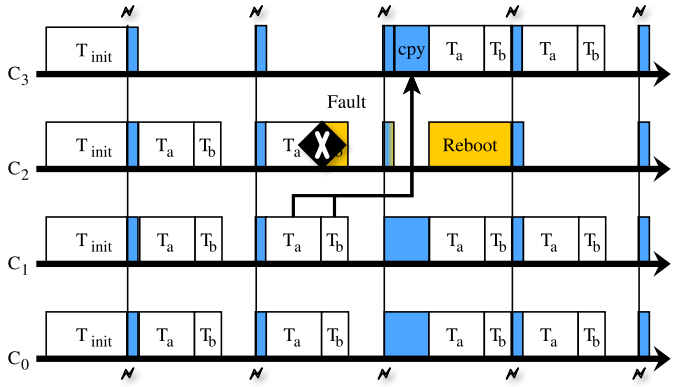


Fig. 3: Tile initialization and a complete Stage 1 lockstep cycle. Application threads are replicated across on a set of tiles, with execution being interrupted by checkpoints (blue), enabling fault tolerance through forward error correction. The checkpoint frequency and the scheduling algorithm can be freely chosen by the developer, requiring only work-equivalence between tiles upon reaching a checkpoint.

available. A fault has occurred during the second lockstep cycle on core C_2, which is subsequently replaced with the idle core C_3. C_3 must retrieve a copy of the state of its threads T_a and T_b from another core. The replaced core, C_2, can subsequently be tested for permanent defects by the OS and the supervisor.

This example illustrates Stage 1's mechanics only and was originally published as part of [18]. [18] also contains a much more detailed explanation of the mechanics outlined in this section, including performance overhead measurements.

IV. The MPSoC Architecture

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

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

A. Supervision & Reconfiguration

Stage 1 can be implemented on a single chip, but we utilize an off-chip supervisor to facilitate FPGA reconfiguration and transient fault scrubbing in the running configuration. The outlined multi-stage FT approach puts only minimal load on the supervisor, and it can thus be again implemented using a traditional radiation hardened or tolerant microcontroller. The FeRAM-based TI-MSP430FR family would be a solid somewhat radiation-tolerant but non-FT substitute, which is today widely used aboard a broad variety of CubeSats and low-performance COTS products designed for nanosatellite use. The level of performance offered by such microcontrollers is usually sufficient only for educational CubeSats and federated systems. However, a supervisor in our architecture only receives the majority voting results from the coarse grain lockstep, controls the FPGA, and facilitates reconfiguration indirectly through ICAP. Hence, the low level of performance of an MSP430FR, for example, is sufficient, and allows an ultra-low-cost implementation of our approach for academic CubeSat projects and scientific instrumentation.

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

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

As depicted in Fig. 1, the supervisor only reacts to disagreement between tiles, otherwise remaining passive. It maintains a fault-counter for each tile and acts as a watchdog. When resolving transient faults within a tile, it increments the fault-counter and induces a state update through a low-level debug interface. After repeated faults, the supervisor will replace the tile by adjusting the thread-mapping of a spare tile, activating it, and rebooting the faulty tile. In case a system developer indicated threshold is exceeded, the disagreeing tile is assumed permanently defunct and not re-used as a spare.

To allow supervisor access to a tile and its address space, each tile is equipped with an AXI debug-bridge (Fig. 4). The supervisor can trigger execution of self-test functionality within a tile to detect faults in peripherals. It can also trigger an adjustment of a tile's thread allocation as part of Stages 1 and 3, making the MPSoC's computational performance, robustness and energy consumption adjustable at runtime.

Majority voting between tiles can be implemented as distributed majority decision [36], then requiring no direct intervention of the supervisor during regular operation. If this is not desired, or lockstep through interrupt triggered checkpoints is implemented, then the supervisor should also take care of receiving the voting results generated on each tile. In that case, the supervisor can access each tile's thread mapping via each tile's debug interface, and if necessary induce a reset or otherwise manipulate a tile without requiring its cooperation.

B. Tile Architecture

Our MPSoC design implements multiple isolated SoC-compartments accessing shared main memory and OS code. Even though the purpose and function of these compartments is different, the topology resembles a tiled architecture instead of a conventional MPSoC design, in which cores share infrastructure and peripherals. This topology allows to maximize Stage 1's fault-coverage capacity and allows task mapping for general-purpose software. Each such tile contains a processor core, local interconnect, and peripheral IP-cores and interfaces as depicted in Fig. 4, resides in its own clock domain, and can be reset independently. Allocating a clock domain to each tile improves timing, and reduces logic-overlap and interdependence between tiles. Furthermore, we can then also utilize partial reconfiguration and frequency scaling for each tile, as well as clock gating.

A tile executes a set of thread replicas, and its loss can be compensated by the rest of the system. To assure a failed tile can not cause performance degradation in the rest of the system (e.g., by continuously accessing DDR or program memory), it can be disconnected off from the global interconnect by the supervisor. Non-masked faults (due to radiation, aging, and wear) disrupt the data or control flow of the software running on a tile. Stage 1 builds upon this capability at the thread-level, as state differences can be detected by other tiles and often even by the malfunctioning tile itself [18].

All tiles are equipped with an identical set of peripheral interfaces, with controllers being mapped to identical locations and address ranges. The tile address space layout is uniform across the system and tiles are indistinguishable for software. Hence, application code and data structures are portable between tiles, simplifying thread migration drastically. This allows us to reduce the computational cost and complexity of software-lockstepping.

Thread allocation and information relevant to the coarse-grain lockstep is stored in a dedicated dual-ported on-chip BRAM on each tile. One port is accessible to the tile's processor core, while the other is read-only accessible to the system, allowing low-latency information exchange between tiles without requiring inter-tile cache-coherence or main memory access.

C. Interconnect Topology & Shared Memory

Figure 5 depicts the MPSoC's high-level topology with clock domains, reset lines and supervisor access facilities. Our MPSoC design utilizes an AXI interconnect in crossbar mode to allow tiles access to shared main and non-volatile memory controllers, though we are currently reworking our MPSoC to instead use a NoC [17].

Main memory is shared between tiles, as SD- and DDR memory controllers are too large and require too much I/O to instantiate for each tile. Each tile has full access to a segment of main memory, which is mapped to the same address range on all tiles (the MMU component in the figures). All tiles can access the main memory read-only to simplify state synchronization and IPC.

For nanosatellite missions to LEO, often only SECDED ECC support is required and readily available in library IP already, while basic error scrubbing can be facilitated in software. For critical, deep-space, and long-term missions, block coding should be used instead to compensate for the increased impact of SEEs and higher likelihood of MBUs in high-density SDRAM. Reed-Solomon ECC as well as error scrubbers are available commercially, or can be assembled from open-source IP. The main memory scrubbers are controlled by the supervisor to avoid potential interference by malfunctioning tiles.

To safeguard main memory, FeRAM [37], MRAM [38], and mass memory from SEFIs, as well as permanent failure, these memories, their controllers, and their AXI interconnects are implemented redundantly to enable fail-over. This also enables further protective measures [39], and allows load distribution for timing critical main memory through segment interleaving. Thereby the available DDR memory bandwidth is increased and the overall latency for memory access can be reduced. This also enables us to recover an instance of a memory controller on short notice without requiring the full system to be halted[2].

[2]Note that depending on the used OS, a reboot of a tile may be required. Linux supports modifications to the memory layout and relocation, while simpler OS, such as RTEMS, do not currently know such functionality.

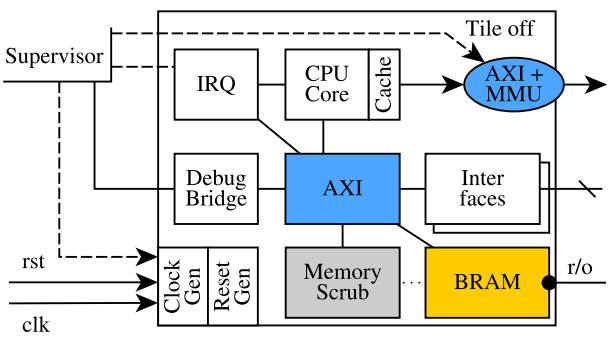


Fig. 4: The logic-side architecture of a tile. Access to local IP bypasses the cache, while access to global memory passes is cached for performance reasons.

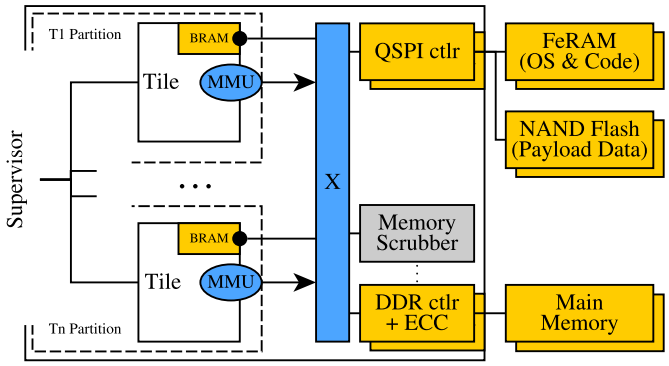


Fig. 5: The topology of our tiled MPSoC design. Each tile exists in its own reconfiguration partition and therefore also clock domain, simplifying routing.

Tiles compete for DDR memory access. As our architecture is implemented on FPGA, the clock frequency of each tile's processor core is lower as on ASIC implemented MPSoCs. In consequence, the global interconnect as well as DDR memory controllers offer abundant throughput at drastically higher clock frequencies. Each processor core caches access to shared memory, drastically reducing the strain on the memory subsystem[3]. Hence, while in principle competing for memory bandwidth, even an 8-tile system can not saturate the two available DDR4 channels in our current MPSoC design. Ideally however, our architecture should be implemented using a NoC instead of a global AXI-interconnect crossbar, which would offer drastically better scalability, more effective caching and buffering, and also a degree of FT.

D. I/O Sanitation

A fault resolved in Stage 1 may cause incorrect data to be emitted through I/O interfaces. This is an inherent limitation of coarse-grain lockstep concepts, and can only be slightly alleviated through additional application-intrusive work-around as described, for example, in [2]. Instead, this limitation is better solved at the logic level through interface-level voting, which is possible with minimal extra logic. For most CubeSats, most nanosatellites, and less critical microsatellite missions, however, this is usually foregone.

Larger spacecraft already utilize interface replications or even voting to assure full hardware TMR, usually requiring considerable effort in hardware or logic to facilitate this replication. Our MPSoC architecture inherently provides interface replications by design, requiring no extra measures to be taken, as the individual tile-interfaces can be directly used for TMRed architecture.

Further safeguards are necessary for very small CubeSats where interface replication is undesirable, for example, due to PCB-space constraints. Most embedded interfaces like I2C and SPI allow a simple majority decision per I/O line. While hardware voting is challenging for large arrays of voters running synchronized at very high frequencies, the CubeSat-relevant interfaces are electrically simple, have a very low pin count, and run at relatively low clock frequencies. Hence, voting for these interfaces can efficiently be implemented on-chip through simple voters assuming tiles signals interface activity. As our coarse grain lockstep mechanics allow software to be executed with slight timing variations, I/O on these interfaces should be buffered and application threads should also indicate the use of an interface.

For packet-based interfaces such as Spacewire, AFDX, CAN, or Ethernet, no hardware- or logic-side solution is necessary. There, packet duplication and integrity checking can be managed efficiently

[3]Access to a tile's state memory still bypasses the cache, but this is implemented directly in high-speed, low-latency on-chip BRAM

at the data link, network and transport layers (OSI layers 2 - 4) or above through frame switching or packet routing. Today, this is common practice in relevant industrial applications such as AFDX [40] and TTEthernet [41] used in related fields such as atmospheric aerospace or safety critical automotive applications.

V. Concept Validation and Test Campaign

The fault-coverage capacity of our architecture is dependent on the correct functioning and effectiveness of the coarse-grain lockstep mechanics described in Section III. Before pursuing a hardware-prototype unit for radiation testing, we must therefore validate the software mechanics first. A proper validation of software has been shown to require systematic testing [29], which is not feasible due to beam/chamber time constraints and cost reasons. The method of choice for validating software-FT therefore is fault-injection, which can be facilitated at different levels and granularity [29], [42], [43].

A. Fault-Injection Technique Selection

Our approach is a hardware/software hybrid design, and the MPSoC is an FPGA design synthesized from source. Thus, fault-injection using netlist simulation [44] or injection into an MPSoC on an FPGA [45], [46] could be facilitated with comparably limited development effort (as compared to developing a new FPGA design from scratch), as we already utilize a development-board based MPSoC design implementation. Several partially [45]–[47] and fully automated test frameworks [42], as well as commercial applications [44], have been developed for this purpose. However, netlist simulation is computationally disproportionately expensive. In prior research, MPSoCs are thus occasionally simulated using SystemC to demonstrate architectural features. Though, implementing fault injection via SystemC for an entire MPSoC running a full OS would still be excessively time consuming, and only viable for fault injection into very simple designs executing less complex software [48]. Both of these approaches therefore prevent a meaningful level of test coverage from being achieved, as testing our FT mechanics requires a full processor-system running an OS.

Faults could also be injected via widely available standard debug tools into software using a variety of academic tools (see also [26], [49], [50]), offering excellent scalability even for very large code sizes [51]. However, this is only viable for simple userland applications [2], the effects of faults on an actual OS cannot be simulated properly [52], unless the entire development toolchain and source code is adapted and utilized [51]. The kind and type of faults which can be simulated by injecting faults into a virtual machine by attaching a debugger are significantly constrained [29] due to the limited means of interactions of a debugger on the virtual machine itself [28].

Fault-injection using system emulation can combine the ease of use of fault-injection into software and the power and flexibility of netlist or SystemC-based techniques. ISA-level fault-injection utilizing system emulation has been shown to be powerful and efficient for conducting black- and grey-box fault-injection [27]. Several test frameworks implementing this approach have emerged in recent years, but most are custom tailored for specific use-cases, or are closed-source and unavailable for public use [26]. Notable exceptions here are the two open source frameworks FAIL [53] and FIES [54], which are freely available as open source software and comparably mature. Hence, we use ISA-level fault-injection to systematically validate our FT approach using an automated test toolchain.

FAIL utilizes a powerful C++ based test controller for thoroughly analyzing small binaries in a fully automated test campaign. While the execution of a single fault-injection run is fully automated, it requires a test-specific controller application. To develop such an application, deep knowledge of the victim application's structures and its compiled intrinsics are required, and must be obtained manually. The development of FAIL is mainly focused on the Intel platform, while ARM is available via GEM5 for a single virtual target SoC or through (potentially destructive) fault-injection into silicon [55].

FIES [54] was developed specifically to validate ARM-based COTS-based critical systems and builds upon the faster and more mature QEMU virtual machine monitor, thereby supporting a broad variety of SoCs and peripheral hardware[4]. Despite allowing slightly less control over virtual hardware than FAIL, it can efficiently handle testing a full OS, without requiring the developer to develop a test monitor with knowledge about application intrinsics. The test campaign described in the remainder of this paper is thus being carried out using an automated test toolchain built around FIES.

B. The Test Pipeline

FIES implements fault-injection through full system emulation in QEMU, and is licensed under GPLv2. In the process of developing our automated test toolchain, we extended FIES' functionality to better support different tracing techniques and added functional improvements, and released the necessary patches[5] to the public. Specifically, we reworked and improved the rule-driven fault-injection engine, rebased FIES from QEMU 1.17 to 2.12 (QEMU-head in December 2017), and added support for the THUMB2 instruction set, as most OS kernels use both ARM and THUMB2 assembly intermixed. We would like to invite the interested reader to try out this extended and reworked version of FIES for their own fault-injection experiments. A detailed function-graph of our Stage 1 logic is available in [18].

While FIES does not support automated testing, it is capable of scripted and systematic fault-injection into opaque binaries. Its fault-injection engine is rule based utilizing an XML-based fault-library, with rules lending themselves well to being generated in bulk based on instruction and memory access traces. This enabled us to develop an automated fault-injection toolchain around this framework, which performs the following steps as set of python scripts:

1) We run the OS image without fault-injection and tracing, outputting the application and OS state for comparison during later steps. We obtain the victim's correct process state, results and correct Stage 1 checksums for each protected payload application upon each checkpoint. To do so, we add additional logging to the test-implementation, therefore producing a different RTOS binary than used in the subsequent fault-injection experiment steps.
2) Execute the test-subject's unaltered binary (without debug code) to generate traces of the process counter and executed opcodes, register access and memory access (golden run).
3) Process the generated traces to constrain fault-injection to lock-step relevant code and data (e.g., omitting platform bring-up and shutdown code). We remove duplicates, and annotate each trace-entry with the number of occurrence in the trace, and generate the actual test-campaign input.
4) For each instruction address and occurrence, generate a fault definition library and launch an instance of FIES, which performs the actual fault-injection.
5) For each run, determine the result of the fault-injection (e.g., OS crash, incorrect checksum, etc.) based on a comparison to the known-correct results obtained in the first step, and log the result to a database.

Steps 1 to 3 are performed at the beginning of a test campaign, whereas steps 4 and 5 are computationally comparably expensive,

[4]Source code publicly available at https://github.com/ahoeller/fies.git

[5]We made our changes in the form of the reworked *FIESer fault injection tool* available at https://fieser.dependable.space rebased as QEMU-git fork.

and executed in parallel by splitting the processed traces. Besides collecting and interpreting the results of a fault-injection run, we also retain tile state information to enable manual analysis if necessary. This includes a tile's output, CPU and QEMU processor context dumps, as well as the logs generated by FIES during the fault-injection, and its exit code.

During development, fault-injection was also conducted manually by targeting specific locations in the applications' binary structure. We chose interesting data and logic which could cause an incorrect application state, alter the applications' control flow, or would result in a different run-time behavior in a tile. These experiments were conducted to verify the functionality of our approach, the experiment setup and injection toolchain.

C. Target Implementation and Payload

Our fault-injection campaign was conducted against an implementation of our approach in RTEMS 4.11.2, using the ARMv7a-Zynq board-support-package, which closely resembles the tiles of our MPSoC. RTEMS is a real-time OS used in a broad variety of space applications, from platform control to instrumentation. We cross-compiled the kernel image from Fedora 28 x86_64 with standard compile flags (`-marm -mfpu=neon -mfloat-abi=hard -O2`) in RTEMS GCC 4.9.3. We chose not to utilize the Linux kernel for our fault-injection experiments to maximize the level of control, and reduce the time overhead due to kernel bootup, even though our approach was designed with Linux-compatibility in mind.

As payload application, we utilized ESA's Next Generation DSP benchmark[6] run as POSIX threads within RTEMS, which is an ESA standard benchmark application used to measure and compare DSP system performance. To re-confirm our results, we performed the same experiments with the same application used to conduct the performance estimation in [18], resembling the NASA/James Webb Space Telescope's Mid-Infrared Instrument's readout software [56].

D. Test Space and Target Components

In practice, choosing the right test-space for a practical OS-scale implementation is non-trivial, in contrast to what is described as ideal in literature. Sufficient test coverage for such software can often be unobtainable in practice, and even fault-injection using state-of-the-art tools requires a compromise between realism and test-coverage to avoid runaway test-times and extreme equipment. Besides test coverage, our architecture has to cope with not merely transient faults, but also radiation-induced permanent faults, as these are common in modern memories and processor components flying in space.

When considering fault-injection into program code, a broad variety of secondary faults may occur, and many of these can only be analyzed by the actual developer of an application using source code [51]. However, for our architecture, we are interested in the practical effect an injected fault induces in the system and on an application, and if our coarse-grain lockstep can gracefully handle them. Our Stage 1 implementation exists as part of the OS's scheduler and as a set of application callbacks, and therefore faults will have the following effects on software executed on a tile:

- Data corruption associated with access to main memory, caches, registers and scratchpad memory due to non-correctable ECC words, faults in read/write logic, and misdirected access in control logic.
- Incorrect or non-execution of instructions in the processor pipeline during the Instruction Fetch, decode, execute and write-back stages.
- Control-flow deviations and data corruption due to failure of interfaces and tile peripherals, and faults in controller logic or the FPGA's I/O components.

We developed a set of template fault definitions which our fault-injection toolchain utilizes to generate suitable fault definitions to reproduce faults in these components based on the aforementioned program traces. Our toolchain utilizes these templates to generate a FIES fault library for each instruction and memory address, allowing scripted fault-injection.

Transient Fault-Injection: Transients are being injected as bit-flips into registers and the processor pipeline using the program counter as trigger. Time triggered injection (also supported by FIES) would be insufficient, as it would prevent exact predictability faults. For instructions which are visited more than once, we can trigger faults after the n-th occurrence enabled by extending the FIES framework's fault definition mechanics. In the same manner, faults were injected into memory access operations based on the read or written physical address, thereby simulating non-correctable upsets in ECC protected words in caches and main memory, as well as general faults in address logic or buffers. To better simulate ECC-errors and faults in the address logic, we can also directly replace accessed data or the address of the operation, instead of just injecting bit-flips.

Permanent Fault-Injection: Permanent faults are injected into every access to the interconnect by the system, including access to main memory and devices address space. They were not injected into general purpose registers, special registers, and the CPU pipeline, however, as the effects of faults in these components are fatal at the latest after a brief time period. In our MPSoC architecture, such faults will result in crash of the RTOS and can then be detected at the next checkpoint by other tiles. While it is important to not ignore parts of our fault model, testing for faults with a known outcome would needlessly inflate the test space.

Functional Interrupts: FIES allows injecting periodic and intermittent faults (the effects of which persist for a short period of time and are resolved afterwards). This functionality was used to simulate SEFIs. We chose 100ns as fault-duration for SEFIs, the period-equivalent to 10 clock cycles at 100MHz, the clock speed emulated by QEMU for the Zync MPSoC, with approximately 20 instructions executed by a Cortex-A CPU per guest-second. We believe this represents reasonably well the interruption effect and the reset-induced outage of specific circuit groups due to SEFIs. However, we are not aware of research further analyzing the actual timing and interruption behavior SEFIs in different components of a SoC and parts of the FPGA fabric.

Fault Placement during Execution: After executing bring-up code and OS initialization, our victim binary processes payload software for 3 lockstep cycles, and then terminates the RTOS. We chose a time interval of 2 seconds as checkpoint frequency, which is reasonable for operation in LEO when passing through increased radiation zones such as the South Atlantic Anomaly, based on radiation-testing data for Ultrascale [8], [57] with preliminary information obtained from Ultrascale+ FPGAs [58]. In our current victim binary implementation, execution during the golden run takes approximately 7 seconds of guest-virtual time, which on our test system is equivalent to approximately 30 seconds of host-time. In case the experiment does not terminate in time, for example, due to control flow corruption or infinite loops, the experiment is terminated externally. We terminate the FIES/QEMU process after 45 seconds, and configured FIES to end an injection run after executing given number of instructions (e.g., 10 times the number of instructions executed in the golden run).

Faults are injected after the first and until after the second checkpoint. This allows faults to propagate within the system, corrupt the OS and application state, potentially causing deviations in the program

[6]Source code publicly available at https://essr.esa.int

Impact	Fault Detectable	Detectable by victim tile	Detectable by other tiles	Recovery through	Observed Effect per Fault Type: Transient	Permanent	Intermittent
Corrupted State	yes	yes	yes	state-update	49%	44%	53%
Thread Crash	yes	yes	no	state-update	8%	17%	10%
Lockstep Failure	yes	no	yes	reboot	1%	2%	1%
OS Crash	yes	no	yes	reboot	10%	18%	15%
Masked (no effect)	(some*)	(yes*)	(no*)	(reboot*)	32%	19%	21%

TABLE I: Fault injection experiment results for our RTOS implementation divided into transient, permanent, and intermittent faults. Notice that our setup does not enable us to detect data corruption that does not impact on the thread state. Masked faults affecting OS data structures could be detected through OS-level EDAC, while memory protection and virtual memory would allow us to detect a majority of these faults through access violations. Neither of these measures are in place in our current RTEMS proof-of-concept implementation, but would improve system robustness during a mission.

flow, without requiring excessive experiment time. Subsequently, we can analyze if our coarse-grain lockstep approach could detect the effects of a fault on the system (if any), and if they were resolved through a state update from another compartment. Upon reaching the third checkpoint, the application state should have recovered and thereby generated checksums, and the CPU state should match the golden run's results. This allows us to verify the full FDIR cycle from fault injection to recovery. To reduce the test space, we decided to limit fault injection and exclude the OS's platform bring-up and shutdown code. The actual bootup and shutdown sequences of a tile are not relevant to validating our implementation, and therefore fault-injection in these parts would yield little insight into its performance.

Limitations: We chose the duration of fault-injection run to allow our victim binary to exhibit the entire FDIR circle, while assuring a reasonably short test-run to still allow systematic testing and test coverage. However, this does not allow detection and observation of dormant or latent faults, for example, affecting OS data structures and logic resulting time-delayed regressions. The time allotted to each fault-injection run therefore is a direct trade-off between test-coverage and the ability to observe more long-term effects (e.g., after 10 to 20 checkpoints) in a system.

It would be feasible to inject faults in QEMU's virtual hardware directly as well. However, this is not supported in FIES in a scripted manner at the time of writing, requiring instead source-code modification for each device in QEMU. We can elevate this limitation, we simulate the effects of such faults on the program flow through data corruption, we inject faults during access to device address space. This allows us to represent certain effects of faults on control logic, such as incorrect addressing, and upsets in the peripheral's buffers.

E. Results & Comparison to Related Work

Table I contains results of our fault-injection experiments. We grouped the observed effects into different categories indicating their outcome for simplicity. In payload-application code, a majority of the injected transient faults resulted in a corruption to the payload applications' state. With less than 20% of all faults, the application of the entire OS crashed or terminated prematurely (tile-resets were treated as early termination). Faults affecting the lockstep mechanics themselves (e.g., resulting in false comparison or incorrectly generated checksums from correct data) were observed as well, but were rare due to the minimal code and data footprint of the lockstep implementation.

During permanent fault injection, a comparable share of bit-flips resulted in a corrupted thread state and thus checksum-comparison mismatch. However, this number by itself is misleading, as the amount of masked upsets without noticeable effects plummeted to just 19%, while the share of thread- or OS-crashes increased. Therefore, we can deduct that a number of faults which due to transient faults would have resulted in just thread state corruption, now instead result in crashes. The amount of detected faults in turn was increased again by faults which were previously masked. Intermittent faults have a similar effects to permanent ones, though with slightly fewer crashes and more faults affecting only the payload application.

To provide context, we compare our results to literature. Unfortunately, few published coarse-grain lockstep concepts have been implemented at all and most of these were tested using statistical projections only. At the time of writing, we are aware of only a single realistic, publicly-released validation report by Dobel et al. [2] considering practical fault-injection instead of statistical estimation. Therefore, we compare our results to Dobel et al. in order to provide a second point of reference for verification.

When directly comparing our results to Dobel et al.'s *transient-only* fault-injection report, the share of faults causing application and OS crashes measured with our approach is increased. For transient faults, this can at least in part be explained with the different capabilities of Dobel et al.'s proposed lockstep mechanics, which is facilitated through application intrusive function call hooking. Thereby, they can offer more fine-grained protection than our approach, but introduce considerable code overhead and constrain the concept's application to one specific OS. The encountered differences are consistent across all categories of fault-effects. We measure a higher amount of masked faults, a decreased amount of detected state deviations, and an increased amount of crashes with our approach.

Dobel et al. consider their fault-injection measurements overly optimistic, as they utilized only payload *"applications of little complexity (leading to few potential candidates for fault injection)"* [2]. Their validation and FT concept is constrained to handling transient faults, while SEFIs or permanent effects are not covered as these faults were injected into a user-land application of their approach through a debugger. Dobel et al. assume the OS to be guaranteed fault-free, we instead inject faults into a full OS including POSIX libraries with payload application threads. In light of this bias, the reduced detection rates of our approach can be considered reasonable.

VI. Lessons Learned

Comparing our results to Dobel et al.'s fault-injection results, we conclude that our approach performs comparable to related work (considering the different underlying fault detection and coverage mechanics). It is important to note that a major share of faults resulting in no observable effect may indeed be masked and require no measures to be taken, as they have no impact on the application state [59]. This is a limitation of our current fault injection toolchain, as faults are also injected into registers and memory which may be overwritten by subsequent instructions, or faults that cause self-masking control flow deviations. Such situations occur, for example, due to faults in branch or comparison instructions triggering the same iteration of a loop more than once. These have no practical impact on the application state

while, and also do not cause timing deviations significant enough to produce a difference in work conducted to the next checkpoint.

Our coarse-grain lockstep implementation can detect faults resulting in a crash or in corruption of the thread state, but is currently oblivious to silent data corruption in kernel data structures. Velasco et al. propose in [60] to apply erasure coding for critical OS data structures, while code signing is today widely used for tamper-proving embedded devices. Such functionality would allow us to also detect silent data corruption in rarely accessed OS structures and device drivers code and data. In absence of such functionality, a tile's checkpoint handler could directly generate checksums for certain critical kernel data structures, though the extent to which this is possible is limited.

Based on our experiments, we find comparably few faults inducing crash and lockstep-failures, even when specifically relevant code sections were targeted. This is important, as our architecture depends on reaching a majority decision using 3+ thread-replicas in lock step, and a roughly 10% ratio of tile-OS crashes is sufficient to provide the necessary degree of voter stability, making synchronization rare, and thread reassignments an exception. Too low voter stability could have caused constant fluctuations in thread-assignments, requiring frequent state synchronization, putting a high strain on the system as a whole.

When experimenting with different compiler flags, we found that faults injected in equivalent code segments of differently compiled binaries could result in different observed effects. We determined through introspection of the relevant target binary parts, that the changed behavior was caused by specific compiler flags. Loop unrolling (GCC's `-funroll-loops` flag) had a particularly positive effect when injecting permanent and intermittent faults. This is due to the fact that this feature in practice introduces a certain level of code redundancy instead of performing the loops conditional jump and re-running the same instructions. Serrano Cases et al. in [61], [62] as well as Lins et al. in [63] have begun to explore this issue in regards to finding ways to exploit these features for improving reliability, but otherwise industry and literature today seem oblivious on this issue. Designers of software-FT measures must therefore also consider the impact of a broad variety of behavior-altering flags and toolchain settings supported by modern compiler suites, as these have a direct impact on the utilized FT mechanics as well as validation.

FIES originally offered no support for the THUMB instruction set. However, most OS kernels, many device drivers, and even standard library functions tend to mix THUMB and ARM instructions even within code segments, requiring special compiler interwork to support jumps and function calls between the two instruction sets. Jumps from ARM instructions to THUMB instructions without interwork yields an undefined instruction exception, as the opcode-encoding for ARM and THUMB instructions is different. This effectively prevents undetected, incorrect jumps in ARM/THUMB interwoven code segments. Therefore, we added support for the THUMB and THUMB2 instruction sets to FIES, to assure consistent tracing and fault-injection results. Due to the observed concept implementation behavior during fault-injection before solving this limitation, we argue instruction set mixing could be exploited to improve fault detection. Critical code segments could intentionally be assembled with strong instruction-set interweaving to assure that an incorrect jump immediately results in an exception instead of silent data corruption or control-flow deviations. For C-code, this can be achieved per function using target attributes and prefixes, or more fine-grained using preprocessor definitions and pragma.

When designing our coarse grain lockstep approach, we were aware of two ways of inducing checkpoints: timer driven and interrupt induced checkpoints. If timers are used on each tile to trigger a periodic checkpoint at a later time, checkpoint initialization on each tile is effectively decoupled and independent. Interrupt induced checkpoints are however centrally triggered by the off-chip supervisor, creating a potential single point of failure. At design time we therefore considered timer driven lockstep to be a better choice under the viewpoint of fault potential than an interrupt driven approach. However, our fault injection campaign showed us that interrupt induced checkpoints can have significant advantages. When preparing the victim binary, a certain level of determinism is required to assure that the known-correct application state obtained from the golden run still correlates with the fault-injection runs. This showed that the timer-handling related logic is comparably fragile, not due to the timer and alarm triggering functionality itself, but rather due to the vulnerability of the code related to and the setup of timer-handling code used during checkpoints. Instead, the same RTOS implementation using interrupt-triggering can be facilitated with considerably less code and therefore offers better resilience.

VII. Applications

The MPSoC architecture described in this contribution was developed for miniaturized satellite use, as an ideal platform for the software-FT approach described in [18]. It was implemented on a Xilinx XCKU5P FPGA with modest resource utilization (28% LUTs, 33% BRAMs, 16% FFs, 5% DSPs) and 1.92W total power consumption with four Microblaze-equipped tiles. In this design, tiles were equipped each with one peripheral I2C master controller, one SPI master, as well as a dual-channel GPIO controller, which is rather typical for CubeSat applications, while CAN or Spacewire are today not widely used aboard CubeSats. However, in [18] we also showed that a tile's logic footprint is relatively small in comparison to the large area occupied by globally shared resources such as the global AXI interconnects and the DDR memory controllers.

This architecture is not specifically dependent on utilizing ARM processor cores, but can be implemented with any FPGA-implementable core. Our choice of the ARM platform was taken in part to allow thread migration between soft- and hard-cores (e.g., on Zynq Ultrascale+), maximum comparability to COTS mobile-market and embedded MPSoCs with secondary use aboard a major share of CubeSats. Especially for low-budget CubeSat users in research or university projects, standard vendor library cores such as Xilinx Microblaze may be an excellent alternative to our Cortex-A choice, as these are readily available to and often even free of charge.

The relaxed cost, energy, and size constraints aboard microsatellites and larger spacecraft allow an implementation of our MPSoC spanning multiple FPGAs. A multi-FPGA MPSoC variant offers better scalability due to easier routing, can tolerate chip-level defects, and SEFIs to the globally shared memory controllers, these can be distributed to different FPGAs. Thread replicas can then be distributed across FPGAs, allowing non-stop operation even during full reconfiguration.

This approach and architecture could very well be implemented on ASIC without reconfiguration and Stage 2, and we see this as a "big-space" variant of our approach. An ASIC implementation offers lower energy consumption, and allows higher clock rates due to reduced timing and shorter paths. If manufactured in an inherently radiation hard technology such as FD-SoI [64], it would be less susceptible to transients and more robust to permanent faults. Due to the drastically increased development cost and required manpower, the resulting OBC would not be viable for most miniaturized satellite applications (not anymore "on a budget").

VIII. Conclusions

The 3-stage FT approach combined with its MPSoC host system presented in this paper is the first practical, non-proprietary, affordable architecture suitable for FT general-purpose computing aboard

nanosatellites. It utilizes FT measures across the embedded stack, and combines topological with software functionality, utilizing only extensively validated standard parts. Thereby, we enable the use of nanosatellites in critical space missions, while the architecture allows trading processing capacity for reduced energy consumption or fault-coverage.

An OBC relying upon this architecture can be facilitated with the minimal manpower and financial resources. The MPSoC can be implemented using only COTS hardware and extensively validated, and widely available library IP, requiring no proprietary logic or costly, custom space-grade processor cores. It offers a high level of resource isolation for each processor, utilizing architectural features originally conceived for ManyCore systems to achieve FT. Each tile functions as a stand-alone processing compartment with dedicated I/O, existing in its own clock domain and reconfiguration partition, thereby minimizing shared resources and reducing routing complexity. Compartments were purposefully designed to best support thread-level coarse-grain lockstep of weakly coupled cores, while allowing partial reconfiguration without stalling the rest of the system. The architecture was implemented successfully, and tested on current generation Xilinx Zynq/Kintex and Virtex FPGAs with 4, 6 and 8 tiles, and validated through fault-injection into RTEMS.

Having developed a proof-of-concept implementation, our architecture must now undergo radiation testing to validate it for on-orbit use as soon as possible. Before this is possible, each individual component of our architecture first had to be validated separately. This has been achieved or proven by fellow researchers for all individual components comprising our architecture except for our software-FT mechanics. To validate this component, we therefore conducted a fault-injection campaign, to deliver the high level of test-coverage required to assure the effectiveness of our concept implemented in RTEMS. In this paper, we present the results of this campaign, demonstrating that the approach is indeed effective and efficient.

As the other parts of our architecture have been verified separately in related work, the test campaigns outlined in this contribution represent the final step in validating our current development-board based proof-of-concept architecture. In the process of preparing and conducting the fault-injection campaign, we not only validated these mechanics, but also gained a deeper understanding of its behavior under stress and could increase the maturity of our approach from TRL2 to TRL3. The positive outcome of the fault-injection campaign now enables us to develop a hardware prototype OBC to test the resilience of our architecture at the system-level using laser fault-injection and radiation testing to achieve TRL4.

Acknowledgment

We would like to thank Gianluca Furano, Giorgio Magistrati, Antonios Tavoularis and Kostas Marinis at ESTEC/TEC-EDD and Melanie Berg at the NASA Goddard Space Flight Center for their support and invaluable feedback. We thank ARM Ltd. for making available the relevant processor and infrastructure IP.

References

[1] M. Langer and J. Bouwmeester, "Reliability of cubesats-statistical data, developers' beliefs and the way forward," in *AIAA SmallSat*, 2016.

[2] B. Döbel, "Operating system support for redundant multithreading," Ph.D. dissertation, Dresden University, 2014.

[3] F. Kastensmidt and P. Rech, *FPGAs and Parallel Architectures for Aerospace Applications: Soft Errors and Fault-Tolerant Design*. Springer, 2016.

[4] 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.

[5] S. Gupta *et al.*, "SHAKTI-F: A fault tolerant microprocessor architecture," in *IEEE ATS*, 2015.

[6] M. Pigno *et al.*, "A testbench for validation of DST fault-tolerant architectures on PowerPC G4 COTS microprocessors," in *Eurospace DASIA*, 2011.

[7] A. S. Jackson, "Implementation of the configurable fault tolerant system experiment on NPSAT-1," Ph.D. dissertation, Naval Postgraduate School Monterey, 2016.

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

[9] L. A. Tambara *et al.*, "Heavy ions induced single event upsets testing of the 28 nm Xilinx Zynq-7000 all programmable SoC," in *IEEE REDW*, 2015.

[10] M. Wirthlin, "High-reliability FPGA-based systems: space, high-energy physics, and beyond," *Proceedings of the IEEE*, vol. 103, no. 3, 2015.

[11] A. Stoddard *et al.*, "A hybrid approach to FPGA configuration scrubbing," *IEEE Transactions on Nuclear Science*, 2017.

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

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

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

[15] A. K. Singh, M. Shafique, A. Kumar, and J. Henkel, "Mapping on multi/many-core systems: survey of current and emerging trends," in *DAC*. ACM, 2013.

[16] P. Meloni *et al.*, "System adaptivity and fault-tolerance in NoC-based MPSoCs: the MADNESS project approach," in *IEEE DSD*, 2012.

[17] N. K. R. Beechu *et al.*, "Hardware implementation of fault tolerance NoC core mapping," *Springer Telecommunication Systems*, 2017.

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

[19] 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.

[20] A. Höller *et al.*, "Software-based fault recovery via adaptive diversity for COTS multi-core processors," 2015, arXiv:1511.03528.

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

[22] A. D. Santangelo, "An open source space hypervisor for small satellites," in *AIAA SPACE*, 2013.

[23] E. Missimer, R. West, and Y. Li, "Distributed real-time fault tolerance on a virtualized multi-core system," *Euromicro ECRTS, OSPERT*, 2014.

[24] Z. Al-bayati *et al.*, "Fault-tolerant scheduling of multicore mixed-criticality systems under permanent failures," in *IEEE DFT*, 2016.

[25] U. Kretzschmar *et al.*, "Synchronization of faulty processors in coarse-grained TMR protected partially reconfigurable FPGAs," *Elsevier RESS*, 2016.

[26] R. Natella, D. Cotroneo, and H. S. Madeira, "Assessing dependability with software fault injection: A survey," *ACM Computing Surveys*, 2016.

[27] B. Sangchoolie *et al.*, "Light-weight techniques for improving the controllability and efficiency of isa-level fault injection tools," in *PRDC*. IEEE, 2017.

[28] D. Cotroneo *et al.*, "Experimental analysis of binary-level software fault injection in complex software," in *2012 Ninth European Dependable Computing Conference*. IEEE, 2012, pp. 162–172.

[29] R. Natella *et al.*, "On fault representativeness of software fault injection," *IEEE Transactions on Software Engineering*, vol. 39, no. 1, pp. 80–96, 2013.

[30] S. Malik and F. Huet, "Adaptive fault tolerance in real time cloud computing," in *IEEE World Congress on Services*, 2011.

[31] K. Smiri *et al.*, "Fault-tolerant in embedded systems (MPSoC): Performance estimation and dynamic migration tasks," in *IEEE IDT*, 2016.

[32] Z. Al-bayati *et al.*, "A four-mode model for efficient fault-tolerant mixed-criticality systems," in *IEEE DATE*, 2016.

[33] N. T. H. Nguyen, "Repairing FPGA configuration memory errors using dynamic partial reconfiguration," Ph.D. dissertation, The University of New South Wales, 2017.

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

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

[36] N. Katta *et al.*, "Ravana: Controller fault-tolerance in software-defined networking," in *ACM SIGCOMM*. ACM, 2015.

[37] Z. Zhang *et al.*, "Single event effects in COTS ferroelectric RAM technologies," in *REDW*. IEEE, 2015.

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

[39] C. M. Fuchs *et al.*, "A fault-tolerant radiation-robust mass storage concept for highly scaled flash memory," in *Eurospace DASIA*, 2015.

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

[41] V. Gavrilut *et al.*, "Fault-tolerant topology and routing synthesis for ieee time-sensitive networking," in *RTNS*. ACM, 2017.

[42] W. Mansour and R. Velazco, "An automated seu fault-injection method and tool for hdl-based designs," *IEEE Transactions on Nuclear Science*, 2013.

[43] A. Da Silva and S. Sanchez, "LEON3 ViP: A virtual platform with fault injection capabilities," in *DSD*. IEEE, 2010.

[44] K. Suresh *et al.*, "Debug environment for a multi user hardware assisted verification system," Feb. 1 2018, US Patent App. 15/646,003.

[45] J. L. Nunes *et al.*, "Fired–fault injector for reconfigurable embedded devices," in *PRDC*. IEEE, 2015.

[46] D. Cozzi, "Run-time reconfigurable, fault-tolerant FPGA systems for space applications," Ph.D. dissertation, Universität Bielefeld, 2016.

[47] M. Alderighi *et al.*, "Evaluation of single event upset mitigation schemes for sram based fpgas using the flipper fault injection platform," in *DFT*. IEEE, 2007.

[48] P. Lisherness and K.-T. T. Cheng, "SCEMIT: A SystemC error and mutation injection tool," in *DAC*. ACM, 2010.

[49] S. Winter *et al.*, "The impact of fault models on software robustness evaluations," in *ICSE*. IEEE, 2011.

[50] A. Johansson, "Robustness evaluation of operating systems," Ph.D. dissertation, TU Darmstadt, 2008.

[51] E. van der Kouwe and A. S. Tanenbaum, "HSFI: accurate fault injection scalable to large code bases," in *DSN*. IEEE, 2016.

[52] D. Cotroneo and R. Natella, "Software fault injection for software certification," *IEEE Security & Privacy*, 2013.

[53] H. Schirmeier *et al.*, "FAIL: An open and versatile fault-injection framework for the assessment of software-implemented hardware fault tolerance," in *EDCC*. IEEE, 2015.

[54] A. Höller *et al.*, "FIES: a fault injection framework for the evaluation of self-tests for COTS-based safety-critical systems," in *MTV*. IEEE, 2014.

[55] J. Isaza-González *et al.*, "Dependability evaluation of cots microprocessors via on-chip debugging facilities," in *IEEE LATS*, 2016.

[56] M. Ressler *et al.*, "The Mid-Infrared instrument for the James Webb Space Telescope," *Astronomical Society of the Pacific*, 2015.

[57] D. S. Lee *et al.*, "Single-event characterization of the 20 nm xilinx kintex ultrascale field-programmable gate array under heavy ion irradiation," in *REDW*. IEEE, 2015.

[58] T. Lange *et al.*, "Single event characterization of a Xilinx UltraScale+ MP-SoC FPGA," in *SpacE FPGA Users Workshop*, 2018, preliminary.

[59] X. Li and D. Yeung, "Application-level correctness and its impact on fault tolerance," in *HPCA*. IEEE, 2007.

[60] A. o. Velasco, "A hardening approach for the scheduler's kernel data structures," in *CompSpace at ARCS2017*, 2017.

[61] A. Serrano Cases *et al.*, "Automatic compiler-guided reliability improvement of embedded processors under proton irradiation," in *RADECS*. IEEE, 2018.

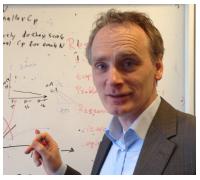
[62] A. Serrano-Cases *et al.*, "On the influence of compiler optimizations in the fault tolerance of embedded systems," in *IOLTS*. IEEE, 2016.

[63] F. M. Lins *et al.*, "Register file criticality and compiler optimization effects on embedded microprocessor reliability," *IEEE Transactions on Nuclear Science*, 2017.

[64] M. Kochiyama *et al.*, "Radiation effects in silicon-on-insulator transistors with back-gate control method fabricated with OKI semiconductor 0.20 μm FD-SOI technology," *Elsevier Nuclear Instruments and Methods in Physics Research*, 2011.

Christian M. Fuchs has been active as consultant in computer reliability and security since 2001, pursue academic studies starting 2009. He received his B.Sc.E. from Hagenberg University of Applied Sciences, Austria, specializing on operating system security and his M.Sc. in computer science with minor on space engineering from Technical University Munich (TUM), Germany. There he joined the MOVE CubeSat project, working on the CubeSat FirstMOVE, launching into LEO in 2014 and conducting operations and later post-mortem, subsequently also developing the successor satellite MOVE-II (launch in Q4/2018). His research since the FirstMOVE post-mortem analysis has been the development a software-FT based fault tolerant OBC architecture for miniaturized spacecraft using COTS technology, and since 2016 he is continuing this research at Leiden University as Ph.D. researcher through an ESA project grant.

Nadia M. Murillo received her B.Sc. degree in Physics in 2011 and a M.Sc. in Astrophysics in 2012 from National Tsing Hua University (NTHU), Taiwan. She received her Ph.D. in Astrophysics from Leiden University in 2017 and is currently a postdoctoral researcher in the James Webb Space Telescope's (JWST) Mid-Infrared Instrument (MIRI) team at Leiden University. Besides her involvement in MIRI, her research interests include the early stages of multiple star formation, and chemical characterization of the envelopes of young embedded protostars. Her current research is focused on combining observation with ground-based and space telescopes, together with chemical and physical models in order to understand the formation of stars.



Aske Plaat studied at University of Alberta, Edmonton, Canada, and received his M.Sc. in Information Management from Erasmus University Rotterdam, the Netherlands, on distributed systems and compiler optimization. In 1996 he received his Ph.D. in artificial intelligence for research on search algorithms, and developed the MTD(f) algorithm, which now is part of Intel's parallel compiler suite. After receiving his Ph.D., he held post-doc and professorship positions at Vrije Universiteit Amsterdam, Tilburg University, NHTV Breda, the University of Alberta, and the Massachusetts Institute of Technology (MIT), working on distributed algorithms, artificial intelligence, and trajectory optimization for space missions. Since 2014, he is professor at Leiden University, where he is currently serving as the director of Leiden Universities computer science department (LIACS). Today, his research interests are in reinforcement learning.

Erik van der Kouwe received B.Sc. degrees in computer science and mathematics at the Vrije Universiteit Amsterdam, followed by an M.Sc. in computer science. He received his Ph.D. in Computer Science in 2015 for his research in operating system reliability, virtualization, and software fault injection with Andrew S. Tanenbaum. He is currently an assistant professor at Leiden University. His research interests in computer systems are centered around security and reliability, specifically software fault injection, software defenses based on compiler instrumentation, and benchmarking practices in systems security. His current research aims to protect legacy software against attacks.

Daniel Harsono received his B.Sc. degree in Astrophysics in 2008 from University of California Los Angeles (UCLA). He received his M.Sc. degree in Astronomy from Leiden University in 2010, and his Ph.D. in 2014. He is currently a postdoctoral researcher working in the Atacama Large Millimeter/submillimeter Array (ALMA) support group. His astrochemistry research focuses on the formation of low-mass protostars, early Solar System, and formation of complex molecules. He combines astronomical observations from X-ray to submm with the development in simulations and massively parallel radiative transfer applications.

Todor P. Stefanov received Dipl.Ing. and M.S. degrees in computer engineering from the Technical University of Sofia, Bulgaria, in 1998 and the Ph.D. degree in computer science from Leiden University, The Netherlands, in 2004. From 1998 until May 2000, he was a Research and Development Engineer with Innovative Micro Systems, Ltd., Sofia., designing ASIC IP and a reconfigurable MicroSystems-on-Silicon In-Circuit Emulator based on FPGAs. After holding Post-Doc positions at Leiden University and TU-Delft, he received a professorship at Leiden University in 2008. His research is focused on system-level design automation for FPGAs, MPSoC design, and parallel computing, and real-time scheduling.