# Validating Software-Based Fault Tolerance for Space Use by Example (DRAFT)

Christian M. Fuchs*†, Nadia M. Murillo†, Aske Plaat*, Erik van der Kouwe*, and Daniel Harsono†
*Leiden Institute for Advanced Computer Science †Leiden Observatory;
Leiden University, The Netherlands email: c.fuchs@liacs.leidenuniv.nl

***Abstract*—Software-side FT measures can be validated using a variety of different techniques, from fault-injection (FI) into physical hardware, to netlist-simulation, and statistical modeling. However, not all techniques are suitable to validate a FT concept for a specific application. This has become a roadblock when adopting software-FT measures for space application, as these require thorough, systematic testing based on a realistic fault model, practical fault-injection, and a proper software implementation at a comparable scope. In consequence, dependable computing research offers a variety of innovative and scalable software-based FT concepts, which however have largely been ignored by the space industry in favor of traditional hardware-FT. In this contribution, we thus demonstrate how systematic validation of software-based FT can be conducted for applications in deep space and similar irradiated environments. We do so by example of our own coarse-grain lockstep approach which implements thread-level forward-error-correction, and was implemented within an industry-standard real-time OS.**

***Index Terms*—fault-tolerance, fault-injection, space, satellite, coarse-grain lockstep**

## I. Introduction

Modern embedded technology is a driving factor in satellite miniaturization, enabling a smaller, lighter, and cheaper class of spacecraft, fueling a massive boom in satellite launches and a rapidly evolving new space industry. These micro- and nanosatellites (100kg-1kg mass) have become increasingly popular for a variety of commercial and scientific missions. However, they suffer from low reliability, discouraging their use in long or critical missions, and for high-priority science.

For larger spacecraft, various hardware-based fault tolerance (FT) concepts are available, which are not common aboard miniaturized spacecraft due to tight energy, mass, and volume constraints, in addition to disproportional costs. Conventional embedded and mobile-market systems-on-chip (SoCs) are deployed in their stead, which usually lack even basic FT functionality. A significant share of post-deployment failure of nanosatellites can be attributed directly to the failure these components and peripheral electronics [1]. Therefore, we developed a non-intrusive, flexible, hybrid hardware/software architecture [2] to assure FT with commercial-off-the-shelf (COTS) mobile-market technology based on an FPGA-implemented MPSoC design.

Central to our FT architecture is a novel software-side FT approach, which combines coarse-grain thread-level lockstep and forward-error-correction. It is platform agnostic and requiring no space-proprietary IP, custom processor cores, or radiation-hardened chip technology to offer strong fault-coverage. It not only offers fault-coverage, but also is used to trigger other protective features within our MPSoC, requiring thorough validation before a custom-PCB based prototype can be developed. Validation of such FT measures requires systematic testing of the actual concept implementation, a realistic fault model, and a suitable testbench. This is non-trivial, and therefore software-side FT concepts are often validated using statistical means only, but not actually implemented to allow practical validation using fault-injection. This resulted in a gap between theory and application, with industry dismissing FT research concepts due to a lack of maturity and an assumed tendency to ignore practical implementation obstacles.

In this paper, we show by example of our own implementation, how realistic and systematic validation of software-based FT can be conducted for space-applications and similar irradiated environments. To do so, we first describe the architecture's intended operating environment, design constraints, and describe the physical fault model encountered in Section II. In Section III, we then discuss how these challenges are handled today in related work, and outline which solutions currently are practically available to the space industry and scientific spacecraft designers. We then provide a brief overview of our multi-stage FT architecture and introduce our coarse-grain lockstep approach in Section IV. For an RTOS implementation of this approach, we then develop a concise fault-model in Section V, and then analyze which testing techniques are available to verify our approach, as well as the advantages and limitations of each option. Having chosen the most suitable approach for our use-case, in Section VII we develop an automated testbench which we use to systematically conduct fault-injection campaigns in an environment closely resembling our target MPSoC with a set of predefined fault-templates, as described in Section VIII. We then present results of our fault injection experiment in Section IX, compare them to related work conducted by Dobel at al. in [3] for an academic coarse-grain lockstep implementation. Before presenting conclusions, the pitfalls we discovered while preparing and executing our fault-injection experiments are discussed in Section X. We also highlight interesting aspects in the behavior of our FT approach.

## II. The Space Environment & Radiation

The form factor constraints aboard miniaturized satellites [4] and the drastically different fault-model [5] prevent the re-use of many FT and testing approaches developed for ground applications. Even in atmospheric aerospace applications, these usually consider availability, non-stop operation, and safety, but rarely guarantee computational correctness in a fully autonomous system.

Physical access to a satellite during a mission is in practice impossible, and servicing missions were conducted only on

rare occasions for satellites of outstanding importance in low-Earth orbit (LEO) in manned space missions. Signal travel times, brief communication windows, and scarce bandwidth make live interaction impractical. Thus, faults detected by our approach are resolved fully autonomously during a satellite mission, which may exceed 10 years.

High-energy particles are the main cause of faults [6] during a satellite mission. They 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. These particles can corrupt logical operations or induce bit-flips within memory and semiconductor logic (single event effects - SEE), and may cause displacement damage (DD) at the molecular level to a chip's substrate and circuitry. The energy threshold above which SEEs induce transient faults decreases in chips manufactured in fine technology node, and the ratio of events inducing multi-bit upsets or permanent faults increases. 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 bit-upsets and SEFIs can be transient or permanent, while DD is always permanent [5]. The accumulative nature of permanent faults implies accelerated and often spontaneous aging, which must be handled efficiently throughout a mission. The cumulative effect of charge trapping in the oxide of electronic devices (total ionizing dose – TID) further impacts the lifetime of an on-board computer (OBC). However, chips manufactured in certain new technology nodes, such as recent generation FPGAs [7] show drastically better than expected TID performance [8] and resistance to latch-up in contrast to projections based on technology scaling [9].

In LEO, the residual atmosphere and Earth's magnetic field provide some protection from radiation, but this absorption effect diminishes quickly with distance. Many miniaturized spacecraft operate in this region, and forego FT in favor of developing a functional satellite within the boundaries of their limited resources and manpower. Most nanosatellites today do utilize COTS microcontroller- and application processors-SoCs, FPGAs and combinations thereof [10], [11], occasionally introducing basic, custom-designed redundancy with ground-triggered fail-over. The choice to utilize such components instead of proven FT technology usually is the result of risk acceptance due to a lack of viable alternatives. Designers in general are aware that these components may fail at any given point in time, and may result in a loss of mission. Risk-acceptance at this scale is a viable approach for low-priority science and missions with brief duration. This is not an option for critical and long-term missions with a scientific or commercial objective.

Most nanosatellite hardware development efforts are more comparable to hardware-prototyping than to the sophisticated and thorough ASIC development process. FPGAs have, hence, become increasingly popular as they are well suited for this design approach, despite being more vulnerable to radiation than ASICs, due to their better FDIR potential [12].

## III. Related Work

FT is traditionally implemented through circuit-, RTL-, core-, and OBC-level majority voting [13]–[15] using space-proprietary IP, which is difficult and costly to maintain and test. Circuit-, RTL-, and core-level voting are effective for small SoCs such as microcontrollers, but this does not scale for the more potent processor cores used in modern mobile-market MPSoCs [16], [17]. Software takes no active part in fault-mitigation in these concepts, as faults are suppressed at the circuit level and usually only represented using hardware-side fault counters, preventing the effective assessment of a processor's health.

SoC architectures for spacecraft usually undergo radiation testing or laser fault injection, as the state of the art in the field today is focused on hardware-level FT measures or specialized manufacturing (RHBD and RHBM – radiation hardened by design/manufacturing). Relevant radiation tests have been conducted for the FPGAs utilized in our project among others by Lee et al. in [18] and Berg et al. in [8], or are currently ongoing (Lange et al. [19]). Tests for further components such as memory and supervisor-$\mu$Cs are available in test databases such as ESCIES, NASA's NEPP[1] and the IEEE REDW Records. For our architecture, radiation tests for the utilized components yielding device-specific data allowing us to estimate fault frequencies, types, and effects on the FPGA on which our MPSoC is implemented. We require this information to choose appropriate checkpoint frequencies [2] for our coarse-grain lockstep approach, but by itself, radiation testing does not allow an assessment of the architecture itself.

Prior research on software-based FT, often utilizes very simplistic fault models, considering faults to be isolated, side effect free and local to an individual application thread [20] or purely transient [3], [21]. Many practical application obstacles could be avoided in many cases through implementation [22], but fault-injection [23] of an actual concept implementation is time consuming, and often requires also a hardware implementation. Especially fault-injection for entire OS instances is non-trivial [24], and through preparation and careful tool-selection is necessary to obtain representative results from a fault-injection experiment [25]. Therefore, a sizable share of FT concepts exists at a theoretical level [26]–[28], and instead of fault-injection or hardware-testing, statistical modeling using different fault distributions are utilized instead. This is a viable approach for validating FT concepts directed towards, e.g., yield maximization [29] and aging [30], but not for validating software-side FT operating in a challenging environment in which faults are stochastic events.

In this contribution, we therefore conduct systematic validation of our coarse-grain lockstep approach using practical fault-injection to verify the effectiveness and effective of our coarse-grain lockstep FDIR mechanics under stress. Specifically, we must assure voter stability, a sufficiently high level of fault detection, and verify fault-isolation and recovery, determine the level of voter stability, hence the likelyhood

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

of a fault to result in a crash or another failure requiring replacement using spare resources. This information is essentially to choose an appropriate checkpoint frequency for the lockstep [2], which mainly defines the fault-coverage level of our MPSoC. Together with FPGA-level fault-information obtained from radiation tests outlined earlier in this section, and information on the mission specific target environment, we can then calculate the appropriate fault-frequency for a specific mission and spacecraft.

## IV. Our Hybrid FT Architecture

The coarse-grain lockstep and forward-error correction mechanics which are the subject of the fault-injection experiment is one of multiple FT measures utilized in our architecture [2]. The high-level logic of this approach is depicted in Figure 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, requiring no modifications to an application or knowledge about intrinsics. Faults are resolved through state re-synchronization and thread-migration to tiles with spare processing capacity. Stage 1 is described in detail in [2], in which was also established an upper bound for the performance cost of the lockstep.

**Stage 2 recovers tiles through FPGA reconfiguration**, thereby counteracting resource exhaustion. It assures the integrity of the FPGA's running configuration and deploys scrubbing as well as Xilinx SEM to correct transients in FPGA fabric. Its objective is to repair defective tiles suffering from upsets in tile logic, and cover permanent faults using alternative configuration variants. The mechanics of Stage 2 have been well researched and validated in [31], [32], and FPGA reconfiguration has been demonstrated on-orbit [33].

**Stage 3** is activated when too few healthy tiles are available due to accumulation of permanent faults, and **re-allocates**

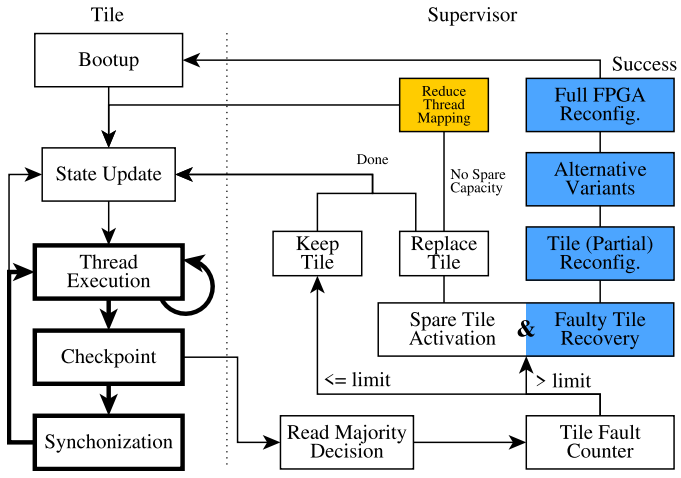


Fig. 1: Stage 1 (white) assures fault detection (bold) and fault coverage, Stage 2 (blue) and 3 (yellow) counter resource exhaustion and adapt to reduced system resources.

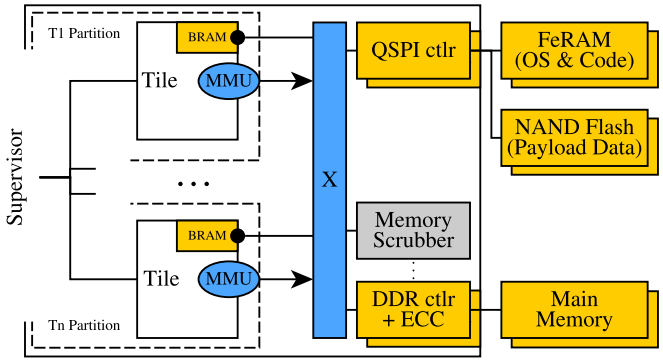


Fig. 2: The topology of our tiled MPSoC design. Each tile exists in its own reconfiguration partition and therefore also clock domain, simplifying routing.

**processing time** to maintain reliability. To do so, it utilizes the thread-level **mixed criticality** found in satellite computing, assuring sufficient compute resources are available to high-criticality applications by sacrificing performance of lower-criticality threads. This functionality as well as the adaptive capabilities enabled are presented in [34].

We deploy this three stage FT architecture on top of a tiled MPSoC design, consisting of multiple isolated SoC-compartments accessing shared main memory and OS code. Each compartment (see also Figure 3) contains a processor core, peripheral-IP (e.g., interrupt controller, timer, etc.) and interface cores, as well as a supervisor access port. While interfaces thus are in general replicated for each tile, this is not viable for external memory controllers (main and program memory) due to their large footprint, package-pin and PCB space limitations. As on-chip memory alone is insufficient for sophisticated data handling applications, tiles utilize a shared set of DDR and SPI controllers via an AXI interconnect in crossbar mode. These controllers are implemented redundantly to enable fail-over, safeguard from SEFIs, and enable interleaved access to reduce congestion.

The objective of our technology development project is to offer FT using commodity processor cores, without requiring space-proprietary processor cores and custom-designed hardware-FT functionality. Hence, the processor architecture considered in our project is the ARM Cortex-A53 application processor, as it is widely used in embedded devices, well supported by standard development toolchains, and has excellent scalability. As our FT approach is platform agnostic, a functionally equivalent MPSoC was also developed using the Xilinx Microblaze cores. This MPSoC design was implemented on a variety of Xilinx Ultrascale FPGAs. On a XCKU5P FPGA this 4-tile design results in modest resource utilization (28% LUTs, 33% BRAMs, 16% FFs, 5% DSPs) and 1.92W total power consumption. Further details on this MPSoC implementation are available in [35] and [**?**].

## V. Practical Effects on our Architecture

As described in Section II, our MPSoC operates in a hostile, irradiated environment. The precise effects on semiconductors induced by a fault vary based on the actually effected logic

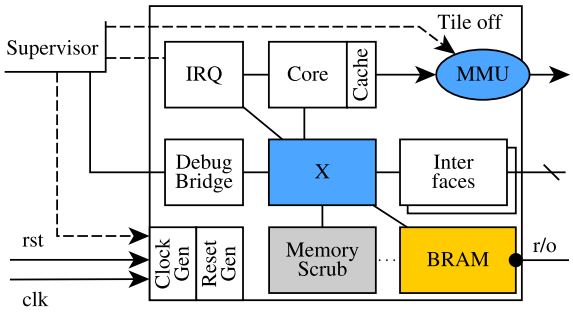


Fig. 3: Logic-side architecture of a tile with clocking and reset facilities. Access to local IP bypasses the cache, while access to global memory passes is cached for performance reasons.

and technology used [5]. Today, determining these effects on a semiconductor is of major concern for traditional hardware-FT based systems, and the only practical way to evaluate them. However radiation testing can occur only at a very late stage in development, and the results may vary even for the same chip-designs manufactured in different fabrication lines [5]. In combined with knowledge of the specific chip design, this effectively yields heritage and increases a system's technology readiness level, instead of verifying the effectiveness of a specific FT mechanic. This is due to the fact that radiation testing is costly, the available test-chamber times are limited, and the test itself is time consuming, as the whole system is exposed to a radiation cocktail designed to mimic the actual operating environment and dosage over time, as far as technically possible.

Besides the expensive nature of radiation testing, the proper validation of software also requires systematic testing of the software-side components. Thus, we must consider the actual effect and impact of faults from a programmatic perspective, which neither statistical estimation nor radiation testing can deliver. Coarse-grained lockstep in our approach is implemented within the scheduler and as a set of application callbacks, and therefore faults will have the following practical effects 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.
- Incorrect behavior of tile peripherals resulting in failure or misfiring of interrupts and timers, or additional interrupts due to ECC error syndromes, and therefore control-flow deviations.
- Failure of individual interfaces due to faults in controller logic or the FPGA's I/O components.

Naturally there will also be faults induced into the FPGA and MPSoC design beyond the listed, or affect the OBC at a larger scale (e.g., full FPGA failure). However, these are either already covered as part of Stage 2 (FPGA-fabric level issues), or detectable and solvable by the supervisor (which also acts as a watchdog), or are beyond the scope of what a single-chip FT approach may deliver. As such, these faults are either fatal to a tile, therefore directly detectable by other tiles, or cannot be handled within the MPSoC itself. Instead, this requires FPGA reconfiguration or system level measures to be taken.

There are several different fault injection techniques which allow us to test the behaviour of an OS-level target application. As our approach is a hardware/software hybrid design, fault injection using netlist simulation [36] would be possible with acceptable additional implementation effort. It grants maximum control over the type and effect of faults and allow near ideal realism, and several partially [37] and fully automated test frameworks [38] as well as commercial applications [36] have been developed for this purpose. Unfortunately, this type of testing is disproportionately time consuming preventing a meaningful level of test coverage from being achieved, unless only a specific component was subjected to the test instead of the MPSoC as a whole.

Faults could also be injected into the software in userland using a debugger, for example. As this is only possible for simple userland applications [3], the effects of faults on an actual OS cannot be simulated [39]. Validation of embedded software for low-power SoCs using ia32/amd64 hosts may also bias the outcome of fault-injection experiments. While benchmarking kernel functionality in userland allows a meaningful worst-case performance estimation, this is not the case for fault-injection [25]. Debugger based fault injection into a virtual machine can alleviate these constraints by allowing an OS to be tested, but suffers performance and introspection limitations imposed by the debugger-interface [24]. In consequence, the kind and type of faults which can be simulated using an external debugger are significantly constrained [25]. Nonetheless, this method has become comparably popular, as it requires only limited implementation effort and can be facilitated using standard open-source tools, and still yields practical fault-injection results instead of just synthetic statistical calculations.

Fault-injection using system emulation can combine many of the advantages of both techniques, without being constrained by a debugger interface regarding the potentially injectable fault types. In prior research, MPSoCs sometimes are simulated using SystemC to close the gap between system netlist simulation and pure software emulation, if only hardware with very simple software is simulated [40]. However, implementing fault-injection via SystemC for an entire MPSoC running a full operating system would not only be excessively time consuming, but also require considerable development effort just to achieve a functional simulation, even if realism was ignored. Virtualization assisted emulation, instead, allows faults to be injected into pre-existing emulated hardware and SoCs, while being computationally comparably cheap and requiring no major changes to the victim application. Several test-frameworks following this approach have emerged in recent years, of which the two open source frameworks FAIL [41] and FIES [42] are comparably mature

and publicly available. Hence, we use this fault-injection technique to systematically validate our FT approach using an automated test pipeline.

## VI. Target OS and Used Payload Applications

Our fault injection experiments were 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 over our experiment and reduce time-overhead due to OS bootup, while e.g. Monson et al. [43] provide an analysis of the Linux kernel itself using fault-injection into an FPGA-based SoC.

As payload application, we utilized ESA's Next Generation DSP benchmark[2] 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 [2], which resembles the NASA/James Webb Space Telescope's Mid-Infrared Instrument's readout software [44].

## VII. The Fault-Injection Pipeline

The choice of which emulation-based fault-injection tool to use is non-trivial as the available tools do not offer equivalent features. They differ regarding the target environment, test setup and intended test subject scope, means of technology used to facilitate fault-injection and functionality. FAIL utilizes a powerful C++ based test controller for thoroughly analyzing small binaries. The controller binary requires deep knowledge of victim binary intrinsincs and the meaning of the program structure which cannot be automatically obtained, but then it can conduct a fully automated test. Development 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 via JTAG into silicon[3] [45]. FIES[4], on the other hand, 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 virtual hardware. While this tool alone does not allow fully automated testing and allows slightly less control over virtual hardware, it enables scripted and systematic fault-injection into opaque binaries. It can efficiently handle testing a full OS, and can be integrated well into a fully automated test pipeline. The test campaign described in the remainder of this paper is therefore being carried out using the open-source fault injection framework FIES [42].

FIES implements fault injection through full system emulation in QEMU, and licensed under GPLv2. In the process of developing our automated test pipeline, we extended FIES's functionality to better support different tracing techniques and added functional improvements, and released the necessary patches[5] to the public. Specifically, we 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. QEMU in principle would allow faults to be injected in virtual hardware, though this is not implemented in FIES today.

Our test pipeline consists of the following steps implemented as a set of python scripts:

1) Obtain the victim application's process state, results and correct Stage 1 checksums for each protected payload application. We run the emulation without fault-injection and tracing, outputing the application and OS state for comparison during later steps.
2) Execute a golden run and generate traces of the process counter and executed opcodes, register access and memory access with the same parameters as in the previous step. This allows us to e.g., include additional debug output or otherwise alter the victim-binary's code in the previous step. Thereby easily obtaining a correct victim OS state without distorting the actual golden-run trace.
3) Filter the traces to constrain fault injection to coarse-grain lockstep 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 trace file.
4) For each address and occurrence, we generate a FIES fault definition library and launch an instance of FIES.
5) For each run, then 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. In our test campaign, we also collect tile state information and human readable output logged to each tiles' serial port, as well as FIES's output to STDERR and the emulator's exit code to enable manual analysis in the future if necessary.

Steps 1-3 are one-time operations, whereas steps 4 and 5 can be executed in parallel by splitting the processed traces.

## VIII. Injected Faults and Target Components

Before our fault injection pipeline was in place, fault-injection was conducted 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

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

[3]Due to constant reboots required for fault injection at the OS-level and the potential of kernel-level faults physically damaging the target hardware.

[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 as rebased as QEMU-git fork.

run-time behavior in a tile. These experiments were initially conducted to verify the functionality of our approach as well as the experiment setup and injection tool.

As of implementation of our fault-injection pipeline, transient and permanent faults are injected systematically with transients being injected as bit-flips into registers and the processor pipeline using the program counter as trigger. 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 upsets and simulate faults in the address logic, we can also directly replace accessed data, instead of just injecting bit-flips. Permanent faults are injected into main-memory only, but not into general purpose registers, special registers, and the CPU pipeline, as their effect in these components is fatal.

SEFIs in different functional units of a tile's processor core, controllers, and interface logic may also have effects which are neither permanent nor truly transient. FIES allows injecting periodic and intermittent faults (the effects of which persist for a short period of time and are resolved afterwards). This function was used to simulate SEFIs. As fault-duration for intermittent faults, we chose 100ns, the period-equivalent to 10 clock cycles at 100MHz[6] with roughly 20 instructions executed by a Cortex-A pipeline. 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 radiation-test data further analyzing the actual timing and interruption behavior SEFIs in different components and parts of the FPGA fabric beyond documenting their existence.

After executing bring-up code and OS initialization, our victim binary executes payload software for 3 lockstep cycles, and then terminates the RTOS experiment. 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], [18] in preliminary information obtained from Ultrascale+ FPGAs [19]. 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, e.g., due to control flow corruption or infinite loops, the experiment is terminated after 45 seconds by killing the FIES/QEMU process.

Faults are injected from the beginning of the first and until before the second checkpoint. This allows faults to propagate within the OS, corrupt the application state, and program flow, without requiring excessive experiment time. Subsequently, we can analyze if our coarse-grain lockstep approach could detect the effects of the injected 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 number of test cases, we decided to limit fault injection and exclude the platform's bring-up code and the OS's shutdown sequences. 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.

[6]The clock speed emulated by QEMU for the Zync board support package.

## IX. Results & Comparison to Related Work

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

To place these results in context, we sought to compare our results to literature. Unfortunately, few coarse-grain lockstep concepts have been implemented practically, and we are aware of only one publicly released validation report by Dobel et al. [3] considering practical fault-injection, instead of statistical estimation. Therefore, we provide these preliminary fault injection results in comparison to Dobel et al. in order to provide a second point of reference for verification, but also to help guide future research on software-side FT measures.

When directly comparing our results to Dobel et al.'s *transient* fault injection report, the share of faults causing application thread and OS crashes measured with our approach is increased. In part, this can be explained by 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 introducing considerable code overhead and constraining its

| Impact | Fault Detectable | Detectable by victim tile | Detectable by other tiles | Recovery through | Observed Effect per Fault Type: Transient | Permanent | Intermittent |
|---|---|---|---|---|---|---|---|
| Corrupted Thread 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% |
| No Effect (Masked) | (some*) | (yes*) | (no*) | (reboot*) | 32% | 19% | 21% |

Tab. I: Preliminary fault injection experiment results. Notice that our setup does not enable us to detect test silent data corruption or faults resulting in incorrect I/O. Masked faults affecting OS data structures could be detected OS-level EDAC, while memory protection would allow our implementation to detect a majority of these faults. Neither of these measures are in place in our current RTEMS proof-of-concept implementation, but should be used utilized for a radiation testing candidate or on-orbit use.

application to one specific OS stack. Dobel et al. also consider their fault injection measurements overly optimistic, as they utilized only payload *"applications of little complexity (leading to few potential candidates for fault injection)* [3]". 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. also assume the OS to be guaranteed fault-free, we inject faults into a full OS including POSIX libraries with payload application threads. In light of this bias, and the fundamentally different fault-detection mechanics, the reduced detection rates can be considered reasonable. This is consistent across all categories of fault-effects we encountered: we measure a higher amount of masked faults, a decreased amount of detected state deviations, and an increased amount of crashes with our approach.

## X. Discussion & Lessons Learned

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 [46]. This is a limitation of our current fault injection pipeline, as faults are also injected into registers and memory which may be subsequently overwritten, or faults that cause self-masking control flow deviations. Such situations occur e.g., due to faults in branch or comparison instructions triggering the same iteration of a loop more than once. 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 currently is oblivious to silent data corruption in kernel data structures and code. Velasco et al. propose in [47] to apply erasure coding for critical OS data structures, while code signing is today widely used for tamper-proving of 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 only directly derive a checksum for certain critical kernel data structures, though the scope to which this is possible is limited.

Low voter stability could cause constant fluctuations in thread-assignments, requiring near constant state synchronization and therefore put a high strain on the system as a whole. Based on our experiments, we find comparably few faults inducing crash and lockstep-failures encouraging, even when these 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.

When experimenting with different compiler flags, we found that the same injected faults could result in different observed effects. Initially, we assumed this to be related to compiler optimizations, which we could verify through introspection of the relevant target binary parts. We discovered that loop unrolling (GCC's `-funroll-loops` flag) had a particularly positive effect when injecting permanent and intermittent faults, likely 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. While being ignored today in literature and industry, designers of software-side FT measures should also consider the broad variety of behavior-altering flags and toolchain settings supported by modern compiler suites, as these can have a direct impact on the utilized FT mechanics as well as validation.

FIES originally offered no support for the THUMB instruction. However, most OS kernels, many device drivers, and even standard library functions mix THUMB and ARM instructions, requiring special compiler-interwork to support jumps and function calls between these two instruction sets. Jumps from ARM instructions to THUMB instructions without interwork yields an undefined instruction exception, as the opcode-numbers of ARM and THUMB instructions do not overlap, effectively preventing incorrect jumps in strongly ARM/THUMB interwoven code segments. Therefore, we added support for THUMB2 mode to FIES, to assure consistent tracing and fault-injection results. Due to the observed guest-vm 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 measure, we were aware of two different 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, checkpointing on each tile is thereby effectively decoupled and independent. Instead interrupt induced checkpoints are directly triggered by the external supervisor, creating a potential single point of failure. At design time we therefore considered timer driven checkpointing to be a better choice than interrupt induced checkpointing, but our fault injection campaign showed us that interrupt induced checkpointing 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 us that the timer-handling related logic is rather fragile, whereas an interrupt-based implementation can be very simple and offer better resilience.

## XI. Conclusions

In this paper, we presented validation results of our software-side fault-tolerance approach presented in [2], and the automated fault-injection toolchain developed for this purpose. This concept is the key element of a multi-stage FT architecture combining different measures across the embedded stack into an FPGA-based MPSoC design. Our architecture is designed to enable an MPSoC consisting only of COTS hardware and widely available, pre-existing library IP, to achieve the high level of reliability required to enable the use of nanosatellites in critical space missions. The resulting architecture enables an on-board computer to adapt to varying performance requirements at run-time as described further in [34], allowing processing capacity, energy consumption or fault coverage to be maximized. To our knowledge, this is the first scalable general-purpose on-board computer architecture that offers strong fault coverage for miniaturized spacecraft, which is not dependent on proprietary processor cores and requires no custom ASICs.

Prior research in the field often foregoes the practical implementation of software-based FT entirely, resorting to validation using statistical means only, instead of actually implementing and practically testing concepts. This has resulted in a large gap between academic theory and practical application, with implementation constraints being ignored or validation being performed based on unrealistic assumptions or with inadequate means. In this paper, we showed that proper validation of software-side FT also requires implementation and practical testing using fault-injection in an environment closely resembling the target processor architecture, and the intended operating environment. Validation of software-side FT also has to be conducted differently than for traditional hardware-voting based systems, and requires not just empirical testing but also systematic validation. Hence, we developed a practical fault-profile based on the threat-scenario found in the intended target-environment, and analyze how these faults can be best simulated based on scope of the target architecture. We discussed different fault-injection techniques and determine the one most suited for validating a full OS, using our application as example. Based on this knowledge, we developed an automated fault-injection pipeline, which enables systematical testing using system emulation to validate the complete FDIR cycle. To place our results into context, we compare them to literature and discuss further knowledge obtained beyond raw numbers while conducting our fault-injection campaign.

The overall results obtained during validation are positive and the software-side FT implementation tested meet our expectations: the approach can deliver the fault-detection and recovery functionality necessary to allow our architecture as a whole to deliver strong fault coverage. In the process of developing our fault-injection pipeline and preparing the test setup as well as the victim binary, we did gain much deeper knowledge on our FT-concept's behavior at runtime under stress. Through practical validation using actual fault injection thus not only allows us to obtain information necessary for choosing parameters in the later system design process, but can also help to improve the concept itself and its implementation.

As the other parts of our architecture have been validated separately in literature, this validation represents also the final step in validating our current development-board based proof-of-concept. This enables us to now proceed to develop a prototype OBC implementation, which is necessary to characterize the resilience of our architecture as a whole using hardware-based testing through laser fault-injection and radiation-testing.

## References


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

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

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

[4] M. Marinella and H. Barnaby, "Total ionizing dose and displacement damage effects in embedded memory technologies," Sandia National Laboratories, Tech. Rep., 2013.

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

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

[7] L. A. Tambara *et al.*, "Heavy ions induced single event upsets testing of the 28 nm xilinx zynq-7000 all programmable soc," in *Radiation Effects Data Workshop*. IEEE, 2015.

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

[9] M. Kochiyama *et al.*, "Radiation effects in silicon-on-insulator transistors with back-gate control method fabricated with OKI semiconductor 0.20 $\mu$m FD-SOI technology." Elsevier, 2011.

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

[11] 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.

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

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

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

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

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

[17] R. DeCoursey *et al.*, "Non-radiation hardened microprocessors in space-based remote sensing systems," in *Int. Society for Optics and Photonics: Remote Sensing*, 2006.

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

[19] P. Lange, Thomas *et al.*, "Single event characterization of a Xilinx UltraScale+ MPSoC FPGA," in *SEFUW: SpacE FPGA Users Workshop*, 2018, preliminary.

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

[23] R. Natella, D. Cotroneo, and H. S. Madeira, "Assessing dependability with software fault injection: A survey," *ACM Computing Surveys (CSUR)*, vol. 48, no. 3, p. 44, 2016.

[24] D. Cotroneo, A. Lanzaro, R. Natella, and R. Barbosa, "Experimental analysis of binary-level software fault injection in complex software," in *2012 Ninth European Dependable Computing Conference*. IEEE, 2012, pp. 162–172.

[25] R. Natella, D. Cotroneo, J. A. Duraes, and H. S. Madeira, "On fault representativeness of software fault injection," *IEEE Transactions on Software Engineering*, vol. 39, no. 1, pp. 80–96, 2013.

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

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

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

[29] L. Jiang, R. Ye, and Q. Xu, "Yield enhancement for 3d-stacked memory by redundancy sharing across dies," in *Computer-Aided Design (ICCAD), 2010 IEEE/ACM International Conference on*. IEEE, 2010, pp. 230–234.

[30] L. Leem, H. Cho, J. Bau, Q. A. Jacobson, and S. Mitra, "Ersa: Error resilient system architecture for probabilistic applications," in *Proceedings of the Conference on Design, Automation and Test in Europe*. European Design and Automation Association, 2010, pp. 1560–1565.

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

[32] D. Cozzi, "Run-time reconfigurable, fault-tolerant FPGA systems for space applications," 2016.

[33] D. Petrick, D. Espinosa, R. Ripley, G. Crum, A. Geist, and T. Flatley, "Adapting the reconfigurable spacecube processing system for multiple mission applications," in *Aerospace Conference, 2014 IEEE*. IEEE, 2014, pp. 1–20.

[34] C. M. Fuchs *et al.*, "Dynamic fault tolerance through resource pooling," in *NASA/ESA AHS*. IEEE, 2018.

[35] ——, "Fault-tolerant nanosatellite computing on a budget," in *RADECS*. IEEE, 2018.

[36] K. Suresh, C. W. Selvidge, S. Gupta, and A. Jain, "Debug environment for a multi user hardware assisted verification system," Feb. 1 2018, uS Patent App. 15/646,003.

[37] M. Alderighi, F. Casini, S. D'Angelo, M. Mancini, S. Pastore, and G. R. Sechi, "Evaluation of single event upset mitigation schemes for sram based fpgas using the flipper fault injection platform," in *Defect and Fault-Tolerance in VLSI Systems, 2007. DFT'07. 22nd IEEE International Symposium on*. IEEE, 2007, pp. 105–113.

[38] W. Mansour and R. Velazco, "An automated seu fault-injection method and tool for hdl-based designs," *IEEE Transactions on Nuclear Science*, vol. 60, no. 4, pp. 2728–2733, 2013.

[39] D. Cotroneo and R. Natella, "Software fault injection for software certification," *IEEE Security & Privacy*, vol. 99, no. 1, p. 1, 2013.

[40] P. Lisherness and K.-T. T. Cheng, "Scemit: A systemc error and mutation injection tool," in *Proceedings of the 47th Design Automation Conference*. ACM, 2010, pp. 228–233.

[41] H. Schirmeier, M. Hoffmann, C. Dietrich, M. Lenz, D. Lohmann, and O. Spinczyk, "FAIL: An open and versatile fault-injection framework for the assessment of software-implemented hardware fault tolerance," in *Dependable Computing Conference (EDCC), 2015 Eleventh European*. IEEE, 2015, pp. 245–255.

[42] 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.

[43] J. S. Monson, M. Wirthlin, and B. Hutchings, "A fault injection analysis of linux operating on an fpga-embedded platform," *International Journal of Reconfigurable Computing*, vol. 2012, p. 7, 2012.

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

[45] J. Isaza-González, A. Serrano-Cases, F. Restrepo-Calle, S. Cuenca-Asensi, and A. Martínez-Álvarez, "Dependability evaluation of cots microprocessors via on-chip debugging facilities," in *Test Symposium (LATS), 2016 17th Latin-American*. IEEE, 2016, pp. 27–32.

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

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