Validating Software-Implemented Fault Tolerance Mechanisms for Critical Space Systems

Regular Paper

***Abstract*—Fault-tolerant system architectures for space applications are currently validated using system-level testing. This is viable for systems relying on hardware measures, but unsuitable for fault tolerance (FT) implemented in software. Fault injection using a realistic test-setup is considered good practice to validate software, but also challenging. Hence, few published software FT concepts have been validated in this way. Practical implementation, validation, and the possibility to compare an implementation's performance to literature, however, are prerequisites for the space industry to consider FT concepts. In consequence, software measures have largely been ignored for space applications, and instead the field resorts to hardware-only functionality, which is ineffective for modern, low feature-size semiconductors. This must change. To increase acceptance of software FT concepts, we therefore provide a guide and test template for conducting systematic validation of software FT measures. We do so by example of a thread-level coarse-grain lockstep implementation we developed for use aboard satellites.**

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

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. Micro- and nanosatellites (100kg-1kg mass) have become increasingly popular for a variety of commercial and scientific missions, which were considered infeasible in the past. However, they suffer from low reliability, discouraging their use in long or critical missions, and for high-priority science.

For larger spacecraft, various protective concepts are available to assure fault tolerance (FT) through hardware measures. However, these concepts are effective only for semiconductors manufactured in traditional, sturdy technology nodes. Such hardware can not be utilized aboard miniaturized spacecraft due to tight energy, mass, volume constraints, and high cost. Conventional embedded and mobile-market systems-on-chip (SoCs) are deployed in their stead, which only utilize error correction to handle the handle wear and aging effects encountered on the ground. A significant share of post-deployment issues aboard nanosatellites can be attributed directly to the failure of these components and peripheral electronics, mostly due to radiation-related effects [1].

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

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

Validation of such FT measures requires systematic testing of the actual concept implementation, a realistic fault model, a consistent fault model definition, and a suitable test setup. Being software, fault injection at the system level does not offer a sufficient level of test-coverage, and instead a variety of fault injection techniques for software are available. While validation using fault injection using a realistic test-setup is best practice in fault tolerance research and space-hardware development, almost no published coarse grain lockstep concepts have been implemented and validated in this way. In contrast, the concepts described in academic publications today are predominantly validated only using mathematical models only, but usually not actually implemented to allow fault injection.

At the time of writing, we are aware of only a single lockstep concept [2] which is published, was practically implemented, and validated based on a realistic fault profile, while many other publications utilize validation using synthetic tests and models only (see Section III). Practical implementation and the possibility to compare an implementation's performance to literature, however, is seen as a prerequisite by users in industry to consider an FT concept sufficiently mature for implementation. For space industry applications, this then enables further testing of a hardware/software system, e.g., in our use-case, radiation testing and on-orbit demonstration with an on-board computer (OBC) prototype. However, most proposed lockstep concepts, however, are not practically validated today. This has resulted in a gap between theory and application, with industry often dismissing software-implemented FT concepts due to a lack of maturity and an assumed tendency to ignore practical implementation obstacles. Preventing the application of the results of an entire field of research, dependable computing using software measures, for an entire industry segment even though there would be a technological need to do just that.

Contribution:

In this paper, we show how realistic and systematic validation of software-implemented FT concepts can be conducted using ISA-level fault injection for space applications, and fields with a similar fault profile, e.g., critical and irradiated

environments. We do so based on the fault injection campaign we carried out to validate a novel thread-level coarse grain lockstep concept we developed for space applications. This paper includes not only concept validation but is meant as a template for other researchers who wish to validate their own software-implemented FT concepts. We provide a detailed description of the fault profile in the space environment, and a through description of the utilized tools and scripts, which have been made available to the public open source. Thereby, we hope increase acceptance of software implemented FT concepts by industry, but also to increase the share of concepts that are validated in a practically meaningful way. We also believe it is of great importance to offer a second set of validation results to allow fellow researchers to compare their forthcoming results to more than just one single paper. As a single set of data points is insufficient to judge the performance and effectiveness of the entire coarse-grain lockstep concept class. The results of our fault-injection campaign are positive, and resulted in a variety of lessons learned and allowed us to develop a better understanding of our lockstep implementation's behavior.

Paper Organization:

In the next section, we describe our architecture's intended operating environment, design constraints, and the physical fault model. We then discuss how these challenges are handled today in the industry, and outline which solutions currently are available to the space industry and spacecraft designers in academia. A brief overview of our multistage FT architecture and an introduction to our coarse-grain lockstep approach is provided in Section IV. For an RTOS implementation of this approach, we then develop a fault model in Section V, and analyze which testing techniques are available to verify our approach VI. Having chosen the most suitable technique for our use-case, in Section VII we describe our an automated test toolchain which we use to systematically conduct fault injection campaigns against an RTOS implementation of our approach. The experiments are conducted using system emulation of a SoC closely resembling compute tiles of our target MPSoC, and we utilize a set of predefined fault-templates to simulate the different faults types in different components as described in Section VII-D. Subsequently the results of our fault injection campaign are presented in Section VIII, and we compare them to related work in IX. Before presenting conclusions, the pitfalls we discovered while preparing and lessons learned from conducting our fault injection experiments are discussed in Section X.

II. The Space Environment & Radiation

The form factor constraints aboard miniaturized satellites [3] and the drastically different fault model [4] 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 [5] 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 [4]. 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 [6] show drastically better than expected TID performance [7] and resistance to latch-up in contrast to projections based on technology scaling [8].

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 [9], [10], 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 cause mission failure. Risk-acceptance at this scale is a viable approach for low-priority science and missions with brief duration. This is not possible for for complex, critical, and long-term missions with a one or multiple scientific or commercial objective.

Most nanosatellite hardware development efforts today are more comparable to 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 fault detection, isolation and recovery (FDIR) potential [11].

III. Background & Related Work

FT is traditionally implemented through circuit-, RTL-, core-, and OBC-level majority voting [12]–[14] 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 [15], [16]. Software takes no active part in fault mitigation within such systems, as faults are suppressed at the circuit level and usually only indicated using hardware fault counters, without a direct feedback between fault-mitigation and the OS state.

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. Relevant radiation tests have been conducted for the FPGAs utilized in our project, among others by Lee et al. in [17] and Berg et al. in [7], or are currently ongoing (Lange et al. [18]). Tests for further components such as memory and supervisor-μCs 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 yield device-specific data, which enabling us to estimate fault frequencies, types, and effects on the FPGA on which our MPSoC is implemented. We require this information to choose appropriate checkpoint frequencies and frame times for our coarse-grain lockstep approach, but by itself, radiation testing does not allow an assessment of the capabilities of software-implemented FT measures.

Prior research on software FT often utilizes simple fault models, considering faults to be isolated, side effect free and local to an individual application thread [19] or purely transient [2], [20]. Many practical application obstacles could be uncovered and resolved before publication by implementing these concepts [21]. However, implementation and fault injection into a concept implementation is time consuming [22], and often requires also custom hardware and new software implementations, as outlined among others by Sangchoolie et al. [23]. Especially fault injection for entire OS instances is non-trivial [24], as thorough 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. This is a viable approach for validating FT concepts directed towards, e.g., yield maximization [29] and aging [30], but not for validating software-implemented FT measures operating in a critical environment.

In this contribution, we therefore conduct systematic validation of our coarse-grain lockstep approach using fault injection to verify the effectiveness and efficiency 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 of a fault to result in a crash or another failure requiring replacement using spare resources. This information is essential to define an appropriate checkpoint frequency for the lockstep, 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.

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

IV. Our Hybrid FT Architecture

The high-level logic including each FT stage in our architecture is depicted in Figure 1, and consists of three interlinked fault mitigation stages implemented across the embedded stack. The coarse-grain lockstep together with its forward error correction mechanisms we are validating in this publication, is the first FT stage in our architecture [Anonymous 2017], and provides fault-detection capacity for the others.

Stage 1 implements forward error correction and utilizes coarse-grain lockstep to generate a distributed majority decision regarding the integrity of the software replicas run on a set of isolated, weakly coupled processor cores. Fault detection is facilitated through application-provided callback functions, requiring no modifications to an application or knowledge about intrinsics. Faults are resolved through state re-synchronization and thread migration to processors with spare processing capacity. In this contribution, we conduct a validation of the mechanisms of Stage 1.

Stage 2 recovers defective compute 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

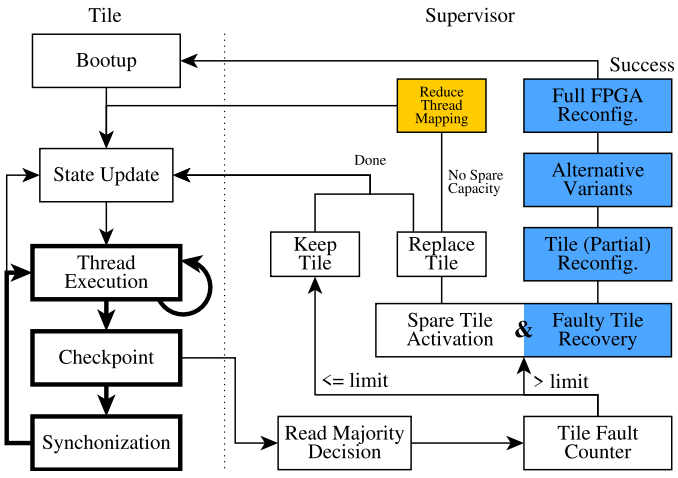

Fig. 1: Stage 1 (white) assures fault detection (bold) and fault coverage, Stage 2 (blue) and 3 (yellow) counter resource exhaustion and adapt to reduced system resources.

tiles affected by upsets in tile logic, and to cover permanent faults using alternative configuration variants. Individual stand-alone concepts utilizing comparable mechanisms as Stage 2 have been researched and validated e.g. by Nguyen et al. in [31] and D. Cozzi in [32]. FDIR concepts using scrubbing and partial reconfiguration have been demonstrated on-orbit [33].

Stage 3 [Anonymous 2018a] is activated when insufficient processing time in the remaining intact processor cores is available due to accumulating permanent faults. It **re-allocates processing time** between application threads to maintain a functioning system, to allow the system to gracefully age for a strongly degraded system. To do so, it utilizes thread-level **mixed criticality** inherent to data processing aboard satellites, assuring sufficient compute resources are available to high-criticality applications by sacrificing performance of lower-criticality threads.

We deploy this architecture on an FPGA and developed an MPSoC design [Anonymous 2018b] to best exploit the FT capabilities of each stage. Each processor core exists in a separate compartment, a tile, together with peripheral-IP (e.g., interrupt controller, timer, etc.) and interface cores, and has a supervisor access port. It also contains a dedicated on-chip memory slice, which is used to expose tile-internal OS state information to the rest of the system. While peripherals 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 is insufficient for modern data handling applications and platform control, tiles utilize a shared set of DDR and SPI controllers via an AXI interconnect in cross-bar mode. These controllers are implemented redundantly to enable fail-over, safeguard from SEFIs, and allow interleaved access to reduce congestion. On a Xilinx Kintex Ultrascale+ XCKU5P FPGA, a quad-core design outfitted with Microblaze processor cores and standard CubeSat interfaces (SPI, I2C, UART, and GPIO), results in modest resource utilization (28% LUTs, 33% BRAMs, 16% FFs, 5% DSPs) and 1.92W total power consumption.

V. Practical Fault Model Definition

To properly validate software-implemented FT measures, knowledge of the relevant target environment is required, as well as the physical fault scenario. Hence, in the remainder of this section, we reduce the physical fault scenario of our operating environment to a practical fault model applicable to fault-injection. This enables us to subsequently determine the most suitable fault injection technique as well as to build a concrete test-space for our fault injection campaign.

In the space environment, the main challenge towards an MPSoC is radiation, though the precise effects of radiation on semiconductors induced by a fault however varies. The effect of a fault depends on the particular effected chip-region, logic, and microfabrication technology used [4]. Today, the characterization of these effects is of major concern when implementing traditional hardware-FT based systems, and the only practical way to evaluate them is radiation testing. However, radiation testing can occur only at a very late stage in development, and the results may vary even for identical chip-designs manufactured in different fabrication lines or fabs. This form of testing effectively yields heritage and increases a system's technology readiness level, instead of verifying the effectiveness of a specific FT mechanism. In practice, a software-implemented FT concept thus has be to validated especially considering also permanent faults and intermediate faults, which are neither transient nor truly permanent. In contrast, literature today mostly considers transient faults, which is unsuitable for applications intended for irradiated environments.

In our case, we validate how well our lockstep implementation can actually detect faults that have occurred. We can do this through injection of bit-flips and new-value fault injection. Random fuzzing or type-fault injection are widely used for finding exploits and vulnerabilities in software, as well as logic bugs, but are not useful for our purposes due to the different physical fault scenario. Hence, proper validation must also include systematic testing, which is even theoretically impossible using system-level testing with hardware. Thus, software must be tested separately and systematically, before actually conducting system-level testing. Therefore, we must consider the actual effect and impact of faults on the system from a programmatic perspective, which neither statistical models nor system-level radiation testing can deliver.

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 within one of our MPSoC's tiles:

- Data corruption associated with access to main memory, caches, registers and scratchpad memory due to non-correctable ECC words caused by SEEs.
- Bit upsets, new-value, and zero-value faults due to SEEs and SEFIs in address and control logic of peripheral IP due.
- Incorrect or non-execution of instructions in the processor pipeline during the entire sequence of processing, i.e. from instruction fetch, execute to write-back, as well as incorrect decoding of instructions and execution of different instructions with the given parameters.
- Control-flow deviations and data corruption due to failure of interfaces and tile I/O peripherals, due to faults in controller logic of FPGA's I/O components.

A broad variety of synthetic, theoretical failure types are well described in literature, e.g. in [25]. In practice these do emerge as one of the described fault types. As discussed among others e.g. in [34] most of these synthetic failure modes [25] actually emerge as one of the aforementioned effects. To validate the fault-detection and mitigation capabilities of our lockstep to radiation effects, we are only interested in the practical effects of a fault, not its theoretical origin.

Radiation can induce subtle effects into the FPGA itself, and may affect the OBC at a larger scale (e.g., full component failure or reset) [7]. Such faults either emerge disguised as one of the aforementioned ones, or are covered by Stage 2

(FPGA fabric corruption) and detectable by the supervisor. These are thus beyond the scope of what we need to validate in this contribution. As such, these faults are either fatal to a tile, therefore directly detectable by other tiles by majority decision, or cannot be handled within a single-chip MPSoC. To overcome this system-level limitation, FPGA reconfiguration and other system-level measures are required and implemented as part of Stage 2, and do not need to be considered separately when testing pure software as discussed also by Sangchoolie et al. in [23].

VI. Fault-Injection Technique Survey

As our approach is based upon an MPSoC on FPGA, fault injection using netlist simulation [35] or directly into the FPGA [32], [36] could be facilitated with comparably[2] little development effort, as we already utilize a development-board based MPSoC design implementation. Therefore, this technique would grant maximum control over the type and effect of faults and the simulation would be based on a design which closely corresponds to the real MPSoC. Several proprietary partially [32], [36], [37] and fully automated test frameworks [38] as well as commercial applications [35] have been developed for this purpose. Unfortunately, netlist simulation is computationally disproportionately expensive, 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. The same applies to fault injection in a live system (e.g. using OpenOCD), or into an FPGA running our MPSoC, but this approach is time too consuming and potentially destructive [36].

Faults could also be injected via widely available standard debug tools (e.g., GDB) into software running entirely in userland. This is only representative for simple userland applications [2], the effects of faults on an actual OS cannot be simulated [39]. Furthermore, validation of embedded software for low-power ARM or RISC-V SoCs using desktop-grade ia32/amd64 hosts may bias the outcome of a fault injection experiment. Fault injection into kernel functionality emulated in userland can result in a different run-time behavior, and will therefore not always produce meaningful validation results [25], in contrast to performance estimation, where this approach can very well deliver worst-case performance estimates. Debugger based fault injection into a virtual machine can alleviate these constraints by allowing an entire OS to be tested, but suffers performance limitations due to protocol overhead. Furthermore, permanent fault injection into components other than memory and the current execution context are infeasible due to the limited means of interactions of a debugger on the virtual machine itself [24]. In consequence, the kind and type of faults which can be simulated using an external debugger are significantly constrained [25], though can be facilitated using standard open-source tools.

Fault injection using system emulation can combine many of the advantages of the aforementioned techniques, without being constrained by the limited capabilities of a debugger interface regarding the potentially injectable fault types. In some prior research, computer architectures were simulated using SystemC to demonstrate architectural features, and it could also be used as compromise between fault injection using netlist simulation and pure userland-software based debugging. 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. Therefore, it is only viable for fault injection into very simple process designs executing less complex software [40].

ISA-level fault injection has been shown powerful and efficient for conducting black- and grey-box fault injection [23], though the tools discussed today in relevant publications are mostly proprietary to individual research groups, largely experimental, and usually not open sourced [41]. Virtualization assisted emulation, instead, allows faults to be injected into pre-existing emulated hardware and SoCs using standard tools, while being computationally comparably cheap and requiring no further changes to a victim application. Several test frameworks implementing this approach have emerged in recent years, though most are still custom tailored for specific use-cases or have not been released to the public [22] and are thus not relevant for our purposes. Notable exceptions here are the two open source frameworks FAIL [34] and FIES [42], which are publicly and freely available as open source software and comparably mature. Hence, we use this fault injection technique to systematically validate our FT approach using an automated test toolchain.

VII. The Fault Injection Setup

To provide systematic fault coverage, manual fault injection or fault injection relying upon manual binary introspection are unsuitable. Instead, an automated test setup is needed that can then be executed continuously and executed in parallel to achieve the desired test coverage without cross-effects. In this section, we therefore describe how such a test setup can be realized with limited development manpower, and pre-existing standard software based on our own setup.

A. Fault Injection Tool Selection

The available ISA-level FI tools are not functionally equivalent, and differ regarding the target environment, test setup and intended test subject scope, and the way in which they inject faults. FAIL utilizes a powerful C++ based test controller for thoroughly analyzing small binaries in a fully automated test campaign. While the test itself is therefore fully automatic, the development of a test-specific controller application requires deep knowledge of victim binary intrinsincs and program structure, which must be obtained manually. The development of FAIL is mainly focused on the Intel platform, while ARM is

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

available via GEM5 for a virtual target or through (potentially destructive) fault injection via JTAG into silicon[3] [43].

FIES[4] by H"oller et al. [42] 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. This tool alone does not support automated test campaigns, but allows rule-based and systematic fault injection into opaque binaries during each run. Its fault injection engine utilizes a fault library which can be generated automatically using compiler-toolchain functionality and instruction and memory access traces. It can therefore more efficiently handle testing a full OS, without requiring 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 build around FIES.

B. Test Campaign Setup

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

1) Obtain the victim application's process state, results and correct lockstep checksums for each payload application. We run the emulation without fault injection and tracing, outputting the application and OS state for comparison during later steps. This allows us to e.g., include additional debug output or otherwise alter the victim-binary's code for our golden run. Thereby, we can obtain a correct victim OS state without distorting the actual golden-run.
2) Execute a golden run and generate traces of the process counter and executed instructions, register access and memory access with an unmodified binary, using the same parameters as in the previous step. Our lockstep implementation does not require strict timing determinism. It only requires a comparable level of work to be executed between checkpoints allows us to avoiding potential non-determinism due to code-changes in the first step. When validating more timing-sensitive algorithms however, special care must be taken to assure the golden run and fault injection runs are equivalent [41], [42].
3) Filter the generated 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, generating the actual test-campaign input trace.
4) For each address and occurrence, we generate a fault definition library and launch an instance of FIES.
5) For each run, we 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 sqlite database.

Steps 1-3 are executed once at the beginning of a test campaign, whereas steps 4 and 5 are computationally comparably expensive, and but be executed in parallel by splitting the processed traces. Result databases from different systems can be merged and combined, and each test record includes information about the precise injected fault. Besides collecting and interpreting the results of a fault injection run, we also retain tile state information to enable manual analysis in the future if necessary. This includes a tile human readable output to each tiles' serial port, CPU and qemu processor context dumps, as well as the logs generated by FIES during the fault injection, as well as its exit code.

In the process of developing our 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 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.

During development of our toolchain, we also conducted fault injection manually by targeting specific locations in the application binary structure. There, we chose interesting data and logic which could cause an incorrect application state, 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 later, the fault-injection toolchain and changes made to FIES.

C. Target Implementation and Payload

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. A simplified function flow graph is depicted in Figure 2

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 the test time overhead.

As payload application, we utilized two applications:

- The ESA Next Generation DSP benchmark[6] run as POSIX threads within RTEMS. This is a space-industry standard benchmark application used to measure and compare system performance.
- An application alike the NASA/James Webb Space Telescope Mid-Infrared Instrument readout software[7] [44].

While this choice represents today's space-borne computing workloads well, test campaigns for other application fields should utilize representative software for that field. If no specific target application code is available, synthetic algorithm

[3]Due to constant reboots required for fault injection at the OS-level and the possibility of faults in device drivers causing damage in the target device.

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

[5]We made our changes to FIES available in the form of the a rebased QEMU-git form at https://redacted.for.peer.review.

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

[7]See https://github.com/spacetelescope

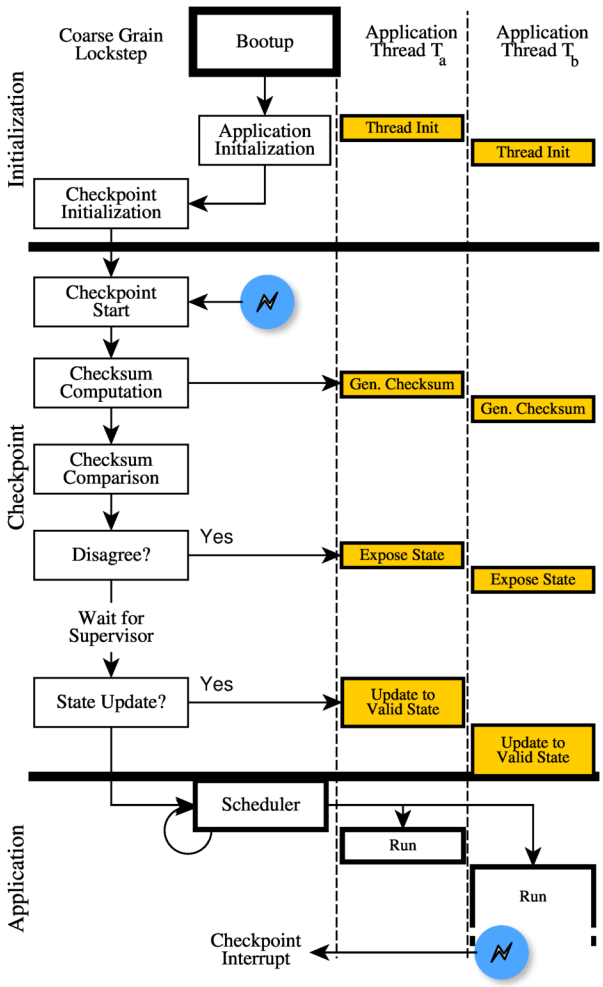

Fig. 2: The execution cycle of our coarse-grain lockstep implementation on a tile. Payload application callbacks are depicted in yellow, checkpoint trigger timers in blue. Faults are injected after initialization.

suites such as the SPEC performance tests[8] can be utilized at a loss of realism due to the limited scope and low complexity.

D. Test Space and Target Components

Using our pipeline, we developed a set of template fault definitions which our fault injection toolchain utilizes to generate suitable fault definitions based on the aforementioned program traces. In practice, choosing the right test-space for a practical OS-scale implementation is non-trivial. Today, the contrast is large between what is described as ideal in literature for testing software [45], what is technically feasible today with realistic financial and time investment, and what can be achieved for system-level testing in industry [46]. 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 high cost. Besides test coverage, our architecture has to cope with not merely transients, but also radiation-induced permanent faults and SEFIs, as these are common in modern memories and processor components flying in space.

Transient Fault Injection: Transients are injected as bit-flips and new-value errors into registers and the processor pipeline using the program counter as trigger. Simple time triggered injection is insufficient, as the relevant available tools do not assure clock-cycle accurate timing, making test results unpredictable. For instructions which are visited more than once, we trigger faults after the n-th occurrence, which is enabled by an extension of the FIES framework's fault definition language. Faults were injected also into memory access operations based on physical memory addresses, thereby simulating non-correctable upsets in ECC protected words in caches and main memory, as well as 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 replace the address of the operation, instead of just injecting single bit-flips.

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

Functional Interrupts and Intermittent Faults: SEFIs in different functional units of a tile may also induce fault-effects which are neither transient nor permanent. 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[9]. 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 precise actual timing and detailed interruption behavior SEFIs in processor logic and FPGA fabric.

Fault Placement during Execution: After executing bring-up code and OS initialization, our victim binary executes payload software for 3 lockstep cycles, and then terminates. We chose a frame time of 2 seconds as interval between checkpoints, 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 [7], [17] in preliminary information obtained from Ultrascale+ FPGAs [18]. With our victim binary, execution during of a golden run takes approximately 7 seconds of guest-virtual time, which on our test system is equivalent to approximately

[8]see https://www.spec.org/cpu

[9]The clock speed emulated by QEMU for the Zync MPSoC.

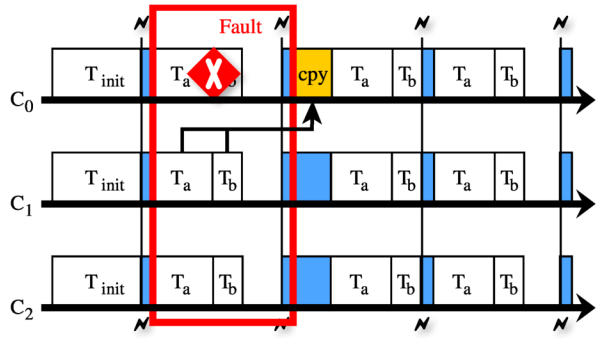

Fig. 3: The experiment sequence and fault placement for a tile. Fault are injected during the red-outlined time period on processor tile C_0.

30 seconds of host-time. In case the experiment does not terminate in time, e.g., due to control flow corruption, the experiment is terminated by the toolchain after 45 seconds (allowing one additional checkpoint to be processed). FIES can also be configured to end an injection run after executing given number of instructions (e.g. 10 times the number of instructions executed in the golden run).

The test sequence is depicted in Figure 3, and faults are injected after the first and until to the end of the second checkpoint. This allows faults to propagate within the system, to corrupt the OS and application state, 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 tile. 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 bootup/shutdown sequences of a tile are not relevant to validating our lockstep.

Limitations: We chose the duration of fault injection run to allow our victim binary to exhibit the entire FDIR circle, while allowing sufficient test coverage. However, this does not allow detection and observation of dormant or latent faults, e.g. such 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 in a system.

Theoretically, it would be possible to inject faults in QEMU's virtual hardware directly as well. However, this is not supported in FIES today in a scripted manner, requiring instead source-code modification for each device in QEMU. We can elevate this limitation by simulating such effects through data corruption during memory access, and we can inject faults into device address space.

VIII. Results & Interpretation

Table I contains raw results of our fault injection experiments. 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 mechanisms (e.g., resulting in false comparison or incorrectly generated checksums from correct data) were rare due to the minimal code and data footprint of the lockstep.

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

Our coarse grain lockstep implementation most importantly contributed fault-detection to the system, whereas the state synchronization functionality serves to reduce the amount of reboots needed to restore the state of each tile. In practice, its fault-detection strength depends on both the frequency at which checkpoints are execute (frame-time) and the likelihood that faults can be covered and corrected. Hence, we analyzed how rapidly a tile itself can detect faults in Figure 4.

	Fault	Detectable by		Recovery	Observed Effect per Fault Type		
Impact	Detectable	victim tile	other tiles	through	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%

TABLE I: Fault injection experiment results for our RTOS implementation divided into transient, permanent, and intermittent faults. Notice that our implementation can not detect silent data corruption with no impact on the thread state. Certain masked faults affecting OS data structures could be detected through erasure coding, while memory protection and virtual memory would allow us to detect misdirected memory access caused by faults. Neither measures is in place in our proof-of-concept.

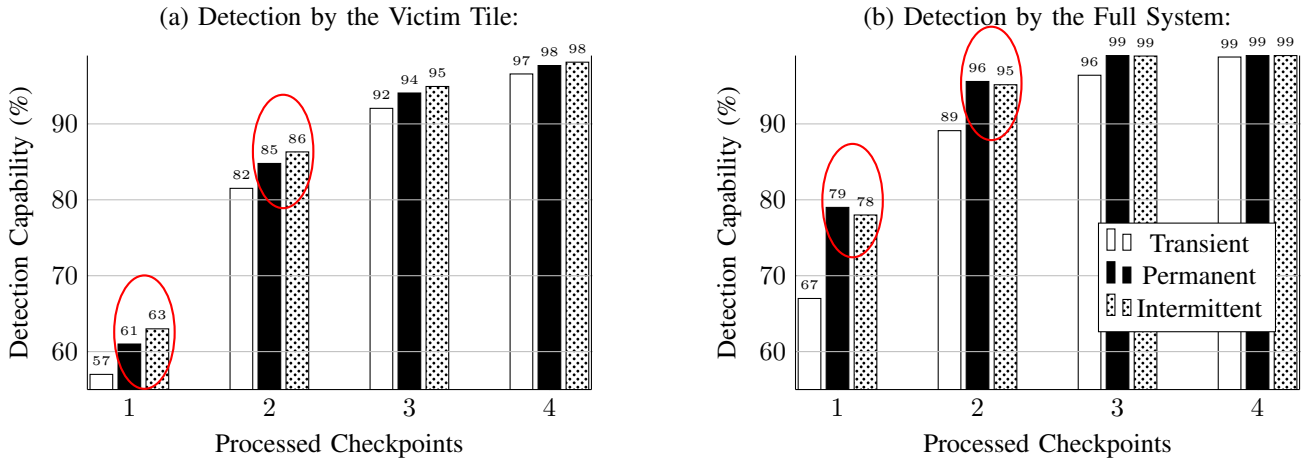

Fig. 4: Payload application and state corrupting fault detection chance of a single tile for different fault types after a given number of execute checkpoints. Notice that intermittent faults are more likely to be detected than permanent faults by the affected tile itself, which is counter intuitive. This is due to the increased percentage of faults that are fatal for a tile, and the system as a whole can detect permanent faults with higher likelihood.

The fault injection campaign shows that there is indeed a measurable different in behavior between transient and permanent faults. As expected, we measures that permanent faults would be more likely detectable than transients, due to the more severe impact of faults in general. However, we also expected permanent faults to be easier detectable by a tile than SEFIs in the same way (see Figure 4a). This, however, was not the case as the increased likelyhood of permanent faults resulting in crashes and the higher percentage of non-fatal state corruption faults due to SEFIs actually make fault detection within the affected tile more likely for SEFIs. For permanent faults, a larger percentage of faults results in a crash, which can no longer be detected by the affected tile. These deductions underline the importance of conducting validation not only using transient faults, but also using permanent and intermittent faults.

The effects of a fault with higher likely also be detected through majority decision by the rest of the system, as the MPSoC as a whole can also detect crashes of an entire tile or lockstep mechanism failure, as shown in Figure 4b. In Figure 5, we therefore provide a direct comparison between self- and consensus-based fault detection for transients, permanent and intermittent faults. While the results for transient faults again match our expectations, for permanent faults and SEFIs, the initial fault detection capability for the full MPSoC even with only a single executed checkpoint is drastically better than for self-detection. Here, a fault detection chance of near 79% and 78% during the first checkpoints also implies a near certain fault detection likelihood during the second checkpoint, see Figure 5b and c. In contrast, for self detection, faults can be detected after with 57%, 61% and 63% during the first checkpoint after fault occurrence and near certain detection only being achieved after three checkpoints.

When designing our lockstep concept, we considered fluctuations in thread-assignment to the MPSoC's tiles due to crashes and reboots of individual tiles critical. Worst-case benchmark results showed that frequent crashes of tiles could degrade performance of the system by between 9% and 26% for high checkpoint frequencies and brief frame times. Based on our experiments, we find comparably few faults, between 11% and 20%, would inducing crash and lockstep-failures encouraging. Hence, even assuming that faults would occur in every single checkpoint period, only few faults would require a reboot to be resolved. Hence, our lockstep implementation can provides the necessary degree of voter stability to making thread synchronization rare, and application reassignments between tiles an exception.

A major share of injected faults that resulted in no observable effect on our implementation may indeed be masked and require no measures to be taken, as they may have no impact on the application state [47]. This is a limitation of our fault injection toolchain, as faults are also injected into registers and memory which may be overwritten by subsequent instructions, or faults that cause self-masking control flow deviations. Such situations occur e.g., due to faults in branch or comparison instructions triggering the same iteration of a loop more than once. They have no practical impact on the application state while, and also cause only minor timing deviations which do not impact the work conducted until to the next checkpoint.

IX. Comparison to Literature

To place these results in context with results from other lockstep concepts, we sought to compare our results to literature in order to provide a second point of reference for verification. Unfortunately, few coarse-grain lockstep concepts have been implemented in practice and tested using means beyond modeling. At the time of writing, we are aware of only one publicly released validation report by Dobel et al. [2] considering practical fault injection with real software and faults, instead of statistical estimation.

When directly comparing our results to Dobel et al.'s *transient* fault injection report [2], the share of faults causing application thread and OS crashes measured with our approach is increased. For transient faults, this can at least in part

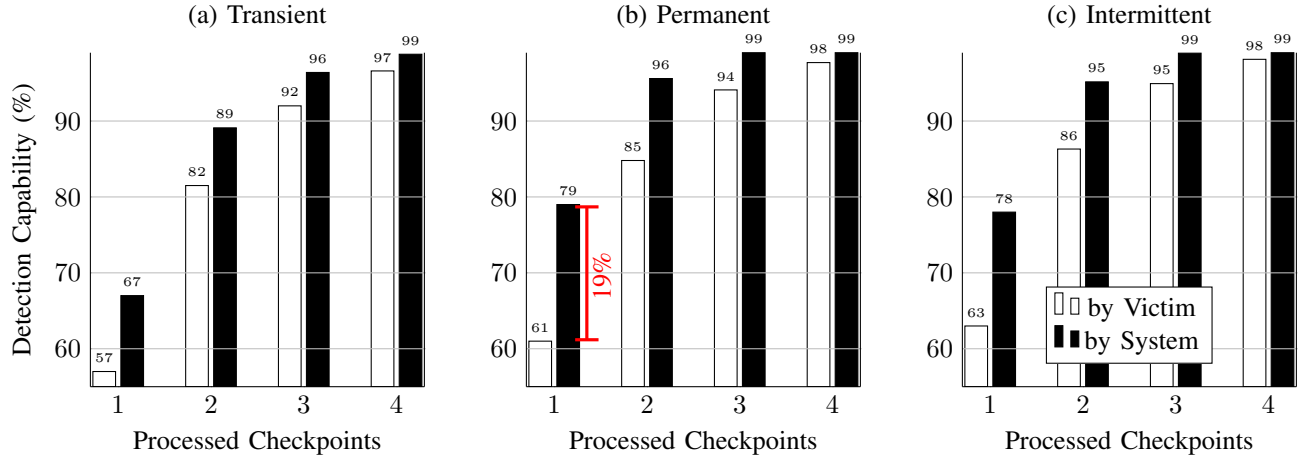

Fig. 5: Comparison of the fault detection capabilities of an individual tile and the by MPSoC through majority decision. The full system can also detect a crash of the OS instance running on a tile, and malfunctions in the lockstep logic.

be explained with the different capabilities of Dobel et al.'s proposed lockstep mechanisms. In their contribution, lockstep is facilitated through application intrusive function call hooking. Thereby, they can offer more fine-grained protection than our approach, but require considerable code overhead and constrain the concept's application to one specific OS. Dobel et al. also consider their fault injection measurements overly optimistic, as they utilized 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. also assume the OS to be guaranteed fault-free, we instead inject faults into a full OS including POSIX libraries with payload applications.

The measured detection differences are consistent across all effect categories: we measure a higher amount of masked faults, a decreased amount of detected state deviations, and an increased amount of crashes with our approach. In light of this bias, we consider our results are in line with Dobel et al.'s, and our lockstep implementation to function as desired.

X. Discussion & Lessons Learned

Fault injection today can be conducted for different reasons, e.g. for detecting exploits in software, memory leaks, or to assure code coverage. However, FI to validate the correction functionality of a fault-detection and lockstep technique in regards to practical faults is very different from FI for security evaluation. Applying the same assumptions as for other fault injection types does not result in proper validation. The choice of fault injection techniques, target implementations, and the used payload software all directly influence the obtained results, and validation using an overly simplistic implementation or too simple payload software will bias the results obtained. Comparing our results to Dobel et al.'s underlines that it is important to conduct fault injection into a realistic concepts implementation with non-trivial payload software. Today, there is only one other publication available that aims to validate a coarse-grain lockstep in practice, and it is of paramount importance for future work in this field to compare to their mechanisms to more than just one single implementation.

Our coarse-grain lockstep can detect faults resulting in a crash or in corruption of the thread state, but currently is oblivious to silent data corruption in OS data structures and code. Currently, a tile's checkpoint handler can only directly derive a checksum for certain critical kernel data structures. However, the scope to which this is possible is limited and the computational cost high. Velasco et al. propose in [48] to apply erasure coding for critical OS data structures in software. The proposed concept is similar to code signing, and today widely used for tamper-proving of embedded devices and e.g. for secure boot. The availability of this functionality would allow our lockstep to also detect silent data corruption in rarely accessed OS structures and device drivers code and data.

When experimenting with different compiler flags, we found that faults injected in equivalent code segments of differently compiled binaries could result in different observed effects. We determined through introspection of the relevant target binary parts, that the changed behavior was caused due to specific compiler flags. Especially loop unrolling (GCC's `-funroll-loops` flag) had a particularly positive effect when injecting permanent and intermittent faults. In practice a compiler flattens the program structure, duplicating code segments instead of executing the same segment multiple times. Serrano Cases et al. in [49], [50] as well as Lins et al. in [51] have begun to explore these effects for improving reliability, but otherwise industry and literature today seem oblivious on this issue. Designers of software-FT measures 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 mechanisms as well as validation.

FIES originally offered no support for the THUMB instruction set. However, most OS kernels, many device drivers,

and even standard library functions mix THUMB and ARM instructions. Therefore, we added support for the THUMB and THUMB2 instruction sets to FIES, to assure consistent tracing and fault injection results. A jump between instruction sets without compiler-interwork yields an undefined instruction exception, as the opcode-encoding for ARM and THUMB instructions differs. This effectively prevents undetected, incorrect jumps in ARM/THUMB interwoven code segments. Due to the observed behavior during fault injection before we had added THUMB support, 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. In this could be facilitated in a similar manner as software diversity through compiler instrumentation or scripted, automated code transformation [52].

When designing our coarse grain lockstep measure, we were aware of two ways of inducing checkpoints: tile internally through timers and externally through interrupts. If timers are used, checkpoint initialization for each tile is independent, as tiles operate independently and do not direct interact with each other. 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 that interrupt induced checkpoints can have significant advantages. The timer-handling related logic is considerably more complex than the logic necessary for interrupt handling, and thus also more prone to faults. While a timer driven lockstep implementation requires interaction with several different subsystems and libraries of the underlying OS, the same RTOS implementation using interrupt-triggering can be very simple and thus more resilient.

XI. Conclusions

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

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

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

Our architecture now has to undergo radiation testing using a hardware prototype for full system-level testing. Systematic validation of our coarse-grain lockstep implementation is therefore an intermediate step. As the other parts of our architecture have been verified separately in related work, our test campaign represent the final step in validating our current development-board based proof-of-concept. The positive outcome of our test enables us to now produce a prototype OBC implementation, which then allows us to then subject it to laser fault injection, radiation testing, and trials on-orbit.

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

[4] J. R. Schwank, M. R. Shaneyfelt, and P. E. Dodd, "Radiation hardness assurance testing of microelectronic devices and integrated circuits: Radiation environments, physical mechanisms, and foundations for hardness assurance," *IEEE Transactions on Nuclear Science*, vol. 60, no. 3, 2013.

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

[6] L. A. Tambara, F. L. Kastensmidt, N. H. Medina, N. Added, V. A. Aguiar, F. Aguirre, E. L. Macchione, and M. A. Silveira, "Heavy ions induced single event upsets testing of the 28 nm xilinx zynq-7000 all programmable soc," in *Radiation Effects Data Workshop*. IEEE, 2015.

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

[8] M. Kochiyama, T. Sega, K. Hara, Y. Arai, T. Miyoshi, Y. Ikegami, S. Terada, Y. Unno, K. Fukuda, and M. Okihara, "Radiation effects in silicon-on-insulator transistors with back-gate control method fabricated with OKI semiconductor 0.20 μm FD-SOI technology," *Nuclear Instruments and Methods in Physics Research, Elsevier*, vol. 636, no. 1, 2011.

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

[10] 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.

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

[12] K. Reick, P. N. Sanda, S. Swaney, J. W. Kellington, M. Mack, M. Floyd, and D. Henderson, "Fault-tolerant design of the IBM Power6 microprocessor," *IEEE micro*, vol. 28, no. 2, 2008.

[13] M. Hijorth, M. Aberg, N.-J. Wessman, J. Andersson, R. Chevallier, R. Forsyth, R. Weigand, and L. Fossati, "GR740: Rad-hard quad-core LEON4FT system-on-chip," in *Eurospace DAta Systems in Aerospace (DASIA)*, 2015.

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

[15] X. Iturbe, B. Venu, E. Ozer, and S. Das, "A triple core lock-step ARM Cortex-R5 processor for safety-critical and ultra-reliable applications," in *IEEE DSN*, 2016.

[16] R. DeCoursey, R. Melton, and R. R. Estes, "Non-radiation hardened microprocessors in space-based remote sensing systems," in *Sensors, Systems, and Next-Generation Satellites X*. International Society for Optics and Photonics, 2006.

[17] D. S. Lee, G. R. Allen, G. Swift, M. Cannon, M. Wirthlin, J. S. George, R. Koga, and K. Huey, "Single-event characterization of the 20 nm Xilinx Kintex Ultrascale field-programmable gate array under heavy ion irradiation," in *Radiation Effects Data Workshop (REDW)*. IEEE, 2015.

[18] T. Lange, M. Glorieux, A. Evans, A.-D. In, T. Bonnoit, A. Dan, C. Boatella Polo, C. Urbina Ortega, V. Ferlet-Cavrois, M. Tali, and R. Garcia Alia, "Single event characterization of a Xilinx UltraScale+ MP-SoC FPGA," in *SpacE FPGA Users Workshop*, 2018, preliminary.

[19] A. Höller, T. Rauter, J. Iber, G. F. H. Macher, and C. J. Kreiner, "Software-based fault recovery via adaptive diversity for COTS multi-core processors," 2015, arXiv:1511.03528.

[20] P. Munk, M. S. Alhakeem, R. Lisicki, H. Parzyjegla, J. Richling, and H.-U. Heiß, "Toward a fault-tolerance framework for COTS many-core systems," in *IEEE EDCC*, 2015.

[21] U. Kretzschmar, J. Gomez-Cornejo, A. Astarloa, U. Bidarte, and J. Del Ser, "Synchronization of faulty processors in coarse-grained TMR protected partially reconfigurable FPGA designs," *Elsevier Reliability Engineering & System Safety*, 2016.

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

[23] B. Sangchoolie, R. Johansson, and J. Karlsson, "Light-weight techniques for improving the controllability and efficiency of isa-level fault injection tools," in *PRDC*. IEEE, 2017.

[24] D. Cotroneo, A. Lanzaro, R. Natella, and R. Barbosa, "Experimental analysis of binary-level software fault injection in complex software," in *EDCC*. IEEE, 2012.

[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, 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, S. Bekri, and H. Smei, "Fault-tolerant in embedded systems (MPSoC): Performance estimation and dynamic migration tasks," in *IEEE IDT*, 2016.

[28] Z. Al-bayati, J. Caplan, B. H. Meyer, and H. Zeng, "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 *ICCAD*. IEEE, 2010.

[30] L. Leem, H. Cho, J. Bau, Q. A. Jacobson, and S. Mitra, "ERSA: Error resilient system architecture for probabilistic applications," in *DATE*. EDAA, 2010.

[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," Ph.D. dissertation, 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 *IEEE Aerospace Conference*. IEEE, 2014.

[34] 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 *EDCC*. IEEE, 2015.

[35] 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.

[36] J. L. Nunes, T. Pecserke, J. C. Cunha, and M. Zenha-Rela, "Fired – fault injector for reconfigurable embedded devices," in *PRDC*. IEEE, 2015.

[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 *DFT*. IEEE, 2007.

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

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

[40] P. Lisherness and K.-T. T. Cheng, "Scemit: A systemc error and mutation injection tool," in *DAC*. ACM, 2010.

[41] M. Kooli, P. Benoit, G. Di Natale, L. Torres, and V. Sieh, "Fault injection tools based on virtual machines," in *Reconfigurable and Communication-Centric Systems-on-Chip (ReCoSoC), 2014 9th International Symposium on*. IEEE, 2014.

[42] A. Höller, G. Schönfelder, N. Kajtazovic, T. Rauter, and C. Kreiner, "FIES: a fault injection framework for the evaluation of self-tests for COTS-based safety-critical systems," in *MTV*. IEEE, 2014.

[43] J. Isaza-González, A. Serrano-Cases, F. Restrepo-Calle, S. Cuenca-Asensi, and A. Martínez-Álvarez, "Dependability evaluation of cots microprocessors via on-chip debugging facilities," in *IEEE LATS*, 2016.

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

[45] R. Natella, S. Winter, D. Cotroneo, and N. Suri, "Analyzing the effects of bugs on software interfaces," *IEEE Transactions on Software Engineering*, 2018.

[46] R. L. Pease, A. H. Johnston, and J. L. Azarewicz, "Radiation testing of semiconductor devices for space electronics," *Proceedings of the IEEE*, vol. 76, no. 11, pp. 1510–1526, 1988.

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

[48] A. Velasco, B. Montruccio, and M. Rebaudengo, "A hardening approach for the scheduler's kernel data structures," in *CompSpace at ARCS2017*, 2017.

[49] A. Serrano-Cases, Y. Morilla, P. Martın-Holgado, S. Cuenca-Asensi, and A. Martınez-Álvarez, "Automatic compiler-guided reliability improvement of embedded processors under proton irradiation," in *RADECS*. IEEE, 2018.

[50] A. Serrano-Cases, J. Isaza-González, S. Cuenca-Asensi, and A. Martínez-Álvarez, "On the influence of compiler optimizations in the fault tolerance of embedded systems," in *IOLTS*. IEEE, 2016.

[51] F. M. Lins, L. A. Tambara, F. L. Kastensmidt, and P. Rech, "Register file criticality and compiler optimization effects on embedded microprocessor reliability," *IEEE Transactions on Nuclear Science*, 2017.

[52] P. Larsen, A. Homescu, S. Brunthaler, and M. Franz, "Sok: Automated software diversity," in *2014 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2014, pp. 276–291.

[Anonymous 2017] Closely related publication describing a multistage fault tolerance architecture and MPSoC design for Spacecraft. Published, but details omitted as any citation to this work would break double-blind reviewing, and we are currently (unfortunately) the only ones in this field publishing papers on the results to the public. IEEE, 2017

[Anonymous 2018a] In Press. Closely related publication describing the dynamic fault tolerance and resource pooling through software measures. Details omitted for double-blind reviewing. IEEE, 2018

[Anonymous 2018b] In Press. Closely related publication describing an FPGA-based MPSoC implementation for satellite computing. Details omitted for double-blind reviewing. IEEE, 2018