Fault-tolerant satellite computing with modern semiconductors

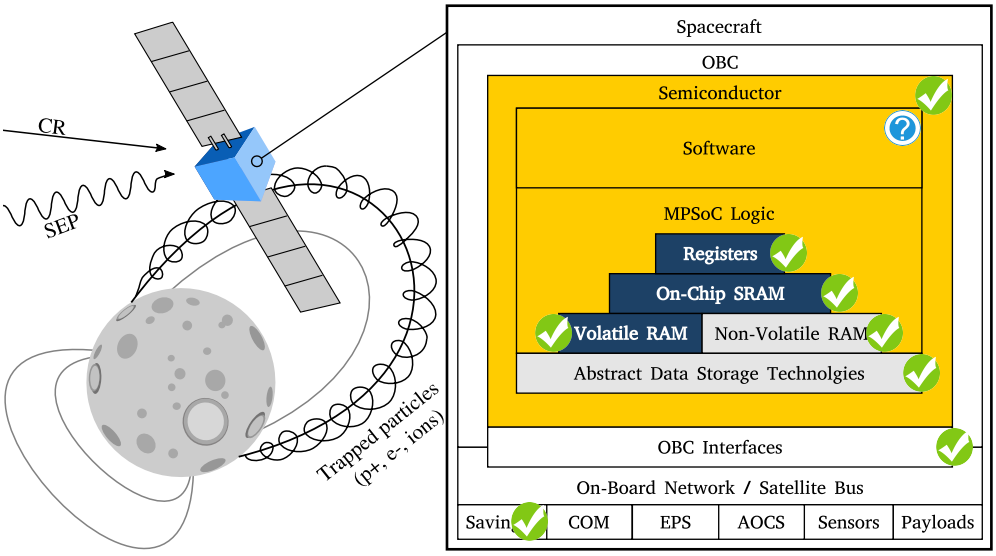
Fuchs, C.M.

Chapter 8

Validating Software-Implemented Fault Tolerance

Systematic Fault Injection

In this chapter, we test and validate the software-mechanisms that are the foundation of our fault tolerance architecture to address RQ5. Therefore, we conducted a fault-injection campaign through system emulation with QEMU into a ARMv7a-SoC matching our architecture target ARM's Cortex-A53. Our results show that our lockstep implementation is effective and efficient for providing FDIR within our system, and the thread-level coarse grain lockstep's performance meets our requirements. To place our results into context, we compared them to literature and discuss lessons learned and knowledge obtained throughout our fault injection campaign.



8.1 Introduction

Modern embedded technology is a driving factor in satellite miniaturization, which today enables an entire class of smaller, lighter, and cheaper class of spacecraft. These 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. They are drivers of a massive boom in satellite launches, new scientific and commercial space missions, laying the foundation for a rapidly evolving new space industry. However, these spacecraft 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 traditional semiconductors manufactured in technology nodes with a large feature size. 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 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 [2], which caused usually by design failures and effects induced by the space environment, e.g., [296].

Therefore, we developed a non-intrusive, flexible, hybrid hardware/software architecture (see Chapter 4) to assure FT with commercial-off-the-shelf (COTS) mobile-market technology based on an FPGA-implemented MPSoC design. Our architecture utilizes multiple FT measures across the embedded stack, and runs software in coarse-grain thread-level lockstep to assure computation correctness through replication. It can offer strong fault coverage without relying upon any space-proprietary logic, custom processor cores, or other radiation-hardening measures in hardware.

The utilized lockstep concept facilitates 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. As our lockstep is part of the operating system kernel, system-level fault injection and application-level testing do 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, very few coarse-grain lockstep concepts have been implemented and validated in this way. Most concepts described in academic publications today, instead are validated only using mathematical models only, but were not actually implemented or practically validated.

At the time of writing the 2018 – 2019 period, careful study of journals and conference proceedings yields only a single coarse-grain lockstep concept [199] that was practically implemented, and validated based on a realistic fault profile. Practical implementation and the possibility to compare an implementation's performance to

literature, however, is seen as a prerequisite by industrial users to consider an FT concept mature enough for practical application. This situation has resulted in a gap between theory and application, with industry often dismissing software-implemented FT concepts due to a (perceived?) lack of maturity and an (assumed?) tendency to ignore practical implementation obstacles. The research results of an entire field of research, dependable computing through software measures, are thus practically barred from application for an entire industry segment even though there would be a pressing technological need and a lack of viable alternatives. For critical applications like in the space industry, practical concept validation is then just the first of many validation and testing steps: eventually system-level testing is conducted with a hardware/software prototype. For space application, this prototype is then subjected to radiation testing followed by on-orbit demonstration.

8.1.1 Contributions

In this chapter, we show how software-implemented FT concepts can be validated for space applications in a realistic and representative manner, and fields with a similar fault profile, e.g., critical and irradiated environments. We do so by example of a fault-injection campaign we conducted to validate a novel thread-level coarse grain lockstep concept we developed for space applications, described in detail in Chapter 4. We utilize ISA-level fault injection into an ARM Cortex-A system through virtualization, and fault injection into a 3-core SystemC-implemented MPSoC. This chapter 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. Thereby, we hope to increase acceptance of software implemented FT concepts by industry, and the share of concepts which are validated in a practically meaningful way.

A single set of data points is insufficient to judge the performance and effectiveness of the entire coarse-grain lockstep concept class. Thus, 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. We document a variety of lessons learned as part of this campaign, which have allowed us to develop a better understand the practical behavior and protective properties of coarse-grained lockstep in critical systems.

Few software-implemented FT concepts proposed today have been implemented, and only a handful have been validated in a realistic and meaningful way. Therefore this chapter serves as practical guide for fellow researchers that can be used as walk-through to make proper testing of fault tolerance techniques a less challenging and time consuming task in an academic environment. The strategy which we describe throughout the remainder of this chapter is depicted in Figure 56, and described briefly below.

8.1.2 Chapter Organization

In the next section, we discuss how the challenges of the space environment described in Chapter 3 are met today in the industry, outline which solutions currently are available, and how these are tested. We then derive a practical fault model for an RTOS implementation of this approach (Section 8.4), and analyze which testing techniques

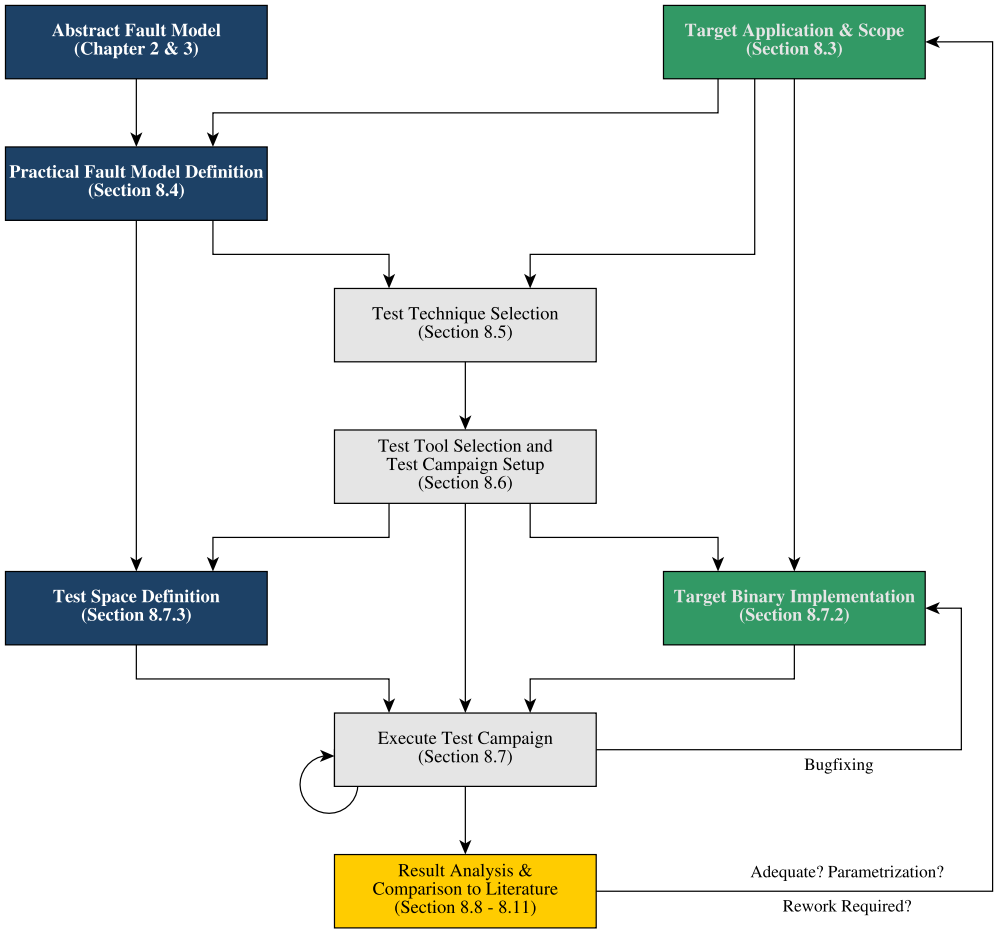


Figure 56: The top-down step-by-step testing strategy described in this chapter, with indications in which section each step is discussed.

are available to verify the lockstep in Section 8.5. Having chosen the most suitable fault injection techniques for our architecture, in Section 8.7 we describe the automated test toolchain we developed to systematically conduct our test campaign. We utilize a set of fault-templates to inject the different faults types described in Section 8.7.3, which we derive from our fault model. The results of our fault injection campaign are presented in Section 8.8, and we compare them to related work in 8.10. Before presenting conclusions, we document pitfalls encountered while preparing and conducting our campaign in Section 8.11, and describe changes made due to lessons learned during validation.

8.2 Related Work

Computer architectures for space-use 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). FT is traditionally implemented through circuit-, RTL-, core-, and OBC-level majority voting [104, 132, 188] 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 [88, 191]. 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 software. Hence, testing is strongly focused on the pure hardware with software functionality during tests often being reduced to stub implementations to assert basic functionality.

The characterization of the effects induced by radiation within a semiconductor is of major concern when implementing traditional hardware-FT based systems. Today, radiation testing is the only practical way to evaluate them, with radiation models offering useful but tentative and often inaccurate high-level fault estimates. Radiation test results for different components including memory and watchdog/supervisor-μCs are available in databases such as ESCIES, NASA's NEPP[1] and the IEEE REDW Records. Relevant radiation tests have been conducted for the FPGAs utilized in our project, among others by Lee et al. in [297] and Berg et al. in [143], or are currently ongoing (Glorieux et al. [298, 299]).

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 fabs and fabrication lines. 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. For our architecture, radiation tests 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 an appropriate checkpoint frequency and frame times for our coarse-grain lockstep approach. By itself, however, radiation tests do not allow an assessment of the capabilities of software-implemented FT measures.

While transient random bit-flips are often considered in academic literature, the otherwise different fault model [5] prevents the re-use of many FT approaches developed for ground applications. Also, the form factor constraints aboard miniaturized satellites [197] prevent the re-use of most high-availability and failover concepts for critical terrestrial control applications. Even for atmospheric aerospace applications, dependable computing usually considers availability, non-stop operation, and safety, but rarely computational correctness in a fully isolated and autonomous system.

Prior research on software-implemented FT often considers faults to be isolated, side effect free and local to an individual application thread [208] or purely transient [199, 205]. Many practical application obstacles could be uncovered and resolved before publication by implementing these concepts [198]. However, implementation of a measure and fault injection are time consuming tasks [300]. They often require not only software to be implemented, but also suitable tools and hardware or a representative substitute, as outlined among others by Sangchoolie et al. in [301]. Especially fault injection for entire OS instances is non-trivial [302], as thorough preparation and careful test-tool selection is necessary to obtain representative results from a fault injection experiment [303]. Therefore, a sizable share of FT concepts exists at a theo-

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

retical level [212–214], instead of having undergone fault injection or hardware testing. To still achieve some degree of validation, many publications thus resort to statistical modeling using different fault distributions. This is a viable approach for validating FT concepts directed towards, e.g., yield maximization [58] and aging [304], but not for software-implemented FT measures for critical environments.

In this chapter, we conduct systematic validation of our coarse-grain lockstep approach using fault injection to verify the effectiveness and efficiency of our coarse-grain lockstep FDIR mechanisms under stress. Specifically, we must assure voter stability and a sufficient level of fault detection to avoid accumulating silent data corruption and excessively brief frame times, while helping assess the amount of spare resources needed. 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.

8.3 Target Implementation

The high-level logic of our architecture is depicted in Figure 57, and consists of three interlinked fault mitigation stages implemented across the embedded stack. It is described in detail in Chapters 4 through 6. At the core of this architecture is a coarse-grain thread-level lockstep implemented within the kernel of an OS, which we refer to as Stage 1. It implements forward error correction and utilizes coarse-grain lockstep to generate a distributed majority decision for an operating system. The thread-level

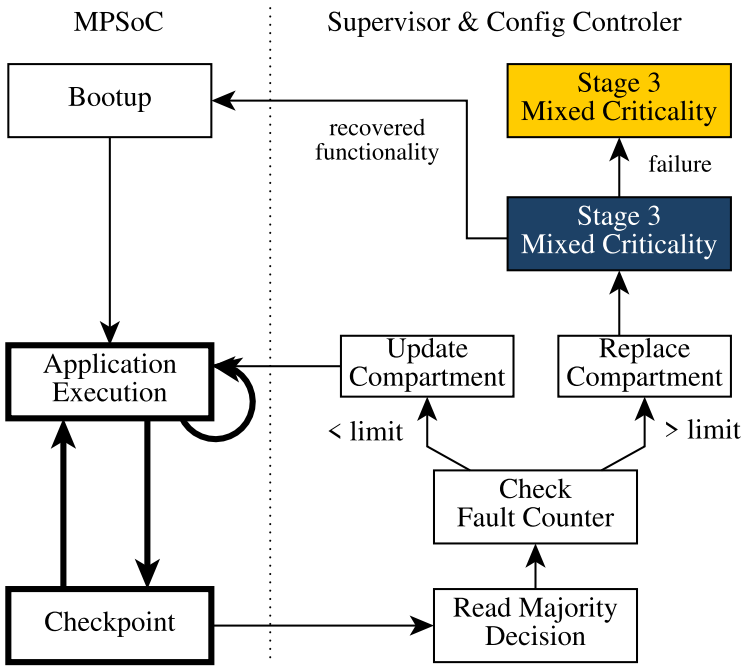


Figure 57: Stage 1 (white) assures fault detection (bold) and fault coverage, Stage 2 (blue) and 3 (yellow) counter resource exhaustion and adapt to reduced system resources.

lockstep assures the integrity of software replicas run on a set of otherwise isolated, weakly coupled processor cores. Fault detection is facilitated through application-provided callback functions, requiring no knowledge about application intrinsics and also no modifications to the application structure. Faults are resolved through state re-synchronization and thread migration to processors with spare processing capacity. Stage 1 is described in further in Chapter 4, where we also establish an upper bound for the performance cost of the lockstep. This coarse-grain lockstep is validated in this chapter, and provides fault-detection capacity for the subsequent stages and short-term fault-recovery.

8.4 Obtaining a Practical Fault Model

To properly validate software-implemented FT measures, information on the physical fault model is required. This information is necessary to choose a fault-injection technique and the right tools to inject the faults. In the remainder of this section, we show how to deduct a practical fault model from our operating environment. 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.

To validate our lockstep implementation, we must specifically test how well our lockstep implementation can detect faults. We need to verify this not only at the system level, following a majority decision by all involved compartments, but also locally by an individual lockstepped compartment into which a fault has been injected. Besides fault detection and the possibility for recovery, it is necessary to determine how stable or unstable a lockstep will behave. For space applications, a software-implemented FT concept must be subjected to transient faults, permanent faults, faults that are neither (intermittent faults). The effect of a radiation induced fault depends on the particular effected chip region, logic, and microfabrication technology used [5].

Our coarse-grain lockstep exists as part of the scheduler and utilizes a set of application callbacks. Therefore, we must consider the actual effect and impact of faults on the system from a programmatic perspective. Radiation induced faults will, thus, have the following effects on the software executed within one of our MPSoC's compartments:

- 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 compartment I/O peripherals, due to faults in controller logic of FPGA's I/O components.

To properly represent these faults, we should inject both bit-flips and new-values. 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. Proper validation for software must be systematic [305], which can not be achieved at the system-level when testing a physical hardware prototype. Software must be tested separately and systematically, so that then a prototype can be developed that can undergo system-level testing.

A broad variety of synthetic, theoretical failure types are well described in literature, e.g., in [303]. In practice these do emerge as one of the described fault types. As discussed among others in [306], most of these synthetic failure modes [303] 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, as discussed further by Sangchoolie et al. in [301].

Radiation can induce subtle effects into logic and may affect the OBC at a system level (e.g., full component failure or reset) [143]. Such faults emerge disguised as one of the aforementioned ones in case their effects are transient or intermittent. Furthermore, we also need to test the lockstep's behavior under permanent faults.

Faults with a permanent effect are either fatal to a compartment, therefore directly detectable by other compartments by majority decision, or affect the system as a whole. Our lockstep is not designed to recover the system from large-scale system-level permanent faults, and utilizes spare resources to cover the permanent failure of individual compartments. These are covered by Stage 2 and, if necessary, escalated to or detected by the on-board computer's external supervisor through time-out.

8.5 Suitable Fault-Injection Techniques

Fault injection into a live hardware-system or an FPGA (e.g., using JTAG or ICAP) would be most straight forward way of conducting fault injection. As research budgets are finite, this naive approach does not allow a meaningful level of test coverage from being achieved, as systematic test coverage is potentially destructive [115], time consuming, and would require a high degree of parallelization. [307]

As our architecture is designed for FPGA, fault injection using netlist simulation [64] or directly into the FPGA [115, 308] could be facilitated with comparably[2] little development effort, as we already utilize a development-board based MPSoC design implementation. This technique would grant precise control over the type and effect of faults and the simulation could be conducted with a system closely corresponding to the real one. Several proprietary partially [115, 308, 309] and fully automated test frameworks [310] as well as commercial applications [64] have been developed for this purpose. Unfortunately, netlist simulation of a full MPSoC is computationally disproportionately expensive. Therefore, netlist simulation, too, does not allow us to achieve meaningful level of test coverage.

Faults could also be injected via widely available standard software debug tools (e.g., GDB) into software running in userland. This is only representative for tests considering only the effects of transient faults in simple userland applications [199]. The effects of faults on a full OS implementation and permanent component damage

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

cannot be simulated [311]. 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, as the platforms and their ABIs are fundamentally different. Fault injection into kernel functionality emulated in userland may also result in a different run-time behavior than when running bare-metal. This technique can therefore only yield meaningful validation results for pure application level FT concepts [303]. Debugger-driven fault injection into a virtual machine can alleviate these constraints by allowing an actual OS to be tested. However, this technique is unable to correctly simulate permanent and intermittent faults in components other than memory and the current execution context. In consequence, the fault injection using debug tools is significantly constrained [303] and insufficient for validating our lockstep. This is an inherent limitation of that can only be alleviated through cooperation of a virtual machine monitor without hardware acceleration [302].

ISA-level binary instrumentation has been shown powerful and efficient for conducting black- and grey-box fault injection [301], and is today widely used for reverse engineering, security and malware analysis purposes. Though most of these tools are tuned towards reverse engineering, not fault injection. Fault-injection capable tools discussed today in relevant publications are mostly proprietary to individual research groups [301, 312]. Without exception, they are rather experimental and tuned towards single applications, and often also simply not publicly available [312]. To be comparable however, proprietary tools unavailable to all but a research group are not relevant.

Fault-injection into a virtual machine (VM), in contrast, allows considerable code and tool reuse: a VM can be constructed using pre-existing virtualized hardware available in widely used standard tools. Due to the considerable optimization effort invested into virtual machine monitors, this technique is computationally relatively cheap. Depending on the used VM technology, it no changes are to a victim application and the emulated machine be can resemble the actual intended target system rather closely. Several test frameworks implementing this approach have emerged in recent years, though most are still custom tailored for specific usecases or have not been released to the public [300, 305]. Notable exceptions here are the two open source frameworks FAIL [306] and FIES [313]. These are publicly and freely available as open source software and reasonably mature, and therefore we began to conduct our fault-injection campaign using this technique. However, these tools are only capable of injecting faults into a single core of an MPSoC, even though they can simulate a VM with multiple processor cores.

Fault injection using system simulation can combine many of the advantages of the aforementioned techniques. In prior research, actual MPSoC architectures were simulated using SystemC to demonstrate architectural features. This could also be used as compromise between the level of detail and extreme computational cost of fault injection using netlist simulation, and limitations of fault-injection using system emulation when targeting an multicore system. Until recently, however, modeling and implementation of an MPSoC capable of running real software software using SystemC required an excessive amount of development effort. With the emergence of modern architecture description languages such as ArchC and in combination with the emergence of more open processor core designs such as RISC-V, the development effort necessary to do so has been reduced to a more realistic level. We therefore conducted further testing of our implementation for with an ArchC implemented SystemC model

our our MPSoC to validate our lockstep in a true multi-core environment without the constraints of system-emulation-based fault injection.

8.6 Test Campaign Setup

Having determined a fault-injection techniques and knowing what kind of faults need to be injected, we must prepare a suitable test environment to properly To achieve systematic test coverage, manual fault injection or injection relying upon manual binary introspection are unsuitable. Instead, an automated campaign setup is needed. In this environment, we can then subject our lockstep implementation to fault injection in bulk. This process can then be paralleled to achieve the desired test coverage. 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.

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

1. Result harvesting: 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. Fault-free simulation: we execute a golden run of our target implementation and generate traces for executed instructions, register and memory access with the actual binary used for fault injection.

3. Filter the traces to constrain fault injection to application relevant code and data (e.g., omitting platform bring-up, OS, and shutdown code).

4. Remove duplicates, and annotate each trace-entry with the number of occurrence in the trace, generating the test-campaign input data.

5. For each address and occurrence, we generate a fault definition based on a template and launch an instance of our fault injection tool.

6. Based on a comparison to the known-correct results obtained in the first step, we determine the impact of the injected fault (e.g., OS crash, incorrect checksum, SDC, etc.) and log the result to an sqlite[3] database. Besides collecting and interpreting the results of a fault injection run, we also retain compartment state information to enable manual analysis in the future if necessary. This includes a compartment's human readable output to each compartments' 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.

Steps 1-3 are executed once at the beginning of a test campaign, whereas steps 4 and 5 are computationally comparably expensive but can be parallelized. As sqlite stores a run's database in an individual file, result databases from different systems

[3]Any database would work, but we want to keep the results portable so they can be combined later one.

can be merged, and each test record includes information about the precise injected fault.

Long fault injection campaigns place considerable strain on host a computer's filesystem. While running our test campaigns, we discovered that this can cause induce significant wear in SSD-based storage device. When replicating this setup, the avid reader may wish to instead conduct fault injection fully in memory to avoid damage the host computer's SSD. This can be achieved by running experiments in a ramdisk, e.g., by mounting tmpfs on the experiment directory.

8.7 Executing a Test Campaign

We conducted our fault-injection campaign using both system emulation with the FIES fault injection framework and through SystemC simulation with a 3-core MPSoC model.

8.7.1 Tool Selection

The available emulation-based FI tools which were available at the time of initiating validation for our lockstep were not functionally equivalent. They differ regarding the target environment, test setup and intended test subject scope, and the way in which they inject faults. The FAIL-framework 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 intrinsics and program structure. This information is target binary and concept dependent, and is hardcoded within a dedicated experiment controller binary [4]. The development of FAIL is mainly focused on the Intel platform. ARM support less mature and only available through GEM5 [314] or through into hard silicon, neither of which are viable for our purposes as discussed earlier.

FIES by Höller et al. [313] was developed specifically to validate ARM-based COTS-based critical systems. It is based upon the much faster and more mature virtual machine monitor QEMU, thereby supporting a broad variety of SoCs and virtual hardware. However, there is no not support for conducting fully 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. We can therefore efficiently test a full OS including its kernel, without requiring a test monitor with knowledge about application intrinsics. The test campaign described in the remainder of this section is thus carried out using an automated test toolchain incorporating FIES.

FIES does not guarantee timing and strict time determinism. Hence, when validating more timing-sensitive algorithms however, special care must be taken to assure the golden run and fault injection runs are equivalent [312,313]. However, our lockstep implementation also does not require strict time determinism during simulation runs. It only requires that a comparable level of work is conducted between checkpoints.

In the process of developing our test toolchain, we extended FIES' functionality to better support different tracing techniques and added functional improvements.

[4] See the src/experiments directory at https://github.com/danceos/fail

Initially, this began as bugfixing effort, but over the course of several months, we in practice rewrote most fault-injection triggering related code, as well as a major part of FIES' state machine. FIES originally was also based on QEMU 1.17, and therefore we rebased the heavily modified FIES code to QEMU-git 2.12 (qemu-head in December 2017). We also added support for the THUMB2 instruction set as FIES originally only could inject faults into ARM instructions, and only used those as fault-triggers, as most common software use both ARM and THUMB2 assembly intermixed. At this point, we had rewritten major parts of FIES, and we therefore made not just patches for FIES available, but released the entire tool as "FIESer – FIES Extended and Reworked" to the public. It is source code is available at `https://fieser.dependable.space` and on `https://github.com/dependableDOTspace/FIESer`.

To realized fault injection via SystemC, we first had to develop a suitable MPSoC implementation. Most SystemC MPSoC models described in literature, however, at close inspection turn out to only be capable of running brief instruction sequences to validate parts of, e.g., an instruction set, or a specific low-level functionality of an MPSoC. Hence, they are incapable and often not even intended to run run actual application software, which we require to test our lockstep implementation. This is no problem for emulation-based fault injection, where only the high-level behavior of a system is emulation, but challenging for more close-to-hardware SystemC-based simulation. Hence, as part of an ongoing international inter-university collaboration, we implemented a true multi-core model of our MPSoC. We implemented this MPSoC through the use of the open RISC-V platform, for which preexisting ArchC models were available. Each processor core existed in its own compartment with dedicated I/O capabilities as described in Chapter 4, and have access to a shared memory segment used to exchange and compare lockstep state information.

8.7.2 Target Implementation and Payload

When conducting fault injection it may seem obvious that these tests should be conducted against a realistic target implementation. However, this is only feasible if the right tools were chosen as described in the previous sections. A majority of publications today does not do so, and often researchers seemingly try to force-use unsuitable fault injection tools to validate their implementation. In the remainder of this section, we thus describe the fault injection target implementation of our lockstep, and outline how and why it is representative for our purposes.

A simplified function flow graph of our lockstep implementation is depicted in Figure 58 for reference, and in full described in Chapter 4. As payload application, we utilized two applications:

- The ESA Next Generation DSP benchmark[5] 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[6] [219].

While this choice represents satellite computing workloads reasonably well, test campaigns for other application should utilize representative software. If no specific target

[5]Source code publicly available at https://essr.esa.int
[6]See https://github.com/spacetelescope

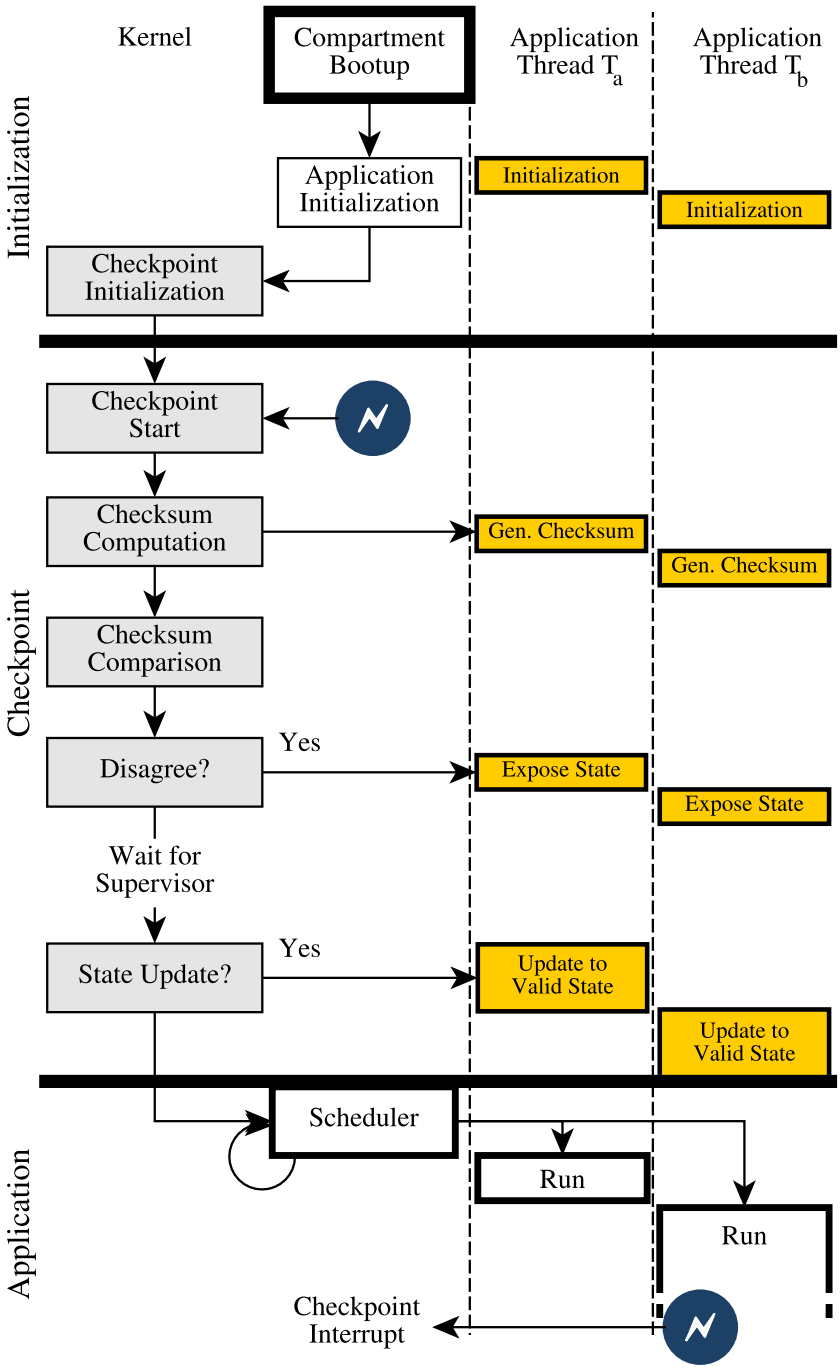


Figure 58: The execution cycle of our coarse-grain lockstep implementation on a compartment. Payload application callbacks are depicted in yellow, checkpoint trigger timers in blue. Faults are injected after initialization.

application code is available, synthetic algorithm suites such as the SPEC performance tests[7] can be utilized at a loss of realism due to the limited scope and low complexity.

Our fault injection experiments using system emulation were conducted against an implementation of our approach in RTEMS 4.11.2 using the ARMv7a-Zynq board-support-package, which closely resembles the compartments of our MPSoC. RTEMS is a real-time OS running bare-metal, and is used in a broad variety of space applications. 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. 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. Note that RTEMS does not utilize privilege separation, enforces no separate between a userland and kernel code, and has no virtual memory support. All these features would make faults more easily detectable and the OS as a whole more robust. Hence, faults in application code can directly interfere with kernel data structures. However, the absence of such functionality is representative for today's space computing even aboard larger spacecraft.

For SystemC-based fault injection, the model used was implemented using SystemC version 2.3.1 and ArchC 2.4.1 with custom patches to enable fault injection. Instruction instrumentation was realized using nightly builds of AspectC++, as the latest released version of AspectC++ is outdated[8]. The excessive amount of compute time necessary for fault injection into the MPSoC prevented the re-use of the same lockstep implementation used as for emulation-based fault injection [315]. Initially, we attempted to re-use the same test application setup we developed for emulation-based fault injection, but a single fault-injection run with this application in our ArchC model on just one processor core would have taken more than 8 hours. Therefore, instead of running a full RTEMS implementation of our lockstep, we constrained our implementation to run bare-metal code without thread-management, interrupts, and timers. This implementation was cross-compiled using the RISC-V toolchain released and maintained by the Andes Technology Corporation at `https://github.com/andestech/riscv-llvm-toolchain` against the `ilp32` ABI of the `rv32ima` RISC-V architecture variant. At the time of writing and conducting these fault injection experiments, the toolchain uses GCC 7.1.1. Naturally, this curtails the fault tolerance capabilities this implementation can achieve, but it allows the test time to be reduced to approximately 1 minute of real-time per injected fault.

8.7.3 Test Space and Target Components

We prepare a set of fault definition templates, which our fault injection toolchain combines with information from the previously generated traces. These templates define the test-space of our campaign. However, choosing the right test-space for testing an OS-scale fault tolerance measure is non-trivial. A test-space as described in literature [316] as ideal for testing software in practice is usually not achievable [317], and stands in stark contrast to the best practices in system-level testing in industry [318, 319]. Even fault injection with state-of-the-art tools requires a carefully chosen compromise between realism and test-coverage to avoid runaway test-times and high cost.

[7] see `https://www.spec.org/cpu`

[8] At the time of writing AspectC++'s latest released 2.2 is more than 2 years out of date and its functionality is no longer comparable to those of the nightly development builds

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 available tools do not assure clock-cycle accurate timing. 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. Our SystemC implementation is designed to allow fault injection also with cycle accuracy in different parts of the processor pipeline, though we consider this functionality to be too unreliable to use it for fault-injection yet. With FIES, we inject faults also into memory access operations based on physical memory addresses. This allows us to approximate the effect of faults in caches and main memory, as well as faults in buffers. To better simulate non-correctable upsets in ECC words and faults in the address logic, we can also directly replace accessed data or replace the address of the operation.

Permanent Fault Injection

Permanent faults should be injected into accessed main memory and devices address space. However, they should not be injected into general purpose registers, special registers, and the CPU pipeline provided little added value for testing software-implemented fault tolerance measures. This is due to the fact that the effects of faults in these components are fatal at the latest after a few clock cycles. Hence, they will interrupt operation of a processor core, and this can be detected through our lockstep by other compartments in the MPSoC, as well as by the supervisor. While it is important to not ignore parts of our fault model, testing with faults with a predetermined and known result would needlessly inflate the test space and time.

Functional Interrupts and Intermittent Faults

Radiation may also cause fault-effects which are neither transient nor permanent. To simulate SEFIs with FIES, FIES' fault types of periodic and intermittent faults can be used. For these, fault effects persist for a user-described period of time and are resolved by the injection framework afterwards.

In our tests, we chose 100ns as fault-duration for SEFIs, the period-equivalent to 10 clock cycles at 100MHz, the frequency emulated by QEMU for the Zynq MPSoC. This represents the interruption effect and the reset-induced outage of specific circuit groups due to SEFIs reasonably well. However, we are not aware of radiation-test data further analyzing the 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 on FIES and 5 lockstep cycles on ArchC, and then terminates. The test sequence is depicted in Figure 59, and faults are injected during the first checkpoint cycle or frame of execution. This allows faults to propagate within the system, to corrupt the application state, without requiring excessive experiment time. During the first checkpoint executed after fault injection, corruption of the application state should be recovered. Upon reaching the second checkpoint after fault

injection, the application state should have fully recovered and thereby the system state should match the golden run's results. This allows us to verify the full FDIR cycle from fault injection to recovery.

For emulation-based fault injection we chose a frame time of 2 seconds as interval between checkpoints. This is a reasonable choice for operation in LEO when passing through increased radiation zones such as the South Atlantic Anomaly, based on radiation-testing data for Ultrascale [143, 297] and Ultrascale+ FPGAs [298]. For SystemC-based fault-injection, checkpoints are executed after each frame the `NIR HAWAII-2RG` algorithm has been processed.

For our RTEMS implementation, a 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, 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). We are not relying upon this functionality as the value has to be hardcoded in FIES.

For our MPSoC, the execution time of a golden run for generating traces does not differ significantly from a run where faults are injected. However, even after much optimization a single run takes approximately 45 minutes of real-time on Core-i7 8700K system. We therefore reduced the NIR detector frame size from 2048x2048 pixels to 32x32 pixels, which then reduced the overall runtime to between 1 minute and 20 seconds, depending on the host system's performance. Naturally, this changes the ratio between code and data due to the much reduced size of the data structures used, but does not change the overall program structure of the executed application and the lockstep. As we already established an upper bound for the performance cost of our lockstep in Chapter 4, we consider this constraint acceptable.

After fault injection has terminated, we analyze if our lockstep could detect the effects induced by the injected fault (if any), and if they could be resolved through a

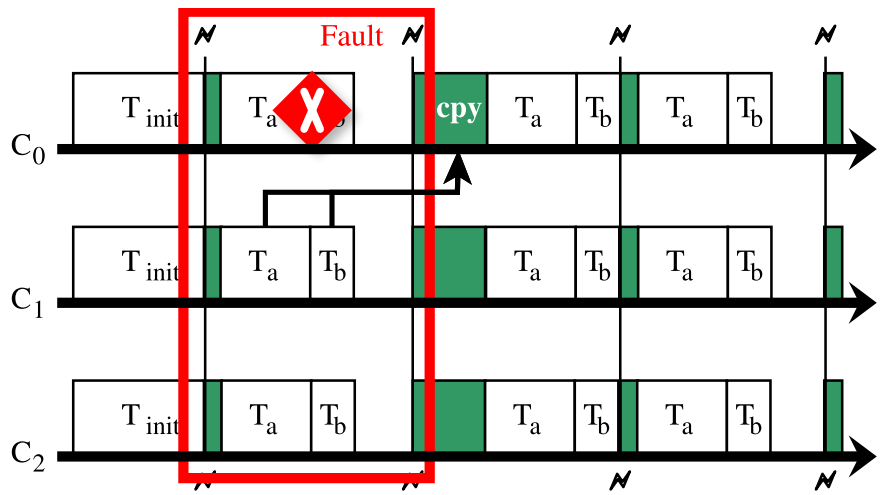


Figure 59: The experiment sequence and fault placement for a compartment. Fault are injected during the red-outlined time period on processor compartment C_0.

state update from another compartment. To reduce the test space, we do not inject faults into platform code, bring-up, an shutdown-related code.

Limitations

We chose the length of a fault injection run to allow our victim binary to exhibit the entire FDIR circle. As we are testing a full OS instead of just code snipplets or brief instruction sequences, this is necessary. In contrast to related work, the runtime of our fault injection campaign is therefore already excessively long, e.g., extended by more than an order of magnitude as compared to Amarnath et al. [305]. However, such a brief run still does not allow dormant or latent faults to be discovered, e.g., such affecting OS data structures and logic resulting time-delayed regressions. Only certain fault will produce immediate effects, and it is infeasible to extend our target binary's runtime even further. Therefore, it is impossible to observe or even determine if a fault results in no effect, silent data corruption, or time-delayed effects. The time allotted to each fault injection run therefore is a direct trade-off between achieving sufficient test-coverage to judge the fault-detection capacity of our lockstep, and to observe long-term effects.

In our ArchC system model, simulate RISC-V processor cores. This instruction set offers a large quantity of general purpose registers, which would inflate the test space as compared to our FIES ARM target (30 general-purpose registers as compared to 12 on the ARM platform). Therefore, we conduct an Architectural Vulnerability Factor (AVF) analysis [320] for the traces used in our fault injection campaign. AVF allows us to reduce the test space to avoid injecting faults into locations which would subsequently be overwritten, reducing masked faults and the overall test space. However, as discussed further by Maniaktakos et al. in [321] AVF overestimates vulnerability by more than 70%, and can not properly model the impact of multi-bit upsets in semiconductors manufactured in technology nodes less than 65nm feature size. In our campaign, we utilize AVF to constrain potential fault location (register address), but not to determine which bits are vulnerable and instead inject faults in each bit of a 32-bit word.

Our need for systematic testing also induces another limitation: Being constrained to running only a few lockstep cycles after fault injection, we also can not making more long-term observations regarding fault recovery. The fault recovery potential of coarse-grain lockstep also are heavily influenced by the protected applications and OS structure. Any fault-recovery statistics obtained for very short term fault recovery thus would be unreliable. Instead, this information should better be obtained through system-level testing with actual on-board data handling software on a prototype.

It would be feasible to inject faults in QEMU's emulated virtual hardware and into the infrastructure of our SystemC-MPSoC model. This would allow faults to be injection more realistically for each emulated or simulated device and MPSoC component. However, this is not supported in FIES and our SystemC-MPSoC model today. To our understanding FIES was also never developed with such functionality in mind. Hence, while technically possible, fault injection in qemu virtual devices would require considerable development effort even for only one set of virtual devices relevant for validating our target architecture. Due to a lack of tools, we can instead approximate the practical effects of radiation by injecting faults during access to memories and device address space, as well as into the CPSR on FIES.

For our SystemC-MPSoC, there is no structural limitation to fault injection as with

FIES, and in the coming months we plan to expand the fault-injection capabilities of this model. At this point in time, have begun adding cycle accurate fault injection support, instead of instruction-based fault injection which is possible with FIES and our ArchC model today. Once this has been accomplished, we plan to inject faults also into the MPSoC's interconnect, as well as CPU peripherals and interfaces that are part of a compartment.

8.8 Results & Interpretation

To test our toolchain and verify its correct functionality, we conducted manual fault injection into specific application structures using FIES. We injected such faults into interesting data and logic which could cause an incorrect application state, or could otherwise alter the run-time behavior of a compartment. This allows us to analyze the practical behavior of our lockstep under faults, and enabled us to directly compare the impact of a fault in a specific location when injected as transient, permanent and intermittent faults. Table 5 shows the behavior of our lockstep under faults, and we subsequently expanded our fault injection campaign in the described automatized way with FIES and our ArchC model. In Table 6, we provide statistics observed when conducting fault-injection with FIES and ArchC.

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 (compartment 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 time spent executing lockstep mechanisms, as its low code and data footprint.

A comparable share of bit-flips with permanent effects 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

	Detection by		Recovery	Recovery Method	
Result	Victim	System	Trigger	State Update	Reboot
Corrupted State	yes	yes	lockstep	yes	yes
Thread Crash	yes	timing only	lockstep	yes	yes
Lockstep Failure	no	yes	supervisor	no	yes
Crash/Hangup	no	yes	victim core	no	yes
No Effect/SDC	no	no	supervisor	sometimes	yes

Table 5: Behavior of our RTOS implementation under faults, considering fault detection at the system level, as well when considering victim-processor core itself. Notice that our lockstep implementation can not detect silent data corruption with no immediate impact on the thread state.

	Effect by Injected Fault Type			
Result	FIES Transient	ArchC Transient	Permanent	Intermittent
Corrupted State	49%	32%	44%	53%
Thread Crash	8%	-	17%	10%
Lockstep Failure	1%	1%	2%	1%
Crash/Hangup	10%	14%	18%	15%
No Effect/SDC	32%	54%	19%	21%

Table 6: Fault injection experiment results to date with FIES and ArchC divided into transient, permanent, and intermittent faults. A share of all masked faults will cause silent data corruption, which can have long-term effects on OS data structures. These 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.

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 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 compartment. 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 compartment itself can detect faults in Figure 60.

The fault injection campaign shows that there is indeed a measurable difference in behavior between transient and permanent faults, and between target applications of different complexity. As expected, permanent faults are more likely detectable than transients, due to their increased severity. However, we also expected permanent faults to be easier detectable by a compartment than SEFIs (see Figure 60a). This was not the case. The increased likelihood of permanent faults resulting in crashes and the higher percentage of non-fatal state corruption faults due to SEFIs made fault detection within the affected compartment 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 compartment. These results underline the importance of conducting validation not only using transient faults, but also with permanent and intermittent faults.

The effects of a fault will be detected through majority decision by the rest of the system. The fault detection rate increases sharply, as the MPSoC as a whole can also detect crashes of an entire compartment or lockstep mechanism failure, as shown in Figure 60b. In Figure 61, we therefore provide a direct comparison between self detection and majority decision 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

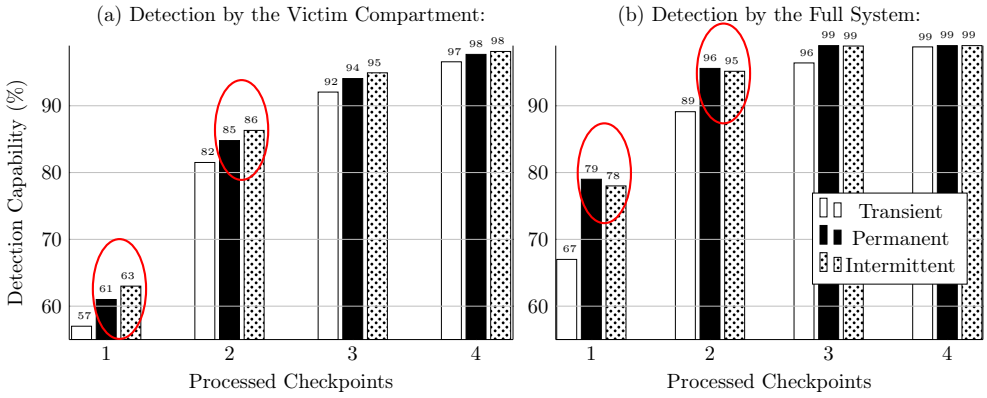


Figure 60: Payload application and state corrupting fault detection chance of a single compartment for different fault types after a given number of execute checkpoints. Notice that intermittent faults are more likely to be detected than permanent faults by the affected compartment itself, which is counter intuitive. This is due to the increased percentage of faults that are fatal for a compartment, and the system as a whole can detect permanent faults with higher likelihood.

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 61b 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 compartments thread assignment within the MPSoC to be critical. This is caused by crashes and reboots of individual compartments. Worst-case benchmark results showed that frequent crashes of compartments 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%, cause crashes and lockstep-failures. Even under the (unrealistic) assumptions that faults were to

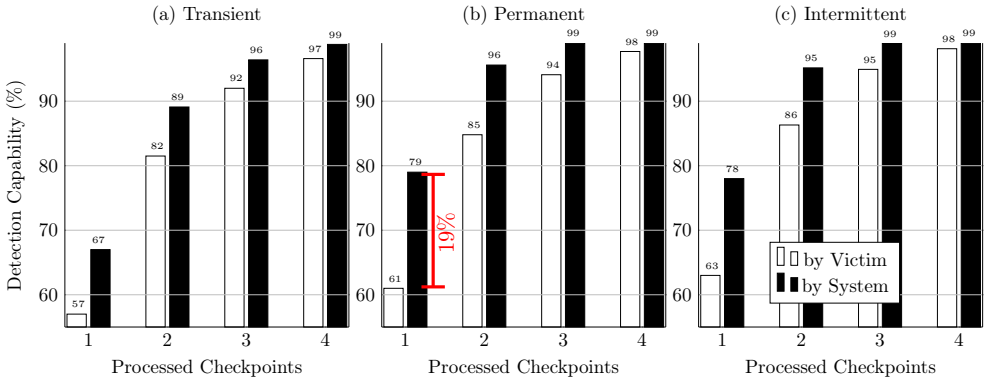


Figure 61: Comparison of the fault detection capabilities of an individual compartment and the by MPSoC through majority decision. The full system can also detect a crash of the OS instance running on a compartment, and malfunctions in the lockstep logic.

Effect	Number of Faults	%	**Thereof:**	Immediate Recovery	Lockstep Timeout	Reboot Required
Non-Masked	47526	46%		22004	10915	14607
				46%	23%	31%
Masked	57379	54%				
All	104905					

Table 7: Fault Recovery statistics for SystemC fault injection.

occur in each checkpoint period, many faults could still be resolved through a state update and do not require a reboot. Hence, our lockstep implementation can provides the necessary degree of voter stability to making application reassignments between compartments rare.

A majority of 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 [322]. 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.

8.9 ArchC MPSoC vs. FIES Result Comparison

Comparing our transient results between ArchC and FIES, we notice that the results are mostly comparable. The share of faults without noticeable effect are increased by approximately 20%, which seems reasonable considering the different lockstep implementations tested: part of this difference can be attributed to the vulnerability overestimation remaining due to limitations of our AVF analysis. Furthermore, the lockstep implementation on ArchC can not exploit the powerful exception handling function available in a proper operating system implementation, as we are here running the test implementation bare-metal. Instead, our FIES implementation exists as part of RTEMS, which allows more precise fault analysis, and overall reduces the chance that a fault will crash the entire OS instead of just the test application thread.

To allow better comparison of the fault effect ratios between system emulation and SystemC fault injection, we have to normalize the results obtained with both techniques. To do so, we apply normalization to the 54% of masked faults to all effect ratios obtained with FIES, where we encountered just 32% masked faults. A comparison between normalized FIES fault effect ratios and ArchC is depicted in Table 8. As depicted, after normalizing the result data, we receive almost identical fault effect ratios with both techniques, with our RTOS implementation showing 6% higher data corruption likelihood than our bare-metal implementation. In our ArchC lockstep implementation, 15% of all faults cause a crash or hangup effect, while in our RTOS implementation 14% of cause such an effect. As our FIES implementation utilizes

threading 6.5% of all crashes remain isolated to the crashed application software, or the lockstep, while our ArchC implementation knows no such separation. In practice, this shows that the additional OS and application isolation functionality implemented within a modern OS also has a positive impact on suitability. In turn, the increased amount of code an data required for an OS-scale implementation also shows that the ratio of faults causing data corruption is slightly higher than when running the same application bare-metal.

In Figure 7, we provide fault effect and recovery statistics obtained from our ArchC MPSoC model. After observing 105905 fault injection runs into our ArchC MPSoC model using AVF-filtered golden run traces, we can observe that: in 46% of cases a corrupted thread-state could immediately be recovered through a state update, required no reboot of the faulty MPSoC core. In further 23% of cases, faults could have been recovered if the lockstep had allowed for more wait time during checkpoint voting, which was severely constrained in our test campaign to assure sufficient test coverage. Only in 31% of cases, fault resolution was unsuccessful, requiring a reboot of the affected processor core. Overall, these statistics are very positive, considering especially the much reduced fault-recovery potential that a bare-metal lockstep implementation has as compared to a full OS implementation.

Considering the different scale and detection capabilities of the two different lockstep implementations analyzed, this different is in line with our expectations: The target implementation we used for ArchC fault injection does not utilize a threaded scheduler, and therefore thread-management and scheduling is eliminated as potential failure source. Overall, injected faults in a threaded RTOS implementation should locally also impact OS-level control logic, and infrastructure data structures, and induce secondary fault effects there. At the same time, the this also means that faults which in an RTOS implementation caused a thread to crash, now would only cause data corruption in the protected application.

8.10 Comparison to Literature

To place these results in context with results from other lockstep concepts, we sought to compare our results to literature. 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

	FIES			
	Ref.	@ 54% SDC	ArchC	Δ
Corrupted State	49%	38.22%	31.72%	-6.5%
Thread Crash	8%	6.24%	0%	-6.24%
Lockstep Failure	1%	1%	1%	0%
Crash/Hangup	10%	7.8%	14.54%	+7.66%
			Δ Total	**5.08%**

Table 8: Transient fault effect comparison between system emulation and SystemC fault injection, normalized to equivalent SDC ratios.

et al. [199] 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 [199], the share of faults causing application, thread, and OS crashes with our approach is noticably increased. For transient faults, this can at least in part 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, Dobel et al.'s lockstep can offer more fine-grained protection than our approach. However, it also require considerable code, deep and non-portable changes in the target OS, has a high performance overhead, and constrains the target OS and application structure. 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.

Dobel et al. consider their fault injection measurements overly optimistic, as they utilized payload applications *"of little complexity (leading to few potential candidates for fault injection)"* [199]. Their validation and lockstep implementation is constrained to handling transient faults, while SEFIs or permanent effects are not covered as these faults were injected into a user-land application of their approach through a debugger. Dobel et al. assume the OS, system libraries, and kernel to be fault-free, while we instead inject faults into a full OS including POSIX libraries with payload applications. 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.

The results we obtained with SystemC fault injection into our ArchC MPSoC confirms this further. There, we can in practice reproduce exactly this same scenario between the two lockstep implementations we have been utilizing for testing with FIES and for our ArchC MPSoC-model. The lockstep implementation there is overall simpler, has fewer calls to critical infrastructure functionality that could break, and therefore offers less overall failure potential than our full RTEMS-implementation. Furthermore, in this MPSoC we utilize RISC-V processor cores with a much simpler and less powerful instruction set than that offered by a full Cortex-A processor core implementing the ARMv7a instruction set, which not only supports one instruction set, but uses two instruction sets in combination (ARM and THUMB).

8.11 Discussions

Fault injection today can be conducted for different reasons, such as to detect security vulnerabilities in software, memory leaks, or to assure test coverage when testing for functional correctness. However, fault injection for validating the correction functionality of a fault-detection and lockstep technique is very different from, e.g., fault injection conducted for security purposes. Applying the same assumptions or test tools to both, while attractive, does not result allow for proper validation. The used fault injection techniques, target implementations, and payload software will influence the obtained results. Validation using an overly simplistic target implementation 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 implementation with non-trivial payload software, but also that more lockstep concepts must be validated.

Our coarse-grain lockstep can detect faults resulting in a crash or in corruption of

the thread state. However, it is unable to detect silent data corruption and latent faults in OS data structures and code. To better handle this, a compartment's checkpoint handler could generate a checksum for certain critical kernel data structures. However, the scope to which this is possible is limited and the computational cost may be high. It would be practically impossible to do this for a larger OS or, e.g., the Linux kernel.

Velasco et al. propose in [323] 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 varying fault 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 then compiler then flattens the program structure, duplicating code segments instead of executing the same segment multiple times within a loop. Serrano Cases et al. in [324, 325] as well as Lins et al. in [326] 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 in the future should 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 had to implement support for the THUMB and THUMB2 instruction sets for FIES, to assure consistent tracing and fault injection results.

A jump between instruction sets without compiler-interwork would yield 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. We argue that 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. This would reduce the likelihood of silent data corruption and introduce a level software diversity through compiler instrumentation or scripted, automated code transformation [327].

When designing our coarse grain lockstep measure, we were aware of two ways of inducing checkpoints: through timers on each compartment and externally through interrupts. If timers are used, checkpoints are triggered independently on each compartment. Interrupt induced checkpoints are 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 better, as it avoids a central authority inducing checkpoints in favor of decentralized triggers. However, our fault injection campaign showed that interrupt induced checkpoints are considerably simpler. The timer-handling related logic requires more code and increases the OS state, and thus also more prone to faults

than a simple interrupt handler. Hence, in future work we decided to use interrupt driven checkpoints instead of timed checkpoints.

8.12 Conclusions

In this chapter, we presented an automated fault injection toolchain, and validation results of the software-implemented fault tolerance (FT) concept described in Chapter 4. Few software-implemented FT concepts proposed today have been validated, and therefore this chapter 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 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.

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. In practice, through this testing, we have exhausted all technically feasible testing techniques for software that are possible today to validate a fault tolerance measure of the scale of our lockstep. 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. Systematic validation of our coarse-grain lockstep implementation is therefore an intermediate step. To further test our architecture, a prototype system must be implemented to then conduct radiation testing.