

TUF: Time-bounded Fault Tolerance for Nano-satellites

Ahmed El Yaacoub
ahmadadam96@gmail.com
Uppsala University
Sweden

Luca Mottola
luca.mottola@polimi.it
Politecnico di Milano,
Uppsala University, and RISE
Italy and Sweden

Thiemo Voigt
thiemo.voigt@angstrom.uu.se
Uppsala University and RISE
Sweden

Philipp Ruemmer
philipp.ruemmer@ur.de
University of Regensburg
Germany

Daniel Primetzhofer
daniel.primetzhofer@physics.uu.se
Uppsala University
Sweden

Bart Royeaerd
bart.royeaerd@physics.uu.se
Uppsala University
Sweden

Abstract

We present TUF, a software technique to handle transient, intermittent, and permanent faults in bounded time for nano-satellites using commercial off-the-shelf hardware. Satellites are subject to bitflips due to high radiation levels. Existing techniques can cope with these issues for specific types of faults, yet are also unable to provide timing guarantees that are, however, crucial for time-sensitive systems. TUF combines triple modular redundancy, checkpoint/restart, and reconfiguration to detect and handle faults in bounded time and, unlike existing works, it does so for transient, intermittent, and permanent faults. We prototype TUF on a SoC with FPGA resources and evaluate it using fault injection methods on a logic simulator and on real hardware, where we also perform real radiation testing to mimic the environments satellites operate in. We show that TUF handles transient faults in as few as two clock cycles. The time to handle permanent faults through reconfiguration remains bounded. Real radiation testing ultimately demonstrates that TUF successfully handles faults in high-radiation environments, with checkpointing handling a fault in under 20 ns.

CCS Concepts

• **Computer systems organization** → **Redundancy**; *Real-time systems*.

Keywords

fault tolerance, real-time systems, FPGA, satellites

1 Introduction

Satellites deployed in space experience higher radiation levels than on Earth [46]. Higher radiation causes faults such as single event upsets (SEUs) [12], which cause bit flips in memory. These faults may lead to critical errors that impair satellites' ability to perform the mission at hand [18].

Problem. Because of power, performance and cost constraints, nano-satellites usually use commercial off-the-shelf (COTS) hardware rather than space-rated hardware [4]. The latter is significantly slower and more expensive [4]. However, COTS hardware is much more susceptible to faults.

Nano-satellites may also include time-sensitive tasks such as coordinating remote sensing functionality and mission schedules [44]. For example, capturing Earth imagery requires timely processing

of location data, or target areas may be missed due to the satellite's fast orbiting around the Earth [2, 7, 32, 61]. Further, packet forwarding in inter-satellite networks operates under strict time constraints to ensure low latency of communications [47].

To ensure fault-tolerant operation, techniques such as *N-modular redundancy (NMR)* [22] replicate memory and execution units N times so that they can be executed in parallel. A *voter* determines the correct output by majority. A type of NMR is triple modular redundancy (TMR). If at most one of the triplicated memory cells is faulty, the voter determines the correct value by selecting the outputs that agree and trigger fault handling and recovery procedures.

These procedures should complete before the correct value is needed to determine further actions. If an additional TMR unit becomes faulty before the first fault is corrected, majority voting fails. This applies to NMR in general, whenever the number of faulty units is greater than or equal to the number of non-faulty ones. The time to recover from a fault should be bounded to ensure that at most $\lceil \frac{N}{2} \rceil - 1$ NMR units are faulty at any time. NMR, however, provides no timing guarantees, as fault recovery may require arbitrary time.

NMR relies on various fault handling and recovery techniques. Consider checkpoint/restart, which is commonly employed in stateful systems [43]. When a fault is detected, the state of the system is restored to the last checkpoint before the faulty computation. The computation is reexecuted and checked again. If a fault is intermittent, that is, it occurs repeatedly before disappearing, or even permanent, the checkpoint/restart procedure may repeat for long or even indefinitely. Checkpoint/restart is therefore insufficient to handle intermittent or permanent faults in bounded time.

Reconfigurable hardware, such as Field-Programmable Gate Arrays (FPGAs), provides a way to address this problem. Units with intermittent or permanent faults in an NMR setup can be restored using reconfiguration [3], while other non-faulty units continue executing. Reconfiguration times are bounded as long as the system provides deterministic execution times and reconfiguration definitely corrects the fault. However, reconfiguration is significantly slower than checkpoint/restart, increasing the chances that another fault occurs meanwhile, preventing majority voting. Moreover, most faults in space are transient [53] and can be appropriately handled using checkpoint/restart without the additional overhead of reconfiguration. We elaborate on the fault model we consider and our problem definition in Sec. 3.

Time-bounded fault tolerance. TUF (Time-Bounded Fault Tolerance) combines NMR, checkpoint/restart, and reconfiguration to handle faults in bounded time and no downtime.

Unlike existing work that mainly focuses on transient faults [17, 20, 43, 45], TUF handles *transient*, *intermittent*, and *permanent* faults, as described in Sec. 4. The fundamental insight is that to achieve this, *we do not need to distinguish* between these different types of fault. We first apply checkpoint/restart and eventually switch to reconfiguration once a carefully chosen time threshold passes since the fault is detected, regardless of the nature of the fault and with no downtime. Detecting permanent faults is, indeed, difficult. Approaches exist [9] that place a bounded delay to make a decision and conclude that a fault is intermittent if it disappears before the delay is reached. Otherwise, it is considered permanent, even though it might not be.

We implement TUF on a Xilinx Zynq-7000 SoC with programmable logic, as described in Sec. 5. The Zynq-7000 features two Arm A9 cores and programmable logic resources. Such platforms are becoming common in nano-satellites [52]. In our NMR setup, we set N to three, corresponding to TMR. We use this prototype to verify that faults are handled in bounded time and to measure the actual fault handling time.

As reported in Sec. 6, we use fault injection techniques using both the Xilinx logic simulator and with the Zynq-7000 SoC. We find that reconfiguration time can be predicted with high accuracy. We also demonstrate that the combination of checkpointing and reconfiguration allows fault handling as fast as two clock cycles, while also handling more severe faults than checkpointing alone. We test with both faults injected at specific times in specific places, and with a realistic distribution of faults observed in low Earth orbit [40]. We also compare TUF with duplication with comparison (DWC) and find that TUF handles faults that DWC cannot.

Finally, we perform *real radiation testing* to evaluate TUF in an environment akin to space [60]. We use specialized hardware and testing environments with the help of trained personnel to carry out the experiments safely. We expose our target platform to an Americium/Beryllium neutron source with an activity of 2.6 GBq at about 5 cm, and to an Amptek 'Mini-X' X-ray source at 30k kV and 200 uA. We report a few noteworthy insights about the nature and duration of faults in these environments, which are unique in the literature, and find that TUF successfully handles faults there too.

We end the paper by discussing limitations of our work in Sec. 7 and with brief concluding remarks in Sec. 8. Before moving on to the technical contribution, Sec. 2 provides background information and surveys related literature.

2 Background and Related Work

Devices exposed to harsh environments, including high-radiation settings, are subject to Single Event Effects (SEEs). SEEs are radiation-induced faults caused by a single charged particle [35]. Single Event Upset (SEU) are soft errors that cause the value of a memory bit to be flipped temporarily. The duration of a fault determines its nature. Transient faults occur on tiny time scales, whereas intermittent faults “are error bursts that occur repeatedly at the same micro-architectural location in a processor” [55]. The impact of intermittent and permanent bit flips is generally more severe than

transient ones, since a single intermittent fault can cause several errors if the faulty bit is involved in multiple stages of processing.

Nano-satellites such as CubeSats have become more and more popular as launch costs fell [4]. This led to an increase in the use of COTS hardware instead of space-rated hardware to save on costs and improve performance [4]. COTS hardware is more susceptible to faults with fault rates two orders of magnitude larger than space-rated hardware [27]. COTS hardware opens up the possibility of using modern reconfigurable computing platforms to build robust and efficient fault tolerance mechanisms. For example, Fuchs et al. [14] use a Multiprocessor System-on-Chip (MPSoC) to enable fault tolerance in nano-satellites using COTS hardware.

Reconfigurable systems. In reconfigurable systems, the “hardware” is defined in a soft manner, that is, it can be changed programmatically. This enables different fault tolerance levels at different fault exposures [20]. For example, a satellite crossing through a high-radiation belt can reconfigure to increase redundancy, while otherwise operating with reduced redundancy but improved performance. Reconfigurable systems allow the migration of functionality from a faulty region to a healthy one, or the reprogramming of softly defined “hardware” components to fix faults while other non-faulty components continue to operate.

An example is FPGAs, which are shown to outperform regular processors for specialized use cases. Qasaimieh et al. [39] compare the energy consumption of vision kernels implemented using an ARM57 CPU, a Jetson TX2 GPU, and a ZCU102 FPGA. They found that the FPGA implementation is more efficient than a CPU or GPU for more complex kernels. Fowers et al. [13] compare the processing speed and energy consumption of convolution between GPU, FPGA, and multicore processors, showing that the FPGA implementation has lowest energy consumption.

FPGAs are being adopted in space systems due to the parallelism they enable, yielding improved performance and energy consumption with specialized tasks, such as image processing [6]. They are, however, more susceptible to SEEs than regular processors because the majority area is dedicated to storing the FPGA configuration [37]. An upset in the configuration memory affects how the circuit operates, for instance, by changing the routes of inputs and outputs [57].

Fault handling and time. Sari et al. [45] combine checkpointing with scrubbing on FPGAs to guarantee that tasks meet their deadlines in the presence of faults. Unlike TUF, they only target transient faults. Jacobs et al. [20] use FPGAs on a satellite to dynamically adjust the level of redundancy and fault mitigation depending on the level of radiation at different orbits. Unlike what we do, they use reconfigurability to adjust the level of fault tolerance, rather than for fault handling, and still do not deal with permanent faults nor with timing. Haque et al. [17] investigate fault bursts, limited time intervals where transient faults may occur, assuming that any computations executed during the fault burst are incorrect and must be re-executed. Their fault model is different from ours, as they only consider transient faults and assume that fault bursts occur once in the system’s execution.

Punnekkat et al. [38] use checkpointing for handling faults. They modify the scheduling algorithm to minimize response time when checkpointing. Rather than changing system schedules, we use redundancy to tolerate faults without affecting response times.

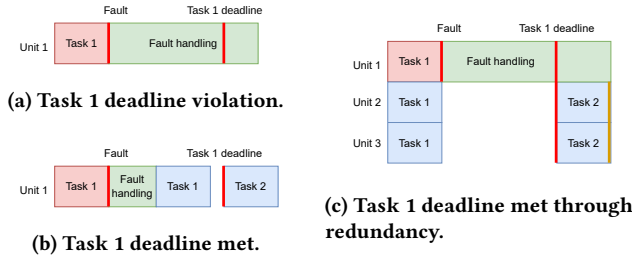


Figure 1: Examples of successful and failed time-bounded fault-tolerant task execution.

Reghenzani et al. [43] propose an extension for mixed criticality scheduling theory that can satisfy failure and timing requirements. In contrast, we use parallelism to handle faults while other tasks execute, rather than relying on scheduling sequential tasks.

Di Carlo et al. [10] tackle permanent faults through migration. However, their approach introduces downtime as the fault is being handled. Nikiema et al. [33] achieve similar fault handling times as we do, but with some downtime, and only handle transient faults. Hernandez and Abella [19] utilize dual-core devices to execute an application on two cores in lockstep. However, they do not explore recovery mechanisms beyond periodic checkpointing.

A different line of work is that of Sousa et al. [48]. They utilize proactive recovery, which performs rejuvenation of components at certain intervals. Sousa et al. [49] combine reactive and proactive recovery, where, in addition to rejuvenating components proactively, faults are also handled once detected, targeting server machines. Verissimo and Casimiro [54] tackle timely fault tolerance for asynchronous applications, but only consider timing faults. Poledna et al. [36] look into the scheduling of redundant tasks, which is unnecessary for us due to the availability of parallel processing.

3 Time-bounded Fault Handling

“Fault tolerant task execution in real-time systems means that the task will *correctly complete* before its *deadline* despite any fault occurrence during its execution” [45]. Fig. 1 demonstrates three examples. In Fig. 1a, the fault handling process violates the task deadline. In Fig. 1b, instead, the fault handling process meets the task deadline, and the task correctly re-executes before its deadline.

Fig. 1c is peculiar because of the use of a TMR setup. A majority voter can determine the correct output provided that the majority of the units are correct. By relying on TMR, the fault handling process does *not* need to meet the deadline. This is because two other identical, non-faulty units can still provide the correct output. The fault handling process must still be time-bounded to ensure that majority voting can yield the correct output before the task deadline. In the case of TMR, only one unit can be faulty at any one time. Whether this occurs or not depends on fault rate and handling time. The fault handling process is now time-bounded by the fault rate, rather than by the task deadline.

Many fault handling techniques are not time-bounded, especially if faults are intermittent or permanent. With checkpoint/restart, the fault must be gone for the execution to proceed; thus, fault handling time depends on fault duration and is not bounded. Similarly,

redundancy allows the execution to proceed until the number of faulty units exceeds the non-faulty ones. This also applies to data redundancy methods, such as Error Correction Codes (ECCs).

Fault model. Fault models describe *what* kinds of faults occur, *where* they occur, and *when*. Fault tolerance techniques are verified against their ability to handle faults matching the fault model. We consider a fault model that entails:

- **What:** transient, intermittent or permanent bit flips.
- **Where:** in application functionality, that is, not in the fault handling functionality.
- **When:** at any time but with a minimum time between the occurrence of any two faults.

These combinations are common in fault tolerance works [23, 53]. Also, we consider permanent faults to be the ones that “remain for indefinite periods till corrective action is taken [31]” and consider reconfiguration as a valid corrective action. This is practically verified in most cases [1].

We choose a fault model including all kinds of bit flips and do not place a time bound on fault duration. Permanent bit flips are particularly challenging for non-reconfigurable devices, since they cannot be corrected and therefore, leave the device vulnerable to errors. Handling permanent bit flips is our key motivation for utilizing reconfigurable devices.

We assume that faults cannot occur in fault handling functionality. This is common in fault tolerance literature [11, 29, 34, 59]. Real-world satellite deployments [58], as well as the evidence we discuss in Sec. 6, show that the fraction of resources invested in fault handling is negligible compared to those of application logic. This makes it easier to protect fault handling functionality with additional redundancy [8].

We also mandate a minimum inter-arrival time between two consecutive faults. This restriction is often found in literature [15, 17, 28, 38, 49, 51] and key for fault tolerance mechanisms that use NMR, yet also supported by empirical evidence [58]. NMR cannot handle more than $\lceil \frac{N}{2} \rceil - 1$ units simultaneously experiencing a fault. Therefore, we need to know how long it takes before we expect $\lceil \frac{N}{2} \rceil$ units to be faulty. While TMR research generally considers different units to fail independently, this is not the case in practice [5].

Target systems. We consider reconfigurable devices where execution units are softly defined. The device has a clock that is coarsely synchronized across all execution units, including the ones used in TUF. Application execution is *time-triggered* [21] and unfolds in *rounds* occurring at given time intervals, based on a synchronized system-wide clock signal. The device can reconfigure an execution unit while other execution units continue executing. The times to detect a fault, perform checkpoint/restart, run the execution unit with a set of inputs, and perform reconfiguration for a specific unit must be bounded. These bounds must be known prior to deployment, for example, through performance profiling.

FPGAs are an example that meets these requirements. Other device with comparable features include Coarse-Grained Reconfigurable Arrays (CGRAs). ASICs and hardware accelerators are not applicable, as they are not reconfigurable.

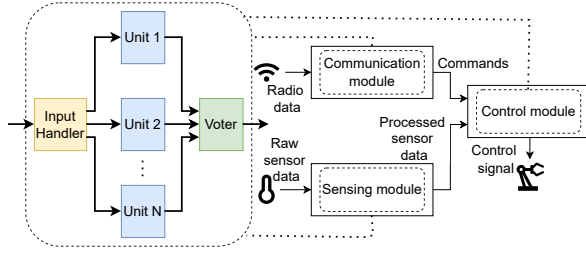


Figure 2: TUF architecture example.

4 Time-bounded Fault Tolerance (TUF)

Time-Bounded Fault Tolerance (TUF) uniquely combines checkpoint/restart and reconfiguration in an NMR setup handling faults in bounded time, regardless of their nature. We describe how TUF structures the programmable logic, the checkpointing process and fault handling procedure we design, including how the system determines whether to apply checkpoint/restart or reconfiguration. We finally demonstrate the time-boundedness of TUF.

4.1 Architecture

TUF organizes the programmable logic into different *modules*. Fig. 2 shows an example using three modules for communication, sensing, and control. Each module includes an input handler, n -replicated units, and a voter. The input handler proxies the interactions between upstream modules and the n -replicated units. The outputs of the units are input into a voter to determine the correct result and determine if some are faulty. The output of the voter cascades into the input handler of downstream modules, if any.

The use of NMR and per-unit reconfiguration allows non-faulty units to continue executing while the faulty unit is reconfigured. The FPGA ability to perform reconfiguration dynamically is crucial for time-sensitive systems because it allows the fault handling process not to consider *application* time requirements, in that the application logic continues to rely on the non-faulty units, as long as they form a majority. It still introduces a timing requirement in that, as explained earlier, the fault handling process must complete *before the number of faulty units reaches or exceeds the number of non-faulty units* or majority voting is no longer possible.

In the example of Fig. 2, say a fault occurs in one of the units of the sensing module. The local voter detects a fault and initiates the fault handling procedure, which we describe in Sec. 4.3. The control module still obtains correct data from the sensing module, since the other units continue executing, and is thus unaffected by the fault. As long as the fault handling procedure is completed before another fault occurs in another non-faulty unit of the sensing module, the whole system continues to operate correctly. This design also allows faults to occur in multiple modules at the same time, as long as at most $\lceil \frac{N}{2} \rceil - 1$ of the units in a module are faulty.

The combination of TMR, checkpoint/restart, and reconfiguration provides additional benefits over using these techniques separately. TMR by itself can only mask faults, it cannot fix them. Checkpoint/restart can handle faults that are temporary, but in a time-bounded manner only if the maximum fault duration is

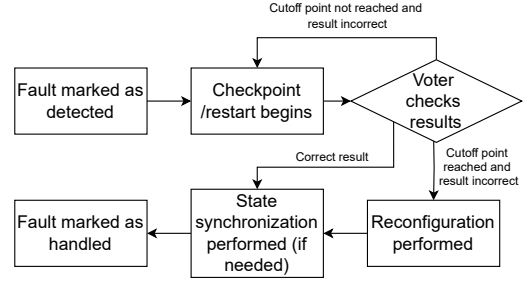


Figure 3: Fault handling procedure from fault detection until the fault is handled.

bounded. Reconfiguration bears energy overhead [26] and is slow compared with checkpoint/restart. It is only through the combination of these techniques that we can operate regardless of the nature of the fault, and do so efficiently.

4.2 Checkpoint/restart

TUF utilizes checkpoint/restart for handling transient faults and short intermittent faults. While the system operates, the input handler and the n -replicated units save checkpoint values at each execution round. The input handler saves the inputs it receives and the n -replicated units save state variables. Older checkpoints are replaced with new ones when the voter determines that the unit produces a correct output in the current execution round.

Using an input handler reduces the complexity of the fault tolerance process. If there is a fault, the input handler sends the checkpoint input to the faulty unit, while handing over the regular input to the non-faulty units, and the faulty unit uses the checkpoint state variables to re-execute. Without an input handler, handling a fault would require coordination across all units, including the non-faulty ones.

To support NMR with $n > 3$, the input handler should save a checkpoint for each unit/input combination, since it may be that different faulty units have different faults occurring at different but overlapping times, and therefore need different inputs. If TMR is used, the input handler can save one checkpoint per module/input combination, since at most one unit can be faulty.

Since the systems we target are capable of parallel execution, the checkpoint mechanism performs checkpoint/restart while other parts of the system continue to execute. For example, while re-execution is performed, the voter determines the new output. Furthermore, parallelism allows the performance of the application not to be affected by the fault handling process. Therefore, TUF can operate under a wide variety of input loads.

4.3 Fault Handling

Fig. 3 shows the fault handling procedure. Whenever the voter detects a fault, the voter instructs the input handler to provide the faulty unit with the checkpoint input, and the faulty unit to switch to utilize the checkpoint state variables. The checkpoint input is the one that resulted in a faulty output, while the checkpoint state variables are the ones that were saved after the completion of the

last successful round. The voter also saves what the correct output should be based on the majority voting, which we call the *reference output*.

Each faulty unit keeps (re-)executing until either the voter determines it now produces the reference output, hence the unit recovered from the fault, or the checkpoint/restart procedure reaches the *cutoff point*. This is a point in time, explained next, when the fault handling procedure switches to reconfiguration.

Reconfiguration may be performed in-place or out-of-place. Unlike out-of-place reconfiguration, in-place reconfiguration is when the faulty unit is reconfigured in the same area of the chip. For permanent faults that would survive in-place reconfiguration, out-of-place reconfiguration is necessary since the area may be permanently damaged. Periodic scrubbing performed in parallel can ascertain beforehand that the new location of the unit is fault-free [16]. A section that is scrubbed and found to be fault-free can be guaranteed to remain fault-free for a certain time period based on the inter-arrival time between faults. Therefore, based on the inter-arrival time assumed in our fault model, we can guarantee that a location is fault-free for the time needed to perform reconfiguration. Regardless of the reconfiguration type, the fault handling process is the same.

Whether the fault is handled by checkpoint/restart or reconfiguration, an additional synchronization step is possibly required. Because we allow non-faulty units to continue executing while the fault is handled, those may be several execution rounds ahead compared to the previously faulty unit. In that case, after the fault is handled, the synchronization step copies the state of the non-faulty units to the previously faulty unit, so all units eventually find themselves in the same state. Synchronization does not occur on units that continue to be faulty and has no impact on non-faulty units, since it occurs in parallel with their execution.

Crucially, TUF does *not* need to detect whether faults are transient, intermittent, or permanent. All faults are handled the same way, with the cutoff point determining when to switch from checkpoint/restart to reconfiguration. In fact, it is difficult to determine whether a fault is permanent or not, since a long intermittent fault may be indistinguishable from a permanent fault. Faults that last long enough to reach the cutoff point are handled using reconfiguration while those that do not are handled using checkpoint/restart.

The fault handling procedure necessitates that NMR units be identical. Using different designs and implementations complicates the fault handling process. For example, state synchronization between units of different implementations requires complex interfaces that are likely to be a source of faults in themselves. Similarly, the voter needs to be more complex if different implementations yield different results, which may be considered as correct within a specific tolerance margin. In cases where multiple versions of the same unit are used, TUF can still work provided there is an interface that can translate input, state, and output variables between the different versions.

4.4 Cutoff Point

The cutoff point determines when to switch from checkpoint/restart to reconfiguration. Intuitively, this is the latest point in time when a checkpoint/restart attempt can still begin while guaranteeing

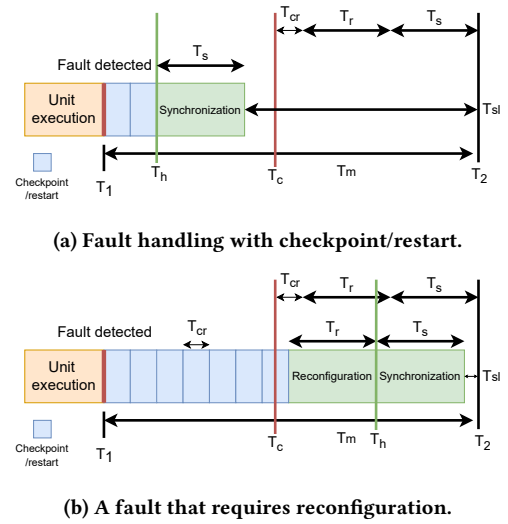


Figure 4: Determining the cutoff point T_c based on when the next fault is expected to occur and how much time is required to handle the existing fault.

that the fault handling completes before the number of faulty units reaches or exceeds the number of non-faulty units. Each faulty unit has its own cutoff point. Therefore, the unit that has been faulty the longest would be the first to be reconfigured. This is desirable since it is the most likely to have a permanent fault.

Say T_1 is when the number of simultaneously faulty units reached $\lceil \frac{N}{2} \rceil - 1$, whereas T_m is the minimum time between faults. The earliest time T_2 the next fault may occur is $T_2 = T_1 + T_m$. The fault needs to be handled by this time, otherwise, we cannot rely on majority voting. We can conservatively estimate T_1 based on the last fault-free execution of the unit because we know that the fault occurred after that time.

In Fig. 4a, checkpoint/restart succeeds in handling the fault before T_2 . Say, however, multiple checkpoint/restart attempts execute, each taking T_{cr} , and yet without successfully handling the fault, which the voter continues to signal. T_{cr} is composed of three times, $T_{cr} = T_{ih} + T_u + T_v$, where T_{ih} is the time for the input handler to provide the checkpoint input, T_u is the time for the faulty unit to re-execute, and T_v is the time for the voter to determine if the output is correct. We need a way to decide when to switch to reconfiguration, which is instead guaranteed to handle the fault. This is a function of T_{cr} , of the time to perform reconfiguration T_r , and the time to perform synchronization T_s .

Fig. 4b shows the worst-case situation. After the last checkpoint/restart attempt, we must still have the time to perform reconfiguration for T_r time units and synchronization for T_s time units. This means there should be a non-negative slack time T_{sl} between T_1 and T_2 , which means multiple checkpoint/restart attempts (unsuccessfully) executed, followed by a single reconfiguration and synchronization. The cutoff point T_c is the latest point in time we can attempt a checkpoint/restart while leaving a non-negative slack time at the end of the process, that is, $T_c = T_2 - T_r - T_s - T_{cr}$, whereas

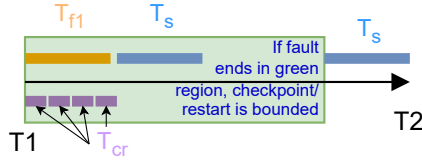


Figure 5: Time-boundedness of fault tolerance using checkpoint/restart. Reconfiguration is time-bounded since it is guaranteed to handle the fault.

the slack time is $T_{sl} = T_2 - T_h - T_s$ with T_h indicating the point in time the fault is handled.

Fig. 4b shows the scenario where a checkpoint/restart attempt finishes after the cutoff point. This means that this attempt is the final one, because an additional (possibly unsuccessful) checkpoint/restart attempt would push reconfiguration and synchronization (if needed) past T_2 . We might be lucky and additional checkpoint/restart attempts do work, yet past the cutoff point, but we cannot guarantee that the fault handling completes before the next fault occurs. Therefore, the last checkpoint/restart attempt in Fig. 4b must be followed by reconfiguration.

4.5 Time Boundedness

Provided that the target platform meets the requirements in Sec. 3, the fault handling procedure is time-bounded. We use a voter with a time-bounded execution, which can vote in just one clock cycle. To demonstrate the time-boundedness of the system, we consider two fault durations, T_{f1} and T_{f2} , where $T_{f1} \leq T_2 - T_c$ and $T_{f2} > T_2 - T_c$. Faults of length T_{f1} are handled using checkpoint/restart, while faults of length T_{f2} are handled using reconfiguration. In both cases, faults must be handled before the next fault occurs. Furthermore, since synchronization may be required, faults must be also handled such that synchronization completes before the next fault occurs. This is described by $T_h \leq T_2 - T_s$.

Consider Fig. 5 and T_{f1} first. Using checkpoint/restart, the fault is handled at time $T_h = T_1 + T_{f1}$, that is, as soon as it ends. We can combine this equation with $T_h \leq T_2 - T_s$, which reduces to $T_{f1} \leq T_2 - T_s - T_1$. This can be further reduced to $T_{f1} \leq T_m - T_s$. Based on this, we can conclude that the fault handling time is bounded by $T_h \leq T_2 - T_s$, provided that the fault duration is less than or equal to the interval between faults, minus the time required for synchronization.

We then consider the other case T_{f2} . Using reconfiguration, the fault is handled at the following time $T_h = T_c + T_{cr} + T_r$ since reconfiguration begins at $T_c + T_{cr}$, requires T_r , and is guaranteed to handle the fault. Since all of these times are bounded, the approach is also time-bounded.

5 Prototype

We implement TUF on a Digilent Arty Z7-20 board, equipped with a Zynq 7000 SoC that includes a dual-core ARM Cortex-A9 processor and 85,000 logic cells as programmable logic, which meets the requirements of Sec. 3. The fault handling procedure uses one of the two ARM cores and the programmable logic. The input handler, the NMR units, and the voter are implemented in the programmable

logic, as well as voting, checkpointing, and state synchronization. As a consequence, our implementation is vulnerable to faults in these components; we return to this limitation in Sec. 7. Also note that reconfiguration requires a sufficient number of available PL resources to successfully complete. Although the device we use already provides enough room, modern devices in the same class offer much greater available resources; as an example, the ZU19EG SoC offers around 11 times as many logic cells than the SoC we use.

The reconfiguration required to address a permanent fault is triggered by the ARM core when the cutoff point is reached. We find, nonetheless, that the cutoff point can be too pessimistic, since in many systems, fault bursts are spread far apart [41]. This is because for faults to manifest into errors, they must occur in the bits that are currently being used. Only a small minority of bits are used at any given time. Using the cutoff point defined in Sec. 4.4 would result in the system waiting hours before transitioning to reconfiguration.

We implement an additional cutoff point that allows us to anticipate triggering the reconfiguration when convenient. We compute this as $T_{c2} = T_1 + T_r$, that is, the system performs checkpoint/restart for the time it takes to reconfigure the unit, T_r , after the first fault is detected at time T_1 . The earlier of the two cutoff points is used.

6 Evaluation

We split the evaluation in four parts. Sec. 6.1 describes the test applications and reports on reconfiguration times. Sec. 6.2 reports on fault injection experiments using the Vivado Logic Simulator. We inject both transient and intermittent faults in carefully chosen configurations to also explore edge cases. In Sec. 6.3, we illustrate similar experiments using fault injection on the real Digilent Arty Z7-20 board with both specific faults injected at specific times and faults injected according to a realistic distribution as found in low Earth orbit [40]. Sec. 6.4 presents real radiation testing experiments with the Digilent Arty Z7-20 board using both neutron and X-ray radiation. Our results show that:

- (1) Reconfiguration times scale linearly with bitstream size and can be predicted accurately.
- (2) TUF successfully handles transient, intermittent, and permanent faults in bounded time.
- (3) Checkpointing handles transient or intermittent faults in as few as two clock cycles (20 ns).
- (4) Random fault injection shows that TUF copes with realistic fault rates in low Earth orbit.
- (5) Radiation testing proves TUF can handle real-world faults using both checkpointing and reconfiguration.

In the following, TUF uses a TMR configuration, that is, $N = 3$ unless otherwise specified.

6.1 Fundamental Measurements

TUF operation is *orthogonal* to the application logic. Our results are thus *not* a function of the application. Different test applications would change the amount of data that needs to be stored and re-stored, or the fault handling time. We use Verilog to implement the necessary modules, Xilinx Vivado to generate the hardware design, and Xilinx Vitis to generate the software for the ARM cores.

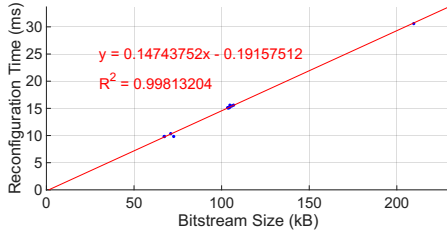


Figure 6: Reconfiguration time for different bitstream sizes.
The linear trend line closely matches the data.

We use Linear Feedback Shift Registers (LFSRs) and Fast Fourier Transform (FFT) as test applications. An LFSR’s output value generally depends only on the state. We use two LFSR modules with one feeding the other to exercise input handlers and voters by making data flow through different modules. The first module is a 16-bit LFSR whose output depends only on its state, and the second one is a 32-bit LFSR whose output depends both on the input, which comes from the 16-bit LFSR, and its own state. We implement TUF on the second LFSR. We use that as representative of applications that compute their outputs based on their inputs and state.

We test FFT as a case of long-running computation. We use the FFT algorithms generated by the Spiral Generator [30]. We use a radix-2 implementation with a transform size of 64. The implementation uses 16-bit fixed point integers. The module is pipelined. Each clock cycle, 2 complex data points are input and output, each composed of 1 real and 1 imaginary part. It takes 32 cycles for the entire 64-element transform to be input, while it takes 145 clock cycles for the data to move from the input to the output stage. Therefore, $145 + 32 = 177$ cycles are required for the entire FFT.

Throughout the experiments, we reconfigure bitstreams of different sizes. Fig. 6 shows 15 reconfiguration times we observed, including a linear trend line that closely fits the data. Reconfiguration times can be predicted accurately using the bitstream size, and can therefore be used to calculate the cutoff point. The trend line has a negative constant term, which is incorrect, and hence, would break down at small sizes. However, the trend line holds for the range of bitstream sizes we tested, from 67 kB to 209 kB. Since LFSR is a very simple example, it is unlikely for bitstream sizes to be much smaller than this in practice. Therefore, we expect the trend line to hold for most practical applications.

To evaluate the PL resource usage of TUF, we compile both the LFSR and FFT designs with TUF. Tab. 1 shows the results. LFSR is so simple that TUF uses many resources compared to it. This affects the fault susceptibility, since fault rates are proportional to component’s resource use. With FFT, the input handler and voter use significantly fewer resources than the execution units. The input handler and voter use more resources than with TMR, but the execution units use the same amount of resources.

To better understand how resource usage scales with redundancy settings, we also explore the resource usage of TUF with 5-modular redundancy (5MR), that is, $N = 5$. We do so only for LFSR. 5MR requires more complex input handlers, voters, and logic inside the units. Therefore, all units use more LUTs than the TMR case, with

Table 1: Resource usage of TUF with the LFSR and FFT designs. LFSR also includes a 5MR implementation. The values are the number of LUTs and FFs used by each component in the design. In LFSR, the TUF logic uses a large percentage of the resources, while in FFT, it uses a small percentage.

Scheme	Component	LFSR LUTs	LFSR FFs	FFT LUTs	FFT FFs
TMR	Input Handler	52	68	169	202
	Units 0-2	37-38	76	1324-1356	1397-1398
	Voter	173	39	312	82
5MR	Input Handler	157	119	-	-
	Units 0-4	58	74	-	-
	Voter	305	75	-	-

Table 2: Simulator and hardware fault injection results, grouped by outcome. Fault detection requires 1 cycle in all cases. All times are in clock cycles.

Outcome	Handling	Sync	Simulator Fault Duration	Hardware Fault Duration
Normal	2	0	1, 500, 997	1, 998, 50,000,000, 90,000,000, 99,999,997
Slow handling	5	0	998	99,999,998
Synchronization required	2	2	999, 1,000, 1,100, 3,500	99,999,999, 350,000,000

the input handler using roughly 3 times as much, the units using around 1.5 times as much, and the voter using around 1.75 times as much. Furthermore, five units are needed instead of three. All in all, 5MR uses roughly 2.2 times as many LUTs as TMR.

6.2 Fault Injection in Simulator

We inject faults using the Vivado Logic Simulator to evaluate the time required for fault handling with checkpointing/restart. We explore settings where the fault duration is limited to a single cycle, thus it is *transient*, and configurations with *intermittent* faults of variable duration, including cases that may or may not require synchronization and edge cases arising due to specific interleaving of TUF operation and application processing. We cannot evaluate reconfiguration because the simulator does not model that. The fault handling time is the time from fault detection until the unit is fault-free, but not necessarily synchronized.

We find that simulation runs over three orders of magnitude slower than the hardware. To achieve practical simulation times and sufficient experimental evidence, we trigger executions of the test application at 100 kHz instead of 1 Hz.

LFSR. Tab. 2 shows the results for LFSR. Fault detection takes one cycle (10 ns) and handling takes two cycles (20 ns). These tiny times are due to the parallel nature of TUF.

The case of a fault lasting 998 cycles is peculiar. The synchronization time depends on whether it is needed, that is, whether another input arrives before the fault is handled. As the input rate is 100 kHz, synchronization is required when the fault lasts 999 cycles or more because the two other units receive new inputs so do not have the same state as the faulty unit while the fault is handled. When the fault lasts 998 cycles, it takes 2 cycles to be handled;

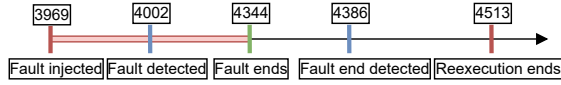


Figure 7: Intermittent fault injection timeline in FFT. Numbers above the line show the timestamps in cycles.

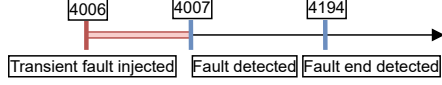


Figure 8: Transient fault injection timeline in FFT. The numbers above the line show the timestamps in cycles.

another input arrives at the exact same time the fault is handled. It takes one cycle for the voter to inform the input handler that the fault is gone. Thus, during that cycle, the input handler sends the checkpoint input value to the previously faulty unit, instead of the new input that the other two units receive. The previously faulty unit produces a different output right after producing another checkpoint/restart. This results in a fault handling time of 5 cycles, including 1 cycle for the (fictitious) detection of a new fault, and 2 cycles for handling.

FFT. We also inject at the output of one of the units an intermittent fault with a random duration of 377 cycles into the FFT implementation. Fig. 7 shows the timeline. The fault takes long to detect because the FFT module takes time to stream its output. Once the FFT module streams its output, the fault is detected in one cycle.

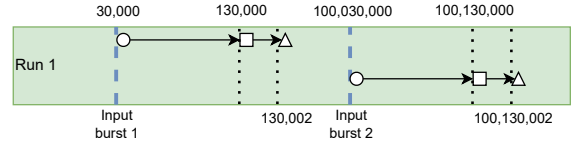
The fault's end is detected 42 cycles after it actually ends. This is because the voter must verify that the entire vector output is correct. Since 32 cycles are required to produce a single output vector, at least 32 cycles are needed to confirm. We find that the additional 10 cycles are due to the fault disappearing while an output was being produced. Thus, once the fault is gone, the fault handling time is bounded by checkpoint/restart, also for longer-running applications.

The longer computation time and pipelined design also have interesting implications. The faulty unit continues to re-execute for 127 cycles after the fault is over. These re-executions are on data already travelling in the pipeline when the input handler is informed that the fault is gone.

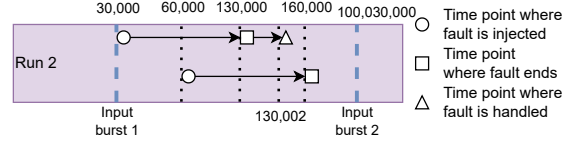
We also inject a transient fault into the FFT implementation. Fig. 8 shows the timeline. The fault is detected in one cycle, and is considered gone in the 187th cycle. This delay is also due to the data passing through the pipeline. Nonetheless, the fault handling time is still time-bounded.

Duplication with comparison (DWC). Duplication with comparison is a fault tolerance approach that uses two identical units to detect faults [42]. It detects a fault when the units have different outputs. We implement DWC to compare it with TUF. Our implementation of DWC uses an input handler and voter for managing the two units.

We inject transient and intermittent faults with FFT as test application. We find that DWC can handle transient faults well, performing identically to TUF. However, it fails with an intermittent



(a) Multiple faults in the same application are successfully handled. The legend is in Fig. 9b.



(b) The second fault is not detected, as it does not turn into an error.

Figure 9: Injection of several faults of 100,000 cycles each, into different runs. Faults injected in the same round impact different units.

fault of exactly 1100 cycles because another input arrives while the fault is still present. With DWC, we cannot detect which unit is the faulty one. Therefore, all units perform checkpoint/restart. DWC either needs to buffer all inputs that arrive in the meantime, or discard them. Furthermore, we cannot perform reconfiguration and synchronization to fix long-lasting faults. In contrast, TUF successfully outputs the correct value and synchronizes the faulty unit with the others once the fault is gone.

DWC consumes less resources than TUF with TMR, since only 2 identical units are needed and synchronization is not needed. TUF with TMR uses 4507 LUTs and 4476 FFs in total. The DWC design uses 2783 LUTs and 2944 FFs. Its input handler (139 LUTs and 72 FFs) and voter (75 LUTs and 77 FFs) use less resources than TUF.

Power consumption. We use Vivado to estimate the power consumption of different configurations. We find that the LFSR implementation using TMR consumes 1.436 W. The DWC and TMR FFT implementations consume 1.59 W and 1.677 W respectively. LFSR and FFT implementations without TUF and any redundancy consume 1.432 W and 1.507 W. Hence, the power overhead of TUF is small, even in comparison to non-redundant implementations.

6.3 Hardware Fault Injection

We check the performance of the checkpoint/restart process on real hardware, using TUF on the Digilent Arty Z7-20 board. We inject faults in the same way as in the simulator. The programmable logic is clocked at 100 MHz and the ARM core runs at 650 MHz. A dedicated time measurement unit in the PL unit counts the number of clock cycles as the execution progresses. It obtains measurements at individual PL clock cycles, which is not possible with the ARM core. This unit operates in parallel to the other units.

Tab. 2 shows the results. They closely match those of the simulator. The only variation is in the duration of faults, because the hardware implementation operates with 1 Hz input rate, instead of 100 kHz. The same observations apply here, with the corresponding scaling factor. The fault duration of 99,999,998 cycles in the hardware, for example, corresponds to the fault duration of 998 cycles

in the simulator. Both occur 2 cycles before the next input arrives and incur in a fault handling time of 5 because of the necessity of an extra checkpoint/restart attempt, as explained in Sec. 6.2.

We inject two intermittent faults into two different units. Fig. 9 shows the execution. The first run, shown in Fig. 9a, demonstrates that both faults are successfully handled. This is because the first fault is handled before the second fault is detected. The second run, shown in Fig. 9b, illustrates the successful handling of the first fault. However, the second fault is not detected. This is because the latter occurs between inputs and, therefore, never manifests into an error.

Quinn et al. [40] report an average number of SEUs/Device-Day from 12 to 18 in low Earth orbit. This is due to differences in the time spent in the South Atlantic Anomaly, an area in orbit where radiation levels are higher. They found that around 92% of events affected 1-bit, 7% affected 2 bits, and events affecting up to 6-bit occurred. We execute a fault injection experiment to observe how TUF handles these faults. Quinn et al. [40] do not report the duration of the faults. However, since they are mostly SEUs, they are likely transient.

We inject transient faults with duration of 1 cycle, according to a random distribution with an average of 18 faults per day per unit, which aligns with the worst-case evidence of Quinn et al. [40]. If a fault would occur, the probability that it affects one bit is 92%, 7% for two bits, and 1% for three bits. All three units have the same probabilities and input bursts are generated every ten cycles. We run the device for over 47 hours, and find four faults, all of which were successfully handled using checkpoint/restart within 2-5 cycles, demonstrating that TUF can handle these faults. While this appears low compared to 18 faults per day, consider that not all faults cause an error, but only faults that are present when the units are executing. For this experiment, input bursts arrive every ten cycles, and hence, each unit executes once every ten cycles. Therefore, many faults will not cause errors.

6.4 Radiation Testing

We perform radiation testing to mimic the environment of space vehicles [60], as shown in Fig. 10. The results and insights we report next are arguably unique in the literature.

Neutrons. We use an Americium/Beryllium (AmBe) source with an activity of 2.6 GBq, shown in Fig. 10a. The source generates a yield of 156,000 neutrons per second. We place the Digilent Arty Z7-20 board with TUF at about 5cm from the source. At this distance, around 1,400 neutrons reach the chip every second. No moderator was used. Although neutrons are not abundant in low Earth orbit, they would do damage in similar ways as the atomic nuclei (protons, He, and galactic cosmic rays consisting of heavier nuclei) that dominate the cosmic ray flux [60]. Much of the existing literature performs radiation testing with neutron sources [24, 25, 56].

We radiate the boards for more than five days and observe no faults. For a fault to occur, a neutron must hit the board exactly where computation occurs on the chip. The design uses a small portion of the FPGA resources: 7.3% of the look-up tables and 4.4% of the registers. As a result, the probability that a fault occurs is rather low.

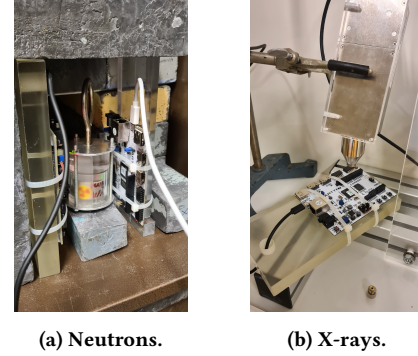


Figure 10: Radiation testing setup.

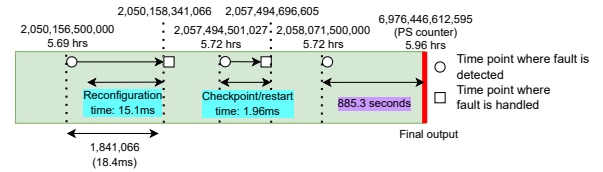


Figure 11: X-ray radiation testing results showing successful fault handling using both checkpoint/restart and reconfiguration. Time units are clock cycles where not specified. All but the final output's cycles are measured using the programmable logic counter at 100 MHz. The final output's cycles are measured using the ARM core's counter at 325 MHz.

X-rays. Since we detect no faults with the neutron source, we irradiate the board using an X-ray beam. We use an Amptek 'Mini-X' miniature x-ray source with silver target at 30kV and 200uA with the collimator removed. We point the source directly at the chip at approximately 5 cm. The dose rate is $32.4\mu\text{Sv/h}$ and the flux is $2.6 * 10^9 \text{ counts/second/mm}^2$. The radiation levels now are much higher and not ideal for generating single event upsets. However, Total Ionizing Dose (TID) radiations [50], that is, radiations that break the transistors over time by degrading their doped silicon, are achievable using X-rays and we use them to evaluate TUF response to aging effects, which can cause permanent faults.

We irradiate the board for 6.4 hours. We observe no faults until the 5.7 hour mark. Fig. 11 shows where new faults occur. TUF takes 1,841,066 cycles to handle the first fault using reconfiguration, which accounts for 15.1 ms in this case. Checkpoint/restart handles the second fault in Fig. 11 before reaching the cutoff point. It takes 1.96 ms to handle the fault. This demonstrates that checkpoint/restart handles a fault roughly one order of magnitude faster than reconfiguration.

A third fault occurs 5.77 milliseconds after the second fault is handled. This fault is never handled successfully. The last output is produced 885.3 seconds after this fault and the board appears to fail completely thereafter, even after rebooting. Our speculation is that either the FPGA logic where reconfiguration is performed or the ARM core where the reconfiguration logic runs were damaged, or the bitstream used for reconfiguration is faulty itself.

Three days later, we successfully power on the board, but faults still occur. The Vitis software fails to verify the program after flashing, indicating that memory is permanently damaged. About 20 days later, we successfully reprogram the board and bitstream verification is now valid, indicating that the hardware is healing over time. What we thought was a permanent damage, really was a long-lasting fault, probably caused by built-up charge in the Flash memory that dissipates over time. We run the board for 79.9 minutes. This time, 22,441 faults are detected; most of them are successfully handled with checkpoint/restart and the remaining 279 are successfully handled using reconfiguration. We find that also in our radiation tests checkpoint/restart successfully handles faults in as low as 2 cycles after detection.

7 Discussion

Since our approach is based on NMR, we inherit its limitations: it can only handle a certain number of faulty units at the same time. This is inherent to NMR and not specific to our work. We may mitigate this issue by extending TUF to work on individual bits instead of the entire unit. This consumes more resources, but increases fault tolerance. Faults would then need to occur at the same time and in the same bit across two of the three units for TUF to break.

Our fault model states that faults can only occur in the application functionality and not in fault handling functionality. In practice, there is no distinction between the two. Fault susceptibility in FPGAs is directly related to the size of the area the functionality occupies on the chip, which is proportional to the complexity of the design. In designs where the application functionality is more complex than the fault handling functionality, the former is more likely to experience a fault. Since fault handling functionality is smaller compared to application logic in most practical cases [8, 58], we can dedicate more resources to provide additional levels of redundancy and fault tolerance to the fault handling functionality. For example, we may duplicate the input handler and voters so that the system is duplicated throughout, with the only exceptions of its interfaces with external systems. Scrubbing may also be employed periodically to correct faults in fault handling functionality.

Our fault model also assumes a minimum inter-arrival time between faults. While this is supported by real-world data [58], this assumption may not hold in all cases. Faults tend to occur in bursts and therefore, it may be that several faults occur in a short time. In such cases, we recommend using TUF with a larger NMR setup, such as 5 or 7, to ensure that the majority voter can still determine the correct output.

8 Conclusion

We presented TUF, a time-bounded fault handling technique that combines TMR, checkpoint/restart, and reconfiguration. TUF handles transient, intermittent, and permanent faults in nano-satellites utilizing COTS hardware. Key to its fault handling capability is to determine when to switch from checkpoint/restart to reconfiguration. We find that TUF's use of checkpoint/restart functionality handles faults in as few as 2 cycles, while reconfiguration allows the system to handle more severe faults, yet in a time-bounded manner. In a real radiation environment, we concretely demonstrate that

TUF successfully handles faults of different natures using checkpoint/restart or reconfiguration.

Acknowledgments. This work was partially funded by the Knut and Alice Wallenberg Foundation through the project UPDATE. It was also supported by the Swedish Foundation for Strategic Research (SSF). The ION technology center at UU is supported by VR-RFI (Grant 2023-00155).

References

- [1] P.A. Bernstein and E. Newcomer. 2009. Chapter 7 - System Recovery. In *Principles of Transaction Processing (2nd Edition)*.
- [2] P. Bissig et al. 2017. Fast and robust GPS fix using one millisecond of data. In *IPSN*.
- [3] C. Bolchini et al. 2007. TMR and partial dynamic reconfiguration to mitigate SEU faults in FPGAs. In *DFT*.
- [4] J. Budroweit and H. Patscheider. 2021. Risk assessment for the use of COTS devices in space systems under consideration of radiation effects. *Electronics* (2021).
- [5] M.J. Cannon et al. 2020. Modeling common cause failures in systems with Triple Modular Redundancy and repair. In *RAMS*.
- [6] Mohammed Alae Chanoui, Imane Khalil, Mohammed Sbihi, Zine El Abidine Alaoui Ismaili, and Zouhair Guennoun. 2025. Trends and applications of On-Board image processing for Earth observation nanosatellites: a systematic review. *International Journal of Aeronautical and Space Sciences* 26, 4 (2025), 1989–2019.
- [7] K. Chen and G. Tan. 2017. SatProbe: Low-energy and fast indoor/outdoor detection based on raw GPS processing. In *INFOCOM*.
- [8] Bradley Denby et al. 2023. Kodan: Addressing the computational bottleneck in space. In *ASPLOS*.
- [9] G. Deng et al. 2014. A discrete event systems approach to discriminating intermittent from permanent faults. *Journal of Aeronautics* (2014).
- [10] S. Di Carlo et al. 2014. A novel methodology to increase fault tolerance in autonomous FPGA-based systems. In *IOLTS*.
- [11] E. Dubrova. 2013. *Fault-tolerant design*. Vol. 8. Springer.
- [12] S. Duzellier. 2005. *Aerospace Science and Technology* (2005).
- [13] J. Fowers et al. 2013. A Performance and Energy Comparison of Convolution on GPUs, FPGAs, and Multicore Processors. *ACM Trans. Archit. Code Optim.* (2013).
- [14] C. M. Fuchs et al. 2018. Fault-Tolerant Nanosatellite Computing on a Budget. In *RADECS*.
- [15] S. Ghosh, R. Melhem, and D. Mosse. 1995. Enhancing real-time schedules to tolerate transient faults. In *RTSS*.
- [16] R. Giordano et al. 2018. Configuration Self-Repair in Xilinx FPGAs. *IEEE Transactions on Nuclear Science* (2018).
- [17] M.A. Haque, H. Aydin, and D. Zhu. 2014. Real-time scheduling under fault bursts with multiple recovery strategy. In *RTAS*.
- [18] H. Henna et al. 2020. Towards fault-tolerant strategy in satellite attitude control systems: A review. In *Conference of the PHM Society*.
- [19] C. Hernandez and J. Abella. 2014. LiVe: Timely Error Detection in Light-Lockstep Safety Critical Systems (*DAC*).
- [20] A. Jacobs et al. 2012. Reconfigurable Fault Tolerance: A Comprehensive Framework for Reliable and Adaptive FPGA-Based Space Computing. *ACM Trans. Reconfigurable Technol. Syst.* (2012).
- [21] H. Kopetz and G. Bauer. 2003. The time-triggered architecture. *Proc. IEEE* (2003).
- [22] Koren and Su. 1979. Reliability Analysis of N-Modular Redundancy Systems with Intermittent and Permanent Faults. *IEEE Trans. Comput.* (1979).
- [23] I. Koren and C. Mani Krishna. 2010. *Fault-Tolerant Systems*.
- [24] G. Korkian et al. 2022. Single Event Upsets Under Proton, Thermal, and Fast Neutron Irradiation in Emerging Nonvolatile Memories. *IEEE Access* (2022).
- [25] F. Libano et al. 2019. Selective Hardening for Neural Networks in FPGAs. *IEEE Transactions on Nuclear Science* (2019).
- [26] S. Liu, R. N. Pittman, and A. Forin. 2010. Energy reduction with run-time partial reconfiguration. In *FPGA*.
- [27] M.N. Lovellette et al. 2002. Strategies for fault-tolerant, space-based computing: Lessons learned from the ARGOS testbed. In *IEEE Aerospace*.
- [28] F. Many and D. Doose. 2011. Scheduling Analysis under Fault Bursts. In *RTAS*.
- [29] S. Mehdi and A. Mehdi. 2020. New hardware redundancy approach for making modules tolerate faults using a new fault detecting voter unit structure. *IET Circuits, Devices & Systems* (2020).
- [30] P. Milder et al. 2012. Computer Generation of Hardware for Linear Digital Signal Processing Transforms. *ACM Trans. Des. Autom. Electron. Syst.* (2012).
- [31] S. Mukherjee. 2008. CHAPTER 1 - Introduction. In *Architecture Design for Soft Errors*.
- [32] S. Narayana et al. 2020. Hummingbird: energy efficient GPS receiver for small satellites (*MobiCom*).
- [33] P.R. Nikiema et al. 2023. Impact of Transient Faults on Timing Behavior and Mitigation with Near-Zero WCET Overhead. In *ECRTS*.

- [34] F. Österlind et al. 2012. Strawman: resolving collisions in bursty low-power wireless networks. In *Proceedings of the 11th international conference on Information Processing in Sensor Networks*. 161–172.
- [35] J.C Pickel. 1996. Single-event effects rate prediction. *IEEE Transactions on Nuclear Science* (1996).
- [36] S. Poledna et al. 2000. Replica determinism and flexible scheduling in hard real-time dependable systems. *IEEE Trans. Comput.* (2000).
- [37] B. Pratt et al. 2011. Reduced-precision redundancy on FPGAs. *Int. Journal of Reconfigurable Computing* (2011).
- [38] S. Punnekkat, A. Burns, and R. Davis. 2001. Analysis of Checkpointing for Real-Time Systems. *Real-Time Systems* (2001).
- [39] M. Qasaimeh et al. 2019. Comparing Energy Efficiency of CPU, GPU and FPGA Implementations for Vision Kernels. In *ICESSE*.
- [40] H. Quinn et al. 2012. On-Orbit Results for the Xilinx Virtex-4 FPGA. In *2012 IEEE Radiation Effects Data Workshop*.
- [41] H. Quinn et al. 2015. The Cibola Flight Experiment. *ACM Trans. Reconfigurable Technol. Syst.* (2015).
- [42] H. Quinn et al. 2017. Robust Duplication With Comparison Methods in Micro-controllers. *TNS* (2017).
- [43] F. Reghenzani et al. 2023. Software Fault Tolerance in Real-Time Systems: Future Research Questions. *ACM Comput. Surv.* (2023).
- [44] C. A. Rigo et al. 2022. A branch-and-price algorithm for nanosatellite task scheduling to improve mission quality-of-service. *European Journal of Operational Research* (2022).
- [45] A. Sari, M. Psarakis, and D. Gizopoulos. 2013. Combining checkpointing and scrubbing in FPGA-based real-time systems. In *VTS*.
- [46] Juraj Slačka and Miroslav Halás. 2015. Safety critical RTOS for space satellites. In *International Conference on Process Control*.
- [47] B. Soret, S. Ravikanti, and P. Popovski. 2020. Latency and timeliness in multi-hop satellite networks. In *ICC*.
- [48] P. Sousa et al. 2006. Proactive resilience through architectural hybridization (SAC).
- [49] P. Sousa et al. 2010. Highly Available Intrusion-Tolerant Services with Proactive-Reactive Recovery. *IEEE TPDS* (2010).
- [50] N. Tebina, N. Zergainoh, and P. Maistri. 2022. X-Ray Fault Injection: Reviewing Defensive Approaches from a Security Perspective. In *DFT*.
- [51] A. Thekkilakattil et al. 2012. Resource augmentation for fault-tolerance feasibility of real-time tasks under error bursts (RTNS).
- [52] K. Varnavas, W. H. Sims, and J. Casas. 2015. The use of FPGAs in small satellite communication systems. In *SPACOMM*.
- [53] A. Vega, P. Bose, and A. Buyuktosunoglu. 2016. *Rugged Embedded Systems: Computing in Harsh Environments*.
- [54] P. Verissimo and A. Casimiro. 2002. The timely computing base model and architecture. *IEEE Trans. Comput.* (2002).
- [55] J. Wei et al. 2011. Comparing the effects of intermittent and transient hardware faults on programs. In *DSN-W*.
- [56] A. E. Wilson, M. Wirthlin, and N.G. Baker. 2023. Neutron Radiation Testing of RISC-V TMR Soft Processors on SRAM-Based FPGAs. *IEEE Transactions on Nuclear Science* (2023).
- [57] M. Wirthlin. 2015. High-Reliability FPGA-Based Systems: Space, High-Energy Physics, and Beyond. *Proc. IEEE* (2015).
- [58] A. Yaacoub et al. 2025. Fault Tolerance in Space with Heterogeneous Hardware: Experiences from a 68-day CubeSat Deployment in LEO. In *EWSN*.
- [59] Y. Zhang and J. Jiang. 2006. Issues On Integration of Fault Diagnosis and Reconfigurable Control in Active Fault-Tolerant Control Systems. In *Fault Detection, Supervision and Safety of Technical Processes 2006*.
- [60] J. F. Ziegler and W. A. Lanford. 1979. Effect of Cosmic Rays on Computer Memories. *Science* (1979).
- [61] D. Örn et al. 2016. A novel multi-step algorithm for low-energy positioning using GPS. In *FUSION*.