Fault Tolerance in Space with Heterogeneous Hardware: Experiences from a 68-day CubeSat Deployment in LEO

Ahmed El Yaacoub
Uppsala University
(Sweden)
ahmed.el.yaacoub[†]
[†]@angstrom.uu.se

Thiemo Voigt
Uppsala University & RISE
(Sweden)
thiemo.voigt[†]

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

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

Abstract

We report on our experience deploying a CubeSat to study fault and error distributions against different fault tolerance schemes when using Common Off-The-Shelf (COTS) hardware in Low-Earth Orbit (LEO). Space radiation commonly causes faults in COTS hardware, such as bit flips in memory, which can lead to errors in a program's execution. Fault tolerance techniques can prevent faults from turning into errors. Accurately quantifying the fault and error distributions is vital for choosing an appropriate fault tolerance scheme. We equip the CubeSat with heterogeneous hardware combining a regular System on a Chip (SoC) with programmable logic resources. Based on in-orbit experiments and post-processing of logs, we check the validity of two fault models. We find the single fault model to be *valid*, encouraging the use of techniques such as triple modular redundancy. We also demonstrate, however, that the single-bit error fault model is *not valid*, which means that common techniques such as Hamming (7,4) codes should not be used. We observe that most errors are *short-lived*, allowing simple re-executions to correct them. Contrary to intuition, we also conclude that *floating-point encodings are more appropriate* to build fault-tolerance schemes in our setting, due to faults being easier to detect than in their integer counterparts. Our insights confirm existing findings in the literature while also providing new ones, while providing a foundation to conceive fault tolerance schemes for COTS hardware deployments in space.

CCS Concepts

• **Computer systems organization** → **Redundancy**; • **Hardware** → **Fault models and test metrics**.

Keywords

fault tolerance, FPGA, satellite, heterogeneous hardware

1 Introduction

Outer space is challenging for electronics [34]. Ionizing particles from radiation cause *faults* such as bit flips in memory that may, in turn, cause *errors* that manifest as undesired deviations in program execution. These may impact program correctness against stated specifications [37], possibly leading to program *failure* [30].

The design and implementation of a *fault-tolerant* system [30], that is, one where faults do not turn into errors, must be verified against specific *fault models*. A fault model represents what faults can occur, where, and when [42]. A system is said to be fault-tolerant for a specific fault model if the faults described by that model do not cause errors [30]. After deployment, however, if faults do not abide to the considered fault model, the system cannot provide guarantees on correct execution.

Challenge and goals. With the recent rise of COTS hardware deployments in space, and particularly in low-earth orbit (LEO) [14], the question arises as to what is the most appropriate fault model. We answer this question *experimentally*, by designing and building an experimental platform in the form of a CubeSat we deploy at 732 km from the Earth. The satellite carries heterogenous hardware including a *regular multicore SoC* coupled to *programmable logic (PL) resources*. Devices with PL, such as field-programmable gate arrays (FPGAs), are emerging targets for space applications [13] and also allow for parallel execution. The SoC allows for data gathering and also doubles as a reference point to compare the outputs of the PL executions. Our goal is twofold:

(1) We aim to *verify the validity of existing fault models* in LEO deployments.
(2) We seek to *measure the error distributions* of different fault-tolerance schemes.

The latter may serve to build an appropriate fault model, in case existing ones turn out not to be valid.

Methods and findings. Verifying the validity of a fault model may be achieved by deploying a fault-tolerant system with respect to a fault model and measuring faults and errors [42].

To achieve the first goal, we deploy aboard the CubeSat a fault-tolerant FFT implementation using a Triple Modular Redundancy (TMR) [52] scheme that assumes the single fault model. TMR achieves fault tolerance by triplicating functional units and using majority voting to detect and correct faults. The single fault model, which is widely used in existing literature [10, 15, 24], states that at most one fault can occur at a time. We also analyze the single-bit fault model, which states that only one bit may be faulty at any given time in any computing unit, by post-processing fault and error distributions as represented in the data from the CubeSat. We find the single-bit fault model to be not valid, but find the single fault model to be valid. This information is key to determine the fault tolerance scheme to use in a given deployment.

We achieve the second goal by observing where and how often faults and errors occur for several configurations and settings:

- We compare the error distributions of using TMR against not using TMR, and also compare different TMR voter strategies, with a focus on bit-wise and word-wise voting. We find little difference between the two since they are both fault-tolerant with respect to the single fault model, which we demonstrate to be valid.
- We compare the error distributions of different data encodings, including floating-point and integer encodings, and

observe that floating-point ones are more resilient to faults, and that the errors they produce are easier to detect.
- We compare the error distributions of an algorithm implementation in C on the regular SoC and the same algorithm implemented with PL resources. We note little difference between the two in terms of fault tolerance. However, we note that errors do not occur during execution but once data is read back from memory, which explains why the two implementations show similar error rates.
- We consider the use of Error Correction Code (ECC) memory to store output data to determine whether it presents a significant reduction in errors. We do not observe any errors in both cases, since we conjecture that faults occur after the data was obtained from memory.

Contribution. We collect a total of 68 days of experimental data, accounting for a total of 31,713,044 outputs. Our analysis provides unique insights into how space radiation impacts the operation of COTS hardware. In particular, we primarily observe that

(1) The single fault model TMR schemes build upon is *valid* in our setting; we find no cases of two or more units experiencing faults within the same execution run.
(2) The single-bit fault model is *not valid* and fault tolerance schemes based on that are not viable; we find multi-bit flips to be far more frequent than single-bit flips.
(3) Faults are primarily *short-lived*; in the absence of strict real-time requirements, re-executions that are sufficiently spread over time may naturally correct the faults.
(4) Faults affecting the floating-point data cause *much smaller or much larger errors* than faults affecting integers; contrary to intuition, this makes errors in the former easier to detect.

The remainder of the paper unfolds as follows. Sec. 2 provides background information and surveys related work. Sec. 3 illustrates the experiment design, implementation, and space deployment. Sec. 4 reports on the outcomes of the study through a series of key observations. Sec. 5 offers lessons we learned in the process of deploying such a unique experimental platform.

Our work is a stepping stone to build more effective fault tolerance schemes for space systems. Our work may be instrumental to evaluate the impact of faults on the functionality of space systems, and to explore the energy overheads of fault tolerance schemes. Also to that end, we release the entire dataset we use to obtain the results described in this paper for others to build upon [6].

2 Background and Related Work

We start with providing background information in nano-satellite designs and proceed by presenting related work across three areas. First, we survey research works that perform fault distribution experiments in LEO and contrast them against our efforts. Second, we report on research involving radiation testing, which is an alternative to testing in space. Third, we discuss existing literature on fault injection as an instrument to validate fault models.

2.1 Faults and Fault Tolerance

Faults are categorized into three classes depending on their fault duration. Transient faults last for one clock cycle [26]. Intermittent faults last for a longer period of time [26]. Permanent faults last indefinitely until action is taken to repair the system [26].

Fault models can be described on different levels. For example, a fault model described on the transistor level includes shorts and opens in transistors, and coupling between circuit nodes [7]. We focus more on the gate and functional levels. Gate level fault models include the stuck-at fault model, where a bit is stuck at 0 or 1 regardless of what value is written into it. Functional level fault models are further categorized into fault models for functional blocks (such as adders), and fault models for memory blocks [7].

Examples of types of faults that can occur in electronics are Single Event Upsets (SEUs), where charged particles change the state of a single bit [47, 48]; Single Event Functional Interrupt (SEFI), where program instructions are executed incorrectly [31]; and single event latch-up (SEL), where a short circuit occurs within a transistor causing it to enter a latch-up state where errors can occur [51]. Multiple Bit Upsets (MBUs) are similar to SEUs but multiple bits are flipped at the same time [47].

Staple fault handling techniques include re-execution, where a system reexecutes a faulty operation until the fault disappears [48]; checkpoint/restart, where a system restores the state of the system to a state prior to the fault, and then restarts execution [48]; and data redundancy methods such as Hamming codes, where extra parity bits are used to detect and correct faults [32].

Whenever redundancy is used for fault tolerance, fault detection and correction often occurs through some form of voting. This can occur in a bit-wise or word-wise manner. In bit-wise voting, there is a voter for each bit in a word [39]. In word-wise voting, there is a single voter for the entire word [39], which requires that at least two inputs are correct in their entirety. Conversely, bit-wise voting requires that for each bit, at least two bits are correct. Therefore, bit-wise voting can still result in the correct result even if two inputs have faulty bits, provided that these bits are not in the same position. Word-wise voting does have some advantages, however, since bit-wise voters may fail silently in specific cases, for example, if two bits flip in the same position [39]. The word voter would detect the words as different, successfully detecting a fault provided that the two faulty words are not identical.

Due to unreliable channels, communications use fault tolerance techniques. TCP (Transport Control Protocol) uses checksums for error detection [29]. In the Internet of Things, more specialized methods are used. For example, Wang et al. [58] use a method called trace back, where timestamps are used to identify lost packets and broadcasting is used to signal that packets were lost, so that the transmitter can resend them.

2.2 Fault Experiments in LEO

Nano-satellites, such as CubeSats, represent a new breed of space vehicle [54]. Unlike the monolithic satellite designs that dominated space operations until the 2000s, nano-satellites offer cheaper operation and allow flexible planning and mission management by independent parties. These features fueled an increasing number of nano-satellite deployments, especially in LEO, by a variety of different subjects, including private companies but also hobbyists, universities, and research institutions.

To reduce costs, nano-satellites employ COTS hardware [57]. As a by-product of this design choice, COTS hardware also provides

nano-satellites with more computing power compared to radiation-tolerant hardware [43]. The construction of radiation-tolerant hardware usually follows their COTS counterpart by several years, and therefore lags in performance and efficiency [35].

In this area, closest to our work is the work of Quinn et al. [45]. They deploy a satellite in LEO equipped with a Virtex-4 FPGA to measure fault distributions. Unlike our work, they do not explore the tradeoff between fault tolerance and resource consumption, such as comparing TMR against not using redundancy, or bit-wise voting against word-wise voting.

D'Alessio et al. [17] deploy four different SRAM memory chips in LEO orbit for three years. Comparing the fault rates of different memory chips in LEO is an important, but orthogonal problem compared to our work, which focuses on investigating the validity of fault models and comparing fault tolerance techniques.

Lovellette et al. [35] compare the use of an RH-3000 radiation-hardened (RAD-hard) processor against an IDT-3081 COTS processor in LEO. The work bears similarities to ours, such as the use of COTS hardware and the choice of benchmarks, yet we use a different class of device. We also compare different fault tolerance techniques running on the same, rather than different hardware.

2.3 Radiation Testing

Unlike deploying satellites to study fault and error distributions, which is costly and time consuming, radiation testing exposes the system to radiations in a controlled environment that approximates the radiation environment in space. The crux of the problem is how close is this approximation, given the general lack of precise information on radiation in outer space.

Duhoon et al. [20] perform radiation testing of different micro-SD cards at nine radiation intervals, with a focus on total ionizing dose. Damkjar et al. [18] perform radiation testing of four different microcontrollers. Unlike their work, we compare different implementations on the same chip, written either in a hardware description language and running on PL resources or written in C and running on a regular multi-core SoC.

2.4 Fault Injection

Fault injection is a technique where faults are intentionally programmed into the system to evaluate its behavior when faults do occur [42]. The fault injections mirror the fault model, specifying what to inject, where, and when [36].

Mao et al. [37] develop a fault injection platform for an FFT algorithm [60] implemented on a Kintex-7 FPGA. They call "essential" bits the ones that are part of the design, that is, bits that are configured on the FPGA as part of the bitstream. Critical bits are such that if they flip, they cause a functional failure of the design [33]. For example, if a flipped bit causes the device to freeze, then that bit is considered critical. Through fault injection experiments, the authors find that 25.5% of the essential bits are also critical bits. The distinction between essential and critical bits is key for our work. We only detect faults that occur in critical bits. Due to resource constraints and platform limitations, we can only detect faults through the manifestation of an incorrect output (an error).

Sari and Psarakis [50] develop a generalized fault injection platform. They test three algorithms and find that FFT experiences the highest fault rate among the tested algorithms, which prompts us to use that as well as the key benchmark in our study, as it represents a challenging case. Villata et al. [56] develop a technique for injecting faults into the configuration bitstream. The benefit of their approach is performance, since both the fault injection and the verification are performed on-device. We do not inject faults but rather measure true faults in space. We do utilize our own on-device verification approach to check whether the real-world fault yield an error, in addition to off-device verification. We cannot rely solely on on-device verification in real scenarios, as the verification itself may be faulty.

A benefit of fault injection is that it allows for testing to scale. For example, Adamu-Fika and Jhumka [8] perform nearly a million fault injection experiments. The disadvantage of fault injection is that the types of faults that can be injected are limited to the fault model. If the fault model is not representative of real systems, then the fault injection results would not accurately reflect what is observed on real systems. This further demonstrates the importance of performing experiments in real conditions, as we do.

Table 1: Goals and corresponding design choices.

Goal	Design choice
Validity of different fault models	One voter is fault-tolerant with respect to single fault model, another one is not
Compare SoC against PL	One SoC core uses a software implementation, a PL unit uses an HDL-based implementation
Use TMR or no redundancy	Use three PL units to compute the same algorithm
Compare different voters	Utilize bit-wise voters, determine in post-processing if word-wise voting yields same result
Compare different encodings	Use an additional PL unit to compute FFT using floating-point, other PL units use integers
Impact of ECC memory	Store output data in both ECC and non-ECC memory
Location of faults	Use additional voters to process intermediate results of FFT algorithms
Frequency of faults	Use an FFT PL unit to run at significantly faster rate in comparison to the rest

3 Experimental Platform

We use an Arty Z7-20 board [1], as it is equipped with PL resources similar to traditional FPGA chips such as the Artix-7 but it also comes with a dual-core ARM Cortex-A9 SoC. The PL runs at 100 MHz, while the ARM SoC cores run at 650 MHz. The A9 SoC enables comparisons between Verilog-based implementations and regular software implementations.

3.1 Design Rationale

To detect the presence of faults to critical bits, we periodically provide a fixed input signal to a deterministic algorithm. Since the input signal is fixed and the algorithm is deterministic, the output is the same every time. Variations from the expected outputs are due to faults to critical bits. Similar to existing work [37, 50], we choose Fast Fourier Transform (FFT) as the deterministic algorithm.

Tab. 1 lists the goals of our work and the corresponding design choices. We derive the different functional units in Fig. 1 from these choices. The units include a bookkeeping SoC core, an FFT SoC core, five FFT PL cores, several voters of different types, and an increment core. The FFT SoC core allows us to compare the performance of a software implementation against the performance of the PL cores. Voters are explained in more detail next. The bookkeeping SoC core

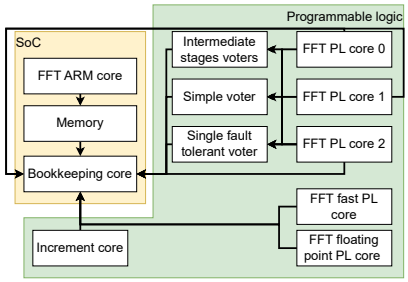


Figure 1: Hardware/software architecture.

collects the data produced by all the other cores and transmits it to the satellite's on-board computer, explained next. To transmit the data, the bookkeeping core needs to convert the data from floating-point or integer to strings of characters. Differently, the increment core is a counter that counts up every second. We use it to compare the fault rates of components that use less PL resources. We also use the increment core to evaluate the use of data redundancy, as it outputs its value four times per execution.

Tab. 2 shows the resource usage in the PL of the different cores. The design as a whole uses 45,684 look-up tables and 54,980 registers. The PL has a total of 53,200 look-up tables and 106,400 registers. Differences among identical cores, such as the three FFT PL cores, are likely due to different routing and placement. Furthermore, the first instance of a core can include some shared components, which explains why FFT PL core 0 uses more look-up tables.

We compare the measured output to the expected output in three different ways. First, we compare the measured output using the bookkeeping core against the expected output stored in its memory beforehand. Secondly, since the three PL cores in Fig. 1 execute the same algorithm at the same rate, their outputs should be exactly identical. The bookkeeping core compares them against each other. These comparisons form a coherence check to ensure both the bookkeeping core and the different units are functioning correctly. Thirdly, a communication device sends the output data to Earth, where we perform a further comparison in a post-processing stage. The different comparisons provide as much information as possible to locate faults, being either in the PL cores, in the bookkeeping core, or during communication back to Earth.

3.2 Benchmark(s)

We use three different FFT algorithm implementations, corresponding to the three different FFT algorithms part in the design of Fig. 1. One implementation is written in Verilog using 16-bit integers, one implementation is written in Verilog using 32-bit floating-point data, and one implementation is written in C using 16-bit integers. We use a publicly available Verilog implementation [2] based on

Table 2: PL resource usage per PL functional unit. Note that this does not include units running on the SoC cores.

Functional unit	Look-up tables	Registers
FFT PL core 0	1,726	1,496
FFT PL core 1	1,358	1,495
FFT PL core 2	1,353	1,495
FFT fast PL core	1,371	1,397
FFT floating-point PL core	14,181	20,163
Increment core	2	33

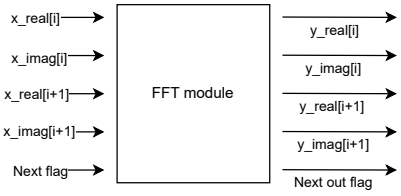
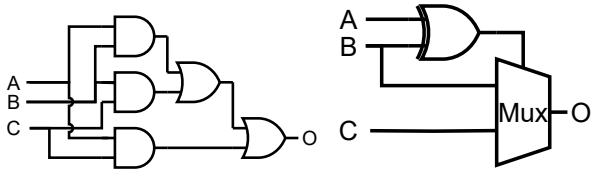


Figure 2: FFT inputs and outputs per clock cycle.



(a) Simple voter. (b) Fault-tolerant voter.

Figure 3: Voter designs.

the work by Milder et al. [38]. This is radix-2 with a transform size of 64. The C implementation is mixed-radix [12].

Fig. 2 shows the inputs and outputs of the FFT algorithms written in Verilog. Two complex numbers are input and output at each clock cycle. There are also two binary flags called *next* and *next_out* that are set to one when there is a vector of inputs or outputs generated. The *next* flag indicates that an input vector of 64 complex numbers is streaming into the FFT unit starting in the next clock cycle. The *next_out* flag indicates to receiving unit that an output vector of 64 complex numbers streams out starting in the next clock cycle.

We choose different execution rates. The three integer PL cores and the floating-point PL core are given a new input once per second. We use the same rate for the SoC implementation An additional PL core, called "fast" in Fig. 1, executes once per millisecond. This unit provides a much finer-grained sampling of the possible faults. Hardware limitations of the CubeSat we deploy, explained later, prevent us from running all cores at this rate.

3.3 Voters

To locate the source of a fault, we place voters at the intermediate stages of the FFT algorithm as well. The 16-bit integer FFT algorithm is a pipeline of 18 stages. We place intermediate voters at the inputs (stage 0), at the outputs of stages 1, 3, 5, 8, 10, 12, and 15, as well as at the final output (stage 17). We use the intermediate voters to pinpoint where in the pipeline the fault originates, using a compact encoding of fault locations we explain next.

We implement two different types of voters. One type of voter, called "simple", is based on the schematic in Fig. 3(a). This design provides majority voting but is not fault-tolerant per se. The other voter design, shown in Fig. 3(b) and operating in parallel to the simple voter, provides majority voting but is also fault-tolerant against the single fault model. Specifically, it can tolerate one fault in the voter, provided none of the inputs are faulty, or tolerate one fault in the inputs, provided the voter is not faulty. This design is based on the work by Ban and de Barros Naviner [10]. Both voters operate as bit-level voters. To compare bit-wise voting with word-wise voting, we also evaluate the output vectors from each FFT unit during post-processing to check if word-wise voting would also yield the correct output.

Table 3: Different outputs sent by the bookkeeping core.

Data	Description
Vector Outputs (128-element vectors)	
FFT 0-2	Vector output (16-bit int) for FFT PL core 0-2
Single fault voter	Vector output (16-bit int) for fault-tolerant voter
Simple voter	Vector output (16-bit int) for simple voter
ARM core FFT	Vector output (16-bit int) for FFT ARM core
FFT 0 no ECC	Vector output (16-bit int) for FFT PL core 0 stored without ECC
FFT float	Vector output (32-bit floating-point) for FFT floating-point PL core
Increment Core Output (4-element vector)	
Increment output	Vector output (32-bit int) for increment core
Fault Indicator Outputs	
FFT fast fault output	Fault indicator (32-bit binary) for FFT fast PL core
Fault out generator	Fault indicator (32-bit binary) for FFT input stage
Fault out stages 2, 4, 6, 9, 11, 13, 16	Fault indicators (32-bit binary) for FFT stages 2, 4, 6, 9, 11, 13, 16
Fault out fault-tolerant voter	Fault indicator (32-bit binary) for the FFT output from the fault-tolerant voter
Fault out simple voter	Fault indicator (32-bit binary) for the FFT output from the simple voter

3.4 Data

We collect three sets of data, summarized in Tab. 3: *vector outputs*, *fault indicators* generated by the voters, and *timestamps*.

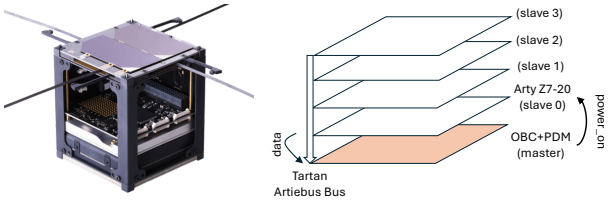
The vectors are the outputs of the FFT algorithm produced by the cores in Fig. 1, each composed of 64 complex numbers with real and imaginary parts. At each clock cycle, two outputs are computed.

Fault indicators are 32-bit numbers computed each clock cycle by the voters, including the intermediate ones. They provide a compact encoding of information about the faults in the two complex outputs produced at that stage of processing and at each clock cycle. A compact encoding is necessary as storing and transmitting down to Earth the entire vector output for all intermediate stages is unfeasible due to scarce storage resources and available bandwidth. The design of fault indicators is different depending on whether they operate as part of the fast PL core or not.

The fault indicator for PL cores running at one second period is split in two parts. Bits 0-15 correspond to the total number of faults observed in the output from Fig. 2 generated in the current clock cycle. Bits 16-30 encode an array of binary flags, each of which is set to 1 if there is at least one fault in a specific location. The locations correspond to a combination of which FFT unit (of the three TMR units) and which output (of the four from Fig. 2) shows a fault. Bit 31 is unused and always set to 0.

The fault indicator for the FFT unit running at 1 millisecond is different because its output is not voted on. Instead, we route the output to a comparison unit, which compares it to the correct values stored in memory. This indicator pinpoints where and when the first fault occurs when computing the full output vector, which takes 32 clock cycles to compute. Bits 0-3 indicate where the fault is from the four outputs in Fig. 2. Bits 4-10 bits indicate in which clock cycle of the 32 the fault occurs. Bits 11-23 indicate the total number of faulty bits. Bits 24-31 are unused and always set to 0.

Most of the PL memory buffers used to exchange data with the regular SoC are equipped with ECC. The only exceptions are the buffer storing the fault indicator data for the fast PL core and the buffer storing the vector output for the FFT PL core 0. The former must be much larger compared to its counterpart for the regular PL cores due the higher execution rate. There are insufficient memory resources in the PL to implement ECC for the larger buffer. Instead, the latter does not use ECC intentionally, as it serves to compare storing data with ECC and without.



(a) Prototype. (b) Layered design.

Figure 4: Experimental CubeSat platform.

We also collect timing information using a 325 MHz clock, since the last time the platform booted. The timestamps serve to identify when a fault occurs, and how long it takes for the fault to disappear. Due to limitations of the CubeSat we deploy, we cannot obtain absolute timing information, as explained later. The single fault model states that only one fault can occur at a time. Relative timestamps are sufficient to verify whether the fault model is valid, in that they allow to detect whether a fault lasts long enough to cross two subsequent execution rounds.

3.5 Satellite

The orbital platform we deploy is the result of a larger effort involving three academic institutions and a total of 50+ people at the different stages of design, construction, launch, and operation. The space vehicle, shown in Fig. 4(a), is built based on the 1Unit CubeSat platform of EnduroSat [3], which provides roughly 18% more internal volume available than existing CubeSat prototypes, still within the 1U form factor. The UHF Transceiver II module from EnduroSat provides downlink communications to the Earth, by relying on the SatNOGS [4] global network of satellite ground-stations for data collection. The satellite has no active propulsion system.

The hardware aboard the satellite uses a layered master-slave design, schematically shown in Fig. 4(b). The device at the bottom serves as the satellite's master On-board Computer (OBC). Its design is centered on a space-rated version of IBM's 6x86 CPU, a 32-bit computing core featuring a superpipelined architecture and hardware floating-point unit. Despite being an almost 30-year old design, its space-rated version is still deployed on satellites of various form factors as space software written for it withstood extensive testing using formal methods and throughout multiple space missions [40, 53]. This is a primary example of how space-rated hardware may lag behind modern COTS hardware. A dedicated radiation-strength aluminum shield separates each of the slave devices from each other and from the OBC.

Besides logging of primary mission-related parameters and general bookkeeping, the software aboard the OBC controls a custom Power Distribution Module (PDM) integrated within the OBC board. The PDM uses the energy coming from four CTJ30 CESI Solar cells [5] to power the OBC. These panels feature up to 29.5% efficiency and offer the largest possible effective cell area for 1U CubeSats, ultimately providing up to 2.4 W per panel in LEO. Any leftover energy is stored in a soft-reconfigurable supercapacitor

array on the PDM itself, which can vary total capacitance according to the net input power. This feature is crucial in the operation of a resource-constrained nano-satellite, as input power from the solar cells may vary widely as the orbit unfolds over time.

The OBC instructs the PDM to provide power to one or more of the four slave devices onboard. These devices are programmed by independent parties to perform various experiments under the general goal of understanding the effect of the LEO environment on COTS hardware. The Arty Z7-20 we use is one of these devices. The OBC determines what device to power among the four slaves depending on their individual energy figures and the amount of experimental data output up to a given point, in an attempt to ensure fairness of energy allocations. The slave devices relay data to the OBC through a simplified version of the Tartan Artibeus Bus [19] using a serial line.

We deploy the Arty Z7-20 board with minimal modifications, including physically removing all USB, Ethernet, and HDMI connectors and directly wiring the necessary lines to the bus. These lines are enclosed within space-hardened enclosures. The OBC is also similarly protected. Communication back to the Earth happens using space-proven FCC protocols and extreme data redundancy. Corrupted packets are filtered at the SatNOGS [4] global network as soon as they fail one of the multiple redundancy checks employed at the ground stations. This ensures that the faults we observe did happen on the board itself and nowhere else, including during communication back to the Earth. Also note that because of inherent limitations of the Tartan Artibeus Bus [19], we cannot accurately match absolute positioning information and global time, both available at the OBC, with the data produced by the Arty Z7-20 board. The analysis that follows cannot make use of this information.

The satellite was launched on November, 4th 2024 using the Polar Satellite Launch Vehicle of the Indian space research organization. It was deployed at 732 km from the Earth with initial attitude and velocity expected to ensure roughly three months of operation. The initial estimates were far exceeded as the satellite remained operational until the end of February 2025. The 68 day time span we refer to is the time the PDM actually powered the Arty Z7-20 out of the 120+ days of satellite operation. Note, however, that the data we ultimately collected does not correspond to 68 days of continuous operation of the Arty Z7-20, as energy failures on the Tartan Artiebus Bus and data corruption during communication back to the Earth reduce the net amount of data we can rely on.

4 Results and Discussion

We receive the data in seven batches over the course of 68 days. Each batch contains a variable number of data dumps produced by the Arty Z7-20 board and transmitted over the Tartan Artiebus Bus. Because of the platform limitations explained earlier, we cannot determine a relative ordering among the dumps in the same batch. We know, however, that all dumps in a batch temporarily precede all dumps in a following batch. Based on timestamp information, we determine that the Arty Z7-20 board rebooted at least four times across different batches, likely due to power losses.

Note that our measurement platform does not detect faults, but can only detect errors. These errors manifest as incorrect outputs caused by faults. We reason about the faults that caused the errors based on the errors we observe and the apriori information we have available. This distinction is important when it comes to faults in functional units, since the fault may be, for example, a bit flipping in a look-up table for an intermediate pipeline stage, causing the stage's output to be incorrect, and hence the output of the final stage also to be incorrect. We do not observe the values of the look-up tables directly, however, but rather only the incorrect output(s).

Tab. 4 summarizes the observations we draw based on the data we collect and their effects and implications, while also providing a roadmap for the rest of this section.

4.1 Fault Patterns

We observe three types of errors in the data we collect:

(1) stuck-at-0 errors: where all bits in an output array element are set to 0, so the final value of the array element is 0;
(2) bit flips: where one or multiple bits in an output array element flip from 0 to 1 or from 1 to 0, but the value of the array element is still not 0;
(3) string errors: corrupted or missing characters in the character representation of the outputs sent to the OBC.

Errors can belong to several categories at the same time. It may be unclear whether an incorrect output is due to corrupted or missing characters in the string representation, or bit flips in the original binary representation. Specific cases provide some clues: for example, if 3.1415926 is incorrectly output as 3.14126, it is more likely that '59' was not output correctly in the string representation rather than specific bits flipped in the binary representation that yield exactly the latter number. Other cases are less obvious, especially for integers, since their bit sequences can represent every integer value within a specific range exactly. Therefore, as long as the string representation is within that range and is an integer, we cannot tell whether the error occurs in the string representation or in the original binary representation.

By examining the nature of the faults, we can conclude that

Observation 1. The majority of the errors we find are bit flips rather than stuck-at-0 errors.

Indeed, 80.5% of the errors in the data we collect are definitely bit flips, while 19.5% of the errors are stuck-at-0 errors. This is encouraging since bit flips are easier to correct than stuck-at-0 errors. Generally, incorrect outputs with stuck-at-0 errors bear no relationship to the correct output, since all the bits in the array element are set to 0 regardless of what is the correct original value. If bit flips occur after the output is computed, then the incorrect output is partially linked to the correct output. Techniques such as Hamming codes can detect and correct bit flips, recovering the correct output from the incorrect one.

We return to this discussion later when we examine the actual number of bit flips we observe. Note also that this observation does not consider string errors, since it is not always clear if an error is a string error or of the other two types.

Looking at the overall number of errors in the data, we note that

Observation 2. The number of incorrect outputs is very small compared to the total number of outputs but it is in line with the expected error rates for COTS hardware in LEO.

Table 4: Summary of observations, effects, and implications.

Observations	Source	Effects and implications
The majority of the errors are bit flips rather than stuck-at-0 errors.	Observation 1	It is easier to detect and correct bitflips so data redundancy methods can be used.
The number of incorrect outputs is very small but around 2 orders of magnitude larger than in radiation-hardened chips.	Observation 2	Fault tolerance schemes must be stronger to compensate.
No error survived two or more computation runs.	Observation 3	Reeexecution with time gaps is a valid approach.
Errors did not occur at the same time in multiple execution units.	Observation 4	TMR is a valid approach provided the fault does not occur after voting. Bit-wise and word-wise voting yield the same result.
Most errors have 6-9 bits flips.	Observation 5	Data redundancy methods that correct for 1 bit flip and detect 2 bit flips are not sufficient.
Errors tend to occur in nearby bits more than would be expected by random chance.	Observation 6	It would be beneficial to spread out the bits if possible, or to have data redundancy schemes where redundancy bits are far from the data bits.
Most string errors are short-lived, within 1-3 seconds.	Observation 7	Reoutputting data with a time gap may eliminate most errors.
Floating-point outputs are better suited towards fault tolerance because they have errors typically much smaller or much larger.	Observation 8	Reasonableness tests work well to detect errors that fall outside the range when floating-point variables are used.
The increment core does not experience any errors.	Observation 9	Functional units using fewer resources are less likely to experience faults.
The on-device fault detection did not register any faults.	Observation 10	Faults occur after voting and after on-device fault detection.

We observe 236 incorrect data items out of a total of 31,713,044 data items. These items lead to 59 incorrect output vectors out of a total of 247,760 output vectors. The data covers a time period of 35,710 seconds. Therefore, one incorrect output vector occurs every 605 seconds, and one incorrect element occurs every 151 seconds.

These error rates are seemingly much higher than existing literature [46], for example, reporting 3.83 SEUs per day. However, this is expected since Quinn et al. [46] use an XQVR1000 FPGA, which is radiation-hardened and built on 220*nm* technology. In contrast, our chip is not radiation-hardened and built on 28*nm* technology. Smaller process nodes are more susceptible to faults, and hence, more likely to manifest errors [11]. The specific setting is also different, for example, as they deploy the satellite at a 560 km orbit and also find a decrease in the rate of faults by 30% at 500 km.

More generally, Lovellette et al. [35] demonstrate that COTS hardware experiences around 2 orders of magnitude more faults than space-rated hardware. In the setting of Quinn et al. [46], this would mean 383 SEU per day, which is roughly comparable with the 572 incorrect data items per day we observe at a higher orbit.

This observation demonstrates the need for fault-tolerance techniques when employing COTS hardware in space. We expect it to hold for other COTS hardware, such as sensors. Errors are not sufficiently rare that they can be ignored or handled with primitive techniques such as watchdogs and periodic reboots [26]. Adding support for automatically applying fault tolerance schemes to existing synthesis tools is arguably a necessity.

Next, we study the time distribution of errors, noting that

Observation 3. No error persists for more than a single execution.

We do find that every error is only visible in the execution where it is detected first. As a result, we also observe no intermittent faults lasting for longer than 1 s or 1 ms, depending on what PL core we consider, and no permanent faults either, since otherwise, the error would have persisted onto another execution. Intermittent and permanent faults are harder to remedy than transient faults [25].

This observation shows that time redundancy methods are effective, such as executing multiple equivalent computations are performed with some time gap between them. The gap should be set long enough for the transient fault to disappear. Unfortunately, we lack the data needed to determine how large of a time gap should be used in our specific setting. An alternative approach may be to immediately use the results of the potentially incorrect computation and take corrective actions later. Wang et al. [59] utilize such an approach for sensor faults.

By examining the time and space distribution of faults we find that

Observation 4. There exist no cases of two or more units in a TMR configuration experiencing errors within the same execution.

Crucially, this means that the single fault model *is valid*, so TMR represents a viable fault-tolerance scheme that can effectively mask faults. Observation 4 also means that *bit-wise and word-wise voting yield the same results*. Bit-wise voting masks faults when two or more units experience faults at the same time but in different bits, which does not occur in the setting we consider. Observation 4 is beneficial for distributed sensor systems, since it means that deploying three identical sensors with TMR provides fault tolerance.

Next, we study errors across multiple bits and also note that

Observation 5. Multi-bit flips are more common than single-bit flips. The majority of the errors are caused by 6 to 9 bit flips, corresponding to 20-30% of the bits in a variable.

This observation entails that the single-bit fault model *is not valid*. Fig. 5 provides concrete evidence by showing the distribution of the number of bit flips in the incorrect outputs. The data covers only the three FFT PL cores, the FFT floating-point PL core, and the simple and single fault-tolerant voters. We obtain these data by comparing the outputs against the expected values. These numbers affect what fault-tolerance schemes can be used, since many fault-tolerance schemes work based on the assumption that a few, say one or two, bits would possibly flip. For example, Hamming (7,4) codes can correct 1 bit flip and detect up to 2 bit flips. This insight, instead, prompts for much more capable schemes, for example, codes that can detect and correct a large number of bit flips, up to 40% of the bits in a variable.

Observation 5 does not align with the work of D'Alessio et al. [17], who find that the ratio of multiple cell upsets to single event

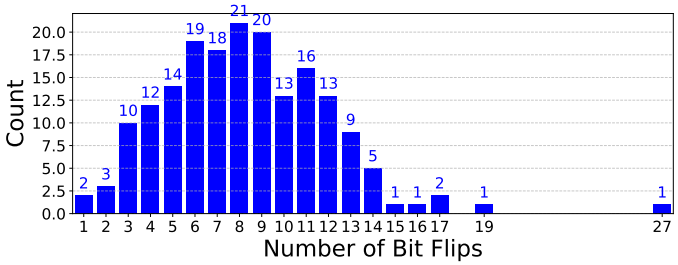


Figure 5: Number of bit flips in incorrect outputs. Most errors have 6-9 bit flips.

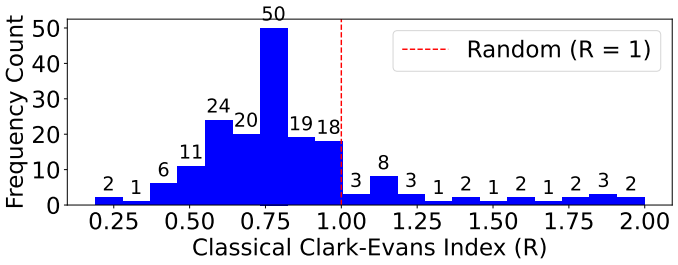


Figure 6: Clark-Evans index of the incorrect outputs. Most errors occur in nearby bits more often than would be expected by random chance.

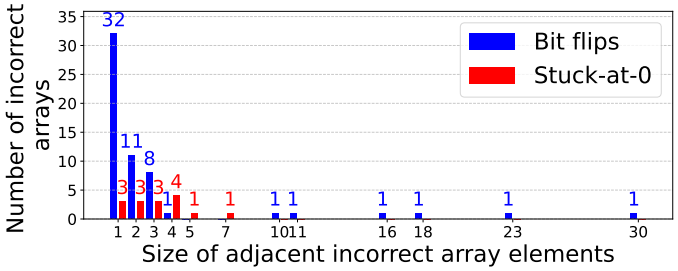


Figure 7: Errors in the number of adjacent array elements. Most errors occur in a single array element, but several occur in nearby array elements.

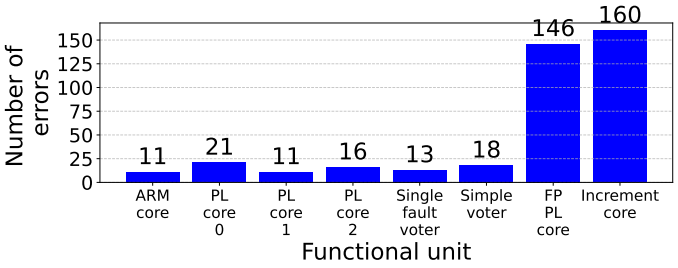


Figure 8: Errors in different output units. There is no significant difference between the PL integer units.

upsets ranges from 4.2% to 22.1% depending on the memory chip they test. It also does not match the work by Quinn et al. [45], who find that 91% of events only affect one bit. D'Alessio et al. [17] only look at memory chips and utilize memory reads and writes, while we look at the outputs of functional units, utilizing FFT. Faults in the execution or functionality are more likely to lead to multi-bit flips than faults in basic memory I/O, because the former involves more intermediate steps, where a fault in a bit in one of the steps may compound into an error with several bit flips in the final output. Quinn et al. [45] also use a 90*nm* chip, which are less dense than the 28*nm* chip we use. This contributes to our observation of a higher proportion of multi-bit flips. Heidel et al. [27], for example, note that double-bit error rates increase from 1.6% for a 65*nm* SRAM chip to 6% for a 45*nm* SRAM.

Connected to the previous observation, we also find that

Observation 6. The majority of the incorrect bits are closer to each other than would be expected for a random distribution.

We also check whether errors tend to occur in nearby bits, or tend to be more spread out. We compute the Clark-Evans index [16] for each error, and plot it in Fig. 6. The index compares the actual gaps between errors with that of a randomly distributed set of errors. It is used as a measure of the level of clustering. Values less than one indicate that the errors are closer than would be for random distributions, while values greater than one indicate that the errors are further apart. The mean index is 0.837. The 95% confidence interval is (0.790, 0.883).

Fig. 6 demonstrates that errors do tend to occur in nearby bits, more than would be expected by random chance. This information is useful for designing fault-tolerance schemes since it may be beneficial to separate important bits from each other, such as the redundancy bits. Furthermore, if sufficient memory is available, it may be beneficial to store bits from the same variable at a distance. This observation aligns with existing work [21].

We also study whether errors tend to occur in adjacent array elements or not. Fig. 7 shows the sizes of contiguous groups of errors. Most errors are of just one array element by itself. However, many do occur in 2 or 3 adjacent array elements as well.

Finally, we investigate whether the voters can mask faults or not, and if the single fault-tolerant voter is more effective than the simple voter. Fig. 8 shows the distribution of errors in different output units. The voters are not particularly more fault-tolerant than the FFT outputs. The single fault-tolerant voter is slightly more effective than the simple voter, but the difference is not significant, and the number of errors that occur is too low to conclude whether what we observe is really due to the voter. This indicates that the faults may occur after execution, once the data is in the bookkeeping core, i.e., after voting. It may also explain why we see no errors in the data that was stored without ECC, since the faults likely occur after the data is received by the bookkeeping core. Because faults occur after execution and voting, the choice of voter does not matter. Therefore, we do not have enough data to conclude whether the single fault-tolerant voter is more effective than the simple voter.

We also observe no significant difference in the number of errors between the SoC and PL implementations. This is another indication that faults occur after execution.

4.2 Data Conversion Errors

As part of the processing pipeline on our platform, data is converted to strings of characters at different stages. Converting to strings allows us to more easily differentiate between errors that occur during data transmission and other errors. This is because errors in the string representation likely produce errors where invalid or inapplicable characters are produced, such as a digit being presented

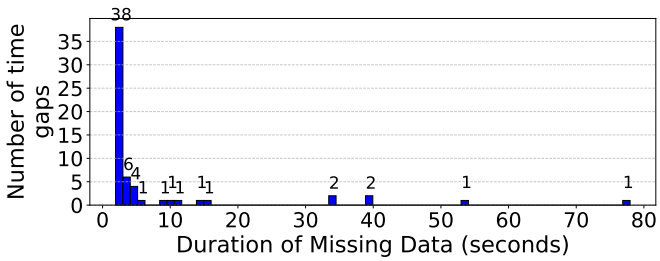


Figure 9: Time gaps between data. Most time gaps are between 1 and 3 seconds.

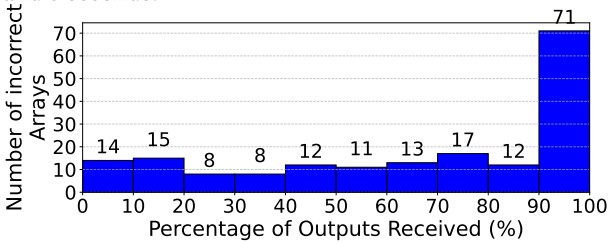


Figure 10: Number of times the output array is missing elements with the fraction of the output array produced. The majority of the output array is produced correctly.

as a letter. In this case, it is clear that the error occurred in the string representation, allowing us to more closely pinpoint when the fault occurred. If we keep the data as integers or floating point data, we lose that ability because any bit flip in an integer or floating point number leads to another valid integer or floating point number, whereas a bit flip in a string may lead to a letter or other invalid characters rather than a number.

Observation 7. String errors are usually short and usually affect few elements in an output vector.

To investigate string errors, we find gaps in the data where no data is produced as reported by the logs we collect. We do so using timestamps. Since most of our functional units have an output rate of once a second, we check whether there are gaps in the data larger than that. We plot those and their frequencies in Fig. 9. The majority of string errors last between one and three seconds.

Some string errors do not manifest as time gaps, but rather as corrupted data. Therefore, it is not sufficient to merely check the time gaps. We also check whether the entire output array is produced, since the output is of a fixed size. Specifically, 282,952 output arrays are produced correctly while 181 are not. We plot the cases where arrays are produced incorrectly in Fig. 10. Most of the time when data is missing, the majority of the elements in the array are correctly produced. There are a few cases though where very few elements in the array are correctly produced.

These results are beneficial, since repetition of data with some time gap can be used to tame these errors. Such repetition can be used in distributed sensor systems by having the sensors repeat the data several times with a time gap.

4.3 Floating-point and Integer Data

We examine how faults affect the accuracy of data encoded either as floating-point or integer data and note that

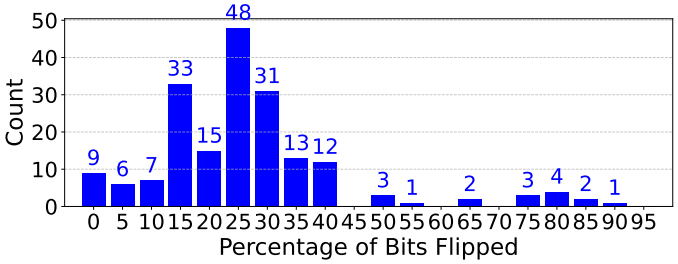


Figure 11: Percentage of bits flipped in the faulty outputs, organized in 5% bins. This includes both 16-bit integers and 32-bit floating-point data. Most errors flip 20-30% of the bits in a variable.

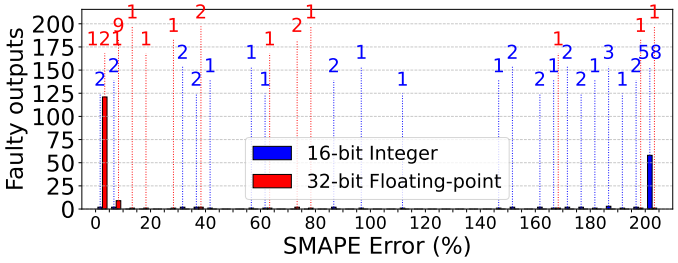


Figure 12: SMAPE of different errors between the floating-point and integer outputs. Floating-point output errors are typically smaller than integer output errors.

Observation 8. When faults occur in floating-point data, they tend to cause either much smaller or much larger errors than faults in their integer counterpart.

We plot the Symmetric Mean Absolute Percentage Error (SMAPE) between correct and incorrect outputs when using floating-point or integer encoding in Fig. 12. We plot SMAPE instead of the percentage error because the latter is not defined when the correct value is 0, which we see often. We compute SMAPE as

$$\frac{2|actual - correct|}{|actual| + |correct|} \times 100\% \quad (1)$$

and evaluate that as zero if the denominator is zero, which only occurs when both values are zero. SMAPE has an upper bound of 200% and errors larger than 100% indicate substantial errors where the error is larger than the mean of the magnitudes of the correct and actual values. We find that the majority of the errors in floating-point data deviate from the correct value by less than 5%. The majority of errors in integers incur much larger percentage differences, in comparison.

Since we have too few data points to evaluate the general impact of using floating-point and integer data, we perform a fault injection experiment by randomly injecting faults into the floating-point and integer vector outputs. Fig. 11 shows the percentage of bits flipped as a percentage of the total number of bits per array element. We use Fig. 11 as an indication that data resembles a Poisson distribution with the largest peak at around 25% and use this distribution to inject faults at random places in an output vector that is the same as the correct output vector in the 16-bit PL FFT computation. However, we use 32-bit integers instead of 16-bit ones for this experiment, so that both encodings have the same number of bits.

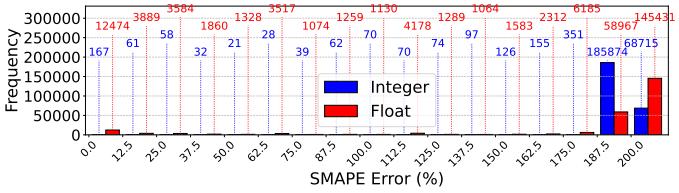


Figure 13: SMAPE in fault injection experiments in floating-point and integer data.

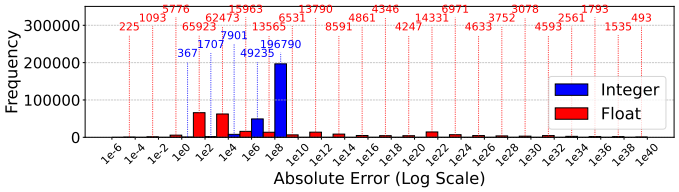


Figure 14: Absolute error in fault injection experiments in floating-point and integer data.

We show the results in Fig. 13 and Fig. 14. Unlike our results based on experimental data, this time the majority of the faults lead to significant errors in *both* floating-point and integer data. This inconsistency may indicate that the faults on the satellite occur in the string representation of the data rather than in the binary one. They may be string faults, which would yield outputs closer to the correct output in the case of floating-point numbers compared with integers, as the majority of the characters in floating-point outputs are to the right of the decimal point.

We also find that the number of fault injections with errors smaller than 12.5% is higher for floating-point data than integers. This is instead coherent with our experiments results, where we find that the floating-point data are generally more fault-tolerant. Fig. 14 also shows that floating-point errors exist across the range of possible values, while integer errors are concentrated within eight orders of magnitude of the correct value. These values are expected, since floating-point numbers have a larger range of possible values than integers. Most integer errors are concentrated from 10^6 to 10^8.

Comparisons between the fault tolerance of floating-point and integer or fixed-point data are prominent in machine learning literature. Syed et al. [55] find floating-point data to be more fault-tolerant, but they only compare 32-bit floating-point with 4-bit fixed-point data. Elliott et al. [22] perform fault injection on floating-point data and find that 90% of the absolute errors are less than or equal one, while around 9% produced non-numeric values. Unlike us, they operate on double-precision floating-point data, which explains the higher percentage of small errors.

In our experiment, any bit flip in the exponent causes huge errors while bit flips in the 20 least significant bits in the mantissa cause very small errors. We demonstrate this in Fig. 15, where we choose the number 1000 as an example of a number within our range of values and compute errors caused by flipping various bits. Errors within 5% are considered small. The exponent is 25% of the bits in single-precision but only 17% of the bits in double-precision. Therefore, it is expected that the percentage of errors less than 1 is lower in our experiments than in those by Elliott et al. [22].

Therefore, we recommend using floating-point arithmetic for computations, whenever the corresponding processing overhead

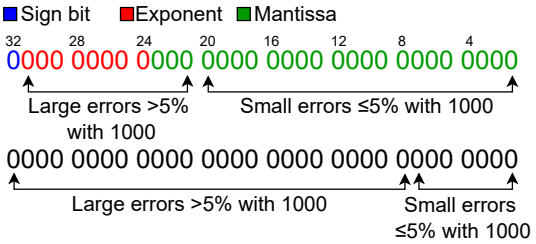


Figure 15: Floating-point (top) and integer (bottom) binary representations showcasing bits that cause large errors. The numbers above the binary number show the bit indices.

is tolerable, since faults in floating-point units more often yield small or (very) large errors, as shown in Fig. 14. Applications that can tolerate noisy data may perform effectively despite the faults causing small errors. A primary example is that of machine learning pipelines, which are robust to limited data errors in the first place [23]. When errors are large, they are easy to detect instead [49], for example, by carefully defining the correct data range and using reasonableness tests [49]. Any value outside that range is considered incorrect. One scenario where we recommend using integers is for applications where the data range does not intersect with the range from 10^6 to 10^8. In this case, integers may actually be more suitable, since the majority of their errors fall within that range, and therefore, reasonableness tests would be very effective.

4.4 Increment Core Errors

Out of the entire data we process, we also note that

Observation 9. The increment core experiences no errors.

To check errors in the increment core, we must take into account that string errors or missing outputs may make it appear as if there is an error. This occurs if, due to a string error, the increment core output is not produced. To deal with this, we rely on the fact that the increment core generates a new output every second. Therefore, we can use the timestamp of the data to check whether any of the increment outputs had failed to be output before, and what should be the current output of the increment core. We find 40 cases where this adjustment is necessary but no errors in the increment core. This result is consistent with Fig. 8. There, we record 160 errors. Since the increment core generates four values per output, and there are 40 cases where the increment core output was not produced, these 160 ($40 * 4$) mismatches are completely accounted for and caused by the missing increment core outputs.

Therefore, all errors in the increment core are due to string errors rather than errors in the core itself. This is likely due to the smaller size of the increment core, as shown in Tab. 2, consuming 2 look-up tables and 33 registers whereas the FFT PL cores each consume around 1,300 look-up tables and 1,500 registers. These results provide evidence that functional units consuming less resources are less likely to experience faults, and hence, errors. Similarly, units using less memory are less likely to experience faults.

4.5 On-device Error Detection

On-device error detection is split into two parts, error detection in the bookkeeping core, and error detection using PL voters. We

output specific messages when the on-device error detection is triggered. We find no such messages in the data, leading to

Observation 10. The on-device error detection did not register any errors during the experiment.

Lack of errors detected by the bookkeeping core may be due to faults in the bookkeeping core error detection or due to the faults not being detectable. The former is unlikely since we never register any error for the duration of the experiment. As for the latter, the bookkeeping core detects errors if the fault occurs within a specific time window, from the start of the execution of an input burst to the point at which the output is checked.

For the off-device error detection to detect an error, the fault must occur before the data is sent over the bus. This time window is smaller than the one for the bookkeeping core. Therefore, the fact that errors occurred in the off-device error detection, but not the on-device error detection indicates that the error is likely not linked to the time window, but rather occurs in the transmission of the data over the bus.

As for error detection using PL voters, the lack of errors detected may be due to the fact that the voters are not functioning correctly or may be an indication that no faults occur in the execution stage. If no faults occur in the execution stage, then that explains why we see no benefit to using either the simple voter or the single fault-tolerant voter. Furthermore, the fact that the voter output experienced errors when none of the individual three units did, also supports this hypothesis.

Certain functional modules are only checked using on-device error detection. These include the pipeline stages, the FFT fast PL core, and the FFT PL core 0's output that was stored without ECC. Therefore, if the on-device error detection is faulty, then we may have missed errors in these modules too.

Generalizability. Reconfigurable devices are becoming common in space. Since these devices share many of the same building blocks with our experimental platform, that is, a general purpose SoC with reconfigurable resources, we expect that some of our findings are relevant towards devices beyond SoCs with programmable logic. Furthermore, our results determine that faults likely occur after execution. Therefore, we expect similar errors to occur with other types of hardware accelerators, such as ASICs. These errors occurred with the second ARM core, demonstrating that moving from reconfigurable to regular hardware still produces the errors.

Observation 9 demonstrates that fault rates of modules implemented in programmable logic depend on the resource usage. When it comes to the reliability of the on-device error detection methods, we can compare the resource usage of the error detection modules with the execution modules. The integer FFT modules use around 1300-1700 look-up tables while the floating point one uses around 14,000. The voter uses around 350, and the modules to compute the fault indicator each use 65. Therefore, we expect the reliability of the on-device error detection to be around one order of magnitude higher than the integer FFT modules and two orders of magnitude higher than the floating-point FFT module.

5 Lessons Learned

The experiments we report on represent one of the very few attempts at designing, implementing, and deploying a fault measurement platform in space [17, 35, 44, 45]. Such a goal is arguably challenging per se; worse, making our device co-exist with the rest of the hardware aboard the CubeSat complicated matters. Moreover, unlike more conventional systems where multiple design iterations are possible or run-time software updates allow adjusting parameters, fixing bugs, or adding functionality, *we essentially had a single shot at this*. In the following, we articulate key lessons we learned from our experience, possibly useful for future attempts.

Energy budgeting. In our design, the time spent performing computations was fairly small compared to the time waiting for data to be transmitted or spent for the actual transmission. This reduced the time that faults may possibly occur during the computation, impacting the sampling time of the phenomena of interest.

Our design choice was intentionally conservative as we had no estimates of the energy available while in orbit, because of the inherent unpredictability of solar energy harvesting in space and for the non-predictable distribution of energy by the PDM. In hindsight, we might have pushed computation rates much higher. This indicates that a more careful energy budgeting is required, used to determine the highest computation rates one can afford.

Sharing data with the OBC. An aspect we could not study was how to relate the observed faults with absolute time information or the location of the satellite. The former would establish an exact ordering of faults, allowing one to study whether different faults are possibly correlated over time. Existing work also shows that space radiation varies significantly between locations, even at the same altitude [9]. For example, faults are more likely to occur over the so-called South Atlantic Anomaly than in other areas [28].

Because of limitations of the OBC and the Tartan Artibeus Bus, communication was unidirectional from the Arty Z7-20 to the OBC. As this happened asynchronously, there was no way to timestamp the data we produced with absolute time information or to add tags with the exact location where it was gathered. Enabling this functionality would have required a radical re-design of the entire platform, impacting both the OBC and all other devices onboard. The time required for this would have pushed the launch at least 9 months later in time, besides increasing overall costs. Absolute time information may be obtained, within reasonable accuracy, with a real-time clock. Location information, however, requires sharing data with the OBC, as equipping every different device with a separate GPS receiver would unnecessarily complicate the design and drastically increase energy consumption [41].

Data redundancy. As discussed earlier, faults did happen also where we did not expect them, that is, in transferring the outputs over the serial line. This represented a loss of precious experimental information. Note that Fig. 9 also shows that whenever data was not output over the serial line for some time, this usually happened for a limited time period.

To address this issue, we recommend adding redundancy in outputting the data. This may as simple as producing the same data multiple times with some time gap, or using different serial lines. This should decrease the number of faults occurring in situations akin to those in Fig. 10. Such an approach does increase the size of the data generated, yet the net amount of data transmitted down to Earth should not increase significantly if some compression scheme is applied. Provided no faults occur, duplicate data should be the exact same as the original data, and thus be highly compressible.

6 Conclusion

We reported on our experiments deploying a nano-satellite equipped with heterogenous COTS hardware to measure fault and error distributions in LEO. We found that the single fault mode is *valid*, which makes it possible to apply fault tolerance techniques such as TMR. By the same token, we observe that the single-bit fault model is *not valid* for COTS hardware in LEO and therefore conclude that techniques such as Hamming (7,4) codes are not viable. The faults we can detect are, moreover, mostly short-lived: in the absence of real-time requirements, re-executing functionality over time may naturally correct the faults. We finally note that the floating-point encoding is, in a sense, more appropriate to build fault-tolerance schemes in our setting, because fault detection is easier compared to integer encodings.

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).

References

[1] [n. d.]. https://digilent.com/shop/arty-z7-zynq-7000-soc-development-board/
[2] [n. d.]. https://www.spiral.net/hardware/dftgen.html
[3] [n. d.]. https://www.endurosat.com/
[4] [n. d.]. https://www.satnogs.org
[5] [n. d.]. https://satsearch.co/products/endurosat-1u-cubesat-solar-panel
[6] 2025. Satellite dataset. https://nanosat.neslab.it
[7] J.A. Abraham and W.K. Fuchs. 1986. Fault and error models for VLSI. *Proc. IEEE* (1986).
[8] F. Adamu-Fika and A. Jhumka. 2015. An investigation of the impact of double single bit-flip errors on program executions. *DEPEND* (2015).
[9] G. D. Badhwar. 2000. Radiation Measurements in Low Earth Orbit: U.S. and Russian Results. *Health Physics* (2000).
[10] T. Ban and L. A. de Barros Naviner. 2010. A simple fault-tolerant digital voter circuit in TMR nanoarchitectures. In *NEWCAS*.
[11] R.C. Baumann. 2005. Radiation-induced soft errors in advanced semiconductor technologies. *IEEE Transactions on Device and Materials Reliability* (2005).
[12] M. Borgerding. 2024. Mborgerding/Kissfft. https://github.com/mborgerding/kissfft
[13] F. Brosser et al. 2014. Assessing scrubbing techniques for Xilinx SRAM-based FPGAs in space applications. In *FPT*.
[14] G. Brunetti et al. 2024. COTS Devices for Space Missions in LEO. *IEEE Access* (2024).
[15] M. L. Bushnell and V. D. Agrawal. 2002. *Fault Modeling*.
[16] P. J. Clark and F. C. Evans. 1954. Distance to Nearest Neighbor as a Measure of Spatial Relationships in Populations. *Ecology* (1954).
[17] M. D'Alessio et al. 2013. SRAMs SEL and SEU in-flight data from PROBA-II spacecraft. In *RADECS*.
[18] S. E. Damkjar, I. R. Mann, and D. G. Elliott. 2020. Proton Beam Testing of SEU Sensitivity of M430FR5989SRGCREP, EFM32GG11B820F2048, AT32UC3C0512C, and M2S010 Microcontrollers in Low-Earth Orbit. In *REDW*.
[19] B. Denby et al. 2022. Tartan aArtibeus: A batteryless, computational satellite research platform. In *Small Satellite Conference*.
[20] A. Duhoon et al. 2021. Total Ionizing Dose Tolerance of Micro-SD Cards for Small Satellite Missions. *Small Satellite Conference* (2021).
[21] M. Ebrahimi et al. 2016. Low-Cost Multiple Bit Upset Correction in SRAM-Based FPGA Configuration Frames. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* (2016).
[22] J. Elliott, M. Hoemmen, and F. Mueller. 2016. Exploiting data representation for fault tolerance. *Journal of Computational Science* (2016).
[23] A. Fawzi, S.M. Moosavi-Dezfooli, and P. Frossard. 2016. Robustness of classifiers: from adversarial to random noise *(NIPS)*.
[24] I.A.C. Gomes et al. 2015. Exploring the use of approximate TMR to mask transient faults in logic with low area overhead. *Microelectronics Reliability* (2015).
[25] L. R. Gómez et al. 2014. Adaptive Bayesian Diagnosis of Intermittent Faults. *Journal of Electronic Testing* (2014).
[26] S. Han, K.G. Shin, and H.A. Rosenberg. 1995. DOCTOR: an integrated software fault injection environment for distributed real-time systems. In *IPDS*.
[27] D. F. Heidel et al. 2009. Single-Event Upsets and Multiple-Bit Upsets on a 45 nm SOI SRAM. *IEEE Transactions on Nuclear Science* (2009).
[28] J.R Heirtzler. 2002. The future of the South Atlantic anomaly and implications for radiation damage in space. *Journal of Atmospheric and Solar-Terrestrial Physics* (2002).
[29] S. Iren, P. D. Amer, and P. T. Conrad. 1999. The transport layer: tutorial and survey. *ACM Comput. Surv.* (1999).
[30] ISO Central Secretary. 2017. *Systems and software engineering – Vocabulary*. Technical Report ISO/IEC/IEEE 24765:2017. International Organization for Standardization.
[31] A. Ju et al. 2021. Analysis of Ion-Induced SEFI and SEL Phenomena in 90 nm NOR Flash Memory. *IEEE Transactions on Nuclear Science* (2021).
[32] U. K. Kumar and B. S. Umashankar. 2007. Improved Hamming Code for Error Detection and Correction. In *ISWPC*.
[33] R. Le. 2012. Soft error mitigation using prioritized essential bits. *Xilinx XAPP538 (v1.0)* (2012).
[34] Z. Li et al. 2023. Efficacy of Transistor Stacking on Flip-Flop SEU Performance at 22-nm FDSOI Node. *IEEE Transactions on Nuclear Science* (2023).
[35] M.N. Lovellette et al. 2002. Strategies for fault-tolerant, space-based computing: Lessons learned from the ARGOS testbed. In *IEEE Aerospace Conference*.
[36] H. Madeira, D. Costa, and M. Vieira. 2000. On the emulation of software faults by software fault injection. In *DSN*.
[37] C.A. Mao et al. 2018. An Automated Fault Injection Platform for Fault Tolerant FFT Implemented in SRAM-Based FPGA. In *SOCC*.
[38] P. Milder et al. 2012. Computer Generation of Hardware for Linear Digital Signal Processing Transforms. *ACM Trans. Des. Autom. Electron. Syst.* (2012).
[39] S. Mitra and E.J. McCluskey. 2000. Word-voter: a new voter design for triple modular redundant systems. In *VTS*.
[40] L. Mottola et al. 2010. Anquiro: Enabling efficient static verification of sensor network software. In *ICSE Workshop on Software Engineering for Sensor Network Applications*.
[41] S. Narayana et al. 2020. Hummingbird: Energy efficient GPS receiver for small satellites. In *MobiCom*.
[42] R. Natella, D: Cotroneo, and H. S. Madeira. 2016. Assessing Dependability with Software Fault Injection: A Survey. *ACM Comput. Surv.* (2016).
[43] G. Pagonis et al. 2023. Increasing the Fault Tolerance of COTS FPGAs in Space: SEU Mitigation Techniques on MPSoC. In *International Symposium on Applied Reconfigurable Computing*.
[44] C. Poivey et al. 2003. In-flight observations of long-term single-event effect (SEE) performance on Orbview-2 solid state recorders (SSR). In *REDW*.
[45] H. Quinn et al. 2012. On-Orbit Results for the Xilinx Virtex-4 FPGA. In *REDW*.
[46] H. Quinn et al. 2015. The Cibola Flight Experiment. *ACM Trans. Reconfigurable Technol. Syst.* (2015).
[47] D. Radaelli et al. 2005. Investigation of multi-bit upsets in a 150 nm technology SRAM device. *IEEE Transactions on Nuclear Science* (2005).
[48] F. Reghenzani, Z. Guo, and W. Fornaciari. 2023. Software Fault Tolerance in Real-Time Systems: Identifying the Future Research Questions. *ACM Comput. Surv.* (2023).
[49] G.K. Saha. 2005. Approaches to software based fault tolerance–a review. *Computer Science Journal of Moldova* (2005).
[50] A. Sari and M. Psarakis. 2016. A fault injection platform for the analysis of soft error effects in FPGA soft processors. In *DDECS*.
[51] J.R. Schwank et al. 2005. Effects of particle energy on proton-induced single-event latchup. *IEEE Transactions on Nuclear Science* (2005).
[52] F. Siegle et al. 2015. Mitigation of Radiation Effects in SRAM-Based FPGAs for Space Applications. *ACM Comput. Surv.* (2015).
[53] P. Stakem. 2004. Migration of an Image Classification Algorithm to an Onboard Computer for Downlink Data Reduction. *Journal of Aerospace Computing, Information, and Communication* (2004).
[54] M.N. Sweeting. 2018. Modern Small Satellites-Changing the Economics of Space. *Proc. IEEE* (2018).
[55] R. T. Syed et al. 2021. Fault Resilience Analysis of Quantized Deep Neural Networks. In *MIEL*.
[56] I. Villata et al. 2014. Fast and accurate SEU-tolerance characterization method for Zynq SoCs. In *FPL*.
[57] T. Villela et al. 2019. Towards the Thousandth CubeSat: A Statistical Overview. *International Journal of Aerospace Engineering* (2019).
[58] K. Wang et al. 2020. Adaptive and Fault-Tolerant Data Processing in Healthcare IoT Based on Fog Computing. *IEEE Transactions on Network Science and Engineering* (2020).
[59] X. Wang et al. 2021. Active fault tolerant control based on adaptive interval observer for uncertain systems with sensor faults. *International Journal of Robust and Nonlinear Control* (2021).
[60] Y. Xie et al. 2017. A novel low-overhead fault tolerant parallel-pipelined FFT design. In *DFT*.