Shields Up! Software Radiation Protection for Commodity Hardware in Space

Haoda Wang
Columbia University
haoda.wang@columbia.edu

Steven Myint
Jet Propulsion Laboratory
California Institute of Technology
steven.myint@jpl.nasa.gov

Vandi Verma
Jet Propulsion Laboratory
California Institute of Technology
vandi.verma@jpl.nasa.gov

Yonatan Winetraub
CryptoSat
yonatan@cryptosat.io

Junfeng Yang
Columbia University
junfeng@cs.columbia.edu

Asaf Cidon
Columbia University
asaf.cidon@columbia.edu

Abstract

Exponentially-declining launch costs have led to an explosion of inexpensive satellites launched to space, often equipped with off-the-shelf chips. These chips, however, lack hardware radiation protection, leaving them vulnerable to space radiation. We thus design Radshield, a software system protecting against the two most ubiquitous and costly radiation fault scenarios: (a) radiation-induced short-circuits that lead to permanent hardware failure; and (b) radiation-induced transient charges that result in single-bit silent data corruption (SDC). Radshield counters these failure scenarios with two components. First, it uses a short-circuit detector that can detect tiny increases in the device's current draw by estimating the normal current draw when resource utilization is low. Second, it duplicates the execution of spacecraft workloads in a CPU and memory-efficient manner, and catches SDCs even when they affect the CPU's pipeline or cache. In our experiments, we show Radshield is very effective at preventing both errors, and is 1.4–35.5× more power-efficient than the state-of-the-art protection mechanisms in detecting SDC. Radshield is deployed on missions in low-earth orbit and in deep space.

> *"The 9000 series is the most reliable computer ever made. No 9000 computer has ever made a mistake or distorted information. We are all, by any practical definition of the words, foolproof and incapable of error."*
>
> *HAL, 2001: A Space Odyssey*

CCS Concepts: • **Computer systems organization** → **Dependable and fault-tolerant systems and networks**; • **Hardware** → *Transient errors and upsets*; • **Applied computing** → Avionics.

Keywords: satellite computing, fault tolerance, radiation hardening

1 Introduction

The last decade has seen an explosion in commercial and public interest in space exploration, driven by exponentially-decreasing launch costs, depicted in Figure 1. The cost of launching a kilogram to low-earth orbit (LEO) in 1981 was \$88K[1] on the Space Shuttle has since dropped to just \$1.4K on SpaceX's Falcon Heavy today [1]. Private and government organizations are launching spacecraft at a rapidly-increasing rate [2], aiming to create large satellite constellations that support a wide variety of use cases from Internet connectivity [3–5], Earth imaging [6, 7], blockchain processing [8, 9], to wireless power delivery [10]. Taking advantage of decreased launch costs, LEO satellite constellations like Starlink have proven capable of providing significant performance and latency improvements over large, expensive satellites in higher orbits [11], attracting significant interest in the systems and architecture research communities [11–17].

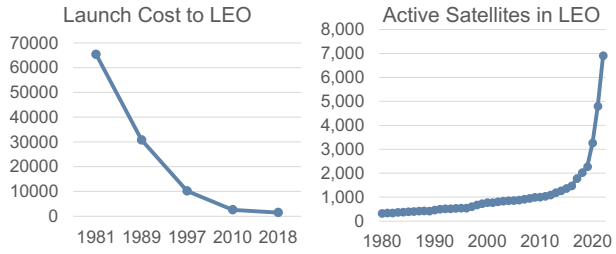

Figure 1. Cost of launching 1kg to LEO on popular launch vehicles, compared to active LEO satellite count over time.

Minimizing the unit cost of the thousands of satellites in each constellation is a key concern for operators [18]. To operate under the harsh space environment, missions traditionally use costly specialized hardware that can withstand ionizing radiation. Due to their specialized nature and their redundant hardware mechanisms, these chips are decades behind commodity ones in terms of their computational capabilities. For example, state-of-the-art radiation-hardened chips currently under development "boast" performance on the order of GFLOPS [19], while even low-power mobile chips today are capable of TFLOPS of compute [20].

Moreover, due to bandwidth limitations at ground stations [3, 4, 6–9, 21], there is increased demand to run sophisticated computations locally onboard the spacecraft [22, 23].

[1]Normalized to 2023 dollars.

However, the computational requirements of such tasks cannot be met with existing radiation-hardened hardware [24].

Thus, many missions have started to deploy commodity devices without any radiation protection [25–27]. For example, many low-cost CubeSats use Linux on off-the-shelf boards like the Raspberry Pi for flight control [26]. Even SpaceX and NASA are adopting off-the-shelf hardware for their aircraft, with the SpaceX Falcon 9 rocket running Linux on x86 CPUs [26] and the Ingenuity Mars Helicopter running Linux on a Snapdragon CPU [27]. These devices are not radiation-hardened, and are thus vulnerable to radiation-induced errors such as silent data corruption (SDC) and hardware overheating that we examine in §2.

To this end, we introduce Radshield, a system that uses software mechanisms to protect commodity hardware components from radiation effects while minimizing the performance penalty. The work on Radshield presents a collaborative effort between two spacecraft operators and academic researchers. Radshield protects against the two most common and costly error scenarios: *single-event latchups* (SELs), radiation-induced localized short-circuits in the device, which cause the device to overheat over time, eventually leading to its malfunctioning; and *single-event upsets* (SEUs), single-bit errors that can cause SDCs or crashes.

Previous work in mitigating these errors approach the software and computer system they are protecting as a black box. For example, SEL detection relies solely on the current draw to try and detect when the current draw reaches beyond a threshold [28–30]. By treating the system as a black box, these approaches experience very high false negative or false positive rates, since they are oblivious to natural variation in current draw, for example due to high CPU consumption in an application that increases the power draw.

The current approach to addressing SEUs is simply running the entire computation R times sequentially. Diverging results between runs indicates a potential SEU. However, this is computationally wasteful and multiplies the device's energy consumption and heat generation. Doing so also stresses the limited thermal and power envelopes of these satellites, affecting their productivity and uptime.

Instead, we show that with a white-box approach relying on software-visible metrics into system performance, Radshield can provide much more efficient error mitigation. Radshield consists of two primary components. *Idle Latchup Detector* (ILD) is a high-fidelity detector for SEL events. It relies on the simple yet powerful observation: since SEL events often trigger very small changes in the system's current draw, the only reliable time to observe the current draw change is when the system is idle. Therefore, ILD uses OS-visible performance counters to automatically determine when the system is naturally idle, and when it is, it uses performance counters to detect whether an SEL has occurred with minimal overhead, using a lightweight ML model.

Radshield's *Efficient Modular Redundancy* (EMR) component efficiently protects against SEU events. EMR is inspired by existing approaches which run the same computation several times [31]. Unlike existing approaches, it does so in parallel rather sequentially, relying on the idea that operators are adopting multi-core commodity CPUs. However, one cannot naively run the computation in parallel, as redundant computations may access the same memory location in a shared cache. If radiation corrupts the shared cache, it would affect all redundant runs and go undetected. To solve this problem, EMR carefully and automatically parallelizes the application to avoid the scenario where parallel tasks store data in shared caches at the same time.

In over 960 hours of testing using real-world workloads on a ground-based testbed, ILD was able to catch all induced SELs, and has a false positive rate of only 0.02%, with a runtime overhead of only 2% (§4.1). Similarly, EMR achieves the same reliability as state-of-the-art mitigations with an average of 63% less runtime overhead and 60% less energy consumption. Radshield's SEL mitigation is actively being tested on LEO SmallSats, and the SEU mitigation is deployed onboard a spacecraft on the surface of Mars (§5). We will open source Radshield's code and experiments.

2 Background and Related Work

Since the first discovery of the Van Allen belts in 1958 [32], radiation errors were known to be a factor in spacecraft system failures [33]. These errors are caused by high-energy ionizing particles hitting computer chips, depositing an electrical charge onto the chip where it is hit and displacing the silicon lattice. This deposited electrical charge may then be interpreted as a digital 1 signal where the particle strikes, while the lattice displacement can cause changes in the arrangement of the circuit within the chip.

Due to the potentially mission-ending consequences of a radiation error striking at the wrong time, the effects of radiation at various wavelengths and intensities on electronics have been extensively studied in the space exploration community [34]. From our own and others' operational experience, two types of radiation errors are of primary concern to spacecraft operators [30, 35, 36]. First, single-event latchups (SELs) cause hardware overheating and permanent chip damage. Second, single-event upsets (SEUs) manifest as bit flips or spurious signals that can cause SDC. This section describes these errors, the state-of-the-art mitigations, and our operators' experiences with them.

Incidence of radiation errors disruptive to spacecraft operations The presence of radiation errors have significant implications on the design and operation of spacecraft missions [33]. Radiation-induced latchups have been observed by multiple spacecraft missions over the past several decades, causing irreversible harm to hardware on spacecraft [37–39]. In fact, multiple commodity computers onboard one of

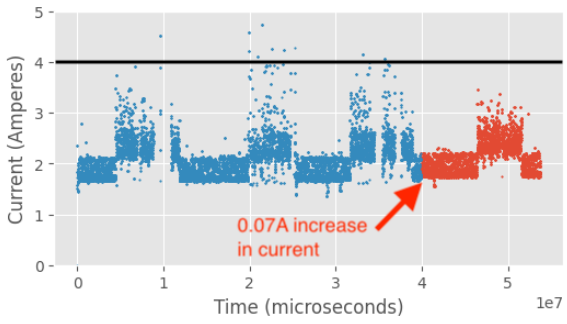

Figure 2. Current draw of a spacecraft navigation workload running on a Raspberry Pi Zero 2 W, before and after a SEL. Blue points represent nominal current draw, while red points occur under SEL. The black line represents an example threshold at 4A, past which the device will power cycle.

our SmallSats have been damaged by an SEL, rendering the entire satellite useless, incurring a significant financial loss.

SEUs have also been shown to be disruptive to spacecraft operations. Multiple flight software errors causing data loss on a Mars rover have been traced to such upsets and has put a pause on multiple days of rover operations [36]. While SEUs leading to software crashes are a nuisance, they can typically be easily dealt with by simply rebooting the device.

More alarmingly, SEUs can silently corrupt onboard data, leading to malfunctions or faulty data. Indeed, commodity hardware onboard the Ingenuity Mars helicopter experienced eight SEU-induced SDCs [36]. In our testing, a single SEU can also drop a ML model's inference accuracy from 85% to 10% [40]. Even worse, SEUs during AES encryption can leak the encryption key to attackers [41], a major security risk. We now provide more details about both types of errors.

2.1 Single-Event Latchups

The residual charge left by radiation particles can change transistor structures and cause a localized short-circuit known as SELs [42]. SELs generate a large concentration of energy on a few gates [43], causing excess heat that cannot be dissipated in the vacuum of space. This causes a sharp increase in the temperature around these gates, which will damage the chip if left unchecked [38]. Fortunately, this error can be instantly fixed by simply power cycling the affected IC. Note that power cycles are not equivalent to a reboot, as reboots may not completely clear out the SEL's residual charge.

Classically, SELs present as large, 1A order-of-magnitude increases in current [44]. However, the shrinking process nodes used for newer computers have introduced the possibility of micro-SELs, which are SELs that might increase much smaller increases current draw. For example, one previous work showed the possibility of SELs causing just a 0.07A jump in current draw on a 7nm process node [45].

Current approaches State-of-the-art approaches to detecting these small SELs treat the hardware as a black box, and simply use time-series analysis on current draw to detect latchups [30, 46, 47]. However, a 0.07A SEL-induced additional current draw [45] is negligible compared to current variations in modern CPUs due to power scaling. For example, normal current draw ranges from 1.7–4.5 A on a commodity ARM SoC, a variance over two orders of magnitude higher than an SEL. This is illustrated in Figure 2, where a CPU under load is shown with and without simulated SEL conditions. Even under SEL, the threshold of 4A is never reached, while high compute activity before the SEL crosses the threshold. Thus, a static current threshold is not sensitive enough to SELs, incurring either too many false positives or false negatives.

More sophisticated methods use ML models to detect SEL-induced current spikes [30], but again treat system resources (i.e. CPU, memory bandwidth consumption) as a black box. Thus, these models to have a high misdetection rate when applied to CPUs (§4.1), as they cannot account for system tasks (e.g. log rotation, interrupts) that also cause current spikes. Current state-of-the-art SEL mitigations are thus inadequate for detecting non-obvious latchups, and has led to (often fatal) SEL-induced damage onboard many satellites [33].

2.2 Single-Event Upsets

The residual charge left by ionizing radiation can also cause SEUs, or a change the logical state of a circuit [48]. Most SEUs result in a bit flip in memory or a spurious signal traveling down a compute pipeline [42]. Previous work shows that SEUs can cause SDC, crashes, and hangs [49].

We observe that for data at rest, commodity storage used on spacecraft computers featuring built-in single-error correcting and double-error detecting (SECDED) error-correction codes (ECC), which provide ample protection from SEUs. Many modern spacecraft SoCs also have DRAM with SECDED ECC, minimizing the chance of SDC for data in memory as well. However, commodity compute pipelines and caches lack any ECC mechanisms, even for single-bit errors. Thus, the compute pipeline and CPU cache are the primary source of application-visible failures in a commodity space computer, and are therefore our focus.

Experimental observations of SEU frequency Simulations using state-of-the-art analysis analysis [50] show that SEUs are expected to flip 1.6 bits per day day on the Snapdragon 801, a commodity SoC used onboard the Perseverance Mars rover [27]. During regular operations, a radiation-hardened RAD750 on the rover records around one SEU each Mars day (24.7 hours), and at least 4 SEUs caused system crashes on the Snapdragon 801 in the past 800 Mars days.

Current approaches The state-of-the-art approach to protect against SEUs treats the computer and program as a black box, and simply relies on running a program multiple

times [51]. This is done through *triple modular redundancy* (3-MR), which allows for the machines to do a tiebreaker vote if the results differ [35, 52, 53]. Another approach involves storing checksums of critical memory values, which are recomputed every time memory is written to and verified every time the memory location is read [54–57]. Both approaches are computationally expensive and draw significant power.

While some work has been done in enabling parallel multicore 3-MR [31, 58], these approaches focus on control flow correctness instead of data flow. However, as data flow may not depend on control flow, these approaches may miss some SEUs. Furthermore, data with the same memory address may be fetched from unprotected intermediate caches, meaning SEUs in cache could affect multiple executions.

An alternative approach relies on application-specific mitigations, such as those for deep learning [59, 60], PDE solving [55], and numerical analysis [61]. While these methods are far more effective and efficient than 3-MR or checksumming, they require significant developer time and effort to implement, and protects only a single class of compute. Operators will also need a more generalized approach to support the wide array of compute tasks onboard spacecraft.

2.3 Radiation Failures on Earth

Most SEL-inducing radiation particles are either deflected by Earth's magnetic field or absorbed by its atmosphere [38], and are thus are not a concern for datacenters on Earth. SEUs do occur on Earth [62], albeit much more rarely, at a rate of $2.3 \cdot 10^{-12}$ per bit per day at sea level, a 700,000× lower rate than in space. However, as datacenters employ tens of thousands of servers, they do likely cause small disturbances in datacenter environments. Indeed, some recent work investigates failures in the cache and execution pipeline of cores of datacenter servers, which might be traced to SEUs [63–65].

However, the high cost of 3-MR coupled with the rarity of SEUs on Earth means such techniques are rarely deployed in datacenters. Byzantine fault tolerance systems are commonly used in datacenters to address such failures [66], but these approaches require replicas and a decentralized system, and thus do not work in a single-processor environment. Approximate computing introduces some radiation tolerance to programs [67], but are not suitable for critical processes such as flight control on spacecraft.

3 Design and Implementation

We now present the design of Radshield, which takes advantage of software-visible OS metrics to efficiently and accurately detect and mitigate radiation errors (Figure 3). Radshield is divided into two components, each targeting one type of radiation error, which work together to increase the overall lifetime of the hardware (and the spacecraft).

Design Principles Our design is guided by the following principles.

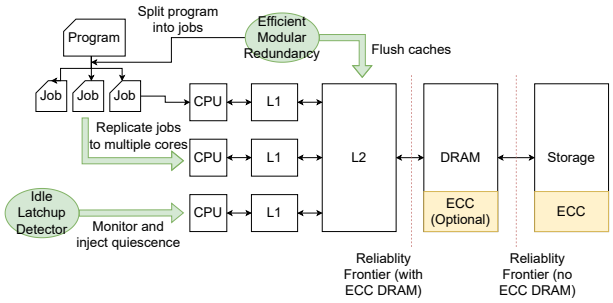

Figure 3. System design of Radshield. Components are protected by hardware (yellow) or software (green). A red line denotes potential reliability frontiers.

1. Runs on unmodified Linux. To facilitate deployment on standard commodity hardware, Radshield should use a userspace software-only design that can be run on standard OSes such as Linux. As many commodity chip suppliers provide support only for common OSes (e.g. Linux, VxWorks), engineering and compliance needs often require spacecraft operators to use unmodified Linux on their target computers.

2. Focus on userspace errors. We focus on protecting against errors in the computational pipeline in userspace, and not in the kernel, since the types of applications running in space *spend very little time in kernel space*. This is because space workloads are CPU-intensive, and spacecraft run few processes at a time, requiring few context switches, thus incurring low kernel overhead. In all of the experiments we ran in §4, we found that the amount of time spent in kernel space is ∼0.01% of the total runtime. Furthermore, spacecraft operators are reluctant to touch kernel code, as modifying the kernel can cause unrecoverable boot errors in space. To minimize the chance of disruption from a kernel SEU, Radshield is pinned to specific threads with maximum priority and uses as few kernel calls as possible. Given these operational restrictions, Radshield is a *best-effort* mitigation that minimizes risk from the most vulnerable operations in spacecraft compute.

3. Efficiency. As power is scarce onboard spacecraft, Radshield should be as efficient as possible, and must incur minimal runtime and power usage overhead on applications. As radiation hits the chip randomly and uniformly, running faster with a smaller memory footprint is also correlated with lower risk of failure [54]. Our operators also tend to use commodity hardware as a way to offload expensive compute operations from hardened primary flight computers [68]. Thus, we focus on minimizing runtime while ensuring the correctness of the computation.

4. Immediately deployable. Radshield should require little integration effort from developers and operators, and support a wide range of spacecraft workloads.

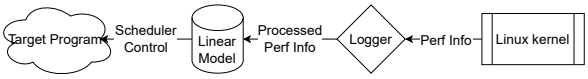

Figure 4. Software design of Radshield's ILD component.

Per-core instruction completion rate	Per-core branch miss rate	Per-core CPU frequency
Per-core bus cycle rate	Per-core cache hit rate	Disk read and write IO count

Table 1. List of metrics used in ILD's linear model.

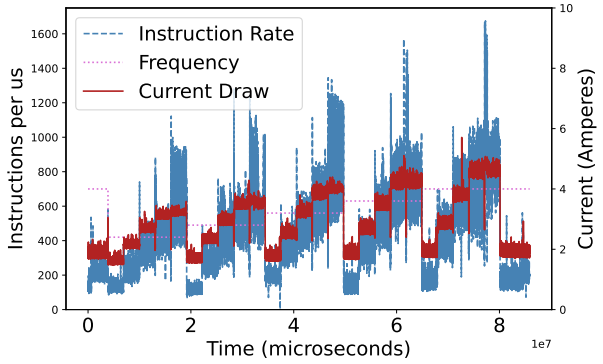

Figure 5. Current draw of a matrix multiplication workload overlaid on CPU frequency and instruction completion rate on a Raspberry Pi Zero 2 W. This experiment cycles between using 0-4 CPUs at increasing frequency steps of 100MHz, and shows a 99.7% correlation between current draw and CPU usage in a CPU-heavy task.

3.1 ILD: White-Box SEL Detection

The SEL mitigation system of Radshield is *ILD* (Figure 4), which detects latchups using current draw and system metrics that are available to the OS, and triggers reboots when an SEL is detected. ILD uses software-available counters to *estimate* the computer's current draw and compare it with the real, measured current draw, attaining far more accurate SEL predictions than state-of-the-art techniques. Our main insight is that to increase prediction accuracy, we detect SELs only when the spacecraft is in *quiescence*, where no workloads are running. We observe that quiescent periods occur frequently in spacecraft, which run computations very intermittently, due to intermittent communications with the ground. In case such quiescence has not occurred naturally, ILD actively injects short idle periods during long-running workloads.

False negatives vs. false positives State-of-the-art thresholding solutions are currently tuned for SELs with around 1A of additional current draw. However, previous work on modern process nodes show that SELs can draw as little as 0.07A of additional current [45]. Thus, we aim to detect unexplained increase in current draw at this magnitude. We note that the cost of a false negative (losing the spacecraft) far outweigh the cost of a false positive (a spurious reboot), so our first-order goal will be to minimize false negatives.

Detection time granularity Flight experiments show that a CPU under SEL takes around five minutes to be damaged by heat. Thus, we set our configurable detection window for SELs to three minutes by default, which accounts for false negatives or extraordinary situations where the thermal headroom may be lower. The naive solution to this issue will be to simply reboot every five minutes, which guarantees that any potential SEL will be cleared. However, such an approach will also prevent long-running jobs from finishing in a timely manner, making this naive solution untenable.

Current monitoring Most modern spacecraft power supplies already include a device which provides per-component current draw and voltage information. ILD uses this device to measure current directly, rather than relying on CPU counters. This is because CPU counters do not actually measure current draw, but instead use a pre-programmed multiplier of the power state and the voltage to report a current estimate. These CPU counters are thus useless for detecting SEL-induced current draw, which do not affect CPU voltages.

Using power consumption to detect SELs As previous work in detecting anomalous power behavior relies on energy usage rather than current draw, we must make a distinction between these indicators. Since energy usage is the integral of current draw times voltage, subtle increases in current draw may be overshadowed by noise in the voltage readings from CPUs switching between power states. Energy usage is also often reported by CPU counters, which as explained above do not actually measure current draw.

OS-visible metrics to estimate current draw As shown in Figure 5, compute activity can be measured with CPU performance counters, such as cycles elapsed and instructions completed per measurement interval. These performance counters are readily-accessible to userspace programs running on Linux.

Armed with this observation, we select a set of `perf` counters, listed in Table 1, to represent per-core compute load. These counters were chosen by first creating a random forest to model current draw, and then selecting the most important features in the resulting random forest model. We found that the instruction completion rate, bus cycle rate, and CPU frequency were by far the most correlated with the computer's total current draw. Similarly, the branch miss and cache hit rates reflect DRAM current draw, and the disk read/write IO reflects the hard disk's current draw. ILD thus collects these metrics once every millisecond.

Spacecraft compute load patterns Spacecraft tend to stay in an *quiescent state* for the vast majority of the time [22, 69, 70], which we define as the target application not running or suspended, while normal OS or housekeeping tasks are still being run. This is because real-world spacecraft tend to work in bursts due to the unpredictable and short communication windows in space [71]. After each "burst," the spacecraft returns to idle until the next set of commands are received at the next communication window [72, 73].

Only detecting SELs during quiescence As discussed previously in §3.1, SELs may be very hard to detect at high CPU loads due to high variance in current draw. ILD addresses this issue by recording system metrics and detecting SELs only when the CPU is quiescent. As described in §5, we use CPU load to determine when the system is quiescent. Applications may also signal to ILD when they are no longer processing data and the system is quiescent.

This scheme works consistently since system maintenance tasks tend to be short and require much less compute than the payload software, so quiescent current draw tends to stay relatively constant compared to during the workloads. For example, the standard deviation during the workload shown in Figure 2 is 0.96 A, compared to just 0.14 A during quiescence. This property holds even after heavy workloads are run, which matches how CPUs work at the architectural level: as current draw is solely determined by the pipelines in use, a (correctly functioning) CPU will not have residual current draw from past computations.

Injecting quiescent time during long jobs To provide a clean measurement environment for the prediction model during long-running jobs, ILD injects three-second "bubbles" of quiescence that temporarily pauses active processes. These bubbles ensure that SELs can be detected even if they occur during a workload. If no SEL is detected during a bubble, ILD institutes a pause period of three minutes, where no bubbles are injected. Thus, the worst-case overhead adds a three-second quiescence segment to every 180 seconds of compute, which is a $3 \div 180 = 2\%$ increase in runtime.

Training a model to detect SELs Initially, we tried using classification algorithms such as naive bayes and random forest on OS metrics to sort between nominal and SEL states, but these proved to be computationally expensive and imprecise. In the end, we adopted a simple linear model which was both efficient and accurate. This model is trained on quiescent data including instruction rate, CPU frequency and cycles elapsed, bus cycles elapsed, branch miss ratio, cache hit rate, and R/W IOs issued to disk.

Satellite operators typically test programs on an Earth-based identical copy of the hardware onboard a satellite, which allows for ILD to be trained before the satellite is launched. The linear model's predictions are then compared to the real-world satellite's current draw, and a running average difference is recorded. We experimentally determined that a >0.055A average difference between real and predicted currents for more than three seconds was an ideal threshold for flagging a potential SEL and rebooting. Specifically, a difference between 0.04A to 0.08A was tested against simulated datasets in 0.005A increments, and 0.055A presented no false negative rates while minimizing false positive rates.

Accounting for current spikes ILD must differentiate between SELs, which incur a permanent increase in current draw, and compute-induced transient spikes which only last for a few microseconds, shown in Figure 5. To decrease the effects of these transient spikes, ILD tracks a rolling minimum current across the $250\mu s$ before and after the measurement. This lowers the standard deviation of current recordings during quiescence from .14A to .02A, allowing us to more effectively target differences of .07A. While this incurs a delay of 2.5ms for each measurement, this delay still orders of magnitude smaller than our three minute time window for detecting a SEL.

Larger current spikes on the order of 1A are already addressed by additional thresholding circuitry available on most modern spacecraft power supplies in use today [74].

3.2 EMR: Efficient SEU Detector

We now describe *EMR*, a runtime and programming model for space applications written in C++, which automatically manages and optimizes 3-MR and checkpointing.

Example application We observe that typical workloads run on spacecraft, such as encryption or image processing, tend to run the same computation across different subsets of a dataset. As a guiding example, we use a global localization image processing algorithm currently run onboard a real-world spacecraft, which determines the location of an object [68]. To do so, every possible N-by-N pixel subset of a large global map is matched against a local map, as shown in Figure 6. The most optimal matching is then used to determine a spacecraft's likely location.

Sequential vs. parallel redundancy An advantage of running in parallel on one device is that it can reduce the application's overall runtime, thus reducing the time window for an SEU to strike. However, consider the image processing example: if two cores run in parallel on overlapping data (e.g. one of them reads block 3 and the other block 4), the shared part of the two blocks might be loaded in their shared L2 cache. If an SEU impacts the L2 cache, the corruption may propagate to both cores, causing both to produce the same incorrect output, while the SDC goes undetected.

To avoid this issue, EMR takes advantage of data access patterns of common algorithms onboard spacecraft to schedule computations more efficiently than 3-MR. EMR ensures that data currently being processed by a particular executor

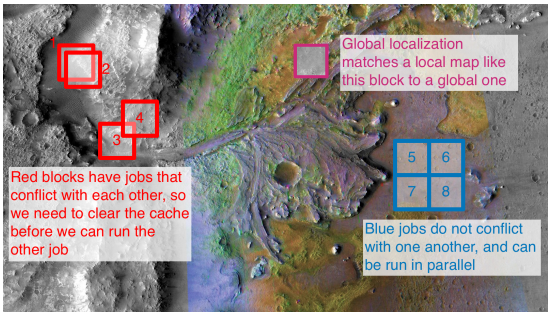

Figure 6. An image of Mars' Jezero Crater, with examples of subimages processed by our use case overlaid.

```
typedef pair<size_t, void *> DTSSInput;

struct InputData {
  vector<DTSSInput> inputs;
  DTSSInput output;
  bool operator==(InputData& b);
};

dtss_compute(unordered_set<InputData *> dataset,
             void (*processor)(InputData *));
```

Figure 7. Snippets of EMR's user API. The developer provides `InputData` structs and the job processing function, and `dtss_compute` automates scheduling and execution.

is read independently from an ECC-protected source, providing the same reliability as 3-MR with a single computer.

Dividing compute into jobs and jobsets EMR requires an algorithm which runs the same computation on multiple subsets of the input data. In our global localization use case, the algorithm runs on every possible cropping of the image, shown as blocks in Figure 6. Each sub-image is a *dataset*, a subset of the data used by one computation. As shown in Figure 7, a developer using EMR specifies datasets as a set of memory regions each computation uses as input (Figure 8), which EMR then automatically replicates as needed.

In EMR, the computation itself is expressed as a *job*, which describes a single run of the target algorithm on one dataset. To better manage conflicts, each job is bound to a core, and as such each dataset has three jobs associated with it. Therefore, each dataset would have three associated jobs, one per CPU. The developer expresses the job as a function, and EMR automatically handles scheduling these jobs.

To maximize parallelism, EMR finds sets of jobs that can run simultaneously without reading data directly from non-ECC protected memory (e.g. the CPU cache). For example, blocks [5,6,7,8] do not overlap and can be accessed simultaneously. On the other hand, EMR cannot schedule jobs using the red datasets in Figure 6 together (e.g. blocks [3] and [4]),

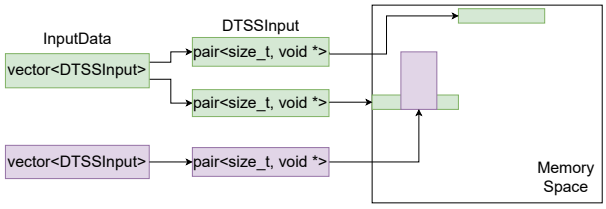

Figure 8. Example mapping of `InputData` structs to memory regions. Purple and green `InputData`s conflict with overlapping memory regions.

as they overlap and need to access the same memory during their jobs, which means they might share (unprotected) cached data. We define such intersections as *conflicts*. Two jobs are in conflict if any part of their dataset requires the same memory access. In our example, as any job that shares even a pixel with another job conflicts with another, each N-by-N-pixel dataset has up to N^2 conflicting datasets.

EMR automatically detects overlapping regions in the input structs and assigns them as conflicts. Developers may also opt to express algorithm-specific conflicts that EMR may not detect by writing their own detection function.

Jobs are then grouped into *jobsets*, which are sets of jobs that do not conflict. For example, the blue datasets shown in Figure 6 may compose part of a jobset. By only running one jobset at a time, EMR ensures that an SEU will only affect one of the executors. Invalidating the cache of the previous jobset will then ensure that no cache SEUs will affect the execution of the next jobset. EMR greedily creates jobsets by assigning jobs to the first available jobset without conflicts.

Reliability frontier We observe that when ECC is available, which is always the case on commodity flash storage and in many cases also in commodity DRAM, it can correct the vast majority of SEUs. However, some older commodity SoCs that have been deployed into spacecraft, such as the one Radshield is currently implemented on in space, do not have ECC DRAM chips [27]. To this end, we define the *reliability frontier* as the last layer of a system that has hardware protections and can be trusted, as shown in Figure 3. EMR uses this property to assume all data stored on an ECC-protected medium is protected, and does not need to be replicated. Thus, only the part of the program that runs on unprotected hardware needs to be replicated. These are the CPU pipelines, the CPU cache and the DRAM (when DRAM ECC is unavailable). Furthermore, if the reliability frontier is at storage, the page cache must also be cleared before proceeding to account for potential SEUs in DRAM.

Depending on the reliability frontier, the input data and program results will either be stored in DRAM or in storage. The memory or storage overhead of this method should be easily handled by the host computer, as in the use cases identified in §4.2, output data of our target programs tend to

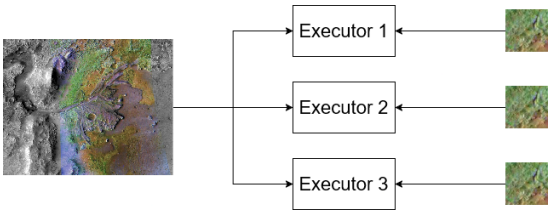

Figure 9. The optimal replication scheme for the image processing use case, where the overall map is shared while the search image is replicated to minimize cache clears.

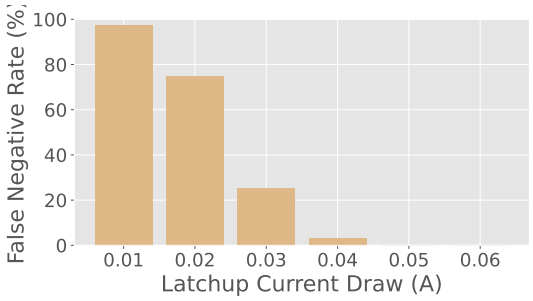

Figure 10. ILD's misdetection rate as latchup current changes. The false negative rate falls to zero as additional latchup current increases beyond 5mA.

be either the same size, or much smaller than the input data. The overhead will be further minimized by having the same instance of input data be shared across all three executors. As all data is ultimately stored within the reliability frontier, input data and output data will not be affected by SEUs. While SEU may strike while data is being replicated from the reliability frontier, this should only affect one of the executors, allowing the other two executors to outvote the erroneous one during execution.

We note that some processors also provide ECC in cache, though not the CPU pipelines. In such a case, EMR simply reverts to 3-MR, as the data in the shared caches no longer need to be replicated.

Minimizing cache clears by replication In many spacecraft algorithms (e.g. image processing or encryption), a common data block such as a target image or encryption key needs to be shared across all executors. As clearing the cache in between every single run cancels out the benefits of parallelization, this common block can simply be replicated locally by each executor in separate memory locations. This ensures reliability without needing cache clears on the replicated memory area, as an SDC in one of the replicated locations would not affect the other two executors.

EMR detects this "common data" by looking for datasets within the input data with identical pointers and offsets. EMR then replicates identical elements with a frequency above some developer-specified threshold across all three executors. By default, we use a threshold of 0.01 (a data element present in at least 1% of the input data), which we experimentally determined to be the most optimal configuration shown in Figure 9. We evaluate different values for the common data threshold in §4.2.4.

Runtime implementation During runtime, EMR is be divided into four threads: three executors and one orchestrator. The orchestrator ensures that worker threads never access datasets with overlapping regions at the same time, which may cause corrupted data to be used by multiple workers.

The worker threads receive input jobs and processes them in order, returning the output data to a shared location within the reliability frontier. After a job completes, the worker flushes the cache lines related to that job. As each job in a jobset accesses a unique memory region, no flushes in a jobset will overlap. When a jobset completes, all potentially conflicting memory regions will have already been flushed. This amortizes the runtime of the cache clears into the execution time, thus minimizing the total runtime.

EMR reserves a full core, or set of cores, for each executor instance. Specific core groups are pinned to each executor instance, thus isolating faults within specific cores to the related instance. For example, if an SEU occurs within one core's ALU, only that instance will produce an incorrect result. However, as the other two executors will never use that core, they will still produce the correct result, out-voting the incorrect result. This design allows for programs to take full advantage of CPU-specific pipelines such as NEON and AES, while preserving the correctness properties of 3-MR.

4 Ground Evaluation

Before integrating Radshield into real-world flight software, we must first ensure that Radshield works correctly on the ground. However, real-world testing requires allocating time on extremely expensive particle beams [25], which still cannot reliably induce our target errors due to the tiny size of transistors and imprecision of the ion beam. Thus, our ground-based evaluation relies on synthetic fault injection methods to answer the following questions:

- **Q1:** Can ILD accurately detect SELs (§4.1.3)?
- **Q2:** What is the performance impact of ILD (§4.1.4)?
- **Q3:** What is the runtime performance of EMR (§4.2.3)?
- **Q4:** How does the reliability frontier affect runtime (§4.2.4)?
- **Q5:** What is the energy efficiency of Radshield (§4.2.5)?
- **Q6:** How vulnerable is EMR to an undetected SEU compared to 3-MR (§4.2.6)?
- **Q7:** What is the developer overhead of Radshield (§4.2.7)?

4.1 SEL Detection (Q1–Q2)

4.1.1 Experimental setup and workload Our testing was done on a Raspberry Pi Zero 2 W with a minimal Linux system running. In order to accurately model usage onboard

	ILD	Random Forest	Static Threshold 1.75A	1.80A	1.85A
False negative rate	0.00%	35.0%	38.2%	54.5%	62.1%
False positive rate	0.02%	62.4%	40.6%	34.0%	27.6%

Table 2. Accuracy of ILD in detecting latchups.

Measurement Overhead	Reboot-Only Overhead
+72 seconds / hr	+ 72.91 seconds / hr

Table 3. Worst-case overhead of ILD per hour of compute when all quiescent periods are induced and when a false positive reboot is triggered.

Reliability Scheme	Relative Area Protected
None	0%
Unprotected parallel 3-MR	75%
3-MR	100%
EMR	100%

Table 4. Relative protected circuit area for our reliability schemes, based on die areas on a Snapdragon 845.

spacecraft, non-essential services related to Wi-Fi and Bluetooth were disabled. This system exactly matches that of our LEO SmallSat satellites. An INA3221, connected to the Raspberry Pi with I2C, was used to measure the system's current draw. A potentiometer, a variable resistor, was added between V_{dd} and *Gnd* in parallel with the Rasperry Pi. The potentiometer causes detect additional current draw that the INA3221 detects, accurately simulating the effects of an SEL.

We tested Radshield's latchup detection on a real-world flight software workload [75]. We emulated latchups by increasing the current draw by +0.07A every 30 minutes over 960 total hours. Quiescence was induced every three minutes to allow ILD to sample the quiescent current.

4.1.2 Baselines We use two baselines for latchup detection described in §2.1: (a) static current thresholds, and (b) a naive ML approach [30]. A static threshold was set to various current draw levels based on quiescent current draw. The ML approach uses a random forest classifier trained on current draw under emulated SEL and during quiescence. Note that this model treats the system as a black box and is trained solely on current draw and not on performance counters.

4.1.3 Latchup Detection Accuracy (Q1) Table 2 presents our results on the latchup detection workload. ILD achieves a 0% false negative rate, or in other words, no latchups were missed by ILD. Compared to other ML methods for latchup detection, ILD is far better at detecting SELs due to its use of system metrics. As the other methods were trained solely on current draw and have no temporal element [30], they cannot distinguish transient high currents from SELs.

We ran a second experiment where ILD was given one minute of increased power draw between +0.01A to +0.1A in increasing order, and every SEL detection trigger was counted. As shown in Figure 10, ILD incurs no false negatives as long as the additional current draw incurred by the SEL is over 0.05A. This is well below the minimum experimentally measured SEL current of 0.07A [45]. Thus, ILD is very unlikely to misclassify a SEL and damage the spacecraft.

Furthermore, as shown in Table 2, ILD experienced a false positive rate of only 0.15% over 960 hours of testing. Thus, in production, we expect to see one spurious reboot due to a false SEL alarm every 22 hours. As most spacecraft tend to stay in quiescence for a large portion of each day [69], from our conversations with operators an additional reboot daily is acceptable for most missions, given that the alternative is irreversibly damaging the spacecraft.

4.1.4 Performance Impact (Q2) During testing, ILD's overhead while under load increased runtimes of representative applications on average by 3% due to induced quiescence. Additional CPU usage during quiescence was not recorded, showing that ILD's overhead is comparable with other common system maintenance tasks. ILD's worst-case overhead, when bubbles are always induced, is shown in Table 3.

4.2 SEU Mitigation (Q3–Q7)

4.2.1 Experimental Setup and Baselines We test the energy usage and runtime of EMR compared to two baselines: sequential 3-MR, and an "unprotected" parallel 3-MR that does not clear the cache. Unprotected 3-MR does not protect against SEUs in the shared cache, leaving about 25% of the die area (see Table 4) vulnerable to errors. Unprotected 3-MR acts as a benchmark to measure how far EMR is from "optimal" performance. By default, unless stated otherwise, we assume the device under test has ECC DRAM.

4.2.2 Workloads As shown in Table 5, our testing methodology for EMR centers on five use cases commonly found onboard modern spacecraft [9, 17, 22, 71, 76]. These workloads exercise a range of compute pipelines and job conflict styles. For example, the DEFLATE algorithm in our compression benchmark relies on data from the block directly preceding it, whereas the AES-256-ECB encryption benchmark only uses data from the block being encrypted. We also test various math pipelines, including SSE2 in our image processing use case and AVX2 for our DNN benchmark.

These tests replicate memory if it is used in 1% or more of input data across jobs, resulting in the replication strategies listed in Table 5. §4.2.4 describes how we found this threshold to be optimal across different use cases.

The image processing benchmark was run on a flight-tested mobile ARM processor [68], while all other workloads were run on a flight-tested x86 CPU [77].

Worload	Library	Replication Strategy
Encryption	OpenSSL	Replicate key
Compression	Zlib	No replication
Intrusion detection	RE2	Replicate search pattern
Image processing	OpenCV	Replicate match image
Neural networks	N/A	Replicate model weights & biases

Table 5. List of tested workloads, along with the corresponding state-of-the-art library used and optimal replication strategy for each workload.

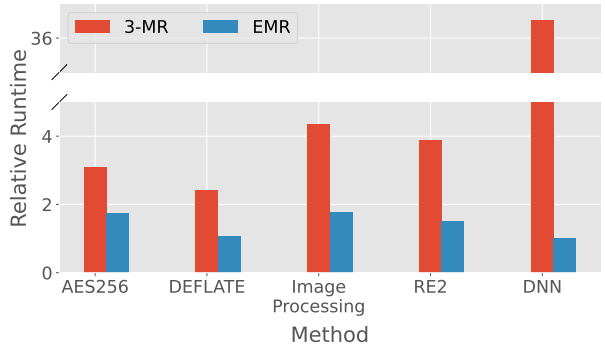

Figure 11. Relative runtimes of serial 3-MR and EMR on the DRAM reliability frontier, normalized to an unprotected parallel 3-MR baseline.

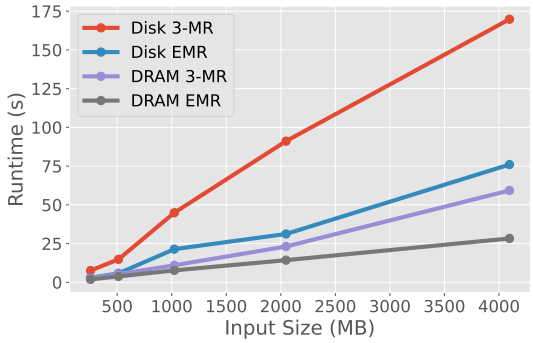

Figure 12. Effect of input sizes on runtime of AES-256, using EMR and 3-MR on both DRAM and disk reliability frontiers.

4.2.3 Performance (Q3) Figure 11 compares the runtime of EMR to 3-MR, normalized against unprotected parallel 3-MR. EMR runs significantly faster than protected 3-MR in all workloads, as EMR multithreads compute and amortizes cache clears. However, both approaches are slower than the unprotected parallel 3-MR, since they clear the cache between jobsets. EMR's slowdown is also relatively modest: 7–77% higher compared to the unprotected baselines.

Figure 12 compares the runtime of EMR and 3-MR on the DRAM and disk frontiers using the encryption workload. The results show that 3-MR is consistently slower than EMR across both frontiers, and the runtime difference is more pronounced as input size increases. While it is much slower to have the reliability frontier to be on storage rather than DRAM, EMR is still significantly faster than 3-MR.

Operation	3-MR	EMR
Disk Read	1.8s	0.6s
Memory Allocation	0.7s	0.7s
Compute	2421s	987s
Cache Clear	22s	30s
Total Runtime	2,514s	1,019s

Table 6. Runtime of image processing algorithm with a DRAM reliability frontier, divided by operation.

4.2.4 Impact of Replication (Q4) We tested EMR's performance on three of our workloads that access the same memory locations across many jobs (e.g. the encryption key discussed in §3.2) while varying the threshold for which memory gets replicated across jobs. The results shown in Figure 13. 0% replication amounts to serial 3-MR, as the shared portion of each dataset cannot be accessed simultaneously, leading to all jobs conflicting. Similarly, 100% replication is a fully-protected version of parallel 3-MR consuming 3× more memory. These tests were selected as they vary in both conflict graph density and amount of shared data. We see that each use case has a "sweet spot" that minimizes runtime and memory usage. For example, the image processing workload worked best when the full image is not replicated, but the image to be matched was. Similarly, encryption worked best when the data was shared, but the key was replicated.

4.2.5 Energy Efficiency (Q5) Figure 14 shows the energy consumption of Radshield with DRAM ECC. We compare 3-MR to EMR only, and Radshield, which includes both EMR and ILD running together. Encryption and packet processing workloads show the lowest relative energy usage, while DNNs in EMR draw more energy than in 3-MR. This is because DNNs require more cache clears to avoid jobset conflicts. Thus, conflict count in a program is correlated with energy usage. ILD's energy overhead is minimal, with only a marginal increase compared to running EMR only.

4.2.6 Vulnerability to SEUs (Q6) The window of vulnerability method introduced by Borchert et al. [54] provides an estimate of vulnerability to radiation in a uniform radiation environment. Essentially, the chance that a SEU impacts a running application is proportional to the area of the target chip is multiplied by the runtime of the application. For the image processing use case, the runtime of EMR is 40% the runtime of 3-MR (Table 6), but about twice as much die area on our ARM chip is active during EMR, as outputs are replicated in memory. Thus, the chance a SEU occurs during EMR in a uniform radiation environment is $0.4 \cdot 2 = 80\%$ of the chance a SEU occurring during the same application in 3-MR. Furthermore, as shown in Table 6, 96% of the runtime is spent in compute, where EMR will mitigate SEUs. Similarly, SEUs

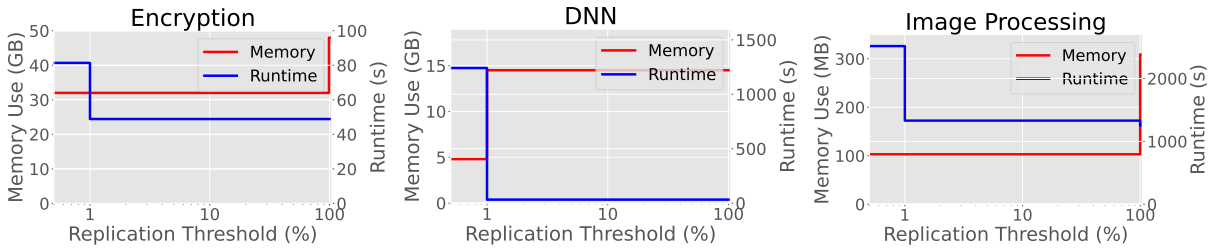

Figure 13. Impact of size of replicated portions on program memory and runtime in EMR.

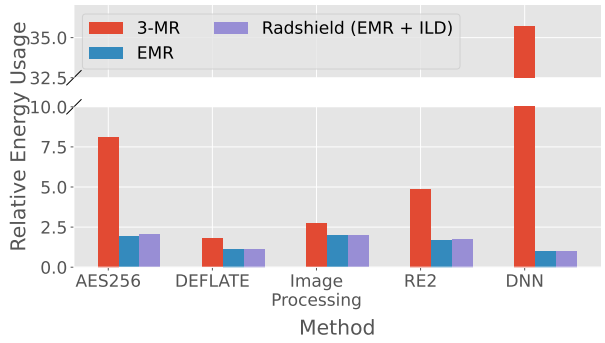

Figure 14. Comparison of relative energy usage of serial 3-MR, EMR, and Radshield on the DRAM reliability frontier, normalized to a parallel 3-MR baseline.

Scheme	Corrected	No Effect	Error	SDC
None	0	8	9	3
3-MR	3	16	1	0
EMR	2	17	1	0
EMR + MBU	2	8	0	0

Table 7. Results of fault injection into our OpenCV workload with EMR, 3-MR, and no redundancy.

during the 3% of runtime spent on memory allocation and cache clears will result in application-visible errors, leaving just the 0.06% of time spent on disk reads vulnerable.

To test this, we ran a synthetic fault injection test on our image processing workload 20 times for each redundancy scheme using a GDB-based fault injection tool [36]. SEU(s) were randomly injected within the runtime of the program, following a uniform distribution based on each component's runtime and memory overhead. To simulate MBUs, two bits were randomly flipped instead. As discussed in §3, SEUs were not injected into the kernel memory, but covers the memory space of the executors and the orchestrator. EMR and 3-MR did not incur SDCs, as shown in Table 7. This is because the window of vulnerability in these schemes lies only in the final comparison of the executor outputs, which takes a minuscule amount of time compared to the compute. However, in one case for both EMR and 3-MR, a pointer in a job being sent to an executor was corrupted and resulted in segfault, which we define as a detected error.

We did not simulate error injection into the cache, as our fault injection tool is based on QEMU, where the memory model is a "pool" of memory instead of a multi-level cache. Though we simulate the cache by tracking memory accesses in QEMU, fault injection into locations currently in the "cache" would also modify the associated memory value. A possible alternative was to use an architectural simulator such as Gem5 [78], but such simulators are very slow (it takes about 15 minutes to just boot Linux with Gem5).

Operation	Net line change
Encryption	8
Compression	6
Image Processing	7
Packet Matching	9
DNN	9

Table 8. Code changes required to implement EMR on different workloads from a 3-MR implementation.

4.2.7 Developer overhead (Q7)

We note that EMR does have a slight development cost in that a developer must integrate this library, while 3-MR can be implemented with a simple loop. However, if code was written in a way that allows for the repeated computation to be easily extracted, only minimal changes are required to implement EMR, as shown in Table 8. Many of these changes actually simplify the code, as EMR handles a significant amount of setup code for developer. In fact, the most significant developer effort is in labelling input and output data for EMR to resolve.

5 Real-World Deployment in Space

We now describe our experience deploying Radshield onboard two active spacecraft. ILD runs in observational mode (without yet the ability to reboot) onboard an LEO SmallSat mission running Raspberry Pi Zero 2 W. EMR also protects a Snapdragon 801-based coprocessor [76] onboard a Mars rover. While these missions represent both ends of the cost spectrum, they share common deployment challenges.

As radiation errors tend to strike randomly, it is hard to regularly test Radshield after its been deployed. For example,

a two-hour run of a custom tool designed to detect SDCs onboard the flagship mission on Mars did not encounter any SEUs. Due to operational concerns that result in low active time for these programs, neither deployment has detected a radiation error yet. However, the Radshield implementation of the global localization algorithm deployed on Mars, and described in §3.2, only uses 26% of the runtime of the non-parallel, radiation hardened approach [68].

Increased visibility into radiation errors After an SEU occurs, its effects range from simply being overwritten to causing significant, irreparable damage to the spacecraft. Operationally, this wide range of possible effects makes it hard to narrow down a failure to an SEU. For example, on a compute element onboard our flagship mission that did not implement EMR, we can only detect SEUs during analysis of downlink data, where checksums do not match. In contrast, as EMR's executors "vote" on a correct solution, it is much quicker to isolate SEUs in an EMR system, as disagreements between executors are detected at the next comparison stage.

Similarly, it was difficult to conclude that SmallSats were damaged by latch-ups, as the commodity computer simply stops responding after burning out. A radiation-hardened monitoring chip also onboard the SmallSat was used to diagnose the SEL, as all components other than the commodity computer worked correctly. To this end, we designed ILD to provide additional insight into these errors by recording fine-grained telemetry which allows ground operators to definitively trace a potential issue to a SEL.

Data collection efforts This work marks the start of a multi-year data collection effort. We aim to provide the academic community with a public dataset of these errors, along with traces and descriptions of the effects of each error on the mission. While some of the data must be redacted due to the sensitivity of our collaborators' missions, we hope that the data collected on these errors will provide additional insight into how systems behave in high-radiation environments.

6 Conclusions

Renewed interest in space exploration has led to a surge in new spacecraft missions, resulting in quickly-developing constellations of LEO satellites. As spacecraft need to run demanding workloads such as image processing, network functions, and advanced navigation, operators are increasingly relying on commodity hardware. However, these chips are susceptible to potentially mission-ending radiation errors, which has incurred significant monetary losses.

We introduce Radshield, the first software-only radiation error mitigation system. The key idea behind Radshield is to treat the software running on the hardware as a white-box, allowing Radshield to detect 1.5× more SELs than thresholding. Radshield also protects against SEUs just as effectively as 3-MR, with up to 70% less runtime and energy. Radshield is currently deployed onboard two real-world spacecraft at both ends of the cost spectrum: a low-cost SmallSat in low-Earth orbit; and a flagship science mission on the Martian surface. This is but the first step in providing software-based fault tolerance in space; indeed many challenges remain. We hope that by enabling commodity hardware to operate safely and reliably in space, we can lower the barrier of entry for spacecraft operations and democratize space exploration. In this spirit, we will open source Radshield's code and experiments.

Acknowledgments

This work was supported by the Department of Defense (DoD) through the National Defense Science & Engineering Graduate (NDSEG) Fellowship Program. The authors' research was also supported by the NSF (CNS-2143868). A portion of this research was carried out at the Jet Propulsion Laboratory, California Institute of Technology, under a contract with the National Aeronautics and Space Administration (80NM0018D0004). The authors thank the anonymous reviewers and our shepherd, David Cock, for their helpful feedback. The authors would also like to thank Jeremy Nash and Steven Guertin of the Jet Propulsion Laboratory for their input on this work.

References

[1] H. Jones. "The recent large reduction in space launch cost". In: 48th International Conference on Environmental Systems. 2018.

[2] A. C. Boley and M. Byers. "Satellite mega-constellations create risks in Low Earth Orbit, the atmosphere and on Earth". In: *Scientific Reports* 11.1 (2021), pp. 1–8.

[3] *Starlink*. 2023. URL: https://www.starlink.com/.

[4] Z. Qu, G. Zhang, H. Cao, and J. Xie. "LEO satellite constellation for Internet of Things". In: *IEEE Access* 5 (2017), pp. 18391–18401.

[5] J. Gedmark and S. Smith. "The Evolution of the Satellite Economy". In: a16z Podcast (Sept. 2023). URL: https://a16z.com/podcast/the-evolution-of-the-satellite-economy.

[6] *Planet*. 2023. URL: https://www.planet.com/.

[7] *Spire*. 2023. URL: https://spire.com/.

[8] *Filecoin Foundation and Lockheed Martin Bring Decentralized Storage to Space*. May 2022. URL: https://filecoinfoundation.medium.com/filecoin-foundation-and-lockheed-martin-bring-decentralized-storage-to-space-db9a15e66264.

[9] Y. Michalevsky and Y. Winetraub. "WaC: SpaceTEE-Secure and Tamper-Proof Computing in Space using CubeSats". In: *Proceedings of the 2017 Workshop on Attacks and Solutions in Hardware Security*. 2017, pp. 27–32.

[10] A. Fikes, M. Gal-Karziri, E. Gdoutos, M. Kelzenberg, E. Warmann, R. Madonna, H. Atwater, A. Hajimiri, and S. Pellegrino. "The Caltech space solar power project: Design, progress, and future direction". In: *Proceedings of the IEEE WiSEE Space Solar Power Workshop*. 2022.

[11] F. Michel, M. Trevisan, D. Giordano, and O. Bonaventure. "A first look at starlink performance". In: *Proceedings of the 22nd ACM Internet Measurement Conference*. 2022, pp. 130–136.

[12] J. Bao, B. Zhao, W. Yu, Z. Feng, C. Wu, and Z. Gong. "OpenSAN: A software-defined satellite network architecture". In: *ACM SIGCOMM Computer Communication Review* 44.4 (2014), pp. 347–348.

[13] D. Bhattacherjee, W. Aqeel, I. N. Bozkurt, A. Aguirre, B. Chandrasekaran, P. B. Godfrey, G. Laughlin, B. Maggs, and A. Singla.

"Gearing up for the 21st century space race". In: *Proceedings of the 17th ACM Workshop on Hot Topics in Networks*. 2018, pp. 113–119.

[14] S. Kassing, D. Bhattacherjee, A. B. Águas, J. E. Saethre, and A. Singla. "Exploring the "Internet from space" with Hypatia". In: *Proceedings of the ACM Internet Measurement conference*. 2020, pp. 214–229.

[15] A. Singla. "SatNetLab: a call to arms for the next global Internet testbed". In: *SIGCOMM Comput. Commun. Rev.* 51.2 (May 2021), pp. 28–30. ISSN: 0146-4833. DOI: 10.1145/3464994.3465000. URL: https://doi.org/10.1145/3464994.3465000.

[16] D. Perdices, G. Perna, M. Trevisan, D. Giordano, and M. Mellia. "When satellite is all you have: watching the internet from 550 ms". In: *Proceedings of the 22nd ACM Internet Measurement Conference*. 2022, pp. 137–150.

[17] M. M. Kassem, A. Raman, D. Perino, and N. Sastry. "A browser-side view of starlink connectivity". In: *Proceedings of the 22nd ACM Internet Measurement Conference*. 2022, pp. 151–158.

[18] J. R. Wertz, R. C. Conger, M. Rufer, N. Sarzi-Amadé, and R. E. Van Allen. "Methods for Achieving Dramatic Reductions in Space Mission Cost". In: *Reinventing Space Conference*. 2011, pp. 2–6.

[19] W. A. Powell. "High-performance spaceflight computing (HPSC) project overview". In: *Radiation Hardened Electronics Technology Conference (RHET) 2018*. GSFC-E-DAA-TN62651. 2018.

[20] J. Gibney. "AMD Ryzen™ 6000 Series for Mobile: Technology Overview". In: *2022 IEEE Hot Chips 34 Symposium (HCS)*. IEEE Computer Society. 2022, pp. 1–24.

[21] B. Denby, K. Chintalapudi, R. Chandra, B. Lucia, and S. Noghabi. "Kodan: Addressing the computational bottleneck in space". In: *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*. 2023, pp. 392–403.

[22] N. Bleier, M. H. Mubarik, G. R. Swenson, and R. Kumar. "Space Microdatacenters". In: *Proceedings of the 56th Annual IEEE/ACM International Symposium on Microarchitecture*. 2023, pp. 900–915.

[23] D. Bhattacherjee, S. Kassing, M. Licciardello, and A. Singla. "In-orbit computing: An outlandish thought experiment?" In: *Proceedings of the 19th ACM Workshop on Hot Topics in Networks*. 2020, pp. 197–204.

[24] L. Burcin. "RAD750 experience: The challenge of SEE hardening a high performance commercial processor". In: *Microelectronics Reliability & Qualification Workshop (MRQW)*. 2002.

[25] S. M. Guertin. "Radiation effects on ARM devices". In: *NEPP Electronics Technology Workshop*. National Aeronautics and Space Agency. Jet Propulsion Laboratory, 2019.

[26] H. Leppinen. "Current use of Linux in spacecraft flight software". In: *IEEE Aerospace and Electronic Systems Magazine* 32.10 (2017), pp. 4–13.

[27] B. Balaram, T. Canham, C. Duncan, H. F. Grip, W. Johnson, J. Maki, A. Quon, R. Stern, and D. Zhu. "Mars helicopter technology demonstrator". In: *2018 AIAA Atmospheric Flight Mechanics Conference*. 2018, p. 0023.

[28] P. Layton, D. Czajkowski, J. Marshall, H. Anthony, and R. Boss. "Single event latchup protection of integrated circuits". In: *RADECS 97. Fourth European Conference on Radiation and its Effects on Components and Systems (Cat. No. 97TH8294)*. IEEE. 1997, pp. 327–331.

[29] J. S. Chang, W. Shu, and J. Jiang. *Electronic circuit for single-event latch-up detection and protection*. US Patent 10,566,780. Feb. 2020.

[30] A. Dorise, C. Alonso, A. Subias, L. Travé-Massuyès, L. Baczkowski, and F. Vacher. "Machine learning as an alternative to thresholding for space radiation high current event detection". In: *2021 21th European Conference on Radiation and Its Effects on Components and Systems (RADECS)*. IEEE. 2021, pp. 1–7.

[31] Y. Shen, G. Heiser, and K. Elphinstone. "Fault tolerance through redundant execution on cots multicores: Exploring trade-offs". In: *2019 49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. IEEE. 2019, pp. 188–200.

[32] W. Li and M. Hudson. "Earth's Van Allen Radiation Belts: From Discovery to the Van Allen Probes Era". In: *Journal of Geophysical Research: Space Physics* 124.11 (Nov. 2019), pp. 8319–8351. ISSN: 2169-9402. DOI: 10.1029/2018ja025940. URL: http://dx.doi.org/10.1029/2018JA025940.

[33] K. L. Bedingfield and R. D. Leach. *Spacecraft system failures and anomalies attributed to the natural space environment*. Vol. 1390. National Aeronautics and Space Administration, Marshall Space Flight Center, 1996.

[34] S. M. Guertin. *ARM Radiation Testing & Collaborations*. June 2020. URL: https://trs.jpl.nasa.gov/handle/2014/52948 (visited on 12/31/2022).

[35] A. G. Schmidt, M. French, and T. Flatley. "Radiation hardening by software techniques on FPGAs: Flight experiment evaluation and results". In: *2017 IEEE Aerospace Conference*. IEEE. 2017, pp. 1–8.

[36] H. Wang, S. Myint, V. Verma, Y. Winetraub, J. Yang, and A. Cidon. "Mars Attacks! Software Protection Against Space Radiation". In: *Proceedings of the 22nd ACM Workshop on Hot Topics in Networks*. 2023, pp. 245–253.

[37] W. Kolasinski, J. Blake, J. Anthony, W. Price, and E. Smith. "Simulation of cosmic-ray induced soft errors and latchup in integrated-circuit computer memories". In: *IEEE Transactions on Nuclear Science* 26.6 (1979), pp. 5087–5091.

[38] L. Adams, E. Daly, R. Harboe-Sorensen, R. Nickson, J. Haines, W. Schafer, M. Conrad, H. Griech, J. Merkel, T. Schwall, et al. "A verified proton induced latch-up in space (CMOS SRAM)". In: *IEEE Transactions on Nuclear Science* 39.6 (1992), pp. 1804–1808.

[39] B. Johlander, R. Harboe-Sorensen, G. Olsson, and L. Bylander. "Ground verification of in-orbit anomalies in the double probe electric field experiment on Freja". In: *IEEE Transactions on Nuclear Science* 43.6 (1996), pp. 2767–2771.

[40] F. Yao, A. S. Rakin, and D. Fan. "DeepHammer: Depleting the intelligence of deep neural networks through targeted chain of bit flips". In: *29th USENIX Security Symposium (USENIX Security 20)*. 2020, pp. 1463–1480.

[41] C. Roscian, F. Praden, J.-M. Dutertre, J. Fournier, and A. Tria. "Security characterisation of a hardened AES cryptosystem using a laser". In: *Technical Sciences/University of Warmia and Mazury in Olsztyn* 15 (1 (2012), pp. 139–154.

[42] J. A. Pellish. "Radiation 101: Effects on Hardware and Robotic Systems". In: (2015).

[43] D. M. Hassler, C. Zeitlin, R. Wimmer-Schweingruber, S. Böttcher, C. Martin, J. Andrews, E. Böhm, D. Brinza, M. Bullock, S. Burmeister, et al. "The Radiation Assessment Detector (RAD) Investigation". In: *Space Science Reviews* 170.1 (2012), pp. 503–558.

[44] J. Tausch, D. Sleeter, D. Radaelli, and H. Puchner. "Neutron Induced Micro SEL Events in COTS SRAM Devices". In: *2007 IEEE Radiation Effects Data Workshop*. 2007, pp. 185–188. DOI: 10.1109/REDW.2007.4342562.

[45] N. Pieper, Y. Xiong, A. Feeley, D. Walker, R. Fung, S.-J. Wen, D. Ball, and B. Bhuva. "Micro-Latchup Location and Temperature Characterization in a 7-nm Bulk FinFET Technology". In: *2021 21th European Conference on Radiation and Its Effects on Components and Systems (RADECS)*. IEEE. 2021, pp. 1–7.

[46] Y. He, J. Zhao, W. Shu, K. S. Chong, J. Chang, et al. "Demonstration of ZES'LDAP (Latchup Detection and Protection) enabling a commercial-off-the-shelf FPGA for space applications". In: *2022 22nd European Conference on Radiation and Its Effects on Components and Systems (RADECS)*. IEEE. 2022, pp. 1–4.

[47] J. Zhao, K.-S. Chong, W. Shu, M. Cho, and J. S. Chang. "Design and Implementation of a High-Speed Low-Power K-Nearest-Neighbors-based Algorithm for Detecting Micro-Single-Event-Latchups". In: *IEEE Transactions on Nuclear Science* (2024).

[48] E. Normand. "Single-event effects in avionics". In: *IEEE Transactions on nuclear science* 43.2 (1996), pp. 461–474.

[49] N. J. Wang, J. Quek, T. M. Rafacz, and S. J. Patel. "Characterizing the effects of transient faults on a high-performance processor pipeline". In: *International Conference on Dependable Systems and Networks, 2004.* IEEE Computer Society. 2004, pp. 61–61.

[50] B. D. Sierawski, M. H. Mendenhall, R. A. Weller, R. A. Reed, J. H. Adams, J. W. Watts, and A. F. Barghouty. "CRÈME-MC: A physics-based single event effects tool". In: *IEEE Nuclear Science Symposuim & Medical Imaging Conference.* IEEE. 2010, pp. 1258–1261.

[51] A. Lyons and G. Heiser. "Mixed-criticality support in a high-assurance, general-purpose microkernel". In: *Workshop on Mixed Criticality Systems.* 2014, pp. 9–14.

[52] M. Rebaudengo, M. S. Reorda, M. Violante, and M. Torchiano. "A source-to-source compiler for generating dependable software". In: *Proceedings First IEEE International Workshop on Source Code Analysis and Manipulation.* IEEE. 2001, pp. 33–42.

[53] N. Oh, P. P. Shirvani, and E. J. McCluskey. "Error detection by duplicated instructions in super-scalar processors". In: *IEEE Transactions on Reliability* 51.1 (2002), pp. 63–75.

[54] C. Borchert, H. Schirmeier, and O. Spinczyk. "Compiler-Implemented Differential Checksums: Effective Detection and Correction of Transient and Permanent Memory Errors". In: *2023 53rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN).* IEEE. 2023, pp. 81–94.

[55] M. Salloum, J. R. Mayo, and R. C. Armstrong. "In-situ mitigation of silent data corruption in PDE solvers". In: *Proceedings of the ACM Workshop on Fault-Tolerance for HPC at Extreme Scale.* 2016, pp. 43–48.

[56] D. Fiala, F. Mueller, and K. B. Ferreira. "Flipsphere: A software-based DRAM error detection and correction library for HPC". In: *2016 IEEE/ACM 20th International Symposium on Distributed Simulation and Real Time Applications (DS-RT).* IEEE. 2016, pp. 19–28.

[57] M. Turmon, R. Granat, and D. Katz. "Software-implemented fault detection for high-performance space applications". In: *Proceeding International Conference on Dependable Systems and Networks. DSN 2000.* IEEE. 2000, pp. 107–116.

[58] N. Burow, S. A. Carr, J. Nash, P. Larsen, M. Franz, S. Brunthaler, and M. Payer. "Control-flow integrity: Precision, security, and performance". In: *ACM Computing Surveys (CSUR)* 50.1 (2017), pp. 1–33.

[59] J. Kosaian and K. Rashmi. "Arithmetic-intensity-guided fault tolerance for neural network inference on GPUs". In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis.* 2021, pp. 1–15.

[60] W. Zheng, B. Xu, J. Gu, and H. Chen. "SAVE: Software-Implemented Fault Tolerance for Model Inference against GPU Memory Bit Flips". In: *2025 USENIX Annual Technical Conference.* USENIX. 2025.

[61] D. Nicholaeff, N. Davis, D. Trujillo, and R. Robey. "Cell-based adaptive mesh refinement implemented with general purpose graphics processing units". In: *Tech. Rep. LA-UR-11-07127* (2012).

[62] M. Donald. "How An Ionizing Particle From Outer Space Helped A Mario Speedrunner Save Time". In: (Sept. 2020). URL: https://www.thegamer.com/how-ionizing-particle-outer-space-helped-super-mario-64-speedrunner-save-time/.

[63] S. Wang, G. Zhang, J. Wei, Y. Wang, J. Wu, and Q. Luo. "Understanding Silent Data Corruptions in a Large Production CPU Population". In: *Proceedings of the 29th Symposium on Operating Systems Principles.* 2023, pp. 216–230.

[64] P. H. Hochschild, P. Turner, J. C. Mogul, R. Govindaraju, P. Ranganathan, D. E. Culler, and A. Vahdat. "Cores that don't count". In: *Proceedings of the Workshop on Hot Topics in Operating Systems.* 2021, pp. 9–16.

[65] A. A. Hwang, I. A. Stefanovici, and B. Schroeder. "Cosmic rays don't strike twice: Understanding the nature of DRAM errors and the implications for system design". In: *ACM SIGPLAN Notices* 47.4 (2012), pp. 111–122.

[66] G. S. Veronese, M. Correia, A. N. Bessani, L. C. Lung, and P. Verissimo. "Efficient byzantine fault-tolerance". In: *IEEE Transactions on Computers* 62.1 (2011), pp. 16–30.

[67] A. Aponte-Moreno, F. Restrepo-Calle, and C. Pedraza. "A Low-Overhead Radiation Hardening Approach using Approximate Computing and Selective Fault Tolerance Techniques at the Software Level". In: *2019 19th European Conference on Radiation and Its Effects on Components and Systems (RADECS).* IEEE. 2019, pp. 1–4.

[68] V. Verma, J. Nash, L. Saldyt, Q. Dwight, H. Wang, S. Myint, J. Biesiadecki, M. Maimone, A. Tumbar, A. Ansar, G. Kubiak, and R. Hogg. "Enabling Long & Precise Drives for The Perseverance Mars Rover via Onboard Global Localization". In: *IEEE Aerospace Conference.* 2024.

[69] V. Verma, F. Hartman, A. Rankin, M. Maimone, T. Del Sesto, O. Toupet, E. Graser, S. Myint, K. Davis, D. Klein, et al. "First 210 solar days of Mars 2020 Perseverance Robotic Operations-Mobility, Robotic Arm, Sampling, and Helicopter". In: *2022 IEEE Aerospace Conference (AERO).* IEEE. 2022, pp. 1–20.

[70] B. Denby and B. Lucia. "Orbital edge computing: Nanosatellite constellations as a new class of computer system". In: *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems.* 2020, pp. 939–954.

[71] P. A. Ilott. "Communications with Mars: A Brief and Informal HIstory". In: *Space-Terrestrial Internetworking Workshop.* IEEE. 2021.

[72] D. McComas, J. Wilmot, and A. Cudmore. "The core flight system (cFS) community: Providing low cost solutions for small spacecraft". In: *Annual AIAA/USU Conference on Small Satellites.* GSFC-E-DAA-TN33786. 2016.

[73] J. J. Biesiadecki and M. W. Maimone. "The Mars exploration rover surface mobility flight software driving ambition". In: *2006 IEEE Aerospace Conference.* IEEE. 2006, 15–pp.

[74] B. Yost, S. Weston, G. Benavides, F. Krage, J. Hines, S. Mauro, S. Etchey, K. O'Neill, and B. Braun. "State-of-the-art small spacecraft technology". In: (2021).

[75] R. Bocchino, T. Canham, G. Watney, L. Reder, and J. Levison. "F Prime: an open-source framework for small-scale flight software systems". In: (2018).

[76] J. Nash, Q. Dwight, L. Saldyt, H. Wang, S. Myint, A. Ansar, and V. Verma. "Censible: A Robust and Practical Global Localization Framework for Planetary Surface Missions". In: *IEEE International Conference on Robotics and Automation* (2024).

[77] J. Maki, D. Gruel, C. McKinney, M. Ravine, M. Morales, D. Lee, R. Willson, D. Copley-Woods, M. Valvo, T. Goodsall, et al. "The Mars 2020 Engineering Cameras and microphone on the Perseverance rover: A next-generation imaging system for Mars exploration". In: *Space Science Reviews* 216 (2020), pp. 1–48.

[78] J. Lowe-Power, A. M. Ahmad, A. Akram, M. Alian, R. Amslinger, M. Andreozzi, A. Armejach, N. Asmussen, S. Bharadwaj, G. Black, G. Bloom, B. R. Bruce, D. R. Carvalho, J. Castrillón, L. Chen, N. Derumigny, S. Diestelhorst, W. Elsasser, M. Fariborz, A. F. Farahani, P. Fotouhi, R. Gambord, J. Gandhi, D. Gope, T. Grass, B. Hanindhito, A. Hansson, S. Haria, A. Harris, T. Hayes, A. Herrera, M. Horsnell, S. A. R. Jafri, R. Jagtap, H. Jang, R. Jeyapaul, T. M. Jones, M. Jung, S. Kannoth, H. Khaleghzadeh, Y. Kodama, T. Krishna, T. Marinelli, C. Menard, A. Mondelli, T. Mück, O. Naji, K. Nathella, H. Nguyen, N. Nikoleris, L. E. Olson, M. S. Orr, B. Pham, P. Prieto, T. Reddy, A. Roelke, M. Samani, A. Sandberg, J. Setoain, B. Shingarov, M. D. Sinclair, T. Ta, R. Thakur, G. Travaglini, M. Upton, N. Vaish, I. Vougioukas, Z. Wang, N. Wehn, C. Weis, D. A. Wood, H. Yoon, and É. F. Zulian. "The gem5 Simulator: Version 20.0+". In: *CoRR* abs/2007.03152 (2020). arXiv: 2007.03152. URL: https://arxiv.org/abs/2007.03152.