Towards Certifiable Software-Implemented Hardware Fault Tolerance

Federico Reghenzani
Politecnico di Milano
federico.reghenzani@polimi.it

William Fornaciari
Politecnico di Milano
william.fornaciari@polimi.it

***Abstract*—Reliability metrics for hardware faults in safety-/mission-critical systems have been historically based solely on hardware failure rates, quantitatively ignoring any effect of the software. Software reliability is usually considered only in terms of bugs/defects, which is a quantity hard to estimate analytically. In this article, we explore the problem of quantifying the impact of software in reliability against Single Event Upsets, highlighting the limits of the current standards that restrict the use of Commercial-Off-The-Shelf components for critical scenarios. We show how to obtain valid software reliability metrics and how this methodology significantly improves reliability estimation compared to hardware-only estimation. The reliability gain is further improved when considering real-time metrics. This analysis is the first step towards a reconciliation between software and hardware reliability and enables the quantification of reliability introduced by Software-Implemented Hardware Fault Tolerance approaches.**

***Index Terms*—Software-Implemented Hardware Fault Tolerance, Real-time**

I. Introduction

ONE of the major obstacles on the introduction of Commercial Off-The-Shelf (COTS) into critical systems is the missing of hardware elements to provide fault tolerance against soft-faults. Single Event Upsets (SEUs) are especially problematic in aerospace applications, due to the reduced shielding effect of atmosphere and Earth's magnetic field at high altitudes that expose the system to radiation. Acceptable failure rates are usually achieved via shielding or via hardware techniques – e.g., by employing multiple redundant processors. However, such techniques are usually expensive in terms of financial cost and do not guarantee the same level of computing performance of COTS [1] [2].

A. Software Fault Tolerance

The resilience techniques to hardware faults can be implemented not only at the hardware-level but also at the software-level. Such techniques are under the umbrella term *Software-Implemented Hardware Fault Tolerance (SIHFT)*. The motivation behind SIHFT is to improve the satisfaction of the reliability requirements without the need for specialized hardware design. SIHFT techniques can perform *fault detection* and/or *fault recovery*. For fault detection, traditional examples include data and instruction duplication, plausibility checks of the output values, error detection codes, an external monitoring device, control flow monitoring, and watchdogs. Fault recovery approaches are further classified in *space redundancy* and *time redundancy*. The first category includes task replicas and standby tasks that provide redundancy at software level. The most common examples for the second group are re-execution (sometimes called *retry mechanism*), checkpoint/restart, recovery blocks, code correcting codes, and forward/backward error recovery.

Moving the problem of the hardware fault tolerance to the software layer brings many advantages, including the reduction of design and implementation costs, as well as more flexibility in the development process [1]. In addition, SIHFT enables the use of COTS platforms in radiation-hostile environments.

The use of SIHFT has also disadvantages:

1) **Increased computational workload**: implementing any SIHFT technique clearly increases the overhead of the system workload.
2) **Inability to cover all faults**: the problem of having *single points of failure* is exacerbated when we move from hardware to software because SIHFT techniques are able to detect most of the faults, but not all of them.
3) **Incompatibility with current standards**: most of all safety-critical standards used nowadays in industry for certification do not allow SIHFT techniques to replace hardware redundancy.

Problem 1) is linked with the real-time requirements: safety-critical systems often need to satisfy temporal constraints in addition to guaranteeing the logical correctness of the output. However, this issue is compensated by the use of more powerful COTS computing platforms. The problem 2) is the main issue that also contributes to the problem 3). Unfortunately, current standards do not split the concept of software failure due to a bug or a defect from the concept of software failure due to a hardware fault. Many of them allow the use of SIHFT only as an extra safety measure with only a qualitative analysis of the impact on failure rate. Due to this limitation, which is detailed in Section II, SIHFT has currently only a marginal role in the satisfaction of safety requirements. However, in this article, we show a precise methodology that can guarantee the safety of SIHFT approaches, with a certain, well-defined, probability of failure. Such methodology, described in Section III, allows the quantification of the increased reliability introduced by the use of SIHFT approaches, making their failure rate computable and exact.

The focus of this paper is limited to SIHFT approaches against soft-faults, in particular SEUs.

This work has received funding by the National Resilience and Recovery Plan (PNRR) through the National Center for HPC, Big Data and Quantum Computing.

B. Intersection with the timing problem

Most of all safety-critical systems are also *hard* real-time systems because need to perform their function within well-defined timing constraints. Hard real-time systems must meet these timing constraints all the times and even a violation of a single one is considered a failure of the system. Guaranteeing such constraints requires an extensive verification of both software and hardware, including the Worst-Case Execution Time (WCET) estimation and the scheduling analysis.

The overhead introduced by SIHFT makes the satisfaction of the timing constraints harder, creating a conflict between the reliability and timing goals. However, the availability of timing metrics provided by the scheduling analysis can be also exploited to improve the reliability analysis by quantifying the software failure rates, as explained in this article.

C. Related Works

SIHFT techniques are not a novel approach to increase fault resilience and their proposal dates back to 1978 with the seminal paper by Wensley et al. [3]. The book by Goloubeva et al. [1] describes the traditional SIHFT techniques. Several state-of-the-art works assessed the SIHFT capabilities in tolerating faults, e.g., Eghbal et al. [4] measured the number of faults in a PIC 16C5x microcontroller equipped with a SIHFT technique. Baroffio et al. [5] [6] presented a compiler-injected SIHFT mechanism for the FreeRTOS operating system and tested the resulting code on a real STM32 microcontroller. A similar approach has been proposed by Sharif et al. [7] for RISC-V processors. In 2015, Schirmeier et al. [8] presented FAIL*, a fault-injection framework that can also evaluate the fault tolerance improvement achieved by SIHFT techniques. The framework is able to identify critical software parts that need a dedicated fault-tolerance design.

Regarding real scenarios, the satellite ARGOS launched in 1999 used SIHFT as additional safety measure against SEUs. Lovellette et al. [9] observed that SIHFT was able to detect and correct 98.7% of the SEUs during the mission.

However, all the aforementioned works, even if measuring the ability of SIHFT mechanisms to detect and tolerate SEUs, lack of a clear and systematic methodology to compute the improvement in the system-level reliability metrics by considering both memory and timing properties of the software.

II. The Analysis of Current Standards

To provide an overview of the status of the current standards, we analyzed the following safety-critical standards from different fields: ISO-26262 [10] for automotive software, EN50128 [11] for railways software, DO-178B [12] for avionics, and the ECSS handbooks and standards [13] for space applications. We analyzed each standard to check whether they allow the software to have a well-defined probability of failure, if SIHFT mechanisms are mentioned, and which are their role.

A. ISO-26262 - Automotive

The ISO-26262 standard allows a software to be a *safety mechanism* [10, Part 1, §1.142, §1.41] (i.e., a fault tolerance mechanism) but the compliance with the failure requirements of the safety mechanism must be *"derived from the hardware architectural metrics"* [10, Part 5, §8.4.6], implicitly disallowing software metrics from the safety goal quantification standpoint. *"The quantitative analysis methods only address random hardware failures. These analysis methods are not applied to systematic failures."* [10, Part 9, §8.2] *"Due to the specific nature of software (e.g. no random faults due to wear out or ageing and lack of a mature probabilistic method), methods established (...) at the system or hardware level often cannot be transferred to software without modifications"* [10, Part 6, §E.3.3] Quantitative analyses are then possible only for hardware components [10, Part 5].

The use of SIHFT is recommended and the following mechanisms mentioned:

- Fault detection: range checks, plausibility checks, error detecting codes, external monitor, temporal monitor, diverse redundancy, access violation control [10, Part 6, §7.4.12]
- Fault recovery: recovery blocks, backward/forward recovery, re-execution, graceful degradation, replicas, diverse programming, error correcting codes [10, Part 6, §7.4.12]

B. EN 50128 - Railway

Probabilistic quantification for software faults are described in the *probabilistic testing* section [11, §D.41]. However, the use of probabilistic testing is not intended to be used to provide evidence of the failure rate safety goals, not even at low criticality levels [11, §Tab. A.5, Tab. A.7]. Event trees are described [11, §D.22], including the standard says that they *"can be used to compute the probability of the various consequences based on the probability and number of conditions in the sequence."* [11, §D.22]. However, how the probabilities of the single components are computed is not specified. Similarly to the other standards, the event trees are only recommended and intended to be used as an additional measure without a quantification of the fault rate.

Specific documents and tests regarding software/hardware integration should show *"that the software can handle hardware faults as required"* [11, §7.3.4.36]. *"Fault detection is (...) mainly to detect hardware faults"* [11, §D.26], including software fault detection mechanisms. Table A.3 [11] includes many of the previously described fault recovery mechanisms as possible techniques to be selected for implementation according to the safety integrity levels. They include re-execution, recovery blocks, diverse programming, forwards/backward recovery. However, while recommended as extra measures, none of them (with the exception of diverse programming) is mandatory to demonstrate the safety requirements.

C. DO-178B/C - Aviation

"This document does not provide guidance for software error rates." [12, §12.3.4]. However, it also leaves open the possibility to provide a rationale, agreed by the authority, to quantifies failure rates of software. However, it also makes explicit that *"software levels or software reliability rates*

based on software levels cannot be used by the system safety assessment process as can hardware failure rates." [12, §2.2.3]

The aviation standard does not provide specific guidance on SIHFT mechanisms, limiting to mentioning that *fault tolerance functions* can be implemented in software [12, §11.1.b].

D. ECSS-Q-* - Space (ESA)

Regarding the ECSS standards ecosystem, we must distinguish between handbooks (-HB-) and pure standards (-ST-). The former group includes only recommendations, while the second includes mandatory rules.

Handbooks discourage the use of software failure rates: *"the use of software reliability models to justify compliance with applicable reliability requirements is not advisable."* [13, Q-HB-80-03A §4.1.2] and the standard makes clear to consider them only qualitative: *"As it is not possible to quantitatively assess the software functions, only a qualitative assessment can be made as the dependability of software is influenced by the software development process."* [13, Q-ST-30C §6.4.1].

Interestingly, the handbook [13, Q-HB-60-02A] has an entire chapter on the SIHFT mechanisms, emphasizing the importance of SIFT in the context of COTS hardware. Various software redundancy approach are proposed, as well as system-level protections, such as watchdogs, error correcting codes, and other techniques.

E. Discussion

Despite the aforementioned standards disallow to quantify the software failure metrics, it should be noted that SIHFT is already encouraged by all of them as an extra mechanism to improve fault tolerance.

The non-quantifiability of the software failures metrics expressed by all the standards is due to the fact that software failures are considered as bugs or defects introduced during the development phase. Indeed, the current knowledge and software models do not allow to estimate a precise probability of failure due to a software bug or defect. This limitation, however, implicitly prevents the quantification of the reliability improvement provided by SIHFT, even if the source of the failure is not a bug/defect but a quantifiable physical quantity. In this article, we would like to change this *standard mindset*, showing that it is possible to quantify the improvement by analyzing software metrics and using SIHFT.

III. Failure Rates and Timing Information

The non-quantifiability of software fault rates mentioned in the standards refer to systematic faults, such as bugs or error during the software development process for which, indeed, there is no reliable methods to estimate the probability of a bug to be present or to cause an error. However, it is possible to mathematically derive the probability that a software failure is caused by a hardware fault. In a first rough approximation, we can consider that each hardware fault directly causes a software failure, thus their rate is equivalent. This is the implicit approach followed by standards, where each hardware fault (including SEU) is potentially considered to be a cause of a failure (if no hardware fault tolerance exists). This approach is, however, very pessimistic, due to spatial and temporal reasons: a task does not fully utilize all computing resources. For instance, not all the memory areas are used by tasks and each task is not running in all time instants.

A. Terminology and Assumptions

We use the following widely-accepted terminology [14]: a *fault* is a defect in the functioning of a hardware component, an *error* is a discrepancy between a computed value or condition and the theoretically correct value or condition, while a *failure* is the inability of the system to perform the required *function* according to the original requirements. A *fault* may or may not cause an *error*, and an *error* may or may not cause a *failure*.

The following assumptions are assumed by the subsequent analysis:

(A1) Hardware components are non-repairable
(A2) The hardware fault event probability is exponentially distributed
(A3) Hardware fault events are independent and identically distributed (i.i.d.)

In critical systems, components are considered non-repairable (A1) because in most of the situation is not possible to replace the failed components until the risky scenario terminates. This is, in any case, a pessimistic assumption and does not pose any safety risk. Regarding (A2) and (A3), an exponentially distributed probability means that the failure rate is constant, while the i.i.d. assumption can be split in two sub-hypotheses: the identically distributed, which is included in (A2), and the independence. Both (A2) and (A3) are generally true when considering SEUs, because the occurrence of each single event has no relation with others. These assumptions are also common in industrial applications [15] [16]. If a system is exposed to different conditions during its life, the worst-case one is assumed so that (A3) remains valid (for instance, the worst-case condition in a space mission is considered in ESA reliability analyses [13, E-ST-10-12C, §5.5.3.1]).

B. Task model

The system $(\Gamma, \mathcal{H})$ is composed of n software tasks $\Gamma = \{\tau_1, \tau_2, ..., \tau_n\}$ and m hardware resources $\mathcal{H} = \{H_1, H_2, ..., H_m\}$. Each task performs a prescribed function multiple times (periodically or sporadically) and each unit of computation that performs the function is called *job*. The task is characterized by the tuple $\tau_i = (C_i, T_i, D_i)$, where C_i is the WCET, T_i the period or minimum inter-arrival time, and D_i the relative deadline. Each resource can be a CPU, a memory, a disk, or any other hardware device, and has a given failure rate λ_i. For simplicity, we assume that each task performs one single *function* at system-level. We call Λ_i^* the target failure rate, usually expressed in probability of failure per hour.

The set $\Delta = \{\delta_1, \delta_2, ...\}$ represents the resource assignment relations, where each relation is: $\delta_i = \langle \tau_j, H_k, s_m, \epsilon_n \rangle$, where τ_j is the task and H_k is the resource, as previously defined. The values s_m and ϵ_n are, respectively, the *space share* and the *exposure time*. In particular:

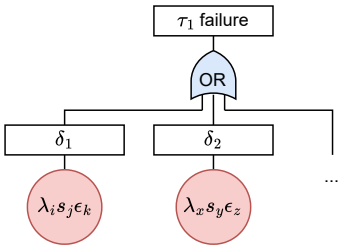

Fig. 1. Fault tree schema of a generic task.

- The space share $s_m \in [0;1]$ defines the usage, in percentage, of the resource H_k by the task τ_j.
- The exposure time $\epsilon_n \in [0;1]$ defines the rate of time that a task τ_j uses a resource H_k. The task is therefore subject to faults in that specific resource for the period of time specified by ϵ_n. How to precisely compute ϵ_n is discussed later in Section III-D.

The tuples in Δ are not necessarily unique, i.e., there can be multiple items with the same task and resource. The reason is that a task can use the resource for different share and portion of time; for instance, a task can use 0.5 of the memory while a job is running, and 0.1 of the memory from the finishing time of one job and the arrival of the next.

Example 1. *Let us consider a system having a single-memory H_1 of 32 kB and a single-processor H_2, running a single-task τ_1. The task runs for 10 time units and sleeps for 40 time units. During the idle time the task maintains a state of 1 kB and during the run-time it uses 16 kB of the memory. Thus:* $\Delta = \{\langle \tau_1, H_1, 1/32, 1\rangle, \langle \tau_1, H_1, 0.5, 0.2\rangle, \langle \tau_1, H_2, 1, 0.2\rangle\}$

C. Task base failure rate

In order to compute the task failure rate – which is the probability that a job fails to provide the correct output over a defined period of time (usually per-hour) – we proceed by building the fault tree depicted in Figure 1. Assuming no software fault tolerance method in place, the failure rate of the task is the combination in *OR* of the basic event failure rates, i.e., the failure rate of each resource λ_i multiplied by the space and time share. This trivial multiplication is valid thanks to the failure rate properties [17], and improves previous works which used direct probabilities and had to resort to more complex analyses [18]. The top-event failure rate of the fault tree of Figure 1, given the assumption of Section III-A, is calculated as the sum of the individual rates [19, Eq. (4.40)]:

$$\Lambda_i = \sum_{\forall \langle \tau_j, H_k, s_m, \epsilon_n \rangle \in \Delta} \lambda_k \cdot s_m \cdot \epsilon_n \tag{1}$$

The symbol Λ_i is the *task basic failure rate*, i.e., the failure rate of the task τ_i in absence of fault tolerant mechanisms. If $\Lambda_i \leq \Lambda_i^*$, then no fault tolerant mechanism in necessary because the task already satisfies the reliability requirement.

D. Exploiting real-time metrics to estimate ϵ_n

The knowledge of the real-time metrics allows us to calculate the value of the ϵ_n or at least upper-bound it. The exposure time ϵ_n depends on the resource acquisition and resource release time. Frequently, most of the resources are acquired at the beginning of the job and released at the end, first and foremost the memory used for local variables or stack in general. In such a case, the exposure time of a resource assignment $\langle \tau_j, H_k, s_m, \epsilon_n \rangle$ is:

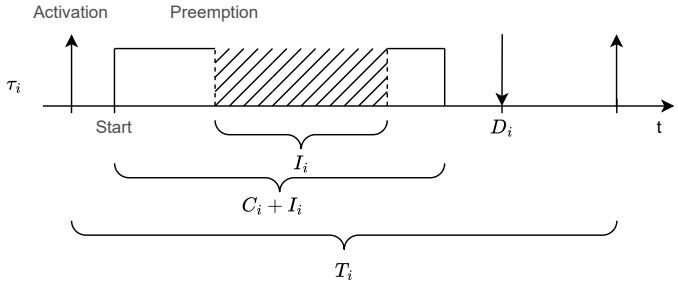

Fig. 2. Faults in many resources are effective only in the time interval in which the job is active, i.e., $C_j + I_j$ every T_i time units. Other jobs present in the system are not depicted in the figure.

$$\epsilon_n = \frac{C_j + I_j}{T_j} \tag{2}$$

where I_j is the worst-case interference time, i.e., the time that the task is preempted or suspended (e.g., to allow tasks with higher priority to run). This case, exemplified in Figure 2, is the simplest form of timing modeling of ϵ_n but highlights how OS-level decisions, such as the scheduling algorithm can impact the reliability metrics: a scheduling algorithm privileging the reduction of I_j would increase the reliability but it reduces the optimality from a real-time standpoint, creating a challenging trade-off to explore.

IV. A Methodology to Analyze a Complete SIHFT-based System

From the quantification of the basic failure rate described above, we show how it is possible to exploit more information to improve the analysis, i.e., introducing SIHFT, operating systems, and splitting memory elements. We focus on the analysis of the memory elements of a generic microcontroller.

A. System Components

In first approximation, we can categorize the memory of most of microcontrollers in the following resources:

- H_P: the memory containing the program code
- H_D: the memory containing the data
- H_G: the General Purpose Registers (GPRs) of the CPU, used by the tasks to perform the computation. In this definition, we include the accessory registers, such as the program counter, the stack pointer, the register file, etc.
- H_S: the Special Function Registers (SFRs) of the CPU, used to configure peripherals and other hardware settings.

Correspondingly, $\lambda_P, \lambda_D, \lambda_G, \lambda_S$ are the respective SEU failure rates of the memories. These failure rates are usually expressed per bit-hour, thus having the following measurement unit: $[1/(\text{bit}\cdot\text{h})]$. For this reason, we replace the ratio s_m with the memory usage:

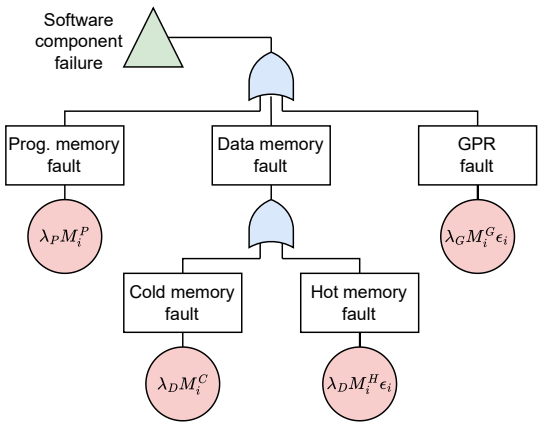

Fig. 3. Fault tree for the calculation of a generic software component.

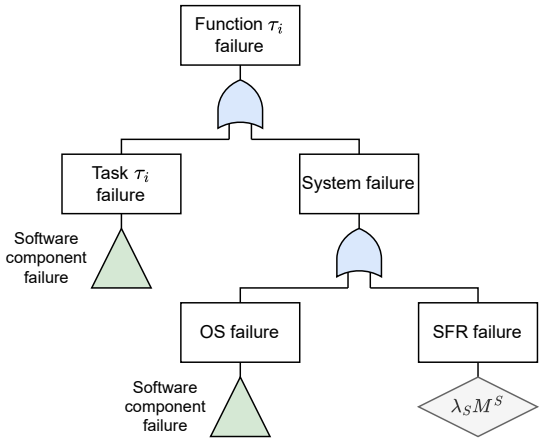

Fig. 4. The extension of the task failure rate to the function failure rate, including system software failures.

- M_i^P: number of bits used in the program memory
- M_i^G: number of bits used in the GPRs
- M_i^S: number of bits used in the SFRs
- $M_i^{D_C}$: number of bits used in the *cold* data memory
- $M_i^{D_H}$: number of bits used in the *hot* data memory

The *cold* data memory represents the data segment used by the tasks that persists across job executions. Instead, the *hot* data memory represents the data segment used by the tasks only within a job execution and released at the termination of the job (for instance, the local variables in the stack).

B. Software component failure

The sub-tree of Figure 3 represents the chain of faults that may cause a software component to fail. We use the term *software component* to identify any task and the operating system itself. The sub-tree is constructed as follows: a software component can fail if there is a fault in the program memory (↲), if there is a fault in any GPR (↳), or if there is a fault in the data memory (↓). The data memory branch is split in cold and hot cases. Consequently, the failure rate of the top event is computed by applying Eq. (1):

$$\Lambda_i = \lambda_P M_i^P + \lambda_D M_i^{D_C} + \epsilon_i \left(\lambda_G M_i^G + \lambda_D M_i^{D_H} \right) \quad (3)$$

C. Function failure

The *failure* is the termination of the ability of the system, and by extension of the software, to perform a *function* [14]. The safety requirements are therefore expressed on the function itself, rather than on the software component performing it. For this reason, the fault tree of Figure 4 has the function failure as top-event, which is the one that must be compared against the requirements. The function failure can be caused by the failure of the task performing that function (↲) or a failure in the system (↳). The system failure can be caused by a problem in the operating system or by an error inside the SFRs. The task failure is modeled with the fault tree of Figure 3, as well as the operating system, that can be considered like a task (in subsequent notation τ_{OS}) for the sake of the fault analysis. SFR failure is represented as an *undeveloped event* because it could potentially be further refined: not all SFRs are actually used or generate an error if a SEU occurs, and they are often protected by extra techniques. The top-event has, therefore, the following failure rate:

$$\hat{\Lambda}_i = \Lambda_i + \Lambda_{\text{OS}} + \lambda_S M_S \quad (4)$$

D. Function failure with SIHFT

When SIHFT is employed, the fault tree must be accordingly updated. The fault tree is, in general, different for each SIHFT mechanism and may require numerical solvers. We provide the analysis of a SIHFT approach with re-execution mechanism and data replication for the cold memory. We assume that the scheduling analysis allocated sufficient free CPU utilization to run the re-execution mechanism with all of its associated overhead. The new fault tree for a generic task is depicted in Figure 5. The task failure can be caused by a fault in the program memory (↲), at least two faults in the cold memory (↓), or the failure of all sub-tasks (↳). The sub-tasks represent the re-execution retries. Each sub-task can fail due to a GPR fault (↲) or an hot memory fault (↳). Moreover, all sub-tasks are considered failed if the detection fails to detect the presence of a fault (failure rate λ_{DET}). In such a case, an undetected error is present in the system and can cause a failure.

Differently from the previous sections, we cannot proceed analytically computing the failure rates by using the ones of each sub-block, due to the non-linearity of the voter and standby nodes. We then proceed by computing the resulting reliability function as a function of time t as follows:

$$R_i(t) = R_P(t) \cdot R_{D_C}(t) \cdot R_{ST}(t) \quad (5)$$

where the multiplication is the operation to apply in presence of an OR gate in the fault tree (the top-level OR gate of Figure 5), and:

- $R_P(t)$ is the reliability of the program memory (↲):

$$R_P(t) = e^{-\lambda_P \cdot M_P \cdot t} \quad (6)$$

- $R_{D_C}(t)$ is the reliability of the cold memory (↓) (defined later in Section IV-E).

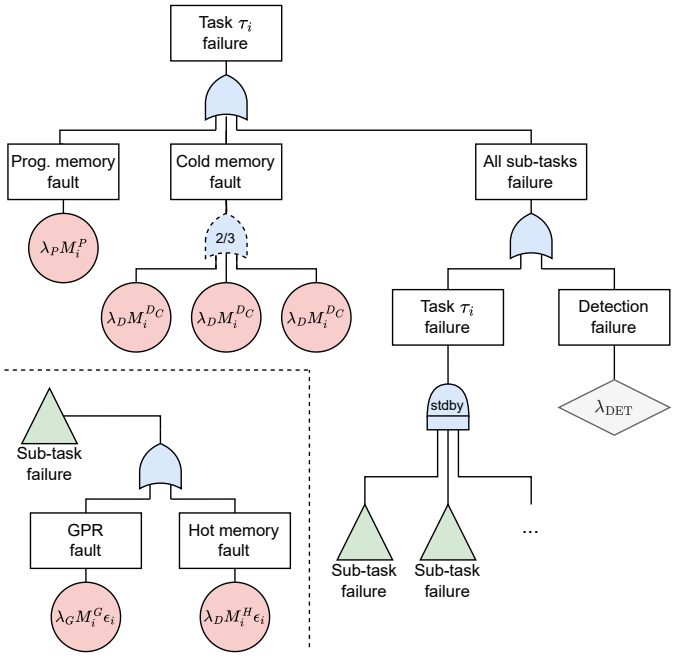

Fig. 5. A complete fault tree for a task τ_i including SIHFT mechanisms.

- $R_{ST}(t)$ is reliability of the sub tasks (↳):

$$R_{ST}(t) = \left[\sum_{i=0}^{\rho_i} \frac{(\lambda_{ST} t)^i \cdot e^{-\lambda_{ST} t}}{i!} \right] \cdot e^{-\lambda_{\text{DET}} t} \quad (7)$$

  where ρ_i is the allowed maximum number of re-executions for the task, and:

$$\lambda_{ST} = \epsilon_i \left(\lambda_G M_i^G + \lambda_D M_i^{D_H} \right) \quad (8)$$

The previous formulas have been derived by applying the standard reliability analysis and composition [19].

To verify the requirement, i.e. $\Lambda_i \leq \Lambda_i^*$, we need to integrate the reliability function to obtain the Mean Time To Failure (MTTF): MTTF $= \int_0^\infty R_i(t)$. Assuming a constant failure rate of fault tree components[1], the total failure rate of the τ_i function is then $\Lambda_i = {}^1/_{\text{MTTF}}$.

E. Computation of the k-out-of-n cold memory branch

This branch and the computation of R_{D_C} need a special attention, because of the following problem. Normally, a k-out-of-n gate has the following reliability formula:

$$R_{D_C}(t) = \sum_{i=k}^{n} \binom{n}{i} z^i (1-z)^{n-i} \quad (9)$$

where $k = 2$, $n = 3$ and $z = e^{-\lambda_{D_C} M_i^{D_C} t}$. However, this formula is very pessimistic for the software case: it considers the resource as a whole, while we can check (and correct) the single bits. An fault to be effective must hit two bits in the same address and the same position of two different memory areas, which has a much lower probability. Taking into account this consideration, we can optimize Equation (10) as follows:

$$R_{D_C}(t) = \left[\sum_{i=k}^{n} \binom{n}{i} z'^i (1-z')^{n-i} \right]^{M_i^{D_C}} \quad (10)$$

where $z' = e^{-\lambda_{D_C} t}$

[1]Due to the standby operator, the failure rate is not constant. However, the failed component is a *job* which has to be considered immediately repaired, thus not creating any sort of dependencies between task re-executions.

TABLE I
TASK PARAMETERS OF THE EXPERIMENTAL SIMULATION.

Task	$T_i = D_i$ [ms]	C_i [ms]	$M_i^{D_C}$ [B]	$M_i^{D_H}$ [B]
τ_1	500	73.0	1024	202
τ_2	100	0.3	4	184
τ_3	5000	610.9	5124	240
τ_4	5	0.7	201	201

V. NUMERICAL ANALYSIS OF A SPACECRAFT USE-CASE

To verify the reliability improvement, we exploit the previous analytical formulation to derive the function failure rate of a realistic use-case.

A. Reference - Hardware

As a reference COTS computing platform for this experimental evaluation, we selected the microcontroller Microchip PIC24FJ256GA110 for the following reasons: 1) It has been used in several CubeSAT missions; 2) A study on the SEU rate of this specific device is available [20]; 3) It is simple enough to be carefully analyzed. In order to match the available data on reliability, the device is assumed to be clocked at 8 MHz. The PIC microcontroller has a Harvard architecture with a total of 16 384 B of data memory, 261 876 B of program memory and 305 bits of general purpose registers (GPRs), 4 148 bits of Special Function Registers (SFRs). The program memory is stored in a flash, while the data memory is implemented as a SRAM. Flash memories are, in general, very resilient to radiations, indeed, Guertin et al. [20] did not identify any SEU in the flash memory during their experiments and the cross-section must therefore be several order of magnitude lower than the SRAM. For this reason, we voluntarily omitted the analysis of the flash program memory, focusing on the critical component, which is the SRAM.

Thanks to the use of the SPENVIS tool by the ESA, we calculated the exact rate of SEUs per-bit in this microcontroller for a small satellites in a Polar-LEO trajectory in unfavorable conditions (South Atlantic Anomaly trajectory and maximum solar activity): $\lambda_{\text{bit}} = 3.0264 \cdot 10^{-12} / (\text{bit} \cdot s)$.

B. Reference - Software

We selected four benchmarks representative of typical workload on previous spacecraft computers: τ_1 (CRC-32), representing a very common hash function used by many space communication tasks; τ_2 (LatNAV) representing the control task of the satellite; τ_3 (Edge Detector) runs an image processing algorithm, which represents a star tracking algorithm for navigation; τ_4 (binary search) representing the access

TABLE II
RESULTS OF THE EXPERIMENTAL SIMULATION.

A: BASELINE, I.E., STANDARD COARSE GRAIN EVALUATION
B: WITH SOFTWARE INFORMATION
C: WITH SOFTWARE AND TIMING INFORMATION (PREEMPTIVE)
D: WITH SOFTWARE AND TIMING INFORMATION (NON-PREEMPTIVE)
E: D + SIHFT
F: E + UNPROTECTED OPERATING SYSTEM AND SFRS

Task	A	B	C	D	E	F
τ_1	1.5e−3	1.1e−4	9.4e−5	9.2e−5	2.2e−6	5.7e−5
τ_2		1.6e−5	2.6e−6	4.1e−7	1.1e−7	5.5e−5
τ_3		4.9e−4	4.8e−4	4.7e−4	4.5e−6	5.9e−5
τ_4		3.5e−5	2.0e−5	2.0e−5	1.3e−6	5.6e−5

to one of the many message queues used to perform inter-task communication. These benchmarks are simple but are a good representative considering the target microcontroller. The selected real-time operating system is FreeRTOS.

The source code has been compiled with the official Microchip XC16 compiler version 2.00 (based on `gcc` version 4.5.1), obtaining memory usage metrics. The task WCETs are reported in Table I approximated at the 100th micro-seconds with a round-up policy, so that the numbers are pessimistic but safe. The total utilization of the task set, without considering SIHFT, is $U = 0.412$.

C. Results.

The results are presented in Table II. The baseline, given by considering only hardware metrics, has a very high failure rate, that cannot even satisfy the lowest criticality level for DO-178C (DAL D is $\Lambda_i^* = 10^{-3}$/hour). The first improvement of the analysis (A→B) was given by the use of software information (Figure 3 with $\epsilon = 0$) and was significant: the tasks reduced their failure rate of 1-2 order of magnitudes. Then, we considered a Rate Monotonic preemptive scheduler (Figure 3 with computed ϵ) and the failure rates reduced (B→C) of another order of magnitude. The use of a non-preemptive scheduling algorithm (implying $I_i = 0$), further reduced (C→D) the failure rate, with a strong impact on τ_2. Then, the introduction of SIHFT via re-execution pattern (Figure 5) made all the tasks to have a failure rate lower than 10^{-5}/hour. Such failure rates satisfy the levels DAL D and C of DO-178C. Finally, we added the OS and SFRs (Figure 4), and the results are shown in column F. As expected, they clearly brought up the failure rate because we considered unprotected OS and SFRs, for which a SIHFT analysis is left for future works.

VI. CONCLUSIONS

We analyzed the current standards and identified that they do not consider the possibility of integrating the software properties in the failure analysis because they consider a software failure only as a bug or defect. We then presented an analytical analysis to show how to mathematically calculate the failure rate improvement given by the software. The numerical results showed that, with software information, timing information, and the introduction of SIHFT, it is possible to reduce the failure rate against SEUs by several orders of magnitude. Timing information plays an important role and different scheduling policies lead to different reliability results. The numerical simulation also highlighted that the introduction of SIHFT to the tasks must be associated with the protection of the operating system and SFRs to avoid jeopardizing the SIHFT improvements.

REFERENCES

[1] O. Goloubeva, M. Rebaudengo, M. S. Reorda, and M. Violante, *Software-Implemented Hardware Fault Tolerance*, Springer, Ed. Morgan Kaufmann, 2006.

[2] F. Reghenzani, "Enabling software technologies for critical cots-based spacecraft systems," in *Proceedings of the 20th ACM International Conference on Computing Frontiers*, ser. CF '23. New York, NY, USA: Association for Computing Machinery, 2023, p. 236–242.

[3] J. Wensley, L. Lamport, J. Goldberg, M. Green, K. Levitt, P. Melliar-Smith, R. Shostak, and C. Weinstock, "Sift: Design and analysis of a fault-tolerant computer for aircraft control," *Proceedings of the IEEE*, vol. 66, no. 10, pp. 1240–1255, 1978.

[4] A. Eghbal, H. R. Zarandi, and P. M. Yaghini, "Fault tolerance assessment of pic microcontroller based on fault injection," in *2009 10th Latin American Test Workshop*, 2009, pp. 1–6.

[5] D. Baroffio and F. Reghenzani, "Compiler-injected sihft for embedded operating systems," in *20th ACM International Conference on Computing Frontiers (CF'23)*. ACM, 2009, pp. 1–7.

[6] D. Baroffio, F. Reghenzani, and W. Fornaciari, "Enhanced compiler technology for software-based hardware fault detection," *ACM Trans. Des. Autom. Electron. Syst.*, apr 2024.

[7] U. Sharif, D. Mueller-Gritschneder, and U. Schlichtmann, "Compas: Compiler-assisted software-implemented hardware fault tolerance for risc-v," in *2022 11th Mediterranean Conference on Embedded Computing (MECO)*, 2022, pp. 1–4.

[8] H. Schirmeier, M. Hoffmann, C. Dietrich, M. Lenz, D. Lohmann, and O. Spinczyk, "Fail*: An open and versatile fault-injection framework for the assessment of software-implemented hardware fault tolerance," in *European Dependable Computing Conference*, 2015, pp. 245–255.

[9] M. Lovellette, K. Wood, D. Wood, J. Beall, P. Shirvani, N. Oh, and E. McCluskey, "Strategies for fault-tolerant, space-based computing: Lessons learned from the argos testbed," in *Proceedings, IEEE Aerospace Conference*, vol. 5, 2002, pp. 5–5.

[10] International Standard Organization, "Road vehicles – functional safety," ISO, Standard ISO-26262, 2018.

[11] European Committee for Electrotechnical Standardization, "Railway applications - communication, signalling and processing systems - software for railway control and protection systems," jun 2011.

[12] RTCA/EUROCAE, "DO-178B - Software Considerations in Airborne Systems and Equipment Certification," Standard, dec 1992.

[13] ESA. ECSS standards and handbooks. https://ecss.nl/.

[14] T. Arbel-Newman, J. Athavale, R. Bhattacharya, R. Bongiwar, W.-R. Chen, M. Diaz, L. Di Mauro, C. Di Napoli, D. Galpin, S. Kasrung, V. Kleeberger, S. Lorenzini, R. Mariani, A. Patel, H. Ptackova, F. Reghenzani, R. Schaaf, M. Turner, and B. Vignasse, "The functional safety terminology landscape," pp. 1–35, 2023.

[15] Jedec Solid State Technology Association, "Methods for calculating failure rates in units of fits," JEDEC, Standard, 2001.

[16] Jedec Solid State Technology Association , "Measurement and reporting of alpha particle and terrestrial cosmic ray-induced soft errors in semiconductor devices," JEDEC, Standard, 2006.

[17] R. L. Fleurence and C. S. Hollenbeak, "Rates and probabilities in economic modelling," *PharmacoEconomics*, vol. 25, no. 1, 2007.

[18] F. Reghenzani, Z. Guo, L. Santinelli, and W. Fornaciari, "A mixed-criticality approach to fault tolerance: Integrating schedulability and failure requirements," in *2022 IEEE 28th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2022, pp. 1–30.

[19] B. Dhillon and C. Singh, *Engineering Reliability: New Techniques and Applications*, ser. A Wiley-Interscience publication. Wiley, 1981.

[20] S. M. Guertin, M. Amrbar, and S. Vartanian, "Radiation test results for common cubesat microcontrollers and microprocessors," in *2015 IEEE Radiation Effects Data Workshop (REDW)*, 2015, pp. 1–9.