POLITECNICO DI TORINO
Repository ISTITUZIONALE

Evaluating Reliability against SEE of Embedded Systems: A Comparison of RTOS and Bare-metal Approaches

Original
Evaluating Reliability against SEE of Embedded Systems: A Comparison of RTOS and Bare-metal Approaches / DE SIO, C., Azimi, S., Sterpone, L.. - In: MICROELECTRONICS RELIABILITY. - ISSN 0026-2714. - 150:(2023). [10.1016/j.microrel.2023.115124]

Availability:
This version is available at: 11583/2981728 since: 2023-09-06T12:28:58Z

Publisher:
Elsevier

Published
DOI:10.1016/j.microrel.2023.115124

Terms of use:

This article is made available under terms and conditions as specified in the corresponding bibliographic description in the repository

Publisher copyright

(Article begins on next page)

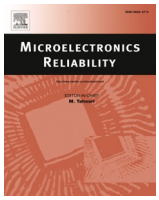
16 October 2026

Microelectronics Reliability 150 (2023) 115124

Contents lists available at ScienceDirect

Microelectronics Reliability

journal homepage: www.elsevier.com/locate/microrel



Evaluating reliability against SEE of embedded systems: A comparison of RTOS and bare-metal approaches

C. De Sio *, S. Azimi, L. Sterpone

Department of Computer and Control Engineering, Politecnico di Torino, Turin, Italy

ARTICLE INFO

Keywords:
Baremetal
Embedded processors
Fault injection
FreeRTOS
Multiple cell upset
Operating system
Radiation effects
Reliability
RTOS
Single event upset

ABSTRACT

Embedded processors are widely used in critical applications such as space missions, where reliability is mandatory for the success of missions. Due to the increasing application complexity, the number of systems using Real-Time Operating Systems (RTOSs) is quickly growing to manage the execution of multiple applications and meet timing constraints. However, whether operating systems or bare-metal applications provide higher reliability is still being determined. We present a comprehensive reliability analysis of software applications running on a device with bare-metal and FreeRTOS against the same faults based on fault models derived from a proton test. Additionally, the FreeRTOS system has been evaluated with a set of software applications dedicated to evaluating specific RTOS functions, providing an additional evaluation for operations crucial for a real-time operating system.

1. Introduction

Embedded processors have become increasingly popular in mission-critical applications such as space missions, where reliability is paramount.

The impact of soft errors on reliability can be significant in these systems, making it essential to ensure that the applications used are reliable and robust. Moreover, the continuous downscaling of transistors and operating voltages has led to more performant devices. Such devices are appealing for high-performance mission-critical applications, such as space missions. However, smaller transistors dimensions, operating voltages, and higher frequency made them more vulnerable to soft errors, which is a primary concern for systems deployed in harsh environments, such as space, where the exposure to ionizing radiation is a source of malfunctions in the device [1]. As the complexity of tasks that embedded systems must perform continues to increase, the bare-metal approach has decreased, leading to migration towards adopting Real-Time Operating Systems (RTOSs). These systems provide an efficient solution for meeting stringent real-time requirements, particularly in safety-critical applications that require the management of the execution of multiple critical applications on the same platform [2]. Despite the widespread use of embedded processors, the robustness of software applications running on systems with RTOS compared to bare-metal has yet to be thoroughly investigated. This leads to a broad question of whether these two platforms' reliability differences exist when they run applications in safety-critical missions.

1.1. Main contributions

This paper is dedicated to performing an accurate and comprehensive reliability analysis of two developed platforms running the same applications in the presence of the same fault models, relying on RTOS and bare-metal, respectively. The paper presents two main contributions. Firstly, a detailed analysis of the fault model occurring in the on-chip SRAM memory of an ARM Cortex-A9 embedded processor during a proton test is presented. The observed events are used to propose a set of fault models for realistically emulating radiation-induced soft errors. These fault models provide a model for radiation-induced errors observed in the on-chip memory from the processor side. Secondly, we proposed two reliability analyses for two platforms running on ARM Cortex-A9 embedded processor of a Zynq-7020 system-on-chip. FreeRTOS and bare-metal platforms are evaluated using the same software applications suite. Finally, since RTOS systems are characterized by additional features that are missing in bare-metal, an additional analysis based on a suite of benchmark applications is provided for exploring the robustness of specific RTOS functions.

* Corresponding author.
E-mail address: corrado.desio@polito.it (C. De Sio).

https://doi.org/10.1016/j.microrel.2023.115124
Received 1 June 2023; Received in revised form 10 July 2023; Accepted 11 July 2023
Available online 1 October 2023
0026-2714/© 2023 The Authors. Published by Elsevier Ltd. This is an open access article under the CC BY license (http://creativecommons.org/licenses/by/4.0/).

2. Related works

Several works have investigated the impact of soft errors on processor systems using different techniques [3]. During accelerated radiation testing, an embedded device is exposed to a high flux of radiation while the application is running, simulating years of functioning in a radiation environment such as space [4].

Even if accelerated radiation testing provides the closest results to the actual case scenario, it has a high cost of money, time, and expertise, making it unsuitable for the early stages of the design development flow. Therefore, other works are dedicated to developing alternative techniques, such as simulation and emulation environments, for assessing the reliability of bare-metal applications running on microprocessors. Fault Injection techniques, as one of the most common emulation approach, is widely exploited to evaluate the impact of Single Event Effects (SEEs) on embedded processors [5,6]. Simulation-based fault injection methodologies are developed for emulating fault by injecting faults in memory resources, CPU registers, and communication infrastructure [7], while the impact of soft errors on the operations of a microprocessor-based architecture by injecting random Single Event Upset (SEU) at a random time is investigated in [8].

However, on the other hand, the increasing complexity of tasks required for embedded systems has led to the rise in the adoption of Real-Time Operating Systems (RTOSs), which provide an efficient solution for meeting stringent real-time requirements. Therefore, with the emergence of RTOS, some works have also investigated software-level techniques for evaluating the sensitivity of software executing on embedded processors with an operating system to soft errors [9]. The vulnerability of FreeRTOS has been evaluated through a software-based fault injection method that targets the most relevant variables and data structure [10], while the authors in [11] developed a workflow for automatic fault injection into program and data memory. Common approaches are based on modifying the operating system kernel or altering the memory content [10]. However, software application-level methods abstract from underlying hardware architecture when considering the impact of faults on the operating system's functionality. Therefore, other approaches are based on the simulation of the hardware description of the embedded processing system, which allows the injection of upsets into registers and hidden elements at any time [12,13].

Although numerous research studies have focused on assessing the dependability of embedded microprocessors using both bare-metal and FreeRTOS approaches, none have specifically compared these approaches regarding their susceptibility to SEE caused by architectural faults. Designers, particularly those working on space applications, often find themselves debating whether to use an operating system or a bare-metal application to achieve higher reliability. This paper aims to fill this gap by conducting a first direct comparison of identical applications running on embedded systems using bare-metal, and RTOS approaches, taking into consideration the fault models derived from a conducted proton radiation test experiment.

3. Fault model resulting from proton testing

We performed a proton radiation test at Switzerland's Paul Scherrer Institute (PSI) proton facility. A Zynq-7020 device has been irradiated with proton beams with energies between 29 and 200 MeV. The content of the on-chip 256 Kb SRAM memory has been continuously monitored through a software routine running on an ARM Cortex A9 during the experiment. The on-chip SRAM memory of Zynq-7020 implements an error detection mechanism exploiting parity bits, while no correction mechanism is implemented. The software test routine is executed on the processor system, reading and writing the memory content. The testing routine writes new values in the memory and checks if the value written during the previous test loop has been corrupted. It also verifies that the current value has been written correctly and can be read correctly. When an erroneous value is detected, it is notified to a host computer connected through a serial connection. The software routine can identify both Single Event Functional Interruption (SEFI) errors (e.g., a memory cell cannot be written or read correctly anymore) and soft errors, such as Single Event Upset and Multiple Cell Upset (MCU). We identified the following fault models and their cross-sections, reported in Fig. 1, that have been adopted in the fault injection campaigns:

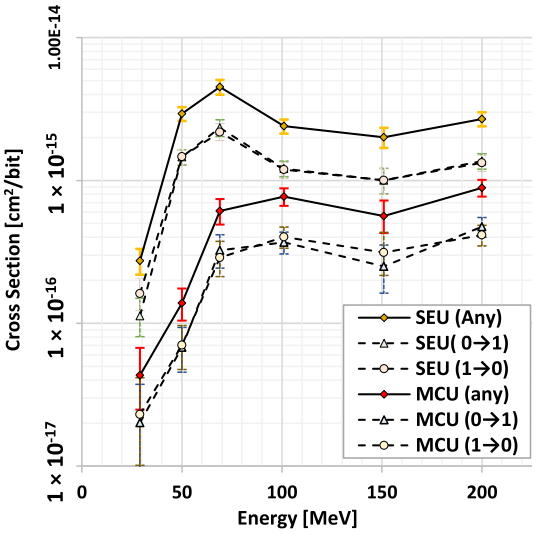


Fig. 1. SEUs and MCUs cross-sections.

Table 1
Normalized occurrences of address offset in MCUs.

Offset in memory between bitflips	Normalized occurrence
128	0.61
4	0.12
124	0.06
132	0.03
Others	Less than 0.01 each

Table 2
Normalized occurrences of affected bits in MCUs.

Number of multiple upset	Normalized occurrence
2	0.65
3	0.20
4	0.08
5	0.03
Others [6;15]	Less than 0.01 each

Single Event Upset is the most commonly observed event during radiation experiments. An SEU produces a change in the value stored in a memory cell due to a bitflip, leading to data corruption in memory. Fig. 1 displays the SEU cross-section for various proton energy experiments, while radiation-induced transitions from 0 to 1 and 1 to 0 have also been examined, revealing comparable ratios and distributions.

Multiple Cell Upset is a group of SEUs that happen simultaneously, usually due to a single event. Indeed, a single particle can be the source of multiple upsets by exciting more than one logic cell while traversing or corrupting the memory control signals, resulting in the corruption of numerous cells. In particular, a Multiple Bit Upset (MBU) is an MCU affecting two bits of the same logical word. Interestingly, no MBU has been observed. However, detected MCUs presented recurrent patterns. The MCUs always affected the same significant bits of equally distanced logic memory words. This characteristic is likely related to the specific characteristics of the layout and architecture of the memory under test that differ from the logical organization. The distribution of the parameters, aggregating the characteristics of events occurring at all the

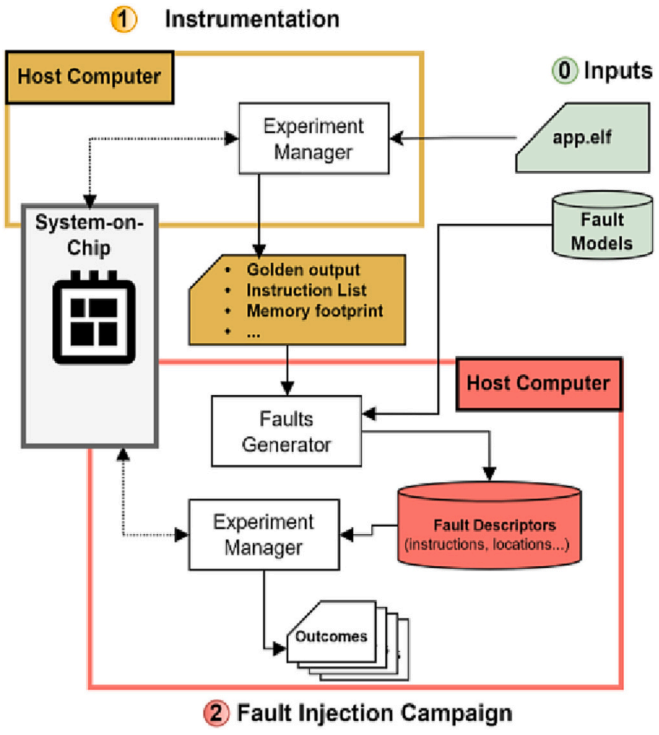


Fig. 2. Fault injection framework.

energies, of MCUs events are reported in Tables 1 and 2. Additional details on the proton test experiments are reported in [14].

4. The reliability analysis environment

An analysis environment has been developed to assess and compare the reliability of the two platforms against the fault models observed during the proton test experiment. The developed framework allows emulating the faults affecting the on-chip memory of the system-on-chip. Since the difference between the two platforms is only in the software stack, emulating the fault at a lower level provides a fair comparison between the two systems. In particular, the same (i.e., same fault model and location) faults are emulated and evaluated on both systems. The experiment manager runs on a host computer connected to the embedded platform through a serial connection. Using the JTAG interface of the SoC, the experiment manager can run the application on the target device, and stop the execution, manipulating the memory of the SoC to emulate the fault model, resuming the execution, and collecting the results. The reliability analysis environment, depicted in Fig. 2, includes two stages. In the first stage, the system runs a version of the application under test without faults to collect data for instrumenting the fault injection process. During this stage, the environment infers information such as the application's memory footprint, average execution time, and executed instructions. This stage is fully automated and generates the information to be used in the second stage. The fault injection process flow is performed as follows: firstly, a fault injection location and time are generated. Fault location is generated among the SRAM memory of the device. Fault injection time is then generated. In this context, fault injection time refers to the point in time during the execution of the program when a fault is intentionally injected into the system. Selecting the specific point in the program's execution when the fault will be injected is done by choosing an instruction from the list of executed instructions retrieved during the instrumentation stage. By injecting faults at different times, it is possible to simulate different scenarios and observe how the system responds. Secondly, the application under test is executed in the FreeRTOS and bare-metal versions.

Table 3
Characteristics of general purpose applications Suite.

Application	Platform	Memory footprint (KByte)	Nominal execution time (ms)
qsort	bare-metal	7932	46.45
qsort	FreeRTOS	68,796	1049.93
matmul	bare-metal	37,580	45.67
matmul	FreeRTOS	75,348	1047.72
basicmath	bare-metal	37,068	120.49
basicmath	FreeRTOS	74,956	1116.15
dhrystone	bare-metal	43,636	194.69
dhrystone	FreeRTOS	78,940	1183.18

Once the application runs, it is stopped at the specific instruction selected as the fault injection time. This is done using the debugging mode, which allows for precise control over the execution of the program. At this point, the fault is emulated at the memory level. This means that the fault is emulated into the system by altering the contents of memory in a specific way that mimics the target fault model. During the fault injection process, monitoring the system for any issues that may arise due to the injected fault is essential. To do this, the experiment manager uses timers instrumented during the process's first stage. These timers can detect system halt or endless loops, indicating that the fault has caused the program to behave unexpectedly or crash.

5. Experimental analysis

The illustrated framework and methodology have been used for two reliability analyses. The former is dedicated to comparing the robustness of systems based on bare-metal and FreeRTOS software stacks against fault models observed in the on-chip memory of an embedded processor during proton testing by using the same suite of software benchmarks and injected faults. The latter analysis evaluates the robustness of a different suite of software applications dedicated to specific features of the RTOSs, such as task communication and scheduling.

5.1. Software systems

We used two software suites in this section's reliability analyses. The first suite of software benchmarks, called *general-purpose software applications*, consists of four software applications. The four software are:

- *qsort:* a quick sort algorithm used for sorting arrays of data
- *matmul:* mathematical operations on matrices,
- *basicmath*: a set of basic mathematical functions, including arithmetic, trigonometric, and logarithmic functions.
- *dhrystone:* it is a computing benchmark that performs string processing operations.

Table 3 provides information on the memory footprint and the nominal execution time of these applications.

The software applications of this suite have been implemented both for bare-metal and FreeRTOS platforms. In particular, the FreeRTOS version of each application is coded to instantiate three copies of the same task that runs concurrently on the processor. Since bare-metal systems do not support the concurrency of tasks, the bare-metal version executes the task in sequence three times through function calls to the same procedure, which consequently share the same code section, similar to what happens when multiple instances of the same task are instantiated in the FreeRTOS system.

The second suite of software benchmarks, referred to as *RTOS software applications*, consists of five software applications extracted from the Rhealstone benchmark applications suite [15]. The Rhealstone

Table 4
Characteristics of ROTS applications suite.

Application	Platform	Memory footprint (KByte)	Nominal execution time (ms)
Task switching	FreeRTOS	67,348	98.68
Task preempting	FreeRTOS	67,412	96.05
Semaphore operations	FreeRTOS	67,668	193.06
Deadlock breaking	FreeRTOS	67,859	209.21
Task communication	FreeRTOS	67,548	204.67

benchmark suite consists of software applications aiming to evaluate operations that are critical in a real-time operating system. In particular, the software applications are:

- *task switching*: it performs synchronous and non-preemptive task switching.
- *task preempting*: it switches tasks due to an event trigger.
- *semaphore operations*: it performs semaphore operations to support mutual exclusion between two tasks.
- *deadlock breaking*: it resolves deadlock conditions by high-priority tasks preempting a low-priority task that acquired a needed resource.
- *task communication*: it makes two tasks exchange a message.

Since these applications are dedicated to evaluating the robustness of specific features offered by RTOS systems, they have been evaluated only for the FreeRTOS platform.

Table 4 provides information on the memory footprint and the nominal execution time of these applications.

All the reported software applications have been evaluated singularly in dedicated fault injection campaigns. All the software applications have been compiled using gcc with the —O2 optimization.

Additionally, on-board DRAM memory is not used by the hardware platform, limiting the memory space for the application (e.g., heap, stack, data, instruction, and so on) to the on-chip SRAM memory.

The memory footprint column reported in Table 3 and the same column in Table 4 does not include the heap and stack. The same stack size was used for all bare-metal applications and set to 14,336 bytes, while the heap size was set to 8192 bytes.

The platform based on FreeRTOS was a FreeRTOS 10 version provided by ARM-Xilinx to be implemented using the Xilinx Vitis IDE v2022.1.0 in a Zynq-7020. We used the standard configuration options provided by the vendor for all the software benchmarks based on FreeRTOS. They include full support for counting semaphores and mutex. The detection of stack overflow using FreeRTOS's methodology 2 is supported as well. The default configuration uses a total heap size of 65,536 bytes, while the minimum stack size is set to 200 words. The stack size we allocated to each FreeRTOS task is 200 words as well [16].

5.2. Fault injection campaigns

We performed a dedicated fault injection campaign based on the two proposed fault models for each software application. In order to compare the bare-metal and FreeRTOS in the fairest way possible for the *general-purpose applications* suite, we evaluated them with the same fault models and locations. However, since the FreeRTOS and bare-metal versions of the application execute different instructions due to different software stacks, it is impossible to inject faults at precisely the same moment (i.e., executed instruction). The fault injection time is generated when comparing the same application on different platforms in order to emulate faults that occur at similar execution times during execution. Each fault injection campaign consisted of 10,000 experiments. During each experiment, a single fault model at a time was injected into the SRAM on-chip memory. The characteristics of the injected fault models (e.g., characteristics reported in Tables 1 and 2) are inferred from the radiation test results. Since during the fault injection experiment, differently from the radiation testing, we emulate the happened fault effect using a fault model, a cross-section of the events is not considered during fault injection campaigns. However, since from the fault injection campaign we obtain the response of the system to an occurring fault, the results of such analysis can be combined with expected event rates in memory for obtaining the expected application cross-section under different radiation conditions.

The presented reliability analysis is based on SEU and MCU fault models emulated in the on-chip SRAM memory of the actual device, but we want to emphasize that faults in the SRAM on-chip memory are only a part of the SoC that is sensitive to radiation events. For instance, due to its high performance and minimal sizes, cache memory has the downside of being extremely sensitive to SEUs, and the choice to disable or not is still debated and based on the specific applications and their reliability and real-time constraints [17,18]. Additionally, cache memory are also a source of unpredictability in the system that can increase the complexity of hard real-time systems. Other memories can also be used with embedded processors, such as external DDR memories. However, we chose to focus our analysis on on-chip memory since it is integrated with the SoC itself; it is the biggest on-chip memory space (e.g., compared to register files and caches) and is particularly sensitive to SEUs and MBUs.

5.3. Results classification

The effects of the injected faults are categorized into four groups accordingly to observed impacts on the system. We identified the following categories:

- *Masked:* the fault did not visibly affect program execution. The program results are correct.
- *Silent Data Corruption (SDC):* the fault produced a corruption of the program output.
- *Crash:* the fault produced a system failure, causing the system to stop functioning. In this case, part of the output was generated before the systems stopped working.
- *Startup Failure*: the fault prevents the application from emitting any output due to an early crash or failing boot.

To clarify further the difference between a *Crash* and a *Startup Failure*, software applications have been coded to output a signature when the application under test starts to execute. A fault is classified as causing a *Startup Failure* when no output, including the starting signature of the program, is generated. Both *Crash* and *Startup Failure* cause the system to halt due to various reasons, such as endless loops or unhandled exceptions.

6. Experimental results

The results of a first reliability analysis dedicated to evaluating the *general-purpose application suite* implemented in bare-metal and FreeRTOS platforms against the SEUs and MCUs are presented. Additionally, we present the result of a second reliability analysis dedicated to features typical of RTOS. This dedicated reliability analysis has been carried out only for the FreeRTOS platform and evaluated against SEUs and MCUs fault models.

Reliability analyses have been conducted using statistical fault injection. We carried out fault injection campaigns of 10,000 singularly-evaluated fault injections. In accordance with [19], it allows us to reach a confidence interval of 95 % with less than 1 % of the margin of error of the measured error rate values. SEUs and MBUs fault models resulting from proton testing have been emulated into the on-chip SRAM memory of the embedded system. The results are reported using the error rate value. Since the reliability analysis is carried out through fault injection campaigns, this ratio represents the number of faults injected

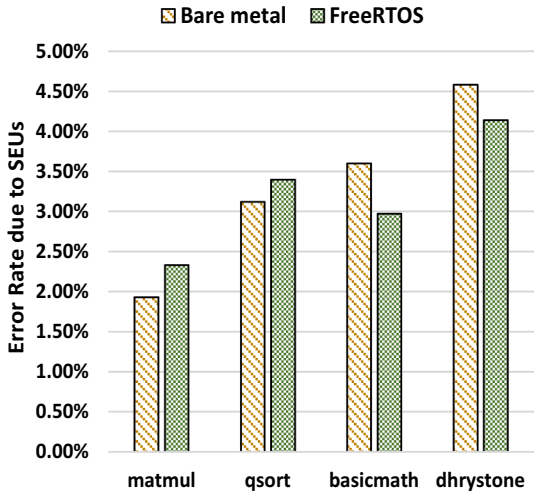


Fig. 3. Error rate resulting from SEUs fault model for *general-purpose applications* benchmarks.

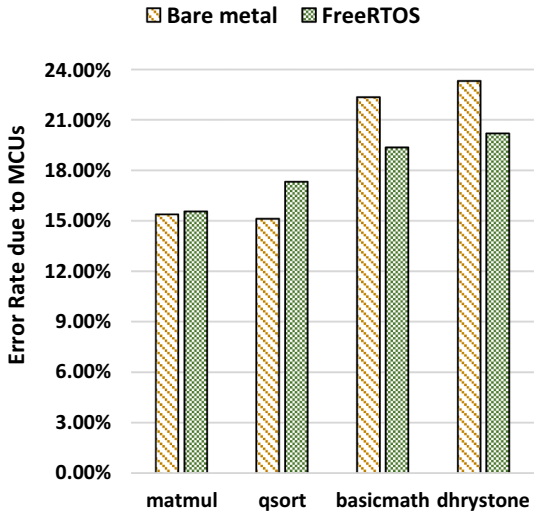


Fig. 4. Error rate resulting from MCUs fault model for *general-purpose applications* benchmarks.

that led to an error (i.e., the number of not masked faults) over the total number of events emulated in the system (i.e., the number of experiments). This means that we are evaluating the probability of having an anomalous behavior of the system, given that an SEU or MCU occurs during execution. Using such information, it is possible to compute the expected application cross-section value for different scenarios based on the expected radiation profile and expected SEU or MCU rate in time. For the same reason, the error rates of the applications resulting from this analysis are also independent of execution times since they only consider a single fault happening during execution. However, the execution time reported in Tables 3 and 4 can be combined with the expected SEU and MCU rates to map the applications' error cross-section to different radiation profiles and scenarios.

6.1. Baremetal and FreeRTOS comparison analysis

Error rates due to SEUs and MCUs affecting the software applications of the *general-purpose application suite* running on both platforms are presented in Figs. 3 and 4, respectively.

The two analyses produced similar reliability results for the evaluated applications. The resulting error rates against these fault models.

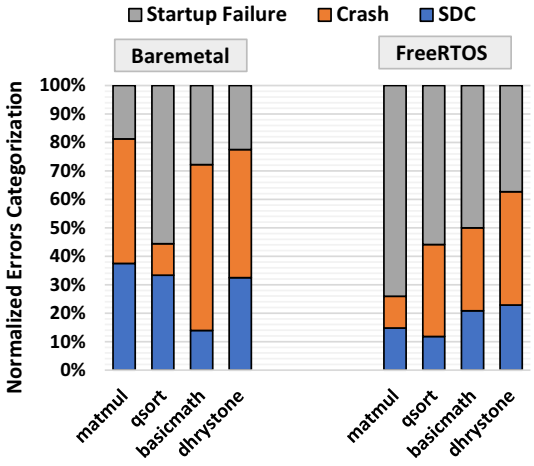


Fig. 5. Normalized error categorization for SEUs fault model for *general-purpose applications*.

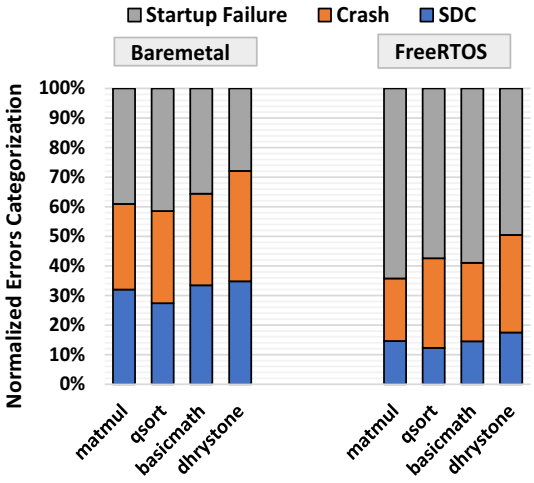


Fig. 6. Normalized error categorization for MCUs fault model for *general-purpose applications*.

Robustness comparisons among software are the same for both fault models, and the error rates vary only marginally between bare-metal or FreeRTOS based on the specific application. As a result, choosing between bare-metal or FreeRTOS can lead to slightly more robust software based on the specific application, but robustness can be considered comparable without significant variations. Since the marginal variation of the error rate when using bare metal or FreeRTOS is very small, the choice between the two mainly depends on other factors, such as more or less strong real-time requirements. However, it is interesting to notice that the distribution of the type of errors presents a pronounced difference for SEUs, which is even more marked for the MCU fault model. As shown in Figs. 5 and 6, while the error rate is similar, FreeRTOS show a significantly higher percentage of execution flow error, such as *Startup Failures and Crashes*. This is likely due to the higher complexity of the operating system layer introduced by FreeRTOS in the software stack. Differently, bare-metal is more prone to SDC errors. As a result, bare metal could be considered more suitable for systems where high availability is essential and erroneous results can be tolerated. On the other hand, SDCs are less common in FreeRTOS, which is a valuable feature since there is no advisory on the system's misbehavior in this type of error. However, since availability is an essential metric for a real-time system, this analysis raises the question of whether FreeRTOS can provide a reasonable level of availability while keeping low SDC when

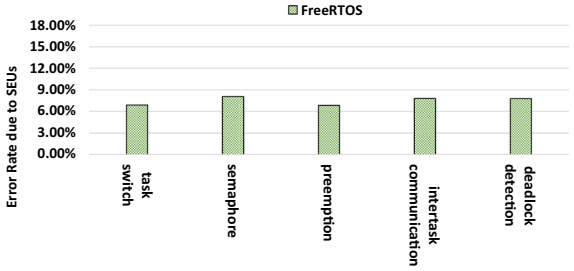


Fig. 7. Error rate resulting from SEUs fault model for *RTOS software applications* benchmarks.

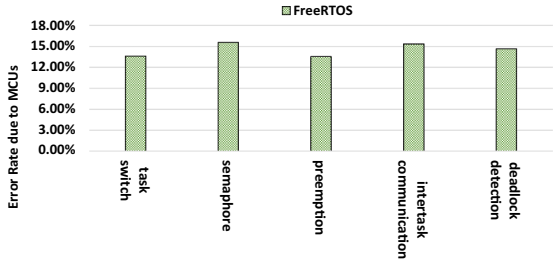


Fig. 8. Error rate resulting from MCUs fault model for *RTOS software applications* benchmarks.

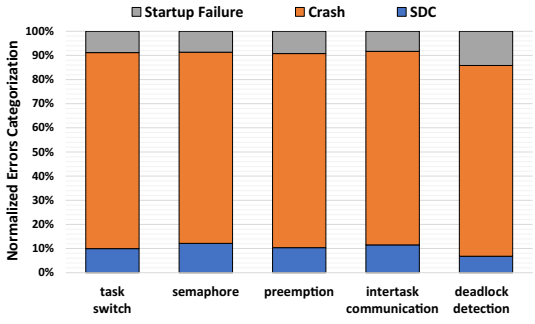


Fig. 9. Normalized error categorization for SEUs fault model for *RTOS software applications*.

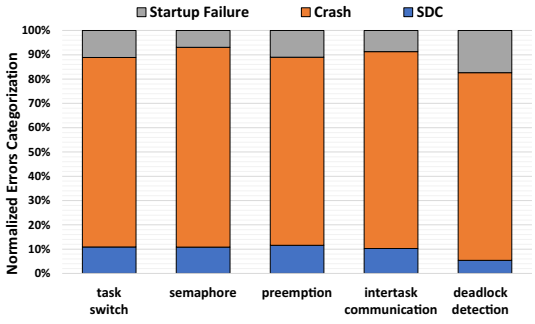


Fig. 10. Normalized error categorization for MCUs fault model for *RTOS software applications*.

operating in radiation environments.

6.2. FreeRTOS functionality analysis

The error rates resulting from reliability analysis against SEUs and MCUs of the *RTOS benchmark suite* are reported in Figs. 7 and 8, respectively. Variations of error rates appear to be less marked among various applications, and also, the error categorization reported in Figs. 9 and 10 are very similar among the software evaluated. Due to the characteristics of the benchmark under test, SDC occurred much less compared to general-purpose applications, while errors due to control flow, such as *Crash* and *Startup Failure,* are more common. This analysis supports the idea that RTOS functionality is more prone to control flow errors, especially compared to bare-metal applications where the operating-system layer introduces much less complexity. All RTOS functionalities evaluated in this analysis seem to be characterized by a similar error rate and error categorization distribution.

7. Conclusions

We proposed a reliability comparison of software running in FreeRTOS and bare metal using realistic fault models of radiation-induced soft errors affecting the on-chip SRAM memory of an ARM Cortex-A9 embedded processor. Even if characterized by a similar error rate, the experimental results highlighted the different sensitivity of the two approaches to SDCs and control flow errors, which should be considered carefully when defining the software platform for real-time safety-critical applications. A second analysis confirmed that features offered by RTOS are particularly prone to control flow errors compared to SDC.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Data availability

Data will be made available on request.

References

[1] S. Azimi, et al., Analysis of single event effects on embedded processor, in: MPDI Electronics Vol 10, 2021, https://doi.org/10.3390/electronics10243160.
[2] P.M. Aviles, et al., Radiation testing of a multiprocessor macrosynchronized lockstep architecture with FreeRTOS, IEEE Trans. Nucl. Sci. 69 (3) (2022) 462–469. March, https://doi.org/10.1109/TNS.2021.3129164. March.
[3] C. De Sio, et al., SEU evaluation of hardened-by-replication software in RISC-V soft processor, in: IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFT), 2021.
[4] S.M. Guertin, et al., Radiation specification and testing of heterogenous microprocessor SOCs, in: 2019 19th European Conference on Radiation and Its Effects on Components and Systems (RADECS), Montpellier, France, 2019, pp. 1–7, https://doi.org/10.1109/RADECS47380.2019.9745708.
[5] H. Cho, et al., Quantitative evaluation of soft error injection techniques for robust system design, in: 2013 50th ACM/EDAC/IEEE Design Automation Conference (DAC), Austin, TX, USA, 2013, pp. 1–10.
[6] A. Portaluri, et al. "On the reliability of real-time operating system on embedded soft processor for space applications"; in Architecture of Computing Systems: 35th International Conference, ARCS 2022, Heilbronn, Germany, September 13–15, 2022, Proceedings. Springer-Verlag, Berlin, Heidelberg, 181–193. doi:https://doi.org/10.1007/978-3-031-21867-5.
[7] F. Rosa, F. Kastensmidt, R. Reis, L. Ost, A fast and scalable fault injection framework to evaluate multi/many-core soft error reliability, in: 2015 IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFTS), Amherst, MA, USA, 2015, pp. 211–214, https://doi.org/10.1109/DFT.2015.7315164.
[8] Q. Lu, et al., LLFI: an intermediate code-level fault injection tool for hardware faults, in: 2015 IEEE International Conference on Software Quality, Reliability and Security, Vancouver, BC, Canada, 2015, pp. 11–16, https://doi.org/10.1109/QRS.2015.13.
[9] T. Santini, et al., Reliability analysis of operating systems for embedded SoC, in: European Conference on Radiation and Its Effects on Components and Systems (RADECS), 2015.
[10] D. Mamone, et al., On the analysis of real-time operating system reliability in embedded systems, in: 2020 IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFT), Frascati, Italy, 2020, pp. 1–6, https://doi.org/10.1109/DFT50435.2020.9250861.
[11] I.O. Loskutov, et al., Investigation of operating system influence on single event functional interrupts using fault injection and hardware error detection in ARM Microcontroller, in: 2021 International Siberian Conference on Control and

Communications (SIBCON), Kazan, Russia, 2021, pp. 1–4, https://doi.org/10.1109/SIBCON50419.2021.9438916.
[12] W. Mansour, R. Velazco, SEU fault-injection in VHDL-based processors: A case study, in: 2012 13th Latin American Test Workshop (LATW), Quito, Ecuador, 2012, pp. 1–5, https://doi.org/10.1109/LATW.2012.6261258.
[13] S. Azimi, et al., Exploring the impact of soft errors on the reliability of real-time embedded operating systems, Electronics (2023), https://doi.org/10.3390/electronics12010169.
[14] C. De Sio, et al., Analysis of proton-induced single event effect in the on-chip memory of embedded process, in: 2022 IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFT), 2022.
[15] T.J. Boger, Rhealstone benchmarking of FreeRTOS and the xilinx zynq extensible processing platform, in: MS thesis, Dept. Elect. and Com. Eng., Temple Univ., Philadelphia, PA, USA, 2013.
[16] FreeRTOS Customisation. https://www.freertos.org/a00110.html. Accessed 2023-1-6.
[17] T. Santini, P. Rech, G. Nazar, L. Carro, F.R. Wagner, Reducing embedded software radiation-induced failures through cache memories, in: 2014 19th IEEE European test symposium (ETS), Paderborn, Germany, 2014, pp. 1–6, https://doi.org/10.1109/ETS.2014.6847793.
[18] M. Rebaudengo, M.S. Reorda, M. Violante, An accurate analysis of the effects of soft errors in the instruction and data caches of a pipelined microprocessor, in: 2003 Design, Automation and Test in Europe Conference and Exhibition, Munich, Germany, 2003, pp. 602–607, https://doi.org/10.1109/DATE.2003.1253674.
[19] R. Leveugle, A. Calvez, P. Maistri, P. Vanhauwaert, Statistical fault injection: quantified error and confidence, in: 2009 Design, Automation & Test in Europe Conference & Exhibition, Nice, France, 2009, pp. 502–506, https://doi.org/10.1109/DATE.2009.5090716.