

ARFT: An Approximative Redundant Technique for Fault Tolerance

Gennaro Severino Rodrigues, Adria Barros Barros de Oliveira, Alberto Bosio, Fernanda Lima Kastensmidt, Edison Pignaton de Freitas

▶ **To cite this version:**

Gennaro Severino Rodrigues, Adria Barros Barros de Oliveira, Alberto Bosio, Fernanda Lima Kastensmidt, Edison Pignaton de Freitas. ARFT: An Approximative Redundant Technique for Fault Tolerance. 33rd Conference on Design of Circuits and Integrated Systems, DCIS 2018, Nov 2018, Lyon, France. 10.1109/DCIS.2018.8681499 . hal-03770905

HAL Id: hal-03770905
https://hal.science/hal-03770905

Submitted on 3 Dec 2022

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

ARFT: An Approximative Redundant Technique for Fault Tolerance

Gennaro Severino Rodrigues, Adria Barros de Oliveira, Alberto Bosio, Fernanda Lima Kastensmidt, Edison Pignaton de Freitas

***Abstract*—This paper presents a novel redundancy technique for software fault tolerance, named Approximative Redundant Fault Tolerance (ARFT). It uses approximate computing in order to provide the same error detection of a classic DWC method with less overhead. In this work ARFT was implemented to protect the ARM Cortex-A9 embedded into Zynq-7000 All Programmable SoC. An extensive fault injection campaign was performed to evaluate the proposed technique. Results show that distinct applications with different approximation methods present a big variation in terms of execution time, memory footprint and error detection capability. Performance analysis shows that ARFT can reduce the overhead in some benchmarks cases up to 40%.**

***Index Terms*—Fault tolerance, Embedded processors reliability, FreeRTOS, Soft error, Fault Injection**

I. Introduction

Radiation-induced Single Event Effect (SEE) is one of the primary concerns of Commercial Of The Shelf (COTS) microprocessors operating in aerospace environments. Transient effects vary from data disruptions to timeout, and function interrupts. With the late technological advances, the dimensions and operating voltages of COTS processors have been dramatically reduced, which leads to a higher susceptibility to radiation-induced errors. Radiation-hardened microprocessors are expensive. Therefore, the industry has turned to COTS embedded processors combined with fault tolerant techniques as a valid alternative for safety-critical applications [1].

Fault tolerant techniques for embedded processors can be applied at hardware-level by duplicating or triplicating the entire component and adding voters and checkers. Hardware-based redundancy introduces a prohibitive area and power overhead. Therefore, software-based fault tolerance is used to protect microprocessors with no extra hardware costs. The literature presents a myriad of software-based techniques for fault tolerance [2]. Redundancy, for instance, can be applied at the task level to protect software. Although software-based techniques present no overhead in hardware area, they introduce execution time and memory footprint overheads in the application. One example of such a technique is the Duplication With Comparison (DWC). DWC duplicates a task and compares the data generated by two executions of the code for discrepancies with a checker.

G. S. Rodrigues, A. B. de Oliveira, and F. L. Kastensmidt are with the Programa de Pós Graduação em Microeletrônica (PGMICRO), A. Bosio is with the Lyon Institute of Nanotechnology (INL), France and E. P. Freitas is with the Graduate Program in Computer Science (PPGC), Federal University of Rio Grande do Sul (UFRGS), Porto Alegre, Brazil. e-mail: {gsrodrigues,aboliverira,fglima,epfreitas}@inf.ufrgs.br, alberto.bosio@lirmm.fr.

Approximate computing has been proposed as an approach to developing energy-efficient systems [3]. The idea is having applications able to tolerate some flexibility in the final result, based on a threshold specified by the application requirements. By relaxing the need for precise data, approximate computing has been proven to improve energy efficiency. Concerning fault tolerance, approximate computing can mask a higher number of errors by relaxing data precision. Past works proved that software with approximative nature presents higher intrinsic fault tolerance than conventional algorithms [4]. Besides, approximative computation can sometimes spend less time in the calculation compared to a standard algorithm counterpart.

Observing the potential design space exploration for approximate computing as a fault tolerance technique, this work aims at investigating its usage in software-based fault detection based on DWC running in embedded microprocessors. The idea is to deploy redundant code using a simpler and faster approximate version of the original code function. A checker compares both the data from the original and the approximated function in a value threshold range that can be adjusted and is defined by the application constraints. The contribution of this work is the proposal of a novel fault tolerance technique that combines approximate computing with redundancy, providing protection with low overhead. The technique is named Approximative Redundant Fault Tolerance (ARFT).

ARFT can be universally applied to any system. In this paper, ARFT was evaluated in the ARM Cortex-A9 processor embedded in the All Programmable System-on-Chip (APSOC) Zynq from Xilinx [5]. ARFT was designed targeting single-core architectures. Benchmarks present different types of algorithms with different approximation strategies, running on top of FreeRTOS operating system. ARFT can also be easily implemented on bare-metal embedded applications as well as in a more sophisticated OS environment like Linux. The efficiency of ARFT detection was evaluated under fault injection emulated on Zynq. Results show that the proposed technique can reduce the time overhead of DWC at software level by using approximate computing without compromising error detection.

This work is organized as follows: Section II presents the background and related works. Section III introduces the proposed approximative-based software fault tolerant technique (ARFT). Section IV provides details of the experimental methodology used to validate the proposed technique, while Section V presents and discusses the obtained results. Section VI concludes the paper providing directions for future work.

II. Related Work and Background

Safety-critical applications, such as those developed to aerospace systems, need protection against radiation-induced errors. An enormous set of techniques implemented in software to protect applications against hardware errors is presented in the literature. Those are called Software-Implemented Hardware Fault Tolerance (SIHFT) [6], and achieve protection with function redundancy and variables replication. Techniques called Application-Based Fault Tolerance (ABFT) are the ones that profit from their individual application characteristics [7]. ABFT shall be specifically designed for the application under protection. Therefore, it is not scalable to a high range of applications and tends to be costly in design. Both SIHFT and ABFT come at the cost of an execution time overhead. Real-time systems do not tolerate execution time overhead, as they have to meet execution deadlines.

Redundancy methods applied to software at task level such as Triple Modular Redundancy (TMR) and DWC are employed in a multitude of systems, both to provide error detection and masking. DWC techniques are combined with re-execution methods to provide error masking. This way, an error can be detected and mitigated before becoming a failure. DWC techniques have an overhead of two times the execution time of the original application for pure redundancy and three times when applying re-execution for error masking. That overhead can, however, be dealt with using parallelism. Past works evaluated those methods resilience on parallel software and showed that, although applying parallelism reduces that overhead, it has a significant impact on the technique fault mitigation and error detection performance [8].

In [9], an approach based on task level migration is proposed for fault tolerance for aerospace FreeRTOS applications. The technique consists of migrating tasks from a faulty processor to a fault-free one. The detection of the error is done by middleware blocks assumed to be fault tolerant. The problem with that approach is that a high amount of assumptions must be taken beforehand by the programmer. Decisions (like which tasks will be migrated to which processor nodes) are made in the programming phase. Unfortunately, a programmer can never safely predict which processing nodes will fail. The approach proposed in this paper, on the other hand, requires no previous assumptions to be made by the programmer. Because of that, it is possible to state that the method here proposed can be considered to be of a more general use than the one suggested by [9].

In most of the cited works, recovery approaches are proposed to deal with the error. Nevertheless, a plethora of safety-critical applications may not need recovery. As stated in [10], real-time systems have to deal with *data freshness* requirements, which defines the time interval on which data is considered updated. For instance, an automatic navigation system may have an error during its execution, but because its data freshness has a minuscule time interval, the error will soon disappear as the algorithm goes on. Because of that, an error correction procedure is not always necessary. However, the system shall be aware of the error to put itself in a fail-safe mode. In those cases, the values of the redundant re-computations are only used for comparison and error checking. For that reason, it is not necessary for the redundant task to always have an exact output value.

III. The Proposed Fault Mitigation Approach

This paper proposes a fault tolerance method based on *approximative redundancy*. In a regular software redundancy at task level, the application to be protected is completely replicated, which may lead to a high-performance cost. This work implements redundancy without full replication of the original task, reducing the execution time for many types of applications.

The approximative redundant fault tolerance (ARFT) method uses approximative redundancy to provide fault tolerance with less overhead than other redundancy techniques. The redundancy is not implemented as a hard copy of the to-be-protected code. Instead, it is replicated and modified to achieve more performance in detriment of exactitude, with approximative computing. Because the approximative redundant task is used only to error verification and its outputs are not to be used as final application result, it can be programmed to be faster and less accurate. Thus, the redundancy overhead is reduced. ARFT consists of duplicating the task to be protected, executing both tasks, and comparing the results. If the results difference is higher than a given variation tolerance, a warning flag is activated, providing the system with the knowledge of the error. The designer may use this warning flag to implement error-correction functions or other safety measures to assure the system reliability.

Another reason to focus the approximate task on performance instead of accuracy is reliability. Execution time has a negative effect on fault tolerance. When exposed to radiation, applications with higher execution times are more prone to errors [11]. Having a faster redundancy task improves both the fault mitigation capacity and the technique's performance. Using approximate computing for fault tolerance is also interesting for means of reliability because, as shown in past works, this kind of computation can be naturally fault-tolerant [4].

Approximative computing can be deployed in several ways, from specialized hardware to software modification [12]. Most of the traditional techniques for approximative computing today make use of neural networks implemented on FPGAs. Programmable logic tends to be more susceptible to radiation errors than hardcore processors [13]. Therefore, for safety-critical systems, software-based approximation running on hardcore processors is preferable over programmable logic implementations or softcore processors. Software approximation approaches typically improve performance by loop perforation [14] or reducing the usage of costly instructions. A myriad of software methods for approximative computing is available in the literature [15], [16]. ARFT's proposal is to use any of those software approximations to implement an ideal redundancy, with only the necessary accuracy and the best performance possible. The approximation strategy to be used depends on the application to be protected. Some algorithms are better approximated in one way than another.

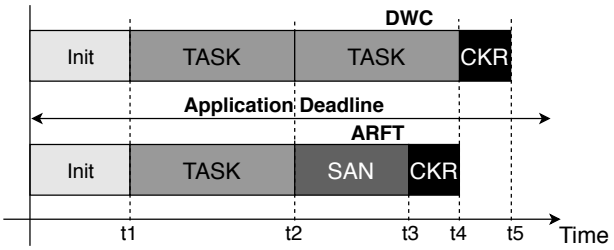


Fig. 1. ARFT Compared With DWC Functional Flow.

The ARFT methodology is a pure software approach to fault tolerance and can, therefore, be deployed in any hardware configuration, with none to little modification. In this work, ARFT is tested on an ARM-A9 embedded in the Zynq-7000 APSoC (detailed in Section IV-A). ARFT is intended for single-core processors and makes use of no parallelization strategy. When using sequential applications on single-core processors, the overhead caused by software redundancy becomes evident. In those cases, a method capable of providing fault tolerance with minimum cost is desired. It is particularly the case for real-time systems, which need to respect strict deadlines. Those systems shall make great use of ARFT. Such configuration highlights the performance improvement of using approximative redundancy instead of full redundancy. Figure 1 presents the execution flow of ARFT running a FreeRTOS task to be protected (TASK) compared with a traditional DWC. ARFT begins by executing the approximative redundancy task (called sanity, SAN) at t2, just after TASK finishes. At t3, a checker (CKR) compares both outputs taking into account the predefined acceptable difference between the values, which is different for each application (defined at Section IV-B for each benchmark). If the difference is higher than this given threshold, an error flag is raised, warning the system that the result is not to be trusted. It is clear by Figure 1 that, compared with a traditional DWC approach, ARFT is capable of safety meeting a deadline (which is a very commonly required by safety-critical systems).

IV. Experimental Methodology

The ARFT method shall be evaluated on performance, fault catch rate and error correction capacity. To acquire that data, this work studies soft-errors that affect only the processor's registers and memory. For that purpose, fault injection experiments are performed by emulation in the studied board affecting only the defined sensible region.

A. The Zynq All-Programable SoC

In this work, ARFT is implemented on the COTS Zynq-7000 APSoC, designed by Xilinx with 28 nm technology. Nevertheless, the proposed fault injection test and error classification are generic and extendable to other APSoCs. The Zynq board has embedded a high-performance dual-core ARM Cortex-A9 processor with two cache levels, alongside with an FPGA based on the Xilinx 7-Series with approximately 27.7 Mb configuration logic bits and 4.5 Mb Block RAM (BRAM). ARFT is implemented in the processor subsystem (PS) part, that is, the ARM processor, while the fault injection system uses the programmable logic (PL) part, as will be detailed further. The dual-core 32-bit ARM Cortex-A9 processor runs a maximum of 667 MHz. It counts with two L1 caches (data and instruction) per core with 32KB each, and one L2 cache with 512KB shared between both cores. A 256 KB on-chip SRAM memory (OCM) is shared between the PS and the PL, and so is the DDR (external memory interface).

B. Benchmarks

Studying only easy-to-approximate algorithms would not be realistic concerning real case scenarios (neither would be studying only hard-to-approximate ones). Therefore, two groups of benchmarks are presented. In the first group, two algorithms are proposed: Trapezoid and Newton-Raphson. Those represent algorithms that have an inherent approximative behavior or can be approximated efficiently, with techniques such as loop perforation. The second group consists of benchmark applications for which the generation of an approximative version is not straightforward. Two algorithms were chosen to represent this type of applications: Matrix Multiplication and Cubic. The four benchmarks are presented and detailed below.

1) Trapezoid: This is the perfect algorithm to be approximated by loop perforation. That is because the Trapezoid rule (numerical calculation of the area under a curve, i.e., an integral) is itself a successive method. For each iteration of the algorithm, the execution comes closer to a result of a great exactitude. By loop perforation approximation, it is possible to obtain a result with an acceptable exactitude with fewer iterations. In this work experiments, the approximative version of trapezoid achieves a result that differs from the original one in 1×10^{-1} (module).

2) Newton-Raphson: Similar to Trapezoid, this benchmark is a perfect example of how loop perforation can be used in approximate computing. It finds the zero of a function by employing the Newton-Raphson method. The difference between this benchmark and Trapezoid is that Newton-Raphson converges much quicker to the correct answer. Because of that, the complete and the approximative versions of this algorithm will have a smaller difference in the number of iterations than the Trapezoid implementations. The approximated version of newton-raphson achieves a result that differs from the original one in 1×10^{-3} on the worst case.

3) Matrix Multiplication: Given the nature of matrix multiplication, it is not an easy to approximate algorithm. There is, however, ways to do it. One is limiting the precision of data (e.g. using *float* precision instead of *double*). In this work, a matrix multiplication using double precision is presented as a non-approximated version. The approximated version is the same multiplication implemented with the use of float precision. In the architecture of the studied processor, *double* data uses 64-bit, while *float* uses 32-bit data. The outputs of the matrix multiplication using *float* precision differs from *double* precision version at a maximum of 0.23 of modular difference.

TABLE I
BENCHMARKS DETAILS

Application		Execution Time [cc]	Executed Instructions [#]	Processed Data Size [kB]	Register Usage [%]	Approximation Strategy
Trapezoid	Standart	4482792	3944856	102.800	75	-
	Approximated	1036678	953744	102.800	75	Loop Perforation
Newton-Raphson	Standart	546250	267662	0.818	56.25	-
	Approximated	434310	299674	0.818	56.25	Loop Perforation
Matrix Multiplication	Standart	2135490	277613	9.600	37.5	-
	Approximated	1995270	259385	4.800	37.5	Data Precision
Cubic	Standart	448118	282314	8.192	75	-
	Approximated	457186	288027	4.096	75	Data Precision

4) Cubic: This application comes from the automotive package of MiBench [17]. It is based on the MiBench benchmark called "Basic Math", which consists of calculations of cubic equations solutions, integer square roots, and angle conversions. For this benchmark, however, only cubic equations solutions are executed. Approximating those calculations shows the same problem of Matrix Multiplication benchmark. The difference is that, in this case, many trigonometrical functions such as sin and cosine calculations are deployed, which are by themselves already approximative calculations. Because of that, using a less precise data representation is believed to have a higher impact on the algorithm outputs. This application is approximated following that same strategy from Matrix Multiplication. The approximated version of cubic achieves a result that differs 1×10^{-6} in module from the original one at the worst case.

Table I presents important details of the benchmarks, such as execution time (in clock cycles) and the amount of data processed by them. The "executed instructions" column presents the executed instructions count. It is based on the number of clock cycles and the instructions per clock cycle of a given benchmark. The "register usage" column presents the percentage of registers used by the application, concerning the general purpose ones (the same where the faults are injected). This data is important, given that this work injects faults in those registers, as it will be further detailed.

C. Fault Injection and Error Classification

The fault injection experiments are performed on the ZedBoard, a Zynq-7000 APSoC board. Figure 2 shows the experiment setup environment. The ZedBoard and the Power Control are connected to the host computer. The USB-TTL Converter, responsible for transmitting the serial data, is connected to the ZedBoard and the host computer. The adopted methodology follows the same scheme presented in [18]. The system consists of the following modules:

- *Injector Module*: Intellectual property (IP) designed in hardware description language (in this case, VHDL) and implemented in the FPGA layer of Zynq. It is responsible for performing the fault injection procedure presented in Figure 3, to be detailed further.
- *Power Control*: Electrical device in charge of power up the board in each injection cycle.
- *System Controller*: Software application running on a host computer responsible for Power Control management. It also saves the fault injection logs, which are received by serial communication.

The injector module injects bit-flips on the processor's register file. The affected ARM registers are the general-purpose ones, from R0 to R12, and the specific ones, which are the Stack Pointer (SP), Link Register (LR), and Program Counter (PC). The faults are injected using an interrupt mechanism, which locks the processor and applies an XOR mask to the target register, provoking a bit-flip. The injection time, target register, bit to be flipped, and processor core where the fault will be injected are randomly defined. The injection time is also randomly determined, being a random point between the start and the finish of the execution. It is intended to simulate real scenarios, where the fault can affect the system any moment during the application execution. Figure 3 presents a procedure flow performed by the injector module. First, the injector is configured with all the random injection data defined by the ARM CPU0, as generating random numbers in FPGA logic has a high complexity. Once configured, the injector counts clock cycles until it reaches the injection time. Next, an interruption is launched to inject the bit-flip in the processor register defined in the configuration. In the end, the module compares the application results with a golden execution output (that is, with no fault injected) to check for errors. Errors are classified as UNACE (no error), SDC (silent data corruption, when there is an error in the output), and HANG (application not finished successfully).

When working with approximated values, it is expected that the applications applying such approximations are capable of tolerating them. However, that may not be the case. For many safety-critical applications, there is no such *margin of tolerated error* allowed. ARFT is expected to achieve the same fault detection rate of a classical DWC method with less overhead when applied to applications capable of tolerating a small deviation from expected results.

V. RESULTS AND DISCUSSION

ARFT is evaluated by comparing its SDC detection rates with the ones from an ordinary redundancy method. The results are compared with single-core complete (non-approximative) redundancy, which is a traditional DWC. The execution overheads from ARFT and DWC are also presented.

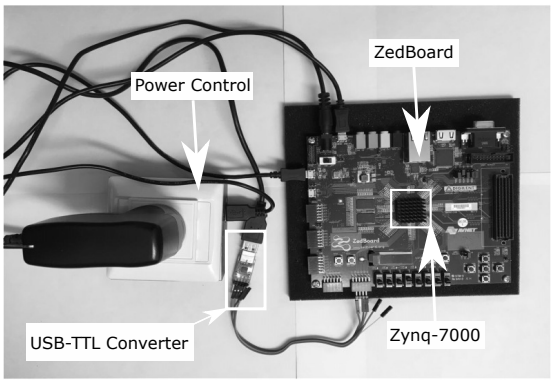


Fig. 2. Experiment setup view

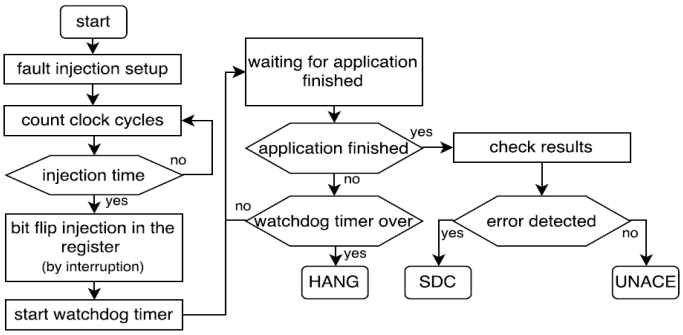


Fig. 3. Fault injection procedure flow [18]

The assessed results are shown on Table II. Two types of DWC are presented. The first one checks for errors using a precise checker, that is, by simply comparing the results. The other one uses an approximate checker that compares the results taking into consideration an acceptable threshold (the same employed by ARFT and presented at Section IV-B for each application). Because ARFT works with approximate values, it always uses an approximate checker. Subsection V-A presents the performance results for ARFT on different applications. A critical analysis of these results is developed in subsection V-B.

A. ARFT Performance

Table II presents the performance of ARFT and traditional DWC methods. Overhead values are shown concerning the unprotected version of the algorithm, so an overhead with the value equal to 2 means that the benchmark spent two times the execution time of the unprotected version.

Results show that ARFT detected the same percentage of SDCs of the DWC using the approximated checker for all the benchmarks, except for Cubic. On the Cubic application, not all the SDCs were detected: even DWC with precise checker is not capable of detecting all errors. Nevertheless, ARFT was still capable of achieving a detection ratio close to 100%. Using approximate checker did not impact the error detection rate of traditional DWC. Data shows that both Trapezoid and Newton-Raphson presented a significant improvement on overhead using ARFT. Nevertheless, even though ARFT achieved an excellent SDC detection on the Matrix Multiplication benchmark, there was a minimal gain on overhead. The same applies to the Cubic benchmark.

TABLE II
ARFT FAULT DETECTION AND PERFORMANCE ANALYSIS

Application	Version	SDC Detection		Execution Overhead
		Precise Checker	Approx. Checker	
Trapezoid	DWC	100	100	2.02
	ARFT	-	100	1.24
Newton-Raphson	DWC	100	100	2.12
	ARFT	-	100	1.93
Matrix Multiplication	DWC	100	100	2.09
	ARFT	-	100	2.05
Cubic	DWC	98.94	99.44	2.30
	ARFT	-	98.28	2.26

B. Critical Analysis of the Obtained Results

On almost all benchmarks, ARFT overhead is smaller than the DWC with the precise checker. The results show that the expectations ARFT were realistic.

The overhead reduction of the Newton-Raphson benchmark is not as remarkable as the one from Trapezoid. As explained at Section IV-B2, Newton-Raphson's approximative version is not much different from its complete counterpart regarding the number of iterations. Because of that, its overhead is not very different either. However, it is clear that even for algorithms like this one, the ARFT method achieves the same efficiency as complete redundancy with considerably lower overhead.

The results from Matrix Multiplication and Cubic show that approximation by changing data precision does not highly impacts the performance. An analysis of the benchmark code easily explains it. The Cubic benchmark makes use of math functions from the *math.h* C library. Those functions return *double* type values. As explained in section IV-B4, this benchmark was approximated by changing the data types from *double* to *float*. Because of that, the usage of the math.h functions imply in type conversions during the execution, causing performance loss. It is, however, important to notice that the used fault injection system always injects one fault per execution, and only on registers. Because of that, the amount of data being used and the application execution time has little impact on fault tolerance in this work.

Past works that examined fault tolerance under radiation proved that the application cross-section is profoundly influenced by the size of memory in use and execution time [19] (which is a behavior not observable in this work experiments). Both Cubic and Matrix Multiplication benchmarks are profoundly affected by the size of their inputs and computed data. The approximation by data precision implies a considerable reduction of the processed data size, as shown in Table I. Therefore, it is reasonable to consider that if tested by a radiation experiment, those algorithms, making use of ARFT, would present better fault detection than an ordinary DWC. In

light of that, the low overhead reduction presented by ARFT for those applications is an acceptable burden.

The fact that the approximate checker does not impact the SDC detection of DWC indicates that approximation may be acceptable for fault tolerance purposes. It shows that faults usually provoke errors of medium or significant magnitude. Because of that, approximation may be used as means of accelerating the fault detection without reducing fault coverage. Errors may occur not only in the thread to be protected but also on the redundant task causing false detections. Taking it into consideration, another positive impact of approximative computing on fault tolerance is that it can be less prone to errors [4], reducing the number of false positives that may be present on conventional redundancy techniques.

It is clear that the approximation strategy is crucial for the overhead reduction. The results show that applications using loop perforation to approximate the code presented a much higher execution time overhead decrease than the ones that used data precision approximation. Both approximation methods seem to achieve the same SDC detection. However, the reduction in processed data size obtained by data precision approximation (Table I) needs to be taken into account: not only because of the behavior it may present under radiation experiments, but also because of resource usage. Some embedded systems have limited resources, making a full DWC impossible. Due to the reduced memory usage of ARFT with data precision approximation, it can be implemented on systems where a classic error check mechanism usage would be impossible. If data precision approximation were applied to both the original and the redundancy task, ARFT would provide complete SDC detection with no memory overhead (comparing with the execution of a single task with *double* precision computation).

VI. Conclusion

This work presented a technique which uses approximative computing for error detection purposes. ARFT is capable of detecting the same amount of error as the ordinary duplication with comparison method with less overhead for most cases. Results show that approximative computing can be safely used for fault tolerance means and present good performance. A previous analysis of the code to be protected is enough to unveil its protection capacity by this technique.

A myriad of algorithms used on safety-critical systems has an approximative output nature, making small variances in output acceptable. Variables such as velocity, acceleration and wind direction are critical for an avionics system, yet their values are approximations. Critical systems' tasks making use of this kind of variables (or outputting them) arise as suitable candidates for protection with ARFT.

Compared to traditional redundancy check techniques, ARFT was able to achieve at least the same level of error detection. Despite the need for an in-depth study of its radiation behavior, data presented in the literature indicate that ARFT would rise as a better error detection technique than ordinary (and approximated) DWC. Future works include repeating the tests on real radioactive environments to achieve more reliable results.

References

[1] M. Pignol, "Cots-based applications in space avionics," in *2010 Design, Automation Test in Europe Conference Exhibition (DATE 2010)*, March 2010, pp. 1213–1219.

[2] G. K. Saha, "Software based fault tolerance: A survey," *Ubiquity*, vol. 2006, no. July, pp. 1:1–1:1, Jul. 2006. [Online]. Available: http://doi.acm.org/10.1145/1147994.1147995

[3] J. Han and M. Orshansky, "Approximate computing: An emerging paradigm for energy-efficient design," in *2013 18th IEEE European Test Symposium (ETS)*, May 2013, pp. 1–6.

[4] G. S. Rodrigues and F. L. Kastensmidt, "Evaluating the behavior of successive approximation algorithms under soft errors," in *2017 18th IEEE Latin American Test Symposium (LATS)*, March 2017, pp. 1–6.

[5] Xilinx. (2017) Zynq-7000 all programmable soc overview. [Online]. Available: http://xilinx.com/

[6] O. Goloubeva, M. Rebaudengo, M. S. Reorda, and M. Violante, "Soft-error detection using control flow assertions," in *Proceedings 18th IEEE Symposium on Defect and Fault Tolerance in VLSI Systems*, Nov 2003, pp. 581–588.

[7] K.-H. Huang and J. A. Abraham, "Algorithm-based fault tolerance for matrix operations," *IEEE Transactions on Computers*, vol. C-33, no. 6, pp. 518–528, June 1984.

[8] G. S. Rodrigues, F. Rosa, . B. de Oliveira, F. L. Kastensmidt, L. Ost, and R. Reis, "Analyzing the impact of fault-tolerance methods in arm processors under soft errors running linux and parallelization apis," *IEEE Transactions on Nuclear Science*, vol. 64, no. 8, pp. 2196–2203, Aug 2017.

[9] M. Fayyaz and T. Vladimirova, "Fault-tolerant distributed approach to satellite on-board computer design," in *2014 IEEE Aerospace Conference*, March 2014, pp. 1–12.

[10] E. P. de Freitas, M. A. Wehrmeister, E. T. Silva, F. C. Carvalho, C. E. Pereira, and F. R. Wagner, *DERAF: A High-Level Aspects Framework for Distributed Embedded Real-Time Systems Design*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, pp. 55–74.

[11] F. Restrepo-Calle, S. Cuenca-Asensi, A. Martínez-Álvarez, E. Chielle, and F. L. Kastensmidt, "Application-based analysis of register file criticality for reliability assessment in embedded microprocessors," *Journal of Electronic Testing*, vol. 31, no. 2, pp. 139–150, 2015. [Online]. Available: http://dx.doi.org/10.1007/s10836-015-5513-9

[12] Q. Xu, T. Mytkowicz, and N. S. Kim, "Approximate computing: A survey," *IEEE Design Test*, vol. 33, no. 1, pp. 8–22, Feb 2016.

[13] L. A. Tambara, P. Rech, E. Chielle, J. Tonfat, and F. L. Kastensmidt, "Analyzing the impact of radiation-induced failures in programmable socs," *IEEE Transactions on Nuclear Science*, vol. 63, no. 4, pp. 2217–2224, Aug 2016.

[14] S. Sidiroglou-Douskos, S. Misailovic, H. Hoffmann, and M. Rinard, "Managing performance vs. accuracy trade-offs with loop perforation," in *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, ser. ESEC/FSE '11. New York, NY, USA: ACM, 2011, pp. 124–134. [Online]. Available: http://doi.acm.org/10.1145/2025113.2025133

[15] E. Schkufza, R. Sharma, and A. Aiken, "Stochastic optimization of floating-point programs with tunable precision," *SIGPLAN Not.*, vol. 49, no. 6, pp. 53–64, Jun. 2014. [Online]. Available: http://doi.acm.org/10.1145/2666356.2594302

[16] S. Misailovic, M. Carbin, S. Achour, Z. Qi, and M. C. Rinard, "Chisel: Reliability- and accuracy-aware optimization of approximate computational kernels," in *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications*, ser. OOPSLA '14. New York, NY, USA: ACM, 2014, pp. 309–328. [Online]. Available: http://doi.acm.org/10.1145/2660193.2660231

[17] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown, "Mibench: A free, commercially representative embedded benchmark suite," in *Proceedings of the Fourth Annual IEEE International Workshop on Workload Characterization. WWC-4 (Cat. No.01EX538)*, Dec 2001, pp. 3–14.

[18] Á. B. de Oliveira, L. A. Tambara, and F. L. Kastensmidt, *Exploring Performance Overhead Versus Soft Error Detection in Lockstep Dual-Core ARM Cortex-A9 Processor Embedded into Xilinx Zynq APSoC*. Cham: Springer International Publishing, 2017, pp. 189–201.

[19] H. Quinn, "Challenges in testing complex systems," *IEEE Transactions on Nuclear Science*, vol. 61, no. 2, pp. 766–786, April 2014.