A Framework and Protocol for Dynamic Management of Fault Tolerant Systems in Harsh Environments

Eduardo Weber Wächter, Server Kasap, Xiaojun Zhai, Shoaib Ehsan, and Klaus McDonald-Maier
University of Essex, UK
{eduardo.wachter,server.kasap,sehsan,xzhai,kdm}@essex.ac.uk

***Abstract*—Robots can be used to deal with hazardous materials like nuclear waste. Unfortunately, electronic components are also susceptible to radiation effects. Current proposals to tackle this issue solve only parts of the problems for the specific scenarios and the specific types of radiation. At the same time, current computational devices should provide run-time capabilities to monitor and adapt to different situations. In this paper, we target a possible solution presenting a framework which provides the flexibility to employ fault-tolerant techniques on distributed systems. As proof of concept, we target a fault-tolerant technique to extend the operating time of systems in harsh environments. Results show a very low overhead, of few microseconds, to execute a majority voter with replicated tasks.**

***Index Terms*—fault tolerance, spatial redundancy, software redundancy, SEE, SEU, soft errors, TID effects.**

I. Introduction

Nuclear power plants have been used for the last six decades in many European countries and across the world. Due to the nature of nuclear power generation, the process of transforming radioactive material into energy also creates a by-product, named radioactive waste. Unfortunately, no country has opened a permanent storage site for such hazardous materials so far. Spent nuclear fuel and other contaminated materials are stockpiled in temporary locations around Europe and the world, sometimes in facilities open to air which might lead to contamination. In the United Kingdom the Nuclear Decommissioning Authority (NDA) [1], for example, estimates that the process to clean up 17 nuclear sites may take around 120 years, with estimated costs of hundreds of billions sterling pounds.

One possible solution to this issue is the adoption of robots since most of these materials are too hazardous for humans handling. Usually, such places are named harsh environment and can generally be described as a setting in which survival is difficult or impossible. In this particular paper, we focus on harsh environments which contain different kinds of radiation that might damage electronic circuits and human beings.

When using electronic devices, human beings are spared from entering harsh environments. However, this is not a straightforward task because electronic circuits in these robots are also susceptible to the radiation damages; this has become even clearer after the Fukushima Daiichi nuclear power plant in Japan suffered a series of meltdowns as a result of the failure of its safety systems due to a tsunami. Robots dispatched into the accident site to monitor radiation levels and facilitate the cleaning-up process have kept breaking down and failing very soon after entering as their circuits were destroyed by the radiation, thus turning the entrance of the facility into a graveyard of robots [2]. Henceforth, if robots are to be deployed in such scenarios, the behaviour of electronic circuits in extreme nuclear radiation environments remains to be thoroughly studied, and radiation effects to be mitigated.

Most of the works targeting on the radiation fault tolerance propose solutions to Single-Event Effects (SEE) [3], [4], [5], [6], [7], therefore focusing on the types of radiations encountered in space. Although Total Ionizing Dose (TID) effects on low-level devices have been reviewed in literature [8], [9], few works focused on the solutions to provide reliability on higher levels. In the case of SEEs, some available techniques can expand the lifetime of the systems, but if exposed for long periods of radiation, it will accumulate TID effects which are inevitably going to result in total failure. Therefore, a distributed approach where multiple entities capable of taking decisions in parallel is required, especially when radiation doses are unknown. For example, in the places with higher radiation doses even systems using triple modular redundancy (TMR) cannot guarantee a correct execution because. One possible solution for this case is to employ a system which understands the fault rate it is exposed to and then take actions based on the available resources.

To deal with such environments, many proposals have been employed in the state-of-the-art: (i) hardware enhancement to provide fault tolerance for circuits [6], [7]; (ii) software fault-tolerant techniques [4], [5] and (iii) a mix of different proposals [3]. To the best of the authors' knowledge, there is no such system or framework which can provide the flexibility to combine such different techniques. Therefore, in this paper, we target a possible solution to this open research challenge.

Our proposal is a high-level approach to manage distributed boards in a harsh environment. These boards can be off-the-shelf (OTS) multi-core SoC or FPGA boards. Our proposal can be used with available techniques, then providing two-layer fault tolerance capabilities. For example, techniques aforementioned using in-board spatial redundancy or software redundancy would benefit from a higher level of management and therefore extending the system's life. The idea is to adopt multiple inexpensive OTS boards instead of using one large and normally expensive board.

We present a framework which provides the flexibility to employ different techniques on distributed systems. We target a well established fault-tolerant technique as proof of concept, showing a case of study to demonstrate the usage of the framework. Further, current computational systems should provide more self-awareness: an emerging computing paradigm that helps systems to understand, manage and report on their system behaviour [10]. In this way, the system should monitor and take actions by itself, required to tackle different solutions for different scenarios. Fault-tolerant systems should also employ self-healing techniques, i.e. an attempt to enable computing systems to discover, diagnose, and repair (or at least mitigate) faults by itself, without human intervention.

The paper is organized as follows. Section II provides the effects of radiation on electronics in general, soft errors and TID effects. Section III describes the proposed framework with a focus on actual examples. Section IV presents a proof of concept using a well-known technique within the proposed solution to provide fault tolerance. Sections V shows results regarding application performance and overheads and finally, conclusions are drawn in Section VI.

II. Background

A. Harsh Environments and Radiation

Ionizing radiation can damage electronics in two significant ways. Radiation effects on electronics can either temporarily change the behaviour of a circuit (a soft error), or permanently damage the circuit (a hard error).

The first-way radiation damages electronics is called as total ionizing dose [11] which refers to the cumulative, permanent damage in an electronic device causing it to degrade over time (i.e. hard error). It takes place when charge carriers are implanted into the device's insulators as radiation strikes, where they consequently get trapped altering the electrical characteristics of the integrated circuits [12].

The second major category of radiation-induced adverse effects is generally called as single-event effects [11]. Most often, this type of faults is transient (i.e. soft errors) and do not cause permanent damage like TID, but they may still induce unwanted behaviour changes. All these transient effects stem from excess charge carriers generated through the ionization of silicon atoms by radiation. If a sufficient amount of these charges gathers in a metal line, the logic value of a line in that area can be upset; this event is referred to as a single-event transient (SET) which has a brief effect until the excess charge dissipates. In case of a SET, if a storage device captures the new state of the line, there would be a longer-lasting effect on the system output, which is identified as a single-event upset (SEU). Nevertheless, an SEU can be generally amended by restoring all flip-flop (FF) values through a system reset. However, it is not possible to fix some SEUs by a simple reset; these type of SEUs are called as single-event functional interrupts (SEFIs). There is another radiation effect called single-event latch-up (SEL) which takes place when ionizing radiation turns on parasitic transistors in the silicon. These parasitic transistors can keep conducting current until a system reset, causing parts of the device burn in some cases (i.e. hard error) [13].

B. N version programming

N-version programming was first introduced by adapting the concepts of hardware fault tolerance in a computer software. In [14] the authors use the following notation to discuss n-version programming. Multiple computations are implemented by N-fold (N $\geq$ 2) replications in three domains: time (repetition), space (hardware), and information (software). The nonfault-tolerant system is characterized by one execution (simplex time IT) of one program (simplex software IS) on one hardware channel (simplex hardware 1H) and is described by the notation: 1T/1H/1S. Later in this paper, we use a case study of an application which can be replicated in a distributed system on a parameterizable way, then using one execution (1T), multiple hardware (NH) and multiple software (NS). In this case, we adopt 1T/NH/NS, since we have spatially replicated hardware and software executing in parallel.

III. Platform and Framework

The approach in [15] proposed a framework which divided the system management into three different layers: device layer, application layer and run-time management layer. This is currently the state-of-the-art approach to apply run-time management since it decouples the management layer to be platform agnostic, facilitating the portability to different platforms or applications and thus supporting the reuse of software. The separation of the system into the three distinct layers — *application*, *run-time management* and *device* — shown in Fig. 1 reduces design complexity and provides flexibility during operation. The application layer comprises any number of software processes, while the device layer includes the hardware and its software drivers. The run-time management layer comprises a (Run-Time Manager) RTM responsible for the control and monitoring of the other two layers. This separation ensures portability and cross-compatibility; applications and device drivers only need to be written once to be used with any implemented RTM.

A. Knobs and monitors

Knobs and *monitors*, shown in the dashed regions of Fig. 1, facilitate communication between the layers. Knobs allow for the run-time tuning of application and device-specific parameters, while monitors enable the measurement of hardware properties and the observation of application behaviour. An RTM's primary objective is to ensure that the monitor values of all applications and the device remain within their specified bounds. Beyond this, it is free to optimise any unbounded monitors in order to meet secondary objectives, *e.g.* reducing power consumption. Minimal modification of applications is required to expose knobs and monitors through the framework. As an example application, an AES encryption process was shown in Fig. 1 which provides the option of selecting hardware or software execution for its operations at run-time. This choice will be controlled by an RTM using an application knob with options $\{0, 1\}$. If the same application requires a minimum throughput, *e.g.* expressed as an encryption rate, an application monitor with this bounds can be provided. In this case, the application will periodically update the current

encryption rate so that the RTM can keep it within the range $[\alpha, \infty)$. On the hardware side, Dynamic Voltage and Frequency Scaling (DVFS) of the CPU is achieved via a device knob with options $\{0, 1, \cdots, 9\}$, enabling the RTM to switch between ten distinct voltage-frequency pairs. Finally, to enable thermal management by the RTM, a temperature sensor is exposed as a device monitor.

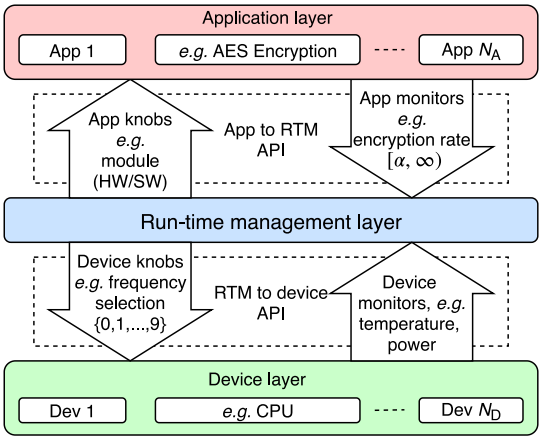

Fig. 1. Framework and API enabling communication between application, run-time management and device layers using knobs and monitors. Examples are given for an AES encryption application executing on a CPU. [15]

B. Local and remote knobs and monitors

This framework built with three layers copes very well with multi-core architectures where all the decisions are executed locally in the same cluster or Processing Element (PE) for a limited number of cores. These type of architectures have an intrinsic communication bottleneck between PEs and memory which does not provide scalability to comply with systems with more than a dozen of PEs. Larger architectures with dozens of general purpose PEs are composed of Networks-on-Chip (NoC), a technology that emerged as a possible solution to the aforementioned limiting factors.

For much larger systems, the framework would show limited results, since it does not target platforms where distributed processing elements with the exchange of network messages is required, i.e., a system which is composed of multiple clusters of multi-core architectures interconnected through a NoC [16] or even a system with multiple computers interconnected by an off-chip network.

Furthermore, all the decisions on the run-time management layer are taken locally, which does not allow a management decision to be taken with all the distributed information. Because of these drawbacks, an extension solution is proposed in Fig. 2 where a hierarchical run-time management layer is included.

The hierarchical RTM presents a subdivision between local and remote communications. Local communications are the ones executed locally only, between application, local RTM (LRTM) and the device layer. On the other hand, remote communications are executed between remote RTMs (RRTM). With the new paradigms of remote and local communications, knobs and monitors can also be categorised in these two types.

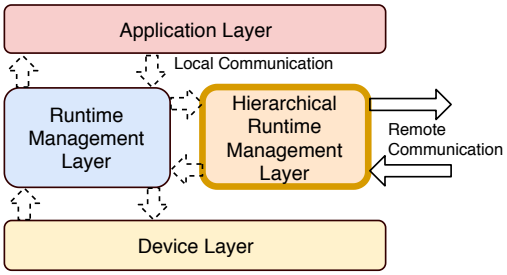

Fig. 2. Proposed communication between application, run-time management and device layers using knobs and monitors. The run-time management layer is divided with local and remote knobs and monitors management.

Remote knobs and monitors represent higher levels of management which should be handled in a remote fashion. Normally, remote knobs and monitors are related to the applications layer only, since device related knobs and monitors would be better managed locally, but there is no such restriction.

The novelty also allows a shift for self-awareness systems: the exposure of different knobs and monitors locally and remotely. Exposing monitors to a remote entity enables the implementation of multiple kinds of management: hierarchical management by a region of the platform, one manager per application, one manager per group of applications, etc. Further sections will discuss such management options.

C. Local and Remote Communications

Local communication is executed in the same device, requiring a thread to thread communication protocol. Fig. 3 shows two examples of how communication is employed between different layers. The first transaction (a) is a local knob request, executed between two threads, the run-time manager thread (RTM) and an example application (APP). First, the APP layer requests a given local knob and the RTM responds with its value. A similar example is shown in (b) where the RTM requests a local monitor and APP returns.

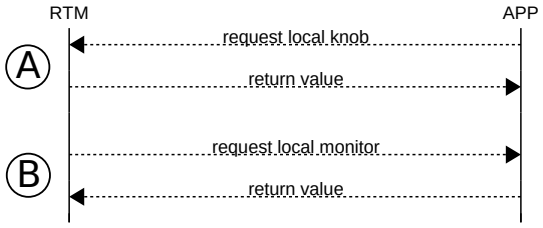

Fig. 3. Local communication example. Dashed lines represents local communications between threads.

All remote communications can also be seen as hierarchical *control* messages, passing through the LRTM, which is going to be processed and, if required, forwarded to the other nodes, specifically to the RRTM. Fig. 4 presents three examples of remote and local communications. In Fig. 4(a), we have a different setup when remote communication is used. The LRTM sends a message to the RRTM to register its execution and the RRTM acknowledges. These remote messages are sent through network sockets. In Fig. 4(b) we have an example of an LRTM updating a remote monitor value in the RRTM. First, the remote monitor is updated locally and later sent to

the RRTM. Note that the LRTM can decide the rate to update the manager with the monitor values, allowing a design space exploration, since it might be more reasonable to the LRTM to send monitor values less regularly that it reads locally. In Fig. 4(c) the manager RTM wants to set up a remote knob which is in a different node. First, it sends a network message to the RTM which then forwards it locally to the application.

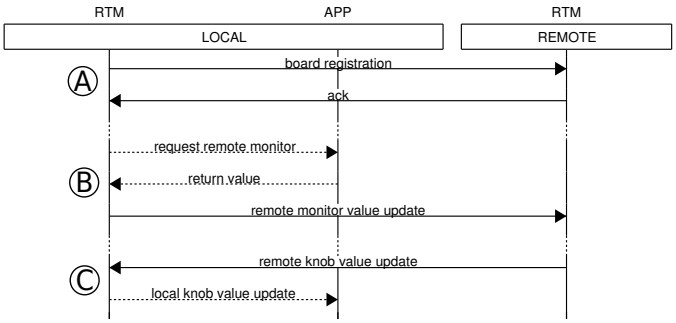

Fig. 4. Remote communication protocol example. Dashed lines represents local communications between threads while solid lines represents remote communications employed with sockets.

D. Manager and Worker Nodes

The ability to control local knobs remotely allows the use of different management layouts. The decision can be taken remotely, e.g. one distributed application can have local and remote knobs and monitors. Locally, the RTM can decide which frequency to set while an application related knob can be controlled remotely by another RRTM. This decentralization process raises another question regarding which entity needs to be responsible for the remote knobs and monitors. The easiest way to tackle it is to visualise a scenario with one manager and multiple workers, where each one of these entities (manager or worker) is a member of a cluster and/or boards/computers interconnected by a NoC or an off-chip network. The manager is responsible for handling remote knobs and reading remote monitors.

In this paper, we present the setup where a central manager is responsible for allocating and controlling tasks, but it is important to emphasize that different scenarios with multiple managers or even a hierarchy between them are possible. Even in scenarios with a central manager a protocol [17] can be used to determine when it has become faulty, and migrate the management software safely to another worker.

IV. Case Study

This section shows an example application of the proposed framework using an established technique to provide fault tolerance. The objective is to show how the characteristics of the framework could be employed in a real case scenario.

The proposal comprises a management approach which can be employed in different systems: future many-core systems with multi-core PE clusters interconnected by NoCs or even distributed approaches with multiple boards interconnected via an off-chip network. The main advantage of such hierarchical approach is that it can adapt to different scenarios, with remote knobs and monitors being handled by a Manager node, for example, while local knobs and monitors are handled locally.

Fig. 5 presents an implementation of an N-version programming example with a majority voter system where N is equal to three. In that case, a, b and c represents a given application which has been replicated three times. Source sends the same data to tasks a, b and c, which are replicated and executing the same task code and forwarding the computed output to the voter. The voter compares the input data from tasks and if at least two of them matches, acknowledges its reception and tasks can send another round of data. If none of the inputs matches, a re-run message is sent to re-execute the last round. In case of a mismatched data from one task, the voter must report one mismatched value to the manager, via a remote monitor. This is an important metric since it allows the manager to observe the system fault rate and decide if a node needs to be replaced. The number of tasks can be communicated as a remote knob. This operation would require idle nodes running the task's code to have its remote knobs updated at run-time. To do so, the application has the knobs and monitors listed in Table I. The manager node S is responsible for controlling the number of replicated tasks and voters in the system.

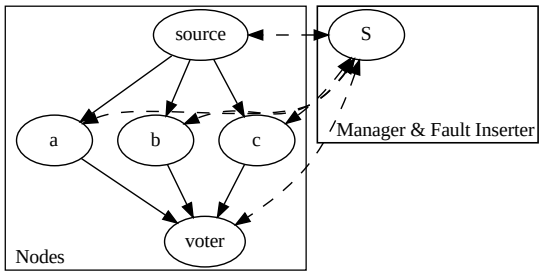

Fig. 5. Example voter application. Manager S can control knobs and read monitors from all the nodes (dashed lines).

TABLE I
Remote and local knobs utilised in example voter application.

node	type	function
source	remote knob	number of task's
source	remote knob	task's addresses
task	remote knob	number of voters
task	remote knob	voter's addresses
voter	remote knob	number of tasks
voter	remote monitor	number mismatches
all	local knob	core frequency

This application target aims to implement a *distributed* and *parameterizable* form of the N-version programming using local and remote knobs and monitors. In this case, the manager node is responsible for tuning the remote knobs observing the remote monitors. Fig. 6 presents an example of how the system's protocol is employed during the insertion and removing of a node from the system. When the system is initialised all the LRTMs should register with the manager RTM as shown in process a. As nodes are being included in the system, the manager needs to update remote knobs in each node (process b). After that, the application can initialize and start sending messages to the voter as in process c; when the voter receives messages from all tasks, it acknowledges each

and every task (see process d) and the cycle restarts. Note that these are data messages, being sent from application to application and differ from control messages. In case one of the tasks becomes faulty, the voter communicates with the LRTM to deregister this task as shown in process e. The LRTM then signalizes that this particular task is faulty to the manager RTM, see process f, which in return updates the remote knobs in the voter and sends a signal to reconfigure the node of the responsible task (refer to process g) – this process will be described later. When the corresponding node is reconfigured, its process should restart with the registration (as in process h).

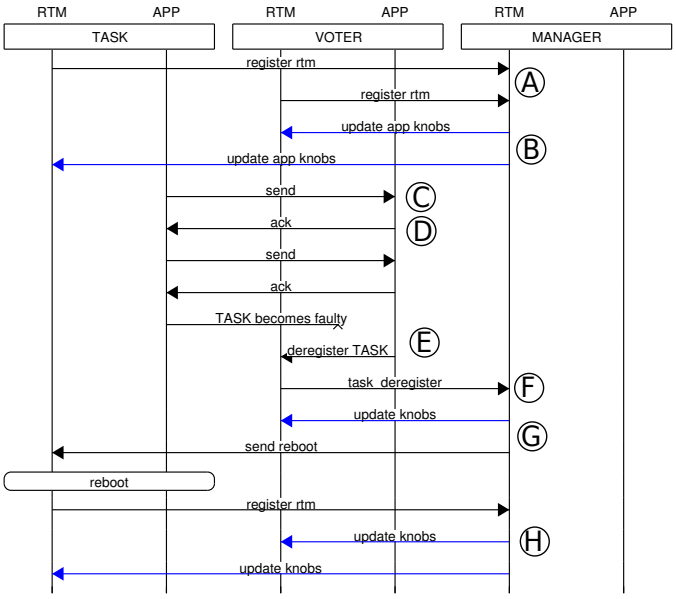

Fig. 6. Illustration of the proposed management protocol. Blue lines represent control messages for remote knobs.

The nodes can be considered faulty in two scenarios: (i) if a node does not send messages within a certain time frame; (ii) if the node's computed output does not match with the final output of the majority voter. In both cases, the voter can employ techniques to detect such events, i.e. a watchdog timer for the first case and simple comparison for the second case. In the meanwhile, since the error rate of each node is delivered to the manager regularly, it can take a preemptive decision to remove the node before it becomes completely unresponsive.

In our context, we can categorize faults into two types: transient and permanent faults. If the number of faults happening in one node is below a given threshold within a certain time frame, we categorize it as transient faults for that particular node. If it exceeds the threshold, we consider it as a permanent fault which means that the board is inoperative.

In case of a transient fault, the reconfiguration process, illustrated in process g, refers to an actual reboot of the node. In case of a permanent fault, since rebooting would not solve the issue, a healthy node has to replace the faulty one within the system. From the voter's point of view, it does not matter which node is fulfilling the task's computation.

V. Results

To evaluate a distributed system with multiple clusters of CPUs requires an SoC with a high density of transistors. Unfortunately, today's SoCs are targetting systems, e.g. NVIDIA GPUs or Intel's Xeon Phi Knight Landing architecture, which only focus on specific applications. For this reason, we try to emulate future generation general purpose systems using many off-the-shelf boards. During our experiments, we employ affordable boards that can mimic future in-chip clusters. This paper carries out an evaluation on multiple heterogeneous multi-core platform, i.e. Odroid-XU4, interconnected by an off-the-shelf network switch. Therefore, each Odroid-XU4 board acts like a cluster, where one instance of the hierarchical framework proposed in Fig. 2 is running.

The Odroid-XU4 is composed of the Samsung Exynos 5422 SoC. It contains four ARM Cortex-A15 (big) CPUs, four ARM Cortex-A7 (LITTLE) CPUs. Such an architecture provides opportunities to exploit different designs as low power processing (LITTLE cores) and high-performance processing (big cores). For the Cortex-A15, the frequency can be varied between 200 MHz and 2000MHz with a 100 MHz step, whereas for the Cortex-A7, it can be varied between 200 MHz and 1400 MHz with a step of 100 MHz.

A. Overhead

The objective is to measure the impact of the proposed N-version programming voter compared to a baseline where there are no replicated tasks. To do so, we create a baseline scenario where no voter is employed and measured the average time that each AES encryption algorithm takes to compute and send the result in 50 runs. Then, we repeated this scenario 50 times. The first graph in Fig. 7 shows the average of each run, taking on most cases around 100 ms. Later, we executed the same measurements for scenarios with 2, 3 and 5 replicated tasks sending messages to the voter. Results showed an expected increase since there is a time overhead to deliver multiple messages and also the voter's computation time. The scenario with two tasks shown an increase of 10 to 30% while scenarios with 3 and 5 tasks shown an increase of 30%.

To evaluate the voter computation overhead, we measured the time it took for the voter to compute and respond since it received the last message. Fig. 8 shows these results, with voters taking on average 20 μs with a small increase in scenarios with 3 and 5 replicated tasks.

B. Fault Insertion Campaign

If desired, the manager can also insert faults in any of the boards to test the correct execution of the proposal. Faults can be inserted at a board level by a thread in the manager RTM. The manager sends a message to LRTM forcing the board to reboot. At current status the grain of fault is a complete board, i.e. a message sent to a board is going to deregister the complete board from the system as explained before. Nevertheless, the system can be modified to a fine grain, per processing core, for example. One can decide the time and a given board to be affected by a fault. During our experiments, faults are being inserted at random boards at random times.

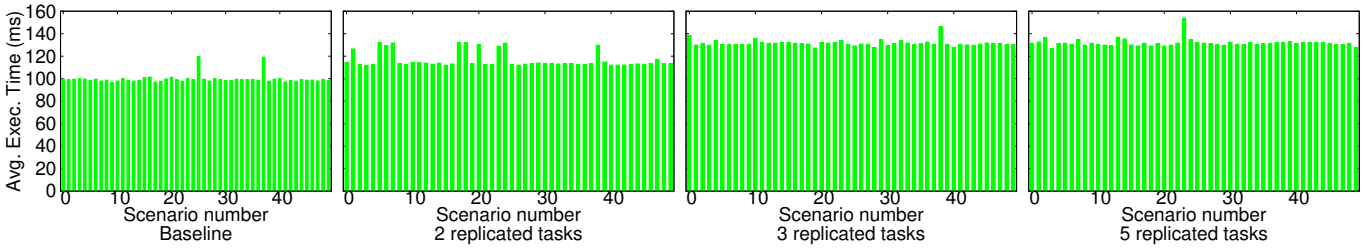

Fig. 7. Overhead scenarios comparing average computation time with baseline design.

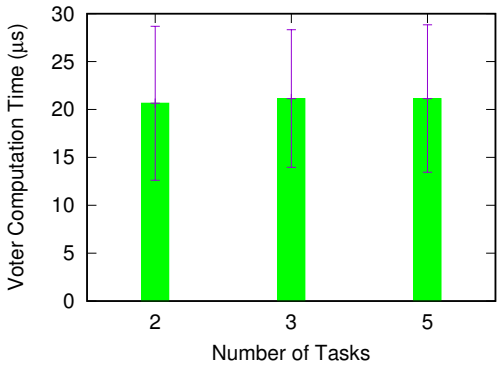

Fig. 8. Voter computation overhead. Boxes represents average computation time and lines the standard deviation.

To stress and test the system, a fault insertion campaign was developed. The same previous scenarios with 3 and 5 tasks were exposed to fault injection. Each scenario executed the AES application for one minute, with one or two faults being inserted at random times at random boards. Then, these processes were repeated one hundred times for each case. All scenarios were capable of rebooting boards and continue its execution after reboot, as illustrated in Fig. 6. In our case, as we are employing a fault injection mechanism which is of a very high level, we cannot mimic a scenario where a fault would go effectless or would cause a system failure. In any case, one fault would cause silent data corruption, which would be detected and masked by the three-input majority voter. However, if an increase in the fault rate is detected, then instead of three, the system should adapt itself by increasing the number of replicated tasks to be able to mask two faults, for example. The faults insertion campaign is automated using a shell script to connect via ssh to the boards, start its execution and collect its results when it is finished.

VI. Conclusions

This paper proposed a framework to provide ways for different fault-tolerant techniques to be employed in a distributed system. A case study was presented using a scenario where a parameterizable number of tasks were replicated and its outputs compared, providing an example implementation of a majority voter. Results have shown that produced overheads were very small. Additionally, fault insertion scheme can be extended to a finer grain, where bit-flip can be inserted at run-time to test soft-error mitigation techniques.

Acknowledgment

This work is supported by the UK Engineering and Physical Sciences Research Council through grants EP/R02572X/1 and EP/P017487/1.

References

[1] "NDA," https://www.gov.uk/government/publications/nuclear-provision-explaining-the-cost-of-cleaning-up-britains-nuclear-legacy/nuclear-provision-explaining-the-cost-of-cleaning-up-britains-nuclear-legacy.

[2] "Fukushima Daiichi nuclear power plant accident," https://techcrunch.com/2017/03/25/japanese-authorities-decry-ongoing-robot-failures-at-fukushima/.

[3] A. Lindoso, L. Entrena, M. García-Valderas, and L. Parra, "A hybrid fault-tolerant leon3 soft core processor implemented in low-end sram fpga," *IEEE Transactions on Nuclear Science*, vol. 64, no. 1, pp. 374–381, Jan 2017.

[4] P. Bernardi, L. Bolzani Poehls, M. Grosso, and M. Sonza Reorda, "A hybrid approach for detection and correction of transient faults in socs," *IEEE Transactions on Dependable and Secure Computing*, vol. 7, no. 4, pp. 439–445, Oct 2010.

[5] J. R. Azambuja, S. N. Pagliarini, M. Altieri, F. G. de Lima Kastensmidt, M. K. Hubner, J. Becker, G. Foucard, and R. Velazco, "A fault tolerant approach to detect transient faults in microprocessors based on a non-intrusive reconfigurable hardware," *IEEE Transactions on Nuclear Science*, vol. 59, no. 4, pp. 1117–1124, 2012.

[6] F. Ferlini, F. A. da Silva, E. A. Bezerra, and D. V. Lettnin, "Non-intrusive fault tolerance in soft processors through circuit duplication," in *2012 13th Latin American Test Workshop (LATW)*, April 2012, pp. 1–6.

[7] A. M. Keller and M. J. Wirthlin, "Benefits of complementary seu mitigation for the leon3 soft processor on sram-based fpgas," *IEEE Transactions on Nuclear Science*, vol. 64, no. 1, pp. 519–528, Jan 2017.

[8] D. M. Fleetwood, "Total ionizing dose effects in mos and low-dose-rate-sensitive linear-bipolar devices," *IEEE Transactions on Nuclear Science*, vol. 60, no. 3, pp. 1706–1730, June 2013.

[9] F. Faccio and G. Cervelli, "Radiation-induced edge effects in deep submicron cmos transistors," *IEEE Transactions on Nuclear Science*, vol. 52, no. 6, pp. 2413–2420, Dec 2005.

[10] A. Jantsch, N. Dutt, and A. M. Rahmani, "Self-awareness in systems on chip— a survey," *IEEE Design Test*, vol. 34, no. 6, Dec 2017.

[11] D. K. Pradhan, Ed., *Fault-tolerant Computer System Design*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1996.

[12] T. Nidhin, A. Bhattacharyya, R. Behera, T. Jayanthi, and K. Velusamy, "Understanding radiation effects in sram-based fpgas for implementing instrumentation and control systems of nuclear power plants," *Nuclear Engineering and Technology*, vol. 49, no. 8, pp. 1589–599, 2017.

[13] M. Wirthlin, "High-reliability fpga-based systems: Space, high-energy physics, and beyond," *Proceedings of the IEEE*, vol. 103, no. 3, pp. 379–389, March 2015.

[14] A. Avizienis, "The n-version approach to fault-tolerant software," *IEEE Transactions on Software Engineering*, vol. SE-11, no. 12, 1985.

[15] G. M. Bragg, C. Leech, D. Balsamo, J. J. Davis, E. Wachter, G. V. Merrett, G. A. Constantinides, and B. M. Al-Hashimi, "An application- and platform-agnostic runtime management framework for multicore systems," in *PECCS*, 2018, pp. 57–66.

[16] A. Karkar, T. Mak, K. Tong, and A. Yakovlev, "A survey of emerging interconnects for on-chip efficient multicast and broadcast in many-cores," *IEEE Circuits and Systems Magazine*, vol. 16, no. 1, 2016.

[17] V. Fochi, L. L. Caimi, M. Ruaro, E. Wächter, and F. G. Moraes, "System management recovery protocol for mpsocs," in *2017 30th IEEE International System-on-Chip Conference (SOCC)*, 2017, pp. 367–374.