applied sciences

MDPI

Article

NMR-MPar: A Fault-Tolerance Approach for Multi-Core and Many-Core Processors

Vanessa Vargas [1,2,*,†], Pablo Ramos [1,2], Jean-Francois Méhaut [3] and Raoul Velazco [2,4]

1 Universidad de las Fuerzas Armadas ESPE, DEEE, Avenida General Rumiñahui S/N, 171-5-231B Sangolqui, Ecuador; pframos@espe.edu.ec
2 TIMA Labs., Université Grenoble-Alpes, Avenue Félix Viallet, 38000 Grenoble, France; raoul.velazco@univ-grenoble-alpes.fr
3 LIG Labs., Université Grenoble-Alpes, 3 Parvis Louis Néel, 38054 Grenoble, France; jean-francois.mehaut@univ-grenoble-alpes.fr
4 National Center for Scientific Research (CNRS), 38000 Grenoble, France
* Correspondence: vcvargas@espe.edu.ec; Tel.: +593-2-398-9400
† Current address: Universidad de las Fuerzas Armadas ESPE, Av. General Rumiñahui s/n, 171-5-231B, Sangolqui, Ecuador, P.O. BOX:171-5-231B.

Received: 14 February 2018; Accepted: 12 March 2018; Published: 17 March 2018

Featured Application: The N-Modular Redundancy and M-Partitions (NMR-MPar) fault-tolerance approach can be used to improve the reliability of embedded systems that are based on Commercial-Off-The-Shelf (COTS) multi-/many-core processors, especially those with a non-critical time constraint allowing a diminution in performance to gain reliability by maintaining power consumption. Typically, mixed-criticality systems, such as avionics or spacecraft, comply with these requirements.

Abstract: Multi-core and many-core processors are a promising solution to achieve high performance by maintaining a lower power consumption. However, the degree of miniaturization makes them more sensitive to soft-errors. To improve the system reliability, this work proposes a fault-tolerance approach based on redundancy and partitioning principles called N-Modular Redundancy and M-Partitions (NMR-MPar). By combining both principles, this approach allows multi-/many-core processors to perform critical functions in mixed-criticality systems. Benefiting from the capabilities of these devices, NMR-MPar creates different partitions that perform independent functions. For critical functions, it is proposed that N partitions with the same configuration participate of an N-modular redundancy system. In order to validate the approach, a case study is implemented on the KALRAY Multi-Purpose Processing Array (MPPA)-256 many-core processor running two parallel benchmark applications. The traveling salesman problem and matrix multiplication applications were selected to test different device's resources. The effectiveness of NMR-MPar is assessed by software-implemented fault-injection. For evaluation purposes, it is considered that the system is intended to be used in avionics. Results show the improvement of the application reliability by two orders of magnitude when implementing NMR-MPar on the system. Finally, this work opens the possibility to use massive parallelism for dependable applications in embedded systems.

Keywords: fault tolerance; many-core; multi-core; partitioning; redundancy; reliability; fault injection

1. Introduction

The widespread use of COTS multi-core and many-core processors in most modern computing systems is a consequence of their flexibility, high performance, low-power consumption and intrinsic redundancy. From High Performance Computing (HPC) to embedded systems, these

devices are preferred thanks to the massive parallelism that can be achieved and their inherent redundancy capabilities that allow improving the system reliability. For instance, in HPC systems, the first Supercomputers Top500 November 2017 list are based on these devices: the 1st Top500: Sunway TaihuLight-Sunway MPP, based on the Sunway many-core processor SW26010 (260 cores), and 2nd Top500: Thiane-2, based on the multi-core processor Intel Xeon E5-2692 (12 cores) [1].

Despite the advantages provided by multi-/many-core processors, there are some dependability issues that have to be overcome for use in implementing critical tasks. Indeed, the reliability, availability and security of systems based on these devices are reduced due to the increasing complexity and integration scale. Consequently, manufacturers, industrial and academic partners are working together to improve their capabilities to allow their usage in critical systems. There are some international research projects involved on this behalf such as Advanced Cockpit for Reduction Of StreSs and workload (ACROSS) [2] and Multi-core Partitioning for Trusted Embedded Systems (MultiPARTES) [3], which search to evaluate and improve the fault-tolerance of a system based on multi-/many-core processors. Furthermore, avionics and spacecraft industries are interested in validating the use of these components for their applications (e.g. international teams are working on this purpose: Certified Associate in Software Testing (CAST), European Aviation Safety Agency (EASA) and Reduced certification Costs for trusted Multi-core Platforms (RECOMP) projects).

One of the main concerns about multi-/many-core reliability is the increase of its sensitivity to the effects of natural radiation caused by shrinking the transistor size. This sensitivity is closely related to the chip complexity and the huge amount of on-chip memory cells. For this reason, designers are continuously searching for new methods to improve manufacturing technologies to reduce Single Event Effect (SEE) consequences. Additionally, manufacturers implement protection mechanisms such as parity and Error-Correcting Code (ECC) in devices' memories. Besides, some mitigation techniques based on hardware and software approaches are proposed in the literature to minimize the consequences of this phenomenon.

In spite of the manufacturing efforts, there are some areas that remain vulnerable to the effects of natural radiation. A few interesting works dealing with the sensitivity of multi-core and many-core processors can be found in the literature [4–8]. Results showed that complementary mitigation techniques are required for the widespread use of COTS processors in safety-critical domains such as avionics. Regarding these techniques, software fault-tolerance for parallel and distributed systems has been largely studied. Nonetheless, studies concerning systems-on-chip's fault tolerance are limited.

The work in [9] exploits several redundancy techniques to improve the reliability in multi-core processors. The authors propose the use of redundancy in all the stages involving the device, from its design to the the execution of applications. Traditionally, the cost of implementing redundant systems has been significant. However in multi-core and many-core processors, various approaches based on redundancy techniques can be considered due to the multiplicity of cores.

In this context, the purpose of this work is to propose an approach based on N-Modular Redundancy (NMR) and partitioning, to improve the reliability of parallel applications running on multi-/many-core processors. To validate the proposal, a case study running two parallel benchmark applications using complementary on-chip resources was implemented. The effectiveness of this approach is evaluated by a Software-Implemented Fault Injection (SWIFI) technique. Results are extrapolated to avionic altitudes to appraise the reliability improvement. The MPPA-256 many-core processor was chosen as the target device because of its similarity to the Sunway SW26010 (Base of the First Top500) and due to the interest of the French industry to use it in the avionics domain (CAPACITES project).

The remainder of the paper is organized as follows: Section 2 presents the related work. Section 3 describes the N-Modular Redundancy and M-Partitions (NMR-MPar) approach and details the case study. Experimental results are presented and analyzed in Section 4. Section 5 discusses the results. In Section 6, the materials and methods used in this work are detailed. Finally, Section 7 concludes the paper.

2. Related Work

In the literature there are significant works dealing with fault-tolerance with software redundancy and with partitioning applied to multi-/many-core processors. However, only a few works evaluated them through simulation or fault-injection.

2.1. Fault-Tolerance by Software Redundancy

Regarding fault-tolerance by software redundancy, it is common to use redundant multithreading techniques for transient fault detection and recovery of multi-core processors working in Symmetric Multi-Processing (SMP) mode [10–14]. Some of them are based on Simultaneous Multi-Threading (SMT) where the systems run identical copies of a process as independent threads. However, in these works, the details about the validation of the proposed techniques are not provided. Moreover, the principle of redundancy in multi-core processors is used to recover the system from permanent hardware faults [15].

In [16] is presented a software technique for transient-fault tolerance: the Process-Level Redundancy (PLR), which leverages multiple cores for low overhead. This approach is evaluated by Single Event Upset (SEU) fault-injection on redundant processes. The faults are injected by means of the Intel tool called Intel Pin dynamic binary instrumentation, which changes one bit of a randomly-selected instruction. Nonetheless, there is no complete independence between processes since this approach uses a unique Operating System (OS).

The work in [17] proposes a reliable resource management layer applied to many-core architectures. The scheduler applies a Triple Modular Redundancy (TMR) on multi-threading applications, detects errors and isolates faulty processing elements. This approach is evaluated at the simulator level by injecting one fault per run. The evaluation was done on a single cluster of 16 cores.

In [18] is proposed a framework to improve the reliability of COTS many-core processors. The framework is illustrated through an adaptive NMR system that selects the better number of replicas to maximize the system reliability. To achieve this, they present an analysis of the reliability of the system that shows a trade-off between the reliability of the voter and the number of replicas. This analysis is only theoretical and has not been experimentally evaluated.

The work in [19] proposes a soft NMR that improves the robustness of classical NMR systems by using error statistics. Its effectiveness is illustrated by an example in image coding.

Lastly, the authors of [20] implement four independent fault-tolerance techniques on the Radiation Hardened By Design (RHBD) Maestro processor aiming at mitigating hardware and software errors. This work describes the implementation of these techniques, which include a kernel-level checkpoint rollback, a process and thread level redundancy and a heartbeat. Nonetheless, there is no evaluation of the effectiveness of their proposal.

2.2. Fault-Tolerance by Partitioning

Concerning fault-tolerance by partitioning, [21] recaps the main challenges of the migration from federated systems to Integrated Modular Avionics (IMA) in avionics and gives some guidance in temporal partitioning. The work in [22] proposes temporal isolation between tasks to schedule mixed-criticality works on multi-core processors.

The authors of [23] introduce the concept of parallel Software Partition (pSWP) supported on a Guaranteed Resource Partition (GRP) applied to many-cores. Their proposal consists of defining a set of physical resources where the pSWP runs in order to guarantee the time isolation for IMA systems. The proposal is evaluated in a simulator compatible with PowerPC ISA.

The work presented in [24] implements multiple partitions with rigorous temporal and spatial isolation. This approach supports mixed-criticality based on the XtratumM, an hypervisor used in real-time embedded systems as virtualization layer. The methodology and tools are described.

The work in [25] is probably the most related work to the NMR-MPar approach proposed in this work, which uses both principles of partitioning and redundancy. The authors propose a hybridization between duplex execution at the task level on COTS and time and space partitioning provided by the bare-board hypervisor Xtratum. The authors have implemented their proposal on a ZYNQ a programmable System-on-CHip (SoC) that hosts an ARM dual-core Cortex A9 processor. The main differences with this work are that in NMR-MPar: (a) the redundancy proposed is totally independent at the system and user level; (b) there is an independent hypervisor for each partition; (c) the implementation was done on a many-core processor with applications running on 280 cores; and (d) the proposal was evaluated by fault-injection.

3. N-Modular Redundancy and M-Partitions Approach

Benefiting from the inherent redundant capabilities of multi-core and many-core processors, the NMR-MPar approach combines the partitioning concept with the modular redundancy technique to improve the reliability of the system.

3.1. Background

3.1.1. Redundancy

This allows a system to continue working properly in case one or more components fail. This is possible due to the fact that parallel components added to the system provide an alternative path to operate. There are several approaches that improve the reliability by the replication of contents of processor components during the execution of an application without modifying the hardware: cache citeKim1999,Zhang2003,Zhang2005,Sugihara2007, instruction [30] and register contents [31,32]. In contrast, other redundancy techniques modify the hardware. Among them, the TMR [33] is a well-known fault-tolerant technique that implements triplication of modules and majority voting. This technique has been largely used to avoid errors in integrated circuits, but imposes very high hardware overhead. Nevertheless, this method becomes more attractive when implemented on multi-/many-core processor architectures.

In fact, an N-Modular Redundancy system can be considered. An NMR consists of N identical parallel systems with the same input. It also includes a voter to obtain the system output. The voter compares the outputs of each parallel systems and applies majority criteria to select the response. In general, N is an odd integer to simplify the voting operation. However, it is possible to use an even integer depending on the characteristics of the system [34]. It is clear that the reliability of an N-modular redundancy systems depends on the voter [35]. Therefore, considering the potential errors occurring in the voter, it is also possible to implement redundant voters. The most common implementations of this kind of system are the Double Modular Redundancy (DMR), which allows error detection, and the TMR, which masks faults by detecting and correcting errors.

Regarding the nature of the replication, there are four types of redundancy: spatial and temporal in data and in execution. This work uses:

- Spatial redundancy: uses different physical components. It can separate identical data signals in space.
- Redundancy in data: replicates the information. It stores the data in different memory spaces.

3.1.2. Partitioning

Mixed-criticality systems use partitioning to increase the reliability of embedded systems [24]. To achieve fault tolerance by partitioning, the behavior of one partition must not be affected by other partitions. Hence, time and spatial partitioning must be guaranteed.

- Time partitioning: The use of resources is exclusively assigned to a partition during a scheduled period of time. In a general manner, this is associated with the Central Process Unit (CPU) time assigned with each partition.
- Space partitioning: A function cannot change the code or data of another partition. This is associated with the memory space assigned to each partition.

Robust partitioning is a traditional concept used by Federal Aviation Administration (FAA) and EASA to certify safety-critical avionic applications. It is defined in the standard RTCA/DO -297 [36]. Space partitioning can be easily carried out by multi-core and many-core processors since the system can prevent invalid memory accesses while achieving spatial partitioning in such devices, which is not a trivial issue. The work in [21] gives some directions concerning temporal partitioning on multi-core processors used for avionics.

3.2. Description of the N-Modular Redundancy and M-Partitions Approach

The partitioning principle is used to create different partitions that co-exist in the same device. This is typically done by the hypervisor, which provides virtual CPUs to each partition. In contrast, the NMR-MPar approach proposes a physical resource distribution to each partition, to minimize the propagation of faults producing dysfunctions in other resources or cores. Each partition can be setup as a mono- or multi-core running on different multiprocessing modes: Asymmetric Multi-Processing (AMP) or Symmetric Multi-Processing (SMP) mode with different programming models: bare-metal, OpenMP, Portable Operating System Interface uniX (POSIX) , etc. Consequently, there is a considerable versatility for the system configuration that is enhanced by the number of cores comprising the device. It is important to note that each partition can be configured in bare-metal or with an independent OS. In bare-metal, no OS is used, then the programmer uses the functions provided by the manufacturer to access hardware resources. There is no abstraction layer from the hardware architecture.

Furthermore, it is proposed for critical functions that N partitions with the same configuration participate in an N-Modular redundancy system. Thus, several partitions execute the same application, and the results are used by a voter to build a fault-tolerant system. The voter system can include one or more cores of the device, or an external one if desired. Depending on the partition of the device, several N-modular redundancy systems may run on it concurrently.

Figure 1 illustrates an example of this approach implemented on a multi-core having 16 processor cores. The example establishes seven partitions ($M = 7$). Partitions P0–P2 are quad-core, while P3–P6 are mono-core. The quad-core partitions are part of a TMR system. Each one of them runs in SMP mode executing the parallel Application 1. On the other side, mono-core partitions (P3–P5) are part of another TMR running Application 2. The last mono-core partition P6 is the voter of both TMRs. The latter is configured in bare-metal to reduce the use of resources, minimizing the impact of faults.

By using this approach, temporal and spatial isolation of the applications is guaranteed by this type of partitioning. The security of each partition against unauthorized access and data modification can be obtained by the configuration of the system.

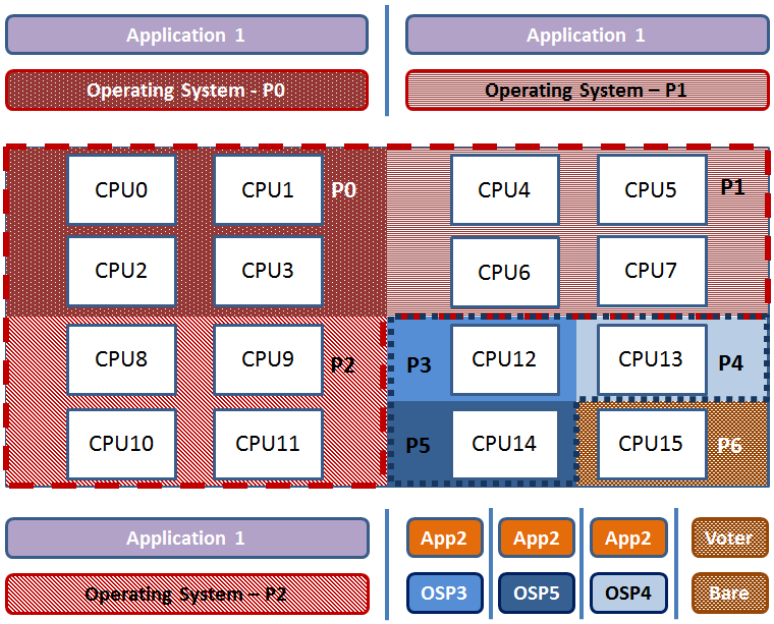

Figure 1. The N-Modular Redundancy and M-Partitions approach.

3.3. Case-Study: NMR-MPar Implemented on the MPPA-256 Processor

This case-study implements the NMR-MPar approach to improve the reliability on parallel applications running on the MPPA-256 many-core processor. The latter is a clusterized device that implements two Input Output cluster (IOs) for external communication and minimal processing and sixteen Compute Clusters (CCs) exclusively for processing. The IO cluster is comprised by eight cores called Resource Managers (RMs), while each CC consists of one RM in charge of managing the resources and sixteen Processing Engines (PEs) cores for computing. Both types of clusters include a private Static Memory (SMEM). The intra-cluster communication is achieved by buses, while the inter-cluster communication is performed by using two Network-on-Chips (NoCs).

The great configuration flexibility of the MPPA allows running independent applications per cluster or programming multi-cluster applications in a classic master/slave scheme, where the IO cluster performs as the master. For inter-cluster communication, the manufacturer provides a library with a set of functions for data exchanges through the MPPA NoC. Since, the MPPA is a coprocessor, a CPU host (HOST) is needed to manage it. The communication between the HOST and the MPPA is achieved using specific drivers provided by the manufacturer.

Two types of distributed applications were evaluated: CPU-bound and memory bound, the Traveling Salesman Problem (TSP) and the Matrix Multiplication (MM), respectively. The importance of selecting two different natures of application lies in testing different complementary on-chip resources, which allows verifying the effectiveness of the approach under different scenarios. Additional details of the target device and both applications are provided in Section 6.

3.3.1. System Configuration

The many-core processor was configured in a typical master/slave scheme to run parallel multi-cluster applications. The master runs on the IO cluster, while the slaves run on the CCs. In this case study, the application was configured without a HOST part.

The present system configuration takes advantage of the intrinsic architecture of the device, which allows dividing the chip into two global independent hardware parts. Each part is used in a master/slave scheme. In this case, a symmetric division is proposed, so that each IO cluster manages half of the cores, and thus, eight CCs are the slaves of each IO cluster. In order to maximize

the parallelism used and due to the fact that the minimal possible processing partition is a CC, this case-study implements two redundant modules per IO cluster, each one composed of four CC. Therefore, the system is configured as a quadruple modular redundancy system with the particularity that if one IO cluster has a dysfunction, the other is able to continue working as a DMR. In addition, this division guarantees the spatial and temporal isolation for critical systems.

The system is partitioned into eight partitions. Four are multi-core partitions, and the other four are mono-core partitions. Each multi-core partition P0–P3 runs independently one instance of the proposed quadruple modular redundancy. A multi-core partition is composed of one RM of the IO cluster and four CC running in AMP mode. The RM of the IO cluster performs as master, and it is configured in bare-board, while the CCs are the slaves that run a NodeOS with POSIX. Regarding the mono-core partitions, each one runs on one RM core of the IO clusters, and it is configured in bare-board. Figure 2 illustrates this configuration, which implements the NMR-MPar approach, being $N = 4$ and $M = 8$. In this scheme, four RM cores of each IO cluster were used:

- The RM0 coordinates the actions between the application and the fault injector. Furthermore, it is in charge of configuring all the inter-cluster communication.
- The RM1 and RM2 are the masters of each instance that runs the application.
- The RM3 is the voter.
- Additionally, the RM3 of IO Cluster 0 is also the fault-injector of the system.

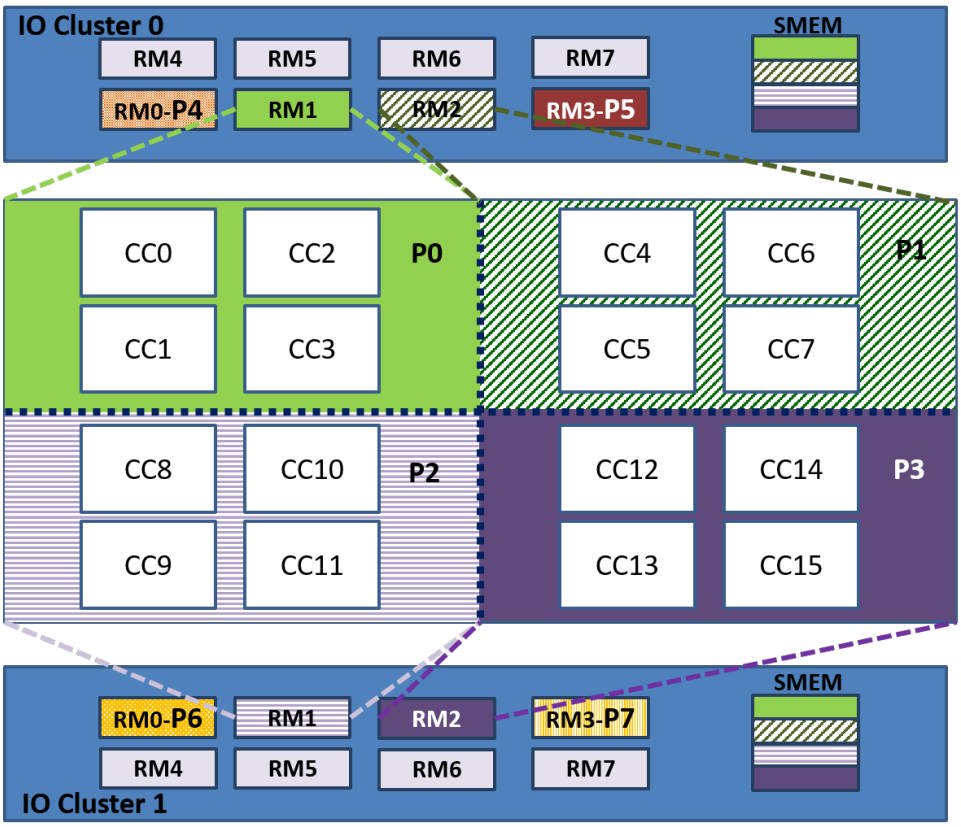

Figure 2. Proposed configuration for the case-study. CCx: Compute Cluster x, being x 0 to 15. RMy: Resource Managery, being y 0 to 7. PM: Partition M, being M 0 to 7. SMEM: Static Memory.

This implementation considers that each instance is comprised of four CCs as the slaves. This means that five RMs and 64 PEs cores work together. The RMs run the kernel services, schedule the tasks inside each cluster and manage the inter-cluster communication through the NoC services.

All the PE cores included in the slaves (CCs) are used to compute the result. The details of the configuration of both applications are presented in Appendix B.

This configuration allows having redundancy in data results. Therefore, when each partition has solved the problem, it sends its results to both IO clusters. In this way, the results are saved in two different physical locations, as shown in Figure 3.

Before voting, the results of each instance are validated. The results of TSP are validated by verifying a valid and complete path, while the MM results are validated by applying a Cyclic Redundancy Check (CRC) method. Once local data are validated, each voter applies majority voting only with valid data to obtain the correct result based on the following criteria: the correct response can be determined if three or four results are equal. Furthermore, it is possible to establish the correct response if two results are equal and the other two are different between them. However, if there are two pairs of two equal results, the voter logs an error. It is important to note that the result includes the response of the application plus the validation data of the result. The correct result is sent to the other IO cluster. The voter that completes the operation first logs the correct results to the HOST.

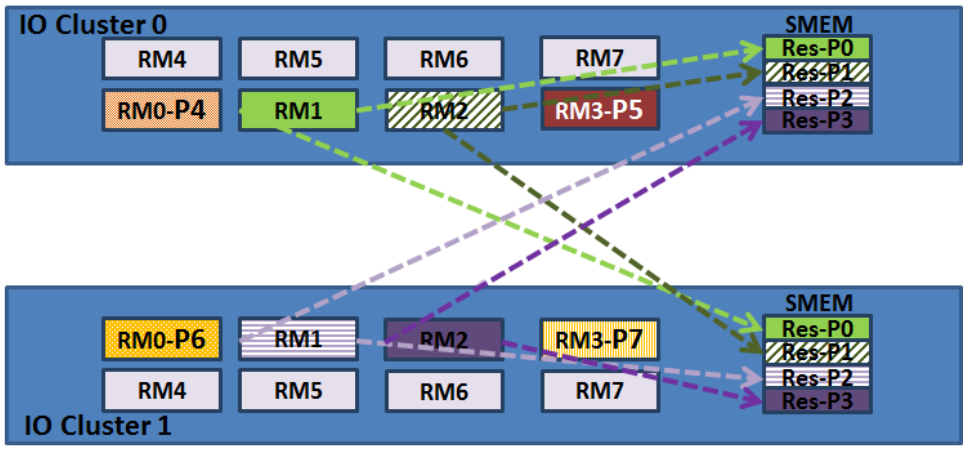

Figure 3. Proposed configuration for case-study (Part 2).

4. Results

The validation of the proposed approach was done by SWIFI, which is a useful technique to emulate SEU effects. The adopted approach was proven in our previous works concerning fault-injection for multi-core processors [7,37–39]. It is important to note that the objective of this approach is to reproduce the SEUs effects, which are bit-flips in memory cells caused by natural radiation. Details of the implementation of the fault-injector are described in Section 6.3.

This case-study implements one core of the device as the fault-injector, the RM3 core being of IO Cluster 0. Fault-injection campaigns for both applications are devoted to inject faults in the General Purpose Registers (GPRs) and 15 System Function Registers (SFRs) of the PEs belonging to the compute clusters. The fault-injection campaigns consider the emulation of one SEU per execution. The authors decided to run fault-injection campaigns only in accessible processors' registers due to the following:

In the authors' previous work [8] was presented an SEU error-rate prediction method that combines radiation experiments with fault-injection. The method was evaluated on the same target device (MPPA-256) implementing a parallel matrix multiplication running on bare-metal without OS. From the radiation experiment results, it was observed that all upset events produced in the internal memories (SMEMs) of the compute clusters of the MPPA were detected and corrected by a combination of ECC and bit-interleaving. ECC detects two errors and corrects one. However, if two or more errors are produced in the same word (Multiple Bit Upset (MBU)), ECC may fail. To solve this problem, the manufacturer implements bit-interleaving in the SMEMs to spread the bits of data onto

several addresses. In this way, no MBUs was produced. In addition, all upsets on cache memories were detected by parity, and the corresponding cache memory was invalidated. For prediction purposes, fault-injection campaigns were performed on the internal memory and processors' registers. However, results concerning fault-injection in the internal memories were not useful since upsets in memories do not contribute to the cross-section during radiation tests.

To analyze the experimental results, the consequences of a bit-flip in a memory cell were classified in:

- Silent fault: There is no apparent effect on the system; all the system and application processes are terminated in a normal way, and the results are correct.
- Application erroneous result: The result of the application is not the expected one.
- Timeout: The program does not respond after a duration equal to the worst-case execution time.
- Exception: The application triggers an exception routine.

The erroneous results, timeouts and exceptions were considered as errors. Among them, the most critical one is the "application erroneous result" because it is not detected by the system and may cause unpredictable consequences. The timeouts and exceptions are not so critical since they are detected and the application/system is able to manage them.

The error-rate of an application is derived from SEU fault-injection campaigns. This quantity is defined as the average number of injected faults needed to produce an error in the application results.

$$\tau_{Inj} = \frac{Number\ of\ Errors}{Number\ of\ Injected\ Faults} \quad (1)$$

4.1. Experimental Evaluation by Fault-Injection on the Application without Fault-Tolerance

To validate the effectiveness of NMR-MPar, it is first necessary to evaluate the application without redundancy. For this purpose, the application was implemented in Compute Clusters 0–3. Two fault-injection campaigns were executed: the first one for the TSP with 8417 runs and the second one for the MM with 72,497 runs. Table 1 shows a general overview of the fault-injection campaigns on the TSP and MM applications.

Table 1. Results of the fault-injection campaigns on applications running on the MPPA-256.

Application	Silent faults	Erroneous Results	Timeouts	Exceptions
TSP	8197	2	21	197
MM	62,071	5215	112	5099

TSP: Traveling Salesman Problem. MM: Matrix Multiplication.

From these results, the error-rates of both applications were calculated by using Equation (1). Table 2 summarizes the obtained results. These results confirm the intrinsic fault-tolerant capability of the TSP application.

Table 2. Error-rate by fault-injection for Portable Operating System Interface uniX (POSIX) scenarios in MPPA.

Application	No. of Faults Injected	No. of Errors in Registers	τ_{Inj}
TSP	8417	220	2.61%
MM	72,497	10,426	14.38%

τ_{Inj} : application error-rate obtained from fault-injection campaigns

During the fault-injection campaigns, three types of exceptions were produced: (1) "core stuck" when the Processing Engine core targeted by the fault-injector was completely stuck and did not

respond any longer to the HOST requests; (2) "segmentation fault" when the bit-flip caused the core to try to access a memory not allocated; (3) "device exit" when the MPPA device stopped its execution and produced an exit of the process. As expected, it was also observed that the critical registers are application dependent.

In order to minimize the impact of errors, this work implements an application with a timeout principle, so if the execution time of the application is greater than the worst-case execution time, the application is finished. In addition, system exceptions are managed using traps in the OS and/or by a monitor in the HOST that kills the process in the MPPA if the application does not respond.

4.2. Experimental Evaluation by Fault Injection on the Applications Implementing N-Modular Redundancy and M-Partitions

As mentioned before, the fault-injection campaigns consider the emulation of one SEU per execution. For this reason, only one bit-flip was injected in the targeted register of one of the 256 PE cores used by the NMR-MPar application. Table 3 provides details about the two fault-injection campaigns. To prevent the propagation of errors between successive executions, at the end of each run, the HOST resets the platform and reloads the code in the many-core processor. Hence, to guarantee randomicity in fault-injection variables, the HOST was in charge of selecting them.

Table 3. Fault-injection campaign details of the 4MR-8Par applications running on the MPPA.

Application	Standard Exec. (Gcycles)	Time (s)	Runs per
TSP	59.06	147.78	8328
MM	5.53	13.83	72,697

To evaluate the effectiveness of the approach, it is essential to consider the behavior of the system in the presence of an exception, since as Table 2 shows, the error-rate is considerable affected by this type of dysfunction. When an exception occurs in one of the cores of the MPPA, the command used in the HOST to execute a process in the MPPA (k1-jtag-runner) takes control to continue or not with the execution of the process in the MPPA. Consequently, there are two possibilities:

- With supervision: If the standard output of the process is a Tele-TYpewriter (TTY), it prints the reason for the exception that occurred on a given core and waits for a debugger connection while allowing the other cores, which did not halt, to continue working.
- Without supervision: If the standard output is not a TTY or no debugger option is passed to the k1-jtag-runner (MPPA runner command), it considers that the user is not present. Hence, it does not print the exception cause and tries to continue the execution of the halted core(s). If the core where the exception was produced is completely stuck, it does not respond any longer to the HOST request to continue the execution. Then, the HOST kills the process in the MPPA [40].

Taking into account these two options, fault-injection campaigns consider also two scenarios: without supervisor (4MR-8Par-not-tty) and with supervisor (4MR-8Par-tty). The main difference between them is the behavior of the system when an exception occurs. Without a supervisor, the system tries to continue immediately with the execution of the halted core, while with a supervisor, the system allows the other cores to continue working, waits for the standard execution of the application and then continues the execution of the halted core. Table 4 shows a general overview of the fault-injection campaigns on the TSP and the MM application considering the absence of the supervisor; Eight thousand three hundred twenty eight (8328) and seventy two thousand six hundred ninety six (72,696) faults were injected , respectively. This significant difference between the number of injected faults on each application is due to the larger execution time of the TSP.

Table 4. Results of fault-injection campaigns on 4MR-8Par applications without a supervisor.

Application	Silent Faults	Erroneous Results	Timeouts	Exceptions
TSP	8124	0	0	204
MM	68,303	0	0	4393

For the scenario including a supervisor, only the runs resulting in exceptions with the same fault-injection parameters were re-executed, this to evaluate the behavior of the system in this scenario under the same conditions. It was thus necessary to inject 204 additional faults in the TSP and 4393 in the MM. Results show that the number of exceptions was drastically reduced from two hundred four to one and from four thousans three hundred ninety three to one hundred fourty one, respectively. The details regarding the exceptions for both scenarios are summarized in Table 5.

Table 5. Exceptions produced by fault-injection on the 4MR-8Par applications.

Scenario	Core Stuck	Segmentation Fault	Device Exit	Total Exceptions
TSP-no-tty	39	165	0	204
TSP-tty	0	0	1	1
MM-no-tty	402	3948	43	4393
MM-tty	0	95	46	141

The considerable decrease in the number of exceptions in the supervisor scenario can be explained as follows: (1) in all the executions where a "core stuck" was produced, the device could find the correct result thanks to redundant instances; and (2) most of the faults producing "segmentation fault" were also overcame by the redundancy. This improvement is possible since the exception does not affect the whole device, so that other cores could continue working until the HOST kills the process when the worst-case execution time has run out.

Applying the results of fault-injection to Equation (1) allows computing the error-rate on registers for both applications. In general, erroneous results, timeouts and exceptions are considered as errors. However, in this case-study, only exceptions were observed. Table 6 summarizes the obtained values.

Table 6. Error-rates for 4MR-8Par applications implemented on the MPPA.

Application	$\tau_{Inj\_no\_supervisor}$	$\tau_{Inj\_supervisor}$
TSP	2.45 %	0.01 %
MM	6.04 %	0.19 %

5. Discussion

The MPPA many-core processor running massive parallel applications on POSIX was evaluated through different scenarios of the application: (1) without redundancy; (2) with the NMR-MPar approach without a supervisor; and (3) with the NMR-MPar approach with a supervisor. This section presents a comparison between the results of fault-injection campaigns. Figure 4 illustrates the consequences of injected SEUs.

From the results, it is possible to observe that all the erroneous results and timeouts were surpassed by applying the NMR-MPar approach. Concerning the exceptions, the best response is obtained under the scenario with a supervisor.

The current study also considers the evaluation of the reliability of a system during its lifetime. Hence, the failure rate function (λ) is assumed to be constant, and the reliability function is characterized by the expression (2).

$$R(t) = e^{-\lambda t} \tag{2}$$

The failure rate at a given operating environment is calculated from the soft error-rate obtained by fault injection (τ_{Inj}) by applying (3).

$$\lambda_{CEU} = \tau_{Inj} \times \sigma_{Static} \times \phi \quad (3)$$

where σ_{Static} is the static cross-section, which is the average number of particles needed to cause a bit-flip in a memory cell. Its value is obtained from radiation experiments. ϕ is the flux of particles at the specific operation environment.

Regarding the reliability, it was considered that applications are intended to be used in the avionics domain for commercial airlines, being the neutron flux at 35,000 ft of altitude (ϕ_{ev} =2993.2 n/cm^2/h) with a system life time of 50,000 h. The $\sigma_{Static} = 12.71 \times 10^{-9}$ cm^2/device was taken from a previous work [8]. Table 7 summarizes the estimated failure rates per hour computed using (3) at avionics altitude for both applications under three scenarios: without the fault-tolerance approach, NMR-MPar running with the supervisor and implemented with NMR-MPar running without the supervisor device.

Table 7. Failure rates per hour at 35,000 feet for 4MR-8PaR applications implemented on the MPPA.

Application	Without_FT	No_Supervisor	Supervisor
TSP	9.93×10^{-7}	9.32×10^{-7}	3.80×10^{-9}
MM	5.47×10^{-6}	2.30×10^{-6}	7.23×10^{-8}

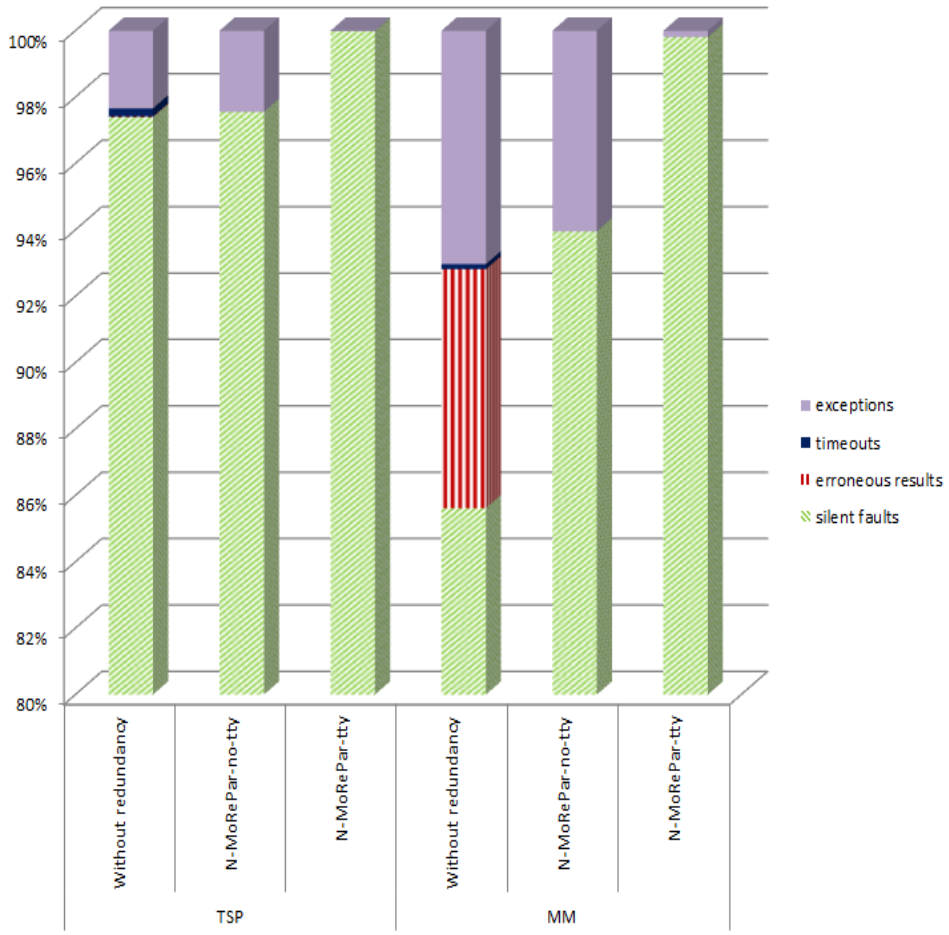

Figure 4. Comparison of Single Event Upset consequences per scenario in N-Modular Redundancy and M-Partions applications.

The reliability of the MM application under the three mentioned scenarios is shown in Figure 5.

A similar comparison was done for the TSP application. Figure 6 illustrates the reliability for the three scenarios. From both results, it is possible to confirm the significant reliability improvement by the use of the NMR-MPar approach.

Considering the implementation of both applications with NMR-MPar running with the supervisor scenario, the reliability of the TSP is 0.9998 at 50,000 h, while that corresponding to the MM case is 0.9964. Only for reference, typical required reliability for space mission applications is >0.9995 [41].

In order to have a first insight into the impact on power consumption of the implementation of the NMR-MPar approach, the power average and the energy consumption of a few runs of the application were measured by using the tool provided by the manufacturer called k1-power. For experimentation purposes, two different operating frequencies were evaluated: the default one (400 MHz) and a lower frequency case (100 MHz). Results are summarized in Tables 8 and 9.

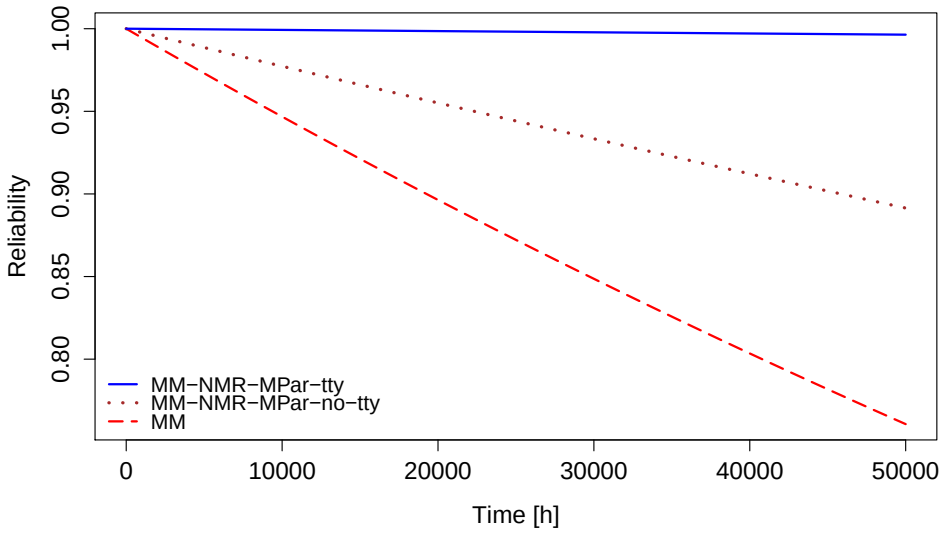

Figure 5. Evaluation of reliability for Matrix Multiplication application at 35,000 ft.

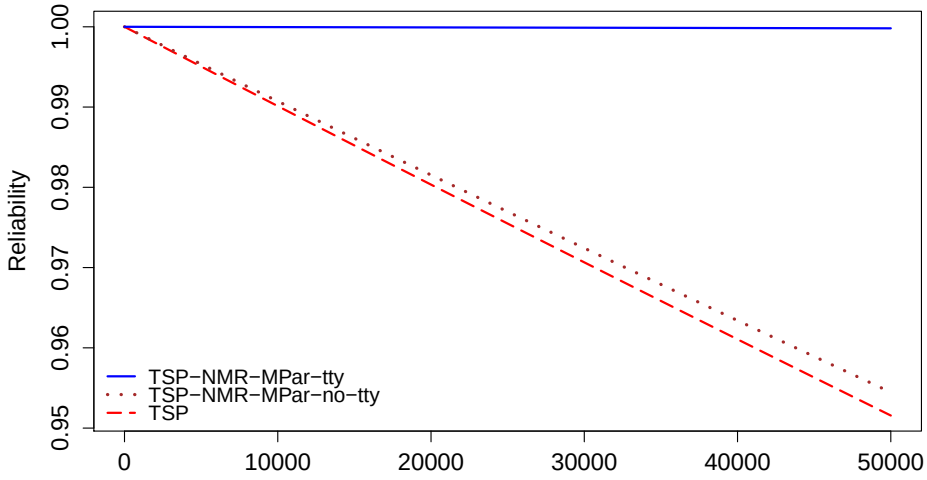

Figure 6. Evaluation of reliability for Traveling Salesman Problem application at 35,000 ft.

Table 8. Power and energy consumption for the studied scenarios of TSP.

Operating Frequency (MHz)	Scenario	Execution Time (s)	Power Average (W)	Energy Consumption (J)
400	w/o redundancy	145.57	5.42	788.38
	NMR-MPar	148.32	9.21	1365.78
100	w/o redundancy	579.38	2.93	1696.73
	NMR-MPar	581.92	4.06	2362.62

Table 9. Power and energy consumption for the studied scenarios of MM.

Operating Frequency (MHz)	Scenario	Execution Time (s)	Power Average (W)	Energy Consumption (J)
400	w/o redundancy	11.86	7.02	83.26
	NMR-MPar	14.38	14.71	211.52
100	w/o redundancy	44.55	3.30	146.99
	NMR-MPar	47.09	5.95	280.06

Since the overhead of the implementation of the NMR-MPar is almost fixed to aroun one Giga cycles independently of the application, for short applications, the impact is greater. From the results presented in the above table, it is possible to observe in the case of TSP that, when the application runs at 400 MHz, there is an increase of about 1.7 times in the power average and energy consumption when executing the application with or without redundancy. This relation is reduced to 1.4-times when running the application at 100 MHz. In the case of MM, operating at 400 MHz, there is an increase of two-times the power average and 2.5-times the energy consumption; while at 100 MHz, there is an increase of 1.8-times the power average and 1.9-times the energy consumption.

It is also important to analyze the behavior of the system when decreasing the operating frequency four times. For all the scenarios, there was a 40–54% reduction of the power, but an increase of 1.3–2.1-times in the energy consumption. Consequently, by reducing the operating frequency, the power dissipation and performance are reduced, while the energy consumption increases.

Regarding future directions, this work opens up various possibilities to continue the investigation in evaluating the NMR-MPar approach under particle radiation to confirm the results obtained from fault-injection, and further tests must be run to appraise its overhead better in terms of energy and power consumption. Furthermore, it is interesting to conjugate this approach with other Software-Implemented Fault Tolerance (SIFT) techniques to better improve the reliability of the system.

6. Materials and Methods

6.1. Target Device

The KALRAY MPPA-256 is a many-core processor manufactured in 28 nm CMOS TSMCtechnology. The processor operates between 100 MHz and 600 MHz, for a typical power ranging between 15 W and 25 W. Its peak floating-point performances at 600 MHz are 634 GFLOPS and 316 GFLOPS for single- and double-precision, respectively. It integrates 256 PE cores and 32 RM cores. The MPPA-256 considered in this work is the second version called Bostan.

The global processor architecture is clustered with 16 CCs and two input/output (I/O) clusters per device. Each compute cluster is built around a multi-banked local static memory (SMEM) of 2 MB shared by the 16 (PE) + 1(RM) cores. Each I/O cluster has two quad-cores and two main banks of 2 MB. Each quad-core is a symmetric multi-processing system. A wormhole switching NoC with 32 nodes and a 2D torus topology connects the compute clusters and the I/O clusters. In Figure 7 is illustrated an overview of the MPPA many-core architecture [42]. The SMEMs and NoC router queues are covered by ECC and the cache memories by parity.

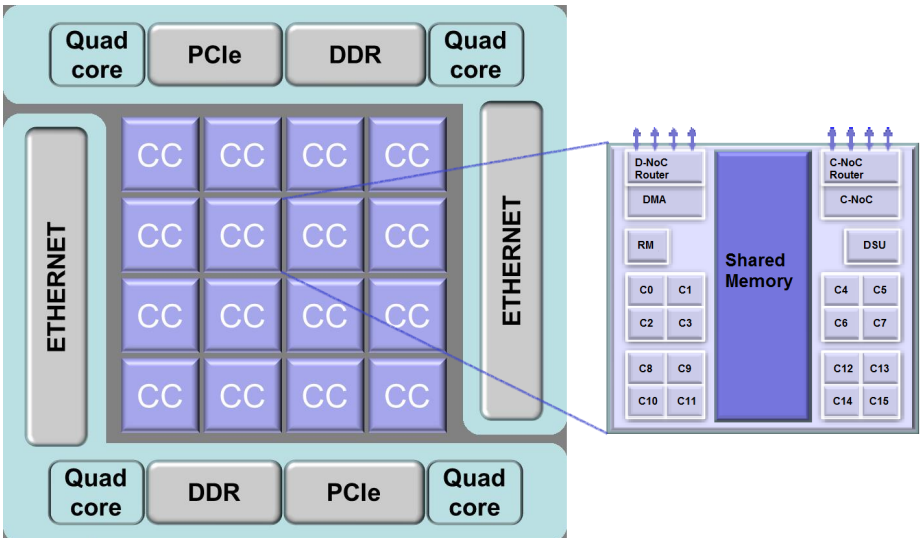

Figure 7. Many-core processor components. CC: Compute Cluster. DDR: Double Data Rate memory. PCIe: Peripheral Component Interconnect express. D-Noc: Data Network-on-Chip. C-Noc: Control Network-on-Chip. DMA: Direct Memory Access. DSU: Debug Support Unit. RM: Resource Manager. Cx: Core x, being x 0 to 15.

6.1.1. Programming Multi-Purpose Processing Array Issues

As was mentioned, the MPPA is a co-processor of the MPPA Developer platform used in the present work. The latter is a development platform based on an Intel core I7 CPU operating at 3.6 GHz and running a Linux CentOS 7 x86 64 , Eclipse 4.3 (aka Kepler) and MPPA ACCESSCORE v1.4. MPPA ACCESSCORE is a set of libraries given by the manufacturer to manage the MPPA-256. It includes three programming models targeting different application needs and development constraints: the POSIX, the Kalray OpenCL and the Lowlevel programming models [43].

The MPPA many-core is available within the Developer as an accelerator of the X86 Host CPU. In order to program the MPPA Developer, the application is commonly divided into a HOST part and an MPPA part. However, it is also possible to run applications only on the MPPA. The HOST part can use full Linux capabilities available on the CPU host. For the MPPA part, each Compute Cluster or IO cluster can run an executable file. Therefore, the many-core processor can simultaneously run as many executable codes as there are clusters. The execution of a multi-binary file on the hardware or on the platform simulator can be accomplished either by Joint Test Action Group (JTAG) or the PCIe expansion bus [40,43].

The main challenges the programmers face when adapting parallel applications to this device are the following: (1) the use of NoC primitives for communication; (2) the cache coherence must be guaranteed by the programmer; and (3) the limited memory inside the cluster (2 MB for OS, code and data).

6.2. Benchmark Applications

This work proposes the use of two types of distributed applications: a CPU-bound and a memory-bound one. The selected CPU-bound application is the Traveling Salesman Problem (TSP), a Non-deterministic Polynomial (NP) hard problem very often used to evaluate computing system optimization [44,45]. This application aims at finding the shortest possible route to visit n cities, visiting each city exactly once and returning to the departure city. To solve the problem, there are several proposals. This work uses a brute force exact algorithm based on a simple heuristic. The implemented version of TSP on the MPPA by the authors of [46] was used as a basis. All the computing operations in TSP are integer.

In this distributed algorithm, the TSP is used to solve a 17-city problem. When a new minimum distance is found by a CC, the new value and the corresponding path are broadcast to the others CCs. The total number of tasks is the number of threads per CC times the number of CC used. To validate the result of the TSP application, the verification of the response includes a valid path. This means that each city must be included in the response only once. It is important to note that the implemented TSP application has an intrinsic fault tolerance characteristic due to the fact that each thread is independent of the others and can find the right result even if other threads crash.

On the other hand, the MM was chosen as a memory-bound application. The MM is widely used to solve scientific problems related to linear algebra, such as systems of equations, calculus of structures and determinants. Concerning avionic applications, MM is used for image processing, filtering, adaptive control and navigation and tracking. In addition, the parallelism of MM is one of the most fundamental problems in distributed and High Performance Computing. There are many approaches proposed to optimize performance. The present work implements the approach divide and conquer. In this work, a 256×256 matrix iterated 8192 times is used. Due to data being typed as single precision (four bytes per matrix element), each matrix occupies 4×256^2 bytes. To validate the results, a standard CRC code was applied line by line to the resulting matrix. Since the size of each matrix element is 32 bits, the CRC-32 set by IEEE 802.3 was used.

Both applications were configured with four CCs in order to maximize the use of resources. Each application tests different resources of the CPU. Furthermore, the use of NoC resources is completely different for both applications. While the MM maximizes the use of resources at the beginning and at the end of the computation, the TSP continuously uses the NoC to request tasks and broadcast the new minimum distance and the path that were found. Implementation details of both applications are presented in Appendices A and B.

6.3. Fault-Injection Details

The fault-injection technique proposed in this approach to emulate SEU is an SWIFI technique. As was previously stated, the approach validated in our previous works concerning fault-injection on multi-core processors [7,37–39] is based on Code Emulated Upset (CEU) principles [47], which reproduce, without intrusion, the effects of SEUs. This is achieved by the assertion of asynchronous interrupt signals. The execution of the interrupt handler produces the error in a randomly-chosen target. The CEU approach uses an external device to produce an interruption that emulates a bit-flip in a targeted memory cell. In the case of multi-/many-core processors, it is possible to benefit from the multiplicity of cores by using one of them as a fault-injector while the others run the chosen application. More details about the implementation of the fault-injector tool can be found in our work [48].

In order to produce the fault-injection by interrupting the targeted core, the portal NoC communication primitive is used. Thus, the configuration of one portal per cluster (portal_fi) is needed for fault-injection purposes. The RM0 of IO Cluster 0 is in charge of configuring the master part of each portal. Note that, although the device is divided into two independent global parts, the communication between clusters is always possible so that IO Cluster 0 could communicate with the 16 CCs. The portal is used to interrupt the CC that contains the selected PE core to perform the fault-injection.

To emulate a bit-flip in the selected register, the fault injector must interrupt the selected core. To achieve this goal, at the random instant within the nominal duration of the application, the fault-injector writes the fault-injection variables, random PE, random address and random bit in the portal_fi of the CC that contains the selected core. This causes an interruption of the RM, which immediately assigns the execution of an interruption handler to any one of the PEs. The assigned PE reads the information in the portal_fi, which contains the selected core, register and bit to modify. If it is the targeted core, it changes the bit in the selected register. Otherwise, it sends an inter-processor interrupt to the corresponding PE, which performs the bit-flip emulation.

During fault-injection campaigns, only the emulation of one SEU per execution was considered to be injected in the targeted register belonging to one of the PE cores used by the application. In order to avoid the propagation of errors to the next execution, the HOST resets the platform and reloads the code to the MPPA processor after each run. Hence, the random variables required by the fault-injector are provided by the HOST.

7. Conclusions

The NMR-MPar is a generic software-approach developed to improve the reliability of applications running in multi-/many-core processors by using partitioning, spatial redundancy and redundancy in data. Spatial redundancy is proposed at the system and application levels, surpassing the weakness of classical redundancy approaches that only replicate processes at the application level, leaving the system level unprotected. In addition, partitioning guarantees the isolation needed to certify safety-critical applications. Experimental results demonstrated that there is a significant decrease of the application error-rate, about two orders of magnitude, when implementing the NMR-MPar approach on the MPPA many-core processor.

Taking into account the improvements obtained by applying the NMR-MPar to the selected benchmarks and considering that (1) no erroneous results and timeouts occurred, (2) all the errors were produced by exceptions, (3) all the exceptions were detected and (4) exceptions can be easily managed by the system through a reset, this approach is very useful for applications that do not have a critical time constraint. Moreover, the reliability of many-core processors could be better improved by implementing SIFT techniques at the application level.

Lastly, the NMR-MPar can be implemented in other multi-/many-core processors, the implementation being specific for each device. Evaluation results of the implemented case study are encouraging since they open the possibility to use massive parallelism for dependable applications in embedded systems.

Supplementary Materials: The codes of the implementation of the case study are available upon request to the corresponding author (vcvargas@espe.edu.ec).

Acknowledgments: This work was supported in a part by the Universidad de las Fuerzas Armadas ESPE and by the Secretaría de Educación Superior, Ciencia, Tecnología e Innovación del Ecuador (SENESCYT) through the grant PIC-2017-EXT-004 and STIC -AmSud (Science et Technologie de l'Information et de la Communication en Amérique du Sud) - Energy-aware Scheduling and Fault Tolerance Techniques for the Exascale Era (EnergySFE) Project PIC-16-ESPE-STIC-001 and by the French authorities through the "Investissements d'Avenir" program (CAPACITES project). The authors thank Stéphane Gailhard from the Societé Kalray for his valuable contribution to solving the MPPA programming issues and to Diego Arcos from Universidad de las Fuerzas Armadas ESPE for his helpful comments and suggestions.

Author Contributions: V.V., J.-F.M. and R.V. conceived of and designed the experiments. V.V. and P.R. contributed the fault-injection tool, performed the experiments and analyzed the data. J.-F.M. and R.V. contributed materials. V.V. and P.R. wrote the paper.

Conflicts of Interest: The authors declare no conflict of interest. The founding sponsors had no role in the design of the study; in the collection, analyses or interpretation of data; in the writing of the manuscript; nor in the decision to publish the results.

Abbreviations

The following abbreviations are used in this manuscript:

ACROSS	Advanced Cockpit for Reduction Of StreSs and workload
AMP	Asymmetric Multi-Processing
CAST	Certified Associate in Software Testing
CAPACITES	Calcul Parallèle pour Applications Critiques en Temps et Sureté
CC	Compute Cluster
CEU	Code Emulated Upset
COTS	Commercial-Off-The-Shelf
CRC	Cyclic Redundancy Check

DMR	Double Modular Redundancy
EASA	European Aviation Safety Agency
ECC	Error Correcting Code
GPR	General Purpose Register
GRP	Guaranteed Resource Partition
HPC	High Performance Computing
IMA	Integrated Modular Avionics
MM	Matrix Multiplication
MPPA	Multi-Purpose Processing Array
NMR	N-Modular Redundancy
NoC	Network-on-Chip
NP	Non-deterministic Polynomial
OS	Operating System
PE	Processing Engine
PLR	Process-Level Redundancy
pSWP	parallel Software Partition
RECOMP	Reduced certification Costs for trusted Multi-core Platforms
RHBD	Radiation Hardened By Design
RM	Resource Manager
SEE	Single Event Effect
SEU	Single Event Upset
SFR	System Function Register
SIFT	Software-Implemented Fault-Tolerance
SMEM	Static Memory
SMT	Simultaneous Multi-Threading
SMP	Symmetric Multi-Processing
SWIFI	Software-Implemented Fault-Injection
TIMA	Techniques de l'Informatique et de la Microeléctronique pour l'Architecture des systèmes integrés
TMR	Triple Modular Redundancy
TSMC	Taiwan Semiconductor Manufacturing Company
TSP	Traveling Salesman Problem
TTY	Tele-TYpewriter

Appendix A

Appendix A.1 Implementation Details of the Code

This Appendix presents details regarding the applications code of the case study that implements the NMR-MPar approach. The pseudo-code of the master code that runs on each IO cluster of the MPPA is shown in Figure A1.

Each IO cluster manages two instances (NUMBER_LOCAL_IO_INSTANCES) of the NMR-MPar implemented on the MPPA many-core processor. Each instance is related to the main_slave procedure. Both main_slave functions are run by the RM1 and RM2, respectively. In spite of each IO cluster running only two instances, it configures a memory space to save the results of the four instances (RESULT [NUMBER_INSTANCES]) of the quadruple modular redundancy.

The RM0 of each IO cluster performs the main part of the NMR-MPar implementation. Due to the possibility of having a crash in a part of the system, the application used the principle of timeout, so if the execution time were greater than the standard execution time, the main application would continue with the next stage. The different stages of the main function considering this principle are outlined in Figure A2.

The RM0 configures all the MPPA connections except the sync primitives of each instance that is configured for the related RM that runs the main_slave function. Some details of the main function of the master code that runs on the RM0 of each IO cluster are listed in the following:

- The function create_mppa_connections_for_instance creates the portal and queue connections used for the inter-cluster communication of each application's instance detailed in Section 6.
- The function create_fault_injection_portal_connection allows opening a channel of communication if fault-injection is performed.
- The function create_mppa_compl_connections creates: (1) two sync primitives for synchronization with the other IO cluster; (2) two portals to send and receive the results of each LOCAL_IO_INSTANCE to/from the other IO cluster; (3) two portals to send and receive the results of the voter and the program_ended variable to/from the other IO cluster.

```
global fault_injection_variables, portals, queues, program_ended

procedure MAIN_SLAVE (nb_threads, application_variables, nb_clusters)
    WAIT_BARRIER( )
    RUN_APPLICATION(nb_threads, application_variables, nb_clusters)
    SEND_RESULTS_OTHER_IO(portal)
    WAIT_FOR (VOTER_OR_TIMEOUT )
    WAIT_FOR(program_ended OR TIMEOUT)
procedure MAIN_VOTER_FAULT_INJECTOR
    WAIT_BARRIER( )
    while program_end != OR NO_TIMEOUT
    do { if cycles_applic == rand_inst AND io_cluster_0
         then { INJECT_FAULT( )
         if every_instance_finished OR TIMEOUT
         then { TASK_VOTER( )
                SEND_VOTER_RESULTS_TO_OTHER_IO (portal)
                if other_io_not_log
                then { LOG_RESULTS( )
    WAIT_FOR(program_ended OR TIMEOUT)
procedure MAIN (fault_injection_parameters, application_variables, nb_clusters)
    for i ← 1 to NUMBER_INSTANCES
      do INITIALIZE_RESULTS(i)
    for i ← 1 to NUMBER_LOCAL_IO_ INSTANCES
      do CREATE_MPPA_CONNECTIONS_FOR_INSTANCES(i,port,nb_clusters )
    if fault_injection = 1
    then { fault_injection_variables ← fault_injection_parameters
           CREATE_FAULT_INJECTION_PORTAL_CONNECTIONS()
    CREATE_MPPA_COMPL_CONNECTIONS( )
    START_RM( 1, & MAIN_SLAVE )
    START_RM( 2, & MAIN_SLAVE )
    START_RM( 3, & MAIN_VOTER_FAULT_INJECTOR )
    WAIT_BARRIER( )
    WAIT_FOR (END_RUN_APPLIC_IN_SLAVES OR TIMEOUT )
    WAIT_FOR (VOTER_OR_TIMEOUT )
    program_ended ← 1
    SEND_TO_OTHER_IO (program_ended)
    WAIT_FOR(program_end_other_io OR TIMEOUT)
    CLOSE_MPPA_CONNECTIONS( )
```

Figure A1. Pseudo-code for the implementation of the NMR-MPar applied to the MPPA.

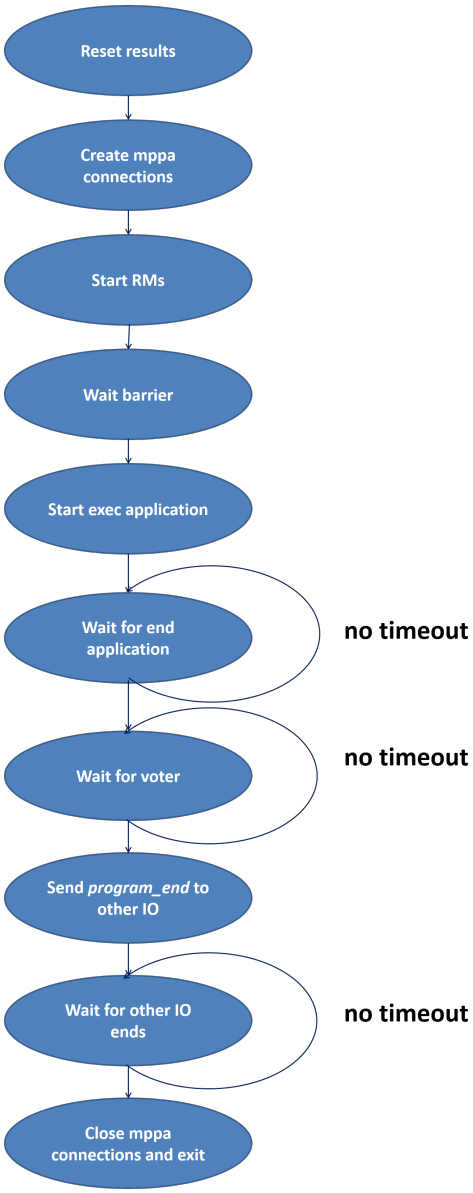

Figure A2. Flow diagram of the main function run in RM0.

On the other hand, the details of the function run_application, which is part of the main_slave code executed by the RM1 and the RM2 of each cluster, are illustrated in Figure A3.

Finally, concerning the function main_voter_fault_injector, it is important to note that both RM3 perform the majority voting according to the principles detailed in Section 3.3.1. Nonetheless, only the RM3 of IO Cluster 0 performs the function inject_fault at the random instant rand_inst.

```
global portals, queues
procedure RUN_APPLICATION(nb_threads, application_variables, nb_clusters)
sync ← CREATES_SYNC_CONNECTION( )
INITIALIZE_VARIABLES( )
for i ← 1 to nb_clusters
  do SPAWN_CLUSTER( SLAVE_WORK(application_variables, nb_threads) )
WAIT_BARRIER(sync)
finished_clusters ← 0
while finished_clusters < nb_clusters
do { TX_RX(portals)
     TX_RX (queues)
     if CLUSTER_FINISHED( ) = 1
     then { finished_clusters ++

WAIT_BARRIER(sync)
LOG_RESULTS( )
CLOSE_SYNC_CONNECTION(sync)
```

Figure A3. Generic master pseudo-code for the applications running on the MPPA.

Appendix B

Appendix B.1 Benchmark Details

The implementation of the Traveling Salesman Problem and Matrix Multiplication applications allows configuring the number of CCs from one to four, as well as the problem size. Table A1 illustrates the execution time of different possible configurations for the TSP application. The time in seconds is done for a configuration of the device with an operating frequency of 400 MHz.

Table A1. Standard execution time for different configurations of TSP on the MPPA.

No. of Cities	1 Cluster		2 Clusters		3 Clusters		4 Clusters	
	(Gcycles)	(s)	(Gcycles)	(s)	(Gcycles)	(s)	(Gcycles)	(s)
16	30.8	77	16.2	40.5	11.3	28.3	8.6	21.5
17	188.8	472	102.8	257	71.2	178	58.1	145.2
18	521.9	1304.2	260.3	650.8	188.2	470.5	159.8	399.5

To perform both applications, this case-study considers four CCs as slaves of the RM1 of the IO cluster. Then, five RMs and 64 PEs are involved in the application itself plus the RM0 that starts, monitors and manages the messages received by the NoC. Both applications were configured without a HOST part, and the multi-binary executable file was loaded to both IO clusters through the JTAG port. Figure A4 illustrates the booting process of the 4MR-8Par application.

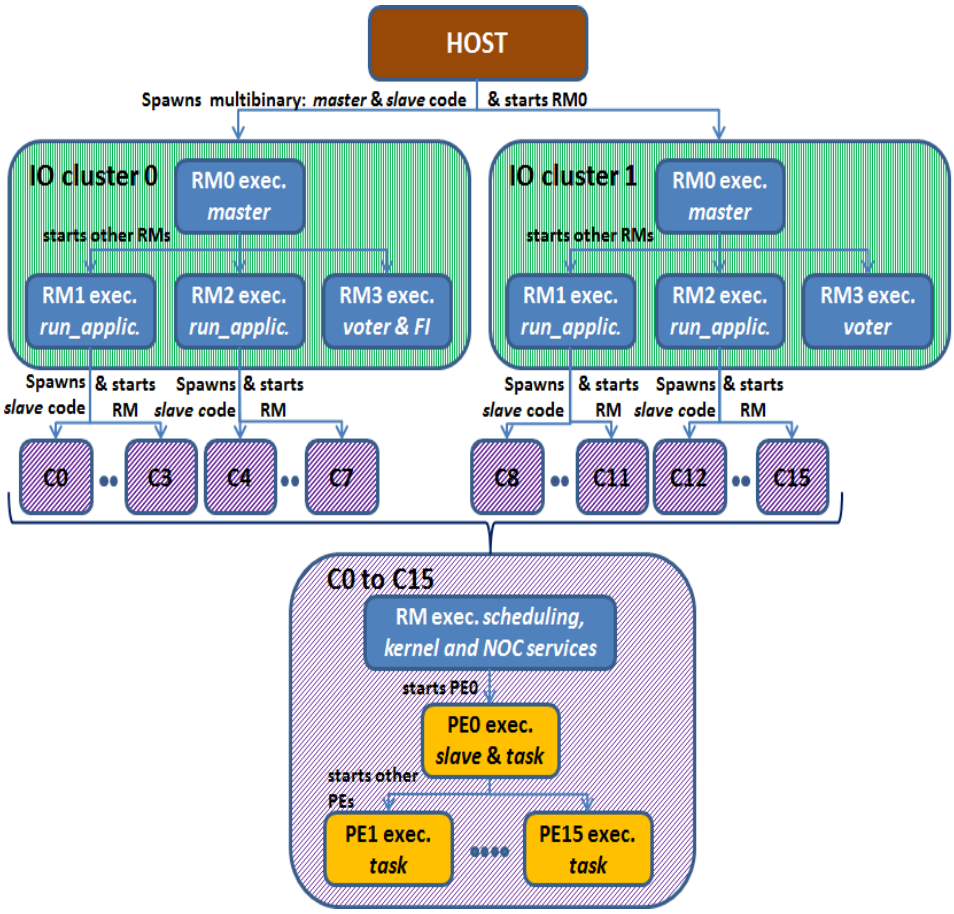

Figure A4. Booting process of the 4MR-8Par application.

Before starting the distributed application, the RM0 initializes the primitives needed for the communication, then wakes up the other RMs, the master of the application and the fault injector on the IO cluster. Once the RM1 is started, it performs the run_application function, the pseudo-code of which is shown in Figure A3. It is important to note that if any cluster does not finish its job in the normal execution time, the RM0 logs a timeout and ends the application.

Appendix B.2 Traveling Salesman Problem Details

The TSP was configured to solve a 17-city problem in order to have enough processing time to observe errors within a reasonable overall simulation time. The details of the stages of the generic master code of the TSP application are the following:

- During the initialization, the master populates a queue of jobs from which each cluster takes the tasks to be executed by its PEs.
- The first barrier serves to wait that all clusters wake up.
- In the loop, the master waits for job requests done by the CCs and then assigns jobs to slaves.
- The second barrier waits for the end of the task of all the involved clusters.
- A consolidation of results is done before logging the minimum distance and the corresponding path.

For this application, the RM0 creates the following MPPA connections for the master: (1) two syncs for synchronization with the slaves, (2) one queue for the job request, (3) four queues for the job response (one per cluster), (4) one portal to receive the results of the minimum distance and its path and (5) four portals to transmit the results.

```
global queue, min_path
procedure GENERATE_TASKS(n_hops, last_city, current_cost, cities)
 if n_hops = max_hops
   then { task ← (last_city, current_cost, cities)
        { ENQUEUE_TASK(queue, task)
   else { for each i ∈ cities
        {   do { if last_city = none
        {      { then last_cost ← 0
        {      { else last_cost ← costs[last_city, i]
        {      { new_cost ← curr_cost + last_cost
        {      { GENERATE_TASKS(n_hops + 1, i, new_cost, cities\{i})

procedure DO_WORK()
 while queue ≠ ∅
   do { (last_city, current_cost, cities) ← ATOMIC_DEQUEUE(queue)
      { TSP_SOLVE(last_city, current_cost, cities)

main
 min_path ← ∞
 GENERATE_TASKS(0, none, 0, {1, 2, ..., n_cities})
 for i ← 1 to n_threads
   do SPAWN_THREAD(DO_WORK())
 WAIT_EVERY_CHILD_THREAD()
 output (min_path)
```

Figure A5. Pseudo-code for the multi-threading TSP version [46].

On the other hand, the code performed by the slaves corresponds to the multi-threading TSP version detailed as follows. A multi-threading algorithm fills a queue of tasks. Each task is one of the branches of the search tree. Each thread takes tasks from the queue and executes the search. The minimum distance is a global variable shared by all the threads. The number of tasks is a function of the levels of the search tree and the number of cities. The pseudo-code for the multi-threading version of this code is illustrated in Figure A5.

Each cluster performs the slave code almost independently of the others. It has its own minimum distance and path variables. The only moment when cores interact with each other is when a new minimum distance is found by a CC. The latter broadcasts this information to the master and other slaves so that each one of them updates its related variables. Both synchronization barriers are intended to synchronize the slaves with the master. Inside the cluster, each PE uses MUTEX directives to access global CC variables.

Each CC executing the slave code uses the following MPPA connections: (1) two syncs for synchronization with the master, (2) one queue for the job request, (3) one queue for the job response, (4) one portal to receive the results of the minimum distance and its path and (5) four portals to broadcast the results.

Appendix B.3 Matrix Multiplication Details

The application is based on an assembler optimized version of a cooperative 256×256 matrix multiplication. The matrix multiplication is performed 8192 times, and the result C is the summation of these computations as stated in (A1). The iteration of the matrix operation is done to guarantee

that each cluster computes enough time so that all the clusters work in parallel during a considerable time slice.

$$C = \sum_{n=1}^{256} A \times B \quad \text{(A1)}$$

A, B and C are single precision floating-point matrices. The size of the matrix was chosen so that data remain in the local SMEM memory.

Each slave computes $^1/_4$ of the result. The main process of each CC creates 16 POSIX threads, one for each PE. Therefore, each PE calculates $^1/_{64}$ of the result by executing the same assembler optimized version. Since there is no hardware memory coherency in the compute cluster, each PE ensures memory coherency by software means, by updating shared data before (read coherency) and after the computation (write coherency). In addition, the RM calls the memory coherency functions when using the shared data.

The MPPA connections created for each instance of this application are summarized as follows. For the master in the IO cluster and the slaves in the CC cluster: (1) two syncs for synchronization, (2) one portal to receive the results and (3) four portals to broadcast the results.

Appendix B.4 Use of NOC Resources

Table A2 summarizes the number of MPPA connections for both applications. The TSP uses 70 MPPA inter-cluster communications, while MM uses 50. The number of the connections gives an idea of the degree of complexity of the collective communications and their costs. From the casting point of view, some of them are used to broadcast information to all the slaves, others to multi-cast information to some clusters and others to uni-cast information to a specific cluster. The majority of these connections are configured in asynchronous mode so that a notify function is configured to interrupt the cluster when data are received. Most of them configure four senders and four receivers. Due to the parallelism, there are some connections trying to access the same NoC resources at the same time. Therefore, the lack of resources is frequent in the default configuration [49]. This was solved by changing the setup configuration of the functions and by adding delays in the application. It is a fact that waiting for new available resources degrades the performance of the application. The use of resources is extremely related to the amount of data sent by each connection.

Table A2. NoC connections on applications running NMR-MPar on the MPPA.

Application	Sync	Portal	Queue
TSP	10	24	20
MM	10	24	0
Fault injector	0	16	0
Total TSP	10	40	20
Total MM	10	40	0

References

1. TOP 500 Supercomputer list November 2017. Available online: https://www.top500.org/lists/2017/11/ (accessed on 16 March 2018).
2. Across. Advanced Cockpit for Reduction Of StreSs and Workload. Available online: http://www.across-fp7.eu (accessed on 16 March 2016).
3. MultiPARTES. Multi-Cores Partitioning for Trusted Embedded Systems. Available online: http://www.multipartes.eu (accessed on 14 April 2017).
4. Guertin, S. Initial SEE Test of Maestro. Presented at the Fifth Workshop on Fault-Tolerant Spaceborne Computing Employing New Technologies, Albuquerque, NM, USA, 29 May–1 June 2012. Available online: https://trs.jpl.nasa.gov/bitstream/handle/2014/42682/12-2145_A1b.pdf?sequence=1 (accessed on 16 March 2018).

5. Stolt, S.S.; Normand, E. A Multicore Server SEE Cross Section Model. *IEEE Trans. Nucl. Sci.* **2012**, *59*, 2803–2810.
6. Oliveira, D.A.G.; Rech, P.; Quinn, H.M.; Fairbanks, T.D.; Monroe, L.; Michalak, S.E.; Anderson-Cook, C.; Navaux, P.O.A.; Carro, L. Modern GPUs Radiation Sensitivity Evaluation and Mitigation Through Duplication With Comparison. *IEEE Trans. Nucl. Sci.* **2014**, *61*, 3115–3122.
7. Ramos, P.; Vargas, V.; Baylac, M.; Villa, F.; Rey, S.; Clemente, J.A.; Zergainoh, N.E.; Méhaut, J.F.; Velazco, R. Evaluating the SEE sensitivity of a 45 nm SOI Multi-core Processor due to 14 MeV Neutrons. *IEEE Trans. Nucl. Sci.* **2016**, *63*, 2193–2200.
8. Vargas, V.; Ramos, P.; Ray, V.; Jalier, C.; Stevens, R.; Dupont de Dinechin, B.; Baylac, M.; Villa, F.; Rey, S.; Zergainoh, N.E.; et al. Radiation Experiments on a 28 nm Single-Chip Many-core Processor and SEU error-rate prediction. *IEEE Trans. Nucl. Sci.* **2016**, *99*, 1–8.
9. Hyman, R.; Bhattacharya, K.; Ranganathan, N. Redundancy Mining for Soft Error Detection in Multicore Processors. *IEEE Trans. Comp.* **2011**, *60*, 1114–1125.
10. Vijaykumar, T.N.; Pomeranz, I.; Cheng, K. Transient Fault Recovery using Simultaneous Multithreading. In Proceedings of the 29th Annual International Symposium on Computer Architecture, Istanbul, Turkey, 18–22 November 2002; pp. 87–98.
11. Basile, C.; Kalbarczyk, Z.; Iyer, R.K. Active Replication of Multithreaded Applications. *IEEE Trans. Parallel Distrib. Syst.* **2006**, *17*, 448–465.
12. Mushtaq, H.; Al-Ars, Z.; Bertels, K. Efficient Software-Based Fault Tolerance Approach on Multicore Platforms. In Proceedings of the Design, Automation & Test in Europe Conference, Grenoble, France, 18–22 March 2013; pp. 921–926.
13. Reinhardt, S.; Mukherjee, S. Transient fault detection via simultaneous multithreading. In Proceedings of the 27th International Symposium on Computer Architecture (IEEE Cat. No.RS00201), Vancouver, BC, Canada, 10–14 June 2000; pp. 25–36.
14. Mukherjee, S.; Kontz, M.; Reinhardt, S. Detailed design and evaluation of redundant multi-threading alternatives. In Proceedings of the 29th Annual International Symposium on Computer Architecture, Anchorage, AK, USA, 25–29 May 2002; pp. 99–110.
15. Holler, A.; Rauter, T.; Iber, J.; Macher, G.; Kreiner, C. Software-Based Fault Recovery via Adaptive Diversity for Reliable COTS Multi-Core Processors. In Proceedings of the 6th International Workshop on Adaptive Self-tuning Computing Systems, Prague, Czech Republic, 18 January 2016; pp. 1–6.
16. Shye, A.; Blomstedt, J.; Moseley, T.; Janapa Reddi, V.; Connors, D.A. PLR: A Software Approach to Transient Fault Tolerance for Multicore Architectures. *IEEE Trans. Depend. Secure Comput.* **2009**, *6*, 135–148.
17. Bolchini, C.; Miele, A.; Sciuto, D. An adaptive approach for online fault management in many-core architectures. In Proceedings of the 2012 Design, Automation Test in Europe Conference Exhibition, Dresden, Germany, 12–16 March 2012; pp. 1429–1432.
18. Alhakeem, M.S.; Munk, P.; Lisicki, R.; Parzyjegla, H.; Parzyjegla, H.; Muehl, G. A Framework for Adaptive Software-Based Reliability in COTS Many-Core Processors. In Proceedings of the ARCS 2015-The 28th International Conference on Architecture of Computing Systems, Porto, Portugal, 24–27 March 2015; pp. 1–4.
19. Kim, E.P.; Shanbhag, N.R. Soft N-Modular Redundancy. *IEEE Trans. Comput.* **2012**, *61*, 323–336.
20. Walters, J.P.; Kost, R.; Singh, K.; Suh, J.; Crago, S.P. Software-based fault tolerance for the Maestro many-core processor. In Proceedings of the 2011 Aerospace Conference, Big Sky, MT, USA, 2–12 March 2011.
21. Löfwenmark, A.; Nadjm-Tehrani, S. Challenges in Future Avionic Systems on Multi-Core Platforms. In Proceedings of the 2014 IEEE International Symposium on Software Reliability Engineering Workshops, Naples, Italy, 3–6 November 2014; pp. 115–119.
22. Mollison, M.S.; Erickson, J.P.; Anderson, J.H.; Baruah, S.K.; Scoredos, J.A. Mixed-Criticality Real-Time Scheduling for Multicore Systems. In Proceedings of the 2010 10th IEEE International Conference on Computer and Information Technology, Bradford, UK, 29 June–1 July 2010; pp. 1864–1871.
23. Panić, M.; Quiñones, E.; Zavkov, P.G.; Hernandez, C.; Abella, J.; Cazorla, F.J. Parallel many-core avionics systems. In Proceedings of the 2014 International Conference on Embedded Software (EMSOFT), New Delhi, India, 12–17 October 2014; pp. 1–10.
24. Trujillo, S.; Crespo, A.; Alonso, A.; Pérez, J. MultiPARTES: Multi-core partitioning and virtualization for easing the certification of mixed-criticality systems. *Microprocess. Microsyst.* **2014**, *38*, 921–932.

25. Galizzi, J.; Pignol, M.; Masmano, M.; Munoz, M.; Coronel, J.; Parrain, T.; Combettes, P. Temporal Duplex-Triplex on COTS processors with XtratuM. In Proceedings of the 2016 Eurospace DASIA Conference, Tallin, Estonia, 10–12 May 2016; pp. 1–5.
26. Kim, S.; Somani, A.K. Area efficient architectures for information integrity in cache memories. In Proceedings of the 26th International Symposium on Computer Architecture (Cat. No.99CB36367), Atlanta, GA, USA, 1–4 May 1999; pp. 246–255.
27. Zhang, W.; Gurumurthi, S.; Kandemir, M.; Sivasubramaniam, A. ICR: in-cache replication for enhancing data cache reliability. In Proceedings of the 2003 International Conference on Dependable Systems and Networks, San Francisco, CA, USA, 22–25 June 2003; pp. 291–300.
28. Zhang, W. Replication cache: a small fully associative cache to improve data cache reliability. *IEEE Trans. Comput.* **2005**, *54*, 1547–1555.
29. Sugihara, M.; Ishihara, T.; Murakami, K. Task Scheduling for Reliable Cache Architectures of Multiprocessor Systems. In Proceedings of the 2007 Design, Automation Test in Europe Conference Exhibition, Nice, France, 16–20 April 2007; pp. 1–6.
30. Sundaram, A.; Aakel, A.; Lockhart, D.; Thaker, D.; Franklin, D. Efficient Fault Tolerance in Multi-media Applications Through Selective Instruction Replication. In Proceedings of the 2008 Workshop on Radiation Effects and Fault Tolerance in Nanometer Technologies, Ischia, Italy, 5–7 May 2008; ACM: New York, NY, USA, 2008; pp. 339–346.
31. Memik, G.; Kandemir, M.; Ozturk, O. Increasing register file immunity to transient errors. In Proceedings of the Design, Automation and Test in Europe, Munich, Germany, 7–11 March 2005; pp. 586–591.
32. Tabkhi, H.; Schirner, G. Application-specific power-efficient approach for reducing register file vulnerability. In Proceedings of the Design, Automation Test in Europe Conference Exhibition (DATE), Dresden, Germany, 12–16 May 2012; pp. 574–577.
33. Lyons, R.; Vanderkulk, W. The use of triple modular redundancy to improve computer reliability. *IBM J. Res. Dev.* **1962**, *6*, 200–209.
34. Shooman, M. *Reliability of Computer Systems and Networks: Fault Tolerance, Analysis, and Design*; John Wiley & Sons, Inc.: New York, NY, USA, 2002.
35. Koren, I.; Su, S.Y.H. Reliability Analysis of N-Modular Redundancy Systems with Intermittent and Permanen Faults. *IEEE Trans. Comput.* **1979**, *c-28*, 514–520.
36. Integrated Modular Avionics (IMA) Developement Guidance and Certification Considerations; RTCA/DO-297; IEEE Global Spec. Available online: https://standards.globalspec.com/std/2018378/rtca-do-297 (accessed on 16 March 2018).
37. Vargas, V.; Ramos, P.; Mansour, W.; Velazco, R.; Zergainoh, N.E.; Méhaut, J.F. Preliminary results of SEU fault-injection on multicore processors in AMP mode. In Proceedings of the IEEE 20th International On-Line Testing Symposium (IOLTS), Girona, Spain, 7–9 July 2014; pp. 194–197.
38. Vargas, V.; Ramos, P.; Velazco, R.; Méhaut, J.F.; Zergainoh, N.E. Evaluating SEU fault-injection on parallel applications implemented on multicore processors. In Proceedings of the 6th Latin American Symposium on Circuits & Systems (LASCAS), Montevideo, Uruguay, 24–27 February 2015; pp. 181–184.
39. Ramos, P.; Vargas, V.; Baylac, M.; Villa, F.; Rey, S.; Clemente, J.A.; Zergainoh, N.E.; Velazco, R. Sensitivity to Neutron Radiation of a 45 nm SOI Multi-core Processor. In Proceedings of the Radiation Effects on Components and Systems, Moscow, Russia, 14–18 September 2015; pp. 135–138.
40. De Dinechin, B.D.; de Massas, P.G.; Lager, G.; Léger, C.; Orgogozo, B.; Reybert, J.; Strudel, T. A Distributed Run-Time Environment for the Kalray MPPA®-256 Integrated Manycore Processor. *Procedia Comput. Sci.* **2013**, *18*, 1654–1663.
41. Villalpando, C.; Rennels, D.; Some, R.; Cabanas-Holmen, M. Reliable Multicore Processors for NASA Space Missions. In Proceedings of the Aerospace Conference, Big Sky, MT, USA, 5–12 March 2011; pp. 1–12.
42. Saidi, S.; Ernst, R.; Uhrig, S.; Theiling, H.; Dupont de Dinechin, B. The shift to multicores in real-time and safety-critical systems. In Proceedings of the Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS), Amsterdam, The Netherlands, 4–9 October 2015; pp. 220–229.
43. Kalray. *MPPA ACCESSCORE V1.4 Introductory Manual*; Available under request to manufacturer when buying the MPPA DEveloper. 2015.

44. Applegate, D.L.; Bixby, R.E.; Chvatal, V.; Cook, W.J. The Traveling Salesman Problem: A Computational Study. In *Princeton Series in Applied Mathematics*; Princeton University Press: Princeton, NJ, USA, 2007; pp. 49–53.
45. Johnson, D.; Paadimitriou, C., Computational Complexity. In *The Traveling Salesman Problem*; Wiley Series in Discrete Mathematics and Optimization; Wiley and Sons: Chichester, UK, 1995; pp. 37–85.
46. Francesquini, E.; Castro, M.; Penna, P.; Dupros, F.; Freitas, H.; Navaux, P.; Méhaut, J.F. On the energy efficiency and performance of irregular application executions on multicore, NUMA and manycore platforms. *J. Parallel Distrib. Comput.* **2015**, *76*, 32–48.
47. Peronnard, P.; Ecoffet, R.; Pignol, M.; Bellin, D.; Velazco, R. Predicting the SEU Error Rate through Fault Injection for a Complex Microprocessor. In Proceedings of the 2008 IEEE International Symposium on Industrial Electronics, Cambridge, UK, 30 June–2 July 2008; pp. 2288–2292.
48. Vargas, V.; Ramos, P.; Méhaut, J.; Velazco, R. *Swifi Fault Injector for Heterogeneous Many-Core Processors*; Pontificia Universidad Católica del Ecuador: Quito, Ecuador, 2018; Volume 106.
49. De Dinechin, B.D.; Ayrignac, R.; Beaucamps, P.E.; Couvert, P.; Ganne, B.; de Massas, P.G.; Jacquet, F.; Jones, S.; Chaisemartin, N.M.; Riss, F.; et al. A clustered manycore processor architecture for embedded and accelerated applications. In Proceedings of the 2013 IEEE High Performance Extreme Computing Conference (HPEC), Waltham, MA, USA, 10–12 September 2013; pp. 1–6.