SWIFI FAULT INJECTOR FOR HETEROGENEOUS MANY-CORE PROCESSORS

INYECTOR DE FALLAS SWIFI PARA PROCESADORES HETEROGÉNEOS MANY-CORE

Vanessa Vargas[1]
Pablo Ramos[2]
Jean-François Méhaut[3]
Raoul Velazco[4]

Recibido: 15 de enero de 2018
Aceptado: 31 de enero de 2018

[1] Universidad de las Fuerzas Armadas ESPE, DEEE, Sangolquí, Ecuador (vcvargas@espe.edu.ec).

[2] Universidad de las Fuerzas Armadas ESPE, DEEE, Sangolquí, Ecuador (pframos@espe.edu.ec).

[3] Université Grenoble-Alpes, LIG Labs, Grenoble, Francia (jean-francois.mehaut@univ-grenoble-alpes.fr).

[4] Université Grenoble-Alpes, TIMA Labs, Grenoble, Francia (raoul.velazco@univ-grenoble-alpes.fr).

SWIFI FAULT INJECTOR FOR HETEROGENEOUS MANY-CORE PROCESSORS

INYECTOR DE FALLAS SWIFI PARA PROCESADORES HETEROGÉNEOS MANY-CORE

Vanessa Vargas, Pablo Ramos, Jean-François Méhaut, Raoul Velazco

Keywords: Error rate, Fault injection, Many-core, Heterogeneous, Reliability, SEE, SEU, Soft Error

Palabras clave: Tasa de error, Inyección de fallos, Many-core, Heterogéneo, Fiabilidad, SEE, SEU, Soft-error

Abstract

This work presents a fault-injection approach for evaluating the impact of soft-errors on applications running on a heterogeneous many-core processor. This evaluation is meaningful to characterize the behavior of the application implemented in advanced devices in terms of reliability. The approach is based on the principles of a mono-core fault-injection model called Code Emulating Upset (CEU) which has been demonstrated to be very efficient to predict the soft-error rate. CEU principles are extended to a heterogeneous many-core processor, in spite of its complex architecture mainly related to its memory management and inter-core communication. The selected target device is the KALRAY MPPA-256 many-core processor manufactured in 28nm CMOS technology and having a

clustered architecture. Considering, the variety of system configurations that can be implemented on a many-core processor, the present work proposes three different scenarios to illustrate the use of the approach. In the first one, a parallel version of a memory-bound application is implemented on bare-board model and configured on Asymmetric Multiprocessing Mode. The second one evaluates a distributed version of the memory-bound application running on a POSIX model. The last one assesses a distributed CPU-bound application running on a POSIX model. Results of the first scenario have been used to predict the soft-error rate of a bare-board application and have been compared to radiation experiments performed in a previous work, showing a good agreement between both techniques. This fact has motivated the extension of the approach to a more useful programming models such as POSIX. The current work retrieves results already presented in previous works by authors in order to compare them with the new ones to provide further conclusions of the proposed approach.

Resumen

Este trabajo presenta un enfoque de inyección de fallas para evaluar el impacto de soft errors en aplicaciones que se ejecutan en un procesador heterogéneo de muchos núcleos. Esta evaluación es significativa para caracterizar el comportamiento de la aplicación implementada en dispositivos avanzados en términos de confiabilidad. El enfoque se basa en los principios de un modelo mono-procesador de inyección de fallas llamado Code Emulating Upset (CEU), el mismo que ha demostrado ser muy eficiente para predecir la tasa de soft errors. Los principios CEU fueron adaptados a un procesador heterogéneo de muchos núcleos a pesar de la complejidad de su arquitectura, relacionada principalmente con la gestión de memoria y comunicación entre núcleos. El dispositivo de prueba seleccionado es el procesador de múltiples núcleos KALRAY MPPA-256 fabricado en tecnología CMOS de 28nm y que posee una arquitectura tipo cluster. Teniendo en cuenta la variedad de configuraciones de sistema que se pueden implementar en un procesador de muchos núcleos, el presente trabajo propone tres escenarios diferentes para ilustrar el uso del enfoque. En el primero, una versión paralela de una aplicación de tipo memory-bound se implementa en un modelo bare-board y se configura en modo de multiprocesamiento asimétri-

co. El segundo evalúa una versión distribuida de una aplicación de tipo memory-bound que se ejecuta en un modelo POSIX. El último evalúa una aplicación distribuida de tipo CPU-bound que se ejecuta en un modelo POSIX. Los resultados del primer escenario se han utilizado para predecir la tasa de soft errors de una aplicación bare-board y se han comparado con experimentos de radiación realizados en un trabajo previo, mostrando una buena concordancia entre ambas técnicas. Este hecho ha motivado la extensión del enfoque hacia modelos de programación más útiles como POSIX. El trabajo actual utiliza los resultados ya presentados en trabajos anteriores por los autores con el fin de compararlos con los nuevos resultados y así proporcionar mayores conclusiones del enfoque propuesto.

Introduction

The use of multi/many-core processor based platforms is becoming more frequent in computing systems due to their flexibility, high performance and redundancy capabilities. However, having complex devices that integrate several cores in the same chip, results in a potential increase of the chip sensitivity to the effects of natural radiation, especially at avionic altitudes or in space environments (Johnston, 2000). This radiation may result in transient and permanent failures, called Single Event Effects (SEE). Among these effects, Single Event Upsets (SEUs) (Baumann, 2005) are the most critical since these effects may produce changes, randomly in time and location, on bit information of a memory cell which may affect the results of an application running on these processors. For this reason, estimating the SEU error rate is a mandatory step for any application requiring reliability, no matter the operating environment.

SEU mitigation techniques can be implemented at hardware (HW) or software (SW) levels (Nicolaidis, 2010). Relevant examples are: HW and/or SW redundancy, time redundancy, bit interleaving, and error detection and correction codes (EDAC). In all the cases, the efficiency of the implemented mitigation technique must be evaluated. This can be achieved by fault simulation, fault injection, accelerated radiation testing and experiments performed in real environment. From the mentioned evaluation strategies, fault injection is the most convenient considering costs and availability issues.

Fault injection is a useful technique for validating the dependability of devices or systems (Arlat, 1990). It provides a way to improve the coverage of hardware and software testing by introducing faults in a controlled manner into system hardware or code paths in order to observe their behavior in presence of faults. In the literature, it is possible to find numerous fault-injection tools based on hardware and software methods. They can be classified into: Hardware-Based Fault Injection, Software-Based Fault Injection, Simulation-Based Fault Injection, Emulation-Based Fault Injection and Hybrid Fault Injection (Benso, 2003).

The most used fault injection technique is Software-Based Fault-Injection also called Software Implemented Fault Injection (SWIFI). This technique reproduces at the software level the errors that would have been produced when a fault target the hardware. It involves the modification of the program running on the target system to provide the ability to perform the fault injection. SWIFI is a convenient fault injection technique for evaluating applications running on COTS (Commercial-Of-The-Shelf) devices since it does not require dedicated complex hardware, gate-level netlist or RTL models that are described in hardware description languages. Faults can be injected in accessible memory cells such as registers and memories that represent the most sensitive zones of the chip. The main drawback of SWIFI techniques is their intrusiveness since they modify the program. This fact may affect the scheduling of tasks since the timing involved during the injection can disrupt the system operation (Ziade, 2012). However, if timing is not a concern, this type of fault injection can be considered as non-intrusive.

Fault injection in processor-based architectures was a topic largely addressed by the scientific community to validate the reliability of critical applications. In this work, a new software fault injection approach based on the Code Emulated Upset (CEU) principles (Velazco, 2000) have been extended to many-core processors. The CEU approach, based on interrupt signals; provides error-rate results close to those obtained in radiation tests, as demonstrated in (Rezgui, 2001) (Velazco, 2010). Its applicability to a complex processor as the PowerPC4748, allows validating this device for aeronautical applications (Peronnard, 2008). It benefits from the power of modern processors to inject the fault into a randomly selected variable of an application while it is under execution. SEU type faults are simulated randomly in time and location similarly to their occurrence in real environments. The method is applied to three different scenarios with different multi-processing modes and programming models.

State of the art and Related Work

State of the Art

Related literature shows several works dealing with fault injection techniques. In this section, only SWIFI techniques will be considered because they are suitable strategies in terms of cost and simplicity while providing a huge amount of significant data. For evaluating the effectiveness of the approach, it is necessary to take into account characteristics such as intrusiveness, target area, fault generation and software level, being the intrusiveness the most important. Table 1 summarizes the main features of the most relevant SWIFI approaches.

Table 1: SWIFI Tools Summary

Fault Injection tools	Technique	Software Level	Target area	Fault generation	Intrusiveness
FIAT	Modify kernel	O S	Memory, registers, communica-tions	Fault list	High
FTAPE	Memory/ Register modification	O S	Memory, registers	Random	High
DOCTOR	Fault injection agent	O S	Memory, registers, communica-tions	Probabilistic Past event	High
EXFI	Trace exception	O S	Memory image, code, registers	Fault list	Low
MAFALDA	Interception kernel calls	OS	Microkernel	Random	High
BOND	Interposition agents	OS	Code data sections, registers, function call parameters.	Fault list	Low
CEU	Interruptions	Bare-metal	Memory, registers	Random	Low
GOOFI	Pre-runtime Scan-chain	Bare-metal	Memory	Random	High
JACA	Computational reflection	OS	Attributes and methods Java application	Fault pattern	Low

Fault Injection tools	Technique	Software Level	Target area	Fault generation	Intrusiveness
FERRARI	System calls	OS	Memory process	Random	High
XCEPTION	Debugging	OS	Memory, data bus, address registers	Fault list	Low
FAUMACHINE	Virtualization kernel compilation	OS	Memory, disk, registers, network	Random	Low
LFI	Interception Library modif.	OS	Shared libraries	Fault profile	High
FIES	Dynamic translation	OS	Memory, registers	Fault defined XML	Low

As Table 1 shows, EXFI, BOND, CEU, JACA, XCEPTION, FAUMACHINE and FIES tools have the lower intrusiveness. Consequently, they are suitable for evaluating critical-embedded applications. Regarding the software level, it is important to consider that by definition, a SWIFI technique could not target all the sensitive zones of a given device. If we also consider the fact that performing fault-injection on applications under Operating System (OS) limits even more the target zones, it is preferred to implement fault injection at bare-metal level. This is explained because the at bare-board level it is possible to access more chip resources, thus better effectiveness is achieved. Indeed, when certification of an application running on critical-embedded system is required, it is commonly tested in bare-metal (Girbal, 2015).

From the listed techniques, only CEU and GOOFI work at bare-metal level. The main difference between them is that CEU injects faults by means of interruptions at run-time, while GOOFI injects faults at compilation-time or by means of Scan-chain. The main disadvantage of GOOFI is that it does not target processor registers. Additionally, Scan-chain fault injection works only for devices compatible with this feature. In addition, as the objective of this work is to provide an evaluation approach for multi-core and many-cores as much general as possible, CEU was selected as a base fault-injection approach.

Related Work

There are some fault-injection works for multi/many-core processors, the most relevant are summarized be-

low. Lanzaro, Pecchia, Cinque, Cotroneo, Barbosa and Silva in 2012 presented a fault-injection framework for multi-core processors for dependability analysis. Shye, Blomstedt, Moseley, Janapa Reddi, and Connors in 2009 proposed a fault tolerant technique that was evaluated by SEU fault-injection on redundant processes. The faults are injected by means of the Intel tool called Intel Pin dynamic binary instrumentation that changes one bit of a randomly selected instruction. Mansour et al in 2014 presents also a fault-injection tool at system level as the first steps to extend the CEU approach to multi-core processors (Mansour, 2014).

Lastly, in our previous work different variations of CEU-based fault-injection were developed for applications running on a Quad-core processor. These variations include a fault-injector based on fork principle to evaluate the SEU impact on parallel applications running on Symmetric Multi-Processing mode (SMP) (Vargas, 2015) and a fault-injector on bare-board based on inter-processor interrupts for application running on Asymmetric Multiprocessing mode (AMP) (Vargas, 2014).

The present work shows the extension of the proposed approaches for many-core processors where a fault-injector has been developed for distributed systems based on NoC and inter-processor interrupts. The fault injector also served as monitor application to end the application in case of timeouts.

Adopted Approach

The proposed fault-injector approach, for applications running on multi-core/many-core processors, is based on CEU principles (Velazco, 2000). One of the main objectives of CEU approach is to reproduce the effects of SEU faults on memory cells accessible by software means. The fault injection is produced when an external device interrupts the target device by an asynchronous interrupt signal. The execution of the interrupt handler in the target emulates a bit-flip in a randomly chosen memory-cell. For mono-core processors, this type of implementation does not needs a deep architectural knowledge of the target device. However, for many-core processors there are several constraints that have to be overcome to implement the fault injector due to the complexity of the device, mainly related to memory management and inter-core communications.

Fault Injection Strategy

In the case of multi/many-core processors, it is possible to benefit of the multiplicity of cores for using some of them as fault injector while the others run the chosen application. Figure 1 illustrates the proposed fault-injector approach when one core is used as fault injector.

Figure 1. Fault Injector based on CEU Principles

For multi/many-core processors, this strategy considers a master-slave scheme where the master core performs as fault injector whereas the slave cores execute the selected application. The master core initializes the data that is going to be used by the other cores and sends a message through an inter-processor interrupt to indicate the slave cores to start the execution of the application. While the application is running on the slave cores, the master core performs the fault injection. It randomly selects the target core, the injection instant (in terms of clock cycles), the address (global array index) and the bit to be altered (Vargas et al. 2014,)

To inject a bit-flip in the target, the following tasks are done:

- Reading the content of the target memory cell
- Performing an XOR operation with an appropriate mask value that contains a "1" for the bits that are going to be flipped and "0" elsewhere.
- Writing the corrupted value to its original location.
-

When slave cores finish the execution of the application, the master core compares the resulting data with a set of correct results previously obtained.

The consequences of the fault-injection are classified as follows:

- Silent fault: it occurs when the injected fault does not cause any consequence in the result of the program (e.g., typical silent faults are those affecting data never used or data already used by the program).
- Erroneous result: the results of the program are not the expected ones.

- Exception: the program halts. It is primarily caused by faults injected on critical registers. A hang is a type of exception that crashes the system.
- Time-out: When the program does not respond after duration equal to the worst-case execution time.

Before starting the fault-injection campaigns, it is neccesary to determine the number of cycles required to execute the selected application. It is done in order to know the range of time in which the fault injection should be performed. Also, it serves to determine the time-out value. If the multi/many-core evaluated processor operates in stand-alone mode, the *monitor* functions of the application to determine *time-outs* and *hangs* are accomplished by the *master* core. For the other cases, when the multi/many-core processor works as a co-processor, the *monitor* functions are accomplished by the *host* processor.

The error-rate of an application (τInj) is derived from the fault injection. This quantity is defined as the average number of injected faults needed to produce an error in the result of the application. For obtaining the error-rate erroneous results, exceptions and time-outs are considered.

$$\tau_{\text{inj}} = \frac{\text{Number of errors}}{\text{Number of injected faults}}$$

Many-core Processors Considerations

Before designing a fault-injector for many-core processors, it is important to take into account the complex architecture of theses devices and some specific software issues.

Architectural Issues

Typically, the Reduced Instruction Set Computer (RISC) is implemented on multi/many-core processors. There is an instruction-level paralleling implemented in the core architecture to increase the speed up of processing based on pipelined. Inside the chip, each core acts as an independent processor. The OS manages the internal resources and its scheduler assigns the processes to the cores. Reference (Vajda, 2011) details the architectural issues of both multi-core and many-core processors. Regarding the many-core processor, the large increase in the number of cores implies a considerable evolution in the architecture of the device. The main constraints are related to the inter-core communications and the management of memory resources and I/O devices. Regarding the inter-core communication mechanisms, traditionally in a multi-core processor each core communicates by a common shared bus with the other cores; however, in many-core processor the use of NoC is essential. On the other hand, for

managing efficiently the memory resources some approaches are proposed such as ring, mesh and crossbar interconnections. Figure 2 illustrates a many-core processor architecture.

Regarding memory management, multi-core and many-core processors include the use of cache memories as fast memories to reduce the memory access time by minimizing the access to main memory. In fact, its use increases significantly the performance of the system, so several levels of caches are proposed. In addition, these processors use shared and distributed memory models. In shared memory models, there is one common shared memory accessed by all processors while in distributed memory, each processor or group of processors has its own local memory. Typically, multi-core processors use shared memory model and many-core processors use a mixed model. Some manufacturers group several cores in clusters. Inside each cluster, they implement a shared memory.

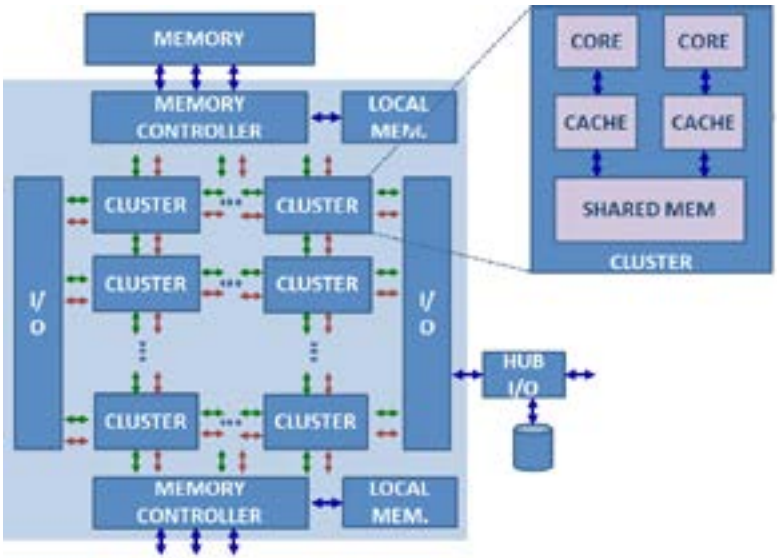

Figure 2. Many-core processor architecture

Software Issues

To exploit massive paralleling, the application developers have to move from serial to parallel execution model and choose the appropriate system configuration to achieve maximum concurrency and consequently performance improvement. In addition, some capabilities need to be isolated to guarantee the dependability of the system. This implies that the software designer has to take into account issues such as multi-processing mode, programming model and the access level to hardware resources to better deploy the application.

Regarding the multi-processing modes, there are two principal models: a) Symmetric Multi-Processing (SMP) and b) Asymmetric Multi-Processing (AMP). A programming model refers to the manner that the software assigns the application tasks to the hardware and the level of abstraction that the programmer has from hardware architecture. Finally, multicore and many-core processors provide different levels of protection that allow the software to access to hardware resources and configurations. These levels are also called privilege modes.

Targeted Zones

The targeted zones could be cache memories, general purpose and special function registers belonging to each processor, as well as shared memory of the multi/many core device. In order to perform fault injection in memory, the variables to be used by the application are placed in the internal memory of the device. In this manner, the variables can be modified at any time by each one of the processor cores. In the case of cache memories, since programmers cannot inject faults directly to them, the fault-injection is performed in the main memory and corrupted data are retrieved by cache. Regarding fault injection in processor registers, only accessible registers can be modified. Since master core has no access to other cores registers, it can execute an indirect fault injection via an inter-core interruption to the selected core, in which the interruption handler launches a code that targets accessible registers emulating bit-flips as previous described. It is important to note that modifying these registers may cause critical failures in the program execution.

Limitations of the Strategy

Concerning fault injection in processor registers, only accessible registers can be modified. As the fault-injection strategy uses one core as fault injector, it is reasonable to think about the intrusiveness of the approach. For multi-cores having few cores, it is clear that a significant part of the sensitive area corresponding to the fault injector core is not targeted and thus, it should

not be taken into account for the estimation of the error rate. On the other hand, when working with many-cores having hundreds of cores, the sensitive area corresponding to the fault injector is negligible. In addition, devices implementing shared memory concentrate most of the sensitive area outside the processor cores. Consequently, the presented fault injection strategy is valid to evaluate the sensitivity of a given application through the estimation of its error rate.

Multiple Case Studies

To illustrate the proposed approach, multiple case studies were considered:

A. A many-core execution with minimal use of NoC services.
B. A many-core execution with intensive use of NoC services.

The evaluation considers the use of AMP and SMP modes as well as three types of programming models:

a. Bare-metal: no OS is used, the programmer uses the Board Support Package (BSP) functions provided by the manufacturer to access hardware resources. There is no abstraction layer from hardware architecture. All the configurations and the distribution of the tasks must be programmed. The programmer has the control of each function.
b. Low Level: no OS is used; however, there are a set of libraries that can be used. It provides a little abstraction from hardware architecture. The functions and commands used are closely related to the specific device capabilities on which the application is implemented.
c. POSIX: is a low-level Application Programming Interface (API) defined on top of OS. It allows the control of parallel tasks (threads) where the programmer must control the management of threads. It is independent of the hardware architecture.

This work proposes the use of two types of parallel applications: *a CPU-bound* and *a memory bound*. In a CPU-bound application, the computation time is the bottleneck in the performance evaluation. On the contrary,

in a Memory-bound application the execution time depends primarily on the time needed for accessing memory. The selected CPU-bound application was the Traveling Salesman Problem (TSP), a Non-deterministic Polynomial (NP) hard problem very used for evaluating computing system optimization (Applegate, 2007). This application aims to find the shortest possible route to visit n cities, visiting each city exactly once and returning to the departure city. In a formal way, the problem is represented by a graph of the cities including the distance among them, where the cost $c(i,j) \geq 0$ represents the distance from city i to j. The goal is to find a Hamiltonian cycle with minimum cost for the travel.

On the other hand, the Matrix Multiplication (MM) was chosen as memory bound application. The MM is widely used for solving scientific problems related to linear algebra, such as systems of equations, calculus of structures, determinants among others. Also, the parallel version of Matrix Multiplication (MM) is one of the most fundamental problems in distributed and High Performance Computing (HPC).

Concerning the target device, this approach was applied to the MPPA Developer which is based on the KALRAY MPPA-256 many-core processor, this device was selected because:

1. The manufacturing technology is advanced CMOS 28nm.
2. The architecture and the number of cores is similar to the many-core processor ShenWei SW26010 (260 cores). The latter is the base of the Sunway TaihuLight Supercomputer, which was ranked in the first position of the TOP500 list on November 2017. More details of the SW26010 can be found in reference (Dongarra, 2016).
3. The MPPA many-core was considered by semiconductor manufacturers and the real-time community for discussing the challenges of using many-core processors in embedded systems. In addition, the project CAPACITES (Calcul Parallèle pour Applications Critiques en Temps et Sûreté) that gathers French academics and industrial partners uses this device to analyze the possibility of using many-cores for critical real-time embedded systems.

Target Device

The KALRAY MPPA-256 many-core processor is a 64 bit many-core processor manufactured in TSMC CMOS 28nm technology. The processor operates between 100 MHz and 600 MHz, for a typical power ranging between 15 W and 25 W. Its peak floating point perfor-

mances at 600 MHz are 634 GFLOPS and 316 GFLOPS for single and double-precision respectively. The second version of this processor, called Bostan, is considered in this work. This device is based on an array of 16 Compute Cluster (CC) and 2 I/O clusters that are connected to the 32 nodes of Network-on-Chip (NoC) with a toroidal 2D topology. The 16 inner nodes of the NoC are connected to the CC while the 16 peripheral nodes are connected to the I/O subsystems. The NoC is comprised of 2 parallel networks: the Data NoC (D-NoC) that is optimized for bulk data transfers while the Control NoC (C-NoC) is optimized for small messages at low latency (Kalray, 2016). Figure 3 illustrates an overview of the device.

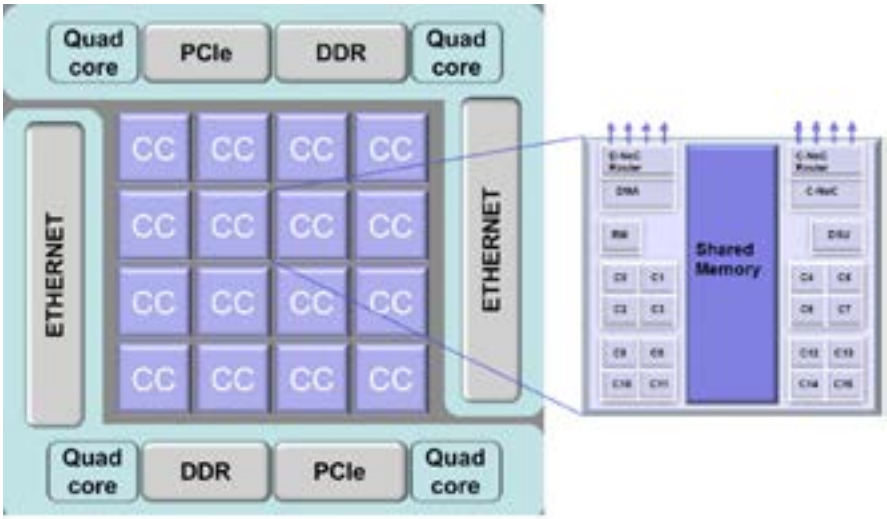

Figure 3. MPPA-256 many-core processor components

The Multi Purpose Processing Array(MPPA) is a distributed memory system. Each Compute Cluster (CC) is built around a multi-banked local Static Memory (SMEM) of 2MB shared by 17 identical Very Long Instruction Word (VLIW) cores: 16 Processing Engine (PE) + 1 Resource Manager (RM) without cache coherency. This configuration creates an interconnection with high bandwidth

and throughput between PEs. The Pes are dedicated to execute the application code while the RM is in charge of managing the NoC interfaces by means of dedicated event lines and interrupts. Each IO cluster features 2 I/O subsystems. Each I/O subsystem comprises 4 network interfaces and a quad-core of RMs in a Symmetric Multi-Processing (SMP) configuration connected to two main banks of 2 MB and to a 4-lane Interlaken controller. One of the I/O subsystem in the IO cluster is connected to a DDR interface (accessto2GB) and 8 lane PCIe controller. The other is connected to a quad 10Gb/s Ethernet controller.

The MPPA integrates 256 PE cores and 32 RM cores. Both types of cores are based on the same VLIW 32-bit/64bit architecture. The VLIW core implements separate 2-way associative instruction and data cache memories. There is no hardware cache coherency mechanism between cores, nor between data cache and instruction cache. However, to enforce memory coherency, several software mechanisms are available to programmers.

Sensitive Zones

The main memory areas of the many-core processor are covered by error protection mechanisms except the instruction and data cache memories of the VLIW core that are protected by parity. The SMEM of the clusters interleaves bits of 8 adjacent 64-bit words which allow localized errors spread as multiple Single ECC (SECC) errors. They are detected and corrected on the fly. The NoC router queues (512 of 32bit flits each) are also protected by ECC. Note that SECC errors are silently corrected while Double ECC (DECC) errors are signaled. On the other hand, registers do not implement any protection mechanism. The VLIW core includes General Purpose Registers (GPRs) and System Function Registers (SFRs). Outside the processor, the MPPA comprises different types of specific registers for controlling DMA, D-NoC ,C-NoC, cluster power controller, trace, debug, and the different I/O.

Programming MPPA Issues

In order to program the MPPA Developer, the application is commonly divided into a HOST part and a 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 PCIe expansion bus (Dinechin, 2016) (Dinechin, 2016b). The MPPA provides a great configuration flexibility. It allows running independent applications per cluster or to program multi-cluster applications in a classic master/slave scheme, where the IO cluster performs as the master. Communication between the HOST and the MPPA is achieved using specific drivers provided by the manufacturer. For inter-clusters communication, it is also provided a library with a set of functions for data exchanges through the MPPA Network-on-Chip (NoC). The main challenges that 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 (2MB for Operating System (OS), code and data).

Case-Study A: Many-Core Execution with Minimal Use of NoC Services

This case-study aims at evaluating the sensitivity of the internal computing-cluster resources. To do this, the many-core processor is configured in bare-metal where each cluster executes independently the same application. Results presented in our previous work (Vargas, 2017) show that during the radiation test campaigns on the MPPA, no errors were produced in SMEMs of the clusters since they implement ECC and interleaving. Consequently, this work only considers fault injection in processors' registers.

System Configuration

The many-core was configured in AMP mode and has implemented a bare-metal application to minimize the use of libraries. The dynamic response of the device was evaluated through the execution of a testing application that must accomplish the following characteristics: (1) intensive use of the cluster resources, (2) code and data size maximum of 2 MB, (3) evenly load distribution among PEs, (4) enough execution time to ensure all PEs running in parallel.

In general, there are no shared resources between compute clusters. However, the NoC resources are used for inter-cluster communications when the IO cluster spawns the executable code to the compute clusters, and when the clusters log the results. The code is loaded by means of the JTAG in the SMEM of the IO cluster 0. This cluster then spawns the same executable into the 16 compute clusters and orders them to start the execution of the program. Within each cluster, the RM core wakes-up the 16 PE cores, and each one of them starts the execution of the application.

Benchmark Details

The application to be tested within each compute cluster of the device is an assembler optimized version of a cooperative 256×256 matrix multiplication. The matrix multiplication is performed 256 times, and the result C is the summation of these computations as stated in (2). 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. For a 256 matrix size, it takes around 1M IO cycles to spawn 1 cluster. Since clusters are spawned one after another, cluster 15 starts execution around 15M IO cycles after cluster 0.

$$C = \sum_{1}^{n} A \, x \, B$$

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 compute cluster is configured in Asymmetric Multi-Processing (AMP) mode and the computational work is distributed evenly among the processing cores, so each PE belonging to the cluster computes 1/16 of the cluster result. The synchronization of the computation is done by events between the RM and the PE cores. The RM wakes up the 16 PEs and sends a notification to each one to start the computation. Then, it waits for a notification from each PE indicating the work was done. Once all PEs computations have finished, the RM core compares the result matrix with a reference result-matrix E, and reports any mismatch including the associated addresses and values. Then, the matrix C is filled up with zeros and the PEs start again the computation. The program executes continuously the same algorithm in each cluster along the test Fault-injector details

This case-study considers one fault injector per cluster due to each CC performs the application independently of the others. This work only considers Single Event Upset (SEU) emulation where one SEU per cluster and per run is injected. The fault injection procedure is repeated several times in order to obtain enough amount of samples to calculate the injection error-rate τinj. The fault-injection campaign is devoted to inject faults in General Purpose Registers (GPRs) and SFRs of the compute cluster's cores (PEs or RM). Since some SFRs are non writable by software means, only 34 SFRs of 51 SFRs were targeted. Among the targeted SFRs, the most critical ones are the 8 registers saved during context switching: Shadow Program Counter (SPC), Shadow Program Status (SPS) Return Address (RA), Compute Status (CS), Processing Status (PS), Loop Counter (LC), Loop Start Address (LS) and Loop Exit Address (LE).

Case-study B: Many-core Execution with Intensive Use of NoC Services.

This case-study aims at evaluating the sensitivity of the MPPA when massive paralleling is used. For that, a CPU-bound (TSP) and a memory-bound (MM) were implemented as parallel inter-cluster applications. The programming model considers different type of communications. For the intra-computing cluster, the directives were POSIX and the inter-cluster communication was done by NoC services. Due to certain constraints, a Low-level configuration was used in the IO cluster. The dynamic response of the TSP and the MM were only evaluated through fault injection in processors registers, since the SMEM of the MPPA many-core implement effective protection mechanisms.

System Configuration

The many-core processor was configured in a typical master/slave scheme for running parallel multi-cluster applications. The master runs on the IO cluster while the slaves run on the Compute Cluster (CC). In this case study, the application was configured without a code running on the HOST . The MPPA part was loaded through the JTAG port to the IO-DDR0. Figure 4 shows the booting process for this type of configuration.

Figure 4. MPPA booting for a POSIX application model

For this scheme, 3 RM cores of the IO-DDR0 were configured:

- The RM1 is the master of the application.
- The RM3 is the fault injector.
- The RM0 coordinates the actions between the application and the fault injector. Also, it is in charge of configuring all the inter-cluster communication.

At present, the Real-Time Executive for Multiprocessor Systems (RTEMS) OS which runs on IOs is only capable of using one RM. For this reason, the Low-level programming model was selected for the IO cluster. The IO was configured in bare-metal without OS, including the Virtual Board Supporting Package (V-BSP) and LibNOC libraries. Figure 5 allows a better understanding of the software configuration details. It illustrates the several abstraction layers included in the software stack of the MPPA ACCESS-CORE SDK.

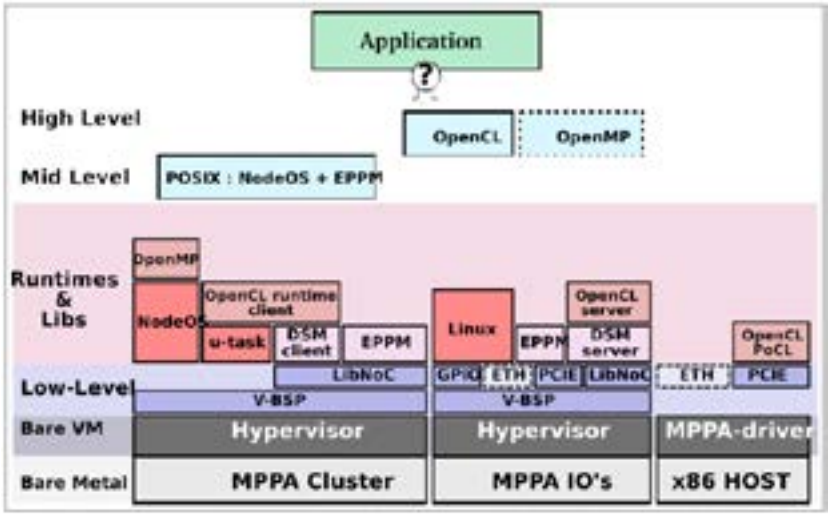

Figure 5 Kalray software stack (Kalray,2016)

These libraries can co-exist independently, and the user can mix and match the libraries he wish to use (Kalray, 2016). On the other hand, the intra-cluster application was configured with the POSIX model. The main process runs on the RM1 and is responsible to spawn the sub-processes from the IO to the target

CCs. Each sub-process is a multi-thread program based on POSIX. The CCs were configured for using the Node OS, an OS POSIX standard. The Node OS implements an asymmetric multi-processing architecture where the RM performs the kernel routines and NoC interface services, while the PEs run one user thread per PE. All the inter-cluster communications are done through the NoC using the MPPA inter process communication (MPPA IPC). This library contains the routing functions, an API for the power-on and spawning of CCs and the communication primitives: the classic POSIX IPC and some specific primitives for the MPPA many-core (Kalray, 2016). Since there is no shared memory between clusters in the MPPA architecture, the implementation of an inter-cluster application requires distributed algorithms.

Benchmark details

The implementation of both applications allows configuring the number of CCs from one to four as well as the problem size. Table 2 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 2: Standard execution time for different configurations of TSP on the MPPA

Nb cities	1 cluster		2 clusters		3 clusters		4 clusters	
	[Gcycles]	[s]	[Gcycles]	[s]	[Gcycles]	[s]	[Gcycles]	[s]
16	30.8	77.0	16.2	40.5	11.3	28.3	8.6	21.5
17	188.8	472.0	102.8	257.0	71.2	178.0	58.1	145.2
18	521.9	1304.2	260.3	650.8	188.2	470.5	159.8	399.5

For performing both applications, this case-study considers 4 CCs as slaves of the RM1 of the IO cluster. Thus, 5 RMs and 64 PEs are involved on the application itself plus the RM0 that starts, monitors, and manages the messages received by the NoC. 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. It is important to note that, if any cluster does not finish its job in the normal execution time, the RM0 logs an timeout and ends the application. The TSP was configured to

solve a 17 cities problem in order to have enough processing time to observe errors within a reasonable overall simulation time.

Regarding the code performed by the slaves, it corresponds to a multi-threading TSP version. Each cluster performs the slave code almost independently of the others. It has its own minimum distance and path variables. The only moment when they 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.

On the other hand, the MM is practically the same cooperative 256 × 256 matrix multiplication described in the case-study A. The main difference is that the computation is iterated 8192 times. 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 of the cooperative MM used in the previous case-study.

Fault-injector Details

This case-study implements one core of the device as fault injector, being the RM3 core of the IO cluster 0 that performs this function. Fault-injection campaigns target only processor registers as stated in the previous case-study. In order to interrupt the targeted core, it is used the portal primitive. Thus, the configuration of one portal per cluster (portal_fi) is needed for fault-injection purposes. In the case of the master, the RM0 is in charge of its configuration.

To emulate a bit-flip in the selected register, the fault injector must interrupt the selected core. For achieving this goal, at the random instant, 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 process causes an interruption of the RM which immediately assigns the execution of the interruption handler to any one of the PEs. The assigned PE reads the information in the portal_fi, that contains the selected core, register and bit to change. 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.

The fault injection campaigns of both applications are devoted to inject faults in GPRs and 15 SFRs of the PEs belonging to the compute clusters. The targeted SFRs are: (1) the 8 registers saved during context switching: SPC, SPS RA, CS, PS , LC, LS and LE, (2) the 6 event registers, and the Performance Monitor Control (PMC) register.

The other SFR could not be targeted due to the following reasons: (1) 5 SFRs are hardwired and 2 are unused by the current Bostan version of the MPPA processor, (2) the 4 performance monitor registers cannot be modified by software, (3) the 5 registers used by the debugger can only be accessed in debug mode, (4) the exception vector is non-writable by the user, and (5) the other SFR are managed by the OS so when the user changes any of them, the OS produces always an exception.

Experimental Results

Experimental Evaluation by Fault Injection Case A

In these experiments, the SEU faults were injected at a random instant within the nominal duration of the executed program which was around 5.3×10^8 clock cycles. The fault-injection results of this case study were already presented in our previous work (Vargas, 2017). Retrieving this information, Table 3 summarizes the fault injection campaign where 94316 faults were injected in the GPRs and accessible SFRs.

Table 3 : Results of the fault-injection campaigns for MM-AMP-MPPA scenario

Targeted Registers	Silent Faults	Erroneous results	Timeouts	Exceptions
GPRs	36472	16387	6678	1996
SFRs	22745	2034	6365	1639
TOTAL	59217	18421	13043	3635

From these results, it was calculated the application error rate applying (3), and considering as errors the erroneous results, timeouts and exceptions.

$$\tau_{inj} = \frac{\text{Number of errors}}{\text{Number of injected faults}} = \frac{35099}{94316} = 37.21\%$$

This result shows that 37.21% of the injected SEUs in the accessible registers cause errors in the application. Since registers have no protection mechanisms, this campaign is very useful to emulate the behavior of the application in presence of SEUs.

Experimental evaluation for case B

The following experiments only consider the emulation of one SEU per execution, so that one SEU is injected in the targeted register belonging to one of the 64 PE cores used by the application. The fault is generated at a random instant within the nominal duration of 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, being the random instant, core, register and bit additional arguments of the main function. Table 4 provides details about the two fault injection campaigns.

Table 4: Fault Injection Campaign details for case B

Application	Standard execution time		Runs per campaign
	[Gcycles]	[s]	
MM	4.55	11.4	72497
TSP	58.06	145.2	8417

Table 5 shows a general overview of the fault-injection campaigns on the TSP and MM applications. These results confirm the intrinsic fault-tolerant capability of the TSP application. From these results, the error-rates of both applications were calculated by using equation (3), being 2.61% for the TSP and 14.38 % for th MM. The erroneous results, timeouts and exceptions were considered as errors. Figure 6 illustrates the details of the error consequences in both application during fault-injection campaigns.

Table 5: Results of the fault-injection campaigns on case B

Application	Silent Faults	Erroneous results	Timeouts	Exceptions
MM	62071	5215	112	5099
TSP	8197	2	21	197

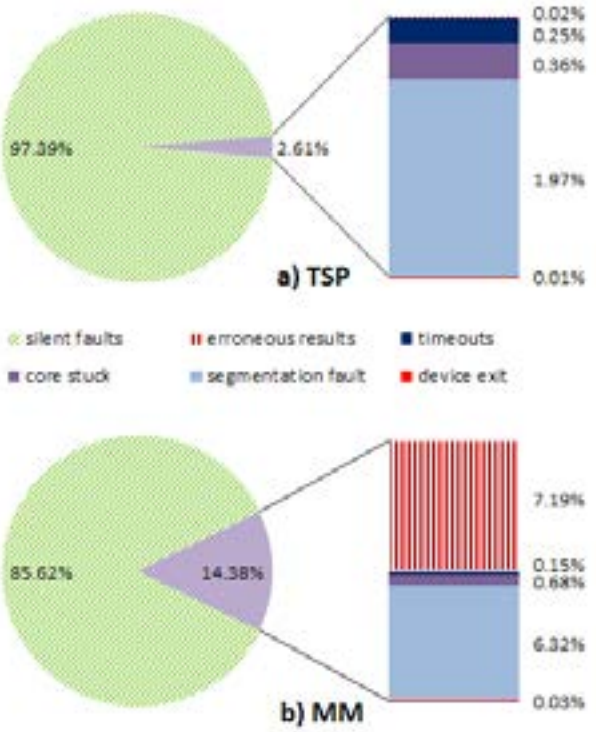

Figure 6. Fault-injection consequences on MPPA when targeting registers

Three types of exceptions were produced: (1) the PE targeted by the fault-injector was completely stuck and does not respond anymore to the JTAG requests "core stuck", (2) the bit-flip caused the core tried to access a memory not allocated producing a "segmentation fault", and (3) the MPPA device stopped its execution and produced an exit of the process "device exit".

As expected, during the fault-injection campaigns, it was observed that the critical registers are application dependent. For instance, in the case of TSP the most critical SFRs registers were: the Shadow Program Counter, the Shadow Program Status and the Return Address. Also, some of the errors were produced by bit-flips in Processing Status and Loop Counter. Concerning the GPRs, the

most critical was the GPR13 followed by GPR43, GPR54 and GPR60. On the other hand, in the MM application, the most critical SFRs were: the Shadow Program Counter, the Shadow Program Status, the Return Address, the Loop Start Address and the LE. Regarding the GPRs: the most critical was the GPR13 followed by GPR10, GPR16, GPR34, GPR35, GPR39, GPR42, and GPR50. Concerning the type of errors, the most critical one is the erroneous result since this error is ignored by the application which considers it as a valid result. The timeouts and exceptions are not so critical since they are detected and the application/system is able to manage them. For instance, this case-study implements an application with a timeout principle, so if the execution time of the application is greater than the standard execution time, the application is finished. In addition, the 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.

Discussion of the overall results

In this work, three scenarios were evaluated:

- A parallel MM running independently on each computing cluster congured on a bare-metal system with shared memory. Two scenarios were proposed: cache memories enabled and cache memories disabled.
- A MM running on a distributed system using a master/slave scheme, where the master runs on the IO Cluster, while each one of the 4 slaves runs in a Compute Cluster. Each slave is an AMP system programmed with POSIX which computes a part of the total result.
- A TSP running on a distributed system using a master/slave scheme, where the master runs on the IO Cluster, while each one of the 4 slaves runs in a Compute Cluster. Each slave is an AMP system programmed with POSIX which computes a part of the total result.

Regarding fault-injection experiments, the used and targeted resources per scenario are summarized in Table 6. Unfortunately, it was not possible to target all the resources used by each scenario for the following reasons: (1) some registers are not-writable by software means, (2) the RM interrupt routine should not be modified by the programmer to guarantee a correct operation of NoC services. On distributed applications, the communication between clusters is achieved by the use of NoC libraries provided by the manufacturer which use interrupts. Thus, it is not possible to overwrite the interrupt routine to program inter-processor interrupts which allow injecting faults in the RM.

Table 6: Summary of the different fault-injection scenarios evaluated on the MPPA

Scenario	Registers Targeted per core	Cores Targeted per CC cluster	Resources used by the application	Resources Targeted
MM-Bareboard	64 GPRs + 34 SFRs	RM + 16 PEs	CCs	CCs
MM-Posix	64 GPRs + 15 SFRs	16 PEs	IO + CCs + NoC	CCs
TSP-Posix	64 GPRs + 15 SFRs	16 PEs	IO + CCs + NoC	CCs

On systems configured in bare-metal, it is possible to target more registers by software means since there are no restrictions caused by the use of an OS. A distribution of the consequences of fault-injection campaigns on the tested applications when targeting registers is shown in Figure 7.

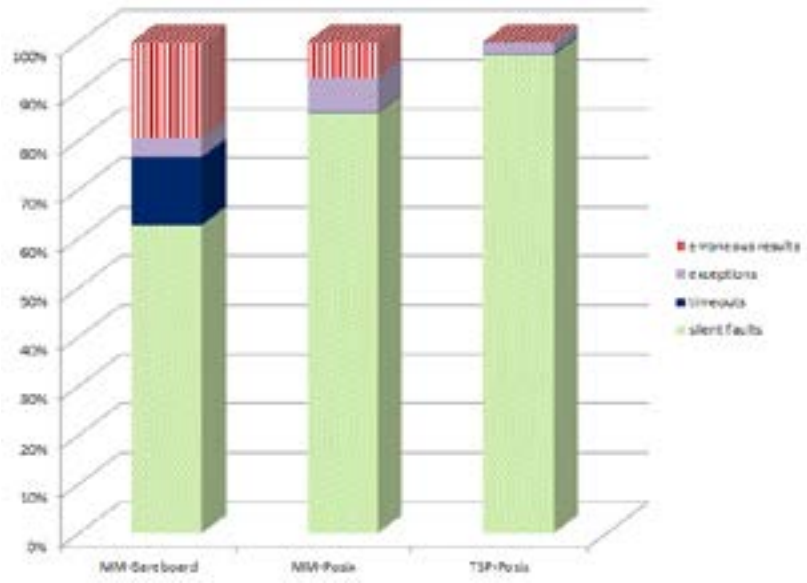

Figure 7. Distribution of fault-injection consequences on MPPA for different scenarios

Conclusions

The fault-injector approach has been proved to be useful to predict application error-rate due to its reasonably closeness to the error-rate measured in radiation experiments (Vargas, 2016). Consequently, in spite of the hardware complexity of the many-core processor, the mentioned work support the relevance of the use of the CEU approach to estimate the error-rate of applications implemented in such devices.

In spite of the limitations of the approach concerning targeted zones, the evaluation through this method gives a clear idea of the SEU sensitivity of the application. From the obtained results, MM-Posix scenario seems to be less sensitive than MM-Bareboard. However, it can not be concluded based only on the results obtained from the fault-injection campaigns due to the considerable underestimation of errors in POSIX scenarios. Note that IO clusters and NoC resources used in POSIX cannot be targeted by fault-injection means. Consequently, further radiation experiments are needed: (1) to evaluate at what extent this fault injection limitation affects the results, considering that RMs are in charge of manage the OS, and (2) to provide a fair comparison of both scenarios. Furthermore, to validate the approach, it is necessary to apply the approach to other many-core processors and system configurations and confront the results to radiation experiments.

Acknowledgments

This work was supported in a part by the Universidad de las Fuerzas Armadas ESPE, by the Secretaría de Educación Superior, Ciencia, Tecnología e Innovación del Ecuador (SENESCYT) STIC-AmSud - EnergySFE project PIC-16-ESPE-STIC-001 and by the French authorities through the "Investissements d'Avenir" program (CAPACITES project). Authors thank to Stéphane Gailhard from the Societé Kalray for his valuable contribution to solve the MPPA programming issues

References

Applegate, D. L., Bixby, R.E., Chvatal,V. and Cook, W.J.(2007) The Traveling Salesman Problem: A Computational Study, pages 49-53. *Princeton University Press*, Princeton, USA, September.

Arlat, J.et al. (1990) "Fault Injection for Dependability Validation: A methodology and Some Applications," *IEEE Trans. On Soft. Eng*. Vol. 16, No 2, pp. 166-182.

De Dinechin, B. D., De Massas, P. G., Lager, G., Léger, C., Orgogozo, B. Reybert, J., and Strudel, T., (2013)."A distributed run-time environment for the kalray MPPA-256 integrated manycore processor," Procedia Computer Science, vol. 18, pp. 1654 – 1663, 2013 International Conference on Computational Science.

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., and Strudel, T.(2013) "A clustered manycore processor architecture for embedded and accelerated applications," in 2013 *IEEE High Performance Extreme Computing Conference (HPEC)*, Sept. , pp. 1–6.

Baumann, R. (2005) "Soft Errors in Advanced Computer Systems", *IEEE Design and Test of Computers*, vol 22, n° 3, pp. 258-266.

Benso, A. and Prinetto, P. (2003) Fault Injection techniques and tools for embedded systems reliability evaluation, USA: Kluwer Academic.

Dongarra, J. (2016) Report on the Sunway TaihuLight System, June.

Ferrel, T. and Ferrel, D. (2014) "RTCA DO-178B/EUROCAE ED12B." *Digital Avionics Handbook*, Third Edition, 195206.

Girbal, S., Pérez, D. G., Le Rhun, J., Faugère, M., Pagetti, C. and Durrieu, G. (2015) "A complete toolchain for an interference-free deployment of avionic applications on multicore systems," 2015. *IEEE/AIAA 34th Digital Avionics Systems Conference (DASC)*, Prague, pp. 7A2-1-7A2-14.

Johnston, A. H. (2000)"Scaling and Technology Issues for Soft Error Rates", *Proceedings of 4th Annual Research Conference on Reliability,* Stanford University, October.

Kalray (2015) "MPPA ACCESSCORE V1.4 Introductory Manual,". V. Vargas, P. Ramos, V. Ray, C. Jalier, R. Stevens, B. Dupont de Dinechin, M. Baylac, F. Villa, S. Rey, N. E. Zergainoh, J. F. Méhaut, and R. Velazco, "Radi-

ation Experiments on a 28nm Single-Chip Many-core Processor and SEU error-rate prediction,," *IEEE Trans. Nucl. Sci.*, vol. 99, pp. 1 – 8, Dec. 2016.

Kalray(2016) "MPPA-256 Bostan Cluster and I/O Subsystem Architecture".

Lanzaro, A., Pecchia, A., Cinque, M.,.Cotroneo, D.,Barbosa, R. and Silva, N. A. (2012) Preliminary Fault Injection Framework for Evaluating Multicore Systems, pages 106-116. *Springer Berlin Heidelberg*, Berlin, Heidelberg, September.

Mansour, W., Ramos, P., Ayoubi, R. and Velazco R. (2014) "SEU fault-injection at system level: method, tools and preliminary results". *15th Latin American Test Workshop – LATW*, pp. 1-5.

Nicolaidis, M.(2010) "Soft Errors in modern electronic systems", *SPRINGER* Ed., ISBN 978-1-4419-6992-7.

Ramos, P.(2017) "Evaluation of the SEE sensitivity and methodology for error rate prediction of applications implemented in Multi-core and Many-core processors." [Online]. Available: http://tima.univ-grenoble-alpes.fr/tima/en/mediatheque/ PhDthesisresult_id452.html, France, ISBN: 978-2-11-129226-0, April.

Ramos, P., Vargas, V., Baylac, M., Villa, F., Rey, S., Clemente, J.A., Zergainoh, N.E. Méhaut, J.F., and Velazco, R.(2016) "Evaluating the SEE sensitivity of a 45nm SOI Multi-core Processor due to 14 MeV Neutrons," IEEE Trans. Nucl. Sci., vol. 63, pp. 2193 – 2200, Aug.

Peronnard, P., Ecoffet, R., Pignol, M., Bellin, D. and Velazco, R. (2008) "Predicting the SEU Error Rate through Fault Injection for a Complex Microprocessor," *in Proc. 2008 IEEE International Symposium on Industrial Electronics*, September, pp. 2288–2292.

Velazco, R., Foucard, G. and Peronnard, P.(2010) "Combining Results of Accelerated Radiation Tests and Fault Injections to Predict the Error Rate of an Application Implemented in SRAM-Based FPGAs," *IEEE Trans. Nucl. Sci.*, vol. 57, pp. 3500–3505, December.

Velazco, R., Rezgui, S. and Ecoffet, R.(2000) "Predicting Error Rate for Microprocessor-Based Digital Architectures through C.E.U. (Code Emulating Upsets) Injection,", *IEEE Trans. Nucl. Sci.*, vol. 47, pp. 2405–2411, December.

Rezgui, S., Velazco, R., Ecoffet, R., Rodriguez, S. and Mingo,J.(2001) "Estimating Error Rates in Processor-Based Architectures," *IEEE Trans. Nucl. Sci.*, vol. 48, pp. 1680–1687, December.

Shye, A., Blomstedt, J., Moseley, T., Janapa Reddi, V., and Connors, D. A. (2009). PLR: A Software Approach to Transient Fault Tolerance for Multicore Architectures. *IEEE Trans. On Dependable And Secure Computing*, 6(2):135-148, April.

Vajda, A.(2011) Multi-core and many-core processor architectures. *In Programming Many-Core Chips*, pages 9-43. Springer.

Vargas, V.(2017) "Software approach to improve the reliability of parallel applications implemented on multi-core and many-core processors" [Online]. http://tima.univ-grenoble-alpes.fr/tima/fr/mediatheque/PhDthesisresult_id453.html Available: France, ISBN: 978-2-11-129227-7, April.

Vargas, V., Ramos, P., Ray, V., Jalier, C., Stevens, R, Dinechin, B. D. D., Baylac, M., Villa, F., Rey, S., Zergainoh, N. E., Méhaut, J.F., and Velazco, R.(2017) "Radiation experiments on a 28 nm single-chip many-core processor and seu error-rate prediction," *IEEE Trans. Nucl. Sci.*, vol. 64, pp. 483–490, January.

V. Vargas, P. Ramos, W. Mansour, R. Velazco, N. Zergainoh, and J. Mehaut,(2014) "Preliminary results of SEU fault injection on multicore processors in AMP mode," *in Proc. IEEE 20th International On-Line Testing Symposium (IOLTS)*, pp. 194–197, September.

Ziade, H., Ayobi, R. and Velazco, R. (2004) "A Survey on Fault Injection Techniques", *The International Arab Journal of Information Technology*, Vol 1, no 2, July, pp. 1-6.