A comparison between Hardware and Software Solutions for Resource Partitioning in Multicore-based Mixed Criticality Applications

Stefano Esposito, Sehriy Avramenko, Massimo Violante
Dipartimento di Automatica e Informatica
Politecnico di Torino
Torino, Italy
stefano.esposito@polito.it
sehriy.avramenko@polito.it
massimo.violante@polito.it

Marco Sozzi, Massimo Traversone
Selex ES (now Finmeccanica)
Nerviano, Italy
marco.sozzi@finmeccanica.it
massimo.traversone@finmeccanica.it

Marco Binello, Marco Terrone
Alenia Aermacchi (now Finmeccanica)
Torino, Italy
marco.binello@finmeccanica.it
marco.terrone@finmeccanica.it

***Abstract*—The paper proposes a comparison between hardware and software solutions for resource partitioning in the scenario of a multi-core based mixed criticality application. A reference avionic application has been implemented in two versions: one using a software partitioning solution and one using a hardware partitioning solution. Both versions of the system have been evaluated using fault injection simulation experiments. Results show that the hardware solution can provide better isolation with respect to the software solutions when soft errors affecting processor memory elements are considered. Conversely, when software defects are of concerns, the two solutions provide the same level of robustness.**

Keywords—- avionic; multi-core; mixed-criticality; consolidation; fault-tolerance; system-on-programmable-chip (SoPC); hypervisor (HV); intellectual property (IP); critical partition (CP); critical application (CA); non-critical partition (NCP); non-critical application (NCA); system watchdog timer (SWDT); watchdog (WD); single event upset (SEU); software bug (SwB)

I. INTRODUCTION

Multicore architectures have been on the market for several years now, and have acquired a clear dominant position. Chip manufacturers are already focused on development and production of multicore chips and are advancing towards ever more complex and performant many-core architectures. Whereas many industries can only benefit from more performant architecture, e.g. consumer electronics, several other industries are struggling with the adoption of such architecture, mainly due to safety regulations and certifications concerns. Such concerns are mainly tied to the challenging problem of certifying a safety-critical system implemented on a multicore architecture according to international regulation prescriptions. In the avionics industry, one of the goals of multicore introduction is the reduction of Space, Weight and Power (SWaP) consumed by avionic equipment, through consolidation of several applications on a single multicore chip. This field of research is very active, and the main avionics players are highly interested in solving the problem since, to the best of our knowledge, no solution has been widely accepted yet.

One of the many challenges in the path towards the definition of a reference architecture for consolidating mixed criticality applications on multicore architectures is resource partitioning. Since two or more applications use a single processing unit (PU), they can virtually access everything is connected to that PU. However, in many applications, this is neither necessary nor desirable, since it may affect the system safety: the misuse of some resources by one particular application may led to starvation on all other applications in the system, which in turn means a potentially catastrophic misbehavior (e.g., an application locking forever a shared resource may lead other applications to miss hard real-time deadline). To solve the partitioning problem, two main solutions have been proposed. A first solution consists in software partitioning, which is implemented by means of a special software layer called type-1 hypervisor: in this solution applications are separated by means of MMU configuration and processor scheduling; the second solution consists in hardware partitioning, which leverages special-purpose hardware units available in multicore processors that allow enforcing resource access policies for each core.

In this paper the application described in [3] is implemented in two versions: one based on a type-1 hypervisor and one based on a hardware partitioning. The reliability of the two implementations is then evaluated by means of fault-injection simulations, considering two fault models: single-event-upsets in the processor registers and in the configuration registers of the target hardware were used to model transient hardware faults, while random bit-flips in the instruction memory were used to model software bugs, as proposed in [4]. Results gathered on a Xilinx Zynq system-on-programmable-chip (SoPC) show that the considered hardware solution is more robust with respect to hardware faults, guaranteeing a better partition than the software solution. Conversely, the two techniques provide the same degree of robustness when software bugs are considered.

This paper is organized as follows. Section II reports a brief overview of the state-of-the-art; section III presents the architecture used in the experiments; section IV discusses the results we gathered on both implementations, while section V draws some conclusions and outlines future steps.

II. Previous Works

Consolidation of several mixed criticality applications has been a goal for avionics for several years. In the past this goal brought to the development of architectural solutions like the Integrated Modular Avionics (IMA). IMA is a design concept by which several hardware resources are tightly controlled by software and can be shared among several applications, achieving a high level of integration [3]. In line with the IMA design concept, the Airlines Electronic Engineers Committee (AEEC) developed and adopted the ARINC653 standard[4][5][6][7][8]. ARINC653 is a standard for services to be provided by a Real Time Operating System (RTOS) to an avionic application. ARINC653 standardization helped reuse of avionic software and simplified its development process. ARINC653 is defined in several documents, however an outline is presented in [9]. ARINC653 always refers to systems implemented on single core architectures.

In [1] a novel architecture was proposed to perform preliminary studies on the mixed-criticality applications consolidation on a multicore system in an avionic scenario. The architecture was implemented using a type-1 hypervisor and results show it to be a promising starting point for building a robust multi-core based avionic architecture. In this paper the same architecture is implemented resorting to a dedicated hardware partitioning solution available in the multi-core processor used for our experiments.

III. Architecture Description

The architecture proposed in [1] implements a mixed criticality application on a multi-core chip with a companion Field Programmable Gate Array (FPGA).

A. *Conceptual architecture*

From a hardware point of view, the architecture relies on a dual-core processor with a watchdog timer, called System Watchdog Timer (SWDT) in the following, and two instances of a Watchdog processor (WDP), called WDP0 and WDP1. Each WDP is associated to a core of the processor, and it implements a Control Flow Check (CFC) strategy. Receiving a sequence of signatures from the software running on the associated core, WDP checks that the time interval between two subsequent signatures is within a predefined max time interval and that the received signatures are in a predefined expected sequence. This architecture includes two applications with different criticality levels, each deployed in its own partition bounded to one processor core and a subset of the available hardware resources (i.e., memory and I/O devices). In this paper we refer to a high criticality application (HCA), that is an application that is safety critical, and a low criticality application (LCA), that is not safety critical. The system includes a Critical Partition (CP) hosting the high criticality application, a Non-Critical Partition (NCP) hosting the low criticality application, and a System Partition (SP), which is in charge of managing error detection and recovery. In the scenario analyzed in this paper, there is no information exchange between the two applications.

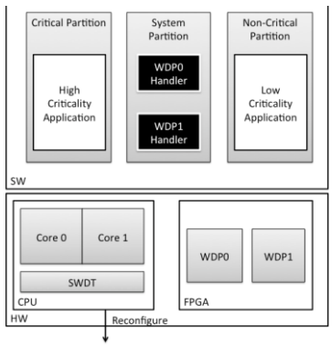

Fig. 1. Conceptual scheme of the architecture, showing hardware and software components.

The watchdogs in the system are allocated to each application as follows: WDP0 is allocated to the CP, WDP1 is allocated to the NCP, SWDT is allocated to the CP. The SP handles both WDPs' Interrupt Requests (IRQs). In response to the WDP1's IRQ the NCP is rebooted, whereas in response to a WDP0's IRQ, the whole system is rebooted and a reconfigure signal is activated on the interface of the architecture. This signal can be used in a hot-standby sparing configuration so that the spare computer can take over functionalities of the malfunctioning computer. The SWDT triggers a system reset entirely managed in hardware, and it also sends the reconfigure signal to the interface. Fig. 1 shows the conceptual architecture described so far.

B. *Architecture implementation using a type-1 hypervisor*

The architecture we developed was implemented using a type-1 hypervisor. When using a type-1 hypervisor as partitioning solution, the mapping of the conceptual architecture is rather straightforward, with the addition of a significant software layer running in Symmetric Multi-Processing (SMP) on the underlying multi-core processor.

The type-1 hypervisor implements the partitioning through memory segmentation enforced by means of Memory Management Unit (MMU). The mapping of the conceptual architecture to the type-1 hypervisor's partitioning scheme is showed in Fig. 2. In this scheme, each of the applications partitions is mapped to one of the two cores, while the SP can run on any core when an interrupt is received, in order to reduce latency.

C. *Architecture implementation using an hardware patitioning solution*

For the sake of this paper, we adopted ARM TrustZone as hardware partitioning mechanism, which allows defining two partitions. The TrustZone partitions are called Secure World and Non-Secure World respectively. The Secure World has all the privileges and is able to access all hardware resources, whereas the Non-Secure World can only access resources that have been explicitly allocated to it. The Secure World is in charge of allocating some of the hardware resources to the Non-Secure World. Any access from the Non-Secure World to a forbidden hardware resource results in a hardware exception to be managed in the Secure World context.

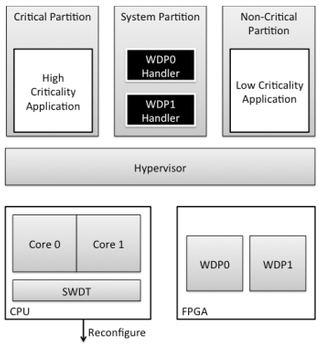

Fig. 2. Scheme of the architecture implemented using a type-1 hypervisor.

In the following the two partitions are called Secure World and Non-Secure World, to adhere to the TrustZone terminology. Core 0, SWDT, and WDP0 were allocated to the secure world, whereas Core 1, and WDP1 were allocated to the non-secure world. Fig. 3 presents the resulting architecture scheme.

IV. Experiments and Results

A. Target Hardware

The architecture was deployed on the same hardware in both implementations. The selected hardware was a Zynq SoPC (System-on-Programmable-Chip), featuring a dual core ARM Cortex-A9 processor and an FPGA fabric on the same chip where the SWDT was implemented through a hard IP core already available in the SoPC, while WDPs were implemented as soft IP cores in the FPGA, as described in [1].

B. Experimental setup

We used fault injection simulations to evaluate the robustness of the two implementations, as described in the following.

1) The benchmark applications

Both implementations were running similar benchmark software composed of two avionic applications. The benchmark applications are part of the Flight Management System (FMS) of a prototype of an Unmanned Aerial Vehicle (UAV) implemented by Alenia Aermacchi and shared in the scope of the EMC^2 project. One of the two applications is a high criticality application and was mapped on the critical partition of the conceptual architecture described in section III, while the other (implementing a house-keeping task) was mapped on the non-critical partition.

C. Fault Injection Simulations

Both system implementations' reliability was evaluated through fault injection simulations. The following terminology is used:

- *Fault Injection Simulation* or *Simulation*: a single fault is injected in the system in a single execution of the benchmark starting from reset state.
- *Fault Injection Simulation Campaign* or *Campaign*: a collection of fault injection simulations.

Each fault injection simulations campaign is characterized by a list of faults to be injected, referred to as the fault list.

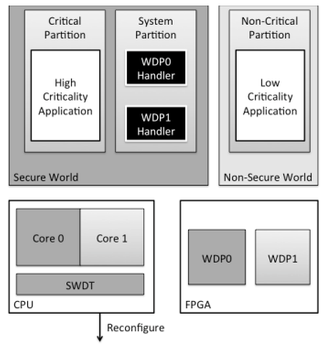

Fig. 3. Scheme of the architecture implemented using the TrustZone. Different tonalities in the hardware components are used to show the assignment of the resource to secure world (dark) or non-secure world (light).

1) Fault List Generation

The fault list is generated by random sampling of a fault space containing all the faults that can occur during a single run of the benchmark. The complete space is too big to allow an exhaustive testing of the system, thus the necessity to operate a random sampling. To obtain a meaningful fault list, faults that could not have an effect were pruned from the fault space. Moreover, only faults affecting the core running the LCA were considered, as the goal of this evaluation was to prove that a misbehaving LCA would not affect the HCA. Cardinality of the final sampling from the pruned fault list has been determined through a Montecarlo approach, observing that beyond the selected cardinality the measured average did not change significantly.

2) Fault classification

The faults were classified comparing the outputs of faulty executions with the expected outputs, gathered at the end of a fault-free execution. Faults were classified as follows:

- *Silent (S):* a fault that did not have any effect on the outputs of the HCA, nor caused a deadline miss.
- *Timeout (TO):* a fault that caused a timeout in the SWDT, triggering a system reset.
- *Failure (F):* a fault that caused a misbehavior in the HCA, i.e. wrong outputs or a missed deadline.

D. Fault Injection Simulations

Hardware faults were simulated, by injecting single bit flips in memory elements within the system. Target memory elements were CPU registers and configuration registers. Software bugs were simulated through random bit flips in the code memory area before execution started. Finally, a bug was introduced in the LCA that simulated a case of resource misuse, forcing LCA to saturate the shared memory bus, to evaluate HCA performance penalty.

1) Hardware Fault Injection Simulation and results

Hardware fault injection simulations were performed as follows. First, the system was booted and

an external debugger stopped execution at the beginning of the main loop. Then execution was continued for a given amount of time, until the injection time was reached, then the execution was stopped and the fault injected through the debugger. Finally, execution was completed and results downloaded to a host workstation, in charge of fault classification. Results are reported in Table 1.

TABLE I. HARDWARE FAULT INJECTION SIMULATIONS RESULTS

	SILENT	TIMEOUT	FAILURE	INJECTED
TYPE-1 HYPERVISOR	5.726	274	0	6.000
HARDWARE PARTITIONING	5.959	41	0	6.000

Results show that the hardware solution achieves better results, due to the absence of the hypervisor software layer, which is a single point of failure.

2) Software Fault Injection Simulation and results

The software fault injection was used to simulate effects of a software bug as proposed in [2]. Simulations were performed in the same way as described above, with the exception that the fault was injected in the code memory are before execution started. Results are reported in Table 2. Results show that HCA is completely immune to software bugs affecting LCA.

TABLE II. SOFTWARE FAULT INJECTION SIMULATIONS RESULTS

	SILENT	TIMEOUT	FAILURE	INJECTED
TYPE-1 HYPERVISOR	10000	0	0	10000
HARDWARE PARTITIONING	10000	0	0	10000

TABLE III. PERFORMANCE PENALTY FOR THE HCA WHEN THE LCA SATURATED THE MEMORY BUS.

	L1 CACHE IN NCP	HCA PERF. PENALTY
TYPE-1 HYPERVISOR	ON	**4.03 %**
	OFF	**4.08 %**
HARDWARE PARTITIONING	ON	**4.12 %**
	OFF	**4.52 %**

3) Resource misuse by LCA

In this simulation, a fault was injected that caused the LCA to continuously access random memory locations in both reading and writing operations, thus saturating the shared memory bus. This simulation targets only effects of the shared memory bus congestion. Memory content corruption side effect has been avoided by simulation design.

Table 3 shows HCA performance penalty in terms of execution time, both when the L1 cache in the NCP was turned on and off. The software benchmark configuration in this simulation was modified in order to avoid that L1 cache in the CP could mask the faulty behavior of the LCA. Results show a similar performance penalty in both implementations.

V. CONCLUSIONS

This paper implemented the architecture proposed in [1] with two partitioning solutions, one implemented through software, the other implemented through hardware. Both implementations' reliability was evaluated using fault injection simulations. Results suggest that the hardware-implemented resource partitioning provides better protection of the critical application against soft errors that may affect the multi-core hardware. Conversely, when software defects are considered (such as misbehavior in the not-critical application that caused it to saturate the memory bus) results show that the two implementation provide comparable results.

ACKNOWLEDGMENTS

The research was partially supported by the ARTEMIS Joint Undertaking project in the Innovation Pilot Programme "Computing platforms for embedded systems" (AIPP5) under grant agreement n. 621429 (project EMC²).

REFERENCES

[1] S. Avramenko, S. Esposito, M. Violante, M. Sozzi, M. Traversone, M. Binello, and M. Terrone, "An Hybrid Architecture for Consolidating Mixed Criticality Applications on Multicore Systems," in *2015 IEEE 21st International On-Line Testing Symposium*, 2015, pp. 26–29.

[2] H. Madeira, D. Costa, M. Vieira, "On the emulation of software faults by software fault injection", *Dependable systems and networks 2000, DSN 2000, Proceedings Internetional Conference on*. IEEE, 2000.

[3] Prisaznuk, Paul J. "Integrated modular avionics." *Aerospace and Electronics Conference, 1992. NAECON 1992., Proceedings of the IEEE 1992 National*. IEEE, 1992.

[4] *ARINC Specification 653: Part 1, Avionics Application Software Standard Interface, Required Services*, ARINC653P1, 2010

[5] *ARINC Specification 653: Part 2, Avionics Application Software Standard Interface, Extended Services*, ARINC653P2, 2012

[6] *ARINC Specification 653: Part 3A, Avionics Application Software Standard Interface, Conformity Test Specification*, ARINC653P3A, 2014

[7] *ARINC Specification 653: Part 4, Avionics Application Software Standard Interface, Subset Services*, ARINC653P4, 2012

[8] *ARINC Specification 653: Part 5, Avionics Application Software Standard Interface, Core Software Recommended Capabilities* , ARINC653P5, 2014

[9] P.J. Prisaznuk, "ARINC 653 role in integrated modular avionics (IMA)." *Digital Avionics Systems Conference, 2008. DASC 2008*. IEEE/AIAA 27th. IEEE, 2008.

[10] O. Goloubeva, M. Rebaudengo, M. Sonza Reorda, M. Violante, "Software-implemented Hardware Fault Tolerance", *Springer Science & Business Media*, 2006.