On the consolidation of mixed criticalities applications on multicore architectures

On the consolidation of mixed criticalities applications on multicore architectures

Stefano Esposito, Massimo Violante

Politecnico di Torino, Torino, Italy
stefano.esposito@polito.it
massimo.violante@polito.it

Abstract In this paper we propose a hybrid solution to ensure results correctness when deploying several applications with different safety requirements on a single multi-core-based system. The proposed solution is based on lightweight hardware redundancy, implemented using smart watchdogs and voter logic, combined with software redundancy. Two techniques of software redundancy are used: the first one is software temporal triple modular redundancy, used for those tasks with low criticality and no real-time requirement. The second software redundancy technique is triple module redundancy for tasks with high criticality and real-time requirements, assisted by a hardware voter. A hypervisor is used to separate each task in the system in an independent resource partition, thus ensuring that no functional interference is occurring. The proposed solution has been evaluated through hardware and software fault injection on two hardware platforms, featuring a dual-core processor and a quad-core processor respectively. Results show a high fault tolerance achieved using the proposed architecture.

Keywords Mixed Criticality; Software Implemented Fault Tolerance; Hybrid Architecture; Multicore Systems; Real Time Applications

1. Introduction

Today's processors market is dominated by multicore architectures. Even though many systems, especially safety-critical ones, are still based on single-core chips, manufacturers are actively phasing out the production of such chips in favor of multi-core architectures, which are proved to be overall more performant than single core architectures. This trend is driven mainly by the need of reducing power consumption while increasing computational capabilities. Markets such as the consumer electronics embraced rapidly this trend, and lately the Internet-of-Things emerged, which is further pushing the evolution of more advanced highly integrated

System-on-Chip (SoCs), featuring several hardware accelerators and peripheral in order to speed-up computations.

However, the industries that have quickly adopted multicore architectures are those that are not concerned with safety standards and regulations. As a consequence, the majority of multicore architectures evolved to optimize an average use-case which is not concerned with deterministic execution and which is facing at best soft real-time requirements, such as the case of video streaming or Voice-over-IP (VoIP).

Nevertheless, there are industries driven by additional concerns other than performance, such as those dealing with safety-critical or mission-critical applications. Industries such as automotive, aerospace, defense, and railways signaling haven't yet started using multicore architectures in all their systems due to safety and regulations concerns. There are mainly two challenges to the use of multicore architecture in such safety-critical use-cases:

1. **Functional isolation:** when several applications are running on the same multicore processor, they should not interfere with one another from a functional point-of-view, i.e. one application should never be able to compromise the correctness of the results of another application, unless there is an explicit dependency.
2. **Temporal isolation:** when several applications are running on the same multicore processor, they should not interfere with one another from a temporal point-of-view, i.e. the worst case execution time (WCET) of one application should not be changed from the presence on the same platform of another application, unless there is an explicit dependency.

Several system architectures based on multicore processors have been proposed, though at the best of our knowledge, none has been widely accepted nor has solved all the challenges faced by such an architecture.

Besides the challenges described above, industries such as aerospace and defense must face strict safety regulations and a long and complex certification process, with the objective of checking that all safety requirements are met. The certification process is based on safety standards adopted by interested industries, like the DO-178C standard, and is carried on by a certification agency. For instance, avionics in the European Union are certified by the European Aviation Safety Agency (EASA).

In this paper, we propose a system architecture designed to allow deployment of mixed criticality applications on multi-processor systems-on-chip (MPSoC). The system architecture is designed to allow consolidation of different applications with different requirements in safety and performance on the same chip. The proposed architecture has been implemented with a prototypical vision-based navigation system as might be found in systems like autonomous rovers for extraterrestrial exploration. In such a system, besides the concerns related to mission critical applications, there is the added problem of the significant exposure of semiconductors to radiations, both during travel to the destination and during the operation phase of the mission. While the effects of the Total Ionizing Dose (TID) can be significant, they

are out of the scope of this paper, as we decided to focus our attention on the operation phase and the soft-errors that might affect it. We modeled effects of the radiations as Single Event Upsets (SEUs), Single Event Latch-ups (SELs), and Single Event Functional Interruptions (SEFIs), using fault injection experiments to evaluate effects of such faults on a prototypical vision-based navigation system. The system is composed of two applications: one is mission-critical, while the other has a lower criticality. The architecture has been implemented on two different Commercial Off-The-Shelf (COTS) multicore chips with a companion COTS FPGA. The FPGA was used to implement some special-purpose hardware to detect errors and allow a recovery action.

This paper's main contribution is the development of a method to consolidate safety- or mission-critical applications on a multicore chip starting from the partitioning method down to special purpose hardware supporting the fault-tolerance. The main focus is on fault-tolerance, which is one of the main issues for some applications domains, whereas the resource sharing issue is not directly tackled in this paper.

The paper is organized as follows: section II contains an overview of previous works on the subject; section III describes in detail the proposed architecture; section IV describes the performed fault injection experiments and gathered results; finally, section V proposes some conclusions.

2. Previous works

Semiconductors behavior in radioactive environment have been widely studied in the past years. The main field of interest for this researches has always been space and several solutions were proposed, starting from the adoption of specifically designed hardware capable of correct operations under the effect of radiation. In this paper we use the same terminology as introduced in Goloubeva et al. [1].

Several well-known techniques exist and are currently used by industry. One of these is the Triple Modular Redundancy (TMR). In TMR a single module, usually hardware, is implemented three times. All the replicas receive the same input at the same time and are expected to produce the same output. A voter receives the outputs of all the replicas and compares them to find a majority agreement, i.e. the voter selects the correct output based on the fact that at least two of the three replicas provided the same output. This technique has been used to implement a space computer based on COTS components [2]. The implementation of this computer shows how TMR can be implemented at different granularities: although it has been used to design radiation-hardened hardware by replicating simple design units, sometimes down to the single device, authors of [2] have triplicated an entire processor module to check correctness of the outputs. TMR is guaranteed to transparently tolerate a single error, as may be caused by a Single Event Effect (SEE).

Goloubeva et al. [1] also present an extensive overview of many software implemented hardware fault tolerance (SIHFT) techniques. Such techniques are based on

the idea of software redundancy, i.e. of additions and modifications to the software with the purpose of detecting and tolerate the effects of hardware faults. The techniques that are most relevant to this paper are those used to detect and tolerate transient faults, as those modeled by Single Event Upsets (SEUs). Among these, the Virtual Duplex System (VDS) is of particular interest for this paper. VDS is based on software duplication at program level. An entire program is executed twice, and the outputs of the replicas are compared for agreement. The original proposal was based on temporal redundancy, meaning that the two executions were performed on the same single processor in subsequent times, but the solution can also be implemented exploiting spatial redundancy, i.e. the replicas can be executed in parallel on different processing cores, reducing time overhead [3]. Moreover, VDS can benefit from an N-versioning approach [4][5]. N-versioning prescribes that the replicas are created separately, possibly by different people, using different toolchains and implementing different algorithms to satisfy the same requirements. This can significantly reduce the probability of common-mode faults leading to a silent data corruption, due to the fact that when both replicas are affected in the exact same way by the same fault, they also produce the same output, although wrong, and VDS would not be able to detect this fault.

VDS and TMR can be combined in Temporal Triple Module Redundancy (TTMR) [6]. In TTMR, two executions are performed and the outputs are compared for agreement, as described for VDS. When no agreement is found, a third execution is performed, and the correct output is selected by a majority vote, as in TMR.

To reduce computational overhead of purely software solutions, many hybrid solutions have been proposed, in which special hardware is used to offload some detection from the software, thus reducing overhead. Two hybrid solutions are proposed by Pignol [7], where a software redundancy scheme is assisted by a special hardware IP-core to implement fault tolerance. The architecture was designed to allow use of COTS in space applications, and needs the use of radiation hardened companion chips implementing some key functionality for detection and tolerance (e.g. safe context storage). Special purpose hardware cores can be used to assist fault detection as in the case of Watchdog Timers (WDTs) and Watchdog Processors (WDPs) [8][9].

Esposito and Violante [10], proposed an architecture based on a combination of WDPs and WDTs focused on space applications. The proposed architecture used software and hardware means for detecting errors and implemented tolerance to transient faults by redundancy of execution. Permanent faults were addressed by use of a standby spare which would be activated after a configurable number of fault detections in the same system.

That architecture was adapted to the mixed criticality avionic use case and tested by Avramenko et al. [11]. The implementation was based on a type-1 hypervisor granting isolation of applications in resource partitions. A specifically designed WDP core was used to implement a control flow check (CFC) technique, used to ensure correct execution flow of all tasks in the system. The WDP implemented a signature-based CFC technique, in which the software has to send a predefined se-

quence of signatures to the WDPs within a given timeout. Each signature is associated to a block in the program, defined as a portion of software with a singled entry and a single exit. A WDT is also used in the architecture described in Avramenko et al. [10] to recover from Single Event Functional Interruptions (SEFIs) and from permanent errors. The software is in charge of resetting the WDT within a given timeout, otherwise the system is reset and a signal is sent to the external interface in order to allow a redundant computer to provide correct outputs to the user. Avramenko et al. [10] evaluated the architecture through fault injection simulation experiments.

The architecture proposed in this paper, is built on top of the one proposed by Esposito and Violante [10] and uses results of Avramenko et al. [11] with some additions and modifications. This paper is an extension of Esposito et al. [12]. The main additions are a more extensive description of the architecture and an evaluation on quad-core architecture, which allowed to experiment with the scalability of the proposed solution.

3. Proposed Architecture

The proposed solution is a multi-layered fault-tolerant architecture. The fault models considered are transient faults affecting at most one memory element in the architecture, i.e. Single Event Upset (SEU) faults. The architecture also targets Single Event Functional Interruptions (SEFIs) and functional interruptions caused by permanent faults, including Single Event Latch-ups (SELs). It uses a hybrid approach for fault detection and tolerance, implementing special purpose hardware cores to support fault detection. The architecture is composed of three layers, which are, from bottom to top:

1. **Hardware:** it is composed of a Multi-Processor System-on-Chip (MPSoC) and of a companion chip. A set of requirements is specified for the MPSoC, as specified below. The companion chip implements three specific Intellectual Property (IP) core, as described below.
2. **Middleware:** it is implemented through a type-1 hypervisor which uses services provided by the hardware layer in order to implement functional separation of different partitions of the application software.
3. **Application Software:** it implements the functionality required of the system, and uses services provided by the two layers below it to implement fault detection and tolerance.

3.1. Hardware Layer

The hardware represents the first layer of the architecture. Its architecture is depicted in Fig. 1

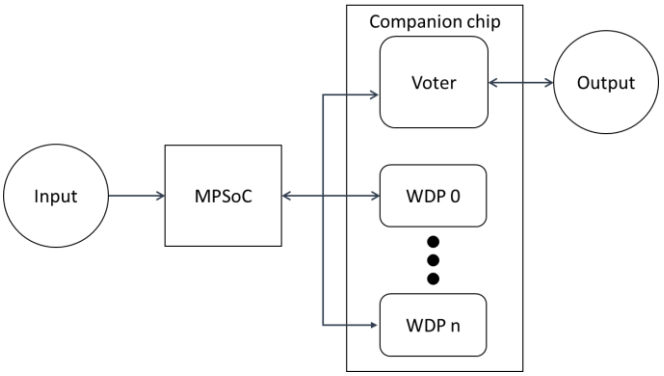

Fig. 1. Hardware layer architecture

The hardware layer is composed of an MPSoC and of a companion chip. The MPSoC includes the main processing units together with some peripherals required for correct functioning of the system. In order for the middleware layer to work properly, the MPSoC should implement cores with a Memory Management Unit (MMU).

The companion chip implements the hardware peripherals required by the system that are not found in the MPSoC. The system architecture requires the availability of three intellectual property (IP) cores:

- **Watchdog Processor (WDP):** used to control that the execution flow of the monitored task is the expected one.
- **Majority Voter (Voter):** used to control that the outputs of a task in TMR configuration are correct.
- **System Watchdog Timer (SWDT):** a WDT able to reset the system when not rearmed within the configured timeout.

The *Watchdog Processor* implements a Control-Flow-Check (CFC) technique based on predefined signatures. To implement this technique, the monitored task has to be partitioned into blocks. A block is defined here as a portion of software with one entry point and one exit point. It is not to be confused with the similar but stricter notion of Basic Block (BB). A BB is defined as a portion of software without jump instructions, except the last one, and in which no instruction is target of a jump instruction, except the first one [1], whereas the block definition used in this paper allows for internal jumps and even function calls to be included, as long as the block has one entry point and one exit point. This definition allows to select he preferred granularity for software partitioning. The finest granularity allowed by the definition is considering a BB as a block; on the other end of the spectrum, the coarsest granularity would be to consider the whole program as a block. In between these two ends, any intermediate granularity can be used. When selecting the desired granularity, a trade-off between performance overhead and error-latency should be

considered. The finer the granularity, the higher the performance overhead, due to the interaction between the software and the WDP. On the other hand, the coarsest the granularity, the longer is the error latency. Once the partitioning has been completed, a signature is associated to each block. The signature is a unique identifier hand-picked by the designer. The signature should be constant at compile time, although the WDPs are not hardcoded with the signatures, meaning that the same WDP implementation can be reused for different software and different partitioning selections. During bootstrap the software is in charge of sending to the WDP the expected signature and timeout for each block, in the expected order. Once the configuration phase has been completed the WDP can be enabled by the software. Starting from the enable signal, the WDP expects to receive the predefined signatures within the configured timeout. The WDP is able to detect the following error scenarios:

- **Unexpected signature:** when the WDP receives a signature from the software it checks if it is the expected one; an error is detected if the received signature is not the expected one. This can be due to a CFE causing the software to send the signature for a different block, or a fault causing the software to send a signature not included in the predefined set.
- **Timeout:** the configured timeout expired before a signature was received;
- **Illegal interaction:** the only legal interactions with the WDP are configuration and sending a signature. Any other way of interaction, including access to configuration registers while the WDP is enabled, is treated as an error condition.

Whenever an error condition is detected, the WDP sends an Interrupt Request (IRQ) to the MPSoC. The system reacts to the IRQ by implementing a proper recovery action, as described below. WDP is able to detect transient faults and permanent faults that do not hinder the ability of the processor to react to IRQs.

The *Hardware Majority Voter* features three registers in which the software writes its results or a signature of its results. Each register is statically assigned to a replica of the software. Once all replicas write their output to the dedicated registers, the voter performs a majority vote to select the correct output. The correct output is presented to the user directly by the voter.

Using WDP and Voter, the architecture is able to manage control flow errors and silent data corruptions. Still, there are some conditions in which the system is not able to perform the expected workload nor to react to IRQs. To detect this condition a third IP is used in the proposed architecture: System Watchdog Timer (SWDT). SWDT is a WDT configured at boot. When it is not rearmed within the configured timeout, SWDT is capable of triggering a system reset. A hardware able to perform the operations required to the SWDT is usually available in the Multi-Processor System-on-Chip (MPSoC) in use, and as such is not further specified in this paper, nor it is considered as included in the companion chip.

The SWDT is able to detect and react to SEFIs and to permanent faults compromising the operations of the MPSoC.

3.2. Middleware

The proposed architecture includes a middleware layer implemented through a commercial type-1 hypervisor. The selected type-1 hypervisor implements separation between the tasks composing the application software. In this architecture the task is the basic software entity. A task implements a functionality that satisfies one or more system requirements. Each task has an associated assurance level. To grant functional isolation between tasks, each task has its own resource partition, which includes a memory area private to the task and hardware peripherals. The type-1 hypervisor implements resource partitioning by using services provided by the hardware layer. The type-1 hypervisor is in charge of granting that tasks assigned to different partitions cannot interfere with one another in case of misbehavior. This includes the use of separated physical memory segments, granting that no task is able to corrupt data used by another task in a different partition. The type-1 hypervisor is also responsible for providing inter-partition communication methods, so that tasks can exchange data as needed. A failure in the type-1 hypervisor is detected by the SWDT.

3.3. Application Software

The top layer of the proposed architecture is the application software layer, which implements fault detection and tolerance using the services provided by the two bottom layers. The focus of this work is on mixed-criticality applications, thus we refer to two levels of criticality, which can be mapped to any two assurance levels of the industry standard of interest. In the proposed architecture, we abstract from the details of the safety standard to focus on the real-time requirements of an application and on the impact of its misbehavior.

Real-time tasks can be classified based on the consequences of a deadline miss. In this paper we use the classification from Shin and Ramanathan [13]:

- **Soft deadline/Soft real-time task:** violation of the deadline causes a degradation of service without catastrophic consequences. Value of results obtained after the deadline is not zero and degrades with time.
- **Firm deadline/Firm real-time task:** violation of the deadline causes a degradation of service without any catastrophic consequences, granted that the number of violations is not too high. Value of results obtained after the deadline is zero.
- **Hard deadline/Hard real-time task:** violation of the deadline causes catastrophic consequences.

Based on the classification of the task, one of the two proposed fault detection and tolerance technique can be used.

3.4. Fault Detection and Fault Tolerance techniques

The fault detection and tolerance techniques implemented in the proposed architectures are TMR and TTMR. Depending on the classification described in the previous subsection, one of the two techniques should be applied to each task as described in this subsection.

3.4.1. Triple Modular Redundancy

Hard Real-Time (HRT) tasks should be implemented using a TMR approach, as described in Fig. 2. This approach is to be used for HRT because tasks of this kind cannot tolerate the delay introduced by the TTMR approach described in Section 3.4.2 when recovery is needed. The TMR approach does not introduce any delay, since any error is detected and recovered in the hardware voter.

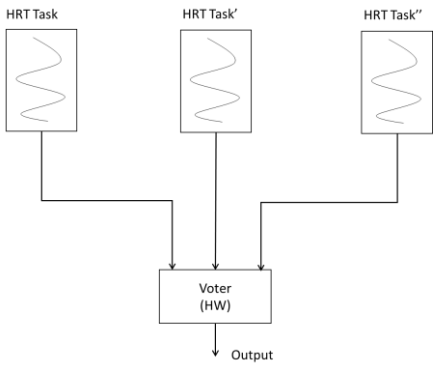

Fig. 2: Triple Modular Redundancy scheme. Each HRT task is triplicated. A voter receives outputs of all replicas and selects the correct one by majority vote.

In the proposed approach, each HRT task that has to be implemented is triplicated. Each replica is inserted in its own hypervisor partition and executed on a dedicated core. The output of each replica is fed to the hardware voter implemented in the hardware layer, which selects the correct output as previously described. This solution is suited to HRT tasks because it guarantees that the correct output is available after the completion of all three replicas, i.e. in a deterministic time. It is to be noted that in this configuration, the type-1 hypervisor is responsible to ensure that each replica can only write on the dedicated voter register, in order to avoid that a misbehavior in one replica could mask the error by writing more than once its result, thus compromising the correct functionality of the voter.

The TMR architecture is able to mask transient and permanent faults which only affects one of the replicas, without adding any delay, which is the main reason it

should be used for HRT tasks. It is able to detect faults affecting two of the replicas, in which case a reset of the board should be performed. In case the same condition is detected several times within a given time or the reset is not acceptable for the application, the system should react to this eventuality by switching to a stand-by spare. Since a stand-by spare is usually mandatory in safety- or mission-critical applications, this does not introduce any additional cost.

3.4.2. Time Triple Modular Redundancy

Firm Real-Time (FRT) and Soft Real-Time (SRT) tasks should be implemented using a TTMR approach as described in Fig. 3. This approach is to be preferred for these kinds of tasks because it does not introduce a hardware cost, since the voter is software-implemented, and the delay introduced by the additional execution can be tolerated by FRT and SRT tasks.

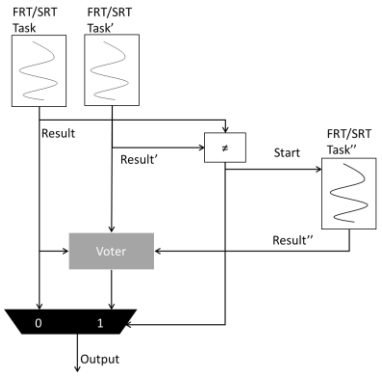

Fig. 3. Temporal Triple Modular Redundancy. White modules are in software, the voter can be either in software or in hardware, the output selection is a logical block.

In TTMR each task is triplicated, but instead of always running the three replicas in parallel, just two instances are executed. Outputs of the two replicas running in parallel are compared by a software module. In case of disagreement, the third replica is executed and the correct output is selected through majority voting. In this implementation of TTMR, no checkpoint storage is needed, because the third execution is performed from scratch. In this configuration, the voter can be implemented either in hardware or in software. The comparator output is also used to select the final output of the TTMR task, which is either the output of the first replica, if comparison finds agreement, or the output of the voter, if comparison does not find agreement.

TTMR is suitable for applications that do not have hard real-time requirements, because in case a fault the time required to perform the computation can be longer than expected. It can be suitable for FRT and SRT tasks, provided that the probability of a fault is acceptably low. It is particularly suited for best-effort tasks, i.e.

tasks with no real-time requirements, because it guarantees that the correct result is eventually achieved.

TTMR is able to detect and tolerate transient and permanent faults affecting one of the replicas, although permanent faults in the TTMR architecture can cause a performance degradation. Such a performance degradation should be tolerated by the system, given the non-safety- nor mission-critical nature of tasks that can be implemented in a TTMR configuration.

3.5. System scheduling

Once the tasks have been classified as HRT, FRT or SRT tasks and the proper detection and tolerance technique has been selected and implemented at task level, the system should be properly scheduled in order to fit all the tasks within respective deadlines. Any feasible [14] scheduling algorithm can be adopted. In order to properly implement the TTMR technique some modifications should be implemented to the obtained system scheduling. In order to explain this concept, let us consider a system composed of two tasks running on a dual-core MPSoC. The two tasks are

1. **Vision (V):** a soft real-time task whose results are still useful if they arrive within a given latency after the deadline. In the scenario considered in this example, the vision task is periodical, its deadline corresponds to the period and the maximum latency of a result is one period.
2. **Control (C):** a hard real-time task whose results are mission critical.

Given its nature, the Vision task can be implemented in a TTMR configuration, whereas the Control task should be implemented in a TMR configuration. This decision yields a system with the following set of tasks:

1. **V, V', V'':** three replicas of the vision task V.
2. **C, C', C'':** three replicas of the control task C.
3. **V_C (Vision Check):** the task implementing the comparator and (optionally) the software voter for the TTMR configuration. This task is responsible for selecting the correct output of the Vision task, as described in the previous subsection.

The scheduler should be aware that in each major cycle, only two Vision tasks should be scheduled and all three Control tasks should be scheduled, and the Vision check should be scheduled too. Considering the availability of only two cores, one possible scheduling for this system is given in Fig. 4. It is to be noticed that for the method to be applicable, requests to shared resources performed by each task should be contained in the execution slot of each task and that the concurrent execution of two instances of the same task should be possible.

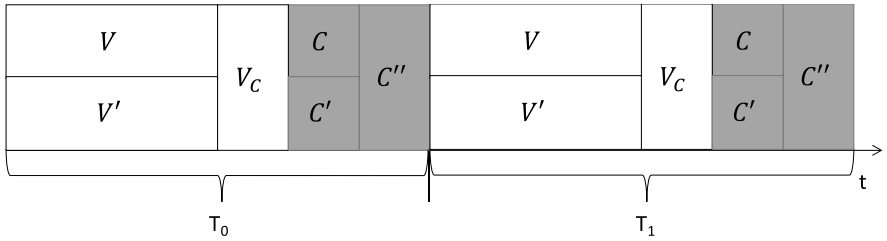

Fig. 4. One possible system scheduling in nominal operation.

The system scheduling showed assumes the system is working as expected, i.e. no faults are active in the system. Fig. 5 shows two consecutive major cycles, which are delimited by the period of the tasks. Both tasks are assumed to have the same period for sake of simplicity, however the proposed solution does not require this condition, as long as a feasible scheduling exists. In this scenario, if a fault affects one of the Control tasks it gets masked by the TMR configuration. A fault affecting one of the two Vision tasks running in parallel is detected by the Vision Check task. The system reacts to such a fault by modifying the scheduling for the next major cycle as showed in Fig. 5.

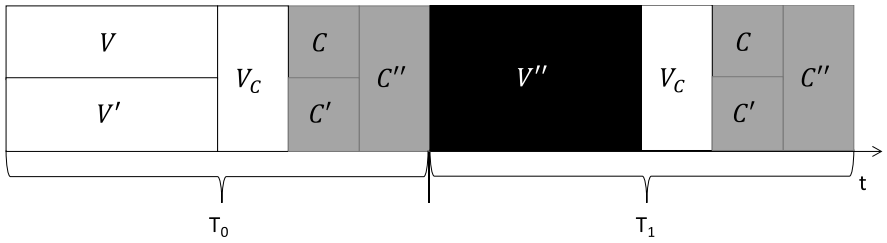

Fig. 5. The scheduling is modified for the next major cycle when an error is detected in a TTRM configuration.

In this scenario, the third replica of the Vision task is executed in the next major cycle, and its results are passed to the Vision Check task. When executed in this context, the Vision Check should implement the software voter and provide the correct output.

A similar example can be proposed for a system based on a quad-core MPSoC, implementing three tasks:

1. **Vision (V):** same as in the previous scenario
2. **Control (C):** same as in the previous scenario
3. **Sensor Logging (S):** a task responsible for collecting and compacting sensors data to be stored or sent to a remote station. This task can be considered neither safety-critical nor hard real-time, however it should not produce wrong results, otherwise the stored sensor data could be useless or misleading.

After applying the proposed tolerance techniques, the system includes the following tasks:

1. **V, V', V'':** three replicas of the vision task V.
2. **C, C', C'':** three replicas of the control task C.
3. **S, S', S'':** three replicas of the sensor logging task S.
4. **V_C (Vision Check):** the task implementing the comparator and (optionally) the software voter for the TTMR configuration of the Vision task. This task is responsible for selecting the correct output of the Vision task, as described in the previous subsection.
5. **S_C (Sensor Logging Check):** the task implementing the comparator and (optionally) the software voter for the TTMR configuration of the Sensor Logging task. This task is responsible for selecting the correct output of the Sensor logging task, as described in the previous subsection.

A possible scheduling for the system is showed in Fig. 6. As for Fig. 4, this is a scheduling assuming that the system works as expected. It is to be noticed that for the method to be applicable, requests to shared resources performed by each task should be contained in the execution slot of each task and that the concurrent execution of two instances of the same SRT task or three instances of the same HRT task should be possible.

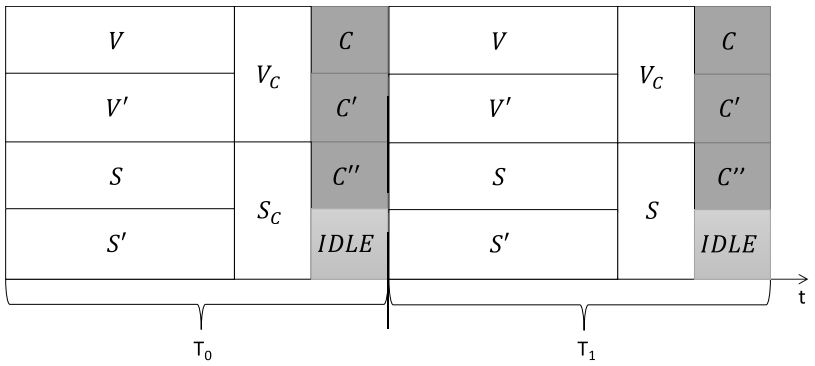

Fig. 6. One possible system scheduling in nominal operations.

Having four cores to use, the three replicas of the Control task can run in parallel, thus reducing the execution time of the task. Moreover, one more task can be consolidated on the same chip, thus scaling up the level of integration. In case a fault is detected in one of the tasks in TTMR configuration, the system reacts in the same way described for the dual-core scenario. In a quad-core scenario, the system can even tolerate two faults in TTMR tasks, as long as the two faults do not affect replicas of the same task, as shown in Fig. 7. In this scenario, the next major cycle is used to run the third replica of both TTMR tasks, while the Control tasks continues unaffected.

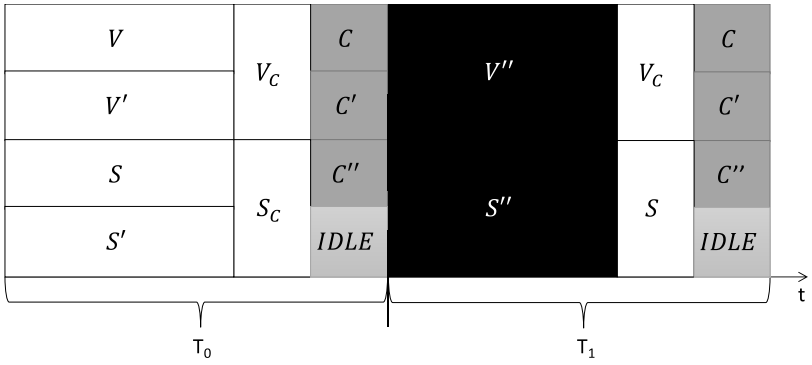

Fig. 7. System reacts to a fault affecting one replica of each TTRM task.

4. Experiments and Results

The proposed architecture was evaluated through fault injection experiments on actual hardware, using an external debugger and targeting architectural registers and MPSoC configuration registers. The workload of the system was selected in order to emulate part of the workload of an autonomous vehicle navigation system. In particular, we selected a processing phase, consisting of one or two tasks, and an actuation or control phase, consisting of one phase. In this section, the program implemented in all the tasks are described, together with the fault injection system used. The experiments have been performed on two hardware platforms.

The first hardware platform is a Xilinx Zynq system, which is a dual-core based MPSoC. The Xilinx Zynq implements a dual-core ARM Cortex-A9 processor as a hard-IP core, together with an FPGA fabric on the same die. In our experiments, the FPGA fabric is used to implement the companion chip.

The second hardware platform is a SanitasEG's Inventami board, featuring an NXP i.MX6Quad MPSoC, based on a quad-core ARM Cortex-A9 processor, together with a Lattice ECP5U FPGA, which has been used to implement the companion chip.

On both hardware platforms, the whole system described in Section 3 has been implemented, using the commercially available type-1 hypervisor we described in Section 3.

4.1. Benchmark application

The benchmark application has been selected to emulate part of the workload of a navigation system of an autonomous vehicle to be used in the context of space ex-

ploration. The target system is an autonomous rover on the surface of an extraterrestrial body. In this context, the electronics systems are subject to a radiation environment which could compromise the correct behavior of the system. Using the proposed architecture, such misbehavior can be detected and tolerated.

In our scenario, the navigation system uses computer vision to decide some actuations. The system is partitioned so that the computational heavy vision algorithms are not considered mission critical, whereas a control task, which is mission critical, is in charge of deciding the actuation based on the results from the vision algorithms. Besides this two tasks there is a third task in charge of collect and compact sensor data to be logged and sent to Earth by a different subsystem.

The vision algorithm used in our scenario is a Sobel-based edge detection algorithm. The result of the edge detection is used by the control task to decide the actuations to be performed. For instance, the edge detection algorithm could identify an obstacle on the path, the control algorithm would then actuate the wheels so that the rover deviates from the original path in order to avoid the obstacle.

In our scenario, a new frame is available every two control task's periods, meaning that the vision task can use two periods to perform its computations. Rather than use all that time in pure computation, the vision task can be implemented in a TTMR configuration, as described in the previous section. On the other hand, the control task is mission critical and has hard real-time requirements, thus it should be implemented in a TMR configuration.

The third task, performing sensor data logging, can be considered a best-effort task, although it is desirable to not have misleading outputs from it. This task can be implemented in a TTMR configuration too.

The resulting system scheduling on the two hardware platforms are as described in the previous section (see Fig. 4, Fig. 5, Fig. 6, and Fig. 7).

The benchmark application code size and WCET are reported in Table 1 and Table 2, for both the vanilla benchmark and the protected version used in our experiments. When the technique is applied, code memory size roughly triplicates, with some overhead due to addition of some operations needed to use WDPs and SWDT. The WCET was estimated by profiling in a multicore system the experimental workload described below with the worst case workload. The worst case workload was crafted based on the knowledge of the implementation.

It is to be noticed, that the actual performance overhead of using WDPs depends on the partitioning granularity. As described above, WDP is designed to monitor that blocks of the software executes in the expected sequence within the expected duration. The block definition used in the WDP specification is generic on purpose, so that the designer can trade-off between error latency and performance overhead. Small blocks yield a lower error latency, but require a higher performance overhead, since the WDP is accessed more often. Larger blocks yield a higher error latency, but require a lower performance overhead. In our experiments, we selected large blocks, thus the performance overhead is low. However, both the Vision and the Sensor Logging tasks have the overhead of the error detection, which amounts to most of the overhead reported in Table 2. In Table 2 the performance overhead is reported in ms, instead of clock-cycles, because we feel that clock-cycles count is

an architecture-dependent measure, whereas the execution time can be more easily compared with results on different architectures.

The Control task does not experience any of this overheads, since its only overhead is the rearming of the SWDT, performed through a single store instruction, and the write of the result in the voter register, which has the same overhead as writing the results on an actuator register. However, in a dual core scenario the control task has a 100% overhead, due to the third execution which has to be executed in sequence after the first two as showed in Fig. 4.

Table 1. Code memory area

	Vanilla	Hardened
Vision	23.4 KiB	93 KiB
Sensor Logging	18 KiB	72 KiB
Control	9.4 KiB	29 KiB

Table 2. WCET

	Vanilla	Hardened
Vision	12 ms	16 ms
Sensor Logging	12 ms	16 ms
Control	4 ms	8 ms (ZYNQ) 4 ms (i.MX6)

4.2. Fault Injection System

The fault injection system has been implemented using an external debugger to access the system through the Test Access Ports (TAPs) available on both hardware platforms. In this subsection, the fault injection system is described. In this section we refer to:

- **Fault injection experiment:** a single execution of the benchmark, in which a fault is injected and results are gathered and compared with a golden result to be classified
- **Fault injection campaign:** a collection of fault injection experiments.

4.2.1. Fault model and fault list generation

The fault model used is that of the bit-flip in a memory cell, which can model most SEEs in a simple and convenient way. The target of the fault injection experiments was composed by CPUs architectural registers and MPSoC configuration registers. Given such target, the possible faults are points of a space identified by the target location and the injection time. The cardinality of such space is easily too big to make an exhaustive fault injection campaign infeasible. In order to reduce the number of faults to be injected, two operations can be performed:

1. **Space cardinality reduction:** the fault space cardinality was reduced by reducing the number of target locations, thus identifying a subspace containing significant faults.
2. **Random sampling of the subspace:** since the cardinality of the identified subspace can still be too large, random sampling can be performed within the subspace.

Space cardinality reduction

In order to reduce the dimension of the faults space, a sampling can be performed on all dimensions of the space.

A first subsampling can be performed on the time dimension, by considering only the core part of the benchmark applications, i.e. discarding the time during which the system is performing bootstrap and configuration. The justification for such subsampling is that the bootstrap and configuration phases are far less frequently executed than the core part, thus the probability of them being affected by a fault is negligible compared to the probability of the core part being affected by a fault.

Moreover, also the location dimension can be subsampled. In order to explain and justify this subsampling, let us explicitly state that each point of the location dimension identifies a bit in the CPU architectural registers and MPSoC configuration registers. As such, not all bits are useful as targets of a fault injection. Indeed, not all architectural registers might be used by the program, for instance, the program might not be using floating point operations, thus the floating point registers can be neglected. In a similar way, not all configuration bits in the MPSoC registers are useful, since many may be used to configured unused subsystems and peripherals, for instance the application may not be using the Controller Area Network (CAN) controller or the integrated High Density Multimedia Interface (HDMI). As a consequence, the cardinality of target locations can be significantly reduced.

Random Sampling

Random sampling consists in randomly selecting a given number of faults from the reduced fault space. The exact number of faults to be sampled is obtained by successive approximation. At the beginning of the process, a sensible number of faults is selected and a fault campaign is executed. Results are collected and significant statistics are extracted (more details on this below). Next a new set of faults of the same cardinality is sampled from the reduced subspace, without repetition, and a new fault campaign is executed. If the statistics extracted from this campaign

are within a confidence interval from the results of the first one, the cardinality is considered sufficient and results are finalized.

4.2.2. Fault injection campaigns and classification

When the fault list is ready, fault injection campaigns are performed. A fault injection campaign consists of a collection of fault injection experiments. Each fault injection experiments injects one fault from the fault list. A fault injection experiments is performed as follows:

1. A fault is extracted from the fault list
2. The system is reset, to avoid fault accumulation
3. As soon as the bootstrap and configuration phase is over a timer is started
4. When the timer expires, the fault injection time has been reached and execution is stopped
5. The target bit is flipped
6. Execution resumes until the end of the actuation cycle
7. Execution is stopped and outputs are downloaded on the workstation
8. Fault classification is performed and results are logged for further processing.

Each fault is classified as:

- **Silent (S):** the fault had no effect on the system. Since faults detected and corrected by the TMR are completely transparent to the user and to the software, those faults are considered Silent in this paper and are not analyzed further.
- **Recovered without reset (w/o R):** the fault was detected and recovered without resetting the system. This means that a third execution of a monitored FRT, SRT or best-effort task was performed to correct the error.
- **Recovered with reset (w/ R):** the fault was detected by the SWDT and caused a system reset.
- **Failure:** the fault led to Silent Data Corruption (SDC) or deadline miss in the Control task.

To classify an output as SDC, a comparison is performed with a golden output. A golden output is obtained by an execution without fault injection, or golden run. Obviously, all fault injection experiments are performed using the same input data as the golden run.

4.3. Results

Two campaigns, each composed of 10,000 faults, were performed on each of the hardware platforms. Of these, one campaign for each hardware platform targeted CPU architectural registers, and one campaign for each hardware platform targeted MPSoC configuration registers. Results of all campaigns are reported in Fig. 8 and

Fig. 9. The number of injected faults has been selected observing that the results did not vary anymore with the execution of more experiments after the first 10,000 were injected, which is indication of statistical significance.

Results show tolerance to injected faults. Most faults are silent or masked by the hardware voter. Of those detected, very few put the system in a state from which it can only be recovered through a reset. No fault at all caused a Failure, i.e. a SDC.

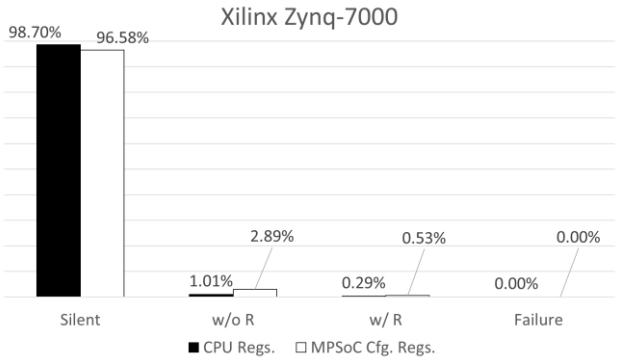

Fig. 8. Results on the Xilinx Zynq-7000 hardware platform.

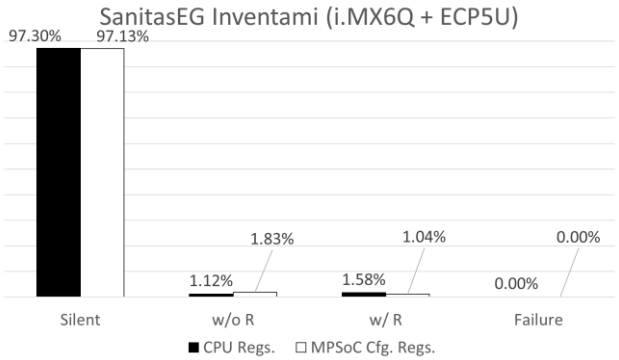

Fig. 9. Results on the SanitasEG Inventami hardware platform

Performance and Code memory size overhead are reported in Table 1 and Table 2 and are discussed in the corresponding subsection.

The tolerance overhead is mainly on the non-time-critical tasks, i.e. Vision and Sensor Logging, whereas the control task is only affected by the tolerance when a reset is required to recover the system execution. This is due to the TMR configuration, in which the hardware voter implements both fault detection and tolerance in a way completely transparent to the software and without overhead.

5. Conclusions

This paper presented a system architecture for mixed criticality applications integrated on a multi-core chip. The proposed architecture combines known techniques in an innovative way to achieve high rates of fault tolerance with a reduced time overhead. The architecture is designed to consolidate high criticality applications with hard real-time requirements with low criticality tasks with softer real-time requirements. The architecture uses both TTMR and TMR to implement fault detection and tolerance, and relies on some special hardware to support software fault tolerance. The TMR is implemented by triple execution of the task and a hardware voter, which drives the actual outputs. The TTMR is implemented in two steps:

1. two tasks are executed in parallel on the same input, and their outputs are compared for an agreement.
2. in case of mismatch, the task is executed a third time and a software implemented voter selects the correct output; in case of agreement, the output is sent to the user.

The architecture was implemented on two hardware platforms, one based on a dual-core processor, the other based on a quad-core processor. The implementation was tested by means of fault injection simulation experiments. Results show high fault tolerance capabilities, with a decisive majority of faults having no effect on the system or being masked by tolerance mechanisms. A few faults affecting non-time-critical tasks were detected and recovered without affecting the time-critical tasks. Fewer faults yet, required a system reset, due to the fault resulting in SEFIs or affecting the execution of the hypervisor kernel causing a kernel panic condition.

Results described in this paper, prove that the solution is a viable way to implement fault-tolerant applications.

6. Acknowledgments

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

7. References

[1] O. Goloubeva, M. Rebaudengo, M. Sonza Reorda, and M. Violante, Software-Implemented Hardware Fault Tolerance. Springer Science and Business Media, 2006.

[2] R. Hillman, G. Swift, P. Layton, M. Conrad, C. Thibodeau, and F. Irom, "Space processor radiation mitigation and validation techniques for an 1, 800 MIPS processor board," Eur. Sp. Agency, (Special Publ. ESA SP, vol. 2003, no. 536, pp. 347–352, 2004.

[3] S.K. Reinhardt and S. S. Mukherjee, "Transient fault detection via simultaneous multithreading," Proceedings of the 27th International Symposium on Computer Architecture, pp. 25-36, 2000.

[4] K. Echtle, B. Hinz, and T. Nikolov, "On hardware fault detection bydivers software," Proceedings of the 13th International Conference on Fault-Tolerant Systems and Diagnostics.

[5] H. Engel, "Data flow transformations to detect results which are corrupted by hardware faults," Proceedings of the High-Assurance Systems Engineering Workshop, 1996, IEEE, pp. 279-285, IEEE, 1996.

[6] D. R. Czajkowski, M. P. Pagey, P. K. Samudrala, M. Goksel, and M. J. Viehman, "Low power, high-speed radiation hardened computer & flight experiment," IEEE Aerosp. Conf. Proc., vol. 2005, no. October 2004, 2005.

[7] M. Pignol, "DMT and DT2: Two Fault-Tolerant Architectures developerd by CNES for COTS-based Spacecraft Supercomputers," Symp. A Q. J. Mod. Foreign Lit., 2006.

[8] A. Mahmood and E. J. McCluskey, "Concurrent error detection using watchdog processors-a survey," IEEE Trans. Comput., vol. 37, no. 2, pp. 160–174, 1988.

[9] M. Namjoo and E. J. McCluskey, "Watchdog Processors and Capability Checking," Twenty-Fifth Int. Symp. Fault-Tolerant Comput. 1995," Highlights from Twenty-Five Years"., vol. III, pp. 245–248, 1995.

[10] S. Esposito and M. Violante, "Mitigating Soft Errors in Processors Cores Embedded in System-on Programmable-Chips," in FPGA and Parallel Architectures for Aerospace Architecture, F. L. Kastensmidt and P. Rech, Eds. Zug, Switzerland: Springer, 2015.

[11] 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.

[12] S. Esposito, S. Avramenko, M. Violante, "On the consolidation of mixed criticalities applications on multicore architectures", in 2016 17th IEEE Latin American Test Symposium, pp. 57-62.

[13] K. G. Shin and P. Ramanathan, "Real-Time Computing: A New Discipline of Computer Science and Engineering," *Proc. IEEE*, vol. 82, no. 1, pp. 6–24, 1994.

[14] C. J. W. L. L.Liu, "Scheduling Algorithms for Multiprogramming in a Hard- Real-Time Environment," J. Assoc. Comput. Mach., vol. 20, no. 1, pp. 46–61, 1973.