COTS-Based High-Performance Computing for Space Applications

COTS-Based High-Performance Computing for Space Applications

S. Esposito, C. Albanese, M. Alderighi, F. Casini, L. Giganti, M. L. Esposti, C. Monteleone, M. Violante

***Abstract*— Commercial-off-the-shelf devices are often advocated as the only solution to the increasing performance requirements in space applications. This paper presents the solutions developed in the frame of the European Space Agency's HiRel program, concluded in December 2014, in which a number of techniques proposed in the past 10 years have been used to design a highly reliable system, which has been selected for forthcoming space missions. The paper presents the system architecture, describes performed evaluations and discusses the results.**

***Index Terms*— Commercial-off-the-shelf (COTS); Space Applications; Software Implemented Fault Tolerance (SIFT); Watchdog Timer; Watchdog Processor; Memory Protection; Memory Encoding; Fault Injection; Single Event Upset (SEU); Central Processing Unit (CPU).**

I. Introduction

USE of commercial off the shelf (COTS) devices is often advocated as the only answer to the growing demand for more on-board computing power in future space missions.

Although a number of high-performance computers for space applications are available that are based on COTS components, like [1] and [2], a solution entirely conceived in Europe able to provide comparable performance was not available. To solve this problem, in 2008 the European Space Agency (ESA) launched three programs focused on using COTS technology for developing space computers to provide either high reliability (HiRel program), high performance (HiP program), or high availability (HiV program).

This paper describes the results of the HiRel program, which concluded in December 2014 and resulted in a highly reliable space computer that is also able to provide high computing performance.

The main novelty of this work is in presenting a case study, where a number of known techniques developed in the past 10 years have been systematically adopted to obtain a highly reliable computer, including extensive experimental results validating robustness of the proposed architecture.

This paper was partially funded by the European Space Agency.

C. Albanese and F. Casini are with Sanitas EG s.r.l., Milan, Italy.
M. Alderighi is with Istituto Nazionale di Astrofisica (INAF), Italy.
S. Esposito and M. Violante are with Politecnico di Torino, Italy (email: stefano.esposito@polito.it).
L. Giganti and M.L. Esposti are with Thales Alenia Space Italia, Milan, Italy.
C. Monteleone is with ESA/ESTEC, Noordwijk, The Netherlands.

The results of the HiRel program were very satisfactory; indeed the computer has been recently selected for a forthcoming space mission.

The paper is organized as follows: section II summarizes the previous works regarding COTS components in space applications and software fault tolerance; section III details the proposed solution; section IV discusses the simulations performed to evaluate the proposed architecture; finally, section V draws some conclusions and proposes future enhancements.

II. Previous Works

Several approaches have been proposed to meet dependability requirements for space applications when COTS components are used. Many have focused on a purely hardware approach, most notably the Maxwell's SCS750 architecture [1], based on hardware triple modular redundancy (TMR). Other solutions focused on software means to harden the system against radiations effects. Some works have taken a hybrid approach to the problem, mixing software and hardware solutions in order to achieve the required dependability. The basic concepts of these techniques are presented in the following, to introduce the readers to the technique deployed in the HiRel computer.

A. Software Implemented Fault Tolerance

Several software approaches to fault detection and fault tolerance have been proposed. Such methods require modifying the software being executed on the system in order to detect and correct errors deriving from transient hardware faults such as Single Event Effects (SEEs). Such methods are grouped under the term *Software Implemented Fault Tolerance* (SIFT) and can be classified as purely software techniques or hybrid techniques relying on some special hardware to detect and correct an error.

Among the purely software techniques we can classify techniques oriented to protect data and techniques oriented to protect control flow. In the following we introduce some of the main techniques, which are covered in more detail in [3].

1) Data Hardening Techniques

These techniques share the main goal of protecting system data against SEEs. All require duplication of computation at some level of granularity [4][5][6][7][8]. The main idea is to duplicate all the variables in the code and to perform the same operations on each replica. Each time a variable is read,

consistency between its two replicas is checked and an error is detected in case of mismatch. Many improvements have been proposed on this idea, both to automate it [5][7] and to reduce its overhead [8].

Instead of duplicating each instruction, procedure calls can be duplicated [9]. The main advantage over instruction level duplication is the lower memory and performance overheads, whereas the main drawback is the longer error latency, due to the fact that error checks are performed after a duplicated procedure execution rather than after each instruction. Procedure calls duplication can be selective and can be merged with instruction level duplication.

The coarser granularity of data hardening technique is the program duplication, in which an entire program is executed twice in a *Virtual Duplex System* (VDS) configuration [10]. This solution has the lowest memory and performance overheads, but also the longest error latency. Performance overhead can be mitigated exploiting the multi-threading capabilities of modern architectures [11].

2) Control Flow Check (CFC) Techniques

Many techniques focus on the effects of transient faults on the control flow of a software program. All CFC techniques are based on the concepts of *Basic Block* (BB) and *Control Flow Graph* (CFG) in order to build a system capable of understanding whether the execution is proceeding as expected or an error occurred at some point.

Path Identification [12] is based on a partitioning of the CFG in which loop-free intervals are identified. A check is performed at the beginning of each loop-free interval, evaluating both a path predicate and an interval identifier, which is unique to each interval.

The *Enhanced Control Flow Checking using Assertion* (ECCA) [13] is an approach using assertions in order to detect control flow errors (CFEs). ECCA assertions are designed to cause a division by zero when an error affected the control flow of the program.

The *Yet Another Control flow Check using Assertion* (YACCA) [14][15] solution uses assertions to check for the correctness of the current control flow. YACCA has a performance overhead lower than ECCA, but it adds conditional branches that might be target of CFE. This can be avoided by moving the check at the end of the program, at the cost of longer error latency.

3) Hybrid methods

Several methods have been proposed to implement fault tolerance through a cooperation of hardware and software. This is achieved chiefly by adding special purpose hardware to the system called a *watchdog* [16]. There are several kinds of watchdog; the simpler ones are timers triggering an interrupt when the central processing unit (CPU) fails to reset them or if the watchdog does not perceive any activity on the system bus within a given timeout. More complex watchdogs are properly called watchdog processors. Several techniques have been proposed using watchdogs [17][18][19][20]. Watchdog processors are mainly used to implement CFC, reducing the overhead introduced by the techniques described in the previous section. Some proposed architectures are described below, each implementing some hybrid method for fault tolerance.

The Proton 100k [2] introduced the concept of Temporal Triple Modular Redundancy (TTMR). In TTMR software is executed three times and the correct output is selected through a majority vote among replicas' outputs. In Proton 100k, TTMR is paired to H-CORE, which is a hardware solution for Single Event Functional Interruptions (SEFIs).

The DMT (Duplex Multiplexed in Time) architecture [21] is composed of hardened software and memory protection hardware. In the DMT architecture, I/O operations are grouped before and after the processing phase. Each phase is duplicated in order to allow fault detection. Each replica has its own set of variables, in order to avoid common mode errors. Fault detection is performed using bit-by-bit comparison of the outputs. Recovery is implemented through checkpoints, stored in a safe-storage memory, protected by a special component implemented on an SEE-free chip.

The DT2 (Dual Duplex Tolerant to Transients) [21] architecture is very similar to the DMT architecture and implements the same recovery strategies, however it uses an hardware duplex system, in which two instances of the *Processing Unit Core* (PUC), composed of the microprocessor, the companion chip and the memory, are implemented. A special hardware component is used to compare the outputs of the two PUCs, and it is implemented on an SEE-free companion chip.

In [22] a hybrid approach is proposed to use hardware to reduce the code size overhead and performance overhead of fault tolerance, aimed specifically at Systems on Chip (SoCs). An *infrastructure IP* (I-IP) is introduced in the architecture, with access to the SoC bus. The I-IP implements both CFC and data hardening. CFC is implemented through specific write operations performed by the software on the I-IP registers, while data hardening is performed monitoring the bus.

III. Proposed Architecture

A. Overview

The target system used in this work is a payload computer implemented for ESA by a team including several Italian universities and companies. Main goal of the cooperation was to implement a space-worthy system using only COTS components. The system is part of ESA's HiRel program and will be part of forthcoming missions.

The system is composed of a CPU based on the PowerPC (PPC) architecture, capable of operations at 1GHz. The central memory is implemented via a double data rate II (DDR-II) memory of 1GB, used to store both code and data. It also features a non-volatile flash memory for long term storage of code and data. The interface between this payload computer and the satellite and/or instruments is implemented via several communications interfaces: a back-panel connector, two high-speed serial links (HSSL) and a bank of Space Wire connections (SpW).

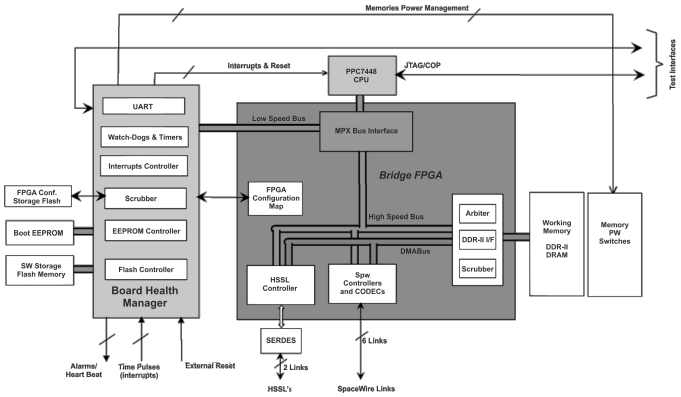

Fig. 1 The overall system Architecture

In this architecture, the CPU is responsible for executing the code, preparing both the input and the output operations. The control of all the interfaces is allocated to a companion field programmable gate array (FPGA), which implements the bridge towards memory and other high-speed functionalities, described in the next subsection. A second companion FPGA is used to implement health control functionalities and low speed functionalities, as described in section III.C.

All the components of the HiRel computer are based on COTS technology, and therefore they exhibit sensitivity to SEEs. To guarantee reliable operations, a number of SEE mitigations techniques have been included in the architecture, as detailed in the following sections. The main techniques we adopted are the following:

- Hybrid software-implemented fault tolerance was adopted to mitigate SEEs affecting the CPU. This design decision was taken by considering the high power consumption of the adopted CPU, which made hardware redundancy for SEE mitigation not suitable. In our architecture dedicated hardware functions are implemented in the companion FPGAs to support detection of SEEs affecting the CPU behavior, as described in Section III.B and III.C; moreover, the software running on the CPU is designed to exploit coarse-grain data hardening and control flow checking, as detailed in III.D.
- Information redundancy is implemented through a Reed-Solomon coding scheme and protects volatile and non-volatile memories.
- TMR is used to harden the design implemented in both the companion FPGAs. The bridge FPGA device is a high-performance SRAM-based component, therefore a configuration-memory scrubber has been implemented to detect and correct SEE in the FPGA configuration memory. The second companion FPGA is a Flash-based device, and therefore no particular mitigation is needed other than TMR of the memory elements in the design.

The focus of this paper is the overall architecture of the HiRel computer, and the description of the SEE mitigation technique we deployed to protect CPU's operations; therefore, details of memory and FPGA hardening are not addressed.

B. Bridge FPGA

The Bridge FPGA is a high-speed commercial FPGA used to implement the system's high-speed functionalities. It implements the following functionalities:

- Memory bridge;
- Memory protection unit;
- HSSL controller;
- SpW controller.

The memory bridge includes two main reliability-related functionalities: it implements the memory encoding, using a Reed-Solomon code, and the memory scrubber, which periodically operates a read-write operation in order to avoid accumulation of faults in the memory.

The memory protection unit is one of the hardware components used to implement fault detection as described in section III.D. When properly configured, this unit is able to detect write attempts to a memory area and to prevent them, enforcing a memory partitioning. The HSSL controller and the SpW controller send and receive data from their respective physical interfaces, using direct memory access (DMA) in order to operate without using CPU time.

C. Board Health Manager

The Board Health Manager (BHM) implements a series of functionalities useful to check whether the system is operating as expected. It is a low-speed component, since it does not operate at the same frequency of the Bridge FPGA. The main components implemented in the BHM are:

- Watchdogs and timers;
- Debug interface;
- FPGA configuration scrubber.

The BHM implements two watchdogs, WD1 and WD2, and two timers T1 and T2. The timers are independent from each other and offer the same functionalities. They can interrupt the processor through two dedicated interrupt lines.

WD1 offers a window of time in which the operations of the CPU must fit. The software is responsible for arming WD1 within the window boundaries. The window is composed of two times: a minimum T_m and a maximum T_M. If the time ΔT between two consecutive arm operations is $\Delta T < T_m$ or $\Delta T > T_M$, the watchdog triggers a panic reaction, stopping the whole system and signaling the problem to the platform computer. The only way to recover from this situation is to reset the system.

WD2 implements the operations necessary to implement a CFC strategy as described in section III.D. It implements a double check on signatures and timeouts. The software is responsible for configuring the WD2, communicating both the correct sequence of signatures it must expect and the maximum time that it must wait between two consecutive signatures and between the start and the first signature. The operations of the WD2 can be summarized as follows:

- Wait configuration and enable signal;
- Wait for the first signature for at most T_1;
- Wait for the second signature for at most T_2;
- ….
- Wait for the n^{th} signature for at most T_n;

- Restart from the first signature.

At each step, WD2 configures an internal counter with the time T_i associated to the i^{th} signature. WD2 is triggered either when the counter reaches zero, meaning that the software timed-out, or when the received signature is not the expected one, meaning that it is either wrong or it is not in the expected order. When WD2 is triggered it sends a non-maskable interrupt to the CPU, allowing the software to implement the proper recovery action as described in section III.D

The debug interface is used as support for the fault injection system. It is used to send pulses to an external system for synchronization purposes and to receive a freeze signal, which stops all watchdogs and timers in the BHM to allow the fault injection operation without generating any watchdog timeout.

The FPGA configuration scrubber is responsible for periodically scrubbing the configuration map of the Bridge FPGA in order to avoid misbehaviors in the Bridge FPGA as consequence of some upset in its configuration memory.

D. Software Fault Tolerance

The implemented software fault tolerance technique is a hybrid technique, using a combination of watchdogs available in the BHM and time redundancy technique.

The software is divided in three main phases:

1. *Input acquisition (I):* in this phase the data to be processed by the software are acquired. The data can be provided either by a communication device available in the system or can be provided as an array in a suitable location in memory.
2. *Processing phase (P):* in this phase the data are processed by the application code and an output buffer is prepared in memory.
3. *Output phase (O):* the results of the processing phase are prepared as output of the system and are sent to an available communication system or are copied to a suitable location in memory.

We also define a terminology used in the following:

- *Major Cycle (MajC)* is the time required to complete the execution of all three phases composing a benchmark software (I, P, O),
- *Minor Cycle (MinC)* is the time required to complete one of the phases composing the benchmark software (I or P or O).

The software is composed at least of one *MajC*, and can include several *MajCs*.

The time redundancy is implemented for each *MinC*. This means that a single execution of the software adheres to the following timeline:

1. The first input acquisition phase (I_0) is executed and immediately followed by the second input acquisition phase (I_1).
2. The first processing phase (P_0) is executed and immediately followed by the second processing phase (P_1).
3. A check is performed to ensure the correct execution so far. In this phase the signatures computed during the processing phases are compared, along with a set of flags signaling the occurrence of a hardware exception or that WD2 has been triggered. These flags are described in more detail in the following of this section. If an error is detected in this phase, either by an exception flag being set or by a mismatch in either the input or the output signatures, the recovery strategy is activated.
4. If all checks pass, the first output phase (O_0) is executed immediately followed by the second output phase (O_1).

Two recovery strategies were devised and implemented. The first strategy is a *Backward Recovery* (BWR) strategy in which the two instances of the software are re-executed when an error is detected. The second strategy is a *Forward Recovery* (FWR) strategy in which a third instance of the software is executed when an error is detected and a voter decides the correct output among the three produced. In the following of this section we describe the hardware used to support the error detection.

Two watchdogs are used to detect CFEs and SEFI. WD1 provides a time window in which the execution of the *MajC* is constrained. The lower boundary of the window is *WD1MIN*, which is the minimum time required to complete the execution of *MajC*. Its value is measured through profiling and is the best-case execution time of the *MajC* when the software is never preempted from the CPU. The upper boundary is *WD1MAX*; its value depends on the recovery strategy adopted. If a BWR strategy is adopted, the value of *WD1MAX* is four times the duration of *MajC*, whereas if a FWR strategy is selected the value of *WD1MAX* is three times the duration of *MajC*. This is due to the fact that in a backward recovery scenario, WD1 should allow the execution of the two replicas of the benchmark to be executed twice, whereas in a forward recovery scenario, the WD1 should allow the execution of just one extra *MajC*. When WD1 is triggered no recovery action is possible besides a full reset of the system, to be performed by an external controller, e.g. a platform computer. WD2 is used to check the correctness of the control flow. It allows controlling both that the path of the execution is in the CFG and that each block of the execution is performed within a timeout, which is measured through profiling. A signature and a timeout are assigned to each phase at compile time. At the end of a phase, the software sends the signature to the WD2 which compares the received signature with the expected one and it is triggered in case of mismatch or if the signature is not received within the timeout. When WD2 is triggered, it sends an interrupt to the CPU; the correspondent interrupt service routine (ISR) is responsible for setting a flag signaling that an error has been detected. The flag is read in the check phase described above. Since the detection of faults through WD2 relies on the capability of the CPU to execute an ISR, this mechanism is not able to react to SEFI or other faults stopping the CPU from executing software, which are detected by WD1.

A DDR protection unit is used in order to isolate input/output buffers of each replica of the software from the other. When properly configured, this unit forbids access to a

region of memory. For each replica two regions are defined, one for the input buffer and one for the output buffer. When an access is performed on a protected region, an interrupt is triggered and the CPU executes the associated ISR that sets a flag to signal the error. The software reads this flag during the check phase described above.

To help the comparison phase a hardware comparator is used, which is able to compare two words. During the check phase the input signatures are sent to the comparator and the result is retrieved. Afterwards, the output signatures are sent to the comparator and result is retrieved. For each comparison if a mismatch is detected, the recovery is immediately triggered.

It is to be noticed that this approach uses procedure call duplication, meaning that each phase can be implemented as a procedure called twice, thus reducing the code size overhead. Moreover, each replica of the software operates in its own memory region, which is protected by the DDR protection unit during the execution of the other replica, so to grant that a fault would not modify the data consumed or produced by the replica which is not currently in execution.

IV. Experimental Setup and Results

The proposed architecture has been validated from a performance point-of-view by measurements of the execution time, while the fault detection mechanisms were evaluated by means of fault injection simulation campaigns targeting the CPU register file, both level one (L1) and level two (L2) cache memories, and the Bridge FPGA configuration memory. In order to validate the SIFT technique a benchmark software was developed for the target system.

A. Benchmark Software

The benchmark software implemented the SIFT technique described in section III.D and was composed of a RICE[a] compression algorithm [23], of a Fast Fourier Transform (FFT) [24] implementation, and of the Dhrystone benchmark [25][26].

Each algorithm has been implemented in its own *MajC* to improve fault containment. Table 1 and Table 2 report duration of the portions of the benchmark, which were target of the fault injection, as measured through the BHM's timers. The output phase duration includes the duration of the checks performed before actually sending results on the outputs. All duration are measured when no fault is injected in the system and are referred to the redundant software, as described before.

TABLE 1. DURATION OF BENCHMARK PORTIONS WITH BACKWARD RECOVERY

	Minor Cycles			*Major Cycle*
	Input	*Proc.*	*Output*	
RICE	0.51 ms	12.41 ms	1.03 ms	13.95 ms
FFT	0.25 ms	7.35 ms	0.50 ms	8.10 ms

TABLE 2. DURATION OF BENCHMARK PORTIONS WITH FORWARD RECOVERY

	Minor Cycles			*Major Cycle*
	Input	*Proc.*	*Output*	
RICE	0.51 ms	12.41 ms	2.03 ms	14.95 ms
FFT	0.25 ms	7.35 ms	1.46 ms	9.06 ms

[a] Name of the first author of the paper that first proposed the algorithm.

B. Fault Injection

The system was evaluated for fault tolerance against faults affecting the CPU. In particular, we considered only Single Event Upsets (SEUs) affecting the instruction set architecture of the CPU, i.e., registers accessible through the instruction set, like the register file, the program counter, and the cache memories. We recognize that the CPU includes many other memory elements, like the boundary registers of the CPU pipeline, where SEUs cannot be inoculated using the approach we adopted. Although further validations are needed, for example using accelerated radiation ground testing; we relied on our fault injection method to get an initial feedback on the robustness of the HiRel computer architecture.

The following sections describe: how the fault list was generated, the injection system specifically conceived for the target system, and the fault injection results.

The following terminology is used in sections below:

- Simulation: a single fault injection consisting of a single execution of the software with a single injected fault
- Simulation Campaign or Campaign: a collection of fault injection simulations.

1) Fault list generation

As far as the CPU registers are considered, the fault list was composed of faults randomly selected from a pruned fault list. The total number of possible faults (software runtime multiplied by the number of CPU registers) is such that an exhaustive campaign is infeasible. As such the fault list has been pruned applying the following considerations:

- The software does not use all the General Purpose Registers (GPRs) and Floating Point Registers (FPRs).
- Since the software uses only single precision floating point arithmetic, it is useless to inject faults on the higher 32 bits in the FPRs.

These considerations allow for a reduction of the fault list size even though it has some specificity on the application.

Although we achieved a very reduced fault list with respect to the original one, the size of this fault list is still too large for a fault injection simulation campaign. To obtain a feasible campaign a preselected number of faults were randomly sampled from the pruned list.

2) Fault injection system

A fault injection system was specifically conceived to run the campaigns on the target system. The systems features three main components, connected as shown in Fig. 2.

Host is a workstation PC and it is responsible for:

- Generating a fault list as described in previous section;
- Selecting a fault to inject for the current simulation,
- Sending needed fault details to the Supervisor,
- Modifying target's fault injection ISR as required by the selected fault,
- Collecting and analyzing the results in order to classify fault effects at the end of each simulation.

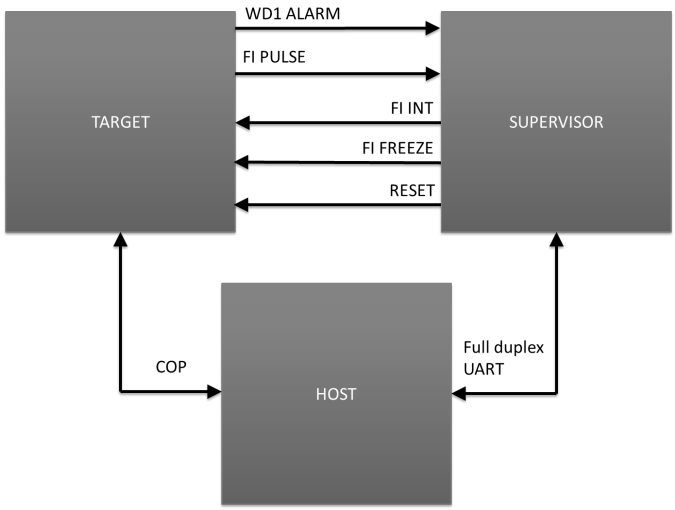

Fig. 2 Fault Injection System architecture

Supervisor is an external board responsible for:

- Generating an interrupt signal for the Target at the injection time for the selected fault;
- Freezing Target's watchdogs during fault injection;
- Reset the board between simulations.

Target is the target system on which the application software runs. In order to support the fault injection, the application software is instrumented as follows:

- Calls to a driver function used to generate a pulse have been added
- A specific fault injection routine has been implemented as the ISR for the external interrupt sent by the Supervisor.

Each fault is injected using the following procedure. Host sends to Supervisor a reset request, to which Supervisor answers resetting itself and then sending a reset signal to Target. Once the reset sequence is completed, Host sends the fault injection time to Supervisor. The fault injection time is expressed in pulses and microseconds, meaning that the Supervisor has to receive the specified number of pulses before starting a timer configured to count a given number of microseconds. When configuration is completed, Supervisor asserts a freeze signal, which stops the watchdogs on Target. After configuring Supervisor, Host loads the program image with a stub of the fault injection routine in Target memory and release Target, which executes up to a specific breakpoint before the first MajC. This portion of software is responsible for system initialization and for watchdogs' configuration; it was considered immune to faults since it only runs once at system bootstrap. At the end of this configuration phase, Host modifies the fault injection routine to include the details of the fault to inject. This is done at this point of execution to be able to use a low complexity routine without having to compile a new binary at each simulation. The binary contains a stub identical for any given fault. The modifications performed by Host at this step change the stub in an injection routine specific for the fault to inject. This step is performed after initialization because the system bootstrap procedure would overwrite the modification during the interrupt vector table loading. Once Target is configured as described, execution resumes and Target sends a pulse to Supervisor, to which Supervisor answers by de-asserting the freeze signal, allowing watchdogs to perform their tasks. During software execution, Target sends a pulse to signal the end of each MinC. Supervisor counts the pulses to reach the target MinC then it starts a timer. When such timer expires, Supervisor asserts the freeze signal and an external interrupt request line. Target reacts to the external interrupt request by executing the fault injection routine. Once the fault injection routine is completed, Target sends a pulse to Supervisor, which de-asserts the freeze signal. Then Target resumes execution until the end of the program. At the end of software execution, Host reads results from the Target memory and classifies the fault as described in section IV.C.

C. Classification of faults

The faults injected during the campaigns were classified with respect to the effect they had on the system. The following classification has been adopted and will be used in the following:

- *Silent (S):* a fault that had no detectable effect on the system.
- *WD1 timeout (WD1 TO)*: a fault that led the system to a state from which it cannot rearm the WD1 or that caused a rearm operation outside the time window defined for WD1.
- *Illegal instruction (I.I.)*: a fault that led to the fetching of an invalid instruction, for instance by modifying the PC so that it no longer points inside the code area of the program memory, or by modifying the return address of a subroutine. This is not a failure, since in a running system, the exception handler would be called and a recovery could be initiated. However, in this experimental setup, the exception associated to an illegal instruction was used by the debugger to implement breakpoints, thus no recovery was possible and the outcome was reported.
- *Signature Collision (C)*: the fault led to the production of two identical output signatures, escaping detection, even though the results were different. Faults of this class can be detected since results produced by redundant executions are different. Using a signature with a lower aliasing probability solves these errors.
- *Failure (F)*: a fault which escaped any detection mechanism and which led to the production of two identical outputs, which were nonetheless different from the expected output.

D. Experimental Results

1) Fault injection simulation campaigns on CPU registers

Three fault injection simulation campaigns were executed, the first on a non-protected version of the benchmark, in which RICE and FFT are simply called one after the other, the other two on the backward recovery version and on the forward recovery version respectively. Results of all three campaigns are presented in Table 3.

In the first campaign a fault list of 1,000 faults was

produced sampling the same pruned fault list described in section IV.B.

Without any SIFT technique, 283 faults end with a failure or a hardware exception, as shown in the second row of Table 3. In plain software, this would mean that wrong results or no results are provided to the user. This campaign provided a baseline reference for the results achieved in the subsequent campaigns, allowing a meaningful evaluation of both recovery strategies. Only in this campaign, hardware exception (H.E.) includes any possible hardware exception cause, since no exception handling mechanism is implemented in the plain software.

In the second campaign 2,000 faults sampled from the fault list described in section IV.B were injected on the backward recovery version. Results are shown on the third row of Table 3. There are no failures and a limited number of illegal instructions. Detected WD1 timeouts are due to faults persisting in the register file and leading to the repetition of the backward recovery, until the WD1 is triggered, or to faults locking the CPU in an infinite loop between checkpoints, thus preventing detection of the WD2 error. A significant increase in silent faults is observed, with respect to the results of the fault injection simulation campaign on the plain version of the software. This is due to the successful recovery performed by the software protection mechanisms. A little number of signature collisions is observed, due to the very simple signature used in our simulations. A more complex signature will lead to complete removal of this class of faults.

In the third campaign 2,000 faults sampled from the fault list described in section IV.B were injected on the forward recovery version. Results are shown on the fourth row of Table 3. Similarly to what was observed for the backward recovery mechanism, there are no failures and a limited number of illegal instructions. The faults detected through WD1 timeout are roughly half the number of faults of the same class detected in the backward recovery version, due to the lack of a backward recovery, which can lock the CPU in a loop if a fault persists in the register file in between recoveries.

TABLE 3 FAULT INJECTION RESULTS ON CPU REGISTERS

RECOVERY	*S.*	*WD1 TO*	*C.*	*I.I.*	*H.E.*	*F.*	*INJ.*
NONE	717	-	-	-	102	181	1000
BACKWARD	1790	165	42	3	-	0	2000
FORWARD	1843	95	34	28	-	0	2000

S.: silent; WD1 TO: WD1 timeout; C.: signature collision; I.I.: illegal instruction; H.E.: hardware exception; F.: failure; Inj.: injected.

2) Fault injection simulation campaigns on CPU caches and Bridge FPGA configuration memory

A similar set of campaigns was performed to evaluate the effects of faults injected in the CPU caches and in the Bridge FPGA (Tables 4 and 5). Results show that the proposed architecture is virtually immune to faults injected in caches or in the Bridge FPGA. The caches are indeed protected either by a parity code or by an ECC by hardware means, while the Bridge FPGA is protected by the BHM through a bit stream scrubber. These fault injection campaigns used a more complex Supervisor system, which was also in charge of timing the communication of results through the Space Wire links.

The results of the campaign targeting the cache memories (Table 4), showed that the implemented SIFT strategy is able to recover faults with no failure, WD1 timeout or collision in over 2000 faults injected, whereas results of the campaign targeting the Bridge FPGA configuration memory (Table 5), show a very low rate of WD1 TO or Transmission Errors (T.E.), i.e. faults leading to an erroneous timing of the communication with respect to the timing a platform computer would expect.

TABLE 4 FAULT INJECTION RESULTS ON CACHE MEMORIES

RECOVERY	*S.*	*WD1 TO*	*C.*	*INJ.*
BACKWARD	2250	0	0	2250
FORWARD	2250	0	0	2250

TABLE 5 FAULT INJECTION RESULTS ON THE BRIDGE FPGA CONFIGURATION MEMORY

RECOVERY	*S.*	*WD1 TO*	*T. E.*	*INJ.*
BACKWARD	995	2	3	1000
FORWARD	994	1	5	1000

3) Overhead evaluation

Table 6 shows performance overhead introduced by both recovery strategies in fault free executions. Results show a performance overhead less than 100% in both cases, thanks to the coarse granularity used in implementing the SIFT strategy. Forward recovery shows a slightly worse behavior due to an additional check, performed in any case to decide whether to execute the third replica or not.

TABLE 6. PERFORMANCE OVERHEAD EVALUATION.

RECOVERY	*EXECUTION TIME*	
NONE	13.88 MS	-
BACKWARD	22.64 MS	+63%
FORWARD	24.60 MS	+77%

Executable image size overhead data are reported in Table 7. The Code column reports the size of executable code. It shows code memory area increase is reduced thanks to the coarse granularity approach to time redundancy. Even though the forward recovery version adds some extra check which are not present in the backward recovery, the lower complexity of the forward recovery strategy with respect to the backward recovery strategy allows for an equivalent overhead. The Data column reports size of all data included in the program image, i.e. initialized, unitialized, and read-only data. It shows a significant increase, due to the need to duplicate, for backward recovery, or triplicate, for forward recovery all input and all output areas of each MajC; the difference among the recovery strategies accounts also for the higher overhead of the forward strategy in this area. The Total column reports the total size of the stripped executable image. The difference on each row between this column and the sum of the other two columns is due to other parts of the executable image which are of little concern in this analysis.

Overall, the forward strategy shows an higher overhead in

both performance and memory area occupation than the backward strategy.

TABLE 7. EXECUTABLE IMAGE SIZE OVERHEAD EVALUATION

RECOVERY	*CODE*		*DATA*		*TOTAL*	
NONE	1892 B	-	183 KB	-	237 KB	-
BACKWARD	1968 B	+4%	250 KB	+37%	317 KB	+34%
FORWARD	1968 B	+4%	264 KB	+44%	336 KB	+42%

V. CONCLUSIONS

This paper proposes a case study where a European space computer based on COTS technology is presented. By combining software techniques with special hardware implementing Watchdog Timer, Watchdog Processor, Memory Partitioning, Memory Encoding, and two scrubbers (one for the Main Memory and one for the Bridge-FPGA configuration memory) the proposed architecture can grant high level of reliability while containing performance and memory area overhead to a reasonable level, as results show. Fault injection simulations show no SEE-induced failures and a limited number of detected but not recovered faults, which should either be addressed by a platform computer using a stand-by spare or by improving the signature computation algorithm in order to reduce aliasing.

Simulations also compared two recovery strategies, named backward and forward recovery. Results shows that forward recovery achieves a better fault recovery, while introducing a slightly higher overhead in both performance and memory area occupation. Although accelerated radiation ground testing experiments are needed to further validate the HiRel computer (for example allowing evaluating the impact of SEEs in the memory elements not accessible though fault injection, like the CPU pipeline boundary registers), the results we obtained suggest a very high tolerance against SEEs.

In the foreseeable future, space mission performance requirements will grow out of the capabilities of contemporary single-core base COTS solutions. Although the strategy presented in this paper could be ported to a multicore system, for instance to parallelize the redundant executions, multicore system's use in mission-critical systems still poses many problems, especially when hard-real time constraints are into play. Since most multicore systems are actually System-on-Chips, time interference among the cores, resource sharing and susceptibility to SEE effects of the configuration registers, are among the main challenges that future designs will have to address.

REFERENCES

[1] 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," *Proc. RADECS 2003*, pp. 347-352, Sep. 2003.

[2] D. R. Czajkowski, M. P. Pagey, P. K. Samudrala, M. Goksel, and M. J. Viehman, "Low Power, High-Speed Radiation Hardened Computer & Flight Experiment,", *Aerospace Conference, 2005, IEEE,* pp. 1-10, IEEE, 2005

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

[4] M. Rebaudengo, M. Sonza Reorda, M. Torchiano, and M. Violante, "Soft-error detection through software fault-tolerance techniques," *Int'l Symp. on Defect and Fault Tolerance in VLSI Systems*, 1999, pp. 210-218, IEEE, 1999

[5] M. Rebaudengo, M. Sonza Reorda, M. Torchiano and M. Violante, "A source-to-source compiler for generating dependable software," in *Proc. of the 1st IEEE Int'l Workshop on Source Code Analysis and Manipulation*, pp. 33-42, IEEE, 2001

[6] P. Cheynet, B. Nicolescu, R.Velazco, M. Rebaudengo, M. Sonza Reorda, and M. Violante, "Experimentally evaluating an automatic approach for generating safety-critical software with respect to transient errors," *IEEE Trans. on Nuclear Science*, vol. 47, no. 6, pp. 2231-2236, 2000.

[7] N. Oh, P. P. Shirvani, and E. J. McCluskey, "Error detection by duplicated instructions in super-scalar processors," *IEEE Trans. on Reliability*, vol 51, no. 1, pp. 63-75, 2002

[8] A. Benso, S. Chiusano, P. Prinetto, and L. Tagliaferri, "A C/C++ source-to-source compiler for dependable applications," in *Proc. Int'l Conference on Dependable Systems and Networks*, pp.71-78, IEEE, 2000

[9] N. Oh and E. J. McCluskey, "Error detection by selective procedure call duplication for low energy consumption," *IEEE Trans. on Reliability*, vol. 51, no. 4, pp. 392-402, 2002

[10] K. Echtle, B. Hinz, and T. Nikolov, "On hardware fault detection by divers software," *Proc. of the 13th Int'l Conference on Fault-Tolerant Systems and Diagnostics*.

[11] S.K. Reinhardt and S. S. Mukherjee, "Transient fault detection via simultaneous multithreading," *Proc. of the 27th Int'l Symp. on Computer Architecture*, pp. 25-36, 2000.

[12] S. S. Yau and F.-C. Chen, "An approach to concurrent control flow checking," *IEEE Trans. on Software Engineering*, no. 2, pp. 126-137, 1980.

[13] Z. Alkhalifa, V.S. Nair, N. Krishnamurthy, and J.A. Abraham, "Design and evaluation of system-level checks for on-line control flow error detection," *IEEE Trans. on Parallel and Distributed Systems*, vol. 10, no. 6, pp. 627-641, 1999.

[14] O. Goloubeva, M. Rebaudengo, M. Sonza Reorda, and M. Violante, "Soft-error detection using control flow assertions," *Proc. of the 18th IEEE Int'l Symp. on Defect and Fault Tolerance in VLSI Systems*, pp. 581-588, IEEE, 2003.

[15] O. Goloubeva, M. Rebaudengo, M. Sonza Reorda, and M. Violante, "Improved software-based processor control-flow errors detection technique," *Proc. of the Annual Reliability and Maintainability Symp.*, pp. 583-589, IEEE, 2005

[16] J.R. Connet, E. J. Pasternak, and B. D. Wagner, "Software defenses in real-time control systems," *Digest of the 1972 Int'l Symp. On Fault-Tolerant Computing*, pp. 94-99, 1972.

[17] M. Namjoo and E. J. McCluskey, "Watchdog processors and capability checking" *Twenty-Fifth Int'l Symp. on Fault-Tolerant Computing, Highlights from Twenty-Five Years*, p. 94, IEEE, 1995.

[18] S. Saib, "Distributed architectures for reliability," *Proc. of the AIAA Computers in Aerospace Conference II*, Los Angeles, 1979.

[19] A. Mahmood, A. Ersoz, and E. J. McCluskey, "Concurrent system-level error detection using a watchdog processor," 1985.

[20] F. E. Allen, "Control flow analysis," *SIGPLAN*, No. 5, Vol. 7, pp. 1-19, Jul 1970

[21] M. Pignol, "DMT and DT2: two fault-tolerant architectures developed by CNES for COTS-based spacecraft supercomputers," *12th IEEE Int'l On-Line Testing Symp, 2006. IOLTS 2006*, pp. 10-pp, IEEE, 2006

[22] P. Bernardi, L.M.V. Bolzani, M. Rebaudengo, M. Sonza Reorda, J.J. Rodríguez-Andina, M. Violante, "A new hybrid fault detection technique for systems-on-a-chip," *IEEE Trans. on Computers*, Vol. 55, No.2, pp.185-198, Feb. 2006

[23] R. Rice, J. Plaunt, "Adaptive variable-length coding for efficient compression of spacecraft television data," *IEEE Trans. on Comm.*, Vol 19 No. 6, pp. 889-897, 1975

[24] J. W. Cooley, J. W. Tukey, "An algorithm for the machine calculation of complex Fourier series," *Math. of computation*, vol. 19, no. 90 pp. 297-301, 1965

[25] R. P. Weicker, "*Dhrystone: a synthetic systems programming benchmark." Comm. of the ACM*, vol. 27, no. 10, pp. 1013-1030, 1984

[26] R. P. Weicker, "*Dhrystone benchmark: rationale for version 2 and measurement rules," AcM SIGPLAn notices*, vol. 23, no. 8, pp. 49-62, 1988.