A Novel Co-Design Approach for Soft Errors Mitigation in Embedded Systems

Sergio Cuenca-Asensi, Antonio Martínez-Álvarez, Felipe Restrepo-Calle, Francisco R. Palomo, Hipólito Guzmán-Miranda, and Miguel A. Aguirre

Abstract—**There is an increasing concern about the mitigation of radiation effects in embedded systems. This fact is demanding new flexible design methodologies and tools that allow dealing with design constraints and dependability requirements at the same time. This paper presents a novel proposal to design radiation-tolerant embedded systems combining hardware and software mitigation techniques. A hardening infrastructure, which facilitates the design space exploration and the trade-offs analyses, has been developed to support this fault tolerance co-design approach. The advantages of our proposal are illustrated by means of a case study.**

Index Terms—**Fault tolerance, radiation effects, reliability.**

I. Introduction

IN the last decades, the progressive miniaturization of electronic components has led to important advances in microprocessors, such as the increase of performance, the growth of number of gates, and the reduction of the silicon area. However, as technology shrinks, voltage source level and noise margins are reduced as well, making electronic devices become less reliable and *CMOS* circuits more susceptible to *transient faults* induced by radiation. These intermittent faults do not provoke a permanent damage, but may cause incorrect circuit behavior by altering signal transfers or stored values, the so called *soft errors* [1], [2].

The application of redundant hardware is the usual way to face reliability problems: from low level structures using techniques like *Error Detection and Correction* (*EDAC*), *parity bits*, *Triple Modular Redundancy* (*TMR*); to more complex components like functional units [3], co-processors [4]; or even exploiting the multiplicity of hardware blocks available on multi-threaded/multi-core architectures [5]–[7]. More recent techniques [8] propose selective hardening of the system, adding protection only to the most vulnerable parts. Most hardware approaches provide a high effective solution for transient faults. However, these techniques are unfeasible in many cases due to the high costs involved, in terms of invested time and economic expenses.

Motivated by the need of low cost solutions, several proposals based on redundant software have been developed [9]–[11]. Most software-based approaches are aimed to detect faults, some of them apply redundancy to high-level source code by means of automatic transformation rules [12], whereas some others use instruction redundancy at low level (assembly code) in order to reduce the code overhead and performance degradation, and improve the detection rates [13]–[15]. Only a few of these techniques have been extended to allow the recovery of the system [12], [16]. Although the software-based approaches are more cost effective than the hardware-based ones, they cannot achieve the same performance and reliability, since they have to execute additional instructions [17]. However, it is worth noting that some of these techniques have already been used in systems for satellites and real space missions [18].

In the case of embedded systems, there are large domains of applications where factors like cost, power and performance are as important as reliability. For these kinds of applications, optimal mitigation strategies can be found combining both software and hardware aspects from different techniques. Some of these hybrid approaches have been proposed in recent years showing promising results [19], [20]. However, they are very specific and they lack the necessary flexibility to get the best trade-offs between design constraints and dependability requirements.

In this context, the first contribution of this work is the use of the co-design methodology [21] in the development of hybrid soft errors mitigation strategies. In other words, the process of exploring the design space of hardware and software techniques to achieve a customized fault tolerant version of the system, which best meets the requirements of the application. To achieve this goal, and as a second contribution, two suites of tools have been developed: a *Software Hardening Environment* [22] aimed to implement, automatically apply and evaluate software-only fault tolerant techniques; and a *FPGA*-based fault emulation tool, called *FTUnshades* [23], that permits the assessment of several reliability metrics of the overall embedded system. In this work, mitigation techniques are applied to a high abstraction level, so the final deployment platform could be either *ASIC* or *FPGA* (flash or anti-fuse technologies). In case of SRAM-based

Manuscript received September 17, 2010; revised December 05, 2010 and January 25, 2011; accepted January 25, 2011. Date of publication April 07, 2011; date of current version June 15, 2011. This work was supported by the 2010 Research National Plan in Spain of the Ministry of Science and Innovation with the project "Integral Analysis of Digital Circuits and Systems for Aerospace Applications (RENASER+)" (TEC2010-22095-C03-01) and by the Generalitat Valenciana in Spain with the research project "Aceleración de algoritmos industriales y de seguridad en entornos críticos mediante hardware" (GV/2009/098).

S. Cuenca-Asensi, A. Martínez-Álvarez, and F. Restrepo-Calle are with the Computer Technology Department, University of Alicante, Carretera San Vicente del Raspeig s/n, 03690 Alicante, Spain (e-mail: sergio@dtic.ua.es; amartinez@dtic.ua.es; frestrepo@dtic.ua.es).

F. R. Palomo, H. Guzmán-Miranda, and M. A. Aguirre are with the Department of Electrical Engineering, University of Sevilla, Camino de los Descubrimientos, 41092 Sevilla, Spain (e-mail: rogelio@zipi.us.es; hipolito@zipi.us.es; aguirre@zipi.us.es).

Color versions of one or more of the figures in this paper are available online at http://ieeexplore.ieee.org.

Digital Object Identifier 10.1109/TNS.2011.2112379

FPGAs, additional protections mechanisms should be considered for the configuration memory, such as *scrubbing*, partial reconfiguration, or re-routing design [24], [25].

As case study for validating our approach, we have explored the fault-tolerant co-design of an embedded application based on the *PicoBlaze* soft-microprocessor [26]. Different trade-offs among code overhead, performance, fault coverage and hardware costs have been investigated as well.

This paper is organized as follows: next Section presents the proposed hardening infrastructure; Section III describes the mentioned case study, including the experiments and their results; and finally, Section IV summarizes some concluding remarks and suggests directions for future work.

II. Hardening Infrastructure

The hardware/software *co-design* methodology is applied to the development of hybrid soft errors mitigation strategies for embedded systems.

Although combining software and hardware fault mitigation techniques can be suitable in some cases, it could result in an over-redundant design with some unacceptable features such as high area and power costs, and disproportionate penalties in performance. Therefore, we propose a fine-grained exploration of the design space by means of the selectively controlled application of protection approaches on both sides, software and hardware. This controlled selectiveness consists of inserting protections only to the most vulnerable parts of the system. In addition, it is possible to choose where the protection is inserted, whether software or hardware, since this selection will affect the system differently. In this way, the designer is able to fine-tune a tailored fault mitigation hybrid approach.

The first step of our approach is the specification of the system requirements from the embedded application point of view. Such requirements can be either design constraints or dependability parameters associated to each specific application. In general, design constraints are related to silicon area, performance, power consumption and hardware cost; whereas, dependability parameters are concerned with fault coverage, reliability, availability, safety, security, and recovery time. The co-design flow is driven by the application specifications and the design decisions are motivated by taking into account both, design constraints and dependability parameters.

The adoption of several software fault mitigation techniques can determine a set of suitable implementations for the system. These software techniques can be fully or selectively applied. Each resulting version is then evaluated to estimate the overheads, in terms of code size and execution time, compared to the non-hardened program. In addition, if there is an available simulation-based reliability evaluation tool, it could be used to make preliminary dependability analyses on software.

Based on the mentioned above evaluations, and according to the system specifications, the best candidates are selected to be tested on the real microprocessor implementation. It is necessary to evaluate the reliability offered by each hardware/software configuration. At this point, the designer can explore several trade-offs among software size, performance, and reliability.

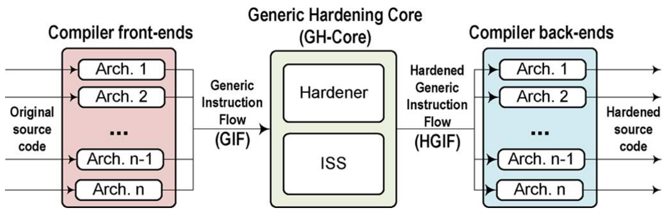

Fig. 1. Software Hardening Environment.

In case that still there is not any configuration that meets all the requirements, the protection strategy has to be complemented by applying hardware-based techniques. Therefore, the designer has to decide suitable strategies to protect the hardware, looking for protecting the most vulnerable parts of the design and inserting redundancy in those parts of the microprocessor where software-based techniques cannot do it. In this way, a new parameter should be considered within the trade-offs analysis: the hardware cost.

After combining the best candidates of the software side with the protected versions of the hardware, the hybrid fault mitigation approach is then evaluated in terms of reliability.

The optimal design is not necessarily a single point in the design space but may result in a set of hardware/software implementations that meet both design constraints and dependability requirements. Nevertheless, the designer has sufficient information at this point to determine the best system configuration based on the trade-offs exploration.

This design approach is supported by two suites of tools: the *Software Hardening Environment* and the fault injection based reliability evaluator called *FTUnshades*.

A. Software Hardening Environment

This environment lets the user to design and implement software-only fault tolerance techniques, and also to automatically apply them to the programs. It is made up of a generic *Hardener*, and a generic *Instruction Set Simulator—ISS*, jointly with several compiler front-ends and back-ends to deal with different microprocessor targets (see Fig. 1).

A given compiler front-end takes the original source code from a supported target, performs lexical, syntactical and semantic analyses, and finally generates a *Generic Instruction Flow* (*GIF*) as output. This flow represents an intermediate high level abstraction of a program. Then the hardening tasks are performed within the *Generic Hardening Core*. Finally, the *Hardener* produces a *Hardened-GIF* (*HGIF*) which is then re-targeted to a custom supported microprocessor. The automatic generation of hardened code is guided by instruction-level assembly transformation rules.

Using this scheme, the hardening environment is also flexible as it is possible to process a code written for a supported architecture and generate a protected code which is targeted to the same original architecture or to a different one by means of one of the available back-ends.

Among the range of advantages our *Software Hardening Environment* has, its *Microprocessor Generic Architecture* is worth being highlighted as it provides a workspace for the *Generic*

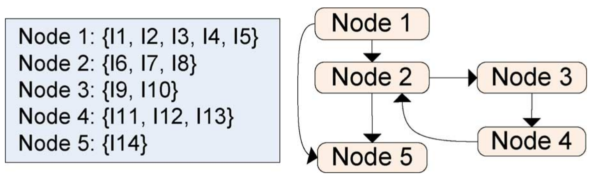

Fig. 2. Control Flow Graph—*CFG*.

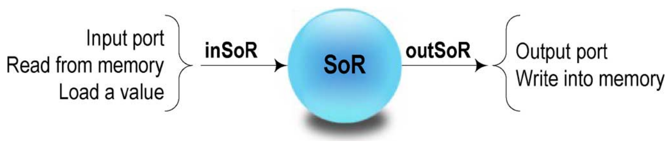

Fig. 3. Sphere of Replication—*SoR*.

Hardening Core and gathers every common element from different architectures. This feature facilitates the design and implementation of the state-of-the-art software-based techniques in an architectural independent way.

The identification of the *Control Flow Graph* (*CFG*) and the insertion of instructions into the source code are the keys for most software-based techniques [14], [13], [16]. In this way, our tool also provides the necessary functionalities to suitably manage the memory map in compilation time and also to analyze the programs *CFG*.

Regarding *Memory Management* tasks, it is possible to identify the memory sections, and carry out modifications to them in order to keep the memory map updated when redundant instructions are inserted. This is possible because of, in a similar way than other approaches, it is assumed that the code being hardened does not exploit dynamic memory allocation, i.e., every data structure is defined statically at compilation time. This is not a significant limitation for developers of embedded applications, who sometimes are forced to code standards that avoid dynamic memory usage [27].

The *CFG* is represented by a directed graph, where each node is defined by a *basic block* of the program. A *basic block* is a group of instructions that are executed sequentially, without any jump instruction or function call, excepting possibly the last instruction. Moreover, a *basic block* does not contain instructions being the destination of any call/jump instruction, with the exception of the first instruction. The flow control changes are represented in the graph as links among nodes. Fig. 2 shows an example of a simple *CFG*.

The *Generic Hardening Core* has two main components: the *Hardener* and the *ISS*.

The *Hardener* comprises an Application Programming Interface–*API* of common hardening procedures that can be used to design software-based techniques. The *Hardener* offers complete control to the designer to configure the protection strategy by means of its available command-line options. Also, it allows designing the hardening strategy by means of partial application of one or more software-based techniques. For instance, if during the hardening process, it finds that there are not enough available resources to go on adding redundancy (e.g., there are not enough registers to replicate), then the process can continue without replication until some resources are released. Furthermore, the selective hardening can be controlled by the designer, being able to determine which resources must be hardened.

Considering hardening purposes, as it was suggested by Reis *et al.* [15], the *Hardener* manages in a special way those instructions whose function implies to cross the borders of the *Sphere of Replication* (*SoR*) [28]. The *SoR* is the logic domain of redundant execution. Therefore, when an instruction directs some data to enter the *SoR* (e.g., reading a port, loading a value into a register or reading from memory), we will classify it as *inSoR*; and consequently when an instruction provokes data going out from the *SoR* (e.g., writing on a port, storing into the memory), we will classify it as *outSoR* (Fig. 3). Note that the boundaries of the *SoR*, and consequently the coverage of the protection, could change according to the implemented technique; e.g., including/excluding the memory subsystem inside the *SoR* jointly with the register file, or even moving some selectively-chosen registers from the register file inside/outside the *SoR*.

Taking into account that when an instruction sends data outside of the *SoR*, it may provoke an unrecoverable error (if those data are corrupted), it would be desirable to verify data correctness before they leave the sphere. Thus, in this work we propose that the *nodes* (*basic blocks*) of the *CFG* should be subdivided into *sub-nodes* after each instruction classified as *outSoR*.

On the other hand, the *ISS* assists the designer in the implementation of new software-based techniques. As usual for the instruction set simulators, the *ISS* generates information about the state of the resources of the architecture during and after the simulation process. Likewise, it performs different analyses on the *GIF* and *HGIF* to check the correctness of the hardening process, and offers useful information about the hardening: code overhead, performance degradation, fault-coverage estimations, program resources utilization (i.e., number of accesses to each microprocessor register, register *lifetime*), etc. Therefore, the *Hardener+ISS* offer a rich set of co-design parameters that can be used to perform an exhaustive soft-wide design space exploration.

Moreover, in order to evaluate the reliability provided by the software techniques, the *ISS* is also able to simulate *Single Event Upset—SEU* faults by means of the *bit-flip* of a single bit. Although the *ISS* has access to the most important architecture resources such as the register file, the program counter, the stack pointer, and the ALU flags; it does not have access to the micro-architectural resources like those registers in the pipeline. Thus, the fault-coverage results obtained by the *ISS* should be considered only estimations, which are useful to easily compare different software-based fault mitigation techniques, but that should be verified by another tool that permits a full access to every micro-architectural resource (*FTUnshades*).

B. SEU-Emulation Tool: FTUnshades

The second main component of the hardening infrastructure is *FTUnshades* [23]. It is a *FPGA*-based platform for the study of digital circuit reliability against radiation-induced soft errors. *SEUs* affecting the circuit are emulated by inducing bit-flips in the circuit under study by means of dynamic partial reconfiguration. The system is composed of a *FPGA* emulation board and a suite of software tools for testing the emulated design and analyzing test results.

FTUnshades permits to inject faults in a selective way on the different parts of the circuit, which is the base for the reliability analysis in microprocessors [29]. This tool provides information of the impact on the reliability of each one of microprocessor submodules due to hierarchical injection [30]. Therefore, it can find the best candidates from the internal parts to be protected.

Moreover, this is a non-intrusive tool, i.e., it permits to have complete access to all the physical resources of the design, but without modifying them. This is possible by using the available resources in the FPGA configuration layer.

In this work, *FTUnshades* is used to assess the reliability of the full hardware/software mitigation strategy applied to the embedded system. It is worth noting that as the test campaigns are executed over the physical implementation of the system, it is possible to evaluate the designs even at micro-architectural level. Faults are injected only in the FPGA user registers (not in the configuration layer), considering that the final deployment technology will be *ASIC* or non-volatile *FPGA*.

The fundamental difference between simulating faults with the *ISS* and emulating them with *FTUnshades* is not in the accuracy of the results, but instead in the coverage of the injection tool; with *FTUnshades*, faults can be injected in all memory elements of the microprocessor, including embedded memories and internal flip-flops, and not only in microprocessor registers. Moreover, only with *FTUnshades* the hardened versions of the hardware can be evaluated.

III. Case Study

In order to validate our proposal, we have designed and evaluated a number of radiation-tolerant versions of an embedded system. The co-design space exploration is driven by a well known application: *mmult*—matrix multiplication. The hardware is composed of a technology-independent (i.e., valid for *ASIC* or *FPGA*) version of *PicoBlaze* developed for this work (*RTL PicoBlaze*). This microprocessor is an 8-bit width soft-core widely used within *FPGA*-based applications. This version is cycle accurate and RTL equivalent to the original *PicoBlaze-3* version of the microprocessor [26]. It supports the following main features: 16 byte-wide general-purpose data registers (numbered from 0 to F in hexadecimal notation), 1 K instructions of programmable on-chip program store, byte-wide Arithmetic Logic Unit (ALU) with CARRY and ZERO indicator flags and 64-byte internal scratchpad RAM.

We have had two sources of variation when tuning our system, the different hardened versions of the software, and the selective hardening of the microprocessor. Both of them are controlled by a sort of design parameters of interest (set of registers hardened by software or hardware, fault tolerance technique, etc.).

A. Development of Hybrid Hardening Strategies

On the software side, an adaptation of *SWIFT-R* [16] has been implemented. This is a software-only recovery technique that consists of triplication of data and instructions, jointly with the insertion of verification points to check data consistency (by means of majority voters).

TABLE I
Registers Usage for *mmult*

Register	# Write	# Read	# Read/Write	Lifetime [%]
0	340	357	183	55.3%
A	20	83	45	85.7%
D	27	135	108	48.7%
E	27	135	108	52.2%
F	9	9	27	83.3%

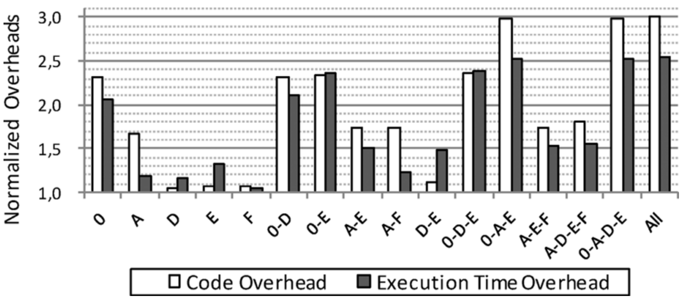

Fig. 4. Normalized code and execution time overheads.

The flexibility of our *Software Hardening Environment* allows to automatically apply specific techniques in a selective way. In this case study, *SWIFT-R* has been implemented to be incrementally applied to several subsets of selected registers from the microprocessor register file. This is possible by moving the remaining registers outside of the *SoR*. For instance, a hardened version of the software obtained by applying *SWIFT-R* only to the register subset $\{0\text{–}A\text{–}E\}$, means that only these registers are protected (from the sixteen possible *PicoBlaze* registers).

Moreover, to help the designer prioritize the registers to protect, the *ISS* gathers information about the number of clock cycles the microprocessor registers have to maintain a valid value, i.e., register *lifetime*, and the number of accesses to registers (*number of times* the registers have been used for writing, reading or reading/writing operations). The first parameter has a high impact on reliability, since the higher the *lifetime* is, the longer the register is prone to *soft errors*. The second parameter, by contrast, has a remarkable impact on code and execution time overheads, as a highly accessed register protected by software has a higher number of redundant instructions. Both parameters need to be taken into account when designing the protection strategy on the software side. Table I presents the information about registers usage (number of accesses and *lifetime*) for the *mmult* program (note that this program only uses the following registers: 0, A, D, E, and F). Register *lifetime* is expressed as the percentage of the total program time. Those registers with the highest *lifetime* are A and F (85.7% and 83.3% of the total clock cycles of the program, respectively). In addition, notice that these registers are the least accessed, thus their protection does not significantly impact the execution time and code overheads. Consequently these registers are the first candidates to be hardened.

Fig. 4 shows the overhead results for several selectively hardened registers subsets using our adaptation of *SWIFT-R*. These results are normalized with a baseline built with the non-hardened version. Notice that when *SWIFT-R* is applied to some highly accessed registers, such as *0*, the overheads increase considerably. However, protecting these registers does not guarantee improved reliability, since the vulnerability of each register depends on its *lifetime* during the program execution. The *lifetime* parameter is not always correlated with the number of times that the register is accessed.

On the hardware side, the fault tolerant co-design strategy was complemented by incrementally hardening some microprocessor resources. It was done by manually applying *TMR* to different subsets of micro-architectural registers. The following versions have been developed:

- *P0*: non-hardened *RTL PicoBlaze*.
- *P1*: microprocessor with hardware redundancy for Program Counter—*PC*, Flags and Stack Pointer—*SP*.
- *P2*: hardware redundancy for all registers in the Pipeline.
- *P3*: hardware redundancy for *PC*, Flags, *SP*, and Pipeline.
- *P4*: full protected version, i.e., microprocessor with redundancy for Register file, *PC*, Flags, *SP*, and Pipeline.

Using the information provided by the *ISS* and the synthesis tools, the designer can select the best candidates for further analyses. However, for demonstration purposes, all the systems have been evaluated using the *FTUnshades* (in total 69 versions of the system, 17 software versions running on the *P0* to *P3* microprocessors, plus the non-hardened program running on the *P4* microprocessor).

B. Reliability Evaluation

In the following tests we focus on the type of transient faults known as *Single Event Upset* (*SEU*). This effect is induced by radiation; succinctly, it is caused by direct or indirect ionization resulting from the impact of energetic particles on an electronic component. In order to represent it, the bit-flip fault model is used, in which only one bit-flip of a storage cell occurs throughout the execution. Despite its simplicity, this model is widely used by the scientific community because it closely matches the real fault behavior [31].

The fault is injected by means of a bit-flip in a randomly selected bit from the microprocessor in a randomly selected clock cycle during the program execution. The memory is assumed to have its own protection mechanisms, thus, memory bits will not be considered for fault injection. In addition, faults have been classified according to their effect on the expected program behavior as it was first proposed by Mukherjee *et al.* [32]. If the fault has not been detected/corrected and provokes the program to finish with an erroneous output, it has been classified as *Silent Data Corruption* (*SDC*). In case the system completes its execution and produces the expected output after a fault is injected, the bit that was flipped, and consequently the fault itself, are classified as *unnecessary for Architecturally Correct Execution* (*unACE*). Finally, when the fault causes an abnormal program termination or an infinite execution loop, the fault has been classified as a *Hang*. Note that *SDC* and *Hang* are both undesirable effects (categorized together as *ACE* faults).

We have performed a first experiment in order to calibrate the *FTUnshades*. It was aimed to calculate the minimal number of fault injection tests needed to obtain accurate reliability results. In this way, an incremental fault injection campaign has been carried out against the register file of the *P0* microprocessor. The non-hardened program was chosen for the tests because it

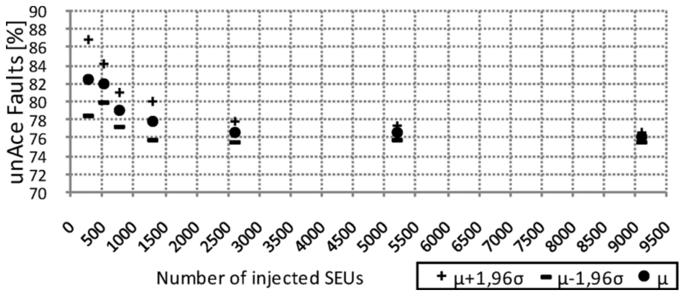

Fig. 5. Incremental fault injection campaign for calibration purposes in *FTUnshades*.

represents the worst case scenario. The results (Fig. 5) show that the 95% confidence interval is less than $\pm 1.0\%$ after 5 200 fault injection tests (emulating only one *SEU* in each test).

A second experiment using test campaigns in *FTUnshades* was performed to evaluate the overall reliability of all the versions of the hardware/software configurations. Every test campaign makes selective attacks on the microprocessor register subsets (including register file, *PC*, flags, *SP* and pipeline). In each register subset, 5 200 fault injection tests (one fault per test) have been performed, i.e., a total of 26 000 faults have been emulated in each version. Faults have been emulated in a randomly selected clock cycle from all the workload duration. Fig. 6 presents the fault classification percentages obtained for each version. These results are the weighted average of the results from the selective attacks to the internal microprocessor register subsets, assuming the same fault probability for all bits on target.

Note that the *SWIFT-R* technique offers a considerable reliability increase, even in the non-hardened hardware (up to 95.38% *unACE* faults in the full hardened program), which is much higher than the reliability of every hardware-hardened approach using the non-hardened program. Results for the *P4* microprocessor are not showed in Fig. 6 because 100% of the injected faults have been classified as *unACE*, as expected. Furthermore, notice that combining *SWIFT-R* with hardware protection only to few critical registers, such as *PC*, Flags, and *SP* (*P1* approach), produces a remarkable reliability increase (up to 98.32% *unACE* faults).

C. Discussion

Taking into account the requirements of the application under design, the analysis of overheads jointly with reliability results can guide further decisions during the co-design process. This analysis facilitates the finding of the solutions having the best reliability/overhead compromise. For instance in this case study, *SWIFT-R* applied only for the register subset $\{A, D, E, F\}$ running in the *P3* microprocessor is an interesting choice, because it offers both, high reliability (98.68% of *unACE* faults), and acceptable code and execution time overheads ($\times$ 1.80 and $\times$ 1.56 respectively).

Although reliability is higher when combining software-hardened programs with hardware-redundant approaches (for instance, up to 99.10% *unACE* faults for the *P3* micro), hardware costs are also higher, which is an important fact that has to be considered.

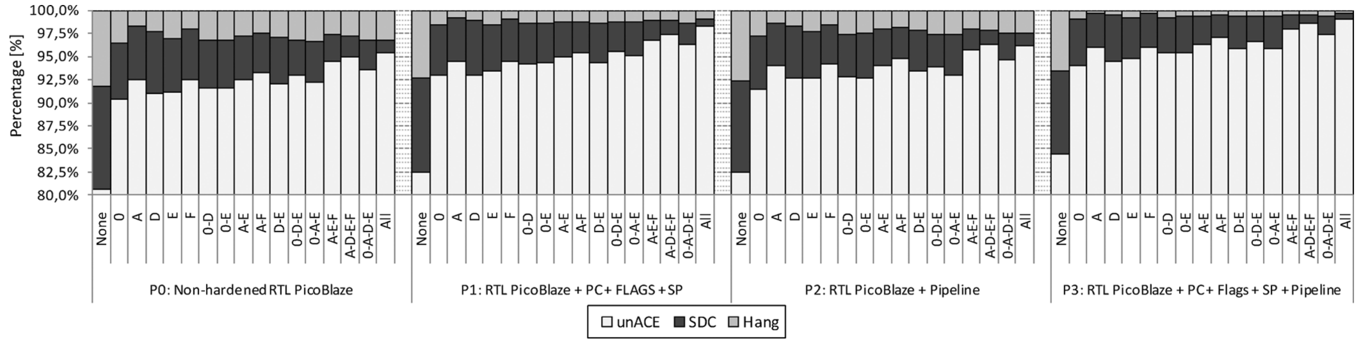

Fig. 6. Fault classification percentages for each selectively-hardened software version and selectively-hardened hardware approach.

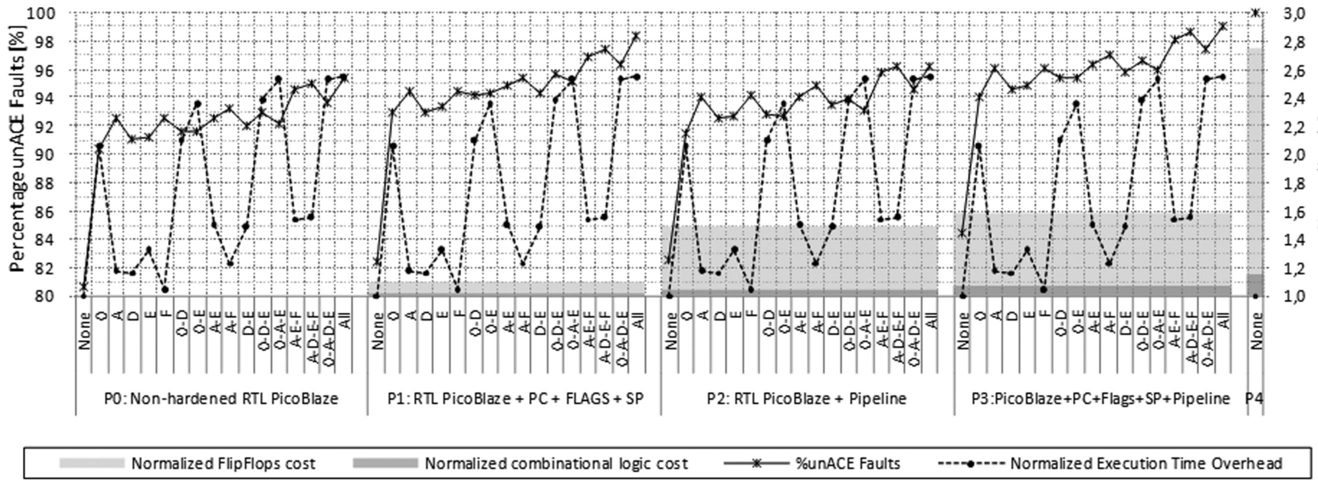

Fig. 7. Reliability (%unACE faults) and Normalized costs (hardware cost and execution time overhead) per every hardware/software version.

Fig. 7 shows the percentage of *unACE* faults obtained by the different hardware/software versions; it also depicts, in a secondary axis, the hardening costs (hardware costs and execution time overhead) of each version. These costs are normalized with a baseline built with the non-hardened *RTL PicoBlaze* (*P0*) running the non-hardened program. This figure permits to see at a glance, how reliability and costs are affected by every studied hardware/software approach. The hardware costs were provided by the *Synopsys Synplify Pro* synthesis tool, using the *Actel RT-ProASIC3* family (flash-based FPGA) as target technology [33]. The hardware cost is expressed in terms of combinational logic area, flip-flops number, and RAM blocks. Nonetheless, since the *Hardener* was able to fit all the hardened software versions in the *PicoBlaze* memory space, the number of RAM blocks remained constant for all the different versions and it is not represented in the figure.

One can observe that hardware cost increases considerably when the registers in the pipeline are hardened (*P2* and *P3*), whereas the reliability only improves slightly in these cases, or even decreases if compared with cheaper approaches (*P2*+*SWIFT-R*). In case of *P4*, high hardware costs may result unsuitable for many designs, even though its reliability is 100%. For comparison purposes, it must be noted that this solution has an approximate triple area overhead for the hardware and no execution time overhead for the software.

Finally, exploration of the resulting trade-offs among performance, code size, reliability and hardware cost allows the designer to decide which hardware/software configuration best meets the requirements of each specific application. For instance, in this case study it might result as a suitable configuration the prototype with the *P1* hardware and *SWIFT-R* applied to the register subset $\{A–D–E–F\}$ on the software side, because it offers high level of reliability (97.43% of *unACE* faults) with acceptable costs and overheads. In some other applications where the excessive performance degradation is produced as a result of the application of software-based techniques, it might be preferable to apply another more lightweight technique for the software, and increase protection and cost on the hardware side.

IV. Conclusions and Future Work

This paper presents a novel approach to design fault tolerant embedded systems. The co-design methodology jointly with the developed hardening infrastructure allow an easy exploration of the design space between hardware-only and software-only mitigation techniques. The resulting hybrid strategies are tuned to best fit the dependability requirements and the design constraints at the same time. The advantages of our proposal have been illustrated by means of a case study.

As future work, the *Software Hardening Environment* will be extended to support 32-bit soft-core microprocessors, given the fact that it is based on a *Microprocessor Generic Architecture*. Moreover, in order to complete the evaluation of our approach, it is planned to perform radiation ground test experiments using a hardened version of the system implemented as an *ASIC*.

REFERENCES

[1] R. Baumann, "Radiation-induced soft errors in advanced semiconductor technologies," *IEEE Trans. Device Mater. Rel.*, vol. 5, no. 3, pp. 305–316, Sep.. 2005.

[2] P. Shivakumar, M. Kistler, S. Keckler, D. Burger, and L. Alvisi, "Modeling the effect of technology trends on the soft error rate of combinational logic," in *Proc. Int. Conf. Dependable Systems And Networks*, 2002, pp. 389–398.

[3] T. Austin, "DIVA: A reliable substrate for deep submicron microarchitecture design," in *Proc. 32nd Annu. Int. Symp. Microarchitecture (MICRO-32)*, Haifa, Israel, 1999, Nov. 16-18, 1999, pp. 196–207.

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

[5] T. Vijaykumar, I. Pomeranz, and K. Cheng, "Transient-fault recovery using simultaneous multithreading," in *Proc. 29th Annu. Int. Symp. Computer Architecture*, Anchorage, AK, May 25-29, 2002, pp. 87–98.

[6] S. Mukherjee, M. Kontz, and S. Reinhardt, "Detailed design and evaluation of redundant multithreading alternatives," in *Proc. 29th Int. Symp. Computer Architecture*, May 25-29, 2002, pp. 99–110.

[7] M. Gomaa, C. Scarbrough, T. Vjaykumar, and I. Pomeranz, "Transient-fault recovery for chip multiprocessors," *IEEE Micro*, vol. 23, no. 6, pp. 76–83, Nov.–Dec. 2003.

[8] P. Samudrala, J. Ramos, and S. Katkoori, "Selective triple modular redundancy (STMR) based single-event upset (SEU) tolerant synthesis for FPGAs," *IEEE Trans. Nucl. Sci.*, vol. 51, no. 5, pp. 2957–2969, Oct. 2004.

[9] B. Nicolescu, Y. Savaria, and R. Velazco, "Software detection mechanisms providing full coverage against single bit-flip faults," *IEEE Trans. Nucl. Sci.*, vol. 51, no. 6, pp. 3510–3518, Dec. 2004.

[10] N. Oh, S. Mitra, and E. J. McCluskey, "(EDI)-I-4: Error detection by diverse data and duplicated instructions," *IEEE Trans. Comput.*, vol. 51, no. 2, pp. 180–199, 2002.

[11] M. Rebaudengo, M. S. Reorda, and M. Violante, "A new software-based technique for low-cost fault-tolerant application," in *Proc. Annu. Reliability and Maintainability Symp.*, 2003, pp. 25–28.

[12] M. Rebaudengo, M. Reorda, and M. Violante, "A new approach to software-implemented fault tolerance," *J. Electron. Testing-Theory Appl.*, vol. 20, no. 4, pp. 433–437, Aug. 2004.

[13] N. Oh, P. P. Shirvani, and E. J. McCluskey, "Error detection by duplicated instructions in super-scalar processors," *IEEE Trans. Rel.*, vol. 51, no. 1, pp. 73–75, Mar. 2002.

[14] N. Oh, P. Shirvani, and E. J. McCluskey, "Control-flow checking by software signatures," *IEEE Trans. Rel.*, vol. 51, no. 1, pp. 111–122, Mar. 2002.

[15] G. A. Reis, J. Chang, N. Vachharajani, R. Rangan, and D. I. August, "SWIFT: Software implemented fault tolerance," in *Proc. Int. Symp. Code Generation and Optimization*, 2005, pp. 243–254.

[16] G. A. Reis, J. Chang, and D. I. August, "Automatic instruction-level software-only recovery," *IEEE Micro*, vol. 27, no. 1, pp. 36–47, 2007.

[17] J. Azambuja, F. Sousa, L. Rosa, and F. Kastensmidt, "The limitations of software signature and basic block sizing in soft error fault coverage," in *Proc. 11th Latin American Test Workshop*, 2010, pp. 1–8.

[18] M. Pignol, "COTS-based applications in space avionics," in *Proc. 13th Design, Automation and Test in Europe Conf.*, Dresden, Germany, Mar. 2010, p. 1213.

[19] G. Reis, J. Chang, N. Vachharajani, S. Mukherjee, R. Rangan, and D. August, "Design and evaluation of hybrid fault-detection systems," in *Proc. 32nd Int. Symp. Computer Architecture*, Madison, WI, Jun. 04-08, 2005, pp. 148–159.

[20] P. Bernardi, L. Bolzani, M. Rebaudengo, M. Reorda, F. Vargas, and M. Violante, "A new hybrid fault detection technique for systems-on-a-chip," *IEEE Trans. Comput.*, vol. 55, no. 2, pp. 185–198, Feb. 2006.

[21] G. DeMicheli and R. Gupta, "Hardware/software co-design," *Proc. IEEE*, vol. 85, no. 3, pp. 349–365, Mar. 1997.

[22] F. Restrepo-Calle, A. Martínez-Álvarez, S. Cuenca-Asensi, F. Palomo, and M. Aguirre, "Hardening development environment for embedded systems," in *Proc. 2nd HiPEAC Workshop on Design for Reliability, in conjunction 5th Int. Conf. High Performance and Embedded Architectures and Compilers*, Pisa, Italy, Jan. 25-27, 2010.

[23] J. Napoles, H. Guzman, M. Aguirre, J. Tombs, F. Munoz, V. Baena, A. Torralba, and L. Franquelo, "Radiation environment emulation for VLSI designs A low cost platform based on xilinx FPGAs," in *Proc. IEEE Int. Symp. Industrial Electronics*, 2007.

[24] F. L. Kastensmidt, L. Carro, and R. Reis*, Fault-Tolerance Techniques for SRAM-Based FPGAs*, ser. Frontiers in Electronic Testing. New York: Springer, 2006, vol. 32, p. 183, xV, ISBN: 978-0-387-31068-8.

[25] L. Sterpone and M. Violante, "A new reliability-oriented place and route algorithm for SRAM-based FPGAs," *IEEE Trans. Comput.*, vol. 55, no. 6, pp. 732–744, Jun. 2006.

[26] K. Chapman, PicoBlaze KCPSM3. 8-bit Micro Controller for Spartan-3, Virtex-II and Virtex-II Pro, Xilinx Ltd., Oct. 2003.

[27] MISRA-C:2004 Guidelines for the Use of the C Language in Critical Systems, Motor Industry Software Reliability Assoc., 2004.

[28] S. Reinhardt and S. Mukherjee, "Transient fault detection via simultaneous multithreading," in *Proc. 27th Int. Symp. Computer Architecture*, Vancouver, BC, Canada, Jun. 12-14, 2000, pp. 25–36.

[29] H. Guzman-Miranda, M. Aguirre, and J. Tombs, "Noninvasive fault classification, robustness and recovery time measurement in microprocessor–type architectures subjected to radiation-induced errors," *IEEE Trans. Instrum. Meas.*, vol. 58, no. 5, pp. 1514–1524, May 2009.

[30] M. A. Aguirre, J. N. Tombs, F. Munoz, V. Baena, H. Guzman, J. Napoles, A. Fernandez-Leon, F. Tortosa-Lopez, and D. Merodio, "Selective protection analysis using a SEU emulator: Testing protocol and case study over the LEON2 processor," *IEEE Trans. Nucl. Sci.*, vol. 54, no. 4, pp. 951–956, Aug. 2007.

[31] M. Rebaudengo, M. S. Reorda, M. Violante, and M. Torchiano, "A source-to-source compiler for generating dependable software," in *Proc. 1st IEEE Int. Workshop Source Code Analysis and Manipulation*, 2001, pp. 33–42.

[32] S. Mukherjee, C. Weaver, J. Emer, S. Reinhardt, and T. Austin, "A systematic methodology to compute the architectural vulnerability factors for a high-performance microprocessor," in *Proc. 36th Int. Symp. Microarchitecture*, San Diego, CA, Dec. 3-5, 2003, pp. 29–40.

[33] Actel, ProASIC3L Low Power Flash FPGAs, Feb. 2009, Actel Corp., Data Sheet Rev. 9.