View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by Repositorio Institucional de la Universidad de Alicante

CDROM Proceedings

ISIE 2010

2010 IEEE International
Symposium on Industrial
Electronics

Palace Hotel Bari
Bari, Italy
04 - 07 July, 2010

Sponsored by

The Institute of Electrical and Electronics Engineers (IEEE)
IEEE Industrial Electronics Society (IES)

Co-sponsored by

IEEE Control Systems Society (CSS)
Society of Instrument and Control Engineers (SICE-Japan)
Politecnico di Bari, Italy

© 2010 IEEE. Personal use of this material is permitted. However, permission to reprint/republish this material for advertising or promotional purposes or for creating new collective works for resale or redistribution to servers or lists, or to reuse any copyrighted component of this work in other works must be obtained from the IEEE.

IEEE Catalog Number: CFP10ISI-CDR
ISBN: 978-1-4244-6391-6

Application-Driven Co-design of Fault-Tolerant Industrial Systems

F. Restrepo-Calle*, A. Martínez-Álvarez*, H. Guzmán-Miranda†, F. R. Palomo† and S. Cuenca-Asensi*
*Computer Technology Department, University of Alicante, Carretera San Vicente del Raspeig s/n, 03690 Alicante, Spain
†Department of Electrical Engineering, University of Sevilla, Camino de los Descubrimientos, 41092 Sevilla, Spain

***Abstract*—This paper presents a novel methodology for the *HW/SW* co-design of fault tolerant embedded systems that pursues the mitigation of radiation-induced upset events (which are a class of Single Event Effects - *SEEs*) on critical industrial applications. The proposal combines the flexibility and low cost of Software Implemented Hardware Fault Tolerance (*SIHFT*) techniques with the high reliability of selective hardware replication. The co-design flow is supported by a hardening platform that comprises an automatic software hardening environment and a hardware tool able to emulate Single Event Upsets (*SEUs*). As a case study, we selected a soft-micro (*PicoBlaze*) widely used in FPGA-based industrial systems, and a fault tolerant version of the *matrix multiplication* algorithm was developed. Using the proposed methodology, the design was guided by the requirements of the application, leading us to explore several trade-offs among reliability, performance and cost.**

I. Introduction

Currently, it is a fact that electronic components are more sensitive to Single or Multiple Event Effects induced by radiation due to its progressive miniaturization [1] [2]. These effects can cause catastrophic consequences in critical industrial applications whose operation take place under harsh environments where there are present ionizing radiation particles. Although these radiation-induced upset events are commonly found in the space environment, they are also present in a lower measure in the atmospheric environment [3] and even at ground level [4]. Therefore, fundamental information on minimum environmental withstand conditions (including *SEEs*) for electronic components has been established by several technical committees in each industrial field. These documents define detailed qualification requirements that electronic components must meet for its use. Some examples of them are: for aerospace applications, ESA PSS-01-609 (The Radiation Design Handbook) [5]; for avionic systems, IEC/TS 62396 (Process Management for Avionics - Atmospheric radiation effects) [6]; for military systems, MIL-HSBK-817 (System Development Radiation Hardness Assurance) [7]; etc.

Reliability problems have been mitigated usually using redundant hardware. Classic designs are based on the full triplication of the internal hierarchical modules [8], despite being very costly approaches. Some others, more recent techniques published in the literature [9] propose the selective hardening of the design, which means that a system needs to detect which parts of the circuit are more vulnerable or have larger probability of provoking a catastrophic behavior of the design itself. However, as mentioned above, electronic components have become more sensitive to transient faults. Therefore, during recent years several proposals based on redundant software have been developed, providing fault detection and recovery capabilities to applications [10] [11] [12]. These works are especially motivated by the need for low cost solutions (using Commercial Off The Shelf - *COTS* hardware), ensuring an acceptable level of reliability.

Since most hardening strategies (based on *HW* or *SW* redundancy) are designed to be applied in the handling of a wide set of applications without look after each constraint of the application, these lead to high costs, increase the development time and rise the performance and code overheads.

In this context, we propose the application-driven *HW/SW* co-design to achieve a customized fault tolerant version of the system that met the requirements of the application (fault coverage level, costs, execution time, memory size, ...).

The co-design flow is supported by a hardening platform, which is aimed to develop and evaluate fault tolerant embedded systems. This platform allows an easy design space exploration, taking advantage of the best of both worlds: the low cost of the software techniques and the high reliability of hardware redundancy. It is made up of a software hardening tool and a hardware *SEU* emulation tool. The first one is based on a generic and extensible architecture that allows handling multiple microprocessor targets and performs automatic source code transformations at low instruction level (assembler). The second one, based on *FPGAs*, permits to assess several reliability metrics of the overall system and identify the *SEU*-critical areas that must be hardened. By emulating SEUs at hardware level, it is possible to obtain more accurate results than by using simulation techniques. As the hardware emulation considers faults in hidden registers such as those ones in pipeline.

The *PicoBlaze* soft-micro was selected as test vehicle to work with. By the software side, a basic application of *matrix multiplication* was chosen. Thanks to their simplicity it was possible to fully understand the problem and to be prepared to use our tools subsequently to more powerful 32-bits architectures.

Next section presents the application-driven co-design flow. Section III describes the hardening platform. Section IV includes a case study applying the presented methodology. Section V presents the experiments and their results. Finally, Section VI concludes the paper and suggests directions for future research.

978-1-4244-6391-6/10/$26.00 ©2010 IEEE

II. Application-driven Co-design of Fault Tolerant Systems

The first step of our design methodology is the derivation of a set of system requirements from the point of view of the embedded application. Such requirements can be directly a system constraint or may target any kind of fault tolerance metric associated to a specific application. System constraints generally are related to silicon area, performance, power consumption and costs; whereas, fault tolerance metrics are concerned with fault coverage (*FC*), detection fault rate, recovery time, overheads, etc. These requirements feed into the generation of a test bench to guide the co-design of the system where constraints and fault tolerance metrics motivate design decisions. The incremental adoption of *SIHFT* techniques can then determine a set of suitable implementations of the software side of the system. Following, fault injection campaigns are performed to evaluate *FC* and identify critical system regions. The solution is not necessarily a single point in the design space but may result in a range of trade-offs.

The proposed co-design flow can be summarized as follows:

1) The specific requirements of the application (constraints and fault tolerance metrics) are fully defined.
2) Several *SIHFT* techniques are applied incrementally to obtain n candidate implementations of the software.
3) Each candidate implementation is evaluated to estimate its overhead comparing with the original program, in terms of code and execution time.
4) All solutions that met the maximum overheads specified are selected to run on the original microprocessor.
5) Using our *SEU* emulation tool, an overall fault injection campaign is performed to estimate the *FC* provided for each candidate running on the microprocessor system.
6) For the selected ones, an exhaustive fault injection campaign is carried out in order to identify the *SEU*-critical flip-flops that are not protected by the *SIHFT* techniques.
7) Hardware redundancy is applied to those identified *SEU*-critical flip-flops using different criteria depending on the reliability requirements of the application.

The result is a set of *HW/SW* configurations that achieve an optimized fault tolerant version of the system.

III. Hardening Platform

The platform that supports the proposed methodology comprises two suites of tools that follow the different nature of the two main steps in the co-design process, a software hardening development environment and a hardware tool for the evaluation of the robustness of the whole system.

A. Hardening Development Environment

We propose the use of a *generic architecture* to implement the hardening tasks. This architecture is useful to provide a uniform hardening core compatible with usual microprocessors by means of automatic code transformations. Therefore, to provide all the needed tools to implement and evaluate *SIHFT* techniques, we propose the scheme showed in Fig. 1.

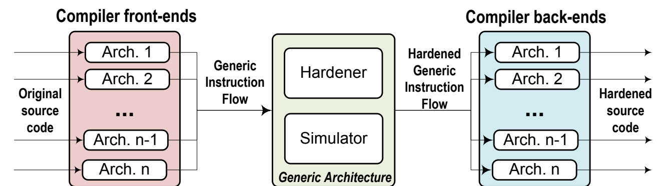

Fig. 1. Hardening Development Environment

The *compiler front-ends* take the original source code from a supported architecture, perform lexical, syntactical and semantic analyses, and finally generate a *Generic Instruction Flow* (*GenInsFlow*) as output. This flow represents a high level abstraction of a program that allows a platform independent implementation of the hardening routines (in the *hardener*). After the hardening process, the *hardener* produces a *hardened GenInsFlow*, which is taken by the selected *compiler back-end* to generate the *hardened source code* for the selected specific architecture. The *hardening environment* includes several functionalities that are common to the state-of-the-art *SIHFT* techniques: insertion of code, compile-time transformations, control flow analysis, management of architecture's resources, etc.

As better results are reported when low level instruction redundancy is applied, the *hardening environment* is conceived to perform code transformations at low level (assembler).

Using this scheme, the environment is prepared to take a code written for a supported architecture, perform its compilation and generic hardening, and finally, generate the output of the hardened source code targeting the same original architecture or to a different one by means of the back-ends.

Moreover, it is worth to mention that the software tools which comprise the hardening development environment are multiplatform, and have been successfully tested in *Debian GNU/Linux* (Kernel 2.6.30) and *Windows XP SP3/Vista*.

1) Generic Architecture: We took into account three topics to develop the generic architecture: generic instructions, memory management and control flow graph.

a) Generic Instruction: The generic architecture is defined by means of generic instructions. Each generic instruction is composed of the following fields.

Address. Memory address where the instruction has been assembled by the compiler front-end.

Mnemonic. Original mnemonic of the instruction.

Generic Operator List. Generic operators present at the instruction. Each operator is a member of the list and each one has three fields, they are: operator type, addressing mode and real name. The *Operator Type* defines the kind of operator, such as: Register, Literal, Address or Flag. The *Addressing Mode* can be: Absolute, Register Indirect, Immediate Literal, among others. Finally, the *Real Name* is the operator's original name.

Affected Generic Flag List. Generic flags affected by the execution of the instruction. Each generic flag is composed of a type and a name. The *Generic Flag Type* can be: Zero,

Carry, Interrupt Enable, etc. The *Real Name* is the original flag name of the target architecture.

Instruction Type. This is used to classify the instructions. It is very important because the hardening process depends on this type. For example, it is different to handle an arithmetic instruction and a control flow instruction during hardening. Some of the supported types are: interrupt, directive, control flow, arithmetic, logic, storage, input/output and shift/rotate.

Tool Message. This is a log the environment tools use to register events.

b) Memory Management: A common task for the *SIHFT* techniques is the insertion of instructions into the original code during compilation time. Therefore, it is necessary to supply the following memory management means within the hardener tool: identification of the memory map, extraction of the code sections and memory map update. The following three possibilities were developed to update the memory map: dilation, displacement and reallocation.

Dilation. When one or more instructions are inserted into a memory section, it grows and the affected instructions addresses should be reassigned.

Displacement. If some instructions are inserted into a previous memory section, it is possible having an overlapping with the following section. Then this section must be completely moved, updating all its instructions addresses.

Reallocation. If there is a memory overflow caused by previous instructions insertions, then it is needed to perform a complete reallocation of all memory sections. During this process, free memory space among memory sections is fully used. This situation may happen because of the reduced memory size in embedded systems.

c) Control Flow Graph: The generic architecture allows identifying the control flow graph from a given *GenInsFlow*. This graph is the key for most *SIHFT* techniques.

The control flow graph is represented by a directed graph. In order to build it, first we must identify the program's *basic blocks*. A *basic block* is a group of instructions that are executed sequentially, without any jump instruction nor function call, excepting possibly the last instruction. Also, a *basic block* does not contain instructions being the destination of a call or jump instruction, excepting the first instruction. Each *basic block* represents a node in the graph. The control flow changes are represented in the graph as links among the nodes. Fig. 2 shows an example of a control flow graph.

Additionally, as it was proposed by Reis et al. [13], only the *store* instructions ultimately send data out of the logical domain of redundant execution. Then it is necessary to perform special verification before the execution of these instructions.

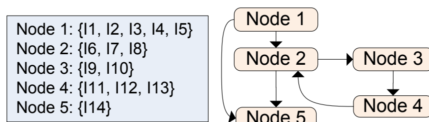

Fig. 2. Control Flow Graph

Therefore, in this paper we propose that the nodes (*basic blocks*) of the control flow graph should be subdivided into subnodes after each *store* instruction (Fig. 3).

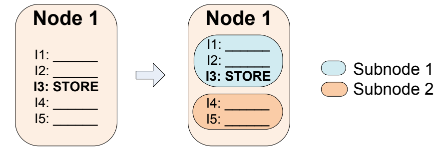

Fig. 3. Subnodes

2) Generic Hardening Core:

a) Hardener: This tool receives the *GenInsFlow* from the compiler front-ends. Then, according to the options selected by the user, it applies the hardening routines. Finally, a new hardened *GenInsFlow* is produced, which is re-targeted to one of the supported architectures by one of the back-ends. The most important hardening options are:

`--method`. Selects the *SIHFT* technique to be applied.

`--mcpu`. Target microprocessor to generate the output.

`--replicationRegisterLevel`. Defines the register redundancy level, such as: 0 - minimum redundancy level, for example in the instruction `ADD s0, s1` only the register `s0` is copied; 1 - every register has a copy, for example in the instruction showed above, registers `s0` and `s1` will be copied.

`--replicationTimes`. Defines the number of copies of each redundant instruction (0 - none, 1 - duplicate, 2 - triplicate).

`--voter`. Used to define voter and recovery routines.

`--NOlookAheadAvailableRegs`. Disable the advanced registers search. This is an optimization that consists of finding available registers for replication purposes looking forward than the current node along the control flow graph.

b) Instruction Set Simulator — ISS: This tool simulates the *GenInsFlow*. It presents information about the state of the resources of the architecture during and after the simulation process.

Likewise, the *ISS* allows verifying if the functionality of the hardened programs matches the original non-hardened programs functionality. This is possible by means of the `check-hardening` option that use information stored in the source code through a compiler *pragma* to know which the expected results are.

After the simulation process, the *ISS* presents a brief summary to inform the code and execution time overheads of the applied hardening technique. Also, it performs a characterization of the simulated programs, informing the percentage of executed instructions by its type: arithmetic, logical, control flow, etc.

B. SEU Emulation Tool — FT-Unshades

The *FT-Unshades* system, described in [14], is a FPGA-based platform for the study of digital circuit reliability against radiation-induced soft errors. *SEU* affecting the circuit are emulated by inducing bit-flips in the circuit under study, by means of partial reconfiguration.

The system is composed of a FPGA emulation board and a suite of software tools for design preparation, testing of the emulated design, and analysis of the test results. The main software of the suite is the *FT-Unshades Test aNalysis Tools* (*TNT*) program, which manages the communications with the board, the partial reconfiguration and the test campaign execution.

In the original version of *FT-Unshades*, two instances of the circuit or module under test (*MUT*) are instantiated in the implemented design: *Target* and *Gold*. Faults are injected over the *Target* instance, whereas the *Gold* instance remains unchanged for comparison purposes.

The system has been extended for the study of microprocessor architectures. An exhaustive description of this extension can be found in [15]. Instead of two instances of the *MUT* (*Gold* and *Target*), the implemented design has just one instance of the *MUT* (*Target*), and the *Golden* instance is substituted by a *Smart Table* (see Fig. 4). This is needed because the typical cycle-by-cycle comparison would classify as *output error* the effect of faults that could be corrected if the *Target* microprocessor was given more processing time. The exact additional time, measured in clock cycles, which the affected microprocessor needs to output the correct value is called *recovery time*.

The *Smart Table* is an automaton which implements the *relaxed time restrictions* needed for the fault injection testing of microprocessors that implement *SIHFT* techniques.

The *Smart Table* can be configured in emulation-time (this is, after synthesis, implementation and *FPGA* programming). First, the *Smart Table* must be configured with the outputs of a *Golden Run* of the *Target* microprocessor. This means a whole emulation of the circuit processing workload is done, but *without* injecting any bit-flips. When being configured during a *Golden Run*, the *Smart Table* not only memorizes the sequence of the correct outputs, but also the time (in clock cycles) where the outputs change.

After filling the *Smart Table* with the *[expectedOutput, expectedCycle]* duplets, the most important parameter the user must configure is the *critical recovery time* (T_{crit}), which is the maximum recovery time allowed for the microprocessor. This means that if the microprocessor outputs the correct value in *expectedCycle* + T_{crit} *cycles*, the fault is classified as producing *no damage*. But if *expectedCycle* + T_{crit} + *1 clock cycles* have passed and the microprocessor has not output any value yet, the *Smart Table* classifies the fault as producing *timeout*. Note that, since the emulation stops at this moment, *timeout* can mean either that the program execution has frozen, or that the damage is so bad that the hardening technique cannot recover the correct values after T_{crit} + *1*. Since we want to relax the time restrictions of the test, but we want the output data sequence to be correct, if the microprocessor outputs a wrong value (*Output != expectedOutput*) before *expectedCycle* + T_{crit} + *1*, the fault is classified as producing *output damage*.

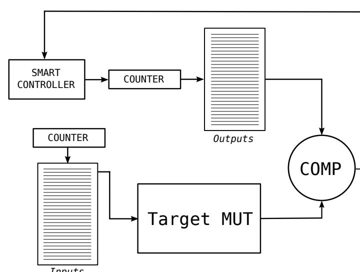

Fig. 4. *FT-Unshades* implementation approach using *Smart Table*

IV. Case Study

As a case study, it is presented the co-design of a hardened version of the *matrix multiplication* algorithm in the *PicoBlaze* soft-micro [16].

A. *PicoBlaze*

This is an 8 bit soft-micro widely used in *FPGA*-based embedded systems. It supports the following main features: 16 byte-wide general-purpose data registers, 1K 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.

In order to transform *PicoBlaze* code with our software hardening environment, a compiler front-end and back-end were developed.

The *PicoBlaze* front-end for takes the original *KCPSM3* source code, performs lexical, syntactical and semantic analyses, and finally generates a *GenInsFlow* as output. This is a multiplatform compiler front-end that provides a very accurate error localization, compared with any other *PicoBlaze* compilers (including the official *KCPSM3* compiler).

After the hardening process (performed by the *hardener*), it is produced a *hardened GenInsFlow*, which is taken by the developed compiler back-end for *PicoBlaze*, transforming the flow back to the *KCPSM3* syntax.

B. *SIHFT Fault Tolerance Techniques*

Since *matrix multiplication* algorithm is highly comprised of arithmetic and logic instructions, several *SIHFT* techniques (detection and recover) aimed to protect those instruction types were implemented. These techniques are based on the well known Triple Modular Redundancy (*TMR*) approach.

First implemented strategy (*TMR1*) can be summarized as follows.

1) Identification of nodes (*basic blocks*) and subnodes in the program.
2) Build the control flow graph of the program.
3) Triplication of the operation.
4) Insertion of majority voters and recovery procedures for protected registers at the following points: just before the last instruction of each node/subnode and also, just before any instruction being the destination of a jump or function call.
5) During the hardening process, majority voters and recovery procedures are dynamically injected when there

are not enough available registers to replicate. By means of this, registers copies will be released to continue with the hardening process.

Second implemented strategy (*TMR2*) consists in detect and correct faults in the program data by computing the values twice and recomputing a third time if a discrepancy between the first two values occurs.

Fig. 5 shows an example of the hardening of a simple program (*KCPSM3* syntax) using *TMR1* and *TMR2* applied to arithmetic instructions.

Fig. 5. Hardened program using *TMR1* and *TMR2* applied to arithmetic instructions

For the remaining of the paper, we will call *arithTMR1* to the technique used when the first hardening strategy (*TMR1*) is applied to arithmetic instructions. When applied to logic instructions, it will be named *logicTMR1*. So, if it is applied to both arithmetic and logic instructions, it will be named *arithTMR1+logicTMR1*. This naming scheme also applies when using the second hardening strategy (*TMR2*).

V. Experiments and Results

Firstly, the requirements of the application must be defined in terms of performance constraints and reliability metrics. For instance, when it is considered an application whose time response is critical, the maximum execution time overhead has to be defined as appropriate, whereas other applications could be more restrictive in other aspects as reliability.

The original version of the *matrix multiplication* (*mmult*) algorithm was written using the *KCPSM3* syntax. The original program can be automatically transformed, using the hardening environment, by applying six *SIHFT* techniques (*arithTMR1*, *arithTMR2*, *logicTMR*, *logicTMR2*, *arithTMR1+logicTMR1*, *arithTMR2+logicTMR2*). The functionality of each hardened version is checked using the *ISS*, assuring that it is equivalent to the original non-hardened program functionality. After this, using the *ISS* as well, the code and execution time overheads can be obtained for each hardened version comparing them with the original program. Table I presents results for *mmult*.

TABLE I
Code and execution time overheads for *mmult*

Program: *mmult*	**Code overhead**	**Execution time overhead**
arithTMR1	×2.30	×2.92
arithTMR2	×2.01	×2.60
logicTMR1	×2.30	×1.33
logicTMR2	×2.01	×1.28
arithTMR1+logicTMR1	×3.61	×3.26
arithTMR2+logicTMR2	×3.03	×2.88

The overheads analyses can motivate important design decisions, e.g. as the *arithTMR1+logicTMR1* approach causes unsuitable overheads for a particular application of the *mmult* algorithm, this hardened version could be discarded for further analyses.

In order to continue with the process, it is necessary to implement the hardware of the system to be tested within the *FT-Unshades SEU* emulation tool. After this, a fault injection campaign is prepared and executed for the whole system. In our case study, the hardware was implemented with the official *Xilinx PicoBlaze* netlist and it was designed a test campaign with following features:

- The emulated design is *RTL*-equivalent to the final hardware.
- Random injection of *SEU* in a target bit chosen from all sixteen 8-bit registers (from *s0* to *sF*) and flags (zero and carry).
- Total number of injected *SEUs*: 13.000 (one per run).
- The clock cycle for the *SEU* injection was randomly selected from all the workload duration.
- T_{crit} was defined as 10 clock cycles, giving to the microprocessor 10 more clock cycles than the usual time usual to recover the system to a fault-free state.

Table II shows the *FC* results obtained for each version of the system, including the non-hardened one. Results have been classified according to the effects caused in the program due to injected fault as: *correct results*, when despite the fault, the expected results are obtained; *incorrect results*, either when wrong results are obtained due to the fault or if the fault causes an infinite loop in the execution of the program.

FC results jointly with overheads results must be taken into account in the next decisions about system design. As it can be seen, the combination of *arith* and *logic* versions does not produce a better coverage than each approach by its self. It could be possible that the system constraints and fault tolerance minimums were already fulfilled by some *SIHFT* version; otherwise hardware redundancy must be applied. For example, if execution time is critical, the *logicTMR2* approach can be selected for further fault coverage improvements. On the contrary, if code size is a restriction, *arithTMR2* and *logicTMR2* could be selected. In our case, *logicTMR1* was selected because it offers the highest *FC* and acceptable overheads.

Afterwards, it must to be prepared an exhaustive fault injection campaign for the chosen approach(es). This time the

TABLE II
FC results for each *SIHFT* technique

mmult	**Correct results**	**Incorrect results**
original	73.14%	26.86%
arithTMR1	90.15%	9.85%
arithTMR2	91.09%	8.91%
logicTMR1	92.27%	7.73%
logicTMR2	89.64%	10.36%
arithTMR1+logicTMR1	90.68%	9.32%
arithTMR2+logicTMR2	91.10%	8.90%

test campaign was performed emulating 100 *SEUs* for each bit from the target (sixteen 8-bit registers and flags), one *SEU* per run in a randomly selected clock cycle. Notice that this experiment is a really exhaustive test campaign because all possible memory cells are tested in 100 different times. The majority of the injection campaigns related in the literature usually performs *SEU* injections in the order of 10^2 from all the architecture registers, whereas we injected in the order of 10^3 for the defined target. Focusing in the *logicTMR1* approach, the Fig. 6 presents the percentage of failure behavior caused by the target bits (flip-flops) when a *SEU* had affected them.

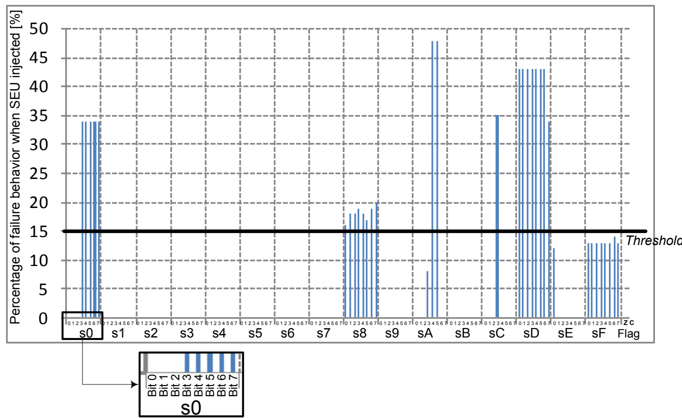

Fig. 6. *SEU*-critical flip-flops on target for *logicTMR1*

These results are useful to identify the *SEU*-critical bits in the system, that are not protected by the *SIHFT* technique and should be hardened using hardware redundancy. It is worth mention that hardware redundancy has a minimum associated cost of $\times 3,2$ (triplication of resources plus voter). Therefore, if there is a cost area constraint and it is not possible to apply hardware redundancy to all of the *SEU*-critical identified flip-flops, they must to be prioritized according to which one has a higher probability to provoke an undesired behavior of the system. For instance, the selected flip-flops will be those ones whose failure percentage is above a predefined *threshold* (in this case 15%).

Finally, after applying the hardware redundancy to the selected flip-flops, it is expected to end up with a *HW/SW* system configuration (or more) that satisfies the reliability requirements of the studied application.

VI. Conclusions and Future Work

In this paper we have presented a new methodology for the co-design of fault tolerant industrial systems affected by radiation-induced upset events. It is based on a guided co-design flow that able the designer to apply hardware and software hardening methods obtaining the best trade-off taking into account the requirements of a specific application (cost, performance, overheads, *FC* level, ...). This methodology is supported by a hardening platform that allows a quick space design exploration to reach the requirements. The platform comprises a *SIHFT* development environment and a hardware *SEU* emulation tool. In order to confirm the feasibility of our proposal, a case study has been considered to the hardening of *matrix multiplication* algorithm in the *PicoBlaze* soft-micro.

The *SIHFT* development environment will be extended for being used with more advanced microprocessors to take advantage of the generic architecture which the generic hardening core is based on.

Acknowledgment

This work makes part of *RENASER* project (`ESP2007-65914-C03-03`) funded by the 2007 Research National Plan of the Ministry of Science and Education in which context this work has been possible. The work presented here has been carried out thanks to the support of the research projects 'Aceleración de algoritmos industriales y de seguridad en entornos críticos mediante hardware: Aplicación al sector calzado' (`GV/2009/098`) (Generalitat Valenciana) and 'Aceleración hardware de algoritmos industriales para el sector calzado' (`GRE08-P11`) (University of Alicante).

References

[1] R. Baumann, "Radiation-induced soft errors in advanced semiconductor technologies," *IEEE Trans. on Device and Materials Reliability*, vol. 5, no. 3, pp. 305–316, Sept 2005.

[2] P. Shivakumar, et al, "Modeling the effect of technology trends on the soft error rate of combinational logic," in *Int. Conf. on Dependable Systems and Networks*, 2002, pp. 389–398.

[3] R. Edwards, et al., "Technical standard for atmospheric radiation single event effects (SEE) on avionics electronics," in *IEEE Radiation Effects Data Workshop (REDW)*, 2004, Proc. Paper, pp. 1–5.

[4] R. Baumann, "Soft errors in commercial semiconductor technology: Overview and scaling trends," *IEEE 2002 Reliability Physics Tutorial Notes, Reliability Fundamentals*, p. 121, April 2002.

[5] ESA, "The Radiation Design Handbook ESA PSS-01-609," European Space Agency, Tech. Rep., 1993.

[6] IEC, "IEC/TS 62396-1," International Electrotechnical Commission, Tech. Rep., March 2006.

[7] DoD, "MIL-HDBK-817, Military Handbook System Develop Radiation Hardness Assurance," Department of Defense. USA, Tech. Rep., 1994.

[8] H. H. Amer and R. M. Daoud, "Fault-secure multidetector fire protection system for trains," *Instrumentation and Measurement, IEEE Transactions on*, vol. 56, no. 3, pp. 770–777, June 2007.

[9] P. Samudrala, J. Ramos, and S. Katkoori, "Selective triple modular redundancy (stmr) based single-event upset (seu) tolerant synthesis for fpgas," *Nuclear Science, IEEE Transactions on*, vol. 51, no. 5, Oct. 2004.

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

[11] M. Rebaudengo, M. S. Reorda, and M. Violante, "A new software-based technique for low-cost Fault-Tolerant application," *Annual Reliability and Maintainability Symposium, 2003 Proceedings*, pp. 25–28, 2003.

[12] 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.

[13] G. A. Reis, J. Chang, N. Vachharajani, R. Rangan, and D. I. August, "SWIFT: software implemented fault tolerance," *CGO 2005: Int Symposium on Code Generation and Optimization*, pp. 243–254, 2005.

[14] 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 *IEEE International Symposium on Industrial Electronics, ISIE 2007*, 2007.

[15] 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 Transactions on Instrumentation and Measurement*, vol. 58, no. 5, May 2009.

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