2010 International Conference on Field Programmable Logic and Applications

Rapid Prototyping of Radiation-Tolerant Embedded Systems on FPGA

F. Restrepo-Calle*, A. Martínez-Álvarez*, F.R. Palomo†, H. Guzmán-Miranda†, M.A. Aguirre† 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*—Technological advances of Field Programmable Gate Array (*FPGA*) are making that this technology becomes the most preferred platform for the rapid prototyping of highly integrated digital systems. In addition, protection of processor-based systems to mitigate the harmful effects of radiation-induced upset events is gaining importance while technology shrinks. In this context, the main contribution of this work is a novel rapid prototyping approach for the co-design of dependable embedded systems using *FPGA*. This is supported by a hardening platform that allows combining software-only fault-tolerance techniques with hardware-only approaches, representing several trade-offs among design constraints, reliability and cost. As case study, several radiation-tolerant embedded systems have been developed based on a technology-independent version of the *Picoblaze* processor.**

I. Introduction

In recent years progressive miniaturization of electronic components has led important advances in microprocessors. However, this fact has both advantages and disadvantages. The most meaningful advantage has been the dramatically increase of microprocessors performance. Although, while technology shrinks, voltage source level and noise margins are reduced, causing that electronic devices become less reliable and microprocessors more susceptible to *transient faults* induced by radiation. These intermittent faults do not provoke a permanent damage, but may result in incorrect program execution by altering signal transfers or stored values [1].

Applying redundant hardware has been the usual way to mitigate reliability problems. This strategy has been applied from low level structures (*ECC*, *parity bits*) to more complex components like functional units [2], co-processors [3], etc. In the same way, several approaches have exploited the multiplicity of hardware blocks available on multi-threaded/multi-core architectures to implement redundancy [4], [5]. More recent techniques published in the literature [6] propose selective hardening of the design, which means that the system is protected only in the more vulnerable parts. The hardware approaches provide a very effective solution for *transient faults*. However, these techniques are unfeasible in many cases due to the increased costs involved.

This work was funded by the Ministry of Science and Education in Spain with the *RENASER* project (ESP2007-65914-C03-03) and the *Generalitat Valenciana* in Spain with the research project 'Aceleración de algoritmos industriales y de seguridad en entornos críticos mediante hardware' (GV/2009/098).

In addition to this, during recent years several proposals based on redundant software have been developed, providing both detection and error correction capabilities to applications. These works are especially motivated by the need for low cost solutions, ensuring an acceptable level of reliability [7], [8], [9]. While software-based approaches are cheaper than hardware-based ones, they cannot achieve the same performance or reliability, since they have to execute additional instructions.

Despite the wide number of different hardware-only and software-only methods, in many cases the optimal solution is an intermediate point and it is needed to design the system combining software and hardware aspects (hardware/software co-design). Some recent works have shown the viability of this hybrid strategy [10], [11].

In this context, it is important to count with suitable tools which allow designers to easily explore the design space in order to find the best trade-offs that satisfy the requirements of a system in terms of performance, reliability, and hardware cost. In addition, every time designers must face shorter Time-to-Market, tighter design constraints and higher reliability goals. Unlike *VLSI* design, the design iterations of *FPGA* based implementations need not be frozen much earlier in the design cycle. Therefore, there is a growing use of *FPGAs* to prototype *ASICs* as part of an *ASIC* verification methodology.

In this paper, *FPGAs* are used as development and verification platform in order to produce *HW/SW* embedded systems that best meet the design and reliability constraints. Mitigation techniques are applied to a high abstraction level so the final deployment platform will be an *ASIC* or an *FPGA*; in the second case, some additional mechanisms, like configuration scrubbing, have to be taken into account to protect the configuration memory.

Our prototyping approach is supported by a hardening platform that is made up of two suites of tools: a software development environment aimed to implement, automatically apply and evaluate software-only fault tolerant techniques [12]; and a *FPGA*-based fault emulation tool called *FTUnshades* [13] that permits to assess several reliability metrics.

As case study for validating our approach, we have studied the hardening of several embedded applications based on the *PicoBlaze* soft-microprocessor [14]. By the software side, the *SWIFT-R* technique [15] has been implemented and automatically applied to several benchmarks;

978-0-7695-4179-2/10 $26.00 © 2010 IEEE
DOI 10.1109/FPL.2010.71

whereas on the hardware side, using a *RTL* implementation (*technology-independent*) of the *Picoblaze* soft-micro, five different versions have been developed varying the hardware redundancy level. Different trade-offs among code overhead, performance, fault coverage and cost have been represented.

II. Fault Model and Terminology

In this paper we will focus on the well known *Single Event Upset* fault model. In this *SEU* fault model, only one bit-flip of a storage cell occurs throughout the execution of the program. This effect is caused by the ionization provoked by an incident charged particle. Despite its simplicity, the *SEU* fault model is widely used in the fault tolerance community to model real faults because it closely matches the real fault behavior [9].

In order to evaluate the reliability of the system, we classify the injected faults according to their effect on the expected program behavior as it was proposed by Reis et al. [15]. If the fault provokes that the program completes its execution, but does not produce the expected output, this fault is called *Silent Data Corruption — SDC*. If the program completes its execution and produces the expected output, the fault is categorized as *unnecessary for Architecturally Correct Execution — unACE*. Finally, if the fault causes the program to abnormally finish its execution or to remain forever into an infinite loop, we categorize this fault as *Hang*. Note that *SDC* and *Hang* are both undesirable effects (categorized together as *ACE* faults).

In addition, it is worth noting that software-based techniques necessary introduce redundancy, and this causes two important facts to take into account. Firstly, these techniques increase the execution time of the programs; therefore, probability of fault occurrence is higher than for the non-hardened program whose execution time is smaller. Secondly, redundancy increases the number of bits present on the system, increasing the number of bits that are susceptible to fault. Therefore, the fault coverage offered by a specific hardening strategy is directly related with the percentage of *unACE* faults and the execution time overhead.

III. Platform for the Rapid Prototyping of Dependable Embedded Systems

A. Software Development Environment

The proposed infrastructure establishes a complete software-hardening development environment allowing the design and implementation of software-based techniques to be automatically applied into programs. The infrastructure is made up of a multi-target compiler supporting several common hardening routines (*Hardener*), an *Instruction Set Simulator* (*Simulator*), and several compiler front-ends and back-ends. Fig. 1 shows the general scheme.

Among various advantages of our proposal we highlight that it is based on a *Microprocessor Generic Architecture* providing a uniform hardening core that allows to design

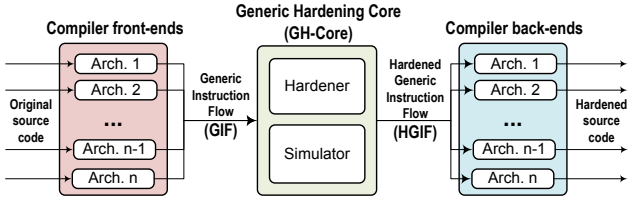

Figure 1. Software-hardening development environment

and implement different hardening techniques in a platform independent way. The automatic generation of hardened code is guided by instruction-level code transformation rules.

A 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 that allows a platform independent implementation of the hardening routines (within the *Generic Hardening Core*). After the hardening process, the *Hardener* produces a *Hardened-GIF* (*HGIF*) which is then re-targeted to a supported processor.

Using this scheme, the hardening infrastructure is also flexible in the sense that is possible to process a code written for a supported architecture and generate protected code targeting to the same original architecture or to different one by means of the several back-ends.

Considering hardening purposes, as it was suggested by Reis et al. [16], we propose to classify in a special way those instructions whose function imply to cross the borders of the *Sphere of Replication* (*SoR*) [17]. The *SoR* is the logic domain of redundant execution. Therefore, when an instruction causes that some data enter inside the *SoR* (e.g. reading an input port, loading a value into a register or reading a value from memory), we will classify it as *inSoR*; and consequently when an instruction provokes data goes out from the *SoR* (e.g. writing on an output port, storing a value into the memory), we will classify it as *outSoR*.

Note that the boundaries of the *SoR*, and consequently the coverage of the protection, could change according to the implemented technique. For instance, in *EDDI* [8] the memory subsystem is inside of the *SoR*, so the instructions responsible to perform read/write operations over the memory do not cause that any data cross the *SoR* borders. In the same way, if the memory subsystem is considered outside of the *SoR*, those instructions reading from memory or writing into memory are causing some data to cross through the sphere frontiers and must be handled in a special way.

The identification of the *Control Flow Graph* (*CFG*) and the insertion of instructions into the source code during compilation time are the keys for software-based techniques [7], [8], [15]. In this way, our tool also provides the necessary functionalities to suitably perform *Memory Management* and analyses to the program's *CFG*.

Regarding the memory management, the *Hardener* is able to: identify the memory map, extract memory sections, and perform modifications over them. In addition, similarly to other approaches, it is assumed that the code being hardened does not exploit dynamic memory allocation, i.e. every data structures are defined statically at compilation time. This is not a significant limitation for developers of embedded applications, which sometimes are forced to code standards that already avoid dynamic memory usage [18]. The following three possibilities are supported to keep updated the memory map: *dilation*, *displacement* and *reallocation*.

Dilation. When one or more instructions are inserted during compilation time into a memory section, this section grows and some of the instructions addresses inside this memory section should be reassigned.

Displacement. If dilation provokes that two or more memory sections share some addresses, which is an illegal situation, then the section must be completely moved and all its instructions addresses updated.

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

On the other hand, the *Microprocessor Generic Architecture* allows the identification of the *CFG* from a given *GIF*. 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. Also, a *basic block* does not contain instructions being the destination of a call or jump instruction, excepting the first instruction. The control flow changes are represented in the graph as links among nodes.

In addition, when an instruction sends data outside of the *SoR*, it may provoke an unrecoverable error if that data is corrupted. In this way, it would be desirable to perform a verification of the data's correctness before it leaves the sphere. Therefore, in this paper we propose that the *nodes* (*basic blocks*) of the *CFG* should be subdivided into *sub-nodes* after each instruction classified as *outSoR*.

The *Generic Hardening Core* has two main components: the *Hardener* and the *ISS*. The *Hardener* is comprised of an Application Programming Interface (*API*) of hardening routines typical from software-based techniques algorithms. Developing new hardening algorithms is a task significantly easier by using this *API*.

The available options in the *Hardener* able the designer to decide which technique will be applied, which is the replication level to be used in redundant instructions (i.e. duplication, triplication), which is the preferred recovery procedure to use (i.e. among different implementations of majority voters, in case of recovery techniques, or routines to be applied when a fault is detected). In this way, the *Hardener* offers a complete control to the user for configuring the protection strategy.

The *ISS* assists the designer in the implementation of new software-based techniques. It allows to perform different analyses on the *GIF* and *HGIF* to check the correctness of the hardening process, and also offers useful information to aid the designer in the co-design process: time and space overheads, and fault-coverage estimations. In order to evaluate the reliability provided by the applied techniques, the *ISS* is able to inject *SEUs* during the simulation by means of *bit-flips* into the registers file bits. However, these reliability results are preliminary estimations, because they do not consider micro-architectural details of the target. This information is obtained by means of a hardware *SEU*-emulation tool included in our hardening platform.

B. SEU-Emulation Tool: FTUnshades

The *FTUnshades* system, described in [19], is a FPGA-based platform for the study of digital circuit reliability against radiation-induced *transient faults*. *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 *FTUnshades Test aNalysis Tool* (*TNT*), which manages the communications with the board, the partial reconfiguration and the campaign execution.

In the original version of *FTUnshades*, 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 [13]. 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*. 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 software-based 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*.

IV. Case Study

In order to validate our proposal, we have designed and evaluated several prototypes of radiation-tolerant embedded systems based on the *PicoBlaze* microprocessor.

Picoblaze is an 8 bit soft-micro widely used in *FPGA*-based embedded systems. In this work, a technology-independent version of *PicoBlaze* has been developed (*RTL PicoBlaze*) that allows to validate the proposal for both, *ASIC* and *FPGA*. This version is cycle accurate and *RTL* equivalent to the original *PicoBlaze-3* version of the micro. 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.

The benchmark software suite used in the experiments is made up of the following test programs: bubble sort (*bub*), scalar division (*div*), Fibonacci (*fib*), greatest common divisor (*gcd*), matrix addition (*madd*), matrix multiplication (*mmult*), scalar multiplication (*mult*) and exponentiation (*pow*).

Every test program was hardened applying the *SWIFT-R* technique. This is an overall method aimed to recover faults from the data section, mainly related with the register file of the micro. Our implementation of this method can be explained as follows.

1) Identification of *nodes* and *sub-nodes* of the program and building its *CFG*.
2) Data triplication the first time that any data comes into the *SoR* (i.e. after *inSoR* instructions). In this case, only the register file is considered as included into the *SoR*, whereas the memory subsystem is not because it is assumed that memory already has its own protection mechanism [15]. Therefore, for every instruction classified as *inSoR* (read input ports, read from memory, load a value into a register), two additional copies will be created, just by means of copying the register values without repeating memory or ports accesses.
3) Triplication of instructions that perform any data operation (e.g. arithmetic, logic, shift/rotate instructions).
4) Check the consistency of the data involved on the following instructions (by inserting majority voters and recovery procedures before execute them): *outSoR* instructions, e.g. store into a memory position or write into an output port; and those instructions located just before a conditional branch. This verification is necessary because these instructions affect the flags, so if a register value is corrupted, resultant flag might be erroneous too, provoking an erroneous branch somewhere in the *CFG*.
5) Redundant registers only can be released in the following situations: if the register is not used anymore in the program, or if next time the register is used is overwritten. Note this latter condition imply a detailed analysis to the *CFG* to avoid consistency loss.

Fig. 2 shows the code and execution time overheads offered by the *ISS* after applying *SWIFT-R* to each test program. These results are normalized with a baseline built with the non-hardened version. In this case, the geometric mean (calculated across all benchmarks) of the normalized code overheads is $\times 3.05$, and the execution time overhead is $\times 2.70$.

On the hardware side, the fault tolerant co-design strategy was complemented by incrementally hardening the microprocessor resources that still were unprotected after the applied software protection for the register file. The microprocessor was hardened by manually applying *Triple Modular Redundancy* (*TMR*) to different subsets of micro-architectural registers. Five versions were developed:

- *P0:* non-hardened *RTL Picoblaze*.

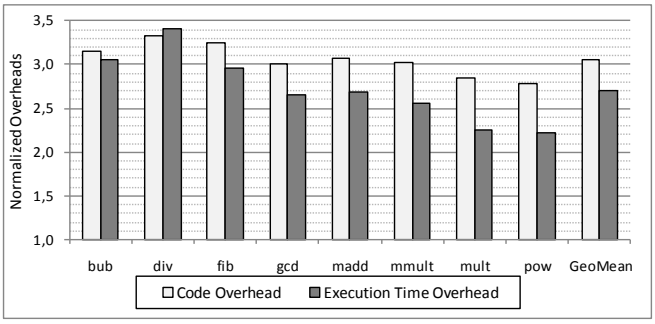

Figure 2. Normalized code and execution time overheads for *SWIFT-R*

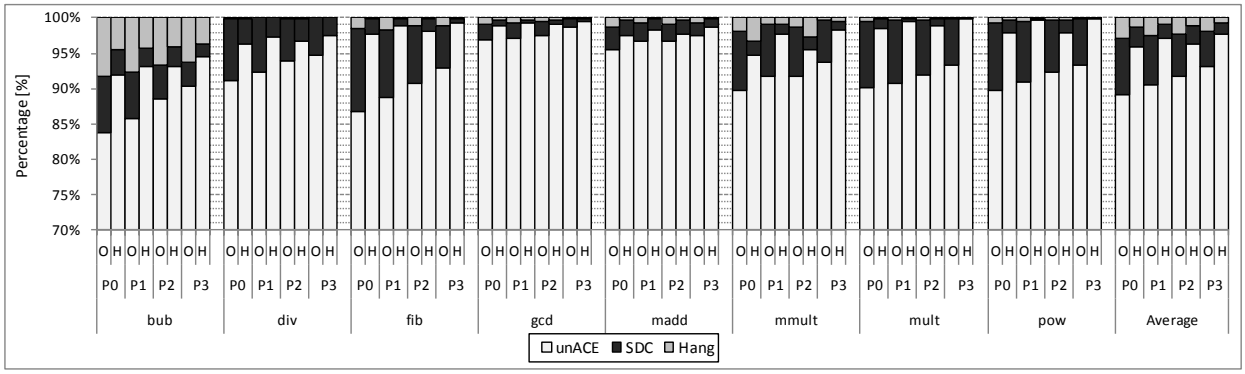

Figure 3. Fault classification percentages for every test program (non-hardened (*O*) and SWIFT-R (*H*)) running on each processor version (*P0* to *P3*)

- *P1:* microprocessor with hardware redundancy for Program Counter (*PC*), Flags and Stack Pointer (*SP*).
- *P2:* microprocessor with hardware redundancy for all registers in the pipeline.
- *P3:* microprocessor with hardware redundancy for *PC*, Flags, *SP*, and Pipeline.
- *P4:* full protected, i.e. microprocessor with hardware redundancy for Register file, *PC*, Flags, *SP*, and Pipeline.

The following experimental setup has been configured to assess the reliability offered by the designed prototypes. For each test program in the benchmark, non-hardened (*O*) and hardened (*H*), and for each approach of the microprocessor (*P0* to *P4*), a fault injection campaign has been executed in the *FTUnshades*. Every fault injection campaign makes selective attacks on the microprocessor register sets: register file (128 bits - 65.0% of the target size), *PC* (10 bits - 5.1% of the target size), Flags (2 bits - 1.0% of the target size), *SP* (5 bits - 2.5% of the target size), pipeline (52 bits - 26.4% of the target size). Moreover, for each one of these register sets, 5000 *SEUs* (one per execution) have been emulated in a randomly selected clock cycle from all the workload duration. A critical time (T_{crit}) of 1023 clock cycles has been chosen for these experiments. Finally, results were classified as described in Section II as *unACE*, *SDC*, and *Hang*. Fig 3 presents the fault classification percentages obtained for each prototype (note that these results have been obtained calculating the weighted average of the results from the selective attacks to the internal microprocessor register sets, assuming the same fault probability for all bits on target).

Note that the *SWIFT-R* technique offers a considerable reliability increment, even in the non-hardened hardware (in average, up to 95.88% *unACE* faults), which is higher than the reliability of every hardware-hardened approach using the non-hardened program. Results for the *P4* micro approach are not presented in Fig. 3 because 100% of the injected faults were classified as *unACE*, as expected. Furthermore, notice that combining *SWIFT-R* with hardware protection in only few critical registers, such as *PC*, Flags, and *SP* (*P1* version), reliability increases markedly (in average, up to 97.18% *unACE* faults).

Although the reliability is higher combining software-hardened programs with hardware-redundant approaches (for instance, up to 97.77% *unACE* faults for the *P3* micro), the costs are also higher and this is an important restriction that must be considered, jointly with design constraints and reliability requirements. In order to obtain an estimation of the area cost, every version of the micro was synthetized using *Xilinx XST v10.1*, and the results expressed in terms of: Flip/Flops and Latches, primitives (mux, luts, etc.), and RAM (distributed and block ram). Fig 4, on the one hand, shows the hardware cost of each approach normalized with a baseline built with the non-hardened *RTL Picoblaze* (*P0*); on the other hand, it also depicts, in a secondary axis, the percentage of *unACE* faults obtained for the non-hardened and *SWIFT-R* test programs (in average). This figure permits to see at a glance, how reliability and costs are affected by every studied hardware approach.

It is worth noting that hardware cost increases considerably when registers in pipeline are hardened (*P2* and *P3*),

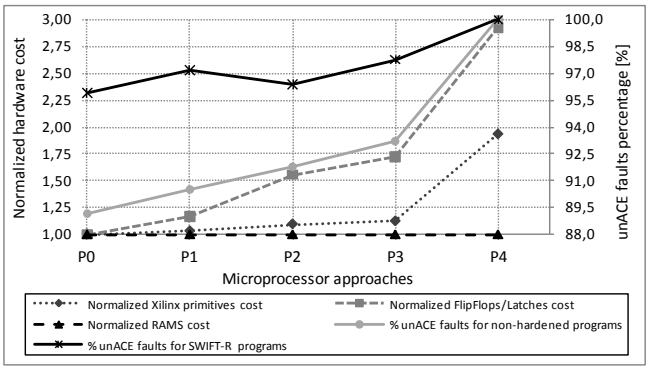

Figure 4. Normalized hardware cost and percentage of *unACE* faults per microprocessor approach

whereas 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 could result unsuitable for many applications, although its reliability is 100%.

Finally, obtained results (overheads, reliability and hardware costs) able the designer to decide which *HW/SW* configuration best meet the requirements of each specific application. For instance, for this case study might result as a suitable configuration those prototypes with the microprocessor with hardened *PC*, Flags, and *SP* (*P1*) and *SWIFT-R* programs, because these prototypes offer a high level of reliability (97.18% of *unACE* faults) with acceptable costs (low hardware costs, $\times 2.70$ execution time overhead, and $\times 3.05$ source code overhead). In some other applications, for example, if *SWIFT-R* performance degradation exceeds required, it might be preferable applying another lightweight technique for the software, and incrementing protection and cost on the hardware side.

V. Conclusions and Future Work

This paper presents a rapid prototyping approach for the design of radiation-tolerant embedded systems using *FPGA*. This approach is supported by a flexible hardening platform, which facilitates the representation of several trade-offs among design constraints, reliability, performance, cost, etc. This rapid prototyping strategy allows designers to easily explore the design space between hardware-only and software-only fault-tolerance techniques. The advantages of the resultant hybrid *HW/SW* implementations are illustrated by means of a case study. In this context, several robust embedded systems based on a *RTL* implementation of the *PicoBlaze* soft-micro have been developed. As a result, this new strategy suggests the implementation of automatic co–hardening tasks within the presented platform and opens up interesting new boundaries in design space exploration. As future work, the hardening development environment will be extended to support 32-bit soft-core microprocessors, such as *MicroBlaze* and *LEON3*.

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] T. Austin, "DIVA: A reliable substrate for deep submicron microarchitecture design," in *32nd Annual Int Symp on Microarchitecture*, 1999, pp. 196–207, israel, Nov 16-18, 1999.

[3] A. Mahmood and E. McCluskey, "Concurrent error-detection using watchdog processors - a survey," *IEEE Transactions on Computers*, vol. 37, no. 2, pp. 160–174, FEB 1988.

[4] S. Mukherjee, M. Kontz, and S. Reinhardt, "Detailed design and evaluation of Redundant Multithreading alternatives," in *29th Annual Int Symp on Comp Arch*, 2002, pp. 99–110.

[5] 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.

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

[7] N. Oh, P. Shirvani, and E. J. McCluskey, "Control-flow checking by software signatures," *IEEE Transactions on Reliability*, vol. 51, no. 1, 2002.

[8] 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.

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

[10] G. Reis, J. Chang, N. Vachharajani, S. Mukherjee, R. Rangan, and D. August, "Design and evaluation of hybrid fault-detection systems," in *32nd Int Symp on Comp Arch*, 2005, pp. 148–159.

[11] 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 Tran on Comp*, vol. 55, no. 2, pp. 185–198, Feb 2006.

[12] F. Restrepo-Calle, A. Martínez-Álvarez, S. Cuenca-Asensi, F. Palomo, and M. Aguirre, "Hardening development environment for embedded systems," 2010, 2nd HiPEAC Workshop on Design for Reliability DFR10. Italy, Jan 25-27, 2010.

[13] 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.

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

[15] 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.

[16] G. Reis, J. Chang, N. Vachharajani, R. Rangan, and D. August, "SWIFT: software implemented fault tolerance," *CGO Int Symp on Code Gen and Opt*, pp. 243–254, 2005.

[17] S. Reinhardt and S. Mukherjee, "Transient fault detection via simultaneous multithreading," in *27th Int Symp on Comp Arch*, 2000, Proceedings Paper, pp. 25–36.

[18] MISRA, *MISRA-C:2004 Guidelines for the use of the C language in critical systems*. Motor Industry Software Reliability Association, 2004.

[19] 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.