Effectiveness and Limitations of Various Software Techniques for "Soft Error" Detection: A Comparative Study

B. Nicolescu[1], *R. Velazco*[1], *M. Sonza Reorda*[2]
[1]*TIMA Laboratory,46, Av. Félix Viallet, 38031, Grenoble, France.*
e-mail: *Bogdan.Nicolescu@imag.fr*
[2]*Politecnico di Torino, Dip. Automatica e Informatica, corso Duca degli Abruzzi 24 I-10129 Torino, Italy*

Abstract

This paper deals with different software based strategies allowing the on-line detection of bit flip errors arising in microprocessor-based digital architectures as the consequence of the interaction with radiation. Fault injection experiments put in evidence the detection capabilities and the limitations of each of the studied techniques.

1. Background

Digital circuits operating under radiation are subject to different kinds of permanent or transient effects [1]. The former are the result of particles trapped at the silicon/oxide interfaces and appear only after long exposure to radiation, while the latter may be provoked by the impact of a single charged particle (*single event effects, SEE*) in sensitive circuit zones. Depending on the impact location, two kinds of SEEs are distinguished. SEUs *(Single Event Upsets)*, are responsible of transient changes, generally called "upsets" or "bit flips", in bits of information stored within an integrated circuit, while SELs (Single Event Latchups) result from the triggering of parasitic thyristors (present in CMOS technologies) and provoke short circuits, capable to damage the component by thermal effect if circuit is not powered-off at time. Despite the potential permanent consequences of Single Event Latchups, the possibility of their detection by a permanent current consumption control, make them less critical than SEUs. Indeed, the consequences of SEUs depend on both the nature of the perturbed information and the occurrence instant, ranging from erroneous results to critic system problems. For "programmable" circuits (circuits able to execute an instruction set) like DSP processors, co-processors, or micro-controllers, the sensitivity to SEUs strongly correlates with the amount of internal memory (registers, memory bits, flip-flops, etc.). It is important to note that due to the constant improvements achieved in the integrated circuit design and manufacturing technology, particles such as neutrons present in the atmosphere (up to now innocuous), have enough energy to provoke bit flips in memory cells contents of present integrated circuits. Moreover, it is expected that future deep submicronic circuits will be subject to transient errors in combinational parts, as the result of the impact of a neutron. This constitutes a serious source of errors particularly for avionics (at 30 000 feet the flux of neutrons is 4 to 8 times higher than at ground level) but also for digital equipment operating at ground level.

Manufacturing technologies such as SOI (*Silicon on Insulator*) attenuate the effects of SEUs, but the high cost entailed by the use of such "little volume" processes make this solution not attractive. Moreover, the 100% immunity to SEUs cannot be guaranteed and the impact of transistor features reduction will certainly increase SEE sensitivity even for SOI circuits. A promising approach to mitigate the effects of SEU, the so-called design hardening, consist in transforming the design of a circuit by suitable techniques to allow manufacturing reliable circuits using commercially available CMOS processes. Proposed design hardening solutions go from the use of well-known fault tolerant techniques (e.g. error correction codes and hardware redundancy) to the development of libraries of hardened cells (e.g. memory cells [2-4], families of logic gates [5]). A wide range of solutions is available in the specialised literature, but up to now, none of them was considered as being capable to reach an acceptable trade-off between cost, performance and reliability.

Approaches based exclusively on software transformations should allow the implementation of dependable systems without incurring the high costs coming from designing custom hardware or using hardware redundancy Representative solutions can be found in [6-8]. The idea is to mitigate the effects of SEUs in main sensitive area (including memory areas where program and data are stored, processor's registers and internal memories,...) by their detection (and correction in some cases) by means of suitable mechanisms.

0-7695-1290-9/01 $10.00 © 2001 IEEE

In [9] we have presented a comprehensive set of rules allowing to transform a code in a new one, called *hardened code* in the following, having the same functionality but with error detection abilities. Targeted faults were bit flips in the memory area where program code and data are stored. Preliminary results issued from fault injection experiments as well as from radiation testing were gathered using a digital board. The main observation issued from the analysis of these results is that the number of undetected faults producing wrong program results is significantly reduced when faults affecting the *code* are considered. Indeed, 45.5% of faults injected in the original program induced wrong answers, while for the hardened code less than 10% of such faults were observed. When faults affecting the *data* are considered, they are *all* detected in the hardened program. Note that for the original program around 80% of faults injected on data areas led to wrong program results. Result issued from radiation testing with the same test set-up running a set of benchmark programs, were in good agreement with those derived from fault injection experiments.

From these experiments we concluded that only a few of injected faults (around 0.2 % in the average) escaped the software detection mechanisms. The effectiveness of the proposed method for upset fault detection is thus proved. Nevertheless, relying on software techniques to obtain dependability often means accepting an overhead in terms code size and a reduction in timing performance. Indeed, transforming the code according to the proposed rule set, to turn it capable of fault detection, leads to a code occupying a memory size more than 3 times bigger than the original one, whose execution time for 4 times longer. Even if for applications having weak memory and performance constraints such overhead can be considered acceptable, we aimed at investigating the code overhead and time performances resulting in the implementation of well known state of the art strategies.

The purpose of this paper is thus to compare the detection/correction capabilities of hardened code obtained by applying our automated transformation strategy, to the performances of well known error detecting and correcting software strategies implemented for the same program and exercised with the same hardware. In section II are briefly presented the considered strategies. The study the effects of bit flip on each of these implementations bit flips were injected in external memory and internal registers of an architecture built around a Transputer (a RISC processor with parallelism capabilities). The main features of the used hardware platform are summarized in section III. Experiments were performed on-line: the faults were injected in selected targets during a randomly selected clock period, concurrently with the execution of a benchmark program. Experimental results are analyzed and discussed in section IV. The observation of the program results allowed to get an insight about both the consequences of the fault and the efficiency of the detection/correction implemented mechanism. Finally, section V presents concluding remarks and future work.

2. Considered Software Error Detection Approaches

Three main different strategies for error detection by software means were compared. The first one consists in transforming the target program according to a comprehensive set of transformation rules devoted to include error detection capabilities. The other two are based on the addition and checking of cyclic redundancy codes (CRC) and Hamming codes.

2.1 Program Hardening By Software Transformation

The method is based on a set of transformation rules applied to a high level code in order to obtain a functionally equivalent program with error detection capabilities. These transformations, which were automatically performed, introduce data and code devoted to detect possible errors corrupting information stored in the memory area. Details about the transformation rules and preliminary results of their application to a set of benchmark can be found in [9] and [10]. Let us summarize here the main features:

- To detect faults affecting the data the main idea is to duplicate all the variables in the program and check for consistency of the two copies each time they are used again. For each read-write operation of a variable, we repeat the operation for the duplicated one and perform a consistency check.

- To detect faults affecting the code, we exploit two principles. The first one is to execute any operation (including the conditional statements) twice and then verify the coherency of the resulting execution flow. Second principle aims at detecting those faults modifying the code so that incorrect jumps are executed (for instance by affecting the operand of an existing jump, or by transforming certain instruction into a jump instruction) producing a faulty execution flow. To detect this type of errors, an extra instruction is added at the beginning of each basic block in the code. To each basic block is associated a flag, whose value is checked at the end of the block.

2.2. CRC Method

To detect errors in the memory containing the code a well known strategy consists in calculating a value function of all the bytes included in the considered area, and checking it prior to the program execution to detect possible errors.

A detailed description of how CRC codes are calculated is out of the scope of this paper. Let us suppose we know the first and the last addresses of the program in memory location. The idea is to granulize the program in independent blocks and to include at the beginning of each block a call to a CRC routine. We have explored two possibilities to implement such error detection mechanism. The first one consists in checking the whole memory area allocated for the code when the CRC routine is launched, and will be called *global CRC* in the following. The second one will be referred as *distributed CRC*, and limits the check only to the piece of memory assigned to each independent block. Table 1 illustrates such code transformation in the case of distributed CRC.

Table 1: Software transformations to include distributed CRC

Original version	Modified version
/* start block k */ ... /* end block k */ /* start block j */ ... /* end block j */	/* start block k */ if(!startCRC()) error(); ... /* end block k */ /* start block j */ if(!startCRC()) error(); ... /* end block j */

2.3 Using Hamming Codes

The main idea is here to associate an extra code information to every variable in the code. It is important to note that, for variables up to 64 bits, Hamming correctors can be stored in only one byte. The corrector byte must be updated every time a new value is assigned to the variable. Two parameters are needed for the decoding procedure: the variable's value and the variable's corrector byte. In the case of the corruption of one of these two parameters as a consequence of a bit flips, the decoding procedure will take one the following decisions:

If one bit is corrupted, then the decision is correction

If two bits are damaged, then the decision is detection, without correction possibility

If more than two bits are affected, the decision is an erroneous correction

Table 2 gives an example for a simple piece of code of the resulting program including Hamming codes.

Table 2: Including Hamming codes in a program

Original version	Modified version

...
a=5;
...
b=a+2;
...
	...
	a=5;
	a_code=code(5);
	...
	b=decode(a_code,a)+2;
	b_code=code(b);
	...

3. Evaluating the efficiency of software hardening techniques

To assess the efficiency of the studied approaches, we implemented them for a simple program, implementing a bubble sort of an integer vector. For the *software hardening* technique, the corresponding C programs was automatically obtained as the output of a translator implementing the set of transformation rules, while for the CRC and Hamming code techniques, transformations were carried out manually, directly working on the C programs.

3.1 Experimental set-up

An available hardware platform, developed at TIMA laboratory within the frame of an international satellite project[1], offered the easiest way to implement and exercise the different program versions under simulated bit flips. This board, called in the following *Transputer board*, is organized around a Transputer and was previously used to perform radiation testing while running software hardened programs. We briefly recall its main characteristics, while further details can be found in [11].

The Transputer board mainly includes:

a T225 transputer (a reduced instruction set microprocessor with parallel capabilities). The T225 is the main core of the board, being in charge of all the operations related with data transfer to/from the user and the implementation of test programs;

a 4 Kbyte PROM, containing the executable code of the programs related with the operation of the board (boot, result transfer, program loading);

a 32 Kbyte SRAM, used for the storage of T225 program workspaces, programs and data. The last 2 Kbytes are reserved to data transfer to/from the user;

an anti-latchup circuit, for the detection of abnormal power consumption situations (resulting from Single Event Latchups occurrence) and the activation of the corresponding recovering mechanisms;

[1] MPTB: Microelectronics and Photonics Testbed of Naval Research Laboratories, Washington DC

a watch-dog system, periodically refreshed by the T225, which has been included in order to avoid system crashes due to events arising on critical targets such as the T225 internal memory cells (registers or flip-flops) or the external SRAM memory areas associated to the program modules (process workspaces).

The board can easily support fault injection experiments: faults are randomly injected in the proper locations during the program execution. To be consistent with the characteristics of upset errors, which occur in actual applications evolving under radiation, we performed the injection of single faults on randomly selected bits belonging to the code and data area. The injection mechanism is implement by a dedicated T225 process, which runs in parallel with the tested program. The two programs (the injection program and the program under test) are loaded in the prototype board memory and launched simultaneously. The injection program waits for a random duration, then chooses a random address and a random bit in the memory area used by the program under test, and inverts its value. After each injection, the behavior of the program is supervised, the fault is classified, and the results are sent to the PC acting as a host system.

3.2 Experimental results

The experiments we performed are based on carrying out extensive fault injection sessions while in the Transputer runs the programs versions corresponding to the three studied methods. Faults were classified in the following categories according to their effects on the program execution:

Effect-less: the injected fault does not affect the program behaviour

Software detection: the fault is detected by the implemented detection mechanism

Hardware detection: the fault triggers some hardware mechanism allowing the detection of the fault (e.g., an illegal instruction exception, watchdog, etc.)

No answer: the program under test triggers some time-out condition, e.g., because it entered an endless loop

Wrong answer: the fault escapes pull trough detection mechanisms and the result is different from the expected one.

One of the advantages of the adopted fault injection strategy (faults are injected in a real prototype as the consequence of the execution of a suitable routine running in parallel with the target program execution) is the possibility to select the target of the injected bit flip. Tables 3 and 4 give the results obtained from bit flip injection in the memory area where programs were stored and in the data area respectively. For each of the studied programs 1000 faults were injected at different instants during program execution. To simulate the random occurrence of real upsets arising as the consequence of radiation, faults were triggered after completion of a *sleep* random period (feature available in Transputer programming model) in the dedicated fault injection process.

3.4 Discussion

A general characteristic of all three hardening techniques is the good detection capability of faults affecting the program code, especially for global CRC. Indeed, the program hardened with this technique was able to detect 99% of the injected bit flips. The software hardening and the CRC based programs achieve respectively 90% and 80% detection.

Table 3: Fault injection results affecting the code memory area

Method	# faults	# effect-less	# wrong answer	# hardware detection	# no answer	# software detection	# corrections
Software Hardening	1000	331	1	63	12	593	0
Global CRC	1000	0	1	5	2	992	0
Distribute CRC	1000	200	23	95	41	641	0
Hamming	1000	312	38	95	16	536	2
Original program	1000	108	498	231	163	0	0

A significant number of effect-less faults, ranging from 20% to 30% was identified. These cases correspond to faults injected after the use of the corrupted information, and have no effect because the program code is reloaded after each fault injection. The absence of effect-less faults for the non-distributed CRC is due to the fact that CRC code is checked and detected prior the execution of a program block. This also explains the low number of sequencing failures (leading to no answer situation provoking the hard reset of the board) for distributed-CRC technique. Finally, it must be noticed the very low success in error correction for bits affecting the code of the program protected by Hamming strategy. Investigation of possible causes of this failure is in progress.

Table 4: Fault injection results affecting the data memory area

Method	# faults	# effect-less	# wrong answer	# hardware detection	# no answer	# software detection	# corrections
Software Hardening	1000	258	0	0	0	742	0
Global CRC	1000	0	0	0	0	0	0
Distribute CRC	1000	0	0	0	0	0	0
Hamming	1000	307	7	0	0	364	322
Original program	1000	215	785	0	0	0	0

Concerning faults corrupting data, they are obviously not protected by CRC because their of dynamic change nature, on the other side, Hamming codes lead to an excellent detection and correction rate close to 99%. Nevertheless, only the half of the detected faults was corrected. As only single bit faults were injected in theory all the faults should be detected and corrected; in our case correction fails can be explained by assuming that those faults corrupting memory bytes where Hamming code intermediate values are stored.

Table 5: Consequences of bit flips injected in Transputer's registers

Method	Register	# faults	# effect-less	# wrong answer	# hardware detection	# no answer	# software detection	# corrections
Original program	Areg	1000	423	573	4	0	0	0
	Breg	1000	824	176	0	0	0	0
	Creg	1000	1000	0	0	0	0	0
	Wptr	1000	79	8	885	28	0	0
	Iptr	1000	120	208	740	34	0	0
Software Hardening	Areg	1000	354	4	5	0	637	0
	Breg	1000	695	0	0	0	305	0
	Creg	1000	1000	0	0	0	0	0
	Wptr	1000	54	0	644	169	133	0
	Iptr	1000	93	9	488	18	392	0
Hamming	Areg	1000	388	37	51	0	496	28
	Breg	1000	636	23	5	0	299	37
	Creg	1000	956	0	0	0	44	0
	Wptr	1000	0	0	645	12	343	0
	Iptr	1000	99	25	543	15	306	12

Faults were also injected in Transputer registers. Targets were the three pipelined registers (Areg, Breg and Creg), the instruction pointer register (Iptr) and the workspace pointer register (Wptr) used to indicate the address in external memory containing the context of the currently executed process. Table 5 summarizes the results of fault injection experiments affecting those internal Transputer sensitive areas.

Regarding faults corrupting the internal Transputer register, we did not implement the CRC technique, which is intended to detect errors in the code only.

The analysis of results given in Table 5 allows to draw some preliminary conclusions concerning faults affecting the processor.

For both hardened methods (Software and Hamming), the number of escaped errors is significant reduced, while more than 50 % of faults lead to erroneous output in the original program.

The relatively low detection for faults affecting the two control registers (Wptr and Iptr), is not surprising considering the critic function of these two registers. Related to first two usual register stack (Areg and Breg) we observe a good rate of detection (more than 90 %). Finally, practically all faults affecting the Creg register of the stack have no effect on the program behaviour.

3.4. Limitations

The major drawback of error detection by software means come from the resulting memory area overhead and the increase in execution time for the modified codes. For the studied simple program, a Bubble Sort of a 100 integer values list Table 6 gives the overhead factors.

Table 6: Program and time overhead factors for a simple program

Method	Time overhead	Memory overhead
Software Hardening	2.5	4
Global CRC	11.2	1.25
Distribute CRC	5.8	1.25
Hamming	12.7	3

As far as the execution time is concerned, the software hardening approach leads to the best performance. At the opposite, CRC based error detecting programs entail a very high time overhead (more than 10 times) but little memory overhead (around 25%). However, note that memory overhead which is in this case totally acceptable, will become quasi negligible for complex programs as the number of extra bytes is only dependent on the number of program blocks.

4. Conclusions and Future Work

This paper aimed at comparing the efficiency of different techniques for fault detection in digital architectures operating under the effect of radiation. Considered faults were bit flips corrupting the values stored in memory elements including external memories or internal processor's registers and memories. Such faults are considered nowadays critic because they may occur in space environment and even in the earth's atmosphere as the consequence of the interaction with neutrons.

A set of fault injection sessions was performed exploiting an existing digital architecture built around a Transputer. A simple program protected according to different error detection software technique was adopted. Thousands of bit flips were randomly injected in sensitive area during the execution of each of the studied programs. The analysis of the obtained results showed that none of the methods could alone be used to guarantee 100% error detection. Nevertheless, the software hardening approach appears as a good compromise between error detection and memory size and time overhead.

Future work includes performing the same kind of comparative study on more complex programs (modules of an industrial application) and two new hardware architectures one of them based on an Intel 8051 microcontroller (circuit widely used in space projects), the other on a SPARC microprocessor.

5. References

[1] B. Randell, "System Structure for Software Fault Tolerant," IEEE Trans. On Software Engineering, Vol. 1, No. 2, Jun. 1975, pp. 220-232

[2] R. Velazco, D. Bessot, R. Eccofet, S. Duzellier, Two CMOS memory cells suitable for the design of SEU tolerant VLSI circuits, IEEE Transactions on Nuclear Science, Vol. 6, n° 41, pp. 2229-2234, 1994.

[3] L. Rockett, "An SEU Hardened CMOS Data Latch Design", IEEE Trans. On Nuclear Science, Vol. 35, No.6, Dec. 1988

[4] M. N. Liu and S Witaker, "", IEEE Trans. On Nuclear Science, Vol. 39, No.6, pp. 1679 – 1684, Dec. 1992

[5] M. Nicolaidis, "Time Redundancy Based Soft-Error Tolerance to Rescue Nanometer Technologies", VTS'99: IEEE VLSI Test Symposium, 1999, pp. 86-94

[6] K. H. Huang, J. A. Abraham, "Algorithm-Based Fault Tolerance for Matrix Operations", IEEE Trans. on Computers, vol. 33, December 1984, pp. 518-528

[7] Z. Alkhalifa, V.S.S. Nair, N. Krishnamurthy, J.A. Abraham, "Design and Evaluation of System-level Checks for On-line Control Flow Error Detection," IEEE Trans. On Parallel and Distributed Systems, Vol. 10, No. 6, Jun. 1999, pp. 627-641

[8] S.S. Yau, F.-C. Chen, "An Approach to Concurrent Control Flow Checking," IEEE Trans. On Software Engineering, Vol. 6, No. 2, March 1980, pp. 126-137

[9] M. Rebaudengo, M. Sonza Reorda, M. Torchiano, M. Violante, "Soft-error Detection through Software Fault-Tolerance techniques", DFT'99: IEEE International Symposium on Defect and Fault Tolerance in VLSI Systems, Austin (USA), November 1999, pp. 210-218

[10] P. Cheynet, B. Nicolescu, R. Velazco, M. Rebaudengo, M. Sonza Reorda, M. Violante, Experimentally Evaluating an Automatic Approach for Generating Safety-Critical Software with Respect to Transient Errors, IEEE Transactions on Nuclear Science, à paraître janvier 2001

[11] R. Velazco, Ph. Cheynet, A. Tissot, J. Haussy, J. Lambert, R. Ecoffet, "Evidences of SEU tolerance for digital implementations of Artificial Neural Networks: one year MPTB flight results", session W, *RADECS'99*, Abbaye de Fontevraud, 13 - 17 septembre 1999.