System Safety through Automatic High-level Code Transformations: an Experimental Evaluation

M. Rebaudengo, M. Sonza Reorda, M.Violante
Politecnico di Torino
Dipartimento di Automatica e Informatica
Torino, Italy
http://www.cad.polito.it/

P. Cheynet, B. Nicolescu, R. Velazco
Institut National Polytechnique Grenoble,
TIMA Laboratory
Grenoble, France
http://tima.imag.fr

Abstract:

This paper deals with a software modification strategy allowing the on-line detection of transient errors. Being based on a set of rules for introducing redundancy in the high-level code, the method can be completely automated, and is particularly suited for low-cost safety-critical microprocessor-based applications. Experimental results from software and hardware fault injection campaigns are presented and discussed, demonstrating the effectiveness of the approach in terms of fault detection capabilities.

1. Introduction

The increasing popularity of low-cost safety-critical computer-based applications requires the availability of new methods for designing dependable systems. In particular, in the areas where computer-based dependable systems are currently being introduced, the cost (and hence the design and development time) is often a major concern, and the adoption of commercial hardware (e.g., based on Commercial Off-The-Shelf or COTS products) is a common practice. As a result, for this class of applications software fault tolerance is an extremely attractive solution, since it allows the implementation of dependable systems without incurring the high costs coming from designing custom hardware or using hardware redundancy. On the other hand, relying on software techniques for obtaining dependability often means accepting some overhead in terms of increased code size and reduced performance. However, in many applications, memory and performance constraints are relatively loose, and the idea of trading off reliability and speed is often easily acceptable. Finally, when building a dependable system, designers need simple and reliable mechanisms for assessing whether the whole system has the required dependability properties, and any solution able to provide by construction the desired fault detection rate is warmly welcome.

Several approaches have been proposed in the past to achieve fault tolerance (or just safety) only by modifying the software. The proposed methods can mainly be categorized in two groups: those proposing the replication of the program execution and the check of the results (e.g., Recovery Blocks [1] and N-Version Programming [2]) and those based on introducing some control code into the program (e.g., Algorithm Based Fault Tolerance (ABFT) [3], Code Flow Checking [4], Signature Monitoring [5], Error Detection and Correction code [6]). None of the mentioned approaches is at the same time *general* (in the sense that it can be used for any fault type and any application, no matter the algorithm it implements) and *automatic* (in the sense that it does not rely on the programmer skill for its effective implementation). Therefore, none of the above methods is a complete and suitable for the implementation of low-cost safety-critical microprocessor-based systems.

To face the gap between the available methods and the industry requirements, we propose an approach, which is based on introducing data and code redundancy according to a set of transformations performed on high-level code for detecting errors affecting both data and code. The main novelty of this strategy lies in the fact that it is based on a set of transformation rules, so their implementation on any high-level code can be completely automated. This frees the programmer from the burden of guaranteeing the application robustness against errors and drastically reduces the costs for its implementation.

The approach presented in this paper is intended to face the consequences of errors originating from transient faults, in particular those caused by charged particles hitting the circuit [7]. This kind of fault is increasingly likely to occur in any integrated device due to the continuous improvements in the VLSI technology, which reduces the size of the capacitance storing information and increases the operating frequency of circuits. The result is a significant increase in the chance that particles hitting the circuit can introduce misbehavior.

Being based on modifying only the high-level code, the method is completely independent on the underlying hardware, it does not require any hardware duplication or modification (apart from some overhead in the required memory), and it possibly complements other already

existing error detection mechanisms. The method extends the one described in [8], which follows a similar approach, but only addresses faults in the data.

To provide the reader with experimental evidence of the effectiveness of the method, we developed a prototypical tool implementing the transformation rules and applied it to some simple benchmark application programs. We then performed a set of fault injection experiments to quantitatively evaluate the fault detection capabilities of the modified code. The experiments have been performed by means of a software fault injection environment; further experiments are currently being performed, using a hardware fault injection environment allowing to irradiate the memory chips and therefore to obtain results very similar to those which can be observed (on a much longer time scale) in practice. For the purpose of the experiments we performed, we focused on a particular error type, called *upset* or *bit-flip*, which results in the modification of the content of a memory cell within a circuit. This perturbation is the result of the ionization provoked either by incident charged particles or by daughter particles created from the interaction of energetic particles (i.e., neutrons) and atoms present in the silicon substrate. However, the method we propose is able to detect a much wider set of transient errors, e.g., those affecting combinational blocks of logic, or affecting multiple memory bits at the same time. The experimental results we gathered show that the method is able to detect *any* error affecting the data, while the coverage is *over 99%* for faults affecting the code.

The paper is organized as follows. Section 2 outlines the transformation rules, and provides some examples of modified code. Section 3 describes the fault injection environment we exploited for gathering the experimental results, which are presented in Section 4. Section 5 draws some conclusions.

2. Transformation Rules

In this section we describe the basic ideas behind a set of transformation rules to be applied to the high-level code. These transformations introduce data and code redundancy, which allow the resulting program to detect possible errors affecting storage elements containing data or code. To preserve the redundancy introduced in the hardened program, compiler optimization flags should be disabled.

The transformation rules described in the following were first introduced in [9], where preliminary results obtained through fault injection experiments were presented. All the examples presented in the following are made on C programs, although rule principles are not limited to this programming language and can be easily extended to other languages.

2.1. Transformations for faults affecting data

The idea behind this class of rules is to duplicate every variable in the program and to check for the consistency of the two copies after each read operation on the variable. Every fault occurring during program execution can be detected as soon as the variable becomes the source operand for an instruction. Errors affecting variables after the last usage are not detected.

The reader should note that the above class of transformation rules is able to detect any error affecting the circuit's memory elements, no matter the number of affected bits and the real location of the storage element (e.g., processor register, cache element, memory cell).

2.2. Transformations for faults affecting code

To detect faults affecting the code, we exploit two ideas. The first is to execute any operation twice, and then verify the coherency of the resulting execution flow. Since most operations are already duplicated due to the application of the rules introduced in the previous sub-section, this idea mainly requires the duplication of the jump instructions. In the case of conditional statements, this can be accomplished by repeating twice the evaluation of the condition.

The second principle aims at detecting those faults modifying the code so that incorrect jumps are executed (either by transforming an instruction into a jump instruction, or by modifying the operand of an existing jump instruction), resulting in a faulty execution flow. This is obtained by associating an identifier to each *basic block* in the code. An additional instruction is added at the beginning of each block of instructions. The added instruction writes the identifier associated to the block in an ad hoc variable, whose value is then checked for consistency at the end of the block.

3. Efficiency of the Approach

In this section we outline the environment that we set up to evaluate the effectiveness of the proposed software hardening method.

3.1. The transformation tool

We built a prototypical tool able to automatically implement the above transformation rules. The tool can potentially work on any program in C language, although some limitations of the current version can prevent its application when some unsupported features are used. The tool has been developed by means of the Bison and Flex freeware compiler construction tools developed in the frame of the GNU Project [10] and comprises about 4,800 lines of C code.

3.2. The software fault injection environment

In order to assess the effectiveness of the proposed transformation rules, we resorted to a set of fault injection campaigns. They have been performed on a prototypical board (in the following *Transputer board*) which has been originally designed for carrying out the

injection of upset-like transient faults. This board, developed at TIMA laboratory within the frame of the Microelectronics and Photonics TestBed international satellite project [11], was initially designed to provide evidence of the intrinsic robust tolerance of digital artificial neural networks. The satellite carrying MPTB boards was launched at the end of 1998 and is still operational.

The Transputer board mainly includes a *T225 transputer* (a RISC microprocessor with parallel capabilities); a *4 Kbyte PROM*, containing the executable code of the system code; a *32 Kbyte SRAM*, used for the storage of T225 program workspaces, programs and data; an *anti-latchup circuit*, for the detection of abnormal power consumption situations and the activation of the corresponding recovering mechanisms; a *watch-dog system*, refreshed every 1.5 seconds 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. For the purpose of our experiments, faults are randomly injected in the proper locations during the program execution. To be consistent with the characteristics of transient errors, which occur in actual applications executing in a radiation environment, we performed the injection of single faults on randomly selected bits belonging to the code and data area. The injection mechanism is implemented by a dedicated 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 monitored, the fault is classified, and the results are sent to the PC acting as a host system.

The adopted technique allows performing injection experiments with a low degree of intrusiveness. It adds about 400 bytes of additional code (for injection purposes) to the program under test. Moreover, thanks to the T225 features, the time spent during context switching between the injection and the application process (and therefore the time overhead for injecting faults) is negligible.

3.3. The radiation environment

To get more confidence on the significance of the results gathered through software fault injection experiments, we performed preliminary radiation testing with equipment based on a Californium fission decay source (available at ONERA/DESP, Toulouse, France). The same Transputer board described above was used during this set of experiments. It is important to note that during these tests, only the Transputer board program memory was exposed to the effects of the particles issued from the Cf_{252} fission.

4. Experimental results

The experiments we performed are based on carrying out extensive fault injection sessions on three benchmark programs:

- *Matrix*: multiplication of two 10x10 matrices composed of integer values
- *BubbleSort*: an implementation of the bubble sort algorithm, run on a vector of 10 integer elements
- *QuickSort*: a recursive implementation of the quick sort algorithm, run on a vector of 10 integer elements.

For each program we performed the following steps:

- Generation of the modified version by exploiting the transformation tool described in Section 2.
- Calculation of the overhead in terms of code size increase with respect to the original version. The resulting code size was around 4 times the original code for the 3 benchmark programs.
- Evaluation of the overhead in terms of slow-down with respect to the original version by running the original and modified codes with the same input values on the Transputer board. The figures we measured range from 2.1 to 2.5 times for the 3 benchmark programs.
- Realization of two fault injection sessions (performed through software fault injection and irradiation) for each benchmark: one on the original version of the program, the other on the modified one. Each fault injection session is split in two experiments. During the first one, faults are injected in the memory area containing the code; during the other, faults are injected in the memory area containing the program data. When considering the software fault injection experiments, the number of faults injected in each session was 1,000 for the original version of the programs. In the modified version we injected a number of faults obtained by multiplying 1,000 by the memory size increase factor, thus accounting for the higher probability that a fault affects the memory.

Faults were classified in the following categories according to their effects on the program:

- *Effect-less*: The injected fault does not affect the program behavior.
- *Software detected*: The rules presented in Section 2 detect the injected fault.
- *Hardware detected*: The fault triggers some hardware mechanism (e.g., illegal instruction exception, watch dog).
- *No answer*: The program under test triggers some time-out condition, e.g., it entered an endless loop.
- *Wrong answer*: The fault was not detected in any way and the result is different from the expected one.

	Version	#Injected Faults	Effect-less	SW detected	HW detected	No answer	Wrong answer
Matrix	Original	1,000	100	0	412	96	392
	Modified	4,488	1,856	2,312	256	52	12
BubbleSort	Original	1,000	108	0	231	163	498
	Modified	4,530	1,498	2,692	284	54	2
QuickSort	Original	1,000	112	0	357	56	475
	Modified	4,956	1,624	2,926	350	44	12

Table 1: Faults are injected in the memory area containing program code.

	Version	#Injected Faults	Effect-less	SW detected	HW detected	No answer	Wrong answer
Matrix	Original	1,000	199	0	0	0	801
	Modified	2,044	385	1,659	0	0	0
BubbleSort	Original	1,000	235	0	0	0	765
	Modified	2,538	657	1,881	0	0	0
QuickSort	Original	1,000	240	0	0	0	760
	Modified	2,056	486	1,570	0	0	0

Table 2: Faults are injected in the memory area containing data.

4.1. Software Fault Injection Results

Table 1 and Table 2 report the results of the software fault injection experiments performed on the memory area containing the code and the data, respectively. Due to the increase in the code and data size, the number of effect-less faults significantly increases in all cases. The hardware detection mechanism implemented by the Transputer processor (a watch-dog) accounts for the relatively high number of faults affecting the code which are detected in this way.

The main observation issued from the analysis of these results is that the number of undetected faults producing wrong program results is nearly reduced to zero when faults affecting the *code* are considered. Indeed, an average 45% of faults injected in the original program induced wrong answers, while for the modified code this percentage is lower than 1%. When faults affecting the *data* are considered, they are *all* detected in the modified program. Note that for the original program around 80% of faults injected on data areas led to wrong program results.

From these experiments we can conclude 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.

4.2. Rule Analysis

In order to provide the reader with an analysis of the relative effectiveness of the different rules we proposed, we grouped rules in three groups:

- Group 1: data duplication
- Group 2: conditional statements
- Group 3: control flow checking.

We then classified each fault detected in the modified version of the programs according to the rule it triggered. We report in Table 3 the percentage of faults detected by each group of rules, with respect to the total number of detected faults.

Faults in the code area			
	Group 1	Group 2	Group 3
Matrix	45.5	39.8	15.7
BubbleSort	51.7	29.8	18.5
Faults in the data area			
	Group 1	Group 2	Group 3
Matrix	97.9	1.9	0.2
BubbleSort	93.8	3.05	3.15

Table 3: Percent number of faults detected by each group of rules w.r.t. total number of detected faults.

Results show that:

- Data duplication is responsible for most of the detection capabilities of the method (about 50% of detected faults in the code, almost all in the data).
- The effectiveness of the rules aiming at hardening conditional statement depends on the considered program, and accounts for about one third of the whole method effectiveness.
- There are still a significant percentage of faults detected through the checks on basic block execution.

4.3. Radiation testing results

Due to schedule constraints, we are currently able to provide results from the radiation experiments concerning only the *matrix* program. The goal of this set of experiments was to collect experimental data about the number of upsets detected by the implemented software rules and to identify possible upsets escaping these rules.

Table 4 reports the obtained results. To allow an easy comparison of radiation data with those obtained through software fault injection, we have included results of a fault injection experiment having approximately the same number of upsets, and affecting the code memory area, only.

The analysis of these data shows a good correlation between the results obtained using the two fault injection environments. An in-depth analysis of the gathered results shows that differences mainly stem from the fact that during the current radiation campaigns, the code and data memory area are not restored to their correct

initial value before every experiment; rather, they are re-initialized when a fault is detected, only. In this way, accumulation of fault effects can happen, producing results which are slightly different than those coming from the software fault injection campaign.

Despite these slight differences, results of Table 4 show that:

- The data gathered through the software fault injection campaigns are mainly confirmed by those provided by the preliminary radiation campaign, thus supporting the confidence in the effectiveness of this evaluation approach.
- The proposed method for software hardening is able to detect a very high percent of the injected faults.

Nevertheless, more complete ground testing is planned to draw firm conclusions about the suitability of the studied technique for guaranteeing safe operation of critical applications.

	Radiation Test	SW Fault Injection
# Upsets	4,377	4,488
Effect-less	2,136	1,856
SW detection	1,920	2,312
HW detection	204	256
Wrong answer	93	12
No answer	24	52

Table 4: Radiation testing vs. software fault injection for the matrix multiplication benchmark.

5. Conclusions

In this paper we experimentally evaluated the effectiveness of a new technique for attaining safety in a microprocessor-based application. The technique is exclusively based on modifying the application code and does not need any special hardware requirement. Since it is based on simple transformation rules to be applied to high-level code, the method can be easily automated and is completely independent on the underlying hardware. We recently performed more extensive fault injection experiments to support this claim, whose results can be found in [12]. The experimental results reported in this paper, gathered by performing fault injection experiments as well as radiation testing on both the original and the hardened version of a set of benchmark programs, show that the method is very effective in reaching a high fault detection level. As a consequence, we can conclude that the method is suitable for usage in those low-cost safety-critical applications, where the strict constraints it involves in terms of memory overhead (about 4 times) and speed decrease (about 2.5 times) can be balanced by the low cost and high reliability of the resulting code.

We are currently working to evaluate the proposed approach on some real industrial applications. At the same time, a new version of the rules is under study to provide more flexibility in trading-off area and time overhead with fault coverage.

6. References

[1] B. Randell, “System Structure for Software Fault Tolerant,” IEEE Trans. on Software Engineering, Vol. 1, No. 2, June 1975, pp. 220-232

[2] A. Avizienis, “The N-Version Approach to Fault-Tolerant Software,” IEEE Trans. On Software Engineering, Vol. 11, No. 12, Dec. 1985, pp. 1491-1501

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

[4] 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

[5] K. Wilken, J.P. Shen, “Continuous Signature Monitoring: Low-Cost Concurrent Detection of Processor Control Errors”, IEEE Trans. on Computer-Aided Design, Vol. 9, No. 6, June 1990, pp. 629-641

[6] P. Shirvani, N. Saxena, E.J. McCluskey, “Software-Implemented EDAC Protection against SEUs”, to be published on IEEE Transactions on Reliability

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

[8] A. Benso, S. Chiusano, P. Prinetto, L. Tagliaferri, "A C/C++ Source-to-Source Compiler for Dependable Applications," International Conference on Dependable Systems and Networks, DSN 2000, pp. 71 -78

[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, 1999, pp. 210-218

[10] J. Levine, T. Mason, D. Brown, Lex & Yacc, 2nd Edition October 1992, O'Reilly & Associates, Inc.

[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”, Proceedings of RADECS'99

[12] M. Rebaudengo, M. Sonza Reorda, M. Violante, P. Cheynet, B. Nicolescu, R. Velazco, "Evaluating the effectiveness of a Software Fault-Tolerance technique on RISC- and CISC-based architectures," IEEE On-Line Testing Workshop, 2000, pp. 17-21