
JOURNAL OF ELECTRONIC TESTING: Theory and Applications 20, 433–437, 2004

A New Approach to Software-Implemented Fault Tolerance*

M. REBAUDENGO, M. SONZA REORDA AND M. VIOLANTE
Dip. Automatica e Informatica, Politecnico di Torino, Torino, Italy

Received September 9, 2002; Revised May 12, 2003

Editors: F. Vargas and V. Champac

Abstract. A new approach for providing fault detection and correction capabilities by using software techniques only is described. The approach is suitable for developing safety-critical applications exploiting unhardened commercial-off-the-shelf processor-based architectures. Data and code duplications are exploited to detect and correct transient faults affecting the processor data segment, while control flow instruction duplication is used for detecting and correcting faults affecting the code segment. Results coming from extensive fault injection campaigns showed the effectiveness and the limitations of the method.

Keywords: software-implemented fault tolerance, single event upsets, fault injection

1. Introduction

In the last years several software-based approaches have been proposed to guarantee fault detection capabilities to programs running on un-hardened processors (e.g., [1, 4–6]). These approaches offer the possibility of detecting hardware errors, provoked by permanent or transient faults, without resorting to special-purpose hardware, but just by relying on carefully crafted software. These works are motivated by the need for low-cost approaches for guaranteeing acceptable levels of dependability in new application areas where cost is a major issue, and where the adoption of commercial-off-the-shelf components is mandatory.

As far as faults affecting program execution flow are considered, the most recent works suggest to modify the application code by introducing suitable instructions, either in the high-level source code or in the low-level assembler code, in charge of checking whether faults have modified the expected program control flow. Furthermore, when faults affecting stored data are concerned, the approaches rely on information redundancy to store multiple copies of the same information and on consistency checks introduced to verify the coherence of the replicated information. These approaches are effective in dealing with both transient faults [1] and permanent faults [4–6], as experimentally demonstrated by fault injection and radiation testing experiments. However, they are not intended for tolerating faults. Indeed, when a fault is detected, they either abort the computation or call an external fault recovery procedure. As a result, they do not provide any support for masking the effects of detected faults.

The purpose of the approach proposed in this paper is to provide programs with the capabilities to detect and correct transient faults that modify either the data the programs manipulate or their normal execution flow. The approach is based on transformation rules that can be automatically applied to the program source code. As a result, it achieves fault tolerance while simplifying the task of hardening programs.

Several transient phenomena, like electromagnetic interference, power disturbances and highly energized particles, may cause temporary, i.e., transient,

*This work has been partially supported by the Italian Ministry for University through the Center for Multimedia Radio Communications (CERCOM).

alteration to the data that programs manipulate or to the expected execution flow of programs. In this paper we focus only on the effects of highly energized particles, like alpha particles, neutron, etc., that by interacting with the silicon substrate of memory devices within processor-based systems may alter the stored information, thus modifying in un-predicable manners the results of programs. The fault model that is normally exploited to model this effects, known as Single Event Upsets (SEUs), is the single transient bit-flip [3].

In the approach presented in this paper program variables are duplicated and so are the operations on them using a technique similar to that presented in [1]. In order to tackle fault tolerance (not considered in [1]), one additional checksum is also computed and stored. During the program execution, the checksum is updated every time the content of a variable is updated. Faults affecting the data the program manipulates are detected by checking for coherence the two copies of a variable after every read operation on it. When a mismatch is found, a recovery procedure is executed that corrects the fault, by relying on the duplicated variables and the stored checksum. Thanks to this solution, the approach is able to provide fault tolerance without the high memory overhead a straightforward approach as variable triplication implies. In this paper, which is based on the work we presented at the Latin American Test Workshop 2002, we extend and improve the work presented in [7] by tackling faults affecting the program execution flow, too. This type of faults is detected using the control flow instruction duplication method that is described in [1]. The recovery procedure we adopted upon detection of this kind of faults is based on a rollback scheme: as soon as a fault affecting the program execution flow is detected the program execution is restarted. The approach is thus able to *detect* and *correct* transient faults affecting the data and some of the faults affecting program execution flow (e.g., transient faults in the processor internal memory elements like program counter and stack pointer). Moreover, it is able to *detect* transient faults producing permanent modifications to the program execution flow (e.g., SEUs in the memory area storing the program code).

In order to gather experimental evidence of the effectiveness of the proposed approach, we adopted three simple benchmark programs, we hardened them according to our approach and we performed several fault injection experiments on both versions (the unhardened and the hardened one) for each benchmark. Moreover, we evaluated the overhead the method introduces in terms of memory occupation and program execution time. From our experimental analysis, we observed that all the faults affecting program execution are detected and when possible corrected, thus improving the dependability of the considered application. This result is obtained by increasing the program data segment by an average factor of about 2.2, while the code segment is increased in the average by a factor of about 3.5. The experiments also showed that the time penalty we have to pay is about 3.1.

The paper is organized as follows. Section 2 describes the proposed approach in details, while Section 3 reports the results we gathered during extensive fault injection campaigns. Finally, Section 4 draws some conclusions.

2. Software-Based Fault Tolerance

In this paper we focus on computing-intensive applications, only. Therefore, we assume that the program to be hardened begins with an *initialization phase*, during which the data to be elaborated are acquired. This phase is then followed by a *data manipulation phase*, where an algorithm is executed over the acquired data. At the end of the computation, the computed results are committed to the program user, through a *result presentation phase*. The code transformation rules we present in the following are intended to be applied on the algorithm executed during the data manipulation phase.

The approach exploits several code transformation rules providing fault detection and, for most cases, fault correction. The rules are intended for being automatically applied to the program source code and can be classified in two broad categories: rules for detecting and correcting faults affecting data (described in Section 2.1.1) and rules for detecting and when possible correcting faults affecting code (described in Section 2.1.2).

2.1. *Detecting and Correcting Transient Faults Affecting Data*

Data hardening is performed according to the following rules:

- Every variable x must be duplicated: let x_0 and x_1 be the names of the two copies. Every write operation performed on x must be performed on x_0 and x_1. Two sets of variables are thus obtained, the former (set 0) holding all the variables with footer 0

and the latter (set 1) holding all the variables with footer 1.
- After each read operation on x, the two copies x_0 and x_1 must be checked, and if an inconsistency is detected a recovery procedure is activated.
- One checksum c associated to one set of variables is defined. The initial value of the checksum is equal to the exor of all the already initialized variables in the associated set.
- Before every write operation on x, the checksum is re-computed, thus canceling the previous value of $x(c' = c \hat{} x_0)$.
- After every write operation on x, the checksum is updated with the new value $x'(c' = c \hat{} x'_0)$.

The recovery procedure re-computes the exor on the set of variables associated to the checksum (set 0, for example), and compares it with the stored one. Then, if the re-computed checksum matches the stored one, the associated set of variables is copied over the other one; otherwise the second set is copied over the first one (e.g., set 0 is copied over set 1, otherwise set 1 is copied over set 0).

2.2. *Detecting and Correcting Transient Faults Affecting Code*

To detect faults affecting the code we exploit the techniques introduced in [1]. The first technique consists in executing 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 technique aims at detecting those faults modifying the code so that incorrect jumps are executed, 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.

The recovery procedure consists in a rollback scheme: as soon as a fault affecting the program execution flow is detected, the program is restarted (i.e., the program execution is restarted from the data manipulation phase). Thanks to this solution, we are able to:

- *Detect and correct* transient faults located in the processor internal memory elements (e.g., program counter, stack pointer, stack memory elements) that temporarily modify the program execution flow.
- *Detect* transient faults originated in the processor code segment (where the program binary code is stored) that permanently modify the program execution flow. As soon as a SEU hits the program code memory, the bit-flip it produces is indeed permanently stored in the memory, causing permanent modification to the program binary code. Restarting the program execution when such a kind of fault is detected is insufficient for removing the fault from the system. As a result, the program enters in an end-less loop, since it is restarted every time the fault is detected. This situation can be easily identified by a watchdog timer that monitors the program operations.

3. Experimental Results

This section describes the experimental results we gathered in order to evaluate the effectiveness of the proposed approach. Starting from a set of benchmark programs, we first obtained their fault tolerant versions by applying the source code transformation rules presented in Section 2. For this purpose we exploited an extended version of the tool presented in [8]. We then evaluated the area overhead our method introduces by measuring the size of the code and data segments of the fault tolerant versions and by relating them with those of the unhardened ones. Moreover, we measured the time overhead the method introduces, as the ratio between the number of clock cycles needed for executing the fault tolerant programs and the unhardened ones. Following the overhead analysis, we evaluated the error detection and correction capabilities of our method. During the experiments, we injected randomly selected (both in time and space) bit-flips in the program *data segment*, storing the data the program manipulates and the stack, and in the *code segment* storing the binary code the processor executes. To take into account for the increased data segment size, we injected a higher number of faults during the analysis of the hardened version of the benchmark. The number of injected faults for the fault tolerant program is computed by multiplying the number of injected faults for the unhardened one by the data/code segment size increase. For each benchmark, we executed a preliminary set of fault injection experiments to measure the impact of faults in the

unhardened program; we then executed a new set of fault injection experiments on the fault tolerant version of the same program. Fault injection experiments were performed resorting to the emulation-based environment presented in [2]. Fault effects are classified according to the following categories:

- *Wrong answer*: the results produced by the faulty processor are different than those produced by the fault-free processor.
- *Effect-less*: the results produced by the faulty processor are equal to those produced by the fault-free processor.
- *Time-out*: the faulty processor is not able to produce the expected result after a given amount of time.
- *Corrected*: the fault is detected and corrected, and thus the processor is able to produce the expected results.
- *Detected*: the fault is detected but cannot be corrected.

In our experiments we considered three programs: *Sieve* implements the sieve of Eratosthenes over a set of N_0 bytes; *Bubble sort* implements the Bubble sort algorithm over a set of N_1 integers; *Matrix* implements the product of two $N_2 \times N_2$ matrices of integer numbers. The processor core we considered implements the Intel 8051 instruction set and it offers to programmers a 128-bytes internal memory. Moreover, it is able to run programs up to 1024-bytes long. Given these constraints we adopted the following set of parameters for the considered benchmark programs: $N_0 = 40$, $N_1 = 10$, $N_2 = 2$. While evaluating the overhead our approach introduces, we recorded an increase factor for the data segment of all the benchmarks of 2.2, an average increase factor of 3.5 for the code segment, and an average time overhead of 3.1. To compare these figures with a reference approach, we implemented the TMR version of the considered benchmarks, obtaining an average data segment overhead of 3.5, an average code segment increase of 3.0 and an average time overhead of 3.1. As a result, we can state that the proposed approach is able to provide fault tolerance while reducing the memory overhead with respect to the TMR approach, while the performance penalties introduced by the two methods are comparable.

The results we gathered during fault injection experiments are reported in Tables 1 and 2, where transients faults injected in the unhardened programs are categorized according to their effects and then compared with those injected in the fault tolerant versions. The

Table 1. Fault injection in data segment.

	Sieve		Bubble sort		Matrix	
	Unhardened	Hardened	Unhardened	Hardened	Unhardened	Hardened
Injected	10,000	22,000	10,000	22,000	10,000	22,000
Effect-less	8,294	14,271	9,227	17,728	9,398	18,068
Corrected	0	1,789	0	3,449	0	1,436
Detected	0	5,934	0	823	0	623
Wrong answer	1,701	0	773	0	580	0
Time-out	5	6	0	0	22	1,881

Table 2. Fault injection in code segment.

	Sieve		Bubble sort		Matrix	
	Unhardened	Hardened	Unhardened	Hardened	Unhardened	Hardened
Injected	10,000	32,000	10,000	39,000	10,000	33,000
Effect-less	9041	17,487	9,136	23,587	8,944	22,319
Corrected	0	1,594	0	2,283	0	1,037
Detected	0	614	0	39	0	430
Wrong answer	416	0	637	0	579	0
Time-out	543	12,305	227	13,091	487	9,214

figures show that the proposed method is able to significantly improve the error detection and correction capabilities of a given applications. As far as faults inside the data segment are considered, we observed that the method provides complete fault coverage: the number of *wrong answer* is indeed reduced from 3,054 (for the unhardened programs) to 0 (for the hardened ones). The same result was observed when faults affecting the code segment were analyzed, where wrong answers are reduced to 0 starting from the initial figure of 1,632. From Tables 1 and 2, we can also observe that many faults exist that can only be *detected*. Most of them are provoked by SEUs hitting the memory area storing the result of the program at the very beginning of the program execution. Furthermore, many faults hitting the code area are classified as *time-out*. These are faults that let the program enter in an end-less loop as described in Section 2.1.2 and that trigger the watchdog timer our fault injection system embeds.

4. Conclusion

The paper presented a software-implemented approach for tolerating the effects of transient faults affecting both the data and the code segments of unhardened processors. The major novelty of the approach is the possibility of correcting SEU effects before they result in program errors. Fault detection and correction is obtained by resorting to variable duplication and checksum computation for guaranteeing the coherency of the replicated information. Moreover, control flow instruction duplication in conjunction with a rollback scheme is used to detect and correct faults modifying the program execution flow. Results issued from emulation-based fault injection experiments show that the method is able to tolerate all the injected faults, at a cost of about 2.5 times increase in the required memory and of a 3 times increase in the program execution time. As future work we plan to extend our approach to cope with other fault type, namely permanent faults and faults provoked by electromagnetic interference.

References

1. P. Cheynet, B. Nicolescu, R. Velazco, M. Rebaudengo, M. Sonza Reorda, and M. Violante, "Experimentally Evaluating an Automatic Approach for Generating Safety-Critical Software with Respect to Transient Errors," *IEEE Trans. on Nuclear Science*, vol. 47, no. 6, pp. 2231–2236, 2000.
2. P.L. Civera, L. Macchiarulo, M. Rebaudengo, M. Sonza Reorda, and M. Violante, "Exploiting Circuit Emulation for Fast Hardness Evaluation," *IEEE Trans. on Nuclear Science*, vol. 48, no. 6, pp. 2210–2216, 2001.
3. M. Nicolaidis, "Time Redundancy Based Soft-Error Tolerance to Rescue Nanometer Technologies," *IEEE VLSI Test Symposium*, pp. 86–94, 1999.
4. N. Oh, S. Mitra, and E.J. McCluskey, "ED4I: Error Detection by Divers Data and Duplicated Instructions," *IEEE Trans. on Computer*, vol. 51, no. 2, pp. 180–199, 2002.
5. N. Oh, P.P. Shirvani, and E.J. McCluskey, "Error Detection by Duplicated Instructions In Super-Scalar Processors," *IEEE Trans. on Reliability*, vol. 51, no. 1, pp. 63–75, 2002.
6. N. Oh, P.P. Shirvani, and E.J. McCluskey, "Control Flow Checking by Software Signature," *IEEE Trans. on Reliability*, vol. 51, no. 1, pp. 111–122, 2002.
7. M. Rebaudengo, M. Sonza Reorda, and M. Violante, "A New Software-Based Technique for Low-Cost Fault-Tolerant Application," *IEEE Reliability and Maintainability Symposium*, pp. 25–28, 2003.
8. M. Rebaudengo, M. Sonza Reorda, M. Torchiano, and M. Violante, "A Source-to-Source Compiler for Generating Dependable Software," *IEEE Intl. Workshop on Source Code Analysis and Manipulation*, pp. 33–42, 2001.