Software dependability techniques validated via fault injection experiments

Authors: Benso A., Di Carlo S., Di Natale G., Prinetto P., Tagliaferri L.,

Published in the Proceedings of the IEEE 6th European Conference on Radiation and Its Effects on Components and Systems (RADECS), 10-14 Sept. 2001, Grenoble, FR.

DOI: 10.1109/RADECS.2001.1159292

Software Dependability Techniques validated via Fault Injection Experiments

A. BENSO, S. DI CARLO, G. DI NATALE, P. PRINETTO, L. TAGLIAFERRI

Abstract

The present paper proposes a C/C++ Source-to-Source Compiler able to increase the dependability properties of a given application. The adopted strategy is based on two main techniques: variable duplication/triplication and control flow checking. The validation of these techniques is based on the emulation of fault appearance by software fault injection. The chosen test case is a client-server application in charge of calculating and drawing a Mandelbrot fractal.

1. INTRODUCTION

Nowadays, the use of computer-based systems manages multiples aspects of our life and an increasing number of critical applications relies on their functions. The tasks in which ECS (Embedded Computer Systems) are involved are becoming more and more complex concerning crucial duties like aircraft, trains and medical control systems. In this context, ECS plays a crucial role in ensuring data security and human safety; therefore it is mandatory that their tasks were appropriately accomplished.

It can be observed that, while circuits size decrease, clock frequency increase. These aspects, coupled with the fact that processors are often placed in electrically active environments, can favor transient errors incidence. The commonly used technique to detect this kind of errors is based on the on-line testing techniques able to ensure high dependability without heavily affecting the system performance.

The development of custom products with high performance and dependability level it is not always an acceptable task both from the economical point of view and for the manufacturing time. These constraints force the massive use of commercial off-the-shelf components (COTS) both in software and in hardware domains. These components are usually not developed to work in unfavourable environments where high dependability is the essential requirement. The goal is to realize fault tolerant and reliable systems starting from off-the-shelf hardware and software components.

The techniques involved in building fault tolerant ECSs rely both on hardware and software redundancy.

Hardware redundancy is a powerful and very effective resource but sometimes it is inapplicable for the cost it implies. On the other hand, software redundancy, while often effective, can slow down the system performances. However, this second solution can be implemented with very low costs.

Software redundancy techniques exploits additional memory and/or execution time to guarantee the correctness of the computation (and, hence, the code integrity) and of the data stored in memory. The techniques employed in the construction of such software are called *Software Implemented Hardware Fault Tolerance* (*SIHFT*) since they handle hardware errors with the software aid. In particular, many studies show how it is possible to verify the integrity of the variables that populate a program [1][2] and of the executed code [4-13]. All these strategies rely on ad hoc modification of the high-level source code, with the introduction of routines able to periodically test the memory integrity. Even though these methods differ in their approach (data protection or code protection) their purpose is always producing a *Fail-Silent* system, i.e., a system that produces only correct results.

Methods based on variable duplication aim at reducing the situations in which the *ECS* produces incorrect results, whereas the application gives the impression to correctly terminate. This kind of malfunction is called *Fail-Silent Violation* and typically is caused by an alteration of a variable value [14] [15].

Methods based on control flow point to verify the correctness of the program control flow. They are, therefore, suitable to detect faults appearing in the code more than on the variables. These solutions mainly rely on the use of software signature checking [6-13]. The application program is split into elementary blocks, i.e., block with one identified entry and exit point. A signature is computed off-line by means of the instructions contained in the block and then is stored in a suitable data structure. At run-time the signature is computed again and compared with the previously stored one. The hardware deputed to maintain this kind of statistics is a so-called *watchdog processor* [5].

This approach has demonstrated to be very effective but unluckily shows two main drawbacks: first of all a hardware modification (an effort which cannot always be supported); second, *watchdogs* can only cover main memory faults but not the memory cache ones.

In order to solve these two weaknesses the research has moved in the direction of pure software implementation. Typical solutions are Block Signature Self Checking (BSSC) [16] and Control Checking with Assertion (CCA) [17]. These researches essentially exploit the previously introduced concepts of *watchdogs* but the computation of the signatures is performed by a software process and not by a hardware component.

This paper presents a new reliable compiler able to enhance the dependability of a given C/C++ source code. The compiler joins the approaches presented by the authors in [2] with the RECCO (Reliable C/C++ Compiler) tool and in [18] obtaining a single integrated approach able to deal with both data and code errors. This new tool named RECCO* targets the improvement of the dependability properties of C/C++ source code by introducing apposite routines able to protect the data stored in memory (via data duplication/triplication) and detecting the deviations from the right control flow due to erroneous code executions (with the use of control flow checking) . The main purpose is to show how it is possible to couple these two techniques to produce a high level dependability application able to self-detect errors injected in its memory area. Our approach has been then validated with the use of software fault injection tool [19].

The paper is organized as follow: Section 2 introduces some basic concepts about the structure of the compiler itself whereas Section 3 defines the adopted fault model and the fault injection environment. To prove the effectiveness of the work Section 4 reports experimental results performed on a benchmark. Finally Section 5 draws some conclusions.

2. The Tool

RECCO* is a source-to-source compiler; it converts a C/C++ code into a reliable version with the same functionalities. The high reliability level is reached by the introduction of routines that periodically check the content of the memory (both data and code) to detect corruptions.

Figure 1 sketches the structural design of RECCO* identifying the different tasks of the compiling flow.

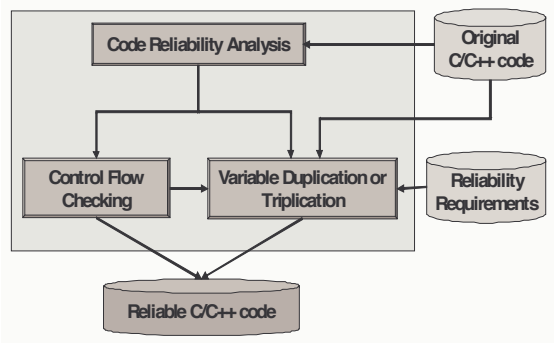

Figure 1: The RECCO* tool

First of all (*Code Reliability Analysis*) the compiler acquires information about the code structure itself and the variables used in the program. The tool builds up a dependency graph that defines the correlation and the dependencies among the variables.

At the same time the control flow of the code is examined: the program is split into branch-free blocks (defined as the biggest blocks with one entry and one exit point) and each of them is assigned to a unique identifier.

By means of these identifiers the compiler builds a graph describing how the program control flow can progress. This report is saved in an auxiliary file using the regular expression formalism [18].

Figure 2 sketches an example of how the compiler handles a typical control flow. The *Block* labels are associated with sequential operations that do not contain branch instructions whereas *Dec* labels show the presence branch instructions. The regular expression describing the example control-flow is: *a[b|c*]d*.

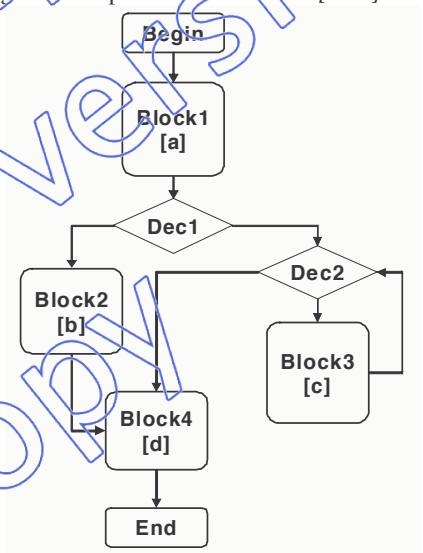

Figure 2: Control Flow

In the *Control Flow Checking phase* the code is enriched with a concurrent process, in charge of monitoring the control flow and verifying its correctness. This kind of test has been suited to intercept code modifications that cause the program to deviate from its standard flow. Realizing the checker as an independent process introduces a modest execution-time overhead since its functions can be efficiently scheduled by the Operating System when the main program is waiting for external inputs (such as I/O operations). As a matter of fact this approach implies the presence of a multitasking Operating System.

During the *Variable Protection phase* the compiler introduces redundant data to allow error detection/correction of the program variables: the variables considered in the *Code Reliability Analysis* are duplicated/triplicated, depending on the user choice. Each time a variable is written its copies are updated whereas when a variable is read, the values stored in its copies are checked for consistency. Therefore, the compiled program is able to asses the reliability of its data and detect (if variables are duplicated) or even correct (if variables are triplicates) the errors occurring in memory locations.

In order to reduce fault latency and to avoid fault propagation through the system, both the tests concerning code and data integrity are performed concurrently with the normal operations.

3. Fault Injection

Concerning the Fault model, data corruption has been reproduced by a single bit flip (*Single Error Upset*, SEU) in the memory locations of the test-case both in the code and in the data (global and stack) areas. The question of how much this fault model represents an appropriate defect induced by the occurrence of real phenomena is crucial. Several software-implemented fault injection studies are dedicated to the analysis of the relationship between fault injected by software and physical faults. In particular NASA [20] researches set up statistical investigations about the most common errors occurring in modern digital circuits. These studies lead to the conclusion that, due to the high miniaturization and the high work frequencies, today circuits are becoming more and more susceptible to the effect of ionizing radiation and noise source. The most commonly observed effects of these kind of disturbs is the SEU.

The effectiveness of the used fault model is increased when dealing with space applications, where the probability of SEU is very high.

To emulate the faults in the test-case memory a Fault Injector has been implemented as a UNIX daemon able to inject errors in random locations of the targeted program at random execution time.

The daemon could be driven by the user to inject faults in different sections of the running program: code, data and stack segment. Looking at the faulty program results and comparing them with the correct ones the injector is able to section the fault effects into the following three categories:

- No effect: the error has no effect on the system;
- Wrong result, i.e., Fail Silent Violation (FSV): the program end but the program results are wrong;
- Crash: the system crashes due to an unrecoverable problem.

The information produced by the fault injector can be used to statistically characterize the effectiveness of RECCO*

4. Experimental results

To prove the effectiveness of the techniques developed in *RECCO**, a test-bench and a specific fault injection policy have been set up.

The test-case is a program able to draw images based on Mandelbrot fractals. It is organized as a client-server application. The client is in charge of drawing the picture using data provided by the server whereas the server waits for client requests to produce new pictures. Once the request is triggered, the server performs the computation and sends the data flow through the network.

This structure has been intended to distribute the workload between two different machines, to support the fault injection experiments and to help the statistics registering. In fact, any injection experiment that causes a malfunction on the server turns in an erroneous depict on the client which can be easily compared with the right one. The results' checking is based on image files comparison. Only the server section of the program has been compiled with *RECCO** and the faults have been injected only on it.

The experiments are repeated on different dependable versions of the same program to underline which are the capabilities of each technique and the influence on the program performance. Each version is compiled selecting some of the options provided by RECCO*. Five benchmarks have been defined:

- Control flow checking only (CF)
- Variable duplication only (VD)
- Variable triplication only (VT)
- Control flow checking and variable duplication (CF+VD)
- Control flow checking and variable triplication (CF+VT)

For each benchmark, a set of 1000 injections is performed.

Table 1 summarizes the overhead introduced by the dependable techniques.

	Original	CF	VD	VT	CF+VD	CF+VT
Binary Code (KB)	16	24	17	19	25	27
Execution Time (s)	4,2	5,1	5,4	6,2	6,5	7,3

Table 1: *Memory and Time overhead*

The code overhead is comprised between 3KB and 11Kb. Nevertheless, for the control flow technique, the most of it is wasted by the control flow checker while a small part is used by the synchronizations routines; for this reason the incidence of this overhead decreases with the code growth.

The first experiment aims at underlining how many code errors and crashes can be detected when the

Control Flow Technique is employed. Table 2 summarizes results of injections on the code segment of the program. In this case, only the CF benchmark has been used because data redundancy is not able to cover transient errors on the code.

For both the original program and the CF benchmark the following information have been provided:

- no effect
- number of crashes;
- number of control-flow errors;
- number of errors not belonging in the set of control-flow errors;
- number of detected control-flow errors.

	Original	CF
# No Effect	620	622
# Crashes	274	269
# Control Flow Errors	75	24
# Other Errors	31	30
# Detected Flow Errors	-	55

Table 2: *Injection on code*

As we can see, the number of crashes decreases by 5 units whereas the number of control flow errors is reduced by 68%.

The second experiment is performed injecting faults in the data segment and the stack segment. In this case, the whole set of benchmarks is used. In fact, control flow checking technique is able to cover (even if in a limited way) some faults occurring in the data segment (for example, indirect jumps or function calls). Table 3 sketches the obtained results. For both the original program and the benchmarks, the following information have been provided:

- No effect
- number of crashes;
- number of Fail Silent Violation (i.e., the program terminates its execution but it generates a distorted image);
- number of detected but not corrected errors.

	Original	CF	VD	VT	CF+VD	CF+VT
# No Effect	718	716	714	978	716	977
# Crashes	143	143	13	12	11	12
# Fail Silent Violation	139	139	12	10	10	11
# Detected but Not corrected Errors	0	2	261	0	263	0

Table 3: *Injection on data*

The number of crashes and fail silent violations is highly reduced. This means that this huge percentage of errors that before the compilation caused an altered image are automatically corrected by the program itself.

To validate our approach, further benchmarks, with different characteristics, have been set up; the results are shown on Table 4 - Table 7. As it can be seen from the two tables the results are similar to the ones obtained in the previous test.

5. Conclusions

The present paper describes a source-to-source compiler able to automatically integrate methodologies to achieve high software dependability. The main feature of the approach is the possibility of checking the different memory areas of a running program with two different methods of action. The data area is protected with variables duplication or triplication whereas the code section is checked for error by control flow checking. This last technique has been implemented resorting to a multi-process approach in order to minimize both memory and execution time overheads.

Experimental results demonstrate the effectiveness of the approach and the low overhead introduced both in terms of additional memory and execution time.

Comparing the percentage of CFE detected by BSSC technique [16] (about 50%), the result is worse than the one achieved by *RECCO**. Moreover, BSSC is applicable to machine code only.

Watchdogs and Triple Modular Redundancy (TMR), instead, can both reach accuracy in detecting CFE and data corruption of about 80-95% with an overhead in the execution time of about 10%. The main drawback is the dependence from dedicated hardware with consequent noticeable modifications of the system.

The disadvantages of these techniques are not shown by *RECCO** that instead deals with C/C++ code which is target machine independent.

6. References

[1] V. Strumpen, *Portable and Fault-Tolerant Software Systems*, IEEE Micro, September-October 1998, pp. 22-32

[2] A. Benso, S. Chiusano, P. Prinetto, L. Tagliaferri, A C/C++ Compiler for Dependable Applications, The International Conference on Dependable Systems and Networks (FTCS-30), New York (NY), USA, June 2000, pp. 71-78

[3] S. S. Yau, F. Ch. Chen, "An Approach to Concurrent Control Flow Checking", IEEE Transaction on Software Engineering, Vol. SE-6, No. 2, pp. 126-137, 1980.

[4] R. Leveugle, T. Michel, G.Saucier, "Design of Microprocessors with Built-In On-Line test", 20th International Symposium on Fault-Tolerant Computing (FTCS-20), pp. 450-456, 1990.

[5] A. Mahamood, E. J. McCluskey, "Concurrent Error Detection Using Watchdog Processor - A Survay", IEEE Transaction on Computer, Vol. 37, No. 2, pp. 160-174, 1988.

[6] M. Namjoo, "Techniques for Concurrent Testing of VLSI Processor Operation", International Test Conference (ITC-82), pp. 461-468, 1982.

[7] M.A. Schutte, J.P. Shen, D. P. Siewiorek, Y. X. Zhu, "Experimental Evaluation of Two

Concurrent Error Detection Schemes", 16th International Symposium on Fault Tolerant Computing (FTCS-16), pp. 138-143, 1986

[8] K. Wilken, J.P. Shen, "Continuous Signature Monitoring: Low-Cost Concurrent Detection of Processor Errors", IEEE Transaction on Computer Aided Design and Systems, Vol. 9, Issue 6, pp. 629-641, June 1990.

[9] 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", Nuclear Science, IEEE Transactions on , Volume: 47 Issue: 6 Part: 3 , Dec. 2000 Page(s): 2231 - 2236

[10] H. Madeira, J. G. Silva, "On-line Signature Learning and Checking", 2nd IFIP Working Conference On Dependable Computing for Critical Applications (DCCA-2), pp. 170-177, Feb. 1991

[11] T. Michel, R. Leveugle, G. Saucier, "A New Approach to Control Flow Checking without Program Modification", 21th International Symposium on Fault- Tolerant Computing (FTCS-21), pp. 334-341, 1991.

[12] Shambhu Upaddhyaya, Bina Ramamurthy, "Concurrent Process Monitoring with No eference Signatures", IEEE Transaction on Computer, Vol. 43 no. 4, pp. 475-480, April 1994

[13] G. Miremadi, J. Ohlsson, M. Rimen, J. Karlsson, "Use of Time and Address Signatures for Control Flow Checking", 5th IFIP Working Conference on Dependable Computing for Critical Application (DCCA-5), pp. 113-124, 1995

[14] A. M. Amendola, A. Benso, F. Corno, L. Impagliazzo, P. Marmo, P. Prinetto, M. Rebaudengo, M. Sonza Reorda, Fault Behavior Observation of a Microprocessor System through a VHDL Simulation-Based Fault Injection Experiment, EURO-VHDL'96, September 1996, Geneva (CH), pp. 536-541

[15] J. G. Silva, J. Carreira, H. Madeira, D. Costa, F. Moreira, Experimental Assessment of Parallel Systems, Proc. FTCS-26, Sendaj (J), 1996, pp. 415-424

[16] G. Miremadi, J. Karlsson, U. Gunneflo, J. Torin, "Two software techniques for on-line error detection", 22th International Symposium on Fault-Tolerant Computing (FTCS-22), pp. 328-335, July, 1992

[17] 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 Transaction on Parallel and Distributed Systems, Vol. 10, No. 6, pp. 627-641, June 1999.

[18] A. Benso, S. Di Carlo, G. Di Natale, P. Prinetto , L. Tagliaferri, Control-Flow Checking Via Regular Expressions, submitted to ATS 2001.

[19] A. Baldini, A. Benso, S. Chiusano, P. Prinetto, "BOND: An Interposition Agents based Fault Injector for Windows NT", IEEE International Symposium on Defect and Fault Tolerance in VLSI Systems (DFT'2000), pp. 387-395, October 2000.

[20] http://tvdg10.phy.bnl.gov/seutest.html

	Floating Point Benchmark		Dicotomic Search		Matrix Multiplication Benchmark		Quick Sort		List Insertion	
	Orig.	Mod.	Orig.	Mod.	Orig.	Mod.	Orig.	Mod.	Orig.	Mod.
Code Size (KB)	48	58	36	43	14	22	15	23	20	29
Execution Time (s)	1,5	1,8	0,4	0,6	6,1	10,5	0,5	0,6	1,5	2
# Crashes	484	471	452	443	420	414	440	432	471	464
# No Effect	465	466	501	503	537	538	512	513	475	478
# Control Flow Errors	25	19	18	9	13	6	19	9	23	12
# Other Errors	26	26	29	29	30	30	29	29	31	31
# Detected Flow Errors		18		16		12		17		15

Table 4: *Benchmarks results for Control Flow* (injections on code)

	Floating Point Benchmark		Dicotomic Search		Matrix Multiplicatio n Benchmark		Quick Sort		List Insertion	
	Orig.	Mod.	Orig.	Mod.	Orig.	Mod.	Orig.	Mod.	Orig.	Mod.
Binary Code Size (KB)	48	63	36	49	14	25	15	27	20	33
Execution Time (s)	1,5	1,8	0,4	0,6	6,1	10,5	0,5	0,6	1,5	2,5
# No Effect	690	983	663	981	653	979	696	992	657	983
# Crashes	139	8	178	7	159	5	135	4	214	9
# Fail Silent Violation	171	9	159	12	188	16	169	4	129	8

Table 5: *Benchmarks results for Control Flow and Data Triplication* (injections on data)

	Floating Point Benchmark		Dicotomic Search		Matrix Multiplication Benchmark		Quick Sort		List Insertion	
	Orig.	Mod.	Orig.	Mod.	Orig.	Mod.	Orig.	Mod.	Orig.	Mod.
Code Size (KB)	48	53	36	42	14	17	15	19	20	24
Execution Time (s)	1,5	1,8	0,4	0,6	6,1	10,5	0,5	0,6	1,5	2,2
# No Effect	690	984	663	983	653	981	696	992	657	985
# Crashes	139	8	178	7	159	5	135	4	214	8
# Fail Silent Violation	171	8	159	2	188	14	169	4	129	7

Table 6: *Benchmarks results for Data Triplication* (injections on data)

	Floating Point Benchmark		Dicotomic Search		Matrix Multiplication Benchmark		Quick Sort		List Insertion	
	Orig.	Mod.	Orig.	Mod.	Orig.	Mod.	Orig.	Mod.	Orig.	Mod.
Code Size (KB)	48	60	36	46	14	24	15	25	20	31
Execution Time (s)	1,5	1,8	0,4	0,6	6,1	10,5	0,5	0,6	1,5	2,2
# Crashes	139	7	178	7	159	5	135	5	214	9
# No Effect	690	685	663	660	653	651	696	691	657	655
# Fail Silent Violations	171	9	159	11	188	16	169	4	129	7
# Detected Errors		299		322		328		300		329

Table 7: *Benchmarks results for Data Duplication+ Control Flow* (injections on data)