Validation of a software dependability tool via fault injection experiments

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

Published in the Proceedings of the IEEE 7th International On-Line Testing Workshop (IOLTW), 9-11 July 2001, Taormina, IT.

DOI: 10.1109/OLT.2001.937809

Validation of a Software Dependability Tool via Fault Injection Experiments

ALFREDO BENSO, STEFANO DI CARLO, GIORGIO DI NATALE, LUCA TAGLIAFERRI, PAOLO PRINETTO
Politecnico di Torino
Dipartimento di Automatica e Informatica
Corso Duca degli Abruzzi 24 - I-10129, Torino, Italy
Email: { benso, dicarlo, dinatale, tagliaferri, prinetto }@polito.it
http://www.testgroup.polito.it

Abstract

The present paper presents the validation of the strategies employed in the RECCO tool to analyze a C/C++ software; the RECCO compiler scans C/C++ source code to extract information about the significance of the variables that populate the program and the code structure itself. Experimental results gathered on an Open Source Router are used to compare and correlate two sets of critical variables, one obtained by fault injection experiments, and the other applying the RECCO tool, respectively. Then the two sets are analyzed, compared, and correlated to prove the effectiveness of the RECCO's methodology.

1. Introduction

The design of dependable digital systems has become a crucial problem from an economical point of view, due to the increasing complexity and quality required by the market. Although the hardware approach, i.e., adding redundant circuitry, offers high performances, it could be not always the most economical solution, due to the high hardware overhead. This reason moves the dependability issue in designing hardware using COTS at the software layer. The fulfillment of these new constraints needs investigating how systems can be designed by assembling commercial hardware and software COTS (Components Off The Shelf). Since these components are not normally designed to guarantee high level of dependability, new approaches able to add reliability to the final system should be defined. In this context, the *software fault tolerance* aims at addressing system failures caused by a *hard* or *soft error* appearing in the system.

The software dependability problem can be mainly faced at three different layers:

- *Operating System layer*: the Operating System can be modified in order to guarantee a higher level of dependability. The approach is very effective but requires a very detailed modification of the Operating System services and a free access to its source code [1].
- *Middleware layer*: interposition techniques are used to design an intermediate software layer able to intercept and possibly modify all the communications between the user application and the Operating System. This technique is very effective when the user application is not modifiable, e.g., when the source code is not available as in commercial software COTS [2].
- *Application Software layer*: check-pointing, algorithm-based fault tolerance, source-to-source recompilation, and software data redundancy are some possible solutions proposed so far to make a software application more dependable. These techniques are the best solution whenever the source code of the target application is available and can be freely modified [3] [4] [5] [6] [7] [8] [9] [10].

All the proposed techniques aim at reducing the unsafe situations in which the system produces incorrect results, whereas the application gives the impression to correctly terminate. This kind of malfunction is called *Fail-Silent Violation* [11] [12]. Recent researches show the relationship between this kind of fails and the appearance of a fault in memory locations. A computer-based system is said to be *Fail-Silent* if its outputs correspond only to correct results. If a fault occurs, the system can either stop the application showing an error message or crash. In both cases, the user is able to detect anomalous behaviors.

Obtaining a Fail-Silent behavior is the subject of many researches that led to the development of techniques like *Algorithm Based Fault Tolerance* (ABFT) [3], *Assertions*, *Time redundancy*, and *Control Flow* checking [4] [5] [6] [7] [8] [9] [10]; these strategies have been proved to furnish high level of fail-silent behaviors in ordinary computers (e.g., not specifically designed to always produce a Fail-Silent behavior) when coupled with the intrinsic Error Detection Mechanisms (EDMs) of the system (exceptions, memory protection, etc.) [13]. Otherwise, a systematic approach for introducing data and code redundancy into an existing source code written in C language is presented in [9]. Nevertheless, all these approaches show various restrictions that reduce their field of action.

In [14] the authors proposed RECCO (*RE*liable *C*ode *CO*mpiler), an ad-hoc Source-to-Source C/C++ compiler able to increase the reliability level of C/C++ source code.

This paper aims at validating the effectiveness of the RECCO approach by the evidence of a case study. Although many simple test benches have been already tested to validate the effectiveness of the tool, a real case study has to be tried. To solve this problem, the network architecture presented in [15] has been targeted as test case.

The accomplishment of the study proposed in this paper relies on a Fault Injection environment realized to inject faults inside the memory of the network system.

The fault model implies the employment of SEU errors (Single Event Upset) that reflect a representative and observed defective behavior [16] [17].

The validation of this approach is made possible comparing the theoretical results obtained with RECCO with the empirical ones collected in [14].

The paper is organized as follow: Section 2 gives a brief overview about the RECCO tool whereas Section 3 presents a test-bench description; section 4 demonstrates the compatibility between the experimental results with the supposition achieved with the aid of RECCO. The overall conclusions are reported in Section 5.

2. The RECCO Tools

RECCO is a source-to-source compiler. It starts from a C/C++ ANSI code and produces a reliable version of it applying a set of code modification rules. Figure 1 sketches the structural design of RECCO identifying the needed tasks during the compiling flow.

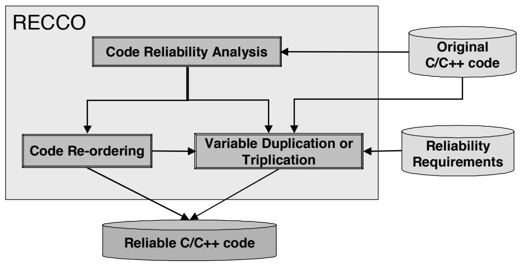

Figure 1: The RECCO tool

The ***RECCO*** Source-to-Source compiling process can be split into three main phases:

- *Code Reliability Analysis*: starting from the original source code the *dependability properties* of the whole variable set is evaluated. For each variable the set of *dependencies* and a *reliability-weight* are computed. The first parameter gives a representation of the *functional dependencies* of each variable; the second one gives an estimation of the variable criticality from a dependability point of view. These parameters are strictly related to the *life-time* of each variable [14].
- *Code Reordering phase*: starting from the information supplied by the Code Reliability Analysis, a first improvement of the code dependability is obtained resorting to a code reordering. The code-reordering phase tries to minimize the reliability-weight of each variable keeping safe the variable dependencies. The generated code is referred to as *Reliable C/C++*.
- *Variable Protection Phase*: it introduces information redundancies into the reordered code to allow on-line error detection/correction capabilities. This goal is achieved by resorting to *variable duplication or triplication*. The resulting code is able to detect faults inside the memory according to the targeted fault model (see Section 1) and in case of variable triplication is able to correct the fault and to recover a safe state. The generated code is referred to as *Reliable*^{2} *C/C++*.

3. Test bench description

The goal of this paper is the validation of the effectiveness of the RECCO approach in a real situation. Therefore, a suitable test environment has been set up to perform a set of fault injection experiments.

As sketched in Section 1, the targeted test architecture is the network system presented in [15].

The adopted network (see Figure 2) has been chosen to meet two mandatory requirements: it should be simple enough to monitor its behavior during fault experiments and, at the same time, enough sophisticated to reflect a real case of study.

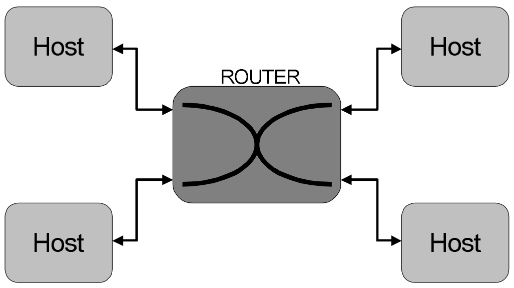

Figure 2: Network architecture

The presented system can reflect a real scenario in which the involved hosts are considered as a complex subnet and the router is deputed to join the different subnets.

Since the most critical element in the targeted architecture is the router, it is the best candidate to be protected using RECCO strategies. To have free access to the source code of this critical component, an *Open-Source* router implemented on a PC running the Linux Operating System equipped with multiple network interfaces [18] has been used.

The fault injection experiments are performed using a complete fault injection tool, distributed among the router and the host computers. The fault injector has been initially used to perform a preliminary fault analysis on the system and to characterize the router behavior in presence of faults [15]. This preliminary study has been used in the present paper as term of comparison to evaluate the effectiveness of the results offered by RECCO.

As mentioned in the introduction, the adopted fault model is the SEU (*Single Event Upset*), consisting of a temporary bit flipping in a memory location.

Faults have been injected within the *Routing Table* and the *Local Data Segment*. The expected router behaviors in case of error are:

- *The router keeps on working properly* (the fault has been overwritten before any further reading)
- *The router still works but packets are not correctly routed* (if an error occurs in the Routing Table the packets are likely to be lost or routed in wrong directions)
- *The router crashes*.

The fault injections have been performed in random moments and memory positions within the above-mentioned sections. The variables corresponding to those locations have been classified by means of the effects they produced:

- *Very Critical Variables* (the router always crashes)
- *Critical Variables* (the router sometimes crashes and the clients do not receive any packets)
- *Non-critical Variables* (the fault produces no effect).

Since the overall list of the variable classification cannot be reported, Table 1 summarizes the percentage of variables in the three mentioned above classes.

Variable effect	Percentage
Non critical	76,3%
Critical	2,3%
Very Critical	21,4%

Table 1: Router variables classification

The injection experiments also show a strict connection between the variable type and its critical level: function pointers are mainly very critical; data pointers are critical whereas the remaining variables are commonly non-critical.

4. RECCO Results

This paragraph presents the set of experiments performed on the Routing System using the RECCO tool.

In particular, we tried to validate the effectiveness of the metrics used to calculate the *reliability-weight* of each variable since it directly affects the code modification performed by RECCO.

After the compiling phase, the list of variables sorted by *reliability-weight* as described in section 2 is compared with the empirical list obtained by fault analysis (see Section 3), to determine a correlation between the two results.

All the variables identified in the empirical analysis as critical are generally positioned at the top rank in the list of variables sorted by *reliability-weight*; however, a direct comparison between the two results can be better articulated. In fact, RECCO does not exploit the concept of criticality and it is thus not possible to identify, in the variable list, a sharp boundary between critical and no-critical variables.

From a first analysis, the only result one can draw is that the function used to calculate the *reliability-weight* ranks highest the most critical variables. To strengthen this hypothesis, we let RECCO re-compile the router several times, each time duplicating a different percentage of variables. Since we are targeting single faults, the duplication of a critical variable is able to transforms it in a not-critical one. If during the recompiling phases, the number of critical variable decreases, we can assume the decision strategy made by RECCO correctly works. Table 2 shows the number of critical and very critical variables related to the number of duplicated ones. The starting point (no variable duplication) is obviously the same presented in Table 1. As expected, increasing the percentage of duplicated variables, the number of critical and very critical variable decreases. The same results are shown in a graphical way in Figure 3.

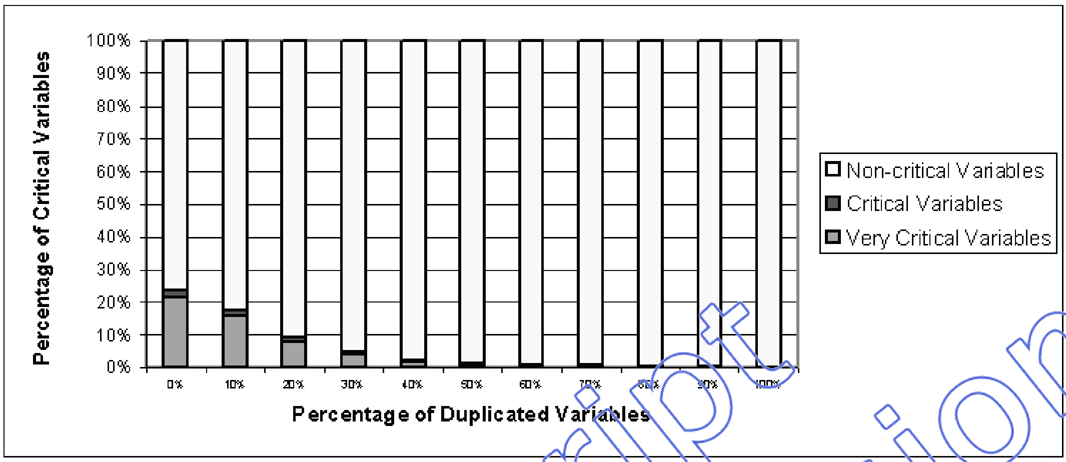

Figure 3: Percentage of Critical Variables with respect to duplicated Variables

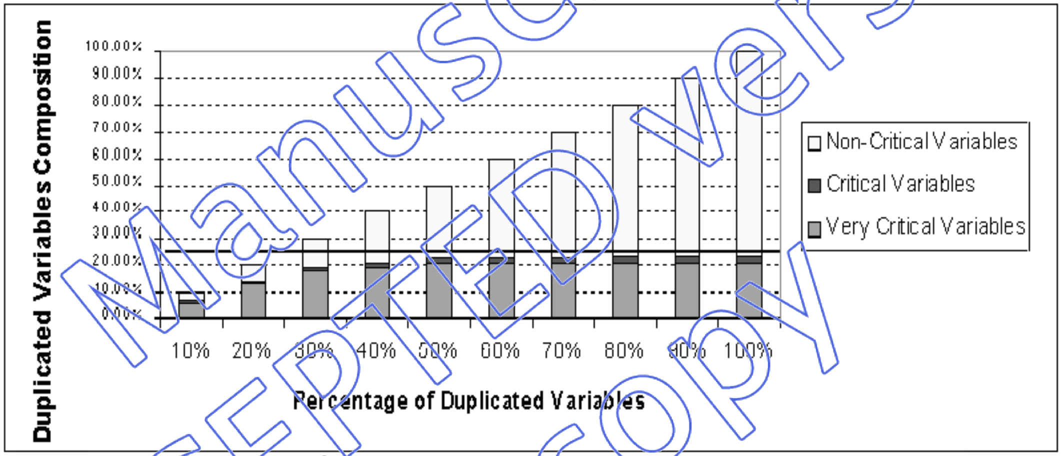

Figure 4: Duplicated Variables Composition

Different considerations can be obtained rearranging the compiling results to underline the precision with which the duplicated variables are distributed among very critical, critical and non-critical classes. In fact, as shown in Table 3, the duplicated variables are initially mostly chosen between the very critical and critical ones. The same results are shown in a graphical way in Figure 4.

The two tables confirm the ability of RECCO to estimate, with an acceptable precision, the criticality of the variables involved in the compiling process.

As final test, to prove our conclusions, the same injection experiments exploited to produce the results in Table 2 has been employed again together with the 100% variable duplication reliable router. Looking at the obtained results, a small percentage of variable (about 1%) fall in the class of very-critical ones.

This imprecision can be mainly imputed to the kind of targeted application. In fact, the router source code exploits many multiple-level pointers, whereas RECCO is able to deal with 2-level pointers only. This limit has been set up since most programs do not address more then 2-level pointers and the needed complexity to handle them is too high.

5. Performance Degradation

The consistency check routines and the redundant data exploited by the compiler introduce a certain amount of time and memory overhead; obviously, these costs are proportional to the number of duplicated variables

Figure draws the performance degradation that ranges from 5% (30% of variables duplicated) to 17% (all the variables duplicated).

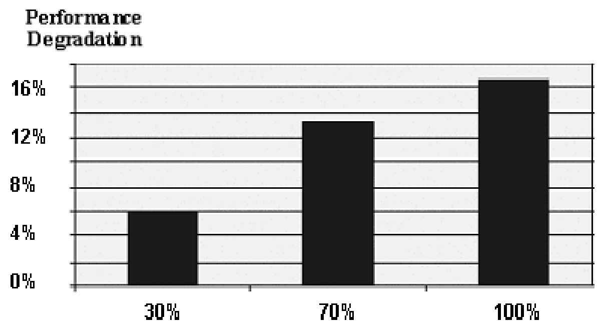

Figure 5: Performance degradation using RECCO

Finally, Figure 6 reports the memory overhead for storing and managing the duplicated variables; the values range from 15% (30% of variables duplicated) to 35% (all the variables duplicated).

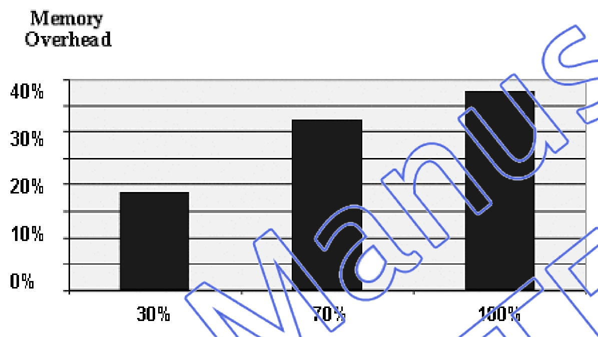

Figure 6: Data memory overhead using RECCO

6. Conclusion

In this paper, we validated an approach to achieve fault tolerance in software applications subjected to hard or soft error appearance, by the evidence of a real test case.

The targeted application is a network architecture containing an open-source router element.

The designed tool is a source-to-source compiler that exploits software techniques (data redundancy); with the aid of this method, applications can detect (or even correct) faults occurrence and inform the user of malfunctions. The results presented in the paper show a significant correlation between the reliability weight assigned by the compiler and the results obtained trough fault injection experiments.

We are currently working toward extending the error detection and correction capabilities to faults occurring in the code section of the target programs.

7. References

[1] C. Perez, G. Fabregat, R.J. Martinez, G. Martin, *Incremental messages: micro-kernel services for flexible and efficient management of replicated data,* FTCS-29: Twenty-Ninth Annual International Symposium on Fault-Tolerant Computing, 1999, pp. 56–63

[2] A. Benso, S. Chiusano, P. Prinetto, *A COTS Wrapping Toolkit for Fault Tolerant Applications under Windows NT*, International On-Line Test Workshop (IOLTW 2000), Majorca (ES), July 2000, pp. 9-16

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

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

[5] K. Wilken, J.P. Shen, *Continuous Signature Monitoring: Low-Cost Concurrent-Detection of Processor Control errors*, IEEE Transaction on Computer Aided Design, vol. 9, No. 6, pp. 629-641, June 1990

[6] H. Madeira, J.G. Silva, *On-line Signature Leraning and Checking*, Dependable Computing for Critical Applications 2, Springer-Verlag, pp. 395-420, 1992

[7] D.J. LU, *Watchdog Processor and Structural Integrity Checking*, IEEE Transaction on Computers, vol. C-31, No. 7, pp. 681-685, July 1982

[8] J.H. Patel et al., *Concurrent Error Detection in ALUs by Recomputing with Shifted Operands*, IEEE Transaction on Computers, vol. C-31, No. 7, pp. 589-595, July 1982

[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, November 1-3, 1999 - Albuquerque, New Mexico, USA, pp. 210-218

[10] Y.M. Hsu et al., *Time redundancy for error detecting neural networks*, Proc. IEEE Int. Conf. on Wafer Scale Integration, pp. 111-121, Jan. 1995

[11] 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

[12] 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

[13] M. Zenha Rela, H. Madeira, J. G. Silva, *Experimental Evaluation of the Fail-Silent Behavior in Programs with Consistency Checks*, Proc. FTCS-26, Sendaj (J), 1996, pp. 394-403

[14] 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

[15] A. Benso, S. Di Carlo, G. Di Natale, P. Prinetto, *SEU Effect Analysis in a Open-Source Router via a Distributed Fault Injection Environment*, Design Automation and Test in Europe, Munich (D), March 2001, Accepted for publication

[16] http://www.research.ibm.com/journal/rd/ziegl/

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

[18] Linux Web Site: http://www.linux.org/

[19] M. Rebaudengo, M. Sonza Reorda, M. Torchiano, M. Violante, *An experimental evaluation of the effectiveness of automatic rule-based transformations for safety-critical applications*, DFT'00: IEEE International Symposium on Defect and Fault Tolerance in VLSI Systems, 2000 pp. 257 –265.

Duplicated Variables	0%	10%	20%	30%	40%	50%	60%	70%	80%	90%	100%
Non-Critical Variables	76.3%	82.8%	90.9%	95.3%	97.7%	98.7%	99.0%	99.3%	99.6%	99.8%	100%
Critical Variables	2.3%	1.6%	1.2%	0.8%	0.6%	0.5%	0.4%	0.3%	0.2%	0.1%	0.0%
Very critical Variables	21.4%	15.7%	8.0%	3.9%	1.7%	0.8%	0.6%	0.4%	0.3%	0.1%	0.0%

Table 2: Percentage of critical variables with respect to the duplicated variables

Duplicated Variables	**10%**	**20%**	**30%**	**40%**	**50%**	**60%**	**70%**	**80%**	**90%**	**100%**
Non-Critical Variables	3.5%	5.45%	11%	18.6%	27.6%	37.3%	47%	56.7%	66.5%	76.3%
Critical Variables	0.7%	1.15%	1.47%	1.7%	1.8%	1.9%	2%	2.1%	2.2%	2.3%
Very critical Variables	5.7%	13.4%	17.5%	19.6%	20.6%	20.8%	21%	21.1%	21.3%	21.4%

Table 3: Duplicated variables composition