Soft-error Detection through Software Fault-Tolerance techniques

Maurizio REBAUDENGO, Matteo SONZA REORDA,
Marco TORCHIANO, Massimo VIOLANTE
Politecnico di Torino, Dip, Automatica e Informatica
[reba, sonza, torchiano, violante]@polito.it

Abstract

The paper describes a systematic approach for automatically introducing data and code redundancy into an existing program written using a high-level language. The transformations aim at making the program able to detect most of the soft-errors affecting data and code, independently of the Error Detection Mechanisms (EDMs) possibly implemented by the hardware. Since the transformations can be automatically applied as a pre-compilation phase, the programmer is freed from the cost and responsibility of introducing suitable EDMs in its code. Preliminary experimental results are reported, showing the fault coverage obtained by the method, as well as some figures concerning the slow-down and code size increase it causes.

1. Introduction

The increasing popularity of low-cost safety-critical computer-based applications in several areas such as automotive or medical devices asks for the availability of new methods for designing dependable systems.

In particular, in the new areas where computer-based dependable systems are currently being introduced the cost (and hence the design and development time) is a major concern, and the adoption of commercial hardware is a common practice. As a result, software fault tolerance is often adopted, since it allows the implementation of dependable systems without incurring in the high costs coming from designing custom hardware or using hardware redundancy. On the other side, relying on software techniques for obtaining dependability often means accepting some overhead in terms of increased size of code and reduced performance.

Finally, when adopting a software approach for building a dependable system the designer needs some simple way for both writing the code and verifying whether the whole system has the dependability properties he wishes.

The term *software fault tolerance* has been traditionally used for different purposes [1]: in this paper we refer to it as a way for facing the consequences of hardware errors, in particular those originating from transient faults caused for example by small particles hitting the circuit [2]. We do not consider the issue of eliminating software bugs: we assume that the code is correct, and the faulty behavior is only due to transient faults affecting the system.

In [3] it has been shown that it is possible to achieve a high degree of safe behavior in ordinary computers by complementing the intrinsic Error Detection Mechanisms (EDMs) of the system (exceptions, memory protection, etc.) with a set of carefully chosen software error detection techniques. These techniques include *Algorithm Based Fault Tolerance* (*ABFT*) [4], *Assertions* [3], and *Control Flow Checking* [5], *procedure duplication* [1] and *automatic transformations.*

ABFT is a very effective approach but lacks of generality. It is well suited for applications using regular structures, and therefore its applicability is valid for a limited set of problems [3].

The use of *Assertions*, i.e., logic statements inserted at different points in the program that reflect invariant relationships between the variables of the program can lead to different problems, since assertions are not transparent to the programmer and their effectiveness largely depends on the nature of the application and on the programmer's ability.

The basic idea of *Control Flow checking* is to partition the application program in *basic blocks*, i.e., branch-free parts of code. For each block a deterministic signature is computed and faults can be detected by comparing the run-time signature with a pre-computed one. In most control-flow checking techniques one of the main problems is to tune the test granularity that should be used.

Considering the *Procedure Duplication*, the programmer decides to duplicate the most critical procedures and to compare the obtained results. This approach requires that the programmer define a set of procedures to be duplicated and introduces the proper checks on the results. These code modifications can be executed only manually and may introduce errors.

In this paper we propose an approach based on introducing data and code redundancy according to a set of transformations to be performed on the high-level source code. The transformed code is able to detect errors affecting both data and code: the former goal is achieved by duplicating each variable and adding consistency checks after every read operation. Other transformations focus on errors affecting the code, and correspond from one side to duplicating the code implementing each operation, and from the other to adding checks for verifying the consistency of the executed operations.

The main advantage of the method lies in the fact that it can be automatically applied to a high-level source code, thus freeing the programmer from the burden of guaranteeing its robustness against errors (e.g., by selecting what to duplicate and where to put the checks). The method is completely independent on the underlying hardware, and it possibly complements other already existing error detection mechanisms. Finally, the method is able to detect a wide range of faults, and is not limited to a specific fault model (e.g., single bit-flip, permanent stuck-at, multiple faults, etc.); rather, it addresses any kind of fault affecting either the code or the data.

The approach can be easily complemented with some higher-level recovery mechanism able to restart the program from a safe state and repeat the execution with correct data.

The preliminary experimental results we report show that the method is able to detect an extremely high percentage of faults, at the cost of an increase in the code size of about 2, and of a slow-down factor of about 5.

Section 2 describes the proposed transformation rules (by also providing examples of their application to C programs) and discusses the level of fault detection they guarantee. Section 3 outlines the experiments we performed for practically assessing the feasibility and effectiveness of the approach. Section 4 draws some conclusions.

2. Transformation Rules

In this section we will propose 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 data and code. We do not make any strict assumption neither on the cause nor on the type of the fault: without any loss of generality, we can assume that an error corresponds to one or more bits whose value is erroneously changed while they are stored in memory, cache, or register, or transmitted on a bus. Our method, although devised for transient faults, is also able to detect most permanent faults possibly existing in the system.

All the transformations we propose, being performed on the high-level code, are independent on the host processor that executes the program, as well as on the system

organization (e.g., presence of caches, disks, memory size, etc.). Nevertheless, the optimization flags of the compiler used to produce the executable code software have to be disabled in order to maintain the introduced data and code redundancy through the compilation process.

For the purpose of this paper, we will consider programs written in C, and propose rules to transform the basic constructs of a C program: the extension to the whole language, as well as to other high-level languages is mostly straightforward.

2.1. Errors in data

Some of the rules concern the variables defined and used by the program. We refer to high-level code, only, and we do not care whether they are stored in the main memory, in a cache, or in a processor register. The proposed rules complement other Error Detection Mechanisms that can possibly exist in the system (e.g., based on parity bits or on error correction codes stored in memory). It is important to note that the detection capabilities of our rules are much higher, since they address any error affecting the data, without any limitation on the number of modified bits or on the physical location of the bits themselves.

The basic rules can be formulated as follows:

- Rule #1: every variable *x* must be duplicated: let *x1* and *x2* be the names of the two copies
- Rule #2: every write operation performed on *x* must be performed on *x1* and *x2*
- Rule #3: after each read operation on *x*, the two copies *x1* and *x2* must be checked for consistency, and an error detection procedure should be activated if an inconsistency is detected.

The above rules mean that any variable *v* must be split in two copies v_0 and v_1 that should always store the same value. A consistency check on v_0 and v_1 must be performed each time the variable is read. The check must be performed immediately after the read operation in order to block the fault effect propagation. Please note that variables should be checked also when they appears in any *expression* used as a condition for branches or loops, thus allowing a detection of errors that corrupt the correct execution flow of the program. Each instruction that writes variable *v* must also be duplicated in order to update the two copies of the variable.

Every fault that occurs in any variable during the program execution can be detected as soon as the variable is the source operand of an instruction, i.e., when the variable is read, thus resulting in minimum error latency, which is approximately equal to the distance between the fault occurrence and the first read operation. Errors affecting variables after their last usage are not detected.

Two simple examples are reported in Figure 1, which shows the code modification for an *assignment* operation and for a *sum* operation involving three variables *a*, *b* and *c*.

Original code	*Modified Code*

`a = b;`	`a0 = b0;`
	`a1 = b1;`
	`if (b0 != b1)`
	`  error();`

`a = b + c;`	`a0 = b0 + c0;`
	`a1 = b1 + c1;`
	`if ((b0!=b1) || (c0!=c1))`
	`  error();`

Fig. 1: Code modification for errors affecting data.

The parameters passed to a procedure, as well as the returned values, should be considered as variables. Therefore, the rules defined above can be extended as follows:

- every procedure parameter is duplicated
- each time the procedure reads a parameter, it checks the two copies for consistency
- the return value is also duplicated (in C, this means that the addresses of the two copies are passed as parameters to the called procedure).

Fig. 2 reports an example of application of Rules #1 to #3 to the parameters of a procedure.

Original code	*Modified code*

`res = search (a);`
`…`
`int search (int p)`
`{ int q;`
`  …`
`  q = p + 1;`
`  …`
`  return(1);`
`}`
	`search(a0, a1, &res0, &res1);`
	`…`
	`void search (int p0,int p1,int *r0,int *r1)`
	`{ int q0, q1;`
	`  …`
	`  q0 = p0 + 1;`
	`  q1 = p1 + 1;`
	`  if (p0 != p1)`
	`  error();`
	`  …`
	`  *r0 = 1;`
	`  *r1 = 1;`
	`  return;`
	`}`

Fig. 2: Code transformation for errors affecting procedure parameters.

2.2. Errors in the code

In this subsection we address errors affecting the code of instructions (no matter whether these are stored in memory, in cache, in the processor Instruction Register, or elsewhere). Several processors have built-in EDMs able to detect part of these errors, e.g., by activating Illegal Instruction Exception procedures. Other faults can be detected by software checks (implementing non-systematic additional EDMs) introduced by the programmer. We propose a set of transformation rules to make the code able to detect most of the faults not detected by the other EDMs.

For the purpose of this paper, statements can be divided in two types:

- type S1: statements affecting data, only (e.g., assignments, arithmetic expression computations, etc.)
- type S2: statements affecting the execution flow (e.g., tests, loops, procedure calls and returns, etc.).

On the other side, errors affecting the code can be divided in two types, depending on the way they transform the statement whose code is modified:

- type E1: errors changing the operation to be performed by the statement, without changing the code execution flow (e.g., by changing an *add* operation into a *sub*)
- type E2: errors changing the execution flow (e.g., by transforming an *add* operation into a *jump* or vice versa).

A representation of the possible transformations caused by errors is reported in Fig. 3.

2.2.1. E1 errors affecting S1 statements

As far as errors of type E1 affecting the statements of type S1 are considered, they are automatically detected by simply applying the transformation rules introduced above for errors affecting data. For example, if we consider a statement executing an addition

between two operands, Rule #2 and #3 also guarantee the detection of any error of type E1 which transforms the addition into another operation.

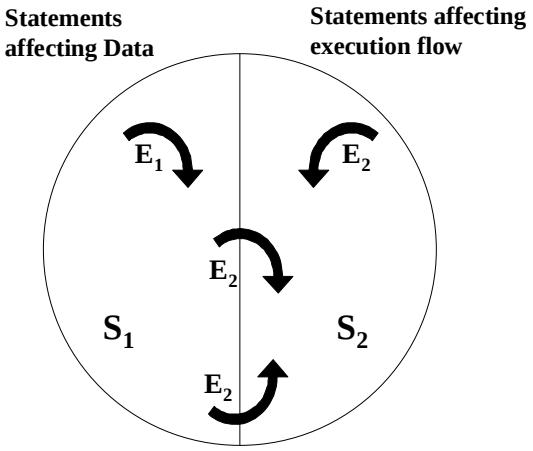

Fig. 3: Classification of the effects of the errors.

2.2.2. E2 errors affecting S1 statements

When an error of type E2 affects a statement of type S1 (e.g., the error transforms an addition operation into a jump), the proposed solution is based on tracking the execution flow, trying to detect differences with respect to the correct behavior. This task is performed by first identifying all the *basic blocks* composing the code. A basic block is a sequence of statements which are always indivisibly executed (i.e., they are branch-free). The following rules are then introduced, in order to check whether all the statements in every basic block are executed in sequence:

- Rule #4: an integer value k_i is associated with every basic block i in the code
- Rule #5: a global execution check flag (**`ecf`**) variable is defined; a statement assigning to **`ecf`** the value of k_i is introduced at the very beginning of every basic block i; a test on the value of **`ecf`** is also introduced at the end of the basic block.

The aim of the above rules is to check whether any error happened, whose effect is to modify the correct execution flow, and to introduce a jump to an incorrect target address. An example of this situation is an error modifying the field containing the target address in a jump instruction. As a further example, consider an error that changes an ALU instruction (e.g., an add) into a branch one: if the instruction format includes an immediate field, this may possibly be interpreted as a target address.

Unfortunately, the above rules have an incomplete detection capability: there are some faults, which can not be detected by the proposed rules, e.g., any error producing a jump to the first assembly instruction of a basic block (the one assigning to **`ecf`** the value corresponding to the block) and any erroneous jump into the same basic block.

Figure 4 provides an example of application of rules #4 and #5.

2.2.3. Errors affecting S2 statements

When errors affecting S2 statements are considered, the issue is how to verify that the correct execution flow is followed.

In order to detect errors affecting a test statement, we introduce the following rule:

- Rule #6: For every test statement the test is repeated at the beginning of the target basic block of both the true and (possible) false clause. If the two versions of the test (the original and the newly introduced) produce different results, an error is signaled.

Figure 5 provides an example of application of the above rule. In order to simplify the presentation of each rule, we do not consider in the examples the combined application of different rules: as an example, in Figure 5 we did not apply Rule #1 and #2 to the variable named **`condition`**, which should be duplicated and checked for consistency after the test.

Modified Code	*Original Code*

`/* basic block beginning */`
`…`
`/* basic block end */`
	`/* basic block beginning #371 */`
	`ecf = 371;`
	`…`
	`if (ecf != 371)`
	`  error();`
	`/* basic block end */`

Fig. 4: Example of code transformation for E2 errors affecting S1 statements.

Original code	*Modified Code*

`if (condition)`
`{/* Block A */`
`  …`
`}`
`else`
`{/* Block B */`
`  …`
`}`
	`if (condition)`
	`{/* Block A */`
	`  if(!condition)`
	`  error();`
	`  …`
	`}`
	`else`
	`{/* Block B */`
	`  if(condition)`
	`  error();`
	`  …`
	`}`

Fig. 5: Code transformation for a test statement.

The code modification for the other S2 statements can be obtained starting from the solution proposed for the test statement.

Special attention has to be devoted to procedure call and return statements. In order to detect possible errors affecting these statements, we devised the following rules:

- Rule #7: an integer value k_j is associated with any procedure j in the code
- Rule #8: immediately before every **`return`** statement of the procedure, the value k_j is assigned to **`ecf`**; a test on the value of **`ecf`** is also introduced after any call to the procedure.

Fig. 6 shows the code modification for the procedure call and return statements. As for the previous Figure, we just applied Rules #7 and #8 to the considered piece of code, ignoring the other previously defined rules.

Rules #7 and #8 allow the detection of a number of errors, including the following ones:

- errors causing a jump into the procedure code
- errors causing a jump to the statement following the call statement
- errors affecting the target address of the **`call`** instruction
- errors affecting the register (or stack location) storing the procedure return address.

Original code	*Modified Code*

…
ret = my_proc(a);
/* procedure call */
…
/* procedure definition */
int my_proc(int a)
{
/* procedure body */
…
return(0);
}
	…
	/*call of procedure #790 */
	ret = my_proc(a);
	if(ecf != 790)
	error();
	…
	/* procedure definition */
	int my_proc(int a)
	{
	/* procedure body */
	…
	ecf = 790;
	return (0);
	}

Fig. 6: Code transformation for the procedure call and return statements (transformations for parameter passing are not shown).

3. Experimental results

In order to practically evaluate the feasibility and effectiveness of the proposed approach, we set up some experiments, which are described in the following.

We first selected a set of simple C programs to be used as benchmarks. We then applied the proposed approach by manually modifying their source code according to the previously introduced rules. Finally, we performed a set of fault injection experiments able to assess the detection capabilities of the resulting system, composed of a given hardware running the modified software.

3.3. Benchmark programs

The following simple programs have been adopted as benchmarks in the current preliminary phase:

- *Bubble Sort*: an implementation of the bubble sort algorithm, run on a vector of 10 integer elements
- *Matrix*: multiplication of two matrices composed of 10x10 integer values
- *Parser*: a syntactical analyzer for arithmetic expressions written in ASCII format. The program also implements a simple software Error Detection Mechanism, which consists in verifying the correctness of each part of the expression.

Tab. 1 reports the ratio between the size of the source code for all the benchmarks after and before the application of the proposed transformations (in terms of number of lines), as well as the ratio between the size of the transformed and original executable code (in terms of bytes) for a Motorola 68040 processor. The adopted compiler is the SingleStep™ 7.4 by SDS, Inc. All compiler optimizations have been disabled when compiling the modified code. The average increase in the size of the executable code is less than 2.

Table 1 reports also the effects of the transformations on the program execution speed. An average slow-down of about 5 times is observed. Times have been computed on the Motorola 68040 system described in the following sub-section.

3.4. Fault Injection environment

The environment we exploited for our Fault Injection experiments is the one described in [6]. The environment is built around an application board hosting a 25 MHz Motorola

68040 processor, 2 MBytes of RAM memory, and some peripheral devices. Fault Injection is performed exploiting an ad hoc hardware device which allows monitoring the program execution and triggering a fault injection procedure when a given point is reached. For the purpose of the experiments, the adopted fault model is the single-bit flip into memory locations. Faults are randomly generated.

	Source code size increase	Executable code size increase	Performance slow-down
Bubble	9.77	1.85	6.77
Matrix	6.92	1.92	3.31
Parser	5.81	1.94	3.25
Average	7.50	1.90	4.44

Tab. 1: Effects of proposed transformations

3.5. Fault Injection Results

By exploiting the Fault Injection environment described in the previous sub-section we evaluated the error detection capabilities of the code after the proposed transformations have been performed. The performed fault injection experiments allowed us to tune the rules by extending them to several fault types that were not originally considered in our analysis.

On the other side, the experiments allowed us to experimentally evaluate the percentage of faults injected in the memory area storing the code, that were not detected using our approach due to its known limitations.

Table 1 and 2 report the results obtained by injecting 4,000 randomly generated faults into each of the above described benchmark programs (2,000 in the memory area containing data, and 2,000 in the memory area containing the code). Faults have been classified as *Fail Silent* (i.e., they did not produce any difference in the program behavior), *HW-detected*, i.e., detected by a hardware EDM (e.g., microprocessor exceptions), *SW-detected*, i.e., detected by the error procedure activated according to the proposed transformation rules, and *Fail Silent Violations* (i.e., they have not been detected by any EDM, and do produce a different behavior). The results show that the size of the last category of faults is really small, and limited to faults injected in the code area, while the percentage of faults detected by the software EDM implemented by the proposed rules is significant. The high number of faults belonging to the Fail Silent category is mainly composed of faults injected in an instruction after its last execution, or in a data variable before writing in it, or after the last read operation from it.

The obtained results show that all the faults injected in the memory containing the data are detected by EDMs or do not cause any effect in the program behavior. Moreover, the only faults that cause a fail silent violation belong to the small category of those affecting a jump (whose target is into the same basic block of the source) or correspond to those faults causing a faulty jump to the first assembly instruction of a basic block (see paragraph 2.2.2).

	Total	Fail Silent	HW-detected	SW-detected	Fail Silent Viol.
Bubble	2,000	943	284	772	1
Matrix	2,000	841	403	754	2
Parser	2,000	1,028	276	694	1
Average	2,000	937	321	740	1

Tab. 2: Fault injection results for faults in the code area.

	Total	Fail Silent	HW-detected	SW-detected	Fail Silent Viol.
Bubble	2,000	196	20	1,784	0
Matrix	2,000	20	1	1,979	0
Parser	2,000	261	0	1,739	0
Average	2,000	159	7	1834	0

Tab. 3: Fault injection results in the data area.

Although we do not report here any result about latency, it is worth noting that our method guarantees that any error in data is detected as soon as the affected variable is read, while nearly any error affecting the code is detected as soon as the affected instruction is executed.

4. Conclusions

We proposed an approach for automatically transforming programs written in any high-level language so that they can be able to detect most of the errors affecting data and code. The proposed transformation rules are suitable to be automatically implemented into a compiler as a pre-processing phase, thus becoming completely transparent to the programmer. This means reducing the cost for developing safe programs, and increasing the confidence in the obtained safety level. Theoretical analysis and preliminary experimental results obtained through fault injection show that the rules are able to reach a very high degree of coverage of the faults which can possibly happen in a microprocessor-based system. On the other side, according to the data gathered on simple benchmarks, the application of the method increases the code size by an average factor of 2, and slow-down its performance by a factor of 5. However, in most safety-critical systems only a limited portion of the code must be fault tolerant, while other parts are not crucial for the correct behavior of the whole system; therefore, the slow-down and code size increase factors related to the whole system are generally lower.

Our method is particularly suited for safety-critical applications implemented by low-cost embedded systems; in these systems the only software often corresponds to a simple application code (no Operating System is needed), while memory availability and execution speed are not a concern. On the other side, a method for automatically transforming the software while guaranteeing the safety of the obtained system is very attractive, since it is compatible with the low costs and the dependability constraints.

We are currently extending our experiments to larger benchmarks, while completing the set of rules to cover all the constructs of a high-level language. We are also working towards the definition of a new set of rules, allowing to reduce the resulting overhead (in terms of memory and speed) at the cost of slightly reduced fault coverage capability.

5. References

[1] D.K. Pradhan, *Fault-Tolerant Computer System Design*, Prentice Hall PTR, 1996

[2] *Scaling Deeper to Submicron: On-line Testing to the Rescue*, Panel Session, IEEE International Test Conf., 1998, pp. 1140

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

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

[5] S. Yau, F. Chen, *An Approach to Concurrent Control Flow Checking*, IEEE Transactions on Software Engineering, Vol. SE-6, No. 2, March 1980, pp. 126-137

[6] A. Benso, P.L. Civera, M. Rebaudengo, M. Sonza Reorda, "*An integrated HW and SW Fault Injection environment for real-time systems*," Proc. IEEE International Symposium on Defect and Fault Tolerance in VLSI Systems, 1998, pp. 117-122