On the Emulation of Software Faults by Software Fault Injection

Henrique Madeira, Diamantino Costa
Centro de Informática e Sistemas
University of Coimbra - Portugal
[henrique, dino]@dei.uc.pt

Marco Vieira
Instituto Superior de Engenharia de Coimbra
Coimbra - Portugal
mvieira@isec.pt

Abstract

This paper presents an experimental study on the emulation of software faults by fault injection. In a first experiment, a set of real software faults has been compared with faults injected by a SWIFI tool (Xception) to evaluate the accuracy of the injected faults. Results revealed the limitations of Xception (and other SWIFI tools) in the emulation of different classes of software faults (about 44% of the software faults cannot be emulated). The use of field data about real faults was discussed and software metrics were suggested as an alternative to guide the injection process when field data is not available. In a second experiment, a set of rules for the injection of errors meant to emulate classes of software faults was evaluated. The fault triggers used seem to be the cause for the observed strong impact of the faults in the target system and in the program results. The results also show the influence in the fault emulation of aspects such as code size, complexity of data structures, and recursive versus sequential execution

1. Introduction

Fault injection has been extensively used in the last decade to evaluate fault tolerance mechanisms and to assess the impact of faults in systems [1, 2]. A major issue is to assure that the injected faults are representative of actual faults, as this is a necessary condition to obtain meaningful results. It is widely accepted that existing fault injection technologies can emulate hardware faults, either transient or permanent. However, the emulation of software faults is still a rather obscure step.

Software faults are recognized as the major cause of system outages. Existing studies show a clear predominance of software faults [3, 4], and given the huge complexity of today's software the weight of software faults tends to increase, which makes clear the relevance of extending the fault injection technologies to the injection of this kind of faults. Among the different techniques, Software Implemented Fault Injection (SWIFI) is clearly the fault injection technique most used today, which makes it the logical choice for this work.

The possible emulation of software faults by fault injection greatly increases the relevance and usefulness of fault injection in the validation of fault tolerance mechanisms and in the evaluation of the dependability features of computer systems. Furthermore, having accurate ways to emulate software faults it is possible to assess the consequences of hidden bugs in a system (experimental risk assessment). This aspect is particularly relevant as it is well known that even the most comprehensive testing process cannot assure that the complex software products are free of bugs.

Fault injection literature is rich in works aiming to study hardware faults. However, only few studies have addressed the problem of injection of software faults. This can be explained by the fact that the knowledge on the software faults experienced by systems in the field is very limited, which makes the definition of meaningful sets of faults (or errors) to inject rather difficult.

The key issue surrounding the injection of software faults is the accuracy of the emulation of this class of faults. Recent studies [5, 6] have proposed the use of field data on discovered software faults to devise a set of rules to generate errors that once injected in a system emulate different types of software faults. However, the accuracy of the errors injected according to these rules has never been evaluated in practice.

To the best of our knowledge, no previous work has evaluated the accuracy of injected faults/errors against actual software faults. Thus we decided to undertake this step and compare the effects of the real software faults with the faults/errors injected by a SWIFI tool (the Xception). In short, the contributions of this study are:

– Evaluation of the capabilities of SWIFI tools in what regards the demands of the emulation of software faults. This has been done by evaluating the use of a typical fault injection tool in two different ways:
  a) Emulation of specific (and real) software faults found in programs. In this case the accuracy of the emulated faults is evaluated against a set of

[1] Work partially supported by FCT, Praxis XXI, contract No. 2/2.1/TIT/1570/95 and grant No. BD/5636/95.

real software faults obtained from a large set of program implementations developed independently by participants in a programming contest;

b) Injection of faults generated by rules similar to the ones proposed in [5].

- Evaluation of the influence in the software fault emulation of aspects such as code size, complexity of data structures, and recursive versus sequential code;
- Proposal of the use of software metrics to guide the fault injection process when field data on actual software faults is not available.

The next section presents related research. Section 3 presents a general discussion of the problem of software faults emulation. The setup used in this study is presented in section 4. Section 5 presents the results of emulation of a set of real software faults by fault injection, and in section 6 the emulation of general classes of software faults is evaluated and discussed. Section 7 concludes the paper.

2. Related Work

The study of software faults has been mainly related to the software development phase, as the software faults are originated during the different steps of this phase (requirement definition, specification, design, coding, testing, etc). This is an important area of software engineering and many studies have contributed to the improvement of the software development methodologies, with particular emphasis on software testing, software reliability modeling and software reliability risk analysis [7, 8].

The fault history during the development phase, the operational profile, and other process measures have been used in software reliability models to estimate the reliability of software and to predict software faults for risk assessment [8, 9, 10].

The study of the software reliability during the operational phase is substantially different from the software under development. The operational environment and the software maturity are different during the operational phase, and the software reliability should be studied in the context of the whole system. The difficulties are not only in the instrumentation required to collect data on software faults but also in the fact that the software faults must be analyzed taking into account the system architecture and not only software modules. Maybe these difficulties account for the fact that the number of works on software faults during the operational phase is lower than the studies available for the development phase. Nevertheless, the study of the effects of actual software faults in the field is of utmost importance for our work, as this is just the kind of faults we want to emulate by fault injection.

The software dependability of Tandem systems are studied in [3, 4] and the impact of software defects on the availability of a large IBM system is presented in [11].

An important contribution to promote the collection and study of observed faults is the Orthogonal Defect Classification (ODC) [12]. ODC is a classification schema for software faults (i.e., defects) in which defects are classified into non-overlapping attributes and used as a source of information to understand and improve the software product and the software development process.

Several fault injection techniques have been proposed [13] and many fault injection studies have been published. However, only few studies have addressed the problem of injection of software faults [14, 15]. In spite of the fact that the issue of how accurately the injected faults emulate real software faults remains largely unknown, fault injection has been used with success in several research works where software faults are the most relevant class of faults. Examples of software weaknesses revealed by faults injected at random can be found in [16, 17].

Mutation testing is a specific form of fault injection that consists of creating different versions of a program by making small syntactic changes [18]. Mutation can be considered as a static fault injection technique, as the source code is changed instead of the program/system's state, as happens in classical fault injection. Mutation has been largely used for software testing. An experimental comparison of the errors and failure modes generated by actual software faults and mutations is presented in [19].

In [20] software faults such as corruption of instructions at machine code-level and specific programming errors have been inserted at random in the code to improve the design of a reliable write-back file cache.

To the best of our knowledge, only three studies focus on the specific problem of the accurate emulation of software faults by fault injection so far [5, 6, 21]. The first two studies propose the same basic set of rules for the generation of errors that emulate software faults. These rules are obtained from the analysis of field data about discovered software faults that have been classified using ODC. In [21] an experimental comparison between fault and error injection is presented and the evaluation is mainly directed towards the comparison of the costs of each approach. These papers are seminal papers somehow, as they give a contribution to the solution of the problem of emulation of software faults by fault injection, but they raise several open questions at the same time:

- If the errors generated by using the proposed rules were injected in a real system would they really emulate the expected software faults?
- Are existing SWIFI tools appropriate to inject the errors for the emulation of software faults? If not, is it possible to enhance these tools to achieve this goal?
- Is the correct emulation of software faults dependent on specific features of the code such as code size, complexity of data structures, recursive versus sequential code, and other features?

- Is there any possibility of turning away the need for field data on actual software faults to generate representative sets of faults?

Answering the above questions is the motivation behind the present work. Although the definite answers to some of those questions clearly need more comprehensive studies, the results presented in this paper reveal significant aspects of the problem.

3. Software fault emulation using SWIFI

The definition of software faults requires the notion of correctness of software. In a broad view, the correctness of software should be measured taking into account the end user/customer needs. However, the needs and degree of satisfaction of the user are too vague to be useful for our purposes. Typically, the development of a software product comprises the requirements, specification, design, code development, test, and product deployment. For the purpose of this work, it is assumed that the requirements and specification are correct (but the code is not correct).

A software fault can be characterized by the change in the code that is necessary to correct it. This is the notion of defect proposed in ODC [22]. In ODC a trigger and a type characterize a defect (fault). The trigger describes the general conditions that make the fault to be exposed and the type represents the fault in the source code. The following fault types in ODC are directly related to the code:

Assignment - values assigned incorrectly or not assigned;

Checking - missing or incorrect validation of data or incorrect loop or conditional statements;

Interface - errors in the interaction among components, modules, device drivers, call statements, etc;

Timing/serialization – missing or incorrect serialization of shared resources;

Algorithm - incorrect or missing implementation that can be fixed by (re)implementing an algorithm or data structure without the need for a design change;

Function - incorrect or missing implementation of a capability that affects a substantial amount of code and requires a formal design change to be corrected.

The classes of triggers defined in ODC are associated to common activities of the development process: review/inspection, function test, and system test. Only the system test class of triggers is relevant for our study, as it represents the broad environmental conditions when the faults are exposed during the operational use in the field. These general conditions (triggers) are startup/restart, workload volume/stress, recovery/exception, hardware/ software configuration, and normal mode. The normal mode category means that the software fault has been exposed when everything was supposed to work normally. This is the trigger category relevant for our study as all the experiments have been done with the target system working in normal conditions (i.e., it was not at startup, etc).

As mentioned above, a software fault is characterized by the necessary change in the source code to correct it. As the source code is normally written in a high-level language and the typical SWIFI tools inject faults at the machine code-level, it means that the fault classification and the fault emulation by fault injection are done at different abstraction levels (see figure 1).

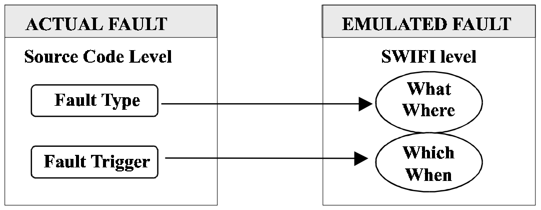


Figure 1 – Fault classification at source code level and fault emulation at SWIFI level.

In a typical SWIFI tool faults are defined according to three main classes of parameters: what (what should be changed/corrupted), where (where, in the code, should the change be applied), when (when, during the program execution, should the change be inserted). The traditional When parameter should, in our opinion, be decomposed in which (which instruction or event acts as fault trigger) and when (when, during the various executions of the trigger instruction or trigger event is the fault injected).

ODC fault types have a clear translation into the what and where fault injection parameters. However, ODC fault triggers cannot be used to define the SWIFI fault triggers because ODC triggers just represent general environmental conditions in which the faults should be injected.

The probability of a software fault resulting into a failure is heavily dependent on the operational profile. Assuming a fault exists, the probability of the faulty code to be executed is p1 (see figure 2). If the faulty code is executed, the probability of error generation is p2. If errors are generated, the probability of these errors resulting into a failure is p3. Thus, the probability of a software fault resulting into a failure is the product of p1, p2, and p3.

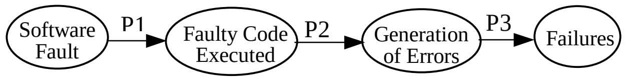


Figure 2 – Probability of a software fault exposure.

Ideally, the fault trigger should reproduce the chain reaction in figure 2. However, the need of accelerating the process suggests that errors should be injected instead of faults (p1 = p2 = 1) which leads us to the paramount question of the representativeness of the injected errors.

The error representativeness can be regarded in two different perspectives:

- *Representativeness concerning fault type:* The injected errors are considered software errors if they could have been caused by a real software fault of any type. That is, what is required is to avoid the injection of errors specific of other kinds of faults such as hardware faults.

- *Representativeness concerning fault trigger:* The injected errors should emulate the errors that would have been caused if the faulty code had been executed with the input data required to generate errors.

Software faults are then emulated by injecting errors (using SWIFI tools) defined in terms of the parameters that describe the What, Where, Which, and When attributes. The exact parameters depend on the SWIFI tool and the target system. As a general indication, the fault type is described by the attributes What and Where and the fault trigger is described by Which and When.

4. Experimental setup

4.1. Target system and the Xception fault injector

The target system is a Parsytec PowerXplorer with four PowerPC 601 processors running under the Parix, which a Unix like operating system for parallel machines. The system has no special fault tolerance mechanisms and is relatively simple, which makes it a good choice for the present experiments. The version of Xception used is targeted for the PowerPC 601, and the experiments are controlled by a host computer (a Sun/Solaris machine).

The Xception is described in detail in [23]. However, a very brief description is provided here to facilitate the discussion on the software fault emulation in the next sections. Xception uses the debugging and performance monitoring features existing in most of the modern processors to inject faults by software and to monitor the activation of the faults and their impact on the target system behavior. Faults are injected with minimum interference for the target application. The target application is not modified, no software traps are inserted, and it is not necessary to execute the target application in trace mode.

Xception provides a comprehensive set of fault triggers, including spatial and temporal fault triggers, and triggers related to the manipulation of data in memory. Faults injected by Xception can affect any process running on the target system and it is possible to inject faults in applications for which the source code is not available.

Xception also includes the Experiment Management software, which runs in the host system and is responsible for the fault definition, experiment execution control, outcome collection, and some preliminary results analysis (detailed analysis is performed off-line using MS Excel).

4.2. Sample programs

The programs used in this study are of two different types: several implementations of two programs resulting from the International Olympiads in Informatics (IOI) from ACM International Collegiate Programming Contest [24] and a program actually used in real life. The two contest programs are Camelot and JamesB and the other program is SOR. These programs represent different degrees of program complexity and different size and all of them are written in C.

Camelot – This program computes the minimum number of moves required to gather all the pieces of a chessboard in the same square. Only two kinds of pieces are considered: one king and a variable number of knights ranging from 0 to 63 knights. The size of the different versions of this program (made by the different teams) ranges from 200 to 360 lines of code.

JamesB – This program codifies strings according to a specific algorithm. A seed received as a parameter with each string determines the actual codification. The result is the coded version of the original string. The size of the available versions of this program is about 100 code lines.

SOR – This program is an implementation of a parallel algorithm to solve the Laplace equation over a grid. The algorithm is based on the over-relaxation scheme with red-black ordering. This program has near 2400 lines of code. The result is given in the form of a matrix.

5. Emulation of actual software faults

The goal of this section is to evaluate the possibilities of a SWIFI tool (the Xception) to emulate real software faults. To do that we need to have access to programs with known real software faults. The programs resulting from the IOI programming contest provide an easy source of software faults. Several factors make these programs an interesting source of software faults for this research:

- All the programs are written according to a formal, clear, and correct problem specification;
- The programs were written by skilled programmers;
- The contest gives us rapid access to several implementations from the same specification (237 teams in the IOI 98 contest). These alternative programs normally use different design strategies and algorithms, which makes it possible to compare the influence of different program control and data structures in the failure modes caused by the software faults;
- There is a test case (i.e., a set of data inputs and correct results) associated to each problem specification, which works as an acceptance criteria for correct programs from the contest judges' point of view. Only bugs found in programs that passed in the test cases were considered as representative of real faults.

The search for software faults among the programs produced in the contest was done following these steps:

1. The programs considered correct in the contest were selected (these programs passed the contest test case);

2. Selected programs were intensively tested by using a very thorough test case. This was achieved by running the programs a huge number of times with random input data sets. These tests may run for hours. Programs failing this intensive test have software faults;

Program	% Wrong results	% Correct results
C.team1	7.3%	92.7%
C.team2	16.9%	83.1%
C.team3	1.0%	99.0%
C.team4	30.8%	69.2%
C.team5	2.9%	97.1%
JB.team6	0.05%	99.95%
JB.team7	1.8%	98.2%

Table 1 – Failure symptoms of the real software faults.

3. Programs with software faults were then analyzed in detail to identify the fault. The input data that exposed the fault was used to help the bug identification.

Seven software faults have been identified in seven different programs. Five software faults were in the implementations of Camelot and two faults in the JamesB. The software faults have been analyzed and classified according to the following fault types (to simplify the identification Camelot programs are named as C.team# and JamesB named as JB.team#):

Assignment – 2 faults (JB.team6, C.team4);

Checking – 1 fault (C.team1);

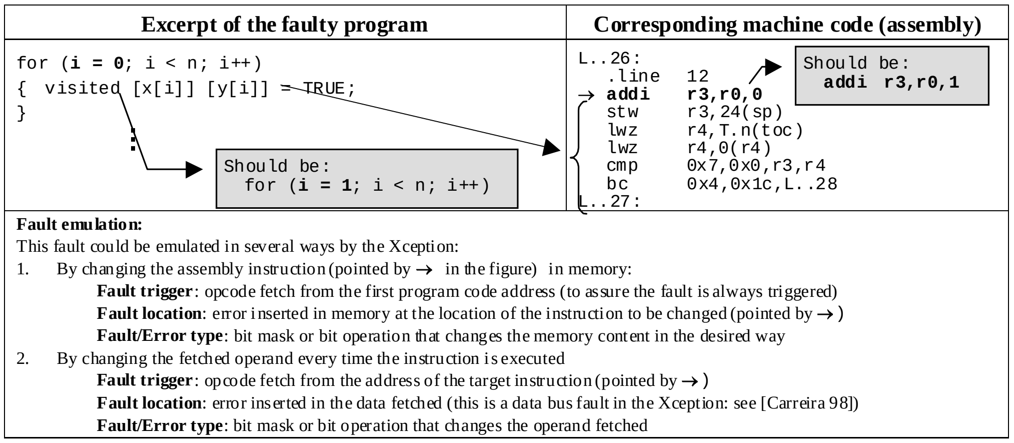
Algorithm – 4 faults (JB.team7, C.team2, C.team3, C.team5).

The failure symptoms due to the software faults observed in the programs are presented in table 1. These results were obtained from the intensive tests and correspond to more than 10.000 runs for each program. Other failure modes such as program hangs or system crashes have not been observed in any of the programs.

One first comment is that the test case used in the programming contest is not very effective, as C.team2 and C.team4 failed quite often and that was not detected by the contest test case. Nevertheless, from more than one hundred of correct (according to the contest test case) programs submitted to the intensive test, only the seven programs above failed.

To evaluate the possibility of accurate emulation of the actual software faults by using the Xception each fault was analyzed in order to determine the adequate Xception fault trigger and fault models. For the algorithm faults we concluded that the accurate emulation by the Xception (or any other machine code-level SWIFI tool) is simply not possible (this will be discussed further on). However, assignment and checking faults could in fact be emulated. In general, one fault can be emulated in several ways, using different possibilities of fault trigger and fault models.

To compare the fault emulation done by Xception with the actual faults, the correct version of each program (i.e. the version obtained after removing the bug) was executed



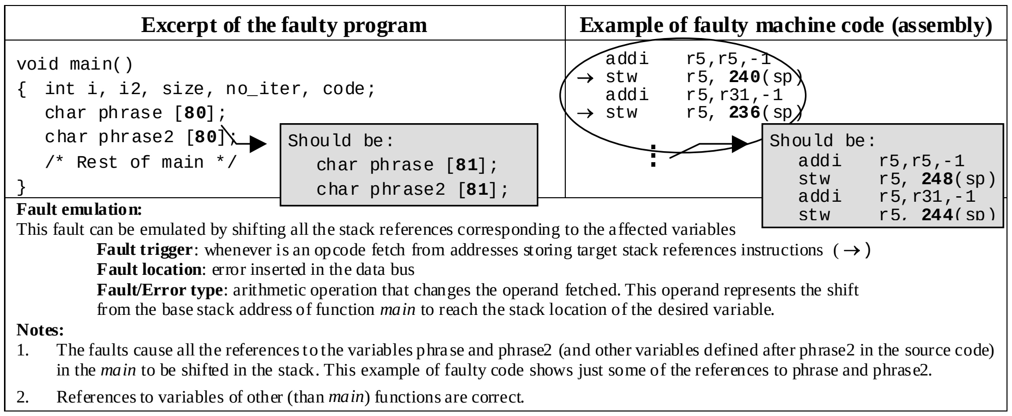
Excerpt of the faulty program	Corresponding machine code (assembly)

`for (i = 0; i < n; i++)`
`{  visited [x[i]] [y[i]] = TRUE;`
`}`
Should be:
`for (i = 1; i < n; i++)`
	`L..26:`
	`.line   12`
	`→ addi   r3,r0,0`
	`stw    r3,24(sp)`
	`lwz    r4,T.n(toc)`
	`lwz    r4,0(r4)`
	`cmp    0x7,0x0,r3,r4`
	`bc     0x4,0x1c,L..28`
	`L..27:`
	Should be:
	`addi  r3,r0,1`

Fault emulation:
This fault could be emulated in several ways by the Xception:
1. By changing the assembly instruction (pointed by → in the figure) in memory:
   Fault trigger: opcode fetch from the first program code address (to assure the fault is always triggered)
   Fault location: error inserted in memory at the location of the instruction to be changed (pointed by →)
   Fault/Error type: bit mask or bit operation that changes the memory content in the desired way
2. By changing the fetched operand every time the instruction is executed
   Fault trigger: opcode fetch from the address of the target instruction (pointed by →)
   Fault location: error inserted in the data fetched (this is a data bus fault in the Xception: see [Carreira 98])
   Fault/Error type: bit mask or bit operation that changes the operand fetched

Figure 3 – Example of an assignment fault in the program C.team4.

with the same test case used in the faulty version. However, a fault was injected in each run to emulate the effects of the bug. If the results are the same in both runs it means Xception do emulate the fault accurately.



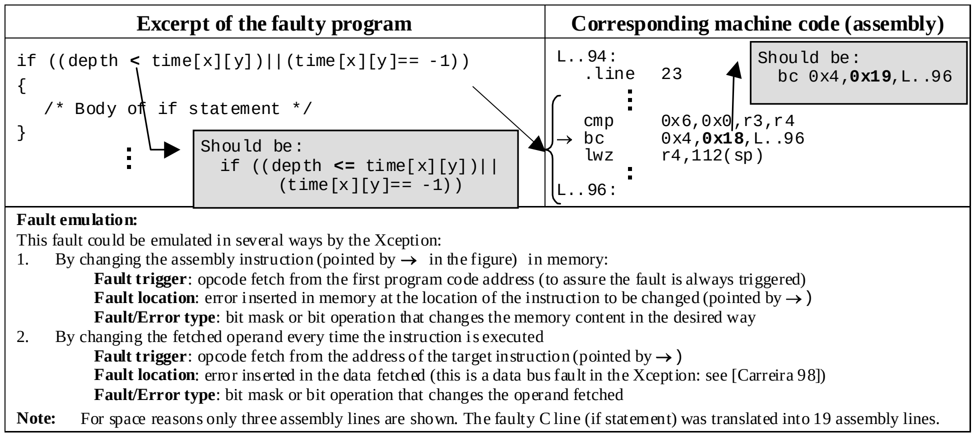
Excerpt of the faulty program	Example of faulty machine code (assembly)

`void main()`
`{  int i, i2, size, no_iter, code;`
`   char phrase [80];`
`   char phrase2 [80];`
`   /* Rest of main */`
`}`
Should be:
`char phrase [81];`
`char phrase2 [81];`
	`addi   r5,r5,-1`
	`→ stw    r5, 240(sp)`
	`addi   r5,r31,-1`
	`→ stw    r5, 236(sp)`
	Should be:
	`addi   r5,r5,-1`
	`stw    r5, 248(sp)`
	`addi   r5,r31,-1`
	`stw    r5, 244(sp)`

Fault emulation:
This fault can be emulated by shifting all the stack references corresponding to the affected variables
Fault trigger: whenever is an opcode fetch from addresses storing target stack references instructions (→)
Fault location: error inserted in the data bus
Fault/Error type: arithmetic operation that changes the operand fetched. This operand represents the shift from the base stack address of function *main* to reach the stack location of the desired variable.

Notes:
1. The faults cause all the references to the variables phrase and phrase2 (and other variables defined after phrase2 in the source code) in the *main* to be shifted in the stack. This example of faulty code shows just some of the references to phrase and phrase2.
2. References to variables of other (than *main*) functions are correct.

Figure 4 – Example of an assignment fault in the program JB.team6.



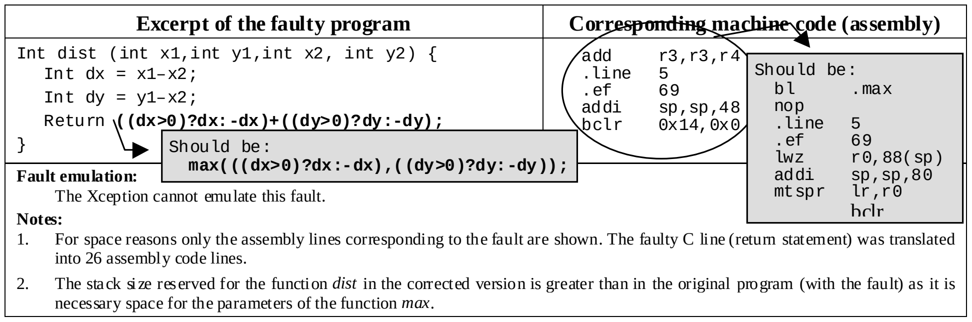
Excerpt of the faulty program	Corresponding machine code (assembly)
`if ((depth < time[x][y])||(time[x][y]== -1))` `{` `/* Body of if statement */` `}` ⋮ Should be: `if ((depth <= time[x][y])|| (time[x][y]== -1))`	`L..94:` `.line 23` ⋮ `cmp 0x6,0x0,r3,r4` → `bc 0x4,0x18,L..96` `lwz r4,112(sp)` ⋮ `L..96:` Should be: `bc 0x4,0x19,L..96`

Fault emulation:
This fault could be emulated in several ways by the Xception:
1. By changing the assembly instruction (pointed by → in the figure) in memory:
   Fault trigger: opcode fetch from the first program code address (to assure the fault is always triggered)
   Fault location: error inserted in memory at the location of the instruction to be changed (pointed by →)
   Fault/Error type: bit mask or bit operation that changes the memory content in the desired way
2. By changing the fetched operand every time the instruction is executed
   Fault trigger: opcode fetch from the address of the target instruction (pointed by →)
   Fault location: error inserted in the data fetched (this is a data bus fault in the Xception: see [Carreira 98])
   Fault/Error type: bit mask or bit operation that changes the operand fetched

Note: For space reasons only three assembly lines are shown. The faulty C line (if statement) was translated into 19 assembly lines.

Figure 5 – Example of a checking fault in the program C.team1.

For a better understanding of the way the discovered software faults have been emulated let us examine some of these faults in detail (see figures 3, 4, 5, and 6).



Excerpt of the faulty program	Corresponding machine code (assembly)
`Int dist (int x1,int y1,int x2, int y2) {` `Int dx = x1–x2;` `Int dy = y1–x2;` `Return ((dx>0)?dx:-dx)+((dy>0)?dy:-dy);` `}` Should be: `max(((dx>0)?dx:-dx),((dy>0)?dy:-dy));`	`add r3,r3,r4` `.line 5` `.ef 69` `addi sp,sp,48` `bclr 0x14,0x0` Should be: `bl .max` `nop` `.line 5` `.ef 69` `lwz r0,88(sp)` `addi sp,sp,80` `mtspr lr,r0` `bclr`

Fault emulation:
The Xception cannot emulate this fault.

Notes:
1. For space reasons only the assembly lines corresponding to the fault are shown. The faulty C line (return statement) was translated into 26 assembly code lines.
2. The stack size reserved for the function *dist* in the corrected version is greater than in the original program (with the fault) as it is necessary space for the parameters of the function *max*.

Figure 6 – Example of an algorithm fault in the program C.team5.

The emulation of specific faults by the Xception requires manual intervention for determining specific locations in memory to set fault triggers or to insert errors. The loader provides this information. Another manual task is the definition of the right mask and bit level operation to insert the desired error.

The Xception could not entirely emulate the assignment fault shown in figure 4. The reason is in the fact that the fault trigger used (opcode fetch from a specified address) is implemented by using the processor breakpoint registers, which are only two in the PowerPC. Using the traditional SWIFI approach of inserting trap instructions to trigger the faults could solve this, but this technique is very intrusive. Another relevant aspect concerning the emulation of this fault is that it requires large manual intervention. Extra software tools to assist the definition of this kind of faults (assignment faults causing shifts in the stack) are required to make the process usable.

Figure 6 shows an example of an algorithm fault. In general algorithm faults cause considerable changes at the machine code-level and cannot be emulated by SWIFI tools. In some cases, algorithm faults can be decomposed in assignment and/or checking faults. For example, the fault in figure 6 corresponds to an incorrect assignment to the return parameter of function dist (it is assigned the sum of two values instead of the larger value). However, the emulation of algorithm faults by "equivalent" assignment/checking faults is very doubtful.

The ODC types function, interface, and timing/ serialization were not analyzed, as we didn't find software faults of these types in the used programs. However, considering the definitions of these types of faults we would say that function faults suffer the same problems as algorithm faults and cannot be accurately emulated by SWIFI tools. Interface faults are somehow similar to assignment faults (wrong assignments at the modules and function interface) and some of them can be emulated. The accurate emulation of timing/serialization faults is heavily dependent on the specific fault.

The results of this experiment can be summarized as follows:

A. Assignment and checking faults can (in general) be accurately emulated by Xception (or SWIFI tools);

B. The present version of Xception cannot emulate some assignment faults (faults affecting stack shifts) or the emulation requires high manual intervention. However, it is possible to define new Xception features to facilitate the accurate emulating of this set of faults (tools for assisting the fault definition, new fault triggers, and fault types);

C. A third set of faults cannot be emulated by the Xception and their emulation by SWIFI tools doesn't seem feasible. Algorithm faults and function faults fall in this category. This set of faults represents the limits of the Xception model (and SWIFI tools models) in emulating real software faults. Considered the field data results published in [5] these kind of faults (algorithm and function) accounts for nearly 44% of the software faults.

6. Emulation of classes of software faults

Much more important than the emulation of a specific software fault by fault injection is the emulation of faults of the same class. This goal requires two steps:

1. Definition of a set of rules for the generation of errors that emulate an entire class of faults (instead of a specific fault as in section 5);
2. Validation and tuning of these rules by comparing the impact of the errors injected using the rules with available data on the impact (failure modes) of real faults of the same class. Additionally, if the injected errors emulate the same class of software faults it is expected (at least as a hypothesis) to identify some patterns in terms of impact in the target system.

The rules proposed in [5] were used starting point for the experiments in this section and the experimental setup is basically the same as described in section 4.

6.1. What can be done with no field data

Even more ambitious than the injection of classes of software faults is the objective of injecting faults/errors that emulate the classes of faults most likely to affect a given program. The use of field data to guide the injection process presumes that it is possible to emulate classes of software faults by fault injection, which is not obvious. Furthermore, the need of having field data on previous faults to emulate software faults represents a strong limitation. This is due to the following reasons:

- Field data on previous software faults is not available in most cases;
- Field data is specific of the application/system in which it has been produced;
- When available it means that the product has already been delivered and has been being used for a long time, which means that the relevance of injecting faults for fault removal or fault forecast in that particular product is doubtful;
- Normally a software fault is registered only one time (when the fault is corrected), no matter the number of users that have experienced a failure due to that particular defect. This means that relevant information about the frequency of failures caused by a defect is normally not registered.

These limitations show the interest of investigating the possibilities of emulating software faults without relying on field data. The analysis of how the field data is used in [5] shows that field data is only used for two purposes: a) to distribute the injected errors by the software components and b) to choose the most common type of errors. All the other parameters are generated at random.

Existing studies [10, 9, 25] indicate that fault probability correlates with the software module complexity. This suggests that existing metrics (and tools) to predict the probability of a given module having software faults could be used when field data is not available. These metrics are based on static or dynamical aspects of the code and, in some cases, the metrics also include information from the development process. Software metrics can be used to choose the modules to inject faults or to decide on the number of faults to inject in each module. Obviously, the number of faults to inject in each module and the type of error can also be chosen at random, which means that all the possible software faults and locations are equally likely.

6.2. Programs, data sets, and result collection

A large set of programs has been used to evaluate the influence of aspects such as different algorithms for the same problem, code size, complexity of data structures, recursive versus non-recursive execution, and parallel versus sequential execution. Naturally, the programs used in section 5 in which we found real software faults of type assignment and checking were selected. The following table shows all the programs and the main reasons (features) why they have been selected.

Programs	Features
C.team1, C.team10	Recursive algorithms, about 280 lines, 1 real faults (corrected)
C.team2, C.team8	Non-recursive algorithms, about 250 lines
C.team9	Non-recursive algorithm, use many dynamic structures, about 250 lines
JB.team6	Non-recursive algorithms, 1 real fault (corrected), about 100 lines
JB.team11	Non-recursive algorithms (≠ from JB.team6), about 100 lines
SOR	Parallel program, real life program, larger size: 2400 lines

Table 2 – Target programs and main features.

A test case composed by 300 input data sets randomly generated has been used for all the programs of the same kind. That is, all the injections in all the Camelot (C.team#) programs, for example, used the same test case. In this way it is possible to compare the results of all the injections in the same program and the results of all the injections in different programs of the same type. Each test case corresponds to 300 runs of the program (each specific input data set) and one fault and the target system is rebooted between injections to assure a clean state.

The results collected are the failure modes (symptoms). The following failure modes are considered:

Correct results – Program terminated normally and the output is correct;

Incorrect results – Program terminated normally but the output is incorrect;

Program hang – The program hangs (possibly went into a dead loop) and was terminated by the experiment manager software after a timeout;

Program crash – The program terminated abnormally and generated errors detected by the system (incorrect instructions, etc).

6.3. The parameters what, where, which, when

The sets of errors to inject are described in terms of the parameters what, where, when, and which. These errors are injected at machine code-level (Xception level) and represent the actual data or instruction corruption injected. Obviously, we need to define a subset of error types to make the experiments tractable, as otherwise the number of faults to inject would be nearly infinite. Table 3 describes the error types in high-level language terms.

Type of fault	Correct	Change	Correct	Change	Correc	Changed	Correct	Change
Checking	>=	>	=	≠	[i]	[i-1]	[i]	[i+1]
					Only for checking over arrays			
	>	>=	≠	≠				
	<=	<	=	>=	<	<=	and	or
	true	false	false	true	=	<=	or	and
Assignment	value	value+	value	value-1	value	unassigned	value	random

Table 3 – Subset of injected error types.

Once the list of used error type is selected, the error parameters are generated in the following way, for each class of faults:

1. All possible fault locations were identified. This was done manually at the assembly level. To assist this process, the assignment and checking statements in the source code were first identified and the compiler facilities in terms of symbol tables and labels were used to help the identification of the assembly instructions corresponding to the assignment and checking statements;
2. Some fault locations were chosen at random from the list of all possible fault locations. These were the locations where the errors were injected (**where** parameter);
3. For each location all possible error types (among the subset previously defined) that can be applied were selected (**what** parameter). For example, in an assignment location four different faults are generated, one for each error type (The number of error types from table 3 that can be applied to each fault location depends on the actual instruction. This is particularly true for the checking error types);
4. The instructions selected to work as trigger for the injection were the same instructions selected as location to inject the fault (**which** parameter);
5. The fault was inserted every time the trigger instruction was executed (when parameter).

Table 4 shows the result of applying the above criteria to the used programs. A total number of 108.600 faults were injected. Each program run corresponds to one fault, no matter the number of times the fault is triggered during the program run.

6.4. Results and discussion

The failure mode results (total faults) are presented in figure 7 and figure 8 for assignment and checking faults respectively. Several aspects should be noted:

- The injected faults have a much stronger impact than typical software faults (only a relatively small percentage of the faults stay dormant and have not affected the system);
- One aspect not represented in the charts (figures 7 and 9) is that when the result of the programs is correct the faulty code (i.e., code affected by the error injected) has been executed. Thus, the reasons why the error generated did not affect the results are related to the input data sets;
- There are no clear patterns in the failure mode results when all the faults of the same type are considered

Program	Assignment			Checking		
	Possible locations	Chosen locations	Injected faults (all error types)	Possible locations	Chosen locations	Injected faults (all error types)
C.team1	92	8	9600	49	8	4800
C.team2	63	5	6000	45	6	7800
C.team8	84	8	9300	31	9	3300
C.team9	87	9	10800	53	9	3300
C.team10	88	9	10800	43	8	4200
JB.team6	29	5	6000	10	5	3300
JB.team11	21	5	5700	11	5	2100
SOR	363	12	14400	195	12	7200

Table 4 – Injected faults.

- The influence of specific features of the programs (code size, algorithm, recursive, etc) is not evident in the global results. However, for the assignment faults

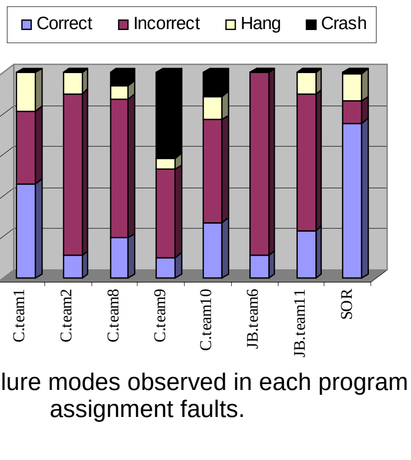


Figure 7 – Failure modes observed in each program for assignment faults.

injected in recursive programs (C.team1 and C.team10) and in the large SOR program a higher percentage of correct results was obtained;

- The program C.team9, which intensively uses dynamic structures (i.e., structures such as linked lists that use intensively memory allocation/free functions), has a high percentage of crashes;
- The SOR program, which is a parallel program, seems to be quite sensitive to checking faults, as a large percentage caused crashes have been observed.
- The percentages of crashes and program hangs observed for the JamesB programs are very low. A possible explanation is in the fact that these programs are small (around 100 C code lines) and very simple, when compared to the other programs.

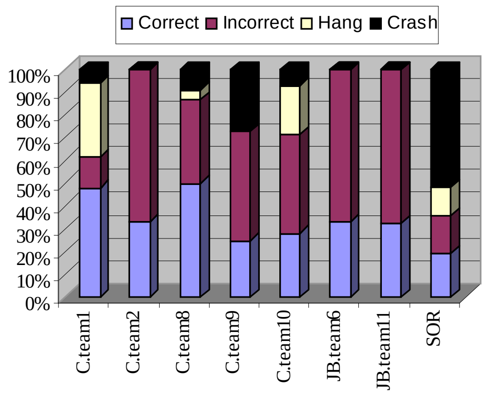


Figure 8 – Failure modes observed in each program for checking faults.

Figure 9 and figure 10 show the failure modes for each error type (What parameter). In figure 10 the error types are represented by a pair of symbols in which the first represents the correct checking and the second the result of the corruption (e.g., <= < means that the error caused a <= checking to be changed into a <). While the results for each error type for the emulation of assignment faults are relatively similar, the same does not apply to the error types used to emulate checking faults. The percentage of correct results in some error types is very low or even null. For example, when the checking assignment is changed from != to = or from true to false the percentage of correct values is very low. On the other hand, when the error injected turns a < into a <= the percentage of correct values is much higher.

Although the results presented in figure 10 are expected, we cannot immediately conclude that some of the error types do not emulate software errors. We should remember that the other parameters (where, which, and when) play an important role as they represent the fault locations and the fault triggers.

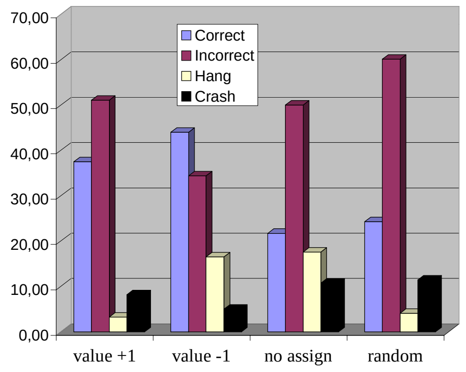


Figure 9 – Failure modes observed for assignment faults (all faults).

The accuracy of the injections should be analyzed in the two perspectives discussed in section 3: fault type and fault trigger. It is easy to understand that random fault triggers (which and when) are not representative of real software faults. Faults injected in this way represent naïve software faults similar to the ones easily found in ad-hoc testing. A central question seems to be the independent evaluation of the accuracy of the fault types (what; where) and the fault triggers (which; when).

Another interesting aspect is that the injected errors also emulate hardware faults, which might explain the general small percentage of correct results. In fact, as mentioned in section 3, it is very difficult or even impossible to prevent injected errors from emulating software and hardware faults at the same time. The random fault trigger used is also typical from hardware faults. Thus, the failure modes observed have the contribution of the hardware faults that are also emulated by the injected errors. In fact, previous experiments using Xception [23] or using hardware (pin-level) fault injectors [26] shown that hardware faults can cause failure modes with a large percentage of incorrect results and system crashes.

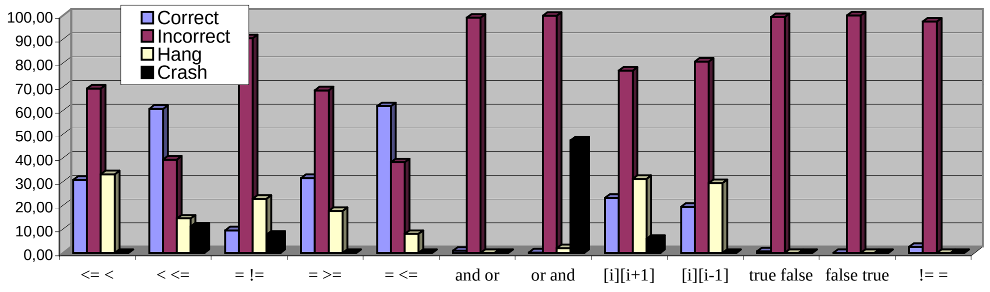


Figure 10 – Failure modes observed for checking faults (all faults).

7. Conclusions

This paper presents an experimental study on the emulation of software faults by fault injection. A set of real software faults found in different programs has been compared with faults injected by the Xception to evaluate the accuracy of the injected faults. The results of this experiment revealed three different kinds of software faults: i) faults that can be accurately emulated by the Xception; ii) faults that could be emulated if Xception was improved with extra fault triggers, fault models, and tools to avoid the need of manual fault definition iii) faults that could never be emulated by Xception (or any other SWIFI tool).

A second issue investigated was the possibility of emulation of all the faults of the same class. The use of field data about real faults was discussed and software metrics were suggested as an alternative to guide the injection process when field data was not available. A set of rules for the injection of errors meant to emulate software faults were evaluated. For each studied class of faults a large number of faults (more than 100.000) were injected according to the set of rules. The impact these faults has been studied in detail and the results obtained with the faults injected to emulate faults of the same class were compared. Results show that the injected faults have a much stronger impact than typical software faults, as only a small percentage of the faults stay dormant and have not affected the system. Another result is that there are no clear patterns in the failure mode results when all the faults of the same type are considered. The analysis of the results obtained for each error type injected reveled significant differences. However, the random fault triggers used seem to be the cause of the strong impact of the faults in the target system and in the program results. The presented results also show the influence in the fault emulation of aspects such as code size, complexity of data structures, and recursive versus sequential execution.

Further research is needed to understand the fault triggers required for the emulation of subtle software faults such as the ones that escape comprehensive test procedures. A promising approach seems to be devising ways to perform an independent evaluation of the accuracy of the fault types and the fault triggers.

8. References

[1] J. Arlat et al, “Fault Injection and Dependability Evaluation of Fault Tolerant Systems”, IEEE Transactions on Computers, vol. 42, no. 8, pp. 919-923, Aug. 1993.

[2] R. K. Iyer, “Experimental Evaluation”, Special Issue FTCS-25 Silver Jubilee, 25th IEEE Symp. on Fault Tolerant Computing, FTCS-25, pp. 115-132, Jun. 1995.

[3] J. Gray, “A Census of Tandem Systems Availability Between 1985 and 1990”, IEEE Transactions on Reliability, vol. 39, no. 4, pp. 409-418, Oct. 1990.

[4] I. Lee and R. K. Iyer, “Software Dependability in the Tandem GUARDIAN System”, IEEE Transactions on Software Engineering, vol. 21, no. 5, pp. 455-467, May 1995.

[5] J. Christmansson and R. Chillarege, “Generation of an Error Set that Emulates Software Faults”, Proc. 26th Fault Tolerant Comp. Symp., FTCS-26, Sendai, Japan, pp. 304-313, Jun. 1996.

[6] J. Christmansson and P. Santhanam, “Error Injection Aimed at Fault Removal in Fault Tolerance Mechanisms – Criteria for Error Selection using Field Data on Software Faults”, Proc. of the 7th IEEE Int. Symp. on Software Reliability Engineering, ISSRE’96, Oct. 30 to Nov. 2, 1996, New York, USA, 1996.

[7] M. R. Lyu, “Handbook of Software Reliability Engineering”, IEEE Comp. Society Press, McGraw-Hill, 1996.

[8] J. Musa, “Software Reliability Engineering”, McGraw-Hill, 1996.

[9] T. Khoshgoftaar et. al., “Process Measures for Predicting Software Quality”, High Assurance Systems Engineering Workshop, HASE’97, Washington D.C., USA, IEEE CS Press, HASE’97, 1997.

[10] J. P. Hudepohl et. al., “EMERALD: A Case Study in Enhancing Software Reliability"” Proc. of IEEE 8th Int. Symp. Soft. Reliability Engineering, ISSRE’98, pp. 85-91, Nov. 1998.

[11] M. Sullivan and R. Chillarege, “Software defects and their impact on systems availability – A study of field failures on operating systems”, Proc. 21st Fault Tolerant Comp. Symp., FTCS-21, pp. 2-9, Jun. 1991.

[12] R. Chillarege, I. S. Bhandari, J. K. Chaar, M. J. Halliday, D. Moebus, B. Ray, M. Wong, “Orthogonal Defect Classification – A Concept for In-Process Measurement”, IEEE Trans. Soft. Eng., vol. 18, no. 11, pp. 943-956, Nov. 1992.

[13] Mei-Chen Hsueh, T. Tsai, R. K. Iyer, “Fault Injection Techniques and Tools”, IEEE Computer, vol. 30, no. 4, pp. 75-82, Apr. 1997.

[14] J. Hudak, B. Suth, D. Siewiorek, and Z. Segall, “Evaluation and Comparison of Fault-Tolerant Software Techniques”, IEEE Trans. on Reliability, vol. 42, no. 2, pp. 190-204, Jun. 1993.

[15] W. Kao, “Experimental Study of Software Dependability”, Ph.D. Thesis, Technical Report CRHC-94-16, Department of Computer Science, Univ. of Illinois at Urbana-Champaign, Illinois, USA, 1994.

[16] J. Voas, et al, “Predicting how Badly ‘Good’ Software can Behave”, IEEE Software, 1997.

[17] D. Blough and T. Torii, “Fault-Injection-Based Testing of Fault-Tolerant Algorithms in Message-Passing Parallel Computers”, 27th IEEE Fault Tolerant Computing Symp., FTCS-27, Seattle, USA, pp. 258-267, Jun. 1997.

[18] R. DeMillo et al, “An Extended Overview of the Mothra Software Testing Environment”, Proc. ACM SIGSOFT/IEEE 2nd Workshop on Soft. Testing, Verification, and Analysis, pp. 142-151, Jul. 1988.

[19] M. Daran and P. Thévenod-Fosse, “Software Error Analysis: A Real Case Study Involving Real Faults and Mutations”, Proc. of 3rd Symp. on Software Testing and Analysis, ISSTA-3, San Diego, USA, pp. 158-171, Jan. 1996.

[20] Wee T. Ng, Peter M. Chen, “Systematic improvement of fault tolerance in the RIO file cache”, Proc. 30th Fault Tolerant Computing Symp., FTCS-29, Madison, WI, USA, Jun. 1999.

[21] J. Christmansson, M. Hiller, and M. Rimén, “An Experimental comparison of Fault and Error Injection”, Proc. 9th IEEE Int. Symp. on Software Reliability Engineering, ISSRE’98, pp. 369-378, USA, 1996.

[22] R. Chillarege, “Orthogonal Defect Classification”, Chapter 9 of “Handbook of Software Reliability Engineering”, Michael R. Lyu Ed., IEEE Computer Society Press, McGrow-Hill, 1995

[23] J. Carreira, H. Madeira, and J. G. Silva, “Xception: Software Fault Injection and Monitoring in Processor Functional Units", IEEE Transactions on Software Engineering, vol. 24, no. 2, Feb. 1998.

[24] Int. Olympiads Informatics, http://olympiads.win.tue.nl/ioi/

[25] S. Benlarbi, K. eman, N. Goel, “Issues in Validating Object-Oriented Metrics for Early Risk Prediction”, Dig. Fast Abst, 10th Int. Symp. on Software Reliability Engineering, ISSRE’99, Boca Raton, Florida, USA, pp. 17-18, Nov. 1999.

[26] H. Madeira and J. G. Silva, "Experimental evaluation of the fail-silent behavior in computers without error masking", Proc. 24th Fault Tolerant Comp. Symp., FTCS 24, Austin, USA, Jun. 1994, pp. 350-359.