

RIFLE: A General Purpose Pin-level Fault Injector

Henrique Madeira, Mário Relá, Francisco Moreira, and João Gabriel Silva

Laboratório de Informática e Sistemas, University of Coimbra
3000 Coimbra, Portugal

Abstract. This paper discusses the problems of pin-level fault injection for dependability validation and presents the architecture of a pin-level fault injector called RIFLE. This system can be adapted to a wide range of target systems and the faults are mainly injected in the processor pins. The injection of the faults is deterministic and can be reproduced if needed. Faults of different nature can be injected and the fault injector is able to detect whether the injected fault has produced an error or not without the requirement of feedback circuits. RIFLE can also detect specific circumstances in which the injected faults do not affect the target system. Sets of faults with specific impact on the target system can be generated. The paper also presents fault injection results showing the coverage and latency achieved with a set of simple behavior based error detection mechanisms. It is shown that up to 72,5% of the errors can be detected with fairly simple mechanisms. Furthermore, for over 90% of the faults the target system has behaved according to the fail-silent model, which suggests that a traditional computer equipped with simple error detection mechanisms is relatively close to a fail-silent computer.

1. Introduction

The validation of fault-tolerance mechanisms is a difficult task. The mechanisms involved in the fault activation and in the error propagation process are highly complex, which greatly complicates the task of analytical modeling. Experimental evaluation by fault injection has become an attractive way of validating specific fault-tolerance mechanisms, allowing the estimation of fault tolerant system measures such as fault coverage and error latency [1].

This paper addresses the problem of injecting physical faults in physical models [2], i.e. actual systems implementing the fault-tolerance mechanisms under evaluation. In this group we consider not only prototypes implementing specific hardware and/or software features of the developed systems but also actual systems working very closely to real world conditions. In this way fault injection can effectively complement other kind of techniques used in the development and validation phases (e.g. modeling and simulation), as fault injection can be used in the final implementation of the developed system. This is quite important as previous researches of different natures [3, 4, 5] have emphasized the impact of the workload on the performance of the error handling mechanisms.

Several techniques have been used for physical fault injection. Most of them fall in two main groups: **1)** fault injection at the IC pin-level [6, 7, 8, 9, 10, 1, 11] and **2)** injection of internal faults into the ICs, either by exposure to different sources of radiation [12, 3] or by interference with the IC power supply [10, 13].

Methods based on the injection of internal IC faults have the inherent advantage of causing actual faults in the ICs, which may be close to a realistic fault model.

A simple method for injecting internal faults in ICs is to interfere with the chip power supply lines [10, 13]. However, it is quite difficult to control the extension of the errors caused inside the chip with this method, and some more sensitive chip areas might be responsible for the majority of the errors.

Fault injection by heavy-ion radiation is another way of injecting faults inside the ICs. A fault-injector based on heavy-ion radiation was presented in [3]. However, this method is difficult to apply to existing computers mainly because the target chip outputs have to be compared pin-by-pin with a gold unit in order to know whether the radiation has produced errors inside the target IC or not.

Most of the physical fault injection studies have been performed using pin-level fault injection [6, 7, 8, 9, 10, 1, 11]. Many of these papers concern the evaluation of fault-tolerance mechanisms in specific target systems. A general methodology for physical fault injection has been presented in [1], which also describes the MESSALINE pin-level fault injector.

However, the injection of pin-level faults in the complex microprocessors available today addresses new problems which have not been answered so far. Most of the processors use techniques such as prefetching, internal caches, pipelining, and delayed branches, which greatly difficult (or even make it impossible) the analysis of the results.

The main problem in injecting pin-level faults in complex chips such as the processors is to find out whether the injected fault should be accounted for the results or not. Fig. 1 illustrates the analysis required at the processor instruction execution level behavior after the injection of each processor pin-level fault in order to accomplish this goal.

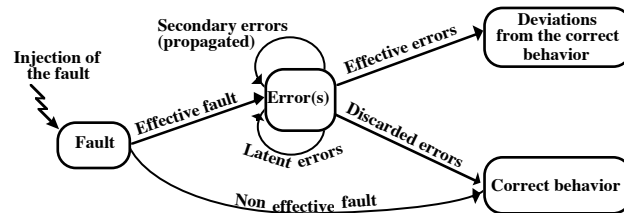


Fig. 1. Impact of a pin-level fault in the target system at the processor instruction level behavior.

The first step is to detect if the injected fault has produced an error or not. It should be noted that not all pin-level faults produce errors. For example, the injection of a stuck-at-high fault in a pin that is already high will not generate errors. After detecting the faults that have caused errors (effective faults) it is necessary to find out if the errors have been discarded as normal consequence of the processor prefetch or pipeline. This analysis depends on the processor features and the actual program execution after the injection of the fault. In most of the cases, only the faults that cause deviations in the target system behavior should be considered in order to obtain correct results.

To the best of our knowledge RIFLE is the only pin-level fault injector capable of performing the above analysis.

2. Abstractions in the Fault Injection Process

Before describing the RIFLE architecture let us present the general framework of the fault injection process. Sets of faults F are generated considering the universe of the experiment U , which is described by three abstractions: the *target system model*, the *benchmarks*, and the *fault model at the pin-level*. The injection of these faults produces *results* and, depending on the fault tolerance technique under evaluation, the adequate *analysis* will be performed on these results. As consequence of the results/analysis new sets of faults or changes in the universe U may be required. Fig. 2 represents these abstractions and their relationship in the fault injection process.

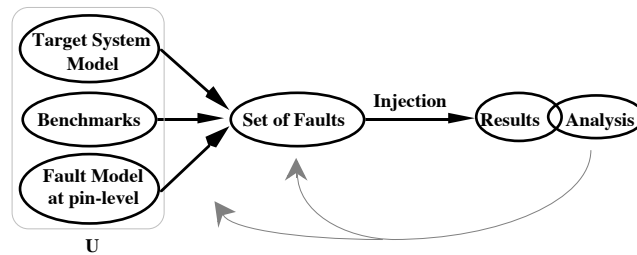


Fig. 2 . Abstractions in the Fault Injection Process

Each fault is defined by a set of attributes such as fault duration, lines affected, type of fault, and specific conditions that trigger the injection of the fault. A set of faults F defined from a given universe U is only valid for this universe. Furthermore, two different faults must be described by two different sets of attributes, in order to achieve repetitive fault injection. In other words, the injection of F in the conditions defined by U always produces the same results. Changes in the universe U , for example changes in the benchmarks, will cause the injection of F to produce different results because the faults F are no longer valid for the new conditions.

2.1 - Target system model from the fault injector point of view

The target system is perceived by RIFLE according to the functional model shown in Fig. 3. Faults can be injected in the IC pins of any module: the processor, the memory, the bus, or other devices. However, in many situations the processor is the most interesting target.

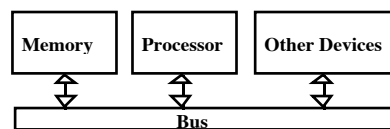


Fig. 3. Target system model as perceived by the fault injector.

The fault injection should be tightly synchronized with the activity of the processor. This is applicable not only to the faults injected in the processor pins but also in cases in which the faults are injected in the pins of other ICs. In this way, a fault is injected when a precise point in the program execution and a specified bit pattern in the target IC are achieved, which allows reproducible faults.

The fault injection is not limited to target systems following this model. For example, in a multiprocessor system the fault injector only considers as target system one processor and the related hardware. This does not seem too restrictive, as in most cases fault-tolerance techniques presume a single system fault model. However, it should be noted that the injection of faults in multiprocessor or multicomputer systems are no longer repetitive, as the fault injector does not take into account the global state of the system.

2.2 -Benchmarks

During the fault injection experiments the programs being executed by the target system are called benchmarks (or workload). Depending on the evaluation goals, these

programs could be the actual application program or "standard" benchmarks. Usually, these benchmarks are small programs meant to be representative of the most common program structures used in actual programs.

Previous works [3, 14] have shown the strong influence of the benchmarks in the performance of the evaluated fault-tolerance techniques. This emphasizes the interest of the fault injection in actual systems running real application programs.

2.3 - Transient fault model at the pin-level

A key aspect in fault injection can be expressed in the following question: which are the pin-level error patterns that are representative of internal IC transient faults?

Transient and intermittent faults are in fact the major physical cause for computer failures [15]. As the objective of the fault injection is to emulate actual faults, or inject faults that cause the same error manifestations as actual faults, it seems obvious that transient faults are the most interesting class of faults.

2.3.1 - The processor as the main target for pin-level fault injection

The choice of the target IC is largely dependent on the evaluation goals. However, in most cases, the processor should be the main target as the system behavior is mainly determined by the processor. In addition to this argument, the following reasons also justify the interest of the fault injection at the processor pins:

- 1) Faults injected in the processor pins can model not only internal processor faults but also memory faults, bus faults, and most of the faults in peripheral devices. For instance, faults in a peripheral device can be emulated by injecting faults in the processor pins during the cycles in which the processor is reading the peripheral device;
- 2) It is possible to cause errors in other parts of the target system by injecting faults in the processor pins. For example, a fault injected in the processor data bus during a memory write cycle will cause an error in the addressed memory cell.

2.3.2 - Fault model at the processor pins

Pin-level manifestations of internal processor faults are very difficult to estimate. In theory, it is possible to determine in advance the faulty pin-level behavior by simulation or by exposing the IC to some source of radiation. However, this is not realistic as simulation models or radiation source facilities are not commonly available.

Few works can be found in the literature addressing the problem of how internal microprocessor faults are manifested at the pins. Several gate-level simulation studies suggested that internal faults cause multiple errors at the output pins [14, 16, 17].

Experiments using radiation and power supply disturbances to inject actual faults inside processors have also provided some insight on the pin-level manifestations of internal faults. The main results published on this topic are summarized in Table 1.

The first thing to be noted is that all these works suggested the single pin error model, which contradicts the results obtained by the simulation works mentioned above. Only a small percentage of faults have caused errors in multiple pins. Another important aspect is the fact that the errors are relatively well distributed among the pins (the greater error percentage in the address pins is because the used processors have more address pins than data and control pins). The small percentage of errors in the data pins result from the fact that these pins are normally asserted as input.

Research	Method	Proc.	Error location				Pins affected		
			Address	Control	Data	Mixed	1 pin	2 pins	Seve.
[3]	Heavy ion	6809	66%	22%	4%	8%	70%	9%	21%
[12]	Heavy ion	Z80	51%	6%	32%	11%	72%	5%	13%
[18]	Heavy ion	6809	64%	27%	5%	4%	72%	10%	18%
[18]	Pow.supply	6809	17%	80%	1%	2%	51%	35%	14%

Table 1. Error manifestation of internal processor faults at the pin-level.

It seems clear that the question of which error patterns can represent internal processor faults cannot be answered by combining the available results from the different papers on processor fault injection experiments. However, from the above analysis the following general rules for processor pin-level error manifestations can be inferred:

- 1) The errors caused by transient internal faults are only visible at the output pins;
- 2) The pin-level fault duration should be one (or few more) bus cycles;
- 3) Both single pin error and multiple pin error models are required.

It is obvious that transient processor faults can produce errors anywhere in the circuit, including the input signals. However, errors can only be observed at the output pins. In this way, the injection of faults in the processor outputs (or in bi-directional pins asserted as output) directly emulates internal processor faults. On the other hand, faults in the processor input pins can emulate not only internal processor faults but also memory, bus, and peripheral faults.

From the point of view of pin-level fault injection, a transient fault is a very short duration fault. Previous researches [9, 11] have shown that the shortest fault should last for one memory cycle. In fact, faults with even shorter duration (e.g. processor clock) cause the same error manifestations of one memory cycle faults or do not cause any error at all.

It should be noted that in many situations it is not necessary to know the exact error distribution at the IC pins. Instead, the fault injector should be able to generate sets of faults with a specific impact on the target system, such as faults causing control flow errors or incorrect memory access. For example, faults causing control flow errors should be used in the verification of a control flow error detection technique.

2.4 - Sets of faults

Set of faults to inject are defined considering the three abstractions mentioned above: the model of the target system, the benchmark running on the target system, and the fault distribution model at the pin-level (Fig. 2). Each fault is described concerning the following aspects:

Fault location	<ul style="list-style-type: none"> - in the target system hardware (in which ICs, pins, etc.) - in the benchmark (in which instruction, memory cycle, etc.) - in time (counted from some event, e.g. the initial reset)
Fault nature	<ul style="list-style-type: none"> - type of fault (stuck-at, bit inversion, etc.) - number of pins affected - duration

2.5 - Results/analysis

The results depend on the fault-tolerance technique(s) under evaluation and the evaluation goals. In most of the cases, physical fault injection give direct measures on the specific technique under evaluation. For example, the coverage and the average latency of an error detection approach can be directly assessed by fault injection.

The fault injection results can be divided in three categories: **1)** binary results (yes or no, corresponding to the occurrence of some event), **2)** counter results (e.g. latency; interval between two events, # occurrences of an event), and **3)** extensive target system behavior results. These results can be used in fault removal and fault forecasting [1].

3. The RIFLE architecture

The leading idea of RIFLE is to combine *trigger and tracing techniques* traditionally used in digital logic analyzers with the logic required for the pin-level fault insertion. The result is a system able to inject practically all types of pin-level faults, and capable of recording extensive information on the target processor (and system) behavior after the injection of each fault. This tracing information is used for the complete characterization of each fault and its impact on the target system (fault effective duration, the type of processor cycle affected, etc.), and for the analysis of the error propagation process. The analysis of the tracing information is automatically performed (after the injection of each fault) by the RIFLE software, which stores the key results in a spreadsheet file. Final statistical results (tables, charts, etc.) are obtained from this file.

The architecture of RIFLE is shown in Fig. 4. It is formed by three hardware modules and the control and management software running in the host computer (a personal computer in the present version):

Adaptation module: this module replaces the target processor in the socket and contains the target processor and the fault injection elements. The faults are injected using the insertion technique and each injection element is just an electronic switch.

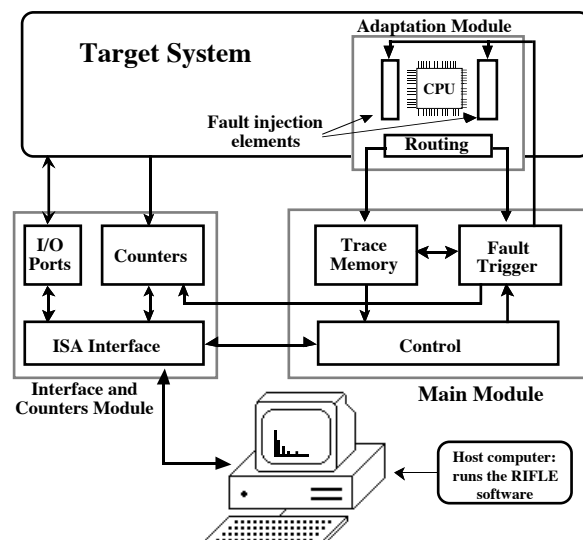


Fig. 4. RIFLE organization.

Main module: receives the target processor bus signals from the adaptation module and contains the fault trigger hardware and the trace memory. The fault trigger hardware receives the parameters describing each fault from the host computer and activates the injection elements when the conditions specified for the injection of the fault are detected in the target processor bus. The trace memory continuously collects the information present in the target processor bus.

Interface and counters module: establishes the interface with the RIFLE host computer and contains the circuits required to gather latency and binary results.

Control and Management Software: the main tasks of the Control and Management Software are: **1)** to assist the experiment set up and the definition of sets of faults, **2)** to control the fault injection sequence, **3)** to validate the faults considering the eventual errors produced, and **4)** to collect the relevant results. In the present implementation, the software was implemented in C++ under Windows 3.1 in order to provide a friendly icon and mouse oriented user interface.

Each fault is described by a unique set of parameters which completely characterize the fault (synchronized fault injection), which means that the injection of the faults is deterministic and can be reproduced if needed. The fault injector is able to detect whether the injected fault has produced an error or not without the requirement of analog feedback circuits (this is achieved by the analysis of the traced information). This feature greatly simplifies the fault injection hardware and avoids the speed restrictions imposed by the feedback circuits [1]. Furthermore, as the Trace Memory takes a snapshot of the target system behavior in the vicinity of the memory cycle(s) in which the fault was injected, a detailed error propagation analysis can be performed. This analysis is carried out by the RIFLE software without user intervention, fulfilling the validation of each injected fault as illustrated in Fig. 1.

The specification of faults to inject can be done manually fault by fault, or automatically by the system, which generates random values for the fault descriptor parameters. In the latter, the user only supplies adequate ranges/restrictions for the fault descriptor parameters. Once a set of faults is generated it is stored in a file. During the fault injection the parameters of each fault are sent to the fault injector hardware that controls the injection of the fault. After the injection of a fault, the RIFLE software waits for a user specified time-out (for fault propagation in the target system) and then reads the content of the Trace Memory in order to validate the fault and collect the results.

Faults are injected by using the insertion technique. In this technique the target IC is removed from the target system and replaced by a piggy-back circuit (the adaptation module in Fig. 4) which contains the target IC and the connection required to intersect the signals at the IC pins. The injection elements were implemented using programmable logic, which means that the actual number of uni-directional and bi-directional injection elements can be adapted to the needs of the target IC.

In normal operation, the injection elements only bypass the signals without introducing errors. When the "Fault" input (Fig. 5) is activated a fault will be injected. The following types of faults can be injected: stuck-at 1, stuck-at 0, stuck-at an external value, inversion, logical bridging at next pin, logical bridging at previous pin, and open circuit. Each injection element has an additional output (Fig. 5) which sends the correct pin state to the Trace Memory. Using this information, and knowing the characteristics of the injected fault, it is possible to find out if the injected fault has produced errors or not (this subject will be detailed later on).

In the present implementation up to 96 bi-directional/uni-directional injection elements can be used, which allows the injection of faults in 96 different points in the target system without changing the set up configuration.

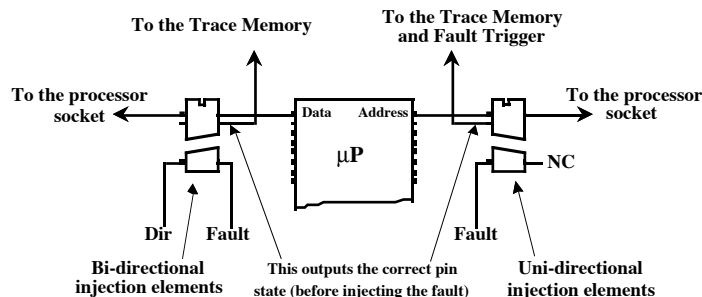


Fig. 5. Fault injection elements.

The Trace Memory is 96 bit width by 4096 depth, which allows the tracing of all 96 injection points. It is implemented with fast memory chips (40 nsec. access time) in order to achieve a high tracing rate. Eight data lines of the Trace Memory can be used optionally for recording the outputs of a counter (time stamps). The trace clock (which depends on the target processor) is generated in the Adaptation Module. The processor bus cycle is normally used as trace clock, because this signal represents the minimum granularity in terms of target processor behavior from a pin-level point of view.

In the RIFLE the injection of faults by forcing technique (in which the faults are applied directly to the IC pins by means of special probes) was not implemented. With this decision the complexity of the fault injector was significantly reduced without greatly affecting its features. In fact, practically all types of faults that can be injected by forcing techniques can also be injected by insertion techniques.

The RIFLE can be used in different target systems. The details concerning one particular target processor/system are grouped into the Adaptation Module and in a file used by the fault injector software. All that is required is to remove the target system processor from its socket and place it in the adaptation module, which is then connected to the processor socket (soldered processors are more complex to handle, but it is still possible to do it).

A RIFLE predecessor (with the same basic architecture) has been extensively used in the evaluation of behavior based error detection techniques [11, 19]. The RIFLE has already been used in computers based on the 68000 and in the Z80 processors. Recently, the adaptation modules for the Intel 486DX processor and the Inmos T800¹ have been finished.

3.1 Fault trigger capabilities

The Fault Trigger module detects the unique conditions defined for the injection of a fault. Each fault is defined by a set of parameters as shown in the table presented in Table 2 (which are a generalization of the fault definition parameters used in [9]).

The size of each parameter is indicated in parenthesis. In the cases of the Activation Address and the Activation Pattern the actual size (which depends on the target system) is conformed by the Adaptation Module.

The injection of the fault is triggered in the following way: the Fault Trigger Module looks for the Activation Address at the address bus of the target system processor. The Activation Address should be detected as many times as the value in

¹ The fault injector for systems based on the T800 is one of the tasks of the FTMPs (Fault-Tolerant Massively Parallel System) ESPRIT III project.

the Activation Address Count. When the specified number of Activation Address is detected the Fault Trigger Module waits a number of memory cycles indicated in Delay. At this point, the Fault Trigger Module will wait until the Activation Pattern is detected. After detecting the Activation Pattern the fault is injected during a time defined in Duration and affecting the lines defined by the Fault Mask.

The Activation Pattern represents a bit pattern in the target system (normally a bit pattern in a random logic) which is of interest to include in the fault trigger conditions. Depending on the Fault Type parameter, the Activation Pattern can be included (as a qualifier) at different points in the fault trigger chain or even be omitted.

<i>Fault parameters</i>	<i>Description</i>
Fault Type	Specifies the fault nature and other attributes
Activation Address (up to 32 bits) Activation Address Count (8 bits) Delay (16 bits) Activation Pattern (up to 16 bits)	These four parameters specify the moment in which the injection of the fault starts
Duration (8 bits)	Specifies the fault duration (e.g. in mem. cycles)
Fault Mask (96 bits)	Specifies the pin(s) where the fault is injected

Table 2. Set of parameters required for the definition of a fault.

The Activation Address Count is useful to specify a point inside a program loop. The value in Activation Address Count represents the number of loops required to inject the fault. The Delay parameter allows the injection of faults during the data read and store cycles. For example, the injection of a fault during the execution of a stack-push-instruction can be achieved by specifying the address of the push instruction as the Activation Address and choosing a Delay equal to the number of bus cycles spent with the instruction fetch.

An innovative feature of RIFLE is the ability of defining specific sets of faults following some given criteria. This is achieved through the combination of the fault trigger capabilities with versatile fault definition software. The ranges/restrictions supplied by the user can work as a filter in the random fault generation process. Some examples of specific sets of faults that can be generated are program space random faults, execution time random faults, control flow faults, and memory behavior faults.

3.2 Tracing capabilities

The recording in the Trace Memory is controlled by the Fault Trigger module. After the injection of a fault the Trace Memory continues the recording until all the 4096 memory positions have been filled. As mentioned above, the Trace Memory records all points in which a fault can be injected (processor data, address, and control lines and, optionally, other IC lines). RIFLE software uses the Trace Memory for three purposes: **1)** error feedback, **2)** detection of discarded errors, and **3)** detailed behavior analysis.

3.2.1 Error Feedback

The injection of a fault may not cause an error. For example, a stuck-at-1 fault injected in a pin that is already at 1 during the time the fault is injected does not cause an error. This effect is highly probable in the injection of very short faults for transient fault emulation and must be accounted for in order to have correct results. With the exception of inversion faults (the logic level at the faulted pin is inverted) in which the

error is guaranteed, all the other types of faults require feedback in order to detect whether the fault has caused an error or not. Previous works avoided this requirement by injecting only inversion faults [9, 11] or proposed feedback circuits [1].

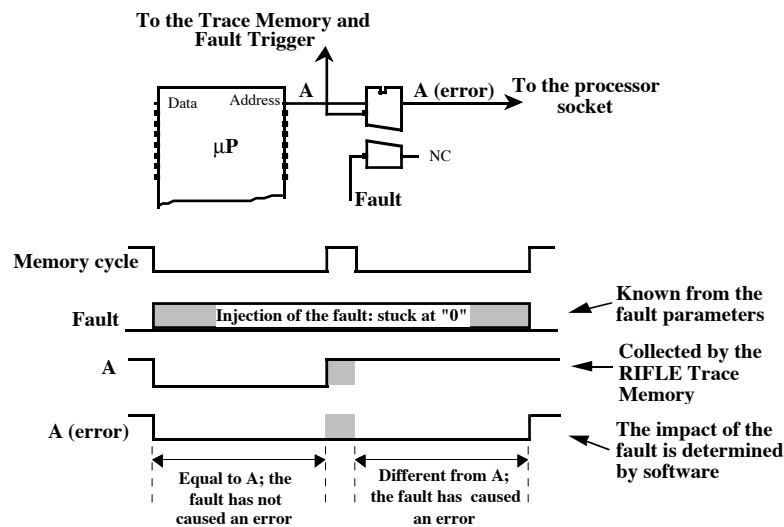


Fig. 6. Error feedback in the RIFLE.

In RIFLE the Trace Memory reads the logic level of the lines before they pass through the Fault Injection elements, i.e. the error free state. Knowing the fault free state of the lines, the effects of the injected fault can be deduced using the following fault parameters: mask, duration, and type. For example, injecting a stuck-at-0 fault during two memory cycles in a line that is at 0 during the first cycle and at 1 during the second cycle means that the fault does not cause an error in the first cycle but it causes an error in the second cycle (see Fig. 6). As we can see, this method not only detects whether the fault has caused an error or not but also determines the effective duration of the fault. This analysis is performed by the RIFLE software in the host computer after the injection of each fault.

It should be noted that the above scheme also works with bi-directional pins. In this case, no matter the direction of the pins, the fault injection elements are programmed in such a way that the Trace Memory always collects the pins error free state.

3.2.2 Detection of discarded errors

Most of the microprocessors available today have internal caches and deep prefetch queues. In this way, the errors that affect the instruction and operand fetch can be discarded when the prefetch queue is discarded or the cache is flushed.

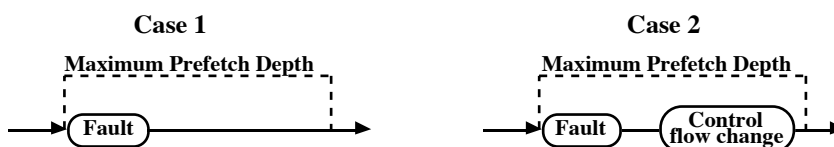


Fig. 7. Analysis of discarded errors due to the prefetch queue.

Discarded errors do not affect the target system and the faults that have produced them should not be considered in most of all fault injection situations. All previous pin-level fault injection researches have ignored this problem.

The problem of the internal cache can be solved by deactivating the cache, which is possible in practically all microprocessors. However, the same cannot be applied to the prefetch queue.

In RIFLE the target system behavior recorded in the Trace Memory is used for the detection of discarded errors. The RIFLE software analyzes the subsequent memory cycles after the injection of each fault considering that the prefetch queue is full (worst case). If no control flow changes are detected in the range of the prefetch queue the errors produced by the fault were not discarded and the fault is tagged as effective (case 1, Fig. 7). If a control flow change is detected (case 2) it is assumed that the errors were discarded and the fault is ignored. In this analysis some effective faults may be tagged as non effective and ignored because it is assumed that the prefetch queue is full (the actual prefetch queue state is unknown). However, the fault injection results are not affected as the problem is not to ignore an effective fault but the opposite, i.e. to consider as effective a fault whose errors have been discarded.

3.2.3 Detailed behavior analysis

In order to take advantage of the trace potential of RIFLE hardware, another software module has been implemented to allow the detailed analysis of the target system behavior after the injection of a fault. This behavior analysis software uses the assembly file generated by the compiler from the benchmark running in the target system, which includes a hexadecimal dump of the assembly instruction codes and the relative address in which each instruction is stored. This file is used as a reference of correct behavior.

The objective is to obtain a tool for the detailed analysis of "odd" faults in order to understand why they escape to the fault/error handling mechanisms. As faults are repetitive in RIFLE, it is guaranteed that the re-injection of the faults will produce the same behavior in the target system.

4. Evaluation of simple behavior based error detection techniques

Computers usually include simple behavior based error detection techniques such as the memory access error detection features delivered by memory management unit, the internal error detection mechanisms included in many processors and a watchdog timer (WDT). Although these error detection techniques are used quite often, it is not clear what percentage of error can be detected by them. Designers have the intuitive idea that these mechanisms can detect errors that cause serious changes in the computer behavior but the actual coverage and latency achieved with them is not known.

This section presents an evaluation of a set of simple concurrent error detection mechanisms with the main objective of showing the comprehensive results that can be obtained with RIFLE. Nevertheless, most of these results are new and bring out some interesting conclusions.

The target system is a Force[®] computer based on the 68000 processor (with the speed reduced to 4 MHz) and the evaluated error detection techniques are:

68K - 68000 processor internal error detection mechanisms.

This processor has several internal error detection mechanisms such as the detection of accesses to non-implemented memory, fetch of invalid instructions, unaligned instruction fetch, and unaligned word access.

MEM - Memory access error detection.

This is a set of error detection mechanisms similar to the memory protection features of a normal memory management unit. The following mechanisms are considered:

AUM - Accesses to unused memory;

ECS - Error in the code segment access (error if it is not an instruction fetch);

EDS - Error in the data segment access (error if it is not a data read or write);

AIM - Accesses to unimplemented memory.

WDT - Watchdog timer

Traditional implementation of WDT by means of a hardware programmable timer.

<i>Criteria used for the definition of sets of faults</i>	<i>Faults</i>	
	<i>Set A</i>	<i>Set B</i>
Fault activation places: chosen at random	1-4 cycles	1 cycle
Faulted pins: all processor pins at random	12500	12500
Number of pins faulted by each fault: 1 pin	753	777
Fault duration: 1 - 4 memory cycles at random	114	411
Type of faults: inversion	11633	11312

Fault generation criteria

Used set of faults

Table 3. Fault generation criteria and sets of faults used in these experiments.

Table 3 summarizes the criteria used in the generation of the faults and shows the two sets of faults used in this evaluation (Set A and Set B). Some faults were not injected because the fault activation parameters correspond to program zones that are never executed with the used program data. Furthermore, the errors generated by some injected faults were discarded by the processor prefetch queue without affecting the system behavior. The number of these faults is greater in set B than in set A because set B only includes faults of one cycle duration. Discarded faults must not be considered and the RIFLE software automatically discarded these faults. Only the effective faults were considered.

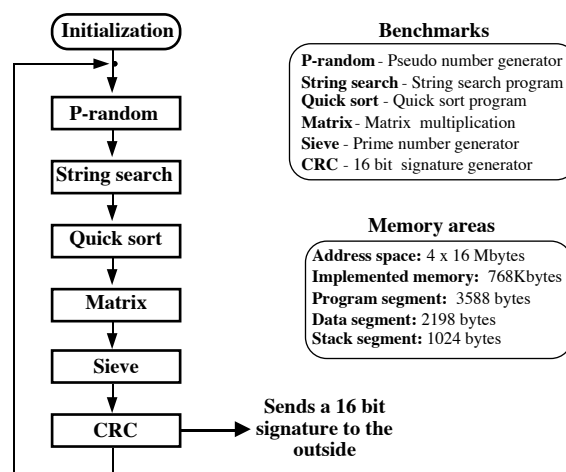


Fig. 8. Program executed by the target system and memory areas.

The program executed by the target systems was written in C and is formed by several benchmarks as shown in Fig. 8. A CRC signature of the results produced by all benchmarks is calculated at the end of each program cycle. That signature (considered as the final result) is sent to the outside by a parallel output port, indicating whether the computer has produced wrong results or not. In this way it is possible to measure the percentage of faults that cause the computer to produce wrong results, i.e. the faults that cause the computer to violate the *fail-silent model* [20].

The injection of the faults is fully automatic and does not require operator intervention. Before injecting each fault the RIFLE causes a reset to the target system, forcing a complete initialization. After the injection of each fault the RIFLE waits 5 seconds before collecting the results. It was experimentally verified that if the errors were not detected within this time they would not be detected.

4.1 - Processor built-in error detection mechanisms

Table 4 shows the coverage and latency obtained for the 68000 built-in error detection mechanisms. The contributions of each individual error detection mechanism are also presented.

<i>Error detection mechanism</i>	<i>Coverage</i>	<i>Average lat. (μSec)</i>	<i>Stand.Dev. (μSec.)</i>
BE - Bus Error	8,6%	16785,4	133807,9
AE - Address Error	11,6%	3521,1	60735,5
II - Illegal Instruction	10,5%	4733,5	54527,9
DZ - Division by Zero	0,3%	65,5	268,1
CHK - CHECK instruction	0,1%	21,4	3,4
TRAPV - Overflow	0,0%	—	—
PV - Privilege Violation	0,2%	32,6	48,7
SI - Spurious Interrupt	0,0%	—	—
AE - \$A code Emulation	1,1%	3345,3	30335,1
FE - \$F code Emulation	7,0%	5790,3	53505,1
HALT	15,6%	31,7	0,5
68K (All mechanisms)	55,2%	5086,7	67501,4

Table 4. Coverage and latency of the 68000 built-in error detection mechanisms (set A).

A coverage of 55,2% and a latency of 5,1 msec (with a large standard deviation) have been measured considering all processor built-in error detection mechanisms. However, a more detailed analysis of the latency distribution shows that most of the faults have been detected within the first 100 μ sec after the fault, which represents a very low latency. This conclusion is quite clear in the chart shown in fig. 9. The small figure at the top represents the latency distribution in detail for the first 50 μ sec after the fault (in linear scale). The maximum coverage near the 32 μ sec corresponds to the HALT mechanism which is activated after a double bus error (32 μ sec result from the 2 x 16 μ sec system bus time-out).

The analysis of the impact of a significant number of faults whose errors have been detected with long latency (this analysis was carried out by using the information recorded in the RIFLE trace memory after the injection of each fault) has shown that most of these faults caused errors in the system stack or have corrupted the code segment. In both situations, the errors stayed latent for a long time and the system behavior was not affected. Only when the corrupted memory area was used by the program did the errors become active and were detected.

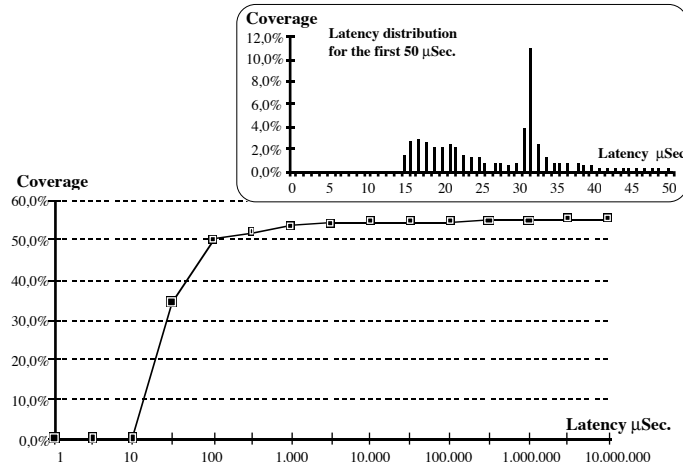


Fig. 9. Latency distribution for the 68000 built-in error detection mechanisms

4.2 - Memory access behavior error detection mechanisms

The evaluation of the memory access behavior error detection mechanisms should take into account the fact that it is not possible to deactivate the 68000 error detection mechanisms. Whenever an error is detected by the 68000 an exception routine is executed and the processor erroneous behavior is interrupted, which makes it impossible to find out whether other error detection mechanisms would detect the error or not. In [9] some ideas are presented to overcome this problem but the fact is that it is not possible to completely decouple the evaluated error detection technique from the processor built-in mechanisms.

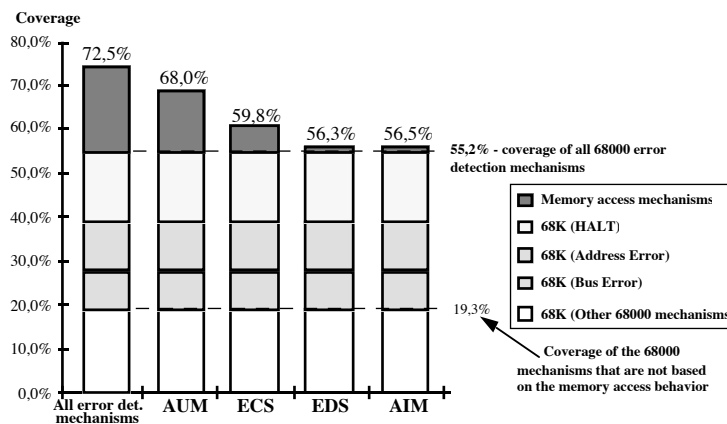


Fig. 10. Coverage increasing with the memory behavior based error detection mechanisms

Fig. 10 shows the results obtained considering the combination of each memory access error detection mechanism with the 68000 error detection mechanisms. Considering the contributions of all mechanisms the coverage achieved was 72,5%. However, only the mechanisms AUM (accesses to unused memory) and ECS (error in the code segment access) give a significant contribution to the coverage improvement.

In fact, the behavior deviations detected by AIM (accesses to unimplemented memory) are also detected by the 68000 bus protocol. The small coverage increasing due to AIM was caused by some faults injected in the control pins that change the type of memory cycle. The low coverage of EDS (error in the data segment access) is mainly due to the small size of the data segment.

Table 5 compares the results obtained with both set of faults. It is quite clear that shorter duration faults (set B) are much more difficult to detect, as the coverage obtained with these faults was only 57,9%. The average latency obtained with Set B was greater than with Set A. However, a more detailed analysis has shown that the latency distribution is not greatly dependent on the fault duration and the average latency increase obtained with Set B was caused by a small number of faults detected with very long latency.

Error Detection Techniques	Set A (duration: 1-4 cycles)		Set B (duration: 1 cycle)	
	Coverage	Ave. latency	Coverage	Ave. latency
68000 built-in mechanisms	55,2%	5,1 mSec.	37,9%	9,7 mSec.
68000 + mem. access mechan.	72,5%	4,7 mSec.	57,9%	6,3 mSec.

Table 5. Coverage and latency obtained with each set of faults.

4.3 - Watchdog timer

Previous research [11] has shown that precise tuning of the WDT period is a factor with major impact on the WDT coverage. In this way, the set of faults A was injected in three different situations:

- Very precise tuning - the WDT period is very close to the benchmark execution time (the WDT period is 60 μ sec. greater than the benchmark execution time);
- Precise tuning - the WDT period is approximately 500 μ sec greater than the execution time;
- Non precise tuning - the WDT period is twice the benchmark execution time

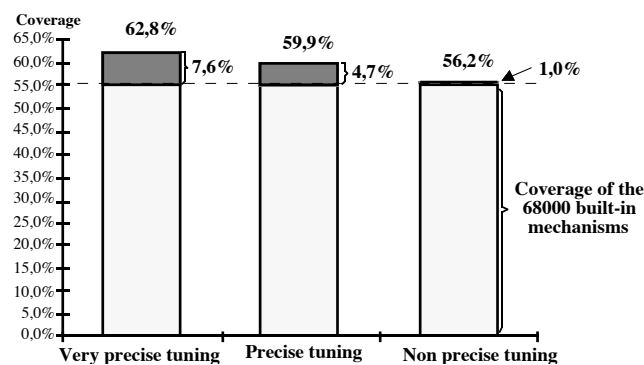


Fig. 11. Coverage variation with the precision of the WDT period.

The results obtained show that the WDT only improves the coverage when its period is very close to the program block execution time. As in most of the actual applications it is not possible to achieve this precision (because the program execution time is very dependent on data) we can conclude that a WDT has little or no use in systems having built-in error detection techniques like the ones existing in the 68000

processor. In fact, the WDT basically detects errors that cause system crashes and should not be considered as a concurrent error detection method.

The average latency obtained is about half the WDT period and it does not depend on the type of fault (duration, pin affected, type of cycle affected, etc.).

4.4 Fail-silent evaluation in a system using behavior based error detection mechanisms [21]

If the errors caused by a fault are detected in time it will be possible to stop the erroneous computer behavior before the output of wrong results, thus preventing the *fail-silent* model violation [20].

Fig. 12 shows the failure manifestations observed when all but the 68000 processor error detection mechanisms were deactivated. This situation represents quite well the failure manifestations of a typical computer without error detection techniques.

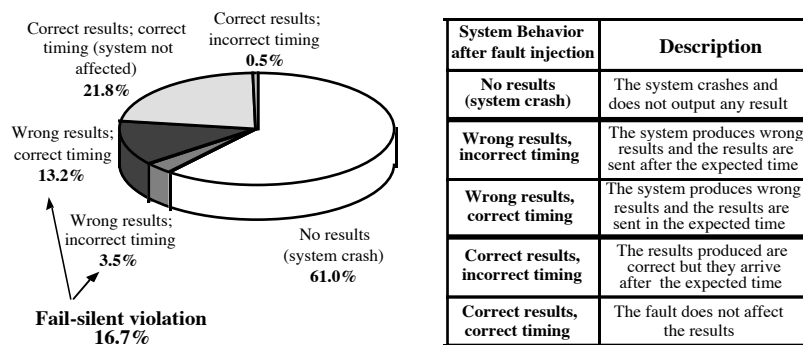


Fig. 12. System behavior after fault (only the 68000 error detection mechanisms are active).

Most of the faults caused the system to crash (61%). However, this case includes actual system crashes (e.g., dead loops) and all the faults whose errors have been detected by the 68000 error detection mechanisms. In fact, as the error handling routines stop the processor, the system assumes the same behavior as in a crash. About 22% of the faults did not affect the program results. The errors generated by these faults have disappeared (or stayed latent) due to the inherent redundancy in the processor and the program [21]. Fail-silent violations have been observed in 17,8% of the faults. Considering only faults of 1 cycle duration (Fig. 13), they went up to 21,8%, which clearly shows that short faults are more likely to produce fail-silent violations. This result does not mean that short faults are more likely to cause the system to produce wrong results. However, as short faults are more difficult to be detected the probability of the undetected errors to cause the fail-silent violation is greater for short faults.

Fig. 13 shows the percentage of fail-silent violations observed considering different error detection mechanisms combinations. SIGN is a signature monitoring error

Error detection			Fail-silent violations	
68K	MEM	SIGN	Set A (dur. 1-4 cycles)	Set B (dur. =1 cycles)
YES	no	no	16.7%	21.8%
YES	YES	no	9.1%	12.2%
YES	no	YES	2.6%	3.4%
YES	YES	YES	1.9%	2.3%

Fig. 13. Percentage of fail-silent behavior violations.

detection method [19]. Considering the contribution of all error detection mechanisms the percentage of fail-silent violations is very low (1,9% for all faults and 2.3% for 1 cycle faults). Without the contribution of the signature monitoring technique (SIGN), which cannot be easily used in ordinary computers, the system has behaved according to the fail-silent model for 90,9% of the faults, i.e., only 9,1% of the faults have generated undetected errors that caused the system to output wrong results. Considering only 1 cycle faults, 12,2% of the faults caused fail-silent violations.

The information stored in the RIFLE Trace Memory allows a detailed analysis of the faults whose errors have caused fail-silent violations. Fig. 14 shows the detailed characterization of these faults. The percentages between parentheses following each item represent the percentage of faults injected according to this item.

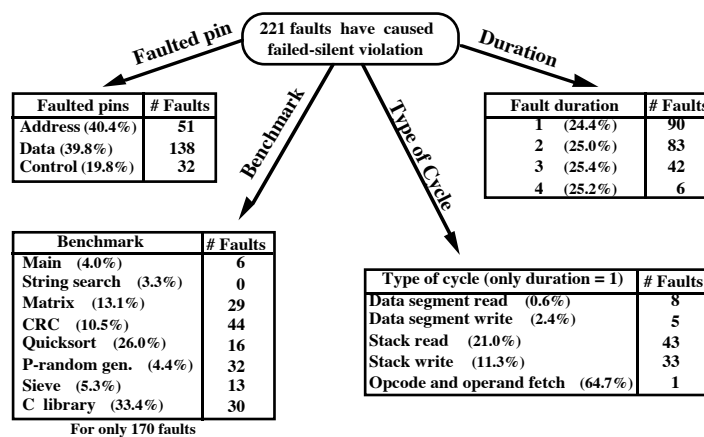


Fig. 14. Characterization of the faults that caused fail-silent behavior violations

The faults that caused the fail-silent violation have the following characteristics:

- Affect (almost) only data cycles (read/write accesses to the stack or data segment);
- Short duration (1 or 2 memory cycles);
- Mainly injected in the data pins.

The explanation for these results is mainly in the fact that the behavior based error detection techniques have a much higher coverage for the faults affecting instruction cycles than for faults affecting data cycles [19, 21]. The benchmark under execution at the moment the faults are injected seems to be very important, which shows that the type of computation performed by the programs are relevant for the probability of fail-silent violation.

5 - Conclusion

This paper presents the architecture of a pin-level fault injector system called RIFLE. This system can be adapted to a wide range of target systems and the faults are mainly injected in the processor pins. A set of abstractions is proposed for the description of the different phases of the fault injection process and each fault is described by a unique set of parameters that completely characterize the fault. The injection of the faults is deterministic and can be reproduced if needed. Faults of different nature can be injected and the fault injector is able to detect whether the injected fault has produced an error or not without the requirement of feedback circuits. RIFLE can also detect specific circumstances in which the injected faults do not affect the target system. Sets of faults with specific impact on the target system can be generated. The evaluation

of a set of simple error detection mechanisms is presented with the main objective of showing the comprehensive results that it is possible to obtain with RIFLE. These results show that up to 72,5% of the errors can be detected with fairly simple mechanisms. Furthermore, for over 90% of the faults the target system has behaved according to the fail-silent model, which suggests that a traditional computer equipped with simple error detection mechanisms is relatively close to a fail-silent computer.

Acknowledgments

This work has been supported in part by the Junta Nacional de Investigação Científica e Tecnológica and FTMPs (Fault-Tolerant Massively Parallel System) Esprit III project.

References

- 1 J. Arlat et. al., "Fault injection for dependability validation: a methodology and some applications", *IEEE Trans. on Software Eng.*, Vol 16, No 2, Feb. 1990, pp. 166-182.
- 2 W. Carter and J. Abraham, "Design and evaluation tools for fault-tolerant systems", *Proc. AIAA Computers in Aerospace Conference*, 1987, pp. 70-77.
- 3 U. Gunneflo, J. Karlsson, and J. Torin, "Evaluation of error detection schemes using fault injection by heavy-ion radiation", *Fault Tol. Comp. Symp., FTCS-19*, June 1989, p.340-347.
- 4 R. Iyer and D. Rossetti, "A measurement-based model for workload dependence of CPU errors", *IEEE Transactions on Computers*, vol. C-35, pp. 511-519, June 1986.
- 5 E. Czeck and D. Siewiorek, "Effects of transient gate-level faults on program behavior", *Fault Tolerant Comp. Symp., FTCS-20*, Newcastle Upon Tyne, June 1990, p. 236-243.
- 6 M. Schmid, et. al., "Upset Exposure by Means of Abstraction Verification", *Proc. 12th Symp. on Fault-Tolerant Comp., FTCS-12*, St. Monica, June 22-24, 1982, p.237-244.
- 7 Y. Crouzet and B. Decouty, "Measurements of fault detection mechanisms efficiency: results", *Proc. of FTCS-12*, Santa Monica, Ca., June 1982, pp. 373-376.
- 8 J. Lala, "Fault detection isolation and reconfiguration in FTMP: methods and experimental results", *5th AIAA/IEEE Digital Avionics Systems Conf.*, 1983, pp.21.3.1-21.3.9.
- 9 M. A. Schuette and J. P. Shen, "Processor Control Flow Monitoring Using Signed Instruction Streams", *IEEE Trans. on Computers*, vol. 36, No. 3, March 1987, p. 264-275.
- 10 A. Damm, "Experimental evaluation of error-detection and self-checking coverage of components of a distributed real-time system", Ph.D. thesis, Univ. of Vienne, Oct. 1988.
- 11 H. Madeira, G. Quadros, and J. Silva, "Experimental eval. of a set of simple error detection mechanisms", *The EUROMICRO Journal*, vol 30, Aug. 1990, North-Holland, pp 513-520.
- 12 J. Cusick, et. al., "SEU vulnerability of the Zilog Z-80 and NSC-800 microprocessors", *IEEE Tran. on Nuclear Science*, Vol. NS-32, NO. 6, Dec. 1986, p. 4206-4211.
- 13 M. Cortes and E. McCluskey, "Properties of transient errors due to power supply disturbances", *Proc. Int.Symp. on Circuits and Systems*, IEEE, pp. 1064-1049, May 1986.
- 14 D. Lomelino and R. Iyer, "Error propagation in a digital avionic processor: a simulation-based study", *Proc. Real Time Systems Symposium*, Dec. 1986, p. 218-225.
- 15 D. P. Siewiorek and Robert S. Swarz, *The Theory and Practice of Reliable Design*, Digital Press, Educational Services, Digital Equip. Corporation, 1984, Bedford, Massachusetts.
- 16 P. Duba and R. Iyer, "Transient fault behavior in a microprocessor: a case study", *Proceedings of ICCD*, October 1988, p. 272-276.
- 17 G. Choi, R. Iyer, et. al. "A fault behavior model for an avionic microprocessor: a case study", *Dependable Computing for Critical Applications*, Springer-Verlag, Avizienis & Laprie (eds), 1990, pp. 177-195.
- 18 J. Karlsson, U. Gunneflo, P. Lidén e J. Torin, "Two Fault Injection Techniques for Test of Fault Handling Mechanisms", *Proc. of International Test Conference*, 1991, p. 140-149.
- 19 H. Madeira and J. Silva, "On-line Signature Learning and Checking", *Dependable Comp. for Critical Applications*, Springer-Verlag, Meyer & Schlichting (eds.), 1992, pp. 394-420.
- 20 D. Powell et. al, "The Delta-4 approach to dependability in open distributed computing systems", *Inter. Symp. on Fault-Tolerant Computing*, FTCS-18, Tokyo- Japan, June 1988.
- 21 H. Madeira and J. G. Silva, "Experimental Evaluation of the Fail-silent Behavior in Computers without Error Masking", to appear at FTCS-24, June, 1994.