

Experimental Evaluation of the Fail-Silent Behavior in Computers Without Error Masking

Henrique Madeira and João G. Silva
 Laboratório de Informática e Sistemas
 University of Coimbra, 3000 Coimbra, Portugal
 Email: hmadeira@mercurio.uc.pt

Abstract

Traditionally, fail-silent computers are implemented by using massive redundancy (hardware or software). In this research we investigate if it is possible to obtain a high degree of fail-silent behavior from a computer without hardware or software replication by using only simple behavior based error detection techniques. It is assumed that if the errors caused by a fault are detected in time it will be possible to stop the erroneous computer behavior, thus preventing the violation of the fail-silent model. The evaluation technique used in this research is physical fault injection at the pin level. Results obtained by the injection of about 20000 different faults in two different target systems have shown that **1)** in a system without error detection up to 46% of the faults caused the violation of the fail-silent model; **2)** in a computer with behavior based error detection the percentage of faults that caused the violation of the fail-silent mode was reduced to values from 2.3% to 0.4%; **3)** the results are very dependent on the target system, on the program under execution during the fault injection and on the type of faults.

Keywords: Fail-silent behavior, failure models, experimental evaluation, pin-level fault injection, error detection coverage.

1. Introduction

A computer is *fail-silent* if it only produces correct results (by results we mean messages, direct actions to the outside world, etc.), i.e., if incorrect results are generated as a consequence of a fault then the computer will not output them (remains silent) [1]. This failure mode represents the same notion as *crash failures* [2] or *fail-fast hardware* [3]. On the other hand, *fail-stop* processors, as defined in [4], can easily be implemented from fail-silent nodes, as the fail-silent model corresponds to the *halt-on-failure* property [4].

The fail-silent model plays an important role in the design of reliable distributed systems and parallel systems in which it is required that an error within one node does not propagate to the other nodes in the system. The fail-silent concept is also very useful in industrial control computers and embedded systems, where it is very important to prevent the output of erroneous commands to the physical process under control.¹

Traditionally, fail-silent computers are implemented by using massive redundancy. The errors are detected by comparing the results produced by replicated modules (either hardware or

software modules) and in case of disagreement the results are not delivered (or the message is not sent, in case of distributed systems).

In this research we investigate if it is possible to obtain a high degree of *fail-silent behavior* in a non-redundant computer by using only simple behavior based error detection techniques. It is assumed that if the errors caused by a fault are detected in time it will be possible to stop the erroneous computer behavior, thus preventing the *fail-silent* model violation.

Figure 1 shows the architecture of a system based on this approach. This architecture is in fact the architecture of a traditional simplex computer with added behavior based error detection techniques.

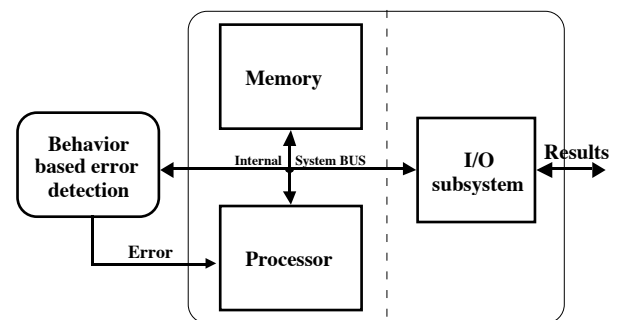


Figure 1 - Traditional architecture with behavior based error detection.

The task of the error detection block is to detect errors in the core of the system, i.e., the subsystem formed by the processor, the memory, and the bus. Errors in the input/output (I/O) subsystem are not directly detected by the error detection mechanisms that form the error detection block.

It is obvious that the fail-silent model can also be violated as a consequence of a fault in the I/O subsystem. This is particularly true in the case of distributed systems in which the I/O subsystem (i.e., the network access modules) can be very complex. However, in many situations it is not difficult to assure fail-silent behavior at the I/O subsystem level, as the duplication (either at the software or hardware level) can be quite cost effective at this level. Furthermore, the message protocol used in distributed systems assures some protection against faults in the network access modules and in the transmission media.

Most of the faults occur in the core of the computer (processor + bus + memory), mainly because these components correspond to most of the computers hardware. Therefore, our study will be focused on this kind of faults.

The evaluation technique used is physical fault injection in the processor pins. Two target systems based on different processors have been used in order to assess the dependency of the results from the specific features of each target system.

¹ In the industrial control field fail-silent systems are usually known as *integer systems* [Kirmann 87]. However, fail-silent systems and integer systems represent the same basic concept.

This report has been accepted for publication outside University of Coimbra and is copyrighted by the publisher. It has been issued as an Internal Report for early dissemination only. Its distribution outside University of Coimbra prior to publication is limited to peer communications and specific requests.

The following section discusses the assumptions and the features of the error detection techniques considered in the architecture shown in figure 1. Section 3 describes the experimental setup and the conditions used in these experiments. This includes the fault model, the fault injection mechanism, the target systems, the workload, and the error detection mechanisms actually implemented in each target system. The results obtained are presented and analyzed in section 4. The results obtained without considering the effect of the error detection mechanisms are presented first. This situation corresponds to considering the target system as a fail-uncontrolled system [1]. The fail-silent results are presented next and analyzed in detail. The analysis includes a detailed characterization of the faults that caused the violation of the fail-silent model in each system. The main differences in the results obtained in each target system are emphasized and explanations for the reasons of these differences are presented. Suggestions to improve the probability of obtaining fail-silent behavior using the architecture shown in figure 1 are presented. The final section summarizes the results and presents the conclusions.

2. Behavior based error detection techniques: assumptions and principles

One major presumption concerning the error detection block in figure 1 is that the cost of all error detection mechanisms should be low. This implies that the error detection mechanisms should not be based on structural redundancy like duplication. On the contrary, all the error detection should be based on the monitoring of the computer behavior.

The behavior based error detection techniques have another important advantage, in addition to the low cost. They can be adapted to existing computers without major changes in the computer hardware/software. This means that results similar to the ones obtained in these experiments can be achieved in off-the-shelf computers, provided that simple behavior based error detection mechanisms are included in those computers.

Using real behavior based error detection techniques means that we have to face two actual traits of these error detection mechanisms:

- the error detection coverage is not perfect;
- the error detection latency is not null.

It is obvious that the violations of the fail-silent model are due to faults whose errors are not detected or are detected with a very long latency. In this context, it is clear that the assessment of the percentage of faults that cause the violation of the fail-silent behavior requires the analysis of the impact of the undetected errors on the program results. In our opinion, this analysis represents a further step in the error detection techniques evaluation. In fact, in spite of the many behavior based error detection techniques proposed and evaluated during the last years, all existing evaluation works concentrate on the assessment of the error coverage and latency, and neglect the analysis of the undetected errors impact.

It seems clear that not all the undetected errors will cause the program to produce wrong results. Considering the inherent redundancy in the computer physical structures and programs, it is expected that some undetected errors will disappear without causing wrong results. This effect of error vanishment has already been reported in several works simulating internal processor faults [5, 6, 7], which suggests that a similar effect might exist at the system level.

In this way, the analysis required for assessing the percentage of faults that violate the fail-silent model also characterizes the

real usefulness of the used error detection techniques. In fact, in addition to the traditional characterization based on the coverage and latency we will introduce a more meaningful measure, which is the percentage of undetected errors that cause the program to produce wrong results.

In the remaining of this section we will review the key aspects of behavior based error detection techniques. The reader already familiar with these techniques should proceed to the next section.

In the behavior-based approach, information describing a particular aspect of the system behavior (e.g. the program control flow) is previously collected. This is usually accomplished from the program source code during the compile/assembly time. This information is then compared to the behavior information gathered at run-time, in order to detect deviations from the correct behavior, i.e. errors.

The entity that checks if the run-time behavior matches the behavioral abstraction collected previously is usually called a watchdog. A watchdog can be implemented using software —*watchdog process*— [8] or, more commonly, using a combination of software and hardware. Several examples can be found in the literature, ranging from quite simple devices such as *watchdog timers* to complex *watchdog processors*. A good survey of these error detection techniques can be found in [9].

Several behavior abstractions can be used for error detection such as program control flow, memory access behavior, control signals, reasonableness of results, object type and restrictions, or timing features. Previous research [10] evaluated and compared some of these behavior abstractions and found that the monitoring of many of these features yields a quite high error detection potential.

Many error detection techniques based on the monitoring of one (or more) of these behavior abstractions have been proposed. Program control flow monitoring techniques (particularly using signature monitoring) have been proposed by several authors [11, 12, 13, 14, 15, 8]. Error detection techniques based on memory access behavior have also been proposed [16, 17]. In [18] several error detection mechanisms based on memory access behavior were experimentally evaluated. Error detection techniques based on the reasonableness of results have also been proposed [19].

3. Experiment conditions

After the injection of a fault it is assumed that the target computer will exhibit one of the following failure manifestation:

- **No results (system crash):** after the fault the system crashes and does not output any result;
- **Wrong results, incorrect timing:** after the fault the system produces wrong results and these results are not outputted on time (they are sent after the expected time);
- **Wrong results, correct timing:** the system produces wrong results and these results are outputted in the expected time;
- **Correct results, incorrect timing:** the results produced after the fault are correct but they arrive after the expected time.
- **Correct results, correct timing:** the fault does not affect the results;

It is assumed that the target computer violates the fail-silent model if *the failure manifestations include the production of wrong results* (second and third failure manifestations). The behavior "correct results, incorrect timing" is considered to comply with the fail-silent model, as for many situations some

This report has been accepted for publication outside University of Coimbra and is copyrighted by the publisher. It has been issued as an Internal Report for early dissemination only. Its distribution outside University of Coimbra prior to publication is limited to peer communications and specific requests.

delay in the results is not very important. Obviously, this is not true for hard real-time systems, as in this case delays in the results might violate the application dead line, which should be considered as a fail-silent violation.

In general terms, the organization of the experiments is as follows: the target system executes a cyclic set of benchmarks and a 16 bit signature is calculated at the end of each cycle from the results produced by each benchmark. This signature (which is considered the final benchmarks result) is sent to the outside and a special circuit checks if the signature is correct or not. The signature corresponding to correct results obtained from a given set of input data is known in advance, which makes it easy to check whether the signature is correct or not. The circuit that checks the result signature also detects the situation in which the target system does not output the signature (system crash; no results).

After the injection of each fault the following results are collected:

- Type of failure manifestation;
- Error detection mechanisms that have detected the errors;
- Latency of each error detection mechanism.

It should be noted that in these experiments the detection of an error does not stop the erroneous processor behavior, as is suggested in figure 1. If the processor is stopped after the detection of an error (by the first mechanism) it will not be possible to find out whether the other error detection mechanisms would have detected the error or not.²

As the latency of each error detection method is gathered after the injection of each fault it is possible to know exactly which error detection mechanisms (or combination of mechanisms) have detected the error in time to avoid the violation of the fail-silent model.

In the following subsections we will discuss in detail the different points raised by this general description.

3.1 Fault model

Transient and intermittent faults are the major physical cause for computer failures. Several studies indicate that even in opti-

might cause a random memory access virtually equivalent to a random memory access caused by a transient fault.

The objective of the fault injection is to emulate actual faults, or inject faults that cause the same error manifestations as actual faults. In this way it seems obvious that the injected faults should emulate transient faults.

As stated before, the fault injection technique used in this research is physical pin-level fault injection. The faults are injected in the processor pins. In this way, the injected faults not only emulate internal processor faults but also directly emulate faults in the memory and the bus.

From the point of view of pin-level fault injection, a transient fault is a very short duration fault. Previous researches [12, 18] have shown that the shortest fault should last for one memory cycle. In fact, faults with even shorter duration (e.g. one processor clock) cause the same error manifestations of one memory cycle faults or do not cause any error at all. This is a normal consequence of the way the processors validate the signals during a memory cycle. On the other hand, faults that span for many cycles are easily detected, even by relatively simple error detection mechanisms [12, 18], which suggests that permanent faults are not a problem in obtaining fail-silent behavior.

A major problem of pin-level fault injection is that the pin-level manifestations of internal ICs faults are very difficult to estimate, thus making the definition of representative sets of faults at the pin-level quite obscure.

Few works can be found in the literature addressing the problem of how internal microprocessor faults are manifested at the pins. A gate-level simulation study in a digital avionics processor [5] has shown that the error activity at the output pin-level is sensitive to the place where the fault is located within the chip. This suggests that within-chip faults can be emulated by injecting the correspondent error distributions at the pin-level. In the same paper, distributions of errors at the output pins are presented and a multiple pin error model is proposed. Other gate-level simulation works also support the multiple pin error model [7].

A different family of research results with interest for the determination of how internal faults are manifested at the pins

Research	Method	Processor	Error location				Pins affected		
			Address	Control	Data	Mixed	1 pin	2 pins	Several
Gunnflo 89	Heavy ion	6809	66%	22%	4%	8%	70%	9%	21%
Cusick 86	Heavy ion/protons	Z80	51%	6%	32%	11%	72%	5%	13%
Karlsson 91	Heavy ion	6809	64%	27%	5%	4%	72%	10%	18%
Karlsson 91	Power supply	6809	17%	80%	1%	2%	51%	35%	14%

Figure 2 - First error manifestation of internal processor faults at the pin-level [10, 22, 23].

num environments it is estimated that more than 90% of the physical faults in computers are transient [20, 21]. Furthermore, the prevalence of transient faults over permanent ones is increasing, due to smaller device dimensions.

Software faults (i.e., bugs) are another important cause for computer malfunctions. However, considering the nature of these faults (software bugs can be regarded as human mistakes), it is very difficult to emulate these faults. Nevertheless, several common software faults cause erroneous computer behavior quite similar to transient faults and can also be detected by behavior based error detection techniques. For example, the use of an uninitialized pointer, which is a typical software fault,

have been obtained by using radiation and power supply disturbances to inject actual faults inside processors. Figure 2 summarizes the main results published on this topic.

The first thing to be noted is that all these works give some support to the single pin error model, which contradicts the results obtained by the simulation works mentioned above. The results in figure 2 correspond to the first error manifestations and subsequent errors may affect more pins. However, these secondary errors can be generated outside the chip as a consequence of the first pin-level error (e.g. a single bit error in the address pins during a read cycle will cause subsequent multiple errors in the data pins because an erroneous memory cell was addressed)

Only a small percentage of internal faults in figure 2 have caused errors in multiple pins. Another important aspect is the

² Internal 68000 error detection methods are an exception to this rule. Everytime the 68000 detects an error an exception handling routine which stops the erroneous computer behavior is executed.

This report has been accepted for publication outside University of Coimbra and is copyrighted by the publisher. It has been issued as an Internal Report for early dissemination only. Its distribution outside University of Coimbra prior to publication is limited to peer communications and specific requests.

fact that the errors are relatively well distributed among the pins (the greater percentage of errors in the address pins is mainly because the used processors have more address pins than data and control pins). The small percentage of errors in the data pins results from the fact that these pins are normally asserted as input (read accesses [data and instruction fetch] are more frequent than write accesses).

It seems clear that there are still strong limitations in understanding the external manifestations of transient processor faults. However, the results presented above give some support to the general criteria used for the definition of sets of faults in the present research (figure 3).

Criteria used for the definition of sets of faults
Fault activation places: chosen at random
Faulted pins: all processor pins selected at random
Number of pins faulted by each fault: 1 pin
Fault duration: 1 - 4 memory cycles
Type of faults: inversion

Figure 3 - Fault generation criteria.

The fault activation places determine the program locations in which the faults are injected. They are described by a set of parameters (see next subsection) and these parameters are generated in such a way that the faults are injected randomly during the program execution (random in the dynamic sense).

All the processor pins (excluding asynchronous input control pins) are equally likely to be affected by the faults. This is a bit contradictory with the results presented in figure 2, in which the percentage of errors in the processor data pins is clearly smaller than in the other pins. However, it should be noted that faults injected in the processor data pins also emulate memory and bus faults.

Only one pin is affected by each fault. From the existing studies it is not clear which error model (single or multiple pin) should be used. Thus, we decided to use single pin faults because these faults are, in general, more difficult to be detected than multiple pin faults, which means that the results obtained in this way represent a pessimistic situation.

Most of the faults are 1 cycle duration because these faults are closer to transient faults than longer duration faults. However, faults of longer duration (2, 3, and 4 cycles) are also injected.

All the faults are inversion type faults (the state at the faulted pin is inverted during the injection of the fault). This fault type assures that all faults will cause bit flips, thus speeding up the fault injection phase. It is worth noting that inversion type faults are equivalent to stuck-at faults for all faults with duration equal to 1 cycle, as stuck-at faults are only effective if the state of the pin is changed.(i.e., inverted).

3.2 The pin-level fault injection tool

Pin-level fault injection has become an attractive way of assessing fault-tolerant system measures and of validating specific fault-tolerance mechanisms. Several pin-level fault injection studies have been reported in the literature, many of them concerning the evaluation of fault-tolerance mechanisms in specific target systems [24, 12, 15]. A more general methodology for physical fault injection has been presented in [25].

However, the injection of pin-level faults in the complex microprocessors (and other chips) 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 obtained from pin-level fault injection.

Figure 4 illustrates the typical analysis required after the injection of each processor pin-level fault in order to find out whether the injected fault should be considered in the results or not. The first step is to detect whether the injected fault has produced an error or not. It should be noted that not all the pin-level faults produce errors. For example, the injection of a stuck-at-high fault in a pin that is already at high state will not generate errors (transition from "Fault" to "Correct behavior" in figure 4). 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. Only the faults that cause deviations in the target system behavior should be considered.

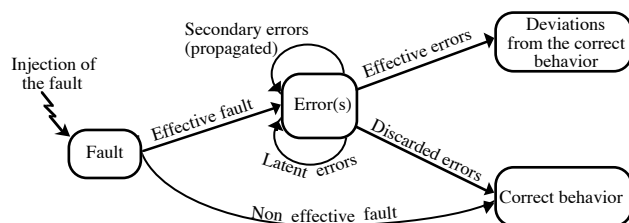


Figure 4 - Impact of a pin-level fault in the target system behavior.

A pin-level fault injector called RIFLE has been developed at the University of Coimbra. To the best of our knowledge RIFLE is the only pin-level fault injector capable of performing the above analysis. A RIFLE predecessor (with the same basic architecture) has been extensively used in the evaluation of behavior based error detection techniques [18, 15].

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 figure 5. It is formed by three modules:

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 [25] and each injection element is similar to an electronic switch.

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

This report has been accepted for publication outside University of Coimbra and is copyrighted by the publisher. It has been issued as an Internal Report for early dissemination only. Its distribution outside University of Coimbra prior to publication is limited to peer communications and specific requests.

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 in the target processor bus.

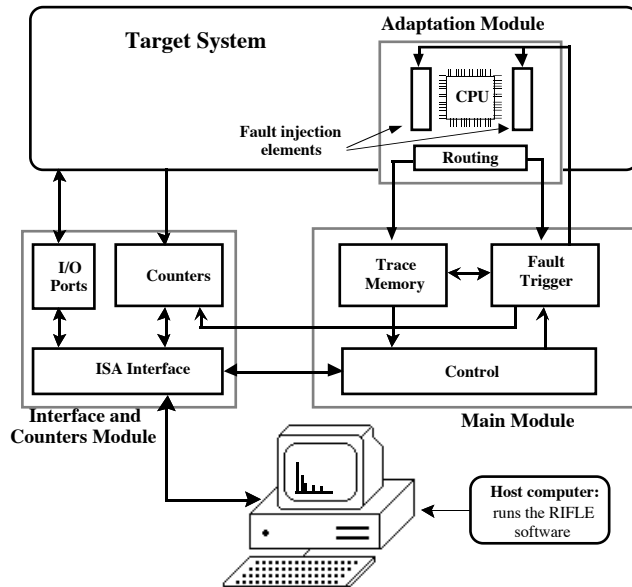


Figure 5 - RIFLE organization.

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

Each fault is described by a unique set of parameters which completely characterize the fault (synchronized fault injection). Particularly, the point during the execution of the target system program where the fault should be injected can be specified with the precision of one bus cycle.

The injection of the faults is deterministic and can be reproduced if needed. Faults of a different nature (stuck-at, inversion, etc) 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 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. Furthermore, the error propagation analysis performed by the RIFLE software can detect non effective faults. One example is the case of faults that cause the processor to fetch an incorrect instruction which is later discarded from the prefetch queue without ever being executed. This kind of faults must be accounted for in order to obtain a correct assessment of the fault-tolerance technique under evaluation.

The RIFLE can be used in different target systems. The details concerning one particular target processor/system are grouped into a specific 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). The RIFLE has 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 T800³ have been finished.

3.3 Target systems

Previous fault injection works on the evaluation of error detection techniques [10, 18, 23, 15, 26] have shown that small changes in the experimental conditions might cause significant changes in the results. Aspects such as the type of faults, the workload, and the type of memory cycle affected by the faults have great impact on the results. These variations are a major problem for the generalization of the results obtained from fault injection.

One of the most important steps towards the generalization of the fault injection results is to assess the impact on the results of using different target systems in the same experiment. For this reason we decided to use two different target systems:

- An experimental system built from the design of a small industrial controller based on the Z80 processor;
- A commercial VME bus computer based on the 68000 processor.

The size and complexity of these two systems are quite different and their processors are quite different too. Whereas the Z80 is a small 8 bit processor the 68000 is a typical 32 bit complex instruction set processor. Considering the instruction set complexity these two processors represent two completely different situations, which suggests that the results obtained in each system are also representative of quite diverse situations. It should be noted that the processor instruction set plays a major role in determining the erroneous processor behavior after a fault, and it consequently has great impact on the performance of the behavior based error detection mechanisms.

The main reason why we did not choose a processor more recent and complex than the 68000 is because the RIFLE was not prepared to inject faults in such processors at the time these experiments were performed.

It is worth saying that the injection of physical faults and the analysis of results in complex processors is not simple. Even the most recent works in this field [10, 23, 26] have used fairly simple processors with about the same complexity as the Z80.

Another reason for the choice of the target system has to do with the fact that it is very difficult (or even impossible) to implement the hardware required by the signature monitoring techniques (which is one interesting error detection technique) in the outside of many recent processors because most of them use complex prefetch queues or pipelining. In fact, the use of signature monitoring techniques in real systems will require the hardware signature generator to be placed inside the processor.

3.4 - Error detection techniques implemented in the target systems

The error detection techniques implemented in the target systems are:

68K - 68000 processor internal error detection mechanisms

This processor has several internal error detection mechanisms. The most relevant ones are the detection of

³ The fault injector for systems based on the T800 is one of the tasks of the FTMPs (Fault-Tolerant Massively Parallel System) Esprit project. The consortium is composed by Parsytec, British Aerospace, University of Erlangen, University of Leuven, University of Paderborn, and University of Coimbra.

This report has been accepted for publication outside University of Coimbra and is copyrighted by the publisher. It has been issued as an Internal Report for early dissemination only. Its distribution outside University of Coimbra prior to publication is limited to peer communications and specific requests.

accesses to non-implemented memory, fetch of invalid instructions, unaligned instruction fetch, and unaligned word accesses.

ECI - Error capturing instructions

This technique was proposed in [27] and has been evaluated in [26]. It is only based in software and consists of the insertion of special instructions (e.g. software interrupt instructions) within the code segment in such a way that these instructions are never executed during normal program execution. In case of a fault there is a chance of occurring control flow errors that cause the execution of one ECI, which detects the error. The ECI can also be inserted within the data segment and the unused memory.

MEM - Memory access error detection

This is a set of error detection mechanisms similar to the memory protection features delivered by a normal memory management unit. The memory is (logically) divided in segments and the errors are detected whenever the rules defined for the access to a given segment are violated. For example, the fetch of instructions outside the program segment, the access to unused memory, or the write accesses over the program segment are examples of erroneous situations. An experimental evaluation of these error detection mechanisms has been done in [18].

SIGN - Signature monitoring technique

The basic idea of signature monitoring techniques is to divide the program code in blocks at compile/assembly time and generate a signature for each block from the opcodes and operands of all instructions in the block. At run-time, as the instructions of each block are being executed, special hardware regenerates the signature of the block and compares the run-time signature with the reference signature previously calculated. An error is detected when these two signatures disagree. Many signature monitoring techniques have been proposed in the last years corresponding to different strategies of defining program blocks and checking signatures. The specific technique implemented in both target systems was proposed in [15].

The memory access error detection mechanisms and the signature monitoring technique were implemented in both target systems. The ECI technique was not implemented in the 68000 system. In fact, as the coverage of this technique is low practically all the errors detected by ECI would have been also detected by the 68000 built in error detection mechanisms. In this way, the coverage of the 68000 error detection mechanisms plus ECI technique is practically the same as the coverage of the 68000 error detection mechanisms alone.

An error detection technique can be characterized by three figures of merit: *coverage*, *latency* and *cost*. The cost of an error detection technique consists of several factors such as the hardware overhead, the memory overhead, the performance overhead, and the software overhead. Depending on the error detection technique, one or more of these overheads can be null.

Error detection methods				
	68K	ECI	MEM	SIGN
Hardware	None	None	Small	Small
Memory	None	> 30%	None	< 15%
Performance	None	Small	None	

Figure 6 - Typical cost of the error detection techniques (from [28, 12, 26, 15]).

Figure 6 shows the typical cost of these error detection techniques, considering the different cost items. As the cost is very dependent on the actual situation each technique is used (system, workload, etc) we present typical cost instead of the actual cost in our implementation (which is smaller than the typical values presented). The coverage and latency of these error detection techniques (evaluated by fault injection) will be presented in the next section.

3.5 Benchmarks

The program executed by both target systems was compiled from the same original program (written in C) and is formed by several benchmarks as shown in figure 7.

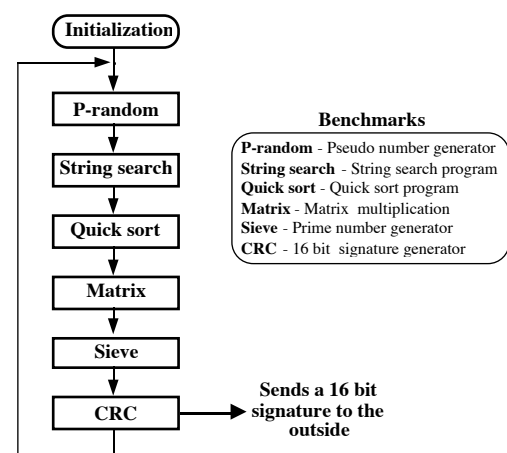


Figure 7 - Program executed by each target system.

A signature of the results produced by all benchmarks is calculated at the end of each program cycle. If any benchmark produces wrong results the final signature will be wrong with a probability of $1 - 2^{-16}$. 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.

4. Results and analysis

The sets of faults used in these experiments are described in figure 8. These faults were defined according to the criteria described in section 3.1. Some faults were not injected because the fault activation parameters correspond to program zones that are never executed with the used program data. In the 68000 system there are some injected faults whose errors were discarded by the prefetch queue. These faults must not be considered and the RIFLE software automatically discarded these faults. In the Z80 system there are no discarded faults because this processor does not have a prefetch queue.

Faults	Set A-Z80	Set B-Z80	Set A-68K	Set B-68K
No. faults	10000	10000	12500	12500
Non injected	502	518	753	777
Discarded	0	0	114	411
Effective faults	9498	9482	11633	11312
Duration	1-4 cycles	1 cycle	1-4 cycles	1 cycle

Figure 8 - Sets of faults used in these experiments.

The injection of these sets of faults by the RIFLE 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 (failure manifestations and error detection latencies).

It was experimentally verified that if the target system does not send the results (signature) within 5 seconds after the fault it means that the system has crashed and will not output results any more. It should be noted that 5 seconds correspond to 8 times the average benchmark execution time in the 68000 system and 4,5 times in the Z80 system.

4.1 Failure manifestations without considering the error detection mechanisms

One interesting aspect that has never been investigated by physical fault injection is the failure manifestations observed in an ordinary computer, i.e. a computer without error detection techniques. This study seems to be very interesting, as in the distributed systems literature a very common fault model is the fail-silent (or crash) model.

Figure 9 shows the results obtained by the injection of the sets of faults A-Z80 and A-68K, disregarding the presence of the error detection mechanisms. However, it should be noted that

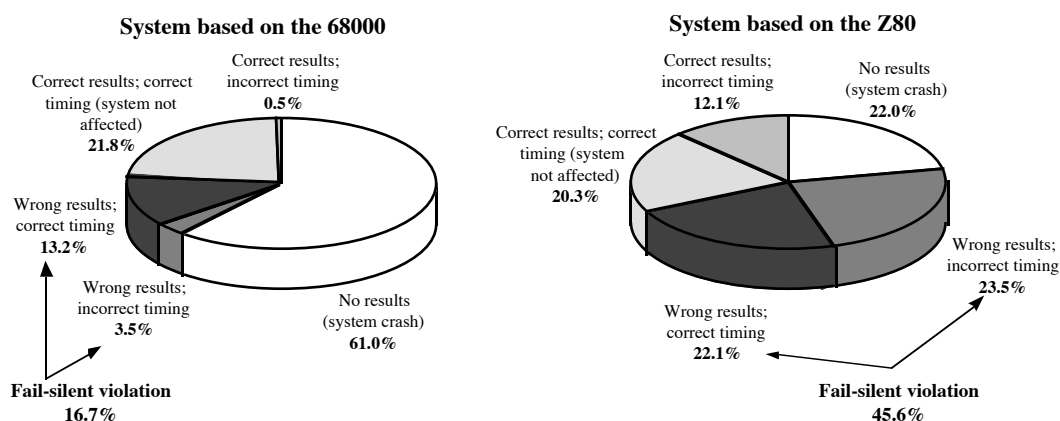


Figure 9 - Results without considering the effect of the error detection mechanisms (in the 68000 system the internal processor error detection mechanisms are active).

the 68000 processor error detection mechanisms cannot be deactivated. Thus, the results obtained in each target system represent two different situations: in the Z80 system they represent, in fact, the failure manifestation in a computer without any error detection (pure fail-uncontrolled system) while in the 68000 they represent the failure manifestations considering only the 68000 error detection. It should be noted that when an error is detected by the 68000 mechanisms the erroneous processor behavior is stopped, which avoids further fail-silent violations.

In the Z80 system 45.6% of the faults caused the fail-silent violation, which clearly demonstrates that it is not correct to assume fail-silent behavior in a computer without error detection (or error masking) techniques. In the 68000 system only 16.7% of the faults violated the fail-silent model. This shows that even fairly simple error detection mechanisms like the ones existing in the 68000 can greatly decrease the probability of violation of the fail-silent behavior.

Comparing the failure manifestations in each target system strong differences can be observed. For example, the percentage

of system crashes is much greater in the 68000 system (61%) than in the Z80 system (22%). However, the 61% of system crashes in the 68000 system includes actual crash situations (e.g. dead loops) and all the faults whose errors have been detected by the 68000. In fact, as the error handling routine stops the processor, the system assumes the same behavior as in a crash.

The percentage of faults that caused "correct results, incorrect timing" is much greater in the Z80 system (12.1%) than in the 68000 system (0.5%). This difference is mainly due to the presence of the 68000 processor error detection techniques. In fact, a more detailed analysis have shown that considering only the MEM error detection technique implemented in the Z80 system will be enough to reduce the 12.1% of faults that caused "correct results, incorrect timing" to less than 1%.

An interesting result is in the fact that in both target systems about 22% of the faults did not affect the program results. The manual inspection of some of these faults (by observing the behavior information recorded in the RIFLE trace memory) has shown several situations:

- Faults whose errors are neutralized by the next instructions;
- Faults affecting the execution of instructions that do not contribute for the benchmark results (some of these "redundant" instructions are due to the inefficiency of the compiler);
- Faults whose errors are tolerated by the semantic of the benchmark under execution. For example, in programs such

as the string search or the quick sort, which perform lots of comparisons, many errors affecting only one of the operands in a comparison do not change the comparison result.

4.2 Fail silence violations in systems with behavior based error detection

Before studying the failure manifestations let us present the main coverage and latency results obtained with each error detection mechanism. Detailed results and analysis are omitted for reasons of space and because the main goal of this work is not the error detection coverage and latency evaluation.

Some aspects should be noted in the results presented in Figure 10. The coverage obtained for the B-68K and B-Z80 sets of faults (duration = 1 cycle) is lower than the coverage obtained considering the sets A-68K and A-Z80 (duration between 1 and 4 cycles). This was expected as faults of longer duration cause greater impact on the target system.

This report has been accepted for publication outside University of Coimbra and is copyrighted by the publisher. It has been issued as an Internal Report for early dissemination only. Its distribution outside University of Coimbra prior to publication is limited to peer communications and specific requests.

The coverage obtained for the signature monitoring technique in the 68000 system is considerably lower than the coverage

a small set of faults whose errors were detected with very long latency. This is particularly true for the signature monitoring

System based on the 68000			
Error Detection Techniques	A-68K set		B-68K set
	Cover.	Aver. lat.	Coverage
68K	55.2%	5.1 mSec.	37.9%
(68K) + MEM	72.5%	4.7 mSec.	57.9%
(68K) + SIGN	87.3%	3.9 mSec.	74.4%
(68K) + MEM + SIGN	90.4%	3.8 mSec.	80.1%

System based on the Z80			
Error Detection Techniques	A-Z80 set		B-Z80 set
	Cover.	Aver. lat.	Coverage
ECI	48.3%	47.8 mSec.	34.4%
MEM	72.6%	9.0 mSec.	58.0%
SIGN	97.1%	0.4 mSec.	91.8%
MEM + SIGN	97.8%	0.3 mSec.	93.7%

Figure 10 - Error detection coverage obtained with the mechanism used in each system.

obtained in the Z80 system. This is mainly due to the fact that the percentage of data cycles in the 68000 system is higher than in the Z80 system (this is typical of CISC processors like the 68000 [29]). It should be pointed out that signature monitoring techniques are not very good in the detection of errors affecting data cycles [12, 15].

An average latency of few milliseconds was observed for

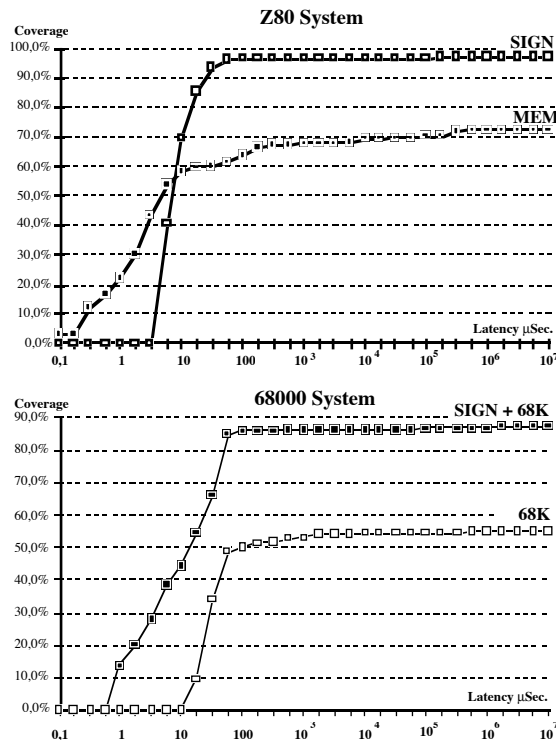


Figure 11 - Latency distribution for some error detection mechanisms.

most of the error detection mechanisms. However, the great majority of the errors were detected with very short latency (< 100 microseconds). The relatively large average latency is due to

techniques. Figure 11 presents the *comulative coverage versus latency* for some error detection methods and shows quite clearly that most of the errors are detected with very short latency. As both target systems are rather slow machines (the average instruction execution time is about 1 microsecond) we can conclude that the average number of instructions executed from the error arising until the error detection is very low. This is very important for the design of error recovery mechanisms.

The percentages of faults that caused the violation of the fail-silent model, considering different combinations of error detection mechanisms, are presented in figure 12. Considering all the error detection mechanisms the percentage of fail-silent violations is very low in both systems. Even in the worst case (faults of 1 cycle duration injected in the 68000) only 2.3% of the faults have violated the fail-silent behavior, which means that the probability of the target system to behave according to the fail-silent model after a fault is 0,977. In the Z80 system the percentage of faults that caused the fail-silent model violation is lower than in the 68000 because the signature monitoring technique presents a better coverage in the Z80 system.

One interesting result (not included in figure 12) is the fact that practically all fail-silent violations were caused by undetected errors. That is, the situation in which the fail-silent is violated because the errors are detected too late (after the program has sent the wrong results) was not verified at all in the Z80 system and only occurs in 4 faults in the 68000 system. This suggests that the fail-silent violations are not very dependent on the rate the results are outputted. Obviously, the short latency of the error detection mechanisms plays a major role in this result.

The signature monitoring technique is the most effective error detection technique. If this error detection technique is not considered the percentage of faults that violate the fail-silent behavior increases significantly in both target systems.

One interesting result was obtained considering only the memory access error detection techniques (or MEM + 68K in the 68000 system) because this represents the case of many existing systems which have hardware memory access protection. As we can see, the results obtained in both systems (about 10% fail-silent violations) suggest that a system like this will behave according to the fail-silent model for about 90% of the faults.

System based on the Z80				
Error detection			Fail-silent violations	
ECI	MEM	SIGN	A-Z80	B-Z80 (dur. =1)
YES	no	no	20.0%	24.5%
no	YES	no	9.9%	14.2%
no	no	YES	0.6%	1.7%
no	YES	YES	0.4%	1.1%

System based on the 68000				
Error detection			Fail-silent violations	
68K	MEM	SIGN	A-68K	B-68K (dur. =1)
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%

Figure 12 - Fail-silent results considering the error detection mechanisms.

4.2.1 Characterization of faults that cause the fail-silent model violation

One interesting aspect is to characterize the faults that escape to all error detection mechanisms and cause the computer to produce wrong results, thus violating the fail-silent model. Figures 13 and 14 show a detailed characterization of these undetected faults in each target system. The percentages between parentheses following each item represent the percentage of faults injected according to this item. For example, 50.1% of the faults were injected in the Z80 address pins (see figure 13).

In both systems most of the faults that caused the fail-silent violation are characterized by the following features:

- 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 of 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. This is particularly true for the signature monitoring technique which detects practically all faults affecting instruction cycles [12, 15]. For example, considering the case of the Z80 system we can see that more than 81% of the faults (of duration = 1) have been injected during instruction cycles. However, none of these faults caused fail-silent violations. It should be noted that the percentage of faults affecting each type of cycle depends on the actual percentage of cycles resulting from the program execution (i.e., the fault locations with respect to the program execution are random).

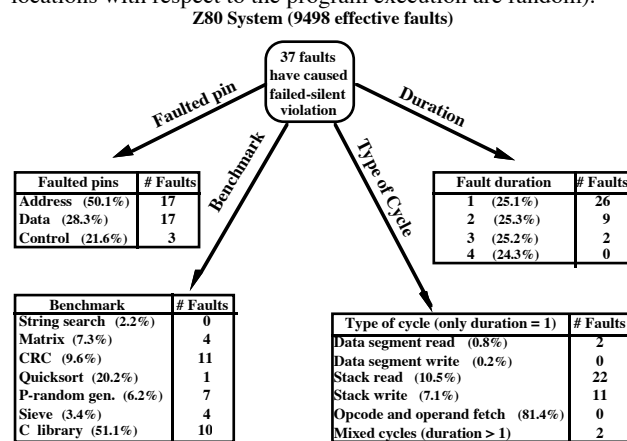


Figure 13 - Characterization of the faults that caused the fail-silent violation in the Z80 system (set A-Z80).

The benchmark under execution at the moment the faults are injected seems to be very important in both target systems.⁴ For example, faults injected during the CRC function or the P-random number generator seem to be more likely to cause the fail-silent violation than faults injected during the execution of the String Search or the Quick Sort. The detailed reasons for these differences are not very clear. However, it should be noted that, for example, the CRC algorithm (signature calculation) is clearly more sensitive than the string search, which shows that

⁴ The percentage of faults injected during the execution of each benchmark (values between parentheses) only depends on the actual benchmark execution time.

the type of computation performed by the programs are relevant for the probability of fail-silent violation.

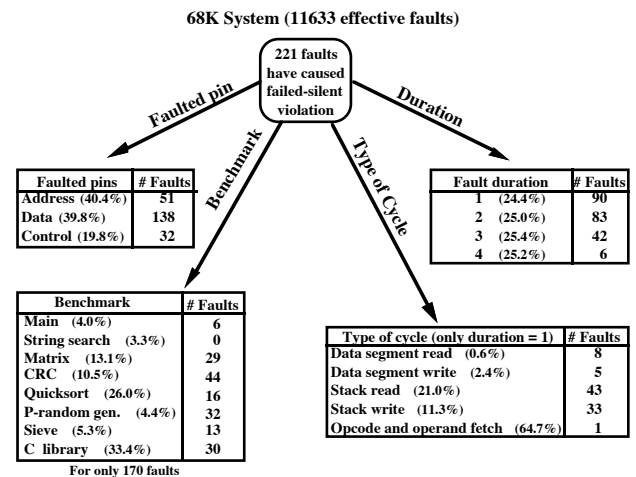


Figure 14 - Characterization of the faults that caused fail silence violations in the 68000 system (A-68K).

The quick sort program is an interesting case because it is the only recursive benchmark. As we can see in figures 13 and 14, in spite of the large percentage of faults injected during this benchmark (20.2% in the Z80 and 26.0% in the 68000 system) only few faults caused the fail-silent violation. It should be noted that the algorithm used for the calculation of the signature from the benchmark results is a cyclic code, which means that it also detects Quick Sort errors such as elements out of order.

4.2.2 Discussion of the results and future work directions

Considering the results obtained in both target systems the probability of the target system to behave according to the fail-silent model ranges from 97.7% to 99.6% (see figure 12). Considering the relative low cost of the used behavior based techniques, these results are in fact quite impressive. However, it should be noted that the signature monitoring technique has great impact on these results. Without considering this technique (which is very difficult to implement in modern systems) the probability of the target system to behave according to the fail-silent model is about 90% for both target systems.

The analysis of the faults that caused fail-silent violations suggests that the improvement of these results will require the use of error detection techniques based on the semantic verification of the data manipulated by the programs. Data manipulation and semantic checking need the notion of correctness, which can be provided by assertions. An assertion is an invariant relationship between the variables of a program, written as a logical statement and inserted at different points in the program [9]. The assertions can be written from the program specification or using some property of the problem or algorithm. However, the assertions are not transparent to the programmer, which is a serious limitation for many situations.

One important aspect is that several features of the used experimental setup make the fail-silent behavior more difficult to obtain. The following arguments support this idea:

- As the benchmark results are compressed into a 16 bit signature only one instruction is required to output the results (two instructions in the Z80). This means that even in the case of

This report has been accepted for publication outside University of Coimbra and is copyrighted by the publisher. It has been issued as an Internal Report for early dissemination only. Its distribution outside University of Coimbra prior to publication is limited to peer communications and specific requests.

faults that cause drastic changes in the system behavior, it is possible to violate the fail-silent behavior because a single instruction is enough to output the wrong results. This situation would not be possible if the results were sent by using a message according some protocol;

- Many fail-silent violations were caused by faults affecting the CRC function, which means that the signature calculation is itself a very critical task;
- The severity of the errors introduced in the program results was not considered (a single bit error is enough to cause a wrong signature). In a more realistic situation it is expected that some of the errors in the results can be corrected. Furthermore, considering the normal situation in which the results are outputted by using messages, some of the errors would cause inconsistent messages and would be detected by the receiver;

Future work directions include performing this study in real distributed systems. Two different systems are being considered: a system based on personal computers connected by Ethernet and a Parsytec Xplorer machine (based on T800 transputers).

5. Conclusions

In this paper we have investigated if it is possible to obtain a high degree of fail-silent behavior from a non-redundant computer by using only simple behavior based error detection techniques. It was assumed that if the errors caused by a fault are detected in time it will be possible to stop the erroneous computer behavior, thus preventing the violation of the fail-silent model.

The evaluation technique used is physical fault injection at the pin level. Two different target systems were used (a small industrial controller based on the Z80 and a VME bus computer based on the 68000 processor) in order to assess the dependency of the results on the target system. During the fault injection both target systems execute the same set of benchmarks compiled from the same original program (written in C).

Results obtained by the injection of more than 20000 faults in two different target systems have shown that **1)** in a system without error detection up to 46% of the faults caused the violation of the fail-silent model; **2)** in a computer with behavior based error detection the percentage of the faults that caused the violation of the fail-silent model was reduced to values from 2.3% to 0.4%; **3)** the results are very dependent on the target system, on the program under execution during the fault injection and on the type of faults.

The results obtained in this research suggest that a typical commercial computer based on a processor with some simple error detection techniques and using memory access error detection techniques like the ones provided by a memory management unit will behave according to the fail-silent model for about 90% of the faults.

References

- [1] D. Powell et. al, "The Delta-4 approach to dependability in open distributed computing systems", Digest of Papers, The 18th International Symp. Fault-Tolerant Computing, Japan, June 1988.
- [2] F. Cristian, H. Aghili, R. Strong, D. Dolev, "Atomic broadcast: from simple to message diffusion to Byzantine agreement", Proc. 15th Fault-Tolerant Computing Sym., Ann Arbor, USA, June 1985, pp. 200-206.
- [3] J. Bartlett, J. Gray, B. Horst, "Fault tolerance in Tandem computer systems", in Dependable Computing Fault-Tolerant Systems, vol. 1, Springer-Verlag, 1987, pp. 55-79.
- [4] F. Schneider, "Byzantine generals in action: implementing fail-stop processors", ACM TOCS, vol. 2, no. 2, May 84, pp. 145-154.
- [5] 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.
- [6] S. Kim e R. K. Iyer, "Impact of device level faults in a digital avionic processor", Proceedings of AIAA/IEEE Digital Avionics Systems Conference, San Jose, CA, Oct. 1988, p. 428-435.
- [7] G. Choi, R. Iyer, et. al. "A fault behavior model for an avionic microprocessor: a case study", Dependable Computing for Critical Applications, Springer-Verlag, 1990, p. 177-195.
- [8] M. Schuette and J. P. Shen, "Exploiting Instruction-Level Resource Parallelism for Transparent, Integrated Control-Flow Monitoring", Proc. of 21th Sympo. on Fault Tolerant Computing, 1991, p. 318-325.
- [9] A. Mahmood, "Concurrent Error Detection Using Watchdog Processors - A Survey", IEEE Transactions on Computers, vol. 37, No. 2, February 1988, p. 160-174.
- [10] U. Gunneflo, J. Karlsson, and J. Torin, "Evaluation of error detection schemes using fault injection by heavy-ion radiation", Fault Tolerant Computer Symp., FTCS-19, June 1989, p.340-347.
- [11] M. Namjoo, "Techniques for Concurrent Testing of VLSI Processor Operation", Proceedings of International Test Conference, Philadelphia, PA, Nov., 15-18, 1982, p. 461-468.
- [12] M. A. Schuette and J. P. Shen, "Processor Control Flow Monitoring Using Signed Instruction Streams", IEEE Transactions on Computers, vol. 36, No. 3, March 1987, p. 264-275.
- [13] J. Sosnowski, "Detection of Control Flow Errors Using Signature and Checking Instructions", Proceedings of 18th International Test Conference, Philadelphia, 1988, p. 81-88.
- [14] K. Wilken and J. P. Shen, "Continuous Signature Monitoring: Low-Cost Concurrent Detection of Processor Control Errors", IEEE Transactions on Computer-Aided Design, Vol. 9, No. 6, June 1990, p. 629-641.
- [15] H. Madeira and J. G. Silva, "On-line Signature Learning and Checking", Dependable Comp. for Critical Applications, DCCA-2, Spriger-Verlag, 1992, p. 394-420.
- [16] M. Namjoo and E. J. McCluskey, "Watchdog processors and capability checking", Proceedings of Int. Symposium on Fault Tolerant Computing, FTCS-12, CA, June 1982, p. 245-248.
- [17] K. Wilken and T. Kong, "Efficient memory access checking", Proceedings of 23th Symposium on Fault Tolerant Computing, 1993, p. 566-575.
- [18] H. Madeira, G. Quadros, and J. G. Silva, "Experimental evaluation of a set of simple error detection mechanisms", Microprocessing and Microprogramming, The EUROMICRO Journal, vol 30, Aug. 1990, North-Holland, pp 513-520.
- [19] A. Mahmood, D. Andrews and E. J. McCluskey, "Executable assertions and flight software", Proc. of AIAA/IEEE 6th Digital Avionics System Conf., Baltimore, Dec. 1984, p. 346-351.
- [20] D. P. Siewiorek and Robert S. Swarz, "The Theory and Practice of Reliable Design, Digital Press, Educational Services, Digital Equipment Corporation, 1984, Bedford, Massachusetts.
- [22] J. Cusick, et. al., "SEU vulnerability of the Zilog Z-80 and NSC-800 microprocessors", IEEE Transactions on Nuclear Science, Vol. NS-32, No. 6, Dec. 1986, p. 4206-4211.
- [21] P. K. Lala, "Fault Tolerant and Fault Testable Hardware Design", Prentice Hall International, New York, 1985.
- [23] J. Karlsson, U. Gunneflo, P. Lidén e J. Torin, "Two Fault Injection Techniques for Test of Fault Handling Mechanisms", Proceedings of International Test Conference, 1991, p. 140-149.
- [24] Y. Crouzet and B. Decouty, "Measurement of Fault Detection Mechanisms Efficiency: Results", 12th Int. Symp. on Fault Tolerant Comp. (FTCS-12), Santa Monica, Ca, 1982, pp. 373-376.
- [25] J. Arlat et. al., "Fault injection for dependability validation: a methodology and some applications", IEEE Trans. on Sof. Eng., Vol 16, No 2, Feb. 1990, pp. 166-182.
- [26] G. Miremadi, J. Karlsson, U Gunneflo, and J. Torin, "Two Software Techniques for On-line Error Detection", proc. of 22th Fault-Tolerant Computing Symposium, FTCS-22, 1992, p. 328-335.
- [27] R. G. Halse and C. Preece, "Erroneous execution and recovery in multiprocessor systems", Software and Microsystems, Vol. 4, No. 3, June 1985, p. 63-70.
- [28] A. Mahmood and E. J. McCluskey, "Watchdog processors: error coverage and overhead", Proc. of 15th Fault-Tolerant Computing Symp., Ann Arbor, June, 1985, p. 214-219.
- [29] J. Hennessy and D. Patterson, "Computer architecture: a quantitative approach", Morgan-Kaufman, ed, 1991.