

Experimental Evaluation of the Fail-Silent Behavior in Programs with Consistency Checks

Mário Zenha Relá, Henrique Madeira, João G. Silva

Department of Computer Engineering

University of Coimbra - Portugal

{mzrela, henrique, jgabriel}@mercurio.uc.pt

Abstract

Previous work has shown that using only simple behavior based error detection mechanisms invisible to the programmer (e.g. memory protection) the percentage of fail-silent violations can be higher than 10%. Since the study of these errors has shown that they were mostly pure data errors, in this paper we evaluate the effectiveness of software techniques checking the semantics of the data such as ABFT and Assertions to detect these remaining errors. The results of injecting physical pin-level faults show that these tests can prevent about 40% of the fail-silent model violations that have escaped to the simple hardware-based error detection techniques. Moreover, the analysis of the remaining errors has shown that most of them remained undetected due to short range control flow breaks. When very simple software-based control flow checking was associated to the semantic tests, the target system behaved — without any dedicated error detection hardware— according to the fail-silent model for more than 98% of all the faults injected.

Keywords: fail-silent behavior, software checks, ABFT, assertions, control flow checking, pin-level fault injection, dependability, fault-tolerance, experimental validation.

1-Introduction

A computer is *fail-silent* if it produces only correct results, i.e., if incorrect results are generated as a consequence of a fault then the computer will not output them [Powell88]. The fail-silent model plays an important role in the design of reliable distributed 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 the design of industrial control computers and embedded systems (in this field fail-silent systems are usually known as *integer systems* [Kirman87]), as it is important to prevent the output of erroneous commands to the physical process under control.

Traditionally, fail silent computers are implemented by using massive redundancy. Errors are detected by comparison of the results produced by duplicated modules (either hardware or software) and in case of disagreement the results are not output (e.g. messages are not sent in the case of a distributed system). The main problem of this approach is the very high overhead associated with the duplication (either by hardware or by software).

Recent work [Madeira94] investigated the possibility of achieving fail-silent behavior without incurring the high costs of duplication techniques. The proposed approach consists in a simplex (traditional) computer enhanced with a comprehensive set of low cost error detection techniques such as memory access error detection, control flow error detection, watchdog timer, illegal instruction detection, error capturing instructions, etc. The idea behind this scheme is 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. Fault injection results have show that the system behaved

according to the fail-silent model for 97.7% to 99.4% of the injected faults. Although this results are interesting, especially taking into account the low cost of the used error detection methods, their use is rather limited as they strongly rely on signature monitoring techniques, which cannot be incorporated in the systems based on the complex processors available today. In fact, the only chance of using signature monitoring techniques in complex processors is to include the specific hardware required by these technique inside the processor IC, and the fact is that none of the existing commercial processors are ready for this techniques.

In this paper the same basic approach for implementing low cost fail-silent nodes is investigated in a different direction. Unlike the research mentioned above, all the error detection techniques used in the present research to achieve fail-silent behavior exists in off-the-shelf computers. On the other hand, these error detection methods have been complemented by a set of software error detection techniques especially thought for the detection of data errors. The reason for this decision was the fact that most of the fail-silent violations observed have been caused by pure data faults, i.e., faults affecting only the manipulated data.

In the next section we describe the error detection mechanisms implemented. Section 3 is devoted to the fault-injection process. In the succeeding sections the results of the experiments are presented and discussed. Finally, some concluding remarks and ongoing work are presented in section 7.

2-Error Detection

Software Error Detection

From the research mentioned above it could be concluded that most of the faults that caused fail-silent violations are characterized by the following features:

- Affect mostly data cycles (read/write access to the data segment).
- Have a short duration (1 or 2 memory access cycles).
- Have been mainly injected in the data pins of the processor.

This results suggest that to raise even further the fail-silent behavior, error detection techniques based on the semantic verification of the data manipulated by the programs are required. Data manipulation and semantic checking need the notion of correctness, which can be provided by the use of *assertions*, an invariant relationship between the variables of a program, written as a logical statement and inserted at different points in the program [Andrews79], [Mahmood83], [Leveson83]. These assertions can be written from the program specification or using some property of the problem or algorithm.

There are several difficulties using this approach: assertions are not transparent to the programmer, which may be a serious limitation in many situations, namely due to the programming effort required. Moreover, it is felt that its effectiveness depends largely on a number of different factors such as the nature of the application, the programmer's ability, etc., very difficult to quantify, and requiring a deep understanding of each new application as it is developed.

An approach that guarantees the correctness of the data manipulated by the programs that overcomes this limitations is *Algorithm Based Fault*

Tolerance (ABFT) first introduced in [Huang84]. In this technique checksum encodings are embedded in the calculation for fault detection purposes. The encoding is done such that the application operations preserve the checksum structure. This checksum is verified by the application. If a difference between the stored and computed checksums is detected, a fault has occurred. This method has been particularly successful for matrix operations, but a number of ABFT schemes have been proposed in different computations (e.g., Fourier transforms [Malek85], [Reddy90], and matrix equation solvers [Luk85]). Since at the moment few algorithms using this method are available, one of the main problems using ABFT is its lack of generality. It is well suited to applications using very regular structures, thus its applicability is limited to this type of problems.

For problems such as the generation of random numbers where it is impossible to find feasible checks based on the semantic value of the data, reexecution remains as the only practical error detection that can be used, based solely on software.

Since very different software checks can be used, depending on the problem under study, and in order to obtain meaningful results we used a set of tests representative of the different categories of software checks that can be used:

- Checks based on the regularity of the structures used.
- Checks based on the external specification of the problem.
- Reasonableness checks on the results produced.
- Checks based on the internal structure of the code.

- Reexecution when no other approach is feasible.

Each of these tests has been applied to a different benchmark reflecting the specific programming situation where it would be more appropriate. The program executed is formed by these benchmarks (Figure 1). The benchmarks and each particular software check are presented in Table 1 and described below.

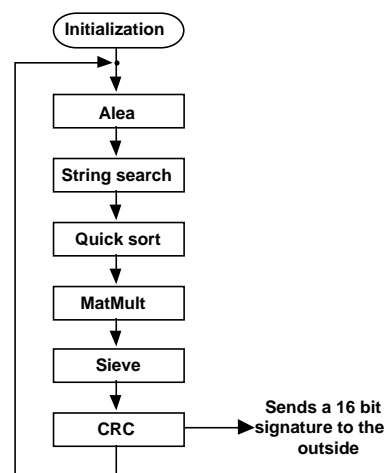


Figure 1-Program executed by the target system

- **MatMult** is a matrix multiplication program. This is the typical situation where very regular structures and operations are used. For error detection purposes we used *Algorithm Based Fault Tolerance* as described in [Huang84].
- **QuickSort** is a classic sorting algorithm. The software check used was a test based on the external specification to assure that the input array was correctly sorted at the end of the routine. More effective tests could have been used such as the order-sum assertion [Saxena94] but we favoured simpler and thus faster tests.
- **Search** is a simple searching algorithm. It looks for a given string in the global array where all the previous results are stored. The position where the string is found, or a “-1” if not

<i>Benchmark</i>	<i>Description</i>	<i>S/W Check used</i>
Alea	Random number generation	Reexecution
MatMult	Matrix Multiplication	ABFT
QuickSort	Sorting of an array	($a_i \geq a_{i+1}$)
Search	Find string in an array	(if FOUND confirm)
Sieve	Prime number generator	(code based)
Crc	Crc generation of result	Reexecution

Table 1-Benchmarks and their associated software checks

found, is appended to this array. Reasonableness checks such as checking the index found against the string limits (or "-1") and confirmation that the reference string is effectively at the index location were the tests applied.

- **Sieve** is the *Sieve of Erathostenes* prime number generator. In this case a set of simple low-level tests based on invariants found in the routine variables was used.

- **Alea** is a random number generator. Since it is impossible to find feasible checks on the semantic value of the data, (the algorithm is essentially an arithmetic expression), we used reexecution for error detection.

- **CRC** calculates the cyclic redundancy check of an array with the output from all the benchmarks used in each experiment. Since this algorithm is also an arithmetic expression we use reexecution. This routine is a special case in the pool of benchmarks since it is used to generate the final signature of the results produced by all benchmarks calculated at the end of each program cycle. This signature (considered as the final result) is sent to the outside by a parallel output port. Since the signature corresponding to correct results is known in advance it is easy to check whether the signature is correct 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. The circuit that checks the result signature also detects the situation in which the target system does not output the signature (system crash).

Since the benchmarks and their associated software checks are running on a physical machine, they benefit from the intrinsic hardware error detection capabilities. It must be noted that since this bottom error detection mechanisms cannot be deactivated, only *incremental* coverage figures can be obtained with any software based error detection technique. We shall now describe this bottom hardware error detection layer.

Hardware Error Detection

In this research the target system used is a commercial VME bus computer based on the MC68000 processor¹. All the hardware based error detection is obtained solely from the intrinsic behavior checking that is performed by the computer as it executes. This raw error detection is present in every modern computer, thus no dedicated error detection hardware is required. For the target system under study these mechanisms are the following:

- **μP-** MC68000 processor built-in error detection mechanisms. This processor has several internal error

¹Force® SYS68K/CPU-6.

detection mechanisms [Osborne83]. The most relevant are 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 typical 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 (i.e., not physically present).

- **WDT** - Watchdog timer. Traditional implementation of a WDT by means of a hardware programmable timer.

Since the memory access error detection (MEM) strongly depends on the memory usage it is important to know what was the memory map. It is shown in Figure 2. It will be essentially the same in the succeeding experiments since software checks in general involve only a slight memory overhead.

3.Fault Injection

Experimental evaluation by physical fault injection has become an attractive way of assessing the effect of faults in computers and validating specific fault tolerance mechanisms. Among the various techniques available for physical fault injection (heavy-ion radiation [Gunnflo89], power supply disturbances [Miremadi92], and pin-level [Arlat89]) we decided to use pin-

level fault injection as it enables great control in the fault injection process

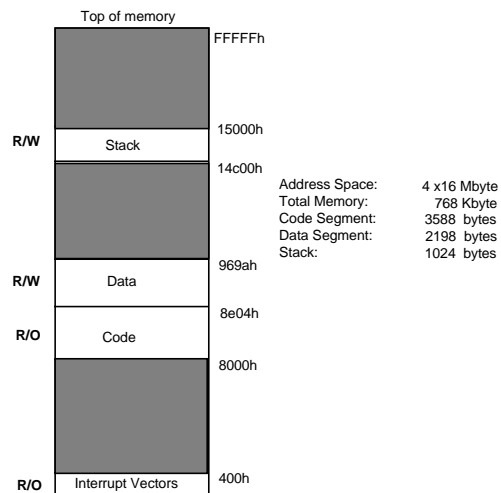


Figure 2-Target System memory map (not to scale).

and allows the injection of repetitive faults (the same fault always cause the same impact on the target system). This latter feature is very important as is often necessary to repeat the injection of specific set of faults in order to understand the impact of some "odd" faults on the target system. In this research we used the RIFLE pin-level fault injector. This tool was already presented in detail in [Madeira94], thus only a very short description is provided here.

The RIFLE Fault Injection Tool

RIFLE is a pin-level fault injector capable of injecting faults in relatively complex processors. 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

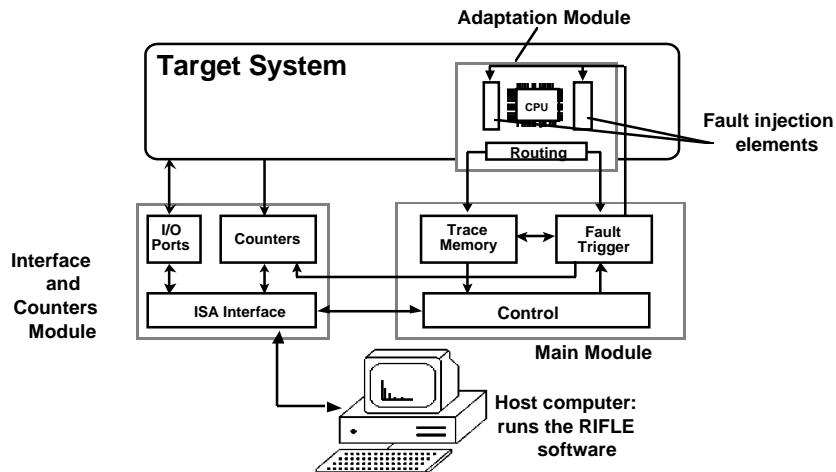


Figure 3 - RIFLE organization.

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 architecture of RIFLE is shown in Figure 3. 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 [Arlat 90] (i.e., the signals are intercepted by an each injection element which 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 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.

Interface and counters module: establishes the interface with the

RIFLE host computer and contains the circuits required to gather latency results and binary results (events).

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)². Furthermore, the error propagation analysis performed by the RIFLE software can detect some non effective faults. One example is the case of faults that cause the processor to fetch an incorrect instruction which is later

²It should be noted that not all the injected faults produce pin level errors. For example, the injection of a stuck-at-zero fault in a pin that is already at zero during all the time the fault is injected does not cause any error.

Criteria used for the definition of sets of faults	
Fault Activation Parameters	Chosen at random on the used segments.
Faulted pins	All processor pins, selected at random.
Number of pins affected by each fault	1 pin only.
Fault Duration	1 to 4 memory access cycles.
Type of faults	Bit inversion.
Number of faults injected	(specified for each experiment)

Table 3-Fault-model used in the experiments.

discarded from the prefetch queue without ever being executed. This kind of faults must be accounted for in most of the cases in order to obtain a correct assessment of the fault-tolerance technique under evaluation.

The Fault-Model

Several studies indicate that even in optimum environments more than 90% of the physical faults in computers are transient [Siewiorek84, Lala85]. Furthermore, the prevalence of transient faults over permanent ones is increasing due to smaller device dimensions and lower energy levels.

Since the objective of fault injection is to emulate actual faults (or cause the same error manifestations), it seems obvious that the injected faults should emulate transient faults. From the point of view of pin-level fault injection, a transient fault is a very short duration fault. The shortest fault lasts for one memory cycle since faults with shorter duration cause the same error manifestations as one memory cycle faults or do not cause any error at all.

We injected faults only in the processor pins since these faults model not only the internal processor faults but also faults in other parts of the target system, such as memory faults, bus faults, and most of the faults in peripheral devices.

While there are strong limitations in understanding the external

manifestations of the transient processor faults, there is some support [Gunnflo89], [Karlsson91] to the general criteria used for the definition of sets of faults in the present research (Table 3).

The faults are injected randomly during the program execution thus reflecting its dynamic profile (i.e. more frequently accessed code/data is more prone to faults).

All the processor pins are equally likely to be affected by the faults and only one pin at a time is affected by each fault. We decided to use single pin faults because these faults are in general more difficult to be detected than multiple pin faults, so that we work on conservative figures.

All the faults have been injected by inversion (the state of the faulted pin is inverted during the injection of the fault) . This fault type assures that all faults cause bit flips, thus speeding up the fault injection process.

Previous fault injection works on the evaluation of error detection techniques such as [Gunnflo89] and [Lomelino86] 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 [Karlsson93]. A great effort has been made in the last years to reach a generic methodology to define fault-injection experiments, such as the work done at LAAS-CNRS [Arlat90].

We do not ignore that when a different set of benchmarks is used, we face a different system. However, the large number of faults injected, the number of experiments and the regularity of the observations support our confidence on the representativeness of the results.

4-Fault tolerance capabilities of software checks

In this section we present the fault injection results obtained for the target system where the software checks described in section 2 have been added to the reference benchmarks.

Classification of erroneous behavior

The fault injection results have been classified according to five possible outcomes depending on the recorded behavior after each specific fault is injected.

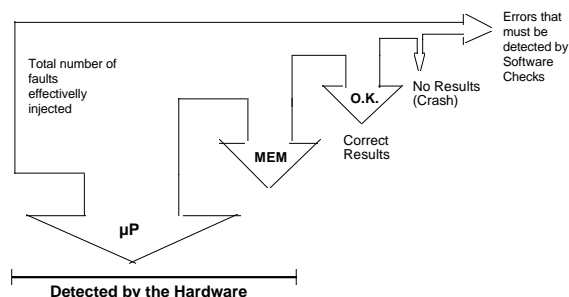


Figure 4-Possible outcomes after a fault is injected

• Detected by the Hardware (μP+MEM)

Errors that have been detected by the error detection layer provided only by the hardware. This bottom layer is composed of the processor built-in error detection mechanisms (μP) and

the memory protection features (MEM).

• No Results (system crash)

After the fault the system crashed and did not output any result. Although these errors are the ones detected by the Watchdog Timer (WDT) and could be considered in the previous class (μP+MEM), we decided to distinguish them since this behavior corresponds to a conceptually different outcome.

• Correct Results (system not affected)

The fault did not affect the results. The manual inspection of some of these faults (by observing the behavior information recorded in the RIFLE trace memory) has shown several situations:

- i. Faults whose errors are neutralized by the next instruction.
- ii. Faults affecting the execution of instructions that do not contribute to the benchmark results (some of these "redundant" instructions are due to inefficiencies of the compiler).
- iii. Faults whose errors are tolerated by the semantics of the benchmark under execution. For example, in programs such as the string search or the quick sort, which performs a large number of comparisons, many errors affecting only one of the operands in a comparison did not change the comparison result.

• Detected by software

Errors that have been detected by the error detection code added to the benchmarks. Note that the software can only detect errors if they have not been previously detected by the hardware layer since whenever an error is detected by the hardware (μP+MEM), the system halts.

• Fail-silent violation

Faults that escape all error detection mechanisms and cause the computer to

	<i>Base System Coverage</i>	<i>Coverage with software checks</i>	<i>Cumulative Coverage</i>
μ P+MEM	72.6%	76.2%	76.2%
Correct Results	18.2%	15.9%	92.1%
No Results (Crash)	0.1%	0.4%	92.5%
Detected by the S/W checks	n/a	2.1%	94.6%
Fail-Silent violation	9.1%	5.4%	7.5% (100%)
Fail-Silent Behavior	90.9%	94.6%	

Table 5-Fault-Tolerance due to embedded software checks

produce wrong results, thus violating the fail-silent model.

If an error is detected after a wrong result is output, it is considered as violating the fail-silent model and not as having been detected. It must be clear that this taxonomy involves a time factor: every fault is included in the class that has occurred first (with the lower latency). A "Correct Result" occurs whenever the *CrashTime*³ considered for the experiment is reached, results have been output, no detection method has been triggered, and no wrong signatures have been produced.

Fault injection results

All the fault-injection experiments described in this paper have been performed according to the fault model presented in section 3. Since some fault parameters are different for each experiment, they are specified for each case. For the current experiment (*Asserted*) the fault parameters are presented in Table 4.

Table 5 shows the coverage figures that have been measured, depicted

³It was experimentally verified that if the target system does not send the results (signature) within a period of time after the fault is injected that means that the system has crashed and it will not output results anymore. This time (designated "*CrashTime*") obviously depends on the benchmarks that are running. It is very important to set correctly this parameter since stopping an experiment too early leads to biased figures with a dominance of "no results" outcome.

graphically in Figure 5. The first point to observe is the slight coverage increase provided by the bottom hardware layer. This means that by simply adding code for error detection there is a variation in the raw hardware coverage⁴. This "observer's effect on the observed system" is a usually overlooked point whenever figures for non transparent error detection techniques such as software-based are considered. That is why we present in the first column the figures for the reference (*Base*) target system (without any software devoted to error detection).

<i>Experiment Identification</i>	<i>Asserted</i>
Number of faults	9000
Not Injected	639
Discarded	387 (4.6%)
Effective faults	7974
Duration	1-4 memory cycles at random
Crash Time	5 sec.

Table 4-Fault parameters for the *Asserted* experiment

From this data it is clear that the universe of faults that ought be detected by the semantic software checks —after they are included— is not the 9.1% observed in the *Base* system, but instead 7.5% (2.1%+5.4%). Nevertheless we cannot

⁴In fact, more code has also been added namely the asynchronous output of the system's internal status information for external monitoring.

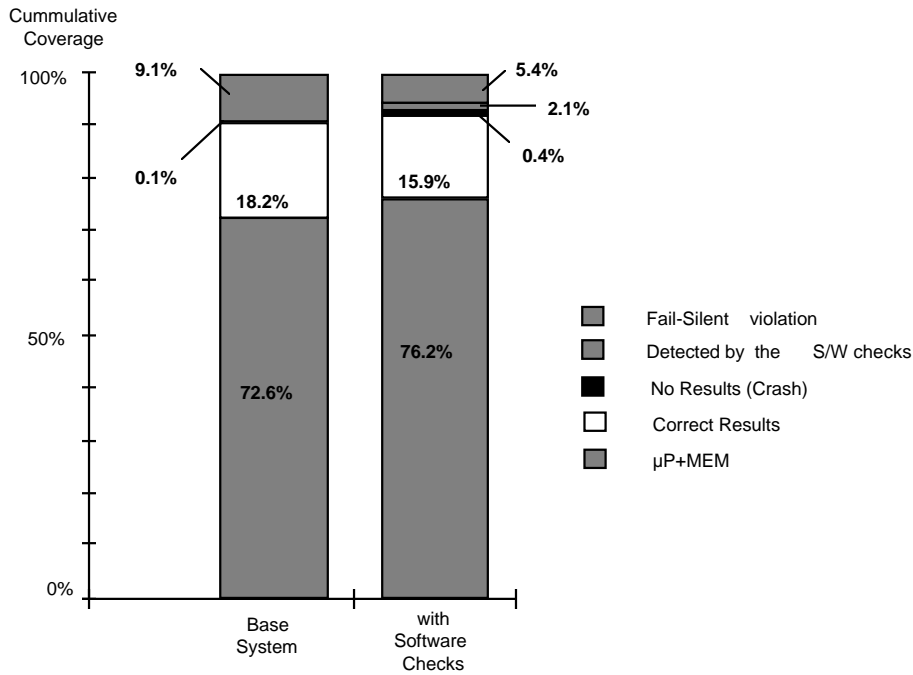


Figure 5-Incremental coverage in a system with embedded software checks

measure the observed 2.1% coverage against the total 7.5% faults that could have lead to fail silent violation in the *Asserted* system, but rather against the original 9.1%.

This 2.1% incremental coverage of the software checks leads to a system that behaves according to the fail-silent model for 94.6% of the injected faults. This figure represents a remarkable 40.7%⁵ decrease in the fail-silent model violations since the original *Base* system followed the model in 90.9% of the injected faults. This reduction confirms our argument that a large number of the faults escaping the inherent hardware error detection, produce a large number of data errors. The overheads measured (Table 6) can be considered reasonable for the reduction in number of fail-silent violations observed.

Due to its low absolute coverage figures (around 2%) software checks

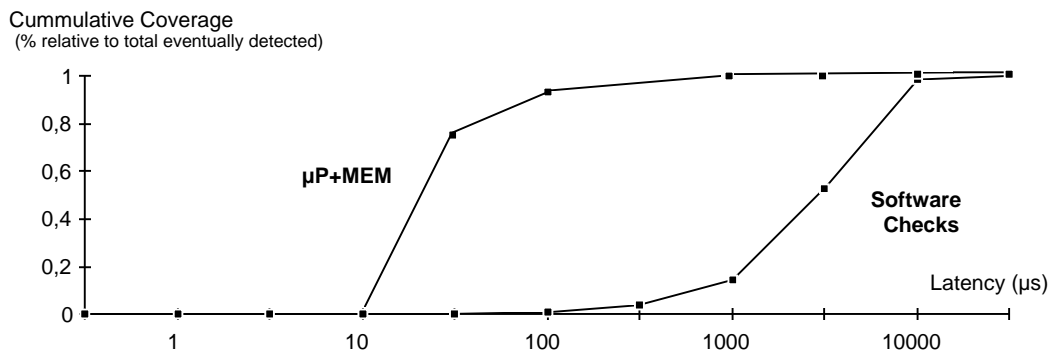
do not seem appropriate as a primary error detection mechanism. It is as a complement to other behavior-based error detection mechanisms that their potential emerges, since they are very effective in detecting most transient errors. Its importance derives from the fact that all these faults, if undetected, would almost certainly lead to fail-silent violation. The presence of the software checks reduced the fail-silent violations by about 40%, dropping its number to 5.4% of all faults effectively injected.

<i>Overheads</i>	<i>Asserted</i>
Time (performance)	+14%
Space (memory)	+10%

Table 6-Overheads in the *Asserted* experiment

In Figure 6 the cumulative coverage versus latency results are presented for the two main error detection layers under comparison, the hardware (μ P+MEM) and software checks. This experimental data quantifies the "time lag" of software error detection as

⁵(9.1%-5.4%)=3.7% in 9.1%.



.c2.Figure 6-Comparison of Cumulative Coverage versus Latency for μ P+MEM and Software Checks

nearly two orders of magnitude greater than what has been observed for the hardware: 1ms after the faults have been injected, assertions have only detected 20% of all the errors they will eventually detect, while 100 μ s after the fault the hardware has already detected almost every error it detects.

These measurements confirm the intuitive notion that the inherent hardware error detection has much lower latency than the executed assertions. While the hardware mechanisms are monitoring the system behavior concurrently with its execution, the software checks are exercised only at specific points in the program, thus tend to last longer to detect errors.

This point is particularly relevant to real-time systems where timeliness, as much as correction, is a primary goal. Also in *transaction* systems [Randell75] the correct estimation of error latency is a critical issue, to ensure that checkpoints are not corrupted. These are systems where, typically, a consistency check is made before a transaction commits.

Very few faults detected by software had shorter latency than those detected

by the hardware⁶. As we have shown above those were mainly data errors, that could have not been detected by the hardware since they didn't affect the system's behavior. This observation discloses much about the erroneous behavior of systems: whenever an error is active within a system, it is almost immediately detected by the hardware. If this doesn't occur, then the error has been absorbed by the intrinsic redundancy present in the system, entered into dormancy (e.g., embedded in the control information stored in the stack) or has been incorporated into the data.

These results have a direct application on checkpointing procedures. They suggest that the system status must be exercised/minimized before checkpoint data is stored. This procedure is more comprehensive than the common "acceptance test" before the *commit*, since it is applied both to the application and system⁷. This procedure has the net effect of reducing the error latency: latent errors

⁶For this experiment (*Asserted*), only 3 faults were detected with shorter latency.

⁷An example: if the programming language used is "C", the "commit" procedure should be called as close as possible to "main" in the function calling hierarchy to empty/reduce the stack.

may show up, so that any disturbance that might have affected the system before the error became latent will not be *committed*. It is not advisable to simply "flush" the system's state (and any possible latent error), since we would be clearing the symptoms, not the problem.

5-ABFT as an ideal software check⁸

In the previous section we have evaluated the fault-tolerance capabilities of a system where a set of software checks has increased the intrinsic coverage provided by the hardware. However, despite these two layers of error detection, about 5.4% of all faults remained undetected leading

protection unit) and do not seem to affect the semantic value of the data manipulated (so assertions are not triggered). However, these faults are responsible for the remaining fail-silent violations observed.

We have already pointed out that many factors affect the effectiveness of software checks, namely the programmer's ability to implement effective tests. In this section we evaluate separately Algorithm Based Fault Tolerance, since this technique has an intrinsic high error detection potential, very close to 100% as has been analytically demonstrated for matrix multiplication in multiprocessors [Huang84]. Using this special case of software error detection, we were able to decouple the intrinsic limitations of software checks

<i>Experiment Identification</i>	<i>MatMult</i>	<i>ABFT</i>
Number of faults ⁹	15 000	15 000
Not Injected	1 858	3 849
Discarded	619 (4.7%)	475 (4.3%)
Effective faults	12 529	10 676
Duration	1-4 memory cycles at random	1-4 memory cycles at random
Crash Time	15 sec.	15 sec.

Table 7-Fault parameters for *MatMult* and *ABFT* experiments

to fail-silent violations. It is manifest from these results that some faults originate errors that do not violate the processor execution model (thus do not activate the processor built-in error detection mechanisms), do not cross the memory segment boundaries (so remain undetected by the memory

from other unrelated factors.

To compare the effectiveness of Algorithm Based Fault Tolerance applied to matrix multiplication, we use as a reference the fault-injection experiment *MatMult* where only the matrix multiplication benchmark — without any error detection software— is executed. Then, we compare the results with those that were obtained in another fault-injection experiment ("*ABFT*") where Algorithm Based Fault Tolerance was added to the Matrix Multiplication code.

⁸Não gosto nada do título desta secção. Sugestões?

⁹Different number of faults were used in the experiments. In the first runs 3 series of 5000 faults were injected (one series for each set of bus lines: Data, Control and Address), but along the study we observed that the results stabilized after 3x3000 faults. Fewer faults injected meant shorter injection times, less disk space and less computation time to obtain the results.

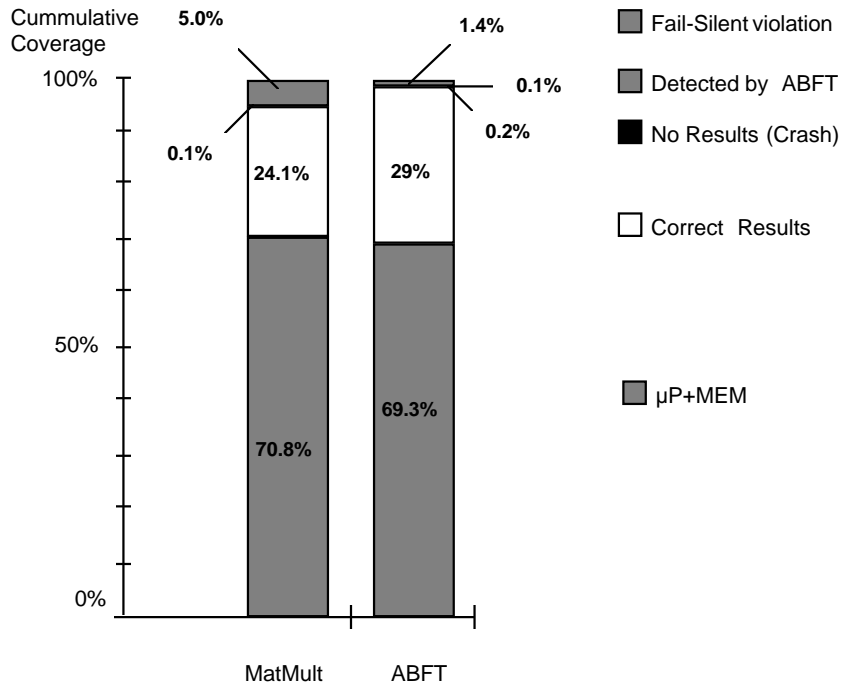


Figure 7-Incremental coverage in a system with ABFT

A marginal 0.1% coverage was obtained by the ABFT technique. However, only 1.4% fail-silent violations have been observed. This algorithm has a dramatic impact on the overall system behavior that indirectly prevents the occurrence of fail-silent violations.

From Table 8 it is clear that this effect is mainly due to the variation on "Correct Results" that has risen to almost 30% of all effectively injected faults. This result can be explained by the increase of code and data (+7.3%) that represent an increase of "redundant" information without a

direct effect on the results. Since the program signature is generated from the manipulated matrices only (not including the additional checksum elements), many errors that have affected this redundant data didn't affect the results whenever they were not used anymore.

Since the theoretical coverage of ABFT for the uniprocessor case should be close to 100% (99.95% according to [Huang84]), we shall now take a closer look on the reasons for the low coverage figures observed experimentally.

The structure of the MatMult algorithm with ABFT is depicted in

	<i>MatMult</i>	<i>ABFT</i>	<i>Cumulative ABFT</i>
68K+MEM	70.8%	69.3%	69.3%
Correct Results	24.1%	29.0%	98.3%
No Results (Crash)	0.1%	0.2%	98.5%
Detected by ABFT	n/a	0.1%	98.6%
Fail-Silent violation	5.0%	1.4%	(100%)
Fail-Silent Behavior	95.0%	98.6%	

Table 8-Fault-Tolerance using ABFT as the only error detection mechanism.

<i>Overheads</i>	<i>ABFT experiment</i>	
Space (code)	+20%	Total: ≈7%, decreasing with matrix size
Space (data)	one additional row and column	
Time (performance)	+5%, decreasing with matrix size	

Table 9-Overheads of applying ABFT to matrix multiplication.

Figure 8. The first matrix to be multiplied is randomly generated. Then the other matrix is generated by transposition of the first matrix. Only when the checksums are added to both matrices do we have code related to the ABFT detection. Afterwards the two matrices and their additional row and column are multiplied. The resulting matrix checksums are then computed and tested. Any error during this process ought to be detected with an extremely high theoretical coverage, but from then on the generation of the program signature (the crc to be output) and the output process itself cannot be checked by the embedded ABFT code.

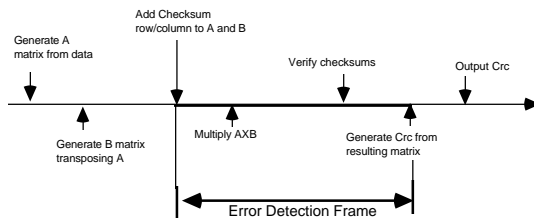


Figure 8-The internal structure of the ABFT benchmark

It is clear by observing Figure 8 that only errors generated inside the "error detection frame" can be detected by ABFT directly. Errors generated before that frame will lead to fail-silent violations but won't be detected since ABFT will be using corrupted data. On the other hand, data errors generated after this frame also can't be detected by ABFT, since it is not being used anymore, thus leading to fail-silent violations.

In fact we observed that only 10 errors out of the 161 leading to fail-silent violation have been injected inside the "error detection frame". This changes drastically the meaning of the figures observed, getting the observed fail-silent violations much closer to the theoretical figures.

The manual inspection of the remaining 10 faults revealed why they haven't been detected by ABFT: the errors generated lead to slight changes in the program control flow¹⁰. This deviation from the normal program execution was such that the checksum was not executed at all or in part (i.e., none or only part of the resulting matrix was checked). Despite this deviation the program executed the output routine and thus outputted erroneous data.

6-Combining control flow and software checking

The study of the faults undetected by ABFT strongly suggests that we must guarantee that software checks are effectively executed, otherwise its error detection capabilities will be useless. This points to the need of using some kind of control flow checking that guarantees that if any pertinent assertion is not executed, this will be flagged as an error.

In this section control flow checking will be added to the *Asserted* system

¹⁰The behavior of 2 out of these 10 faults was not clearly understood, since they lead to such a drastic change in the control bus signals that it was impossible to trace the program execution.

first presented in section 4. Our purpose is to evaluate the effectiveness of combining control flow and semantic checking¹¹.

Control flow error detection

Signature analysis seems to be the most general and effective approach for control flow error detection. The basic idea is to partition the application program in blocks. For each block (also named branch-free interval) there is only one valid sequence of instructions¹² and it is possible to compute a deterministic signature as a function (e.g. a Linear Feedback Shift Register [LFSR]) of the instructions within the block. At runtime the signature of each block is calculated and errors can be detected by comparison of the runtime signature to the precomputed signature. In the current experiments we shall use software based control flow checking. This follows from our initial requirement of using only the available hardware on typical commercial computers. We used an assigned signature monitoring approach [Lu83; Miremadi92] in which a new signature is added through crc convolution immediately after the execution of assertions. The correctness of signatures is only checked before the result is output. This evolves naturally from our approach to check the fail-silentness: since the final result is a crc of all the data manipulated by the program, it is only after that instant that an error becomes a failure.

¹¹We didn't apply it to the *MatMult* benchmark since ABFT alone does not represent a typical application where different control structures are used. It was used simply as a way to decouple the intrinsic lack of coverage of different software checks from other factors (namely the control flow errors as was observed).

¹²Different ways of defining blocks have been proposed.

Therefore that is the point where the control flow check is more effective with the least overhead involved, at the cost of a higher error latency.

This control flow checking mechanism involves a 1.5% performance and 4.1% memory overheads (in table Table 10 are shown the total overheads involved against the initial *Base* system). In most control-flow checking techniques one of the main problems is the test granularity that should be used. If every single execution block is checked, the coverage is good, but the overhead are intolerable. As the tests are more sparsely used the overheads, but also the coverage, decreases. When control flow checking is associated with assertions this problem is naturally solved from its very start: signature justification (or verification) is executed whenever assertions do. Since a new signature is generated only when a software check is performed, the number of blocks (thus valid signatures) is extremely low if compared with the approaches where the basic block is considered at assembly level, thus the overheads introduced are negligible.

<i>Overheads</i>	<i>Assert Locked</i>
Time (performance)	+15.5%
Space (memory)	+14%

Table 10-Overheads of *Assert Locked* relative to *Base*

In this experiment faults have been injected according to the same fault model as before, with the parameters presented in Table 11.

The fault injection results presented in Figure 9 and Table 12 confirm our argument that most of the remaining fail-silent violations were due to small control flow deviations.

From the previously observed 5.4% in the *Asserted* experiment, we have now

	<i>Base System</i>	<i>Asserted</i>	<i>Assert Locked</i>
μ P+MEM	72.6%	76.2%	69.0%
Correct Results	18.2%	15.9%	24.1%
No Results (Crash)	0.1%	0.4%	0.1%
Detected by the S/W checks	n/a	2.1%	5.0%
Fail-Silent violation	9.1%	5.4%	1.8%
Fail-Silent Behavior	90.9%	94.6%	98.2%

Assert: 1.9%
CFC : 3.1%

Table 12-Fault Tolerance when CFC is combined with Assertions

a remarkable figure of 1.8% fail silent violations. This is what remains from the initial 9.1% fail-silent violation figures when only the intrinsic hardware error detection was present. Note that control flow checking reduced the number of errors detected by assertions only (i.e. some errors detected by assertions were due to control-flow errors).

Table 11-Fault parameters for the *Assert Locked* experiment

From these experimental results it is obvious that assertions should always be associated with some sort of control flow checking. In this way, whenever any result is to be output the system first checks if every pertinent assertion has been executed. This would prevent fail-silent violations due to an erroneous jump over the assertion check.

The 1.8% fail-silent violations observed (a total of 250 faults) are a remarkable figure if we consider that no special hardware has been used. Similar figures could only be obtained through special signature monitoring

<i>Experiment</i>	<i>AssertLocked</i>
Number of faults	15 000
Not Injected	671
Discarded	1411
Effective faults	13598
Duration	1 to 4 memory cycles
Crash Time	5 sec.

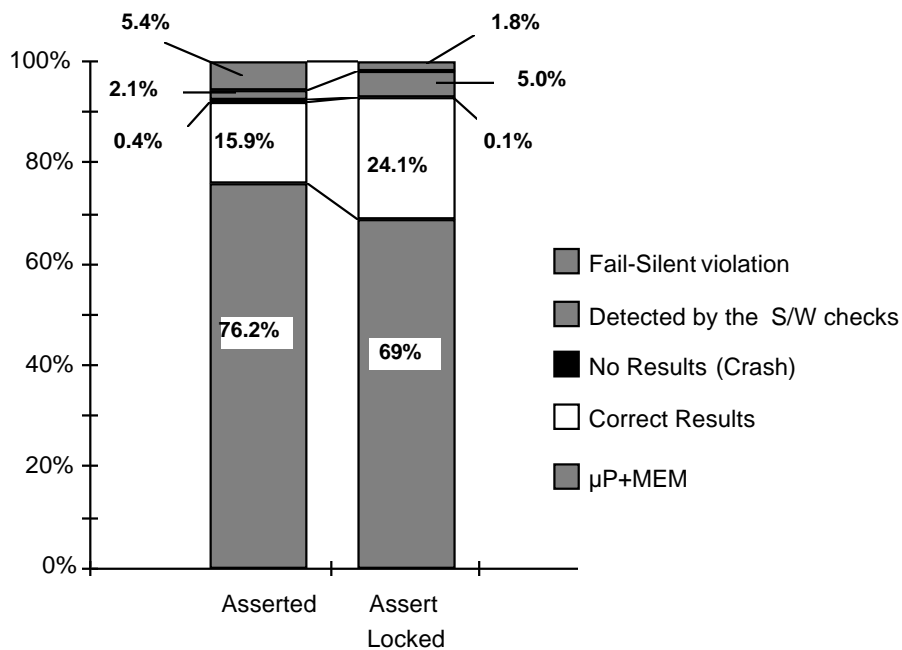


Figure 9-Fault Tolerance when CFC is combined with Assertions

hardware [Madeira94].

Characterization of the final undetected faults

By manual inspection of the information stored in the RIFLE trace memory, we could understand the type of errors generated by the remaining faults leading to fail-silent violation.

i. Intrinsic limitations of the tests

It was found that despite the correct execution of the assertions on the manipulated data, some of the errors have not been detected due to intrinsic lack of coverage of the checks used (e.g., in the quick sort algorithm the final vector was sorted, but a wrong element was present. This wrong element had been fetched from an erroneous memory cell due to the fault). If more sophisticated algorithms had been used such situation would have been detected, but it is not possible to preview every possible error situation¹³, thus we can expect that the occurrence of undetected errors will always occur. What should be retained is the nature of undetected errors in order to keep them as low as possible to obtain useful dependability figures.

ii. Malicious errors

Another very different situation that lead to fail-silent violation was due to the malicious behavior caused by some faults. They generated errors that lead to incorrect instruction fetch. However, the corrupted instruction didn't affect the normal execution (thus it was not detected by any of the error detection mechanisms including the software checks), but simultaneously corrupted previously checked data.

Situations have been observed where the semantics of the high level code

¹³However, a clear understanding of the erroneous behavior of systems under the influence of faults is a prerequisite in the design of error injection tools.

was fulfilled with simultaneous corruption of previously checked data. It was remarkable to observe that even reexecution and subsequent comparison of the results from both runs was not immune to this type of fail-silent violation. An example better illustrates this situation. An array was being checked element by element. The fault mutated a bit masking instruction to a data write on a memory cell in the zone of the array that had already been successfully checked. Coincidentally, the bit masking instruction that was not executed was redundant, since the particular data item in the destination register would not have been modified. Thus, as far as the normal execution path was concerned, it was as if no error had occurred at all. However, the affected memory cell remained corrupted, ultimately leading to fail-silent violation.

From these observations we may conclude that if the error coverage of a particular test is insufficient —no matter how rare an occurrence may be— if we have enough faults, it will happen¹⁴. The designer will have to account on this evidence to evaluate the dependability figures for its system. Even (or specially) when reexecution is used it must be clear that if "clean" data is available it can always be corrupted, hence all data must be protected before being checked. This caution also applies to the ABFT algorithm, because whenever the row and column checksums are being computed, the resulting matrix must be previously protected.

This protection can be perceived as an atomic comparison procedure: if the logical outcome is correct, it reflects

the status of the *whole* data being checked, not just part of it. Thus, an *atomic* software check on composite data requires that:

- i. The test must be effectively executed (e.g. ensured through some sort of control flow checking)
- ii. The data to be checked must first be write protected. This can be accomplished using memory protection.

7-Conclusion

In this paper we investigated the possibility of achieving a high degree of fail-silent behavior in ordinary computers, complementing their intrinsic hardware error detection mechanisms with a set of carefully chosen software error detection techniques. The nature of the faults undetected by the hardware suggested the use of semantic data checking, namely through the use of assertions. Thus, in the first part of the chapter we evaluated the characteristics of the faults detected by some common software checks. Several conclusions could be drawn from this study: due to its low absolute coverage figures software checks do not seem appropriate as a primary error detection mechanism. It is as a complement to other error detection mechanisms that their potential emerges, since they are very effective in detecting most transient errors. Its importance derives from the fact that all these faults, if undetected, would almost certainly lead to fail-silent violation. The presence of the software checks reduced the fail-silent violations by about 40%, dropping its number to 5.4% of all faults effectively injected.

The use of assertions as an error detection mechanism raises the problem of the dependency on the programmer's ability to implement

¹⁴Murphy's Law at its best!

effective software checks. In order to decouple the intrinsic limitations of the tests used from other factors that might affect its error detection capabilities, we evaluated a special class of software checks with an extremely high theoretical coverage, Algorithm Based Fault Tolerance (ABFT). From such study it was possible to unveil several factors that reduce the effectiveness of the tests, but cannot be attributed to their intrinsic limitations:

- Software checks have to be designed on a system basis, otherwise errors generated on unprotected software modules will make useless the high levels of protection provided by other (thought) more critical modules. In fact, no matter how effective a software check is, if it works on data corrupted by other system modules that have not been properly protected from errors. The same reasoning applies downwards: if the succeeding operations are not carefully protected, they will be able to corrupt "clean" data. This point is particularly relevant in the design of the Operating System or Run-time Kernels where the fault-tolerant application is being run, since unreliable system services may thwart the dependability objectives.

- Slight changes in the program control flow followed by resynchronization could not be detected by the ordinary memory protection. This observation indicate that there must be some way to guarantee that software checks are effectively executed. This suggested the potential of using simple control flow techniques to detect the remaining errors leading to fail-silent violation. In fact, after including such mechanism a mere 1.8% fail-silent violations have then been recorded (a total of 250 faults).

- Malicious errors may corrupted previously checked data while

performing the intended function at the high level language. This errors are innocuous to the normal program execution, but simultaneously affect dramatically its most relevant data. It can't be stressed enough that no matter how good a test may be, it can corrupt the data that is being manipulated, and simultaneously flag that data as correct. This situation is possible even when reexecution is used.

In the experiments described in this paper, very simple assertions have been used, and conservative assumptions were made at every design stage. Nevertheless, with overheads below 15%, our target system behaved according to the fail-silent model for more than 98% of all the faults injected, without any dedicated error detection hardware.

References

- [**Andrews79**] Andrews, D. M. "Using Executable Assertions for Testing and Fault Tolerance." *FTCS-9*, 1979.
- [**Arlat90**] Arlat, J., M. Aguera, et al., "Fault Injection for Dependability Validation: a Methodology and Some Applications." *IEEE Trans. SW Eng.*, Vol. 16, No. 2, pp.166-182.
- [**Arlat89**] Arlat, J., Y. Crouzet, et al. "Fault Injection for Dependability Validation of Fault-Tolerant Computing Systems." *FTCS-19*, Chicago IL, 1989. IEEE.
- [**Gunneflo89**] Gunneflo, U., J. Karlsson, et al. "Evaluation of Error detection Schemes Using fault Injection by Heavy-Ion Radiation." *FTCS-19*, 1989. IEEE.
- [**Huang84**] Huang, K.-H. & J. A. A., "Algorithm-Based Fault Tolerance for Matrix Operations." *IEEE Trans. Computers*, Vol. C-33, No. 6, Jun/84, pp. 518-528.
- [**Karlsson93**] Karlsson, J. "Fundamental Problems in Fault and Error-Injection Research." *IEEE Intl. Workshop on Fault and Error Injection for Dependability validation of Computer Systems*, Göteborg, 1993.
- [**Karlsson91**] Karlsson, J., U. Gunneflo, et al. "Two Fault Injection Techniques for Test of Fault handling Mechanisms." *Proceedings of International test Conference*, 1991.
- [**Kirrmann87**] Kirrmann, H. D., "Fault

Tolerance in Process Control: An overview and examples of European Products." *IEEE Micro*, Vol. No. Oct/87, pp. 27-50.

[Leveson83] Leveson, N. a. T. J. S. "Safety Assertions for Process-Control Systems." *13th FTCS*, 1983.

[Lomelino86] Lomelino, D. and R. K. Iyer. "Error Propagation in a Digital Avionic Processor." *RTSS-1986*, 1986. IEEE.

[Lu83] Lu, D. J., "Watchdog Processors and Structural Integrity Checking." *IEEE Trans. on Computers*, Vol. C-31, No. 7, Jul/82, pp. 681-685.

[Luk85] F. T. Luk, "Algorithm-Based Fault Tolerance for Parallel Matrix Solvers", *Proc. SPIE Real-Time Signal Processing VIII*, vol. 564, 1985, pp. 49-53.

[Madeira94a] Madeira, H., M. Rela, et al. "RIFLE: A General Purpose Pin-Level Fault Injector." *EDCC-1*, Berlin-Germany, 1994. Springer-Verlag.

[Madeira94b] Madeira, H. & J. G. S. "Experimental Evaluation of the Fail-Silent Behavior in Computers Without Error Masking." *FTCS-24*, Austin-Texas, 1994. IEEE.

[Mahmood83] Mahmood, A., E. J. McCluskey, et al. "Concurrent Fault Detection Using a Watchdog Processor and Assertions." *1983 Internation Test Conference (ITC)*, 1983.

[Malek85] Malek, M. a. Y. H. C. "A Fault-Tolerant FFT processor." *FTCS-15*, Ann Arbor, MI, 1985.

[Miremadi92] Miremadi, G. & J. K. & U. G. & J. T. "Two Software Techniques for On-Line Error Detection." *FTCS-22*, Boston, Massachusetts, 1992.

[Osborne83] 68000 Microprocessor Handbook. Berkeley-California 1983. OSBORNE/McGraw Hill.

[Powell88] Powell, D., P. Verissimo, et al. "The Delta-4 Approach to Dependability in Open Distributed Computing Systems." *FTCS-18*, Tokyo, 1988. IEEE.

[Randell75] Randell, B., "System Structure for Software Fault-Tolerance." *IEEE Trans. S/W Eng.*, Vol. SE-1, No. 2, Jun/75, pp. 220-232.

[Reddy90] Reddy, A. L. N. and P. Banerjee, "Algorithm-Based Fault Detection for Signal Processing Applications." *IEEE Trans. on Computers*, Vol. 39, No. Oct/90, pp. 1304-1308.

[Saxena94] Saxena, N. and E. J. McCluskey, "Linear Complexity Assertions for Sorting." *IEEE Trans. Software Eng.*, Vol. 20, No. 6, Jun/94, pp. 424-431.