

Experimental evaluation of a COTS system for space applications

Henrique Madeira <i>DEI-FCTUC</i> <i>Univ. of Coimbra</i> <i>3030 Coimbra</i> <i>Portugal</i> henrique@dei.uc.pt	Raphael R.Some <i>Jet Propulsion Laboratory</i> <i>Calif. Inst. of Technology</i> <i>Pasadena, CA 91109</i> <i>USA</i> Raphael.R.Some@jpl.nasa.gov	F. Moreira, D. Costa <i>Critical Software</i> <i>3030 Coimbra</i> <i>Portugal</i> dino@criticalsoftware.com	David Rennels <i>Univ. Calif. Los Angeles</i> <i>USA</i> rennels@cs.ucla.edu
---	--	--	--

Abstract

The use of COTS-based systems in space missions for scientific data processing is very attractive, as their ratio of performance to power consumption of commercial components can be an order of magnitude greater than that of radiation hardened components, and the price differential is even higher. A major problem, however, is that COTS components are susceptible to Single Event Upsets (SEU), which are transient errors caused by space radiation. This means that their actual use in space missions must be preceded by careful study on the impact of faults on system behavior and the identification of weak points that should be strengthened with specific software fault tolerance techniques. This paper evaluates the impact of transient errors in the operating system of a COTS-based system (CETIA board with two PowerPC 750 processors running LynxOS) and quantifies their effects at both the OS and at the application level. The results provide a picture of the impact of faults on LynxOS key features (process scheduling and the most frequent system calls), data integrity, error propagation (among application processes, from applications to the OS, and from OS to the application processes), application termination, and correctness of application results. The study has been conducted using a Software-Implemented Fault Injection tool (Xception) and both realistic programs and synthetic workloads (to focus on specific OS features) have been used. Results show that the impact of errors is very dependent on the application and on the LynxOS module in which they occur. A variable but significant percentage of wrong application results have been observed, which demonstrate that additional fault tolerance techniques are required to guarantee result correctness in the presence of faults.

Keywords: Experimental evaluation, COTS, fault injection, OS robustness

Submission category: practical experience

Word count: ~7000

The material included in this paper has been cleared through authors' affiliations

Contact author:

Henrique Madeira
DEI-FCTUC
Polo II, University of Coimbra
3030 Coimbra
Portugal

Phone: (+351) 239 790003
Fax: (+351) 239 790003
Email: henrique@dei.uc.pt

Carter award: NO

1. Introduction

The use of Commercial Off-The-Shelf (COTS) components (both hardware and software) and COTS-based systems in mission-critical and business-critical applications is a clear trend in the computer industry. They offer a real opportunity to reduce development costs and deployment times, which greatly explains the growing interest in using COTS components in fault tolerance architectures. Additionally, COTS components normally benefit from a large installation base in a multitude of configurations, which is often considered as an effective test in the field.

Following this trend, the use of COTS-based systems in space missions is particularly attractive, as the ratio of performance to power consumption of commercial components can be an order of magnitude greater than that of radiation hardened components, and the price differential is even higher. However, in spite of all these advantages, COTS are not usually designed for the stringent requirements of critical applications. Furthermore, the use of hardware COTS components in space applications introduces new challenges, as COTS hardware components are susceptible to transient errors due to Single Event Upsets (SEU) caused by space radiation. This means that the actual use of COTS components in space missions must be preceded by careful study of the impact of faults on system behavior and the identification of weak points that should be strengthened with specific software fault tolerance techniques.

The Remote Exploration and Experimentation (REE) Project at NASA's Jet Propulsion Laboratory was aimed at bringing COTS-based systems into space. The idea of REE project is to use a set of on-board COTS-based parallel processors for onboard scientific data processing [Some 99], enabling new classes of scientific missions and reducing the need for ground station operations through limited bandwidth communication links.

One key aspect behind the REE project is that the high-performance COTS-based systems are used for scientific data processing and not for spacecraft control. An external, radiation-hardened and independently protected Spacecraft Control Computer (SCC) is responsible for control, and it is also the overall controller of REE – sending it commands and tasks to execute. The SCC sits at the top of a fault-recovery hierarchy. It is responsible for detecting if the REE has ceased to provide results. It can provide a gross, last-ditch, repair function if it observes that REE is no longer responding or providing reasonable outputs. It is empowered to command self-tests of processor nodes and links, to unpower faulty nodes, and to reload and cold start the REE system. Of course, if this gross procedure must be invoked, processing outages in the order of tens of minutes will ensue. Therefore it is necessary to recover from a large majority of transient errors much more quickly within the REE system itself in order to provide acceptable system availability.

Other fault-injection studies by Ravi Iyer at UIUC based on fault-insertions into real REE applications and the supporting software implemented fault-tolerance middleware have shown that very high, but not perfect coverage in error recovery is possible, but this work did not include error insertions into the OS kernel. Their work simulated kernel failures by inducing crashes and hangs into various nodes and tabulating the response.

The emphasis of this work is on the system behavior when errors are induced into the operating system – looking at application termination, data integrity, and correctness of application results. This paper evaluates the impact of transient errors in the operating system of the REE testbed (CETIA board with two PowerPC 750 processors running LynxOS) and quantifies their effects at both the OS and at the application level. The results provide a picture of the impact of faults on

LynxOS key features (process scheduling and the most frequent system calls), data integrity, error propagation (among application processes, from applications to the OS, and from OS to the application processes), application termination, and correctness of application results. Both realistic programs and synthetic workloads have been used. The first ones are meant to evaluate the impact of faults on application termination and result integrity while synthetic workloads focus on the evaluation of specific OS features in the presence of faults.

A Software-Implemented Fault Injection tool [Xception 00] has been used to emulate the effects of SEU through the insertion of bit-flip errors in processor structures (registers, integer unit, internal processor buses, floating point unit, cache, etc) and memory whilst the OS is running. Faults have been injected using different distributions. Typically, a uniform distribution over time and location was used but we also have injected faults directly in specific locations and at specific moments (e.g., faults inserted into the registers when the processor is executing code from a given LynxOS call) to evaluate the impact of specific faults in the LynxOS behavior and try to identify their effects on applications programs or SIFT middleware that may be running.

The structure of the paper is as follows: the next section presents the experimental setup used in this study. Section 3 presents the different experiments and discusses the results and Section 4 summarizes the contributions and concludes the paper.

2. Experimental setup

2.1 Target system and Xception fault injector

Figure 1 shows the test bed layout used in these experiments. The target system is a COTS CETIA board with two PowerPC 750 processors and 128 Mbytes of memory, running Lynx operating system version 3.0.1. The host machine is a Sun UltraSparc-II with SunOS 5.5.1.

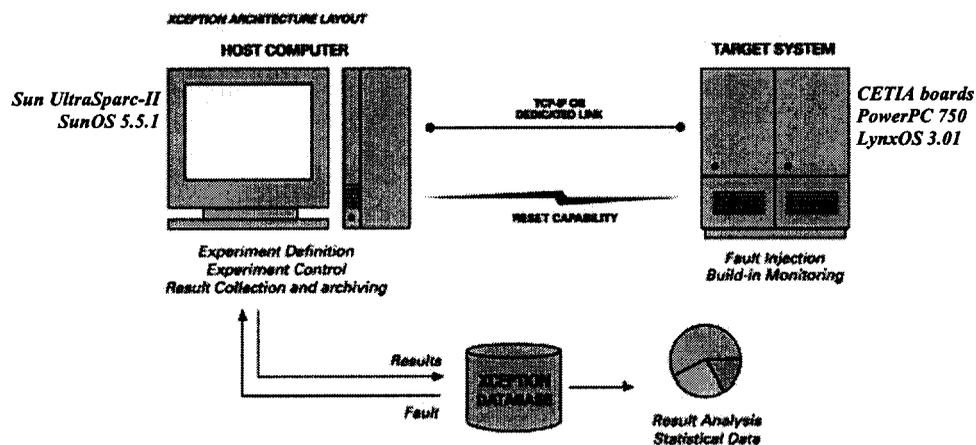


Figure 1 – Test bed layout

The fault-injection tool is Xception PowerPC705/LynxOS, which is a part of the Xception tool [Carreira 98] to the PowerPC705 processor architecture and LynxOS. Xception uses the debugging and performance monitoring features available in the processors to inject faults by software and to monitor the activation of the faults and their impact on the target system behavior. The target applications are not modified for the injection of faults and the applications are executed at full speed. Faults injected by Xception can affect any process running on the target system, and in this

work, we focus on the kernel code. Table 1 shows the target fault locations and basic fault triggers and fault/error types of Xception.

Fault Locations (target system)	Fault Triggers	Fault/Error Types
Integer Unit (IU)	Opcode fetch from a specified address	Bit level operations
Floating Point Unit (FPU)	Operand load from a specified address	Stuck-at-zero
Memory Management Unit (MMU)	Operand store to a specified address	Stuck-at-one
Internal Data Bus (IDB)	After a specified time since start-up	Bit flip
Internal Address Bus (IAB)	A combination of the above	Bit mask (32 bits)
General Purpose Registers (GPR)		Number of bits to be changed
Branch Processing Unit (BPU)		
Memory (M)		

Table 1 – Xception fault locations, fault triggers, and fault/error types.

The definition of the faults parameters, injection process, and collection of results is controlled by the host system. The key steps of the fault injection process are shown in Figure 2. The target system is restarted after each injection to assure independent experiments. Faults are injected after the workload start and, depending on the type of trigger, faults are uniformly distributed over time or are injected during the execution of specific portions of code. The collection of results (the actual location in the code where the fault was injected, the processor context at that moment, etc, which are stored in a small log in the target) is done after resetting the system to assure that the system is stable and can send the results to the host.

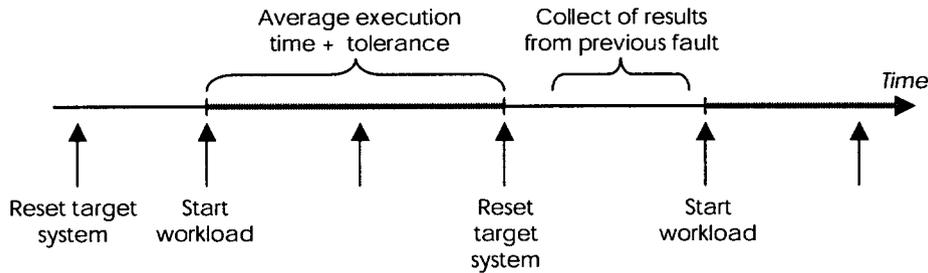


Figure 2 - Typical injection run profile

2.2 Fault model

Faults injected consist of single bit-flips injected in all the possible units that can be reached by Xception. The distribution over location is uniform and during the post-injection analysis we can isolate the faults in a given location, if needed. One important location (concerning space applications) that cannot be reached by Xception is the processor cache (for both the instruction and the data cache). However, faults in cache can be partially emulated by faults injected in the memory, provided that these faults affect memory areas where the processor is executing code or accessing to data, which can be easily achieved with the Xception fault triggers.

The fault triggers have been set up to insert a uniform distribution of faults over time and to inject faults when the processor is executing specific code. Uniform distribution of faults over time is particularly relevant because it matches the expected distribution of SEU whilst the OS is running. On the other hand, faults injected in specific locations, such as the code of specific operating system calls, are very important to evaluate the behavior of the operating system in the presence of specific faults.

2.3 Workload

2.3.1 Synthetic workload

As one major goal of this study is the evaluation of the impact of transient faults on key features of the operating system (LynxOS), we decided to define a synthetic workload in order to exercise core functions of the operating system such as the ones related to processes (schedule, create, kill, process wait), memory (attribute memory to a process, free memory), and input/output (open, read, write). The synthetic workload executes a given number of iterations and in each iteration of the cycle it starts by doing some buffer and matrix manipulations, just to use memory resources that will be checked for integrity later on, and then executes a number of system calls related to the core OS functions mention above (e.g., fork, kill, wait, open, read, write, etc). After each step, the program stores a footprint in a file. For example, after each system call the program stores the return code in the file; after each checksum calculation (performed over all data structures and application code) stores the value of the checksum, etc. At the end of each cycle interaction the program executes additional tests and at the end of the program the final result stored in the file (i.e., all the footprints) is compared with a golden version to check if any of the individual step outputs were wrong. The amount of memory allocated from heap for buffer manipulations is 1 Mbytes and the matrices manipulations (multiplication and other simple manipulations) are performed on three matrices of 250 x 250 integers define as static variables.

Three instances (each instance is a different process) of this synthetic program are used to test the effect of the kernel error on other processes in a multiprogramming environment. The first one (P1) is the one that is going to be used to inject faults. That is, faults are injected (in processor register, integer unit, memory, etc) during the execution of P1 code or during the execution of kernel code (either kernel code called by P1 or other system code). The processes P2 and P3 are used as “miner’s canaries” to evaluate error propagation from P1 to P2 or P3 (through the operating system) and the operating system response after a fault (section 3 presents in detail all the aspects evaluated using this configuration). Here we are concerned that OS errors may compromise the virtual address space or affect the process scheduler so that subsequent processes are not properly started. Figure 3 illustrates this scheme.

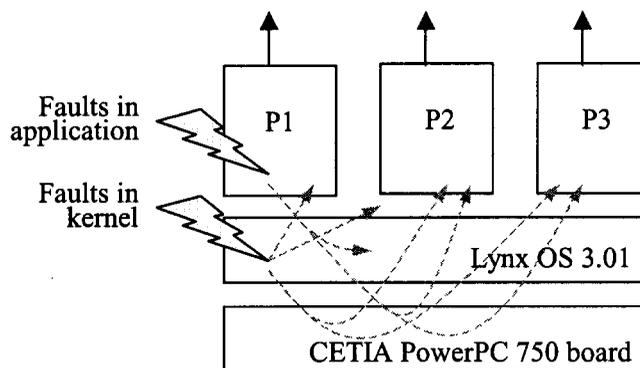


Figure 3 – Process configuration with the synthetic workload

2.3.2 Realistic workloads

The use of realistic workloads is particularly relevant to the evaluation of the effects of OS errors in order to study aspects such as application termination and result correctness in the presence of faults. Due to bureaucratic difficulties, we could not use real REE applications code, so programs

of a similar nature were substituted. Thus we choose three applications with quite different profiles concerning processor and memory needs, use of I/O system calls (especially disk read and write), and size of the results, in order to evaluate the influence of these aspects on the impact of the faults. The workloads are the following:

- **Gravity:** Calculates the trajectory of mass (e.g. a satellite) attracted by a bigger one (e.g. a planet), modeled by Newton's Gravity Law.
- **PI:** Computes the value of π (with a large number of decimal digits) by numerically calculating the area under the curve $4/(1+X^2)$.
- **Matmult:** Matrix multiplication program. In our experiments it multiplies two 400 x 400 integers and two 400 x 400 floating-point matrices.

3. Results and analysis

Since the general objective of this study (to observe the effects of transient faults on a COTS-based system) comprises the evaluation of the impact of faults on key aspects/components of the system, we have organized different sets of experiments aimed at covering the most relevant facets of the problem. In short, the goals of these experiments are the following:

- Experiments using the Synthetic Applications:
 - Evaluate the impact of faults injected while the processor was executing OS code associated with synthetic process P1 on the system behavior and on the OS ability to execute core functions related to processes, memory, and input/output (section 3.1.1). We have used a synthetic workload to be sure that all these OS functions are heavily used.
 - Evaluate the impact of faults inserted while the processor was executing code of the synthetic application process P1 and compare the effects of application faults vs. OS faults (section 3.1.2).
 - Evaluate the OS capability to confine errors to the memory and execution context of the process affected by the fault (section 3.1.3). In other words, the goal in this case is to evaluate the error propagation from OS to application processes and from one process to the others.
- Experiments using Realistic Applications:
 - Evaluate the impact of faults on application termination and the correctness of the application results (section 3.2). In this case, we have used realistic programs and uniform faults distributions to emulate as close as possible the effects of SEU errors in that occur when real applications and their associated OS functions are executing.

3.1. Impact of faults in the OS and error confinement & propagation: experiments with the synthetic workload

In this set of experiments we used the synthetic workload in the scenario shown in Figure 3 (three processes running: P1, P2, and P3). We have concentrated primarily on faults injected while P1 was scheduled and when the processor was executing OS code, for example from a system call called by P1 (for simplicity, we call these faults as **OS faults** or faults injected in **kernel mode**). However, in order to get a more complete picture and observe the differences between OS faults

and application faults, we also injected faults while the processor was executing P1 code (again, for simplicity, we call these faults as **application faults** or faults injected in **user mode**). To achieve this we defined fault triggers in the P1 code space and in the OS code, particularly in the code of system calls related to core functions (e.g., system calls such as fork, kill, wait, open, read, write, etc). We have also injected faults using uniform distribution over time to get results from faults injected in other OS code. A significant number of faults (233 faults) injected with uniform distribution over time were injected while the processor was executing OS code not directly related to user processes, namely while the OS was in its idle loop (we will analyze these faults further on). A small number of faults were injected in other processes and are not used in the analysis.

Table 2 shows the fault distribution by process and Table 3 shows the fault distribution by target unit. In the latter case, the number of faults injected in each unit was weighed with the relative sizes of the silicon areas of the corresponding processor structure, measure (in an approximate way) in the processor die image. The faults injected in the memory were meant to emulate faults in the processor cache, as mentioned before.

Distribution by processes	All faults		User Mode		Kernel Mode	
	# faults	%	# faults	%	# faults	%
Injected while P1 is scheduled	2013	88%	975	48%	1038	52%
Injected while executing OS code PID=0	233	10%	0	0%	233	100%
Injected while executing other processes	30	1%	20	67%	10	33%
Totals	2276	100%	995	44%	1281	56%

Table 2 – Fault injected with the synthetic workload: distribution by processes

Distribution by unit	All faults		User Mode		Kernel Mode	
	# faults	%	# faults	%	# faults	%
GPR	569	25%	249	25%	320	25%
IU	225	10%	109	11%	115	9%
FPU	0	0%	0	0%	0	0%
Memory	529	23%	209	21%	320	25%
Data Bus	465	20%	209	21%	256	20%
Address Bus	488	21%	219	22%	269	21%
Totals	2276	100%	995	100%	1281	100%

Table 3 – Fault injected with the synthetic workload: distribution by unit

The analysis of results in this section is mainly focused in the 2013 faults injected while P1 was scheduled (first line of Table 2), which correspond to 1038 faults injected in kernel mode (i.e., OS faults) and the 975 fault injected in user mode (i.e., application faults).

Figure 4 shows the impact of the faults injected while P1 was scheduled (2013 faults) from the point of view of the process P1 only (i.e., disregarding the impact on the other processes). Both the impact of OS faults and P1 faults are represented, in order to facilitate the comparison and correlation of results observed for each case.

The classification of failure modes is the following:

- **OS crash** – The fault crashed the system and it has to be restarted by a hard-reset.
- **Application hang** – The fault caused the application (P1 in this case) to hang, possibly due to an erroneous infinite loop.

- **Abnormal application termination** – In this case the application process terminated in an abnormal way, either because the return code is abnormal or because the LynxOS terminated the application. Xception records the error codes in order to know what kind of mechanism has detected the error (note that in addition to the inherent error detection mechanisms in the system, P1 includes a routine to detect corruption in the memory structures and P1code).
- **No impact** – The fault has no visible impact on the system: the application P1 terminated normally and the results produced were correct.
- **Wrong results** – The fault caused the application to produce wrong results and no errors have been detected.

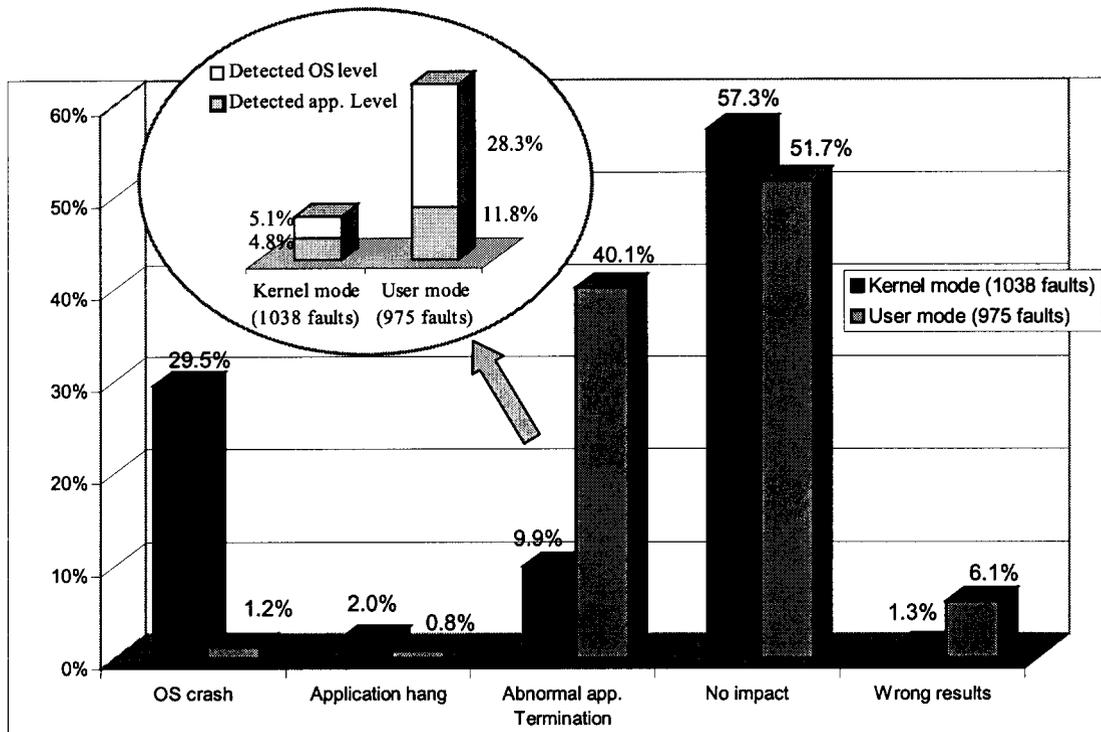


Figure 4 – Impact of faults injected while P1 was scheduled (disregarding the impact on the other processes)

Figure 4 also shows whether the cause of abnormal termination of P1 has been detected at the OS level or at the application level (small chart inside the oval). Even more details about the actual error detection mechanism that caused the abnormal termination of the application are presented in Table 4. All these results will be discussed for OS faults and application faults in the subsequent sections.

A general observation on the results presented in Figure 4 is that a large percentage of faults (more than half for both OS and application faults) had no effect, because the state that was modified was unused or was soon-to-be overwritten. This is consistent with what has been found by others [Arlat 93, Karlsson 98, Madeira 94].

The following sections present and discuss the results obtained for the faults that caused some impact in program execution.

Error detection		Kernel mode		User mode	
		# faults	%	# faults	%
Application level	Memory corruption	0	0.0%	91	9.3%
	Error code returned by OS call	38	3.7%	18	1.8%
	Other error codes	12	1.2%	3	0.3%
	Error codes not defined	0	0.0%	3	0.3%
OS level	SIGTRAP (trace mode)	1	0.1%	32	3.3%
	SIGBUS (bus error)	40	3.9%	4	0.4%
	SIGSEGV (segment violation)	12	1.2%	1	0.1%
	SIGSYS (bad arg. to system call)	0	0.0%	236	24.2%
	SIGPIPE (write on a pipe with no one to read)	0	0.0%	1	0.1%
	Unknown error code	0	0.0%	2	0.2%
Total coverage		103	9.9%	391	40.1%

Table 4 – Error detection details for faults injected while P1 was scheduled and caused abnormal application termination (and disregarding the impact on the other processes).

3.1.1 Faults injected while executing OS code

We observe that OS faults tend to either crash the system (29.5%) or cause no impact (57.3%). A fair percentage of OS faults caused errors that can be detected by the OS or by the application (9.9% total: details about the actual mechanism that detected the error can be seen in Table 4) and only a very small percentage of faults caused the application P1 to hang (2.0%) or to produce wrong results (1.3%). Since the Software Implemented Fault Tolerance (SIFT) techniques of REE were designed to handle crashes and detected errors in applications, we view these as relatively benign outcomes. The last case is the only that causes concern, as the application produces wrong results but there is no way to warn the end-user on that fact, as nothing wrong has been detected in the system. Effective applications-based acceptance checks are needed to improve coverage of application errors.

Another interesting aspect is related to the analysis of the faults injected when the OS was in its idle loop. There are 233 of these faults (see Table 2) from the set of faults injected with uniform distribution over time. The results are represented in Figure 5. As we can see, the impact of these faults is quite clear: they either crash the system or have no impact at all. As mentioned before, this is relatively easy to solve with the SIFT approach of REE that has been particularly designed to handle crashes.

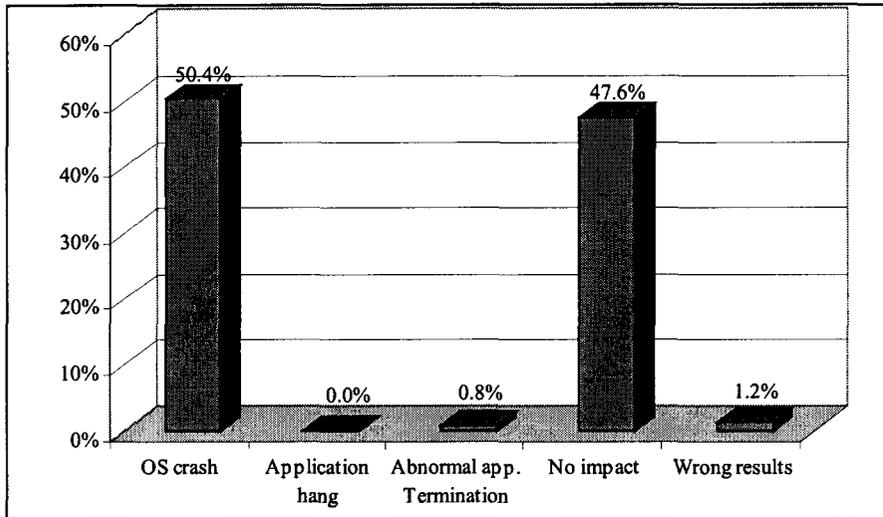


Figure 5 – Impact of faults injected while the processor was executing the OS idle loop (233 faults)

Table 5 shows a breakdown of the results for OS faults that have been injected while the processor was executing specific LynxOS system calls or internal functions. In general, LynxOS calls are compatible with Posix system calls, which makes Table 5 easy to understand.

Concerning the faults that caused wrong results, we can see that most of them have been injected when the processor was executing system calls related to file access, especially *write*, *close_fd*, and *stat*. Another interesting observation is that faults injected during the execution of the *fork* system call are particularly prone to crash the system (more than half).

OS function	Total injected		OS Crash	Application Hang	Abnormal application termination			No impact	Wrong results	
	# faults	%			Mem	OS call	OS			
=.fcopy	2	0.2%	0.0%	0.0%	0.0%	0.0%	0.0%	100.0%	0.0%	
=.resched	32	3.1%	23.1%	0.0%	3.8%	3.8%	0.0%	69.2%	3.8%	
=.fork	82	7.9%	24.2%	3.0%	7.6%	0.0%	1.5%	63.6%	1.5%	
=.kill	115	11.1%	50.5%	0.0%	6.5%	0.0%	0.0%	43.0%	0.0%	
=.read	89	8.6%	29.2%	0.0%	19.4%	1.4%	13.9%	4.2%	50.0%	1.4%
=.write	91	8.7%	34.2%	0.0%	9.6%	0.0%	5.5%	4.1%	47.9%	8.2%
=.close_fd	66	6.3%	17.0%	3.8%	13.2%	0.0%	3.8%	9.4%	62.3%	3.8%
=.close	29	2.7%	34.8%	0.0%	8.7%	0.0%	0.0%	8.7%	56.5%	0.0%
=.open	208	20.1%	28.0%	3.0%	17.9%	5.4%	8.9%	3.6%	50.6%	0.6%
=.stat	41	3.9%	12.1%	0.0%	18.2%	0.0%	3.0%	15.2%	66.7%	3.0%
=.fstat	31	3.0%	32.0%	0.0%	4.0%	0.0%	0.0%	4.0%	64.0%	0.0%
=.wait	58	5.6%	14.9%	8.5%	4.3%	0.0%	2.1%	2.1%	72.3%	0.0%
=.select	192	18.5%	20.0%	0.0%	8.4%	0.0%	0.6%	7.7%	71.6%	0.0%
=.loader	1	0.1%	100.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%

Mem – Detection of memory corruption (by the application memory checking routine)

OS call - Error code returned by OS call to the application

OS - Error detected by the OS (and the OS killed the application)

Table 5 – Impact of faults injected while P1 was executing specific kernel functions (1038 faults)

Figure 6 show the impact of OS faults for different units inside the processor. It's worth noting that faults in the cache are emulated in an approximately way trough faults in the memory in specific areas where the processor is most likely to be executing code or accessing to data.

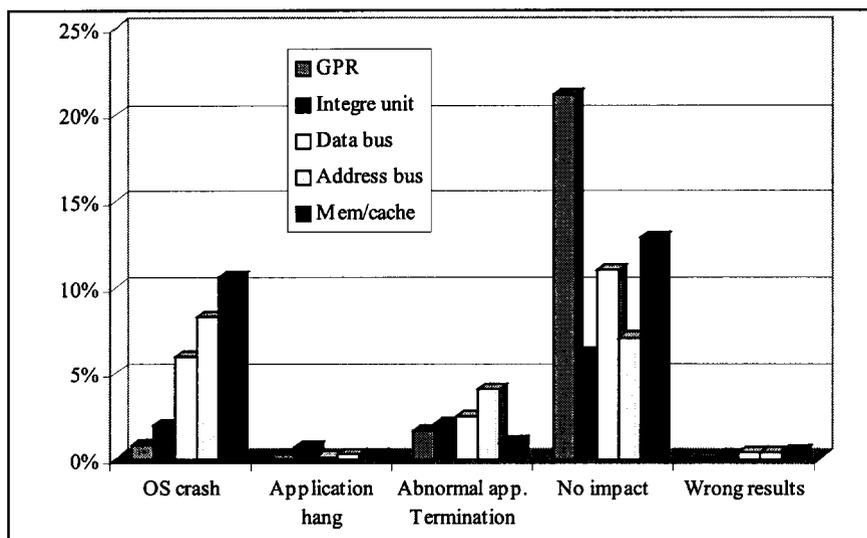


Figure 6 – Impact of OS faults injected in different processor units.

One evident conclusion is that the impact of faults is very dependent on the specific processor area affected by the fault. It is interesting to note that the general purpose registers (GPR) have the highest percentage of faults with no impact, which suggest that uniform distribution of bit-flip errors in the GPR could lead to optimistic results. A more detailed observation of the Xception database has shown that only faults in some registers have caused evident impact, which result from the non-uniform way programs (and compilers) use the GPRs.

Although we don't have space in the paper to show the tables obtained when we cross tables 5 and 6 (i.e., impact for faults injected in specific system calls and in specific processor units), the observation of the Xception database have shown that the most dangerous combinations concerning the production of wrong results consists of faults injected in the processor data and address bus while executing code of the *write* system call and faults injected in the cache while the processor is executing the *close_fd*.

3.1.2 Faults injected while executing application code

Application faults follow a quite different pattern in many aspects, when compared to OS faults. Two evident observations are that application faults tend to produce higher percentages of wrong results (6.1%) and cause a much smaller percentage of system crashes (1.2% in this case). The first one shows that application faults are more dangerous than OS faults (in fact, OS faults tend to crash the system which avoids the production of wrong results). The second observation suggests that LynxOS was not able to protect itself from a mad application, as some faults (although in a small percentage) injected when the processor was executing application code cause the OS to crash.

A close analysis of the Xception log¹ on the application faults that caused the OS to crash has shown that most of these faults correspond to faults that affected registers used to pass parameters of OS calls (interestingly, most of these faults were injected in the data bus and address bus, and not in the registers directly). This is consistent with previous work from CMU and LAAS on OS robustness testing [Siewiorek 93, Koopman 97, Salles 99, Fabre 99] where tools like Ballista and

¹ The Xception log stores the exact location in the code of the instruction under execution when the fault was injected (and a configurable number of instructions) and the processor context in the moment of the injection.

Mafalda have shown that erroneous OS calls parameters can crash the OS. The use of wrappers to filter these erroneous parameters can potentially solve these weak points and make the OS to behave in an acceptable way in the presence of bad-behaved applications. This would turn the 1.2% of application faults in Figure 4 into errors detected by the OS and returned to the calling application.

It is worth noting that LynxOS is fairly robust, as it has detected 24.2% of the injected faults just because these faults have corrupted the arguments of system calls (see Table 4), and the OS has not handled only 1.2% of the injected faults correctly (note that these percentages are absolute, i.e., relative to all the application faults). These results also show that for the application P1 (and remember that P1 uses OS calls very heavily) the percentage of faults typically addressed by robustness testing correspond to 25.4% (sum of both values) of the application faults. Handling these faults correctly, and enhancing the detection of other kind of error by the OS, is quite positive for a quick recovery of the application (as the OS has not crash).

3.1.3 Study of error propagation

A failure mode classification that specifically addresses the error propagation is shown in Table 6.

Classification	Description
System crash	All the processes (applications) have crashed. The OS have to be rebooted.
Application damage	<p>The process P1 (the target application) has been affected somehow but all the other processes (P2 and P3 in our case) executed completely and produced correct results. That is, the damages were confined to P1. The following damages are considered:</p> <ul style="list-style-type: none"> • Application crash - The process P1 (the target application) crashed. No results have been produced by P1. • Errors detected - Errors have been detected at the application level. Following types are considered: <ul style="list-style-type: none"> - Memory consistency checks: the application memory space has been corrupted; - OS call error: system calls executed by the application have returned an error. - Application terminated by OS • Wrong results: The application terminated normally (no errors detected) but produced incorrect results.
Error propagation	<p>Faults injected when P1 is scheduled affected at least one of the other applications (P2 and P3 in our case) but the system did not crash. The following types are considered:</p> <ul style="list-style-type: none"> • Other application crash: The application crashed. No results have been produced by the application. • Errors detected in other application: Errors have been detected at the application level. The same types above. • Wrong results in other application: Other application terminated normally (no errors detected) but produced incorrect results.
No impact	All the applications terminated normally and produced correct results.

Table 6 – Failure mode classification for study of error propagation.

Figure 7 shows a breakdown of the results of OS and application faults, breaking out the effects of error propagation between processes and the effects of application damage.

The error propagation for OS and application faults is quite different. While only 1.0% of the OS faults propagate to the other processes P1 and P2, the percentage observed for application faults is 4.4%. The manual inspection of these faults (from the Xception log) has shown different error

propagation scenarios, but most of them consist of faults injected in the memory (but trying to simulate faults in the cache) and faults that have corrupted parameter of OS calls.

A very important aspect concerning the error propagation caused by application faults is that most of the propagated errors have been detected by these applications or by the OS. Only 3 out of 975 application faults and 6 out of 1038 OS faults have escaped the error detection and caused the other applications (P2 and P3) to produce incorrect results. Figure 7 also shows the error mechanisms in P2 and P3 that have detected the errors caused by these 36 faults.

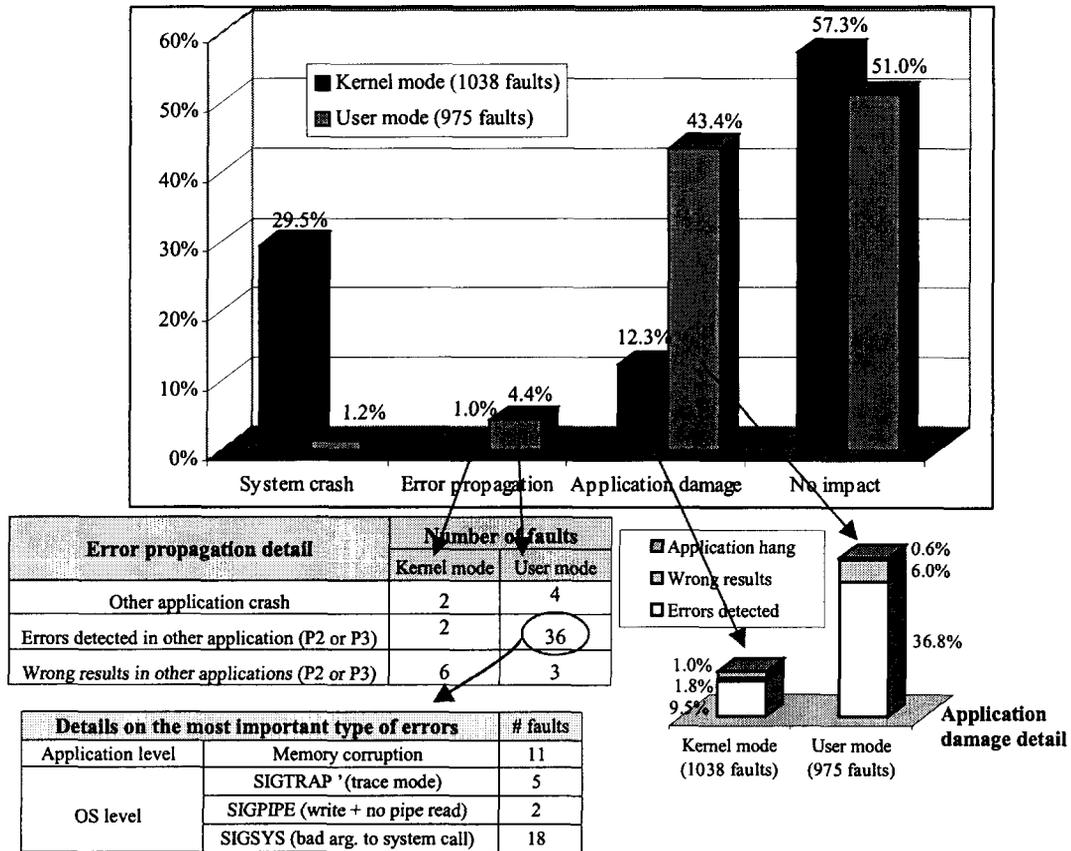


Figure 7 – Impact of faults injected while P1 was scheduled

The relatively small percentage of faults that caused error propagation and wrong results suggests the SIFT approach of REE can in fact handle the vast majority of faults. The error propagation, however, even when the errors are detected, cause increased error latency and recovery time and force the recovery of multiple applications. Improved mechanisms to help confining the errors to the application affected by the faults are necessary.

The percentage of faults whose damage was confined to P1 is very high for the application faults (43.4%). This means that LynxOS does a good job in confining the errors to the application that is directly affected by the fault. Furthermore, as most of these faults have been detected (36.8%), thus suggests that these kind of faults can be recovered effectively using the SIFT approach of REE. Table 7 shows the details on the error detection mechanisms for both OS faults and application faults.

Error detection		Kernel mode		User mode	
		# faults	%	# faults	%
Application level	Memory corruption	0	0.0%	79	8.1%
	Error code returned by OS call	35	3.4%	18	1.8%
	Other error codes	12	1.2%	2	0.2%
	Error codes not defined	0	0.0%	3	0.3%
OS level	SIGTRAP (trace mode)	1	0.1%	27	2.8%
	SIGBUS (bus error)	40	3.9%	4	0.4%
	SIGSEGV (segment violation)	11	1.1%	1	0.1%
	SIGSYS (bad arg. to system call)	0	0.0%	222	22.8%
	SIGPIPE (write on a pipe with no one to read)	0	0.0%	0	0.0%
	Unknown error code	0	0.0%	3	0.3%
Total coverage		99	9.5%	359	36.8%

Table 7 – Error detection details for faults that caused damage confined to P1.

3.2 Workload termination and correction of application results: experiments with the realistic workloads

Analyzing combined OS and application faults for synthetic applications is not particularly useful, because the relative effects of each are very dependent upon the relative fractions of time spent executing applications and OS code. And this may vary widely for different real applications.

This is subtle and requires an explanation. In REE, most state of non-running processes is in second-level cache or main memory. This is protected by error correcting codes and thus relatively immune to transient errors. Most exposure to SEUs therefore comes in the processor or surrounding circuitry, which are devoted to running processes.

Therefore the following experiments use random fault-injection into the processor. The goal is to analyze the impact of faults on application termination and the correctness of the application results. To achieve this we used realistic workloads (although the programs used are simple, they are similar to REE real programs) and injected faults following a uniform distribution over time and processor location, as this is the best way to emulate the effects of real SEU faults.

Workload	Number of faults		
	User mode	Kernel mode	Total
Gravity	36 (5.7%)	589 (94.3%)	625
PI	626 (99.6%)	3 (0.4%)	629
Matmult	1277 (99.3%)	9 (0.7%)	1286

Table 8 – Distribution of the faults injected in the realistic workloads

The experiments with the different applications are independent from each other (i.e., only one application was running each time). Table 8 shows the distribution of the faults. Faults injected in user mode means that the faults have been injected when the processor was executing code from the application and faults injected in kernel mode means that the processor was executing OS code. The distribution of the faults among the different processor units is similar to the one presented in Table 3.

As we can see from Table 8, the profile of the applications plays an important role on the fault distribution. Faults injected when the application is computing results tend to affect application code while fault injected during I/O periods tend to affect operating system code (mainly open, write, and read functions). In order to clarify this point, Table 9 shows the key aspects of the profile of the applications. As we can see, the profile of the Gravity application is very different from the other applications, as this application spends most of the time writing the results into disk (the computation time is less than one second). This is why most of the faults (94%) have been injected in OS code. The patterns for the other applications are very different, especially for the PI application where the I/O activity can be neglected.

Workload	Execution time (sec.)		Size of results
	Calculations	Storing results	
Gravity	~1	24	1.01 Mbytes
PI	~17	Neglected	53 bytes
Matmult	~22	~2	24.04 Kbytes

Table 9 – Key features of the real workloads used in the experiments.

Figure 8 shows the results for the three applications (obtained in three separate experiments, one for each application). The most evident observation is that the results vary a lot with the application. The profile of the application concerning the use of OS calls plays a relevant role, although the impact of faults seems to be dependent on other factors.

The program PI practically does not execute system calls and spend all the time doing calculations. The percentage of wrong results in this case is very high (50.4%), which clearly shows that some applications can be very prone to produce wrong results. The Matmult application also shows a high percentage of wrong results (27.0%). This suggests that computation intensive applications must be protected with application based error detection techniques to avoid the production of wrong results (note that in these cases no errors have been detected). Given the nature of applications required in many space scientific experiments, it suggests that techniques such as Algorithm Based Fault Tolerance (ABFT) could be a good alternative. Previous work showed that these techniques are quite effective for faults similar to the ones used in this work [Silva 98].

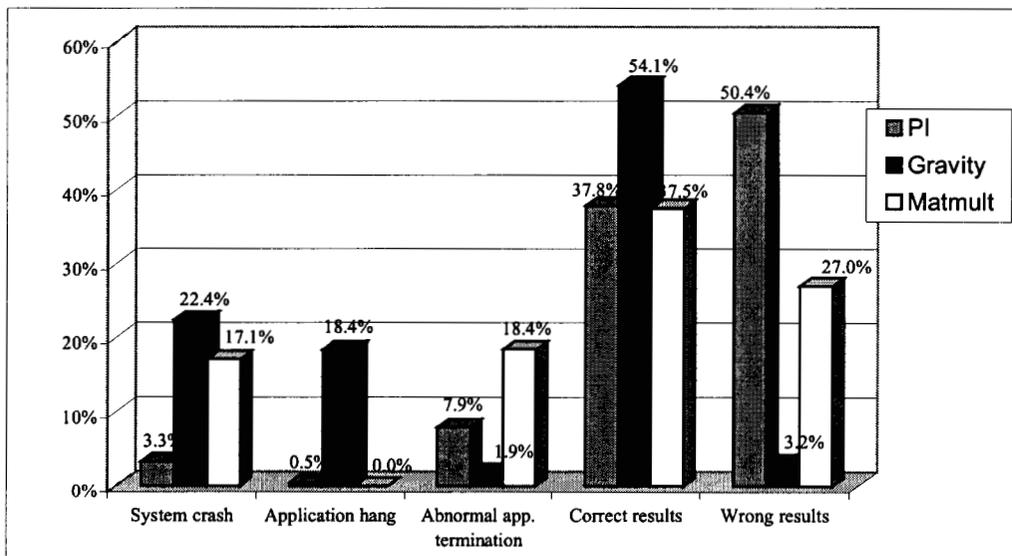


Figure 8 – Impact of faults on application termination and correctness of results.

Gravity application follows a pattern similar to the synthetic application (the only exception is for the “application hang” results). As we saw in Table 9, this application spends most of the time writing the results to disk and this is why most of the faults have been injected in kernel mode.

The percentages of faults that caused abnormal application termination observed in the realistic applications are clearly lower than the ones in the synthetic application. Note that abnormal termination means that some error has been detected, either at OS or application level. The application profile, again, seems to play an important role, as the percentages of errors detected are quite different for the three applications. When comparing these results to the ones from the synthetic application we should take into account that the synthetic application also includes the test of the memory structures and code of the application, which does not exist in the realistic applications.

Table 10 shows details on the different error detection mechanisms that originated the abnormal termination. The detection of erroneous arguments in OS calls (by the OS) is the mechanism with higher coverage, which confirms the results observed in the experiments with the synthetic workload that suggests that LynxOS is fairly robust.

Type of abnormal application termination	Specific error	PI		Gravity		Matmult	
		# faults	%	# faults	%	# faults	%
Error code returned by OS call (Detected at application level)	Ordinary error code	0	0.0%	0	0.0%	2	0.2%
Error detected by OS (OS level detection only: the OS sends a signal to kill the process)	SIGTRAP (trace mode)	3	0.5%	1	0.2%	37	2.9%
	SIGBUS (bus error)	1	0.2%	0	0.0%	7	0.5%
	SIGSYS (bad arg. to system call)	46	7.3%	11	1.8%	191	14.9%
Total		50	7.9%	12	1.9%	237	18.4%

Table 10 – Abnormal application termination details.

4. Conclusions

In this paper we evaluated the impact of faults in a COTS system for scientific data processing in space applications. The target system was a COTS CETIA board with two PowerPC 750 running LynxOS, and the study has been conducted using a Software-Implemented Fault Injection tool (Xception). The faults injected have been defined in order to mimic the effects of SEU in the processor internal units as much as possible. The faults were distributed in a uniform way concerning processor location and both uniform distribution over time and faults in specific locations of the OS code have been used. Two types of workloads have been used: a synthetic workload to exercise core functions of the operating system such as the ones related to processes, memory, and input/output, and three realistic programs similar to the REE applications.

The results provide a picture of the impact of faults on LynxOS key features such as the ability to continue execution of the processes after a fault, data integrity (tested through a specific memory test routine), error propagation (among application processes, from applications to the OS, and from OS to the application processes), application termination, and correctness of application results. The following observations are particularly relevant:

- OS faults are the best to deal with, since tend to crash the system or cause no visible impact at all. As SIFT systems are designed for crash recovery, this conclusion supports the idea that most of the faults can be tolerated by use of SIFT techniques.

- Applications fault results imply that assumptions of fail silent used by many researchers are inadequate. In fact a variable but quite significant percentage of application faults caused wrong results and no error have been detected to warn the user that the results were not correct. More research is needed into application-based acceptance checking to achieve coverage consistent with highly dependable systems
- Applications with intensive calculations and few OS calls are more likely to produce wrong results in the presence of faults than applications that use OS calls in an intensive way (these ones tend to crash the system or cause no impact). Results obtained with realistic workloads show quite clearly the effect of the application profile on the failure modes.
- The LynxOS is quite effective in confining the errors to the process directly affected by the fault. However, small percentages of propagated errors have been observed (from 1% to 4.4%). The higher percentages of error propagation happened for application faults, which suggests that improved application based error detection is necessary.
- For most of the faults that caused error propagation, the propagated errors have been detected in the other applications, which suggests that these kind of faults could be recovered effectively using the SIFT approach of REE.
- The LynxOS seems fairly robust, as most of the faults that caused erroneous parameters in OS calls have been detected by the OS. However, a small percentage of these faults were not detected, which shows that additional wrappers could improve even further the robustness of LynxOS.

Future work will include the analysis of the results obtained for faults already injected but that could not be included in this paper for space reasons. Other types of faults such as burst faults that affect many bits at the same time are also relevant for space missions. Other applications with more processes and tight communication and synchronization will also be analyzed in order to understand better the most relevant error propagation factors.

References

- [Arlat 93] J. Arlat, A. Costes, Y. Crouzet, J.-C. Laprie and D. Powell, "Fault Injection and Dependability Evaluation of Fault-Tolerant Systems", *IEEE Trans. on Computers*, 42 (8), pp.913-23, August 1993.
- [Beahan 00] J. J. Beahan, L. Edmonds, R. D. Ferraro, A. Johnston, D. Katz, R. R. Some, "Detailed Radiation Fault Modeling of the Remote Exploration and Experimentation (REE) First Generation testbed Architecture," *Aerospace Conference Proceedings*, vol. 5, pp. 279-291, 2000.
- [Carreira 98] J. Carreira, H. Madeira, and J. G. Silva, "Xception: Software Fault Injection and Monitoring in Processor Functional Units", *IEEE Transactions on Software Engineering*, vol. 24, no. 2, February 1998.
- [CETIA] VMPC5x-Dual, CETIA Dual Processor VME Board. Part# VMPC5A-DUAL-366-128.
- [Fabre 99] J.-C. Fabre, F. Salles, M. Rodríguez Moreno and J. Arlat, "Assessment of COTS Microkernels by Fault Injection", in *Dependable Computing for Critical Applications* (Proc. 7th IFIP Working Conf. on Dependable Computing for Critical Applications: DCCA-7, San Jose, CA, USA, January 1999), C. B. Weinstock and J. Rushby, Eds., *Dependable Computing and Fault-Tolerant Systems*, 12, (A. Avizienis, H. Kopetz and J.-C. Laprie, Eds.), 1999, pp.25-44.
- [Karlsson 98] J. Karlsson, P. Folkesson, J. Arlat, Y. Crouzet, G. Leber and J. Reisinger, "Application of Three Physical Fault Injection Techniques to the Experimental Assessment of the MARS Architecture", in *Dependable Computing for Critical Applications* (Proc. 5th IFIP Working Conf. on Dependable Computing for Critical Applications: DCCA-5, Urbana-Champaign, IL, USA, September 1995) (R. K. Iyer, M. Morganti, W. K. Fuchs and V. Gligor, Eds.), *Dependable Computing and Fault-Tolerant Systems*, 10, (A. Avizienis, H. Kopetz and J.-C. Laprie, Eds.), pp.267-87, IEEE CS Press, 1998.

- [Koopman 97] P. J. Koopman, J. Sung, C. Dingman, D. P. Siewiorek and T. Marz, "Comparing Operating Systems using Robustness Benchmarks", in Proc. 16th Int. Symp. on Reliable Distributed Systems, SRDS-16, Durham, NC, USA, 1997, pp.72-79.
- [Madeira 94] H. Madeira and J.G. Silva, "Experimental Evaluation of the Fail-Silent Behavior in Computers Without Error Masking," Proc. 24th Int'l Symp. Fault Tolerant Computing Systems, Austin-Texas, 1994, pp. 350-359.
- [Salles 99] F. Salles, M. Rodriguez, J.-C. Fabre and J. Arlat, "Metakernels and Fault Containment Wrappers", in Proc. 29th IEEE Int. Symp. on Fault-Tolerant Computing (FTCS-29), (Madison, WI, USA), pp.22-9, IEEE CS Press, 1999
- [Siewiorek 93] D. P. Siewiorek, J. J. Hudak, B.-H. Suh and Z. Segall, "Development of a Benchmark to Measure System Robustness", in Proc. 23rd Int. Symp. on Fault-Tolerant Computing, FTCS-2, Toulouse, France, 1993, pp.88-97.
- [Silva 98] João Gabriel Silva, Paula Prata, Mário Rela, and H. Madeira, "Practical Issues in the Use of ABFT and a new Failure Model", 28th IEEE Annual International Symposium on Fault-Tolerant Computing Symposium, FTCS-28, Munich, Germany, June 1998, pp 26-35.
- [Some 99] R. R. Some and D. C. Ngo, "REE: A COTS-Based Fault Tolerant Parallel Processing Supercomputer for Spacecraft Onboard Scientific Data Analysis," Proc. of the Digital Avionics System Conference, vol.2, pp.B3-1-7 -B3-1-12,1999.
- [Xception 00] Critical Software, S.A., "Xception: Professional Fault Injection", White Paper, <http://www.criticalsoftware.com>, 2000.