Experimental evaluation of a COTS system for space applications

Henrique Madeira
Univ. of Coimbra
Coimbra, Portugal
henrique@dei.uc.pt

Raphael R.Some
Jet Propulsion Laboratory
Pasadena, CA, USA
Raphael.R.Some@jpl.nasa.gov

F. Moreira, D. Costa
Critical Software
Coimbra, Portugal
dino@criticalsoftware.com

David Rennels
Univ. Calif. Los Angeles
Los Angeles, CA, USA
rennels@cs.ucla.edu

Abstract

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 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. The results provide a comprehensive picture of the impact of faults on LynxOS key features (process scheduling and the most frequent system calls), data integrity, error propagation, application termination, and correctness of application results.

1. Introduction

The use of Commercial Off-The-Shelf (COTS) components and 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, COTS are not usually designed for the stringent requirements of space applications. This means that the actual use of COTS components in space missions must be preceded by careful study of the impact of faults such as the ones caused by space radiation (e.g., Single Event Upsets (SEU)) in order to identify weak points that should be strengthened with specific fault tolerance techniques.

One proposal of using COTS-based systems in space applications is to use these systems for scientific data processing and not for spacecraft control. A good example of this approach is the Remote Exploration and Experimentation (REE) Project from NASA's Jet Propulsion Laboratory [1]. In this project, an external, radiation-hardened and independently protected Spacecraft Control Computer (SCC) is used for the overall spacecraft control, while scientific computations are performed by COTS-based systems. In this scenario, the use of Software Implemented Fault Tolerance (SIFT) techniques in the COTS-based systems to tolerate transient faults seems an interesting alternative [2].

This paper evaluates the impact of transient errors in a CETIA board with two PowerPC 750 processors running the LynxOS and quantifies their effects at both the OS and at the application level. The emphasis of the work is on the system behavior when errors are induced into the operating system – looking at application termination, data integrity, error propagation, and correctness of application results. A fault injection tool [3] has been used to emulate the effects of SEU through the insertion of bit-flip errors in processor structures and memory whilst the processor is executing OS or application code.

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 is a Sun UltraSparc-II with SunOS 5.5.1.

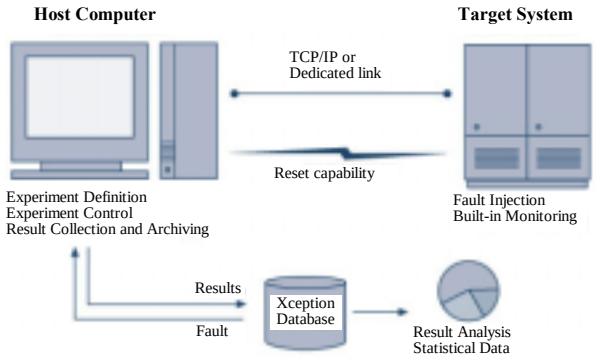

Figure 1 – Test bed layout

The injection tool is the Xception PowerPC705/ LynxOS, which is a port of the Xception tool [4] 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 (including the kernel code) and can be injected in target locations such as processor registers, integer unit, internal processor buses, floating point unit, cache, and memory. In the present experiments, faults consist of single bit-flips injected in all possible units that can be reach by Xception.

The definition of the faults parameters, injection process, and collection of results is controlled by the host system. Figure 2 shows the key steps of the fault injection process. 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.

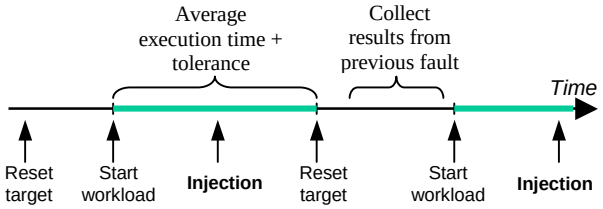

Figure 2 - Typical injection run profile

2.2. Workload

2.2.1. Synthetic workload

As one major goal of this study is the evaluation of the impact of transient faults on key features of 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, 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. After each step, the program stores a footprint in a file. For example, after each system call the program stores the return code; after each checksum calculation (performed over all data structures and application code), it 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 is compared with a gold version to check if any of the individual step outputs were wrong. The amount of memory allocated for buffer manipulations is 1 Mbyte and the matrices manipulations are performed on three matrix of 250 x 250 integers define as static variables.

Three instances 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, etc) during the execution of P1 code or during the execution of kernel code. The processes P2 and P3 are used to evaluate error propagation from P1 to P2 or P3 and the operating system response after a fault. Figure 3 illustrates this scheme.

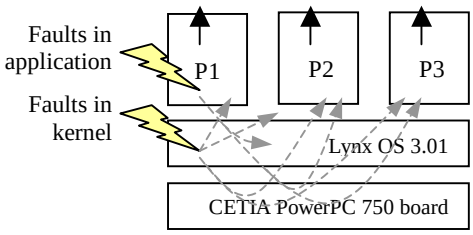

Figure 3 – Process configuration: synthetic workload

2.2.2. Realistic workloads

The use of realistic workloads is particularly relevant to the evaluation of the effects of OS errors on application termination and result correctness. We chose three applications with different profiles (processor and memory needs, use of I/O system calls), and size of the results, in order to evaluate the influence of these parameters on the impact of the faults. The workloads are:

- **Gravity**: Calculates the trajectory of a small mass attracted by a bigger one, as 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

In short, the different sets of experiments performed have the following goals:

- Experiments using the Synthetic Applications:
  - Evaluate the impact of faults injected while the processor was executing OS code (section 3.1.1). We have used a synthetic workload to be sure that core OS functions are heavily used.
  - Evaluate the impact of faults injected 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 error propagation from OS to application processes and from one process to the others (section 3.1.3).
- Experiments using Realistic Applications:
  - Evaluate the impact of faults on application termination and the correctness of the application results (section 3.2). The use of realistic programs and uniform faults distributions try to emulate as close as possible the effects of SEU errors in real applications.

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. For simplicity, we call the faults injected while P1 was scheduled and when the processor was executing OS code as OS faults (or **kernel mode** faults**)**. Similarly, we call the faults injected while the processor was executing P1 code as application faults (or **user mode** faults**)**.

Table 1 shows the fault distribution by process. The fault distribution by target unit was weighted by the relative sizes of the silicon areas of the corresponding processor structure. Some faults (23%) were injected in key location of the memory to emulate faults in the processor cache.

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

Table 1 – Fault injected with the synthetic workload.

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).

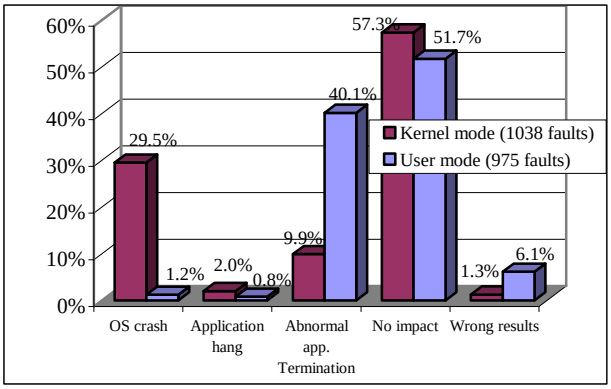

Figure 4 – Impact of faults while P1 was scheduled

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 to hang, possibly due to an erroneous infinite loop.
- **Abnormal application termination** – The process terminated abnormally, either because the return code is abnormal or the LynxOS terminated the application.
- **No impact** – The fault had no visible impact on the system.
- **Wrong results** – The fault caused the application to produce wrong results; no errors have been detected.

A general observation of the results on Figure 4 is that a large percentage of 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 [5, 6, 7].

A considerable number of abnormal terminations were observed, especially when faults are injected in the application code (user mode). Table 2 shows details about the actual error detection mechanism that caused the abnormal termination of P1. All these results will be detailed for OS faults and application faults in the subsequent sections.

Error detection		Kernel mode (1038 faults)		User mode (975 faults)	
		#faults	%	#faults	%
App. level	Memory corruption	0	0.0%	91	9.3%
	Error code ret. 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%
	Total Application level	***51***	***4.8%***	***115***	***11.8%***
OS level	SIGTRAP (trace mode)	1	0.1%	32	3.3%
	SIGBUS (bus error)	40	3.9%	4	0.4%
	SIGSEGV (seg. violation)	12	1.2%	1	0.1%
	SIGSYS (bad arg. sys. call)	0	0.0%	236	24.2%
	SIGPIPE (error on pipe)	0	0.0%	1	0.1%
	Unknown error code	0	0.0%	2	0.2%
	Total OS level	***52***	***5.1%***	***276***	***28.3%***
	Total coverage	**103**	**9.9%**	**391**	**40.1%**

Table 2 – Error detection details for faults injected while P1 was scheduled which caused abnormal application termination.

3.1.1. Faults injected while executing OS code

We observed 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: see details in Table 2) 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 SIFT techniques that could be added to COTS-based systems for space applications are designed to handle crashes and detected errors in applications, we view these as relatively benign outcomes. The last case is the only one that causes concern, as the application produces wrong results and there is no way to warn the end-user of that fact, as nothing wrong has been detected in the system. Effective applications-based acceptance checks are clearly needed.

Table 3 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 3 easy to understand.

Concerning the faults that caused wrong results, most of them were 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.

OS function	Total injected		OS Crash	App. Hang	Abnormal app. termination				No impact	Wrong results
	# faults	%				*Mem*	*OS call*	*OS*		
=.fcopy	2	0.2%	0.0%	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%*	*0.0%*	69.2%	3.8%
=.fork	82	7.9%	24.2%	3.0%	7.6%	*0.0%*	*1.5%*	*6.1%*	63.6%	1.5%
=.kill	115	11.1%	50.5%	0.0%	6.5%	*0.0%*	*0.0%*	*6.5%*	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 *– Det. of memory corruption (by the app. mem. 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 3 – Impact of faults injected while P1 was executing specific kernel functions (1038 faults).

Figure 5 shows the impact of OS faults for different target 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 log has shown that only faults in some registers have caused evident impact, which result from the non-uniform way programs/compilers use the GPRs.

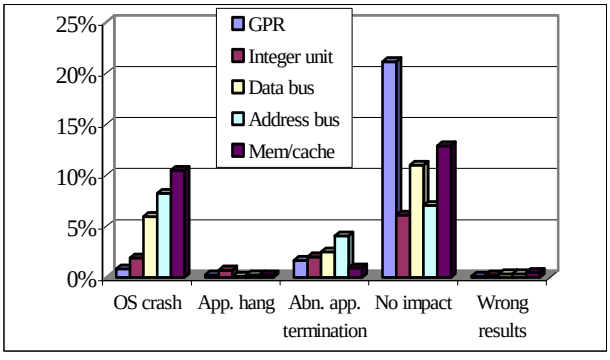

Figure 5 – Impact of OS faults in different processor units.

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). A close analysis of the Xception log has shown that most of the application faults that caused the OS to crash correspond to faults that affected registers used to pass parameters of OS calls. This is consistent with previous works from CMU and LAAS on OS robustness testing [8, 9, 10, 11] that have shown erroneous OS calls parameters can crash the OS. The use of wrappers can potentially solve these weak points and make the OS to behave in an acceptable way in the presence of bad-behaved applications.

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 2), and the OS has not handled correctly only 1.2% of the injected faults. These results also show that for the application P1 the percentage of faults addressed by robustness testing correspond to 25.4% of the application faults (remember that P1 uses OS calls very heavily). Handling these faults correctly is quite positive for a quick application recovery (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 4.

- **System crash** – All the processes crashed: OS has to be rebooted.
- **Application damage -** Fault damages were confined to P1 (other processes executed normally). The following damages are considered:
  - **Application crash** - The process P1 (the target application) crashed. No results have been produced by P1.
  - **Errors detected** - Errors have been detected at the app. level (e.g., mem. consist. checks, error codes ret. by OS calls, app. terminated by OS,…).
  - **Wrong results:** The application terminated normally (no errors detected) but produced incorrect results.
- **Error propagation -** Faults injected when P1 is scheduled affected at least one of the other processes but the system did not crash. The following propagation types are considered:
  - **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.
  - **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 4 – Failure mode classification for study of error propagation.

Figure 6 shows a breakdown of the results of OS and application faults and Tables 5 and 6 show detailed results on 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%. Manual inspection of these faults has shown different error propagation scenarios, but most of them consist of faults injected in the memory (to simulate cache faults) and faults that have corrupted parameter of OS calls. Most of the propagated errors were detected by the other applications or by the OS. Only 3 out of 975 application faults and 6 out of 1038 OS faults escaped detection and caused the other applications (P2 and P3) to produce incorrect results.

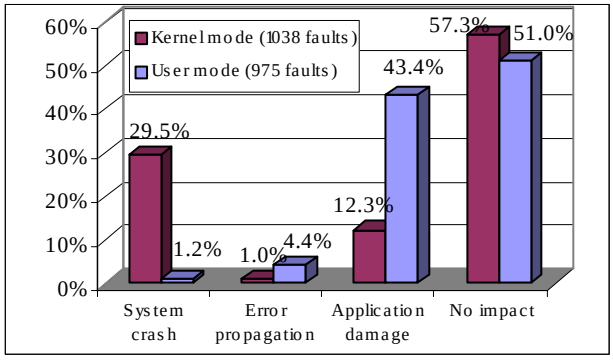

Figure 6 – Impact of faults injected while P1 was scheduled considering error propagation.

Error propagation detail	Number of faults	
	Kernel mode	User mode
Other application crash	2	4
Errors detected in other application (P2 or P3)	2	36
Wrong results in other applications (P2 or P3)	6	3

Details on the most important type of errors		# faults
Application level	Memory corruption	11
OS level	SIGTRAP ' (trace mode)	5
	SIGPIPE (write + no pipe read)	2
	SIGSYS (bad arg. to system call)	18

Table 5 – Error propagation details.

Application damage details	Kernel mode (1038 faults)	User mode (975 faults)
Application hang	1.0%	0.6%
Wrong results	1.8%	6.0%
Errors detected	9.5%	36.8%

Table 6 – Application damage details.

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%), it suggests that these kind of faults can be recovered effectively using a SIFT approach.

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

The goal of the following experiments is to analyze the impact of faults on application termination and the correctness of the application results. To achieve this we used realistic workloads and injected faults following a uniform distribution over time and processor location, as this is the best way to emulate the effects of 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 7 – Percentage of faults injected in the workloads.

The experiments with the different applications are independent one from each other. Table 7 shows the percentages of the faults injected in each workload. 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).

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

Table 8 – Key features of the workloads profile.

Table 8 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. 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 is negligible.

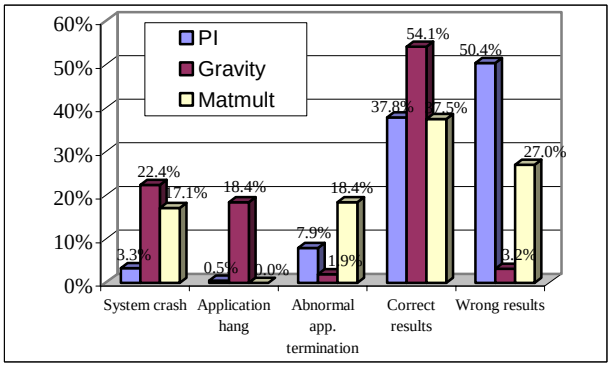

Figure 7 – Failure modes with realistic applications.

The most evident observation from Figure 7 is that the results vary a lot with the application, possibly due to the different application profiles. The program PI does not execute many system calls and spend nearly all of its 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 producing 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.

The percentages of faults that caused abnormal application termination observed in the realistic applications are lower than the ones in the synthetic application. This is partially due to the fact 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 (error detections force an abnormal termination in the synthetic application).

4. Conclusions

In this paper we evaluated the impact of faults in a COTS system for scientific data processing in space applications. The faults were distributed in a uniform way with respect to processor location. Both uniform distribution over time and faults in specific locations of the OS code are 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 simple scientific space applications.

The following observations are particularly relevant:

- OS faults are the easies to deal with, since they tend to crash the system or cause no visible impact at all. Since SIFT systems are designed for crash recovery, this result supports the idea that most of these faults can be handled by SIFT techniques.
- Applications fault results imply that assumptions of fail silent used by many researchers are inadequate. 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.
- 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 were detected in the other applications, which suggests that these kind of faults could be recovered using SIFT techniques.
- The LynxOS seems fairly robust, as most of the faults that cased 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..

5. References

[1] 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.

[2] K. Whisnant, R. Iyer, D. Rennels, and R. Some, "An Experimental Evaluation of the REE SIFT Environment for Spaceborne Applications", paper accepted at the *International Performance and Dependability Symposium*, Washington, DC, June 23rd - 26th, 2002.

[3] Critical Software, S.A., "Xception: Professional Fault Injection", *White Paper*, http://www.criticalsoftware.com, 2000.

[4] 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.

[5] 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.

[6] J. Karlsson, P. Folkesson, J. Arlat, Y. Crouzet, G. Leber, J. Reisinger, "Application of Three Physical Fault Injection Techniques to the Experimental Assessment of the MARS Architecture", *Proc. 5th IFIP Working Conf. on Dependable Computing for Critical App.: DCCA-5*, Urbana-Champaign, IL, USA, Sept. 1995.

[7] 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,

[8] 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.

[9] 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.

[10] F. Salles, M. Rodríguez, 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, 1999.

[11] J.-C. Fabre, F. Salles, M. Rodríguez Moreno and J. Arlat, "Assessment of COTS Microkernels by Fault Injection", in *Proc. 7th IFIP Working Conf. on Dependable Computing for Critical Applications: DCCA-7*, San Jose, CA, USA, Jan. 1999.

Acknowledgements

Funding for this paper was provided, in part, by Portuguese Government/European Union through R&D Unit 326/94 (CISUC) and by DBench project, IST 2000 - 25425 DBENCH, funded by the European Union.