

# Xception: A Technique for the Experimental Evaluation of Dependability in Modern Computers<sup>1</sup>

João Carreira, Henrique Madeira, and João Gabriel Silva  
Dep. Engenharia Informática  
Universidade de Coimbra, Portugal

## Abstract

*An important step in the development of dependable systems is the validation of their fault tolerance properties. Fault injection has been widely used for this purpose, however with the rapid increase in processor complexity, traditional techniques are also increasingly more difficult to apply. This paper presents a new software implemented fault injection and monitoring environment, called Xception, which is targeted for the modern and complex processors. Xception uses the advanced debugging and performance monitoring features existing in most of the modern processors to inject quite realistic faults by software, and to monitor the activation of the faults and their impact on the target system behavior in detail. Faults are injected with minimum interference with the target application. The target application is not modified, no software traps are inserted, and it is not necessary to execute the target application in special trace mode (the application is executed at full speed). Xception provides a comprehensive set of fault triggers, including spatial and temporal fault triggers, and triggers related to the manipulation of data in memory. Faults injected by Xception can affect any process running on the target system (including the kernel), and it is possible to inject faults in applications for which the source code is not available. Experimental results are presented to demonstrate the accuracy and potential of Xception in the evaluation of the dependability properties of the complex computer systems available nowadays.*

**Index Terms:** Fault Injection, RISC Processors, Dependability Evaluation, Real Time

## 1. Introduction

Computer systems are used nowadays in an increasing number of applications that require high levels of dependability. In some cases our lives depend on them, such as in traffic control, medical life support, or nuclear power station management applications. In other cases, such as banking, telecommunications and aerospace, failures can cause tremendous economic losses. Another novel area where dependability is increasingly important is high performance parallel computing. Parallel computers are used to run computation intensive applications such as fundamental physics/chemistry, and airplane/vehicle modeling, during large periods of time. Dependability is important to enable those long runs in spite of the increased probability of fault occurrence caused

---

<sup>1</sup> This work was partially supported by the European Union through Esprit project 6731 - FTMPs "Fault Tolerant Massively Parallel Systems" and by the Portuguese Research Agency - JNICT - through Program PRAXIS XXI.

by the larger number of electronic components in parallel computers.

Generally, a dependable computer should be able to detect software or hardware errors, locate their origin and recover from those errors by using some kind of fault tolerance mechanisms. One important problem is how to evaluate and validate the effectiveness of the fault tolerance mechanisms embedded in these systems before production in order to correct defects or/and provide feedback for improvements.

The validation of the dependability properties of a computer system is an intrinsically complex task and the growing complexity of both the hardware and software tend to make it even more difficult. The use of analytical modeling in actual systems is very difficult as the mechanisms involved in the fault activation and in the error propagation process are highly complex and are not completely understood in most of the cases. Furthermore, the simplifying assumptions usually made to make the analysis tractable reduce the usability of the results achieved by this method. Experimental verification by error logging implies monitoring the systems' behavior until real faults occur and is not appropriate or feasible in most of the cases.

Experimental evaluation by fault injection has become an attractive way of validating specific fault handling mechanisms and allowing the estimation of fault-tolerant system measures such as fault coverage and error latency [1]. Several techniques have been proposed for fault injection. Generally they can either be based on specific hardware, system simulation, or software. Hardware techniques inject physical faults in the target system hardware. Simulation techniques make use of a simulation model of the target system. Finally, a third solution is to emulate hardware faults and errors through software (Software Implemented Fault Injection, or SWIFI for short).

The motivation behind our work was the development of a flexible and portable tool to inject faults in the advanced computers available today. These computers are usually built around high speed RISC processors which typically have high transistor densities, high clock frequencies, large internal caches, and advanced components such as branch prediction units. Superscalar architectures also have replicated arithmetic units to execute machine instructions in parallel. All these factors together pose new challenges to fault injection. Traditional techniques such as hardware fault injection, although appropriate for simpler and older processors, are presently not easy to apply due to the difficulties in controlling and observing the faults effects inside the chips. Other techniques, such as simulation, are also difficult to apply because simulation models of these processors are very complex, and are often considered critical and confidential information by the manufacturers, thus being very difficult to obtain.

However, while modern processors' complexity difficults the application of some fault injection techniques, it provides several potential benefits to others. Within the millions of transistors

that make up these processors (~5 millions for the PowerPC 604), architects included advanced debugging and performance monitoring mechanisms. These new features are accessible to software (through privileged machine instructions) and thus can be directly used by SWIFI tools.

This idea was behind the development of a new software implemented fault injection and monitoring environment, called Xception. Unlike previous SWIFI tools, Xception can inject faults with minimum interference with the target application by directly programming the debugging hardware inside the target processor. The sophisticated debugging exception mechanisms available allow the definition of many fault triggers (events that cause the injection of the fault), including fault triggers related with the manipulation of data.

On the other hand, by using the performance monitoring hardware inside the processor, Xception can record detailed information on the target processor behaviour after the injection of a fault. Some examples are the number of clock cycles, the number of memory read and write cycles, and instructions executed (including specific information on instructions such as branches and floating point instructions) from the injection of the fault until some other subsequent event, for instance the detection of an error (latency). Furthermore, by combining the exception triggers provided by the debugging hardware and the performance monitoring features of the processor, Xception can monitor other aspects of the target behaviour after the fault. For example, it is possible to detect if some memory area was accessed after the fault or if some program function was executed.

Another important aspect is that, because Xception operates very close to the hardware (at the exception handler level), the injected faults can affect any process running on the target system including the kernel. It is also possible to inject faults in applications for which the source code is not available. In addition, the comprehensive fault triggers of Xception makes it suitable for the emulation of software faults, as proposed by Christmansson and Chillarege [2]. In fact, the set of rules to emulate software faults proposed in [2] are difficult, or even impossible, to fulfill by SWIFI tools based on traps or hardware implemented fault injection.

The target system is regarded by Xception as formed by the processor, memory and data/address buses. Injected faults can directly emulate physical faults affecting the following internal target processor units: Data Bus, Address Bus, Floating Point Unit, Integer Unit, Memory Management Unit, General Purpose Registers, Branch Processing Unit and Main Memory. Presently, Xception has been implemented on a Parsytec parallel machine built around the PowerPC 601 processor and running the PARIX [49] operating system (a UNIX alike operating system for parallel machines).

The structure of this paper is as follows: Section 2 describes related research in the fault injec-

tion field. Section 3 discusses the advantages and the problems of the SWIFI approach. The design and implementation of Xception is described in Section 4. This section also presents the processing debugging and performance monitoring features used by Xception, the fault model, and the mechanisms used to inject faults at the low level. Section 5 demonstrates Xception's capabilities and presents results obtained in preliminary experiments. Finally, Section 6 suggests some future work and Section 7 concludes the paper.

## **2. Related Research**

Fault injection has been widely used in the past to evaluate the dependability properties of systems or simply to validate specific fault handling mechanisms. This section summarizes the most relevant work in the area. For other more specific or detailed surveys in the area, the reader is referred to [41,42,43,44].

### **2.1 Hardware Implemented Fault Injection**

A popular approach consists of injecting physical faults into the target system hardware. Several methods have been used, such as pin-level fault injection [1, 3], heavy-ion radiation [4], power supply disturbances [5], and electromagnetic interferences [45]. These methods have the inherent advantage of causing actual hardware faults, which may be close to a realistic fault model. However, all these approaches require special hardware and in some cases (e.g. pin-level injection) the high complexity and high speed of the processors available today make the design of the required special hardware very difficult, or even impossible. The main problem is not in the injection of the faults itself but is related to the difficulties of controlling and observing the fault effects inside the processor. Even the detection of the activated faults is very complex. For example, the injection of faults in processor pins require the use of complex monitoring hardware to know whether the injected faults have produced internal processor errors or not [6]. Similarly, techniques such as heavy-ion radiation and power supply disturbances require the target chip outputs to be compared pin-by-pin and cycle-by-cycle with a gold unit in order to know whether the injected faults have produced errors inside the target chip or not.

### **2.2 Fault Injection by Simulation**

Simulation based fault injection has also been proposed for dependability evaluation. In this approach faults are injected into a simulation model of the target system which allows to control the timing, the type of fault, and the affected component in the model with more or less accuracy depending on the level of abstraction of the simulator. One of the advantages of this technique, that

makes it appealing to system manufacturers is that it can be used early in the design process. With a simulator it is also possible to inject very precise faults and collect detailed information on their effects. However this technique involves developing an accurate simulation model of the target system which can be very time consuming for complex systems.. Furthermore, the simulation models are not usually available from the manufacturers. Some recent examples of simulation-based fault injection tools can be found in [7, 8].

### 2.3 Software Implemented Fault Injection (SWIFI)

SWIFI techniques alter the hardware/software state of the system using special software in order to cause the system to behave as if a real hardware fault occurred. One of the early approaches is FIAT [9], which enabled the corruption of a task's memory image. The selection of the fault location was made by the user at the application level and the physical location within the memory image was obtained from compiler and loader information. Although this work provided valuable results, it was not able to inject transient faults.

The concept of failure acceleration was introduced by Chillarege in [10] where faults were injected by modifying memory contents under software control.

Another tool named DOCTOR [11] is capable of injecting processor, memory and communication faults on a distributed real-time system called HARTS. Processor faults are injected by modifying the applications executable image, specifically changing some instructions generated by the compiler and inserting extra instructions.

In the FERRARI [12] approach, the UNIX *ptrace* function is used to corrupt the process memory image in run-time and insert software trap instructions at the specific instruction addresses where faults should be activated. This tool allows the injection of transient faults and provided valuable results from experiments conducted on a Sparc workstation.

Another tool named FINE [13] has been proposed to inject faults and monitor their effect by using a software monitor to trace the control flow. However, this tool needs the source code of the target application and causes a large overhead. DEFINE [14] is an evolution of FINE that include distributed capabilities. It modifies the programs' executable image in order to emulate memory faults, e.g. by inserting software *traps* at specific memory locations in the text segment. DEFINE also enhances FINE by introducing a modified hardware clock interrupt handler to inject CPU and bus faults with time triggers. Finally, in addition to hardware faults, DEFINE is also capable of injecting some kinds of software faults, i.e. software design/implementation faults. FTAPE [47] was used to inject faults in three prototypes of a commercial fault-tolerant computer. FTAPE is part of a benchmark for characterizing the fault-tolerance of a system and it also in-

cludes a synthetic program to generate CPU, memory and I/O activity. FTAPE injects faults in the CPU, memory and I/O, and can select the time and location of the fault randomly, or based on workload activity measurements. This last technique is known as “stress-based injection” and assures that faults are injected in components undergoing high activity.

The injection of fault types specific to parallel and distributed systems have also been a major concern in several SWIFI tools such as DOCTOR [11], DEFINE [14], EFA [15] and CSFI [16]. These tools are able to inject faults in the communication subsystems of their target systems through software and have been used for several purposes, such as evaluating distributed diagnosis algorithms, the fault tolerant capability of algorithms, or the overall effect of communication faults in parallel applications.

## **2.4 Hybrid Fault Injection**

The hybrid approaches result from a mix of any of the previous techniques. SWIFI was used along with some extra hardware in a recent version of FERRARI [17] to help in the fault injection process, and in HYBRID [18] to trace fault activation and propagation in the target system. Another possibility consists of mixing SWIFI with simulation [19] to take advantage of both the speed of the actual target processor and the accuracy of low-level fault models. Finally, in [20] the dependability properties of the Motorola MC88100 RISC processor were evaluated using a mix of software and simulation techniques.

## **3. SWIFI: Problems and Solutions**

The actual trend in fault injection seems to be the use of SWIFI tools. The advantages of SWIFI are manifold: they use real hardware and software, they are less complex and costly and incur in less development effort than the other techniques. They are easily expanded (for new classes of faults), quite portable, and finally there is no problem with physical interferences or risk of damaging the target system as in physical fault injection. However, existing SWIFI tools still have several major problems which are discussed in the following subsections.

### **3.1 Impact in the target system**

One of the most important problems of SWIFI comes from the fact that existing software fault injection tools have considerable impact on the target system behaviour, either because part or all of the code of the tool has to be executed in the target system (i.e., it becomes part of the target workload) or because the target processor may have to run in trace mode. Previous research of different natures [3, 21, 22, 23] have emphasised the impact of the workload on the performance

of the fault handling mechanisms, which means that the software fault injection tools interfere with the results.

In Xception the impact in the target workload is quite small in both time and space. Concerning time, the exception that triggers the injection of the fault is programmed in the processor debugging hardware before starting the target application. Therefore, the processor is executed at full speed until interrupted by the trigger exception to perform the injection itself. The time spent in exception handlers measured in the current version for the PowerPC ranges from 1  $\mu$ sec to 5  $\mu$ sec, depending on the functional unit to affect. This is a value that can be accommodated in the time constraints of many real-time systems, and thus makes Xception suitable for use in these systems. Concerning space, with Xception the target application does not need to be changed. In addition, the Xception modules resident in the target system (low-level exception handlers and the fault injecting code) occupy as little as 30 Kbytes.

### **3.2 Fault Triggers**

Another problem of existing SWIFI tools is related with the restricted range of fault triggers. Faults are injected either by corrupting the memory image of the application, by inserting traps, or by replacing one set of instructions by another set of instructions. All these methods are related to instruction execution, and no fault triggers related to data manipulation can be defined. In FERRARI [12], DEFINE[14] and FTAPE[47], faults can also be injected by defining a temporal trigger. One advantage of this method lies in the fact that the fault is (almost) always injected, while it is not related with any specific action of the target application. However, faults injected in this way cannot be reproduced, because the system clock used is not accurate and the application execution usually has time uncertainties. It is nevertheless worth noting that this might not be a problem when fault injection experiments are aimed at fault tolerance coverage estimation.

In Xception, a comprehensive set of fault triggers related to instruction execution, (some) data manipulations, and temporal features are available. The temporal triggers are implemented by using the internal timer available in most of the modern processors.

### **3.3 System Monitoring**

The target system monitoring is another problem of existing SWIFI tools. Monitoring is required either for detecting the activation of faults or to collect relevant information on the fault impact. Only few proposals handle the monitoring issue, either by using extra instrumentation [18], by using software monitors [13], or by inserting trap instructions in the adequate locations [14]. The first method needs extra (and complex) hardware, while the other methods cause great

execution overhead and do not achieve detailed monitoring.

In Xception, the use of the dedicated performance monitoring and debugging hardware inside the processor greatly facilitates the monitoring of the target system in the presence of faults.

### 3.4 Accuracy

One of the most long standing argument against SWIFI states that it's accuracy, i.e. it's ability to emulate device-level faults (generally considered as the real faults) is very reduced.

In any fault injection approach it is important to guarantee that the errors produced by injection are as close as possible to the errors produced by real faults. As the ultimate goal of fault injection is the validation of fault handling mechanisms, the set of errors produced by more accurate injections will validate these mechanisms more accurately. In general, lower levels of abstraction provide high accuracy at a higher cost, and higher levels of abstraction provide various levels of accuracy at a lower cost. Specifically for SWIFI, some attempts were recently made to demonstrate its accuracy, with reasonable success. An early tool, developed with this goal in mind was called EMAX [24] and used a Zycad hardware simulator to inject faults into the gate/transistor level description of a circuit.

Another work [25] showed that over 80% of gate-level fault manifestations (errors) do not lead to errors or can be represented by SWIFI techniques. The author used a simulation model of a microprocessor in which gate-level faults were injected. An analysis procedure examined errors occurring from the injected faults and categorized them according their ability to be emulated by SWIFI.

A recent work [26] introduced a microprocessor error behavior function (EBF) that maps faults into errors on the functional level. The study concluded that pin level fault injection was only able to emulate 9-12% of the bit-flip faults. On the other hand it showed that 98% to 99% of device level faults in the processor could be emulated by software implemented tools (SWIFI).

Another recent work from Charles Yount at Carnegie Mellon clarified many issues concerning the representativity of SWIFI. To have the best of both worlds, ASPHALT [19] mixes SWIFI and Fault Simulation to achieve a higher accuracy at a low cost and provided very encouraging results for the SWIFI techniques.

### 3.5 Intrinsic Limitations

Some processor resources and structures, such as the bus control lines and peripheral devices, can not be directly reached by SWIFI tools. For instance, no SWIFI injected fault can cause errors in the low level bus control timings (although we can argue that their consequences can be emulated by data bus errors).

In the same way, the injection of faults in the target memory is partially limited by external logic implementing parity checks or error detection and correction. That is, the memory content is corrupted by the faults but the errors cannot be detected by parity, as the parity bits are set accordingly by the external logic.

These are intrinsic problems of SWIFI for which we can only envision one solution: the use of additional hardware support (hybrid solution).

### **3.6 Portability to other Processors and Systems**

Another important problem with existing SWIFI tools is their portability to other processors. SWIFI tools can largely benefit from modularity in design by separating the low-level and processor specific code from the higher-level fault injection modules. The changes required when porting a SWIFI tool to another system can thus be restricted to the low-level module (usually a device driver). This is the case with, e.g. FTAPE [47] and also Xception.

In Xception, the changes in the lower-level module comprise the adaptation of the exception handling code (written in C) to the specific target processor debugging and performance monitoring features. While the exception handling code itself is rather simple, the integration with the kernel can pose some problems. In the current implementation of Xception, this code was integrated with the PARIX [49] kernel much like a device driver.

The basic requirement to implement Xception on other processors is, however, the existence of the debugging and performance monitoring features. We investigated several contemporary processor architectures to check about the existence of these features with success.

The HP Precision architecture [27] provides an optional SFU (Special Function Unit) for debugging. It supports separate registers sets for data and instruction breakpoints allowing even more sophisticated fault triggers than the PowerPC. There are also dedicated instructions to manipulate the debugging unit registers.

The Pentium processor has a comprehensive set of performance counters similar to the ones existing in PowerPC and it also has four breakpoint registers for establishing breakpoints. Although some of these features are not documented, and are only available through a non-disclosure agreement with Intel, Pentium debugging and performance monitoring features have been reverse-engineered and published in [28].

The Alpha AXP architecture [29] is a 64 bits load/store RISC architecture designed with particular emphasis on clock speed, multiple instruction issue and software migration. Although its debugging facilities are reduced, it includes performance monitoring features like several registers to count hardware events and perform an interrupt upon counter overflow. It also includes an en-

able/disable bit for floating point instructions. All these features are similar to the ones existing in the MPC604.

The MIPS R4400-R10000 processor family [30], although having reduced performance monitoring facilities, contain four special debugging registers implementing comprehensive debugging features similar to the ones existing in the PowerPC.

The Power2 architecture [31] includes a monitor containing several counters for counting instruction execution and data storage events up to a maximum of 320 user-defined events. This includes counting the number of fetched, dispatched, and executed instructions, floating point instructions, number and type of storage operations, etc. It also includes an Instruction Match Register to count the occurrence of specific instructions.

It becomes clear from the short survey presented above that the concept of using the debugging and performance monitoring features for dependability evaluation by fault injection has a wide applicability.

#### 4. Design and Implementation of Xception

Xception consists of three modules, shown in dark gray in Figure 1: A kernel module, a fault setup module, and the Experiment Manager module. The kernel module is a small (20 Kbytes) module statically linked with the kernel of the target system. It consists of the exception handlers (basically glue code) and the code performing fault injection. A kernel incorporating Xception is provided to the user and is used for all fault injection experiments instead of the normal kernel. The fault setup module is a library of functions whose only task is to receive the fault parameters from the host (via message passing) and pass them to the kernel. Fault setup is accomplished by invoking the library function, *StartXception()* from any process in any processor in the target system. It can be invoked by the target application or from a dedicated process if the application source code is not available (as is the case in Figure 1). Finally, the EMM runs on a host system (presently a SUN Sparc) and provides the user interface for fault definition, automatic fault injection experiment control, and collection of results.

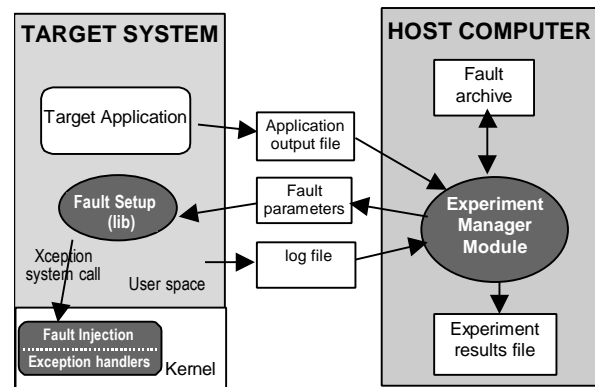


Figure 1. Xception structure

Finally, the EMM runs on a host system (presently a SUN Sparc) and provides the user interface for fault definition, automatic fault injection experiment control, and collection of results.

The target system of Xception is regarded as composed by the processor, system buses and memory. The processor is further divided in its functional units as is described in more detail in

Section 4.4.1. The target can be a single processor system or a multiprocessor, the only difference being that multiprocessors need an additional parameter which is the processor number to inject. The present implementation of Xception is targeted for a parallel machine based on PowerPC processors: the Parsytec PowerXplorer.

#### 4.1 Processor debugging and performance monitoring features used by Xception

The recently introduced performance monitoring and debugging features consist mainly of performance counters and breakpoint registers. The former count user defined events such as load, store, or floating point instructions. The latter enable the programmer to specify breakpoints for a wide range of situations such as load, store or fetch from a specified address or even some instruction types (e.g. floating point instructions). These features are accessed through privileged instructions and are mostly used by debugging and performance analysis tools.

The present implementation of Xception is targeted for systems based on the PowerPC processor family, more specifically for the MPC601 [32]. The PowerPC will therefore be used as a case study. Table 1 shows the list of exceptions types of the MPC601 and MPC604 used by Xception. It is worth recalling that unlike other injection tools, Xception uses hardware exceptions and not software *trap* instructions.

The Decrementer exception is used by Xception to trigger fault injection after a user specified time (clock ticks), thus providing fault trigger definition in a temporal way. Run Mode and Data Access exceptions are used to define fault trigger in a spatial way. For example, faults can be injected when the instruction in a specific address is fetched or when the data stored in some ad-

Exception Type	Causing conditions
Data access	Occurs if the address used in a load or store operation matches the address in a special register named DABR (Data Access Breakpoint Register)
Run mode	Occurs when the effective address (EA) of the instruction being decoded matches the EA contained in IABR (Instruction Address Breakpoint Register)
Trace	When the MPC601 runs in Trace mode, a trace interrupt is taken after each instruction that completes without causing an exception or context change.
FPU unavailable	This exception occurs when no higher priority exception exists, an attempt is made to execute an FP instruction and the FPU is disabled (although the MPC601 have an internal FPU, it can be logically disabled).
Decrementer	The MPC601 includes a programmable register (Decrementer) which decrements its contents at a fixed frequency and generates an exception (if not otherwise masked) after reaching zero.

Table 1. MPC601 (and MPC604) exceptions used by Xception

dress is accessed. Experiments performed by using the spatial method can generally be reproduced because they depend on a specific address. On the other hand, in the temporal trigger method, faults cannot be reproduced due to execution time uncertainties. Trace mode is used basically just to execute the machine instruction affected by an injected fault. This is because to inject transient faults, the Xception methodology (explained in detail in Section 4.6) requires the control of the processor to be taken immediately after executing the corrupted instruction. It is worth noting that this exception is only used at the point of injection,

because otherwise it would slow down program execution in an unacceptable way.

Finally, the Floating Point unavailable exception is used to “locate” FP instruction to be injected to emulate faults in the FPU. First, the FPU is disabled, and when the processor attempts to execute an FP instruction the FPU-unavailable exception is raised, the injection itself is performed at that location, the FPU enabled again, and the processor resumed.

## 4.2 Fault Duration

Xception was primarily designed to emulate hardware transient faults in functional units of the target processor. In fact, previous studies [33, 34] have shown that the vast majority of physical faults affecting digital systems are transient. The emulation of permanent faults by software is difficult. For example, the emulation of a permanent fault in a processor register requires that the content of the register be forced to the wrong value whenever the register is used by the program. The only way of doing this is by executing the program in trace mode, which causes an unacceptable impact on the application execution in most of the cases. For the above reasons, permanent faults have not been implemented in the first Xception version. However, it should be noted that it is possible to emulate permanent faults in the memory without forcing the program to be executed in trace mode, by programming an exception to be activated whenever the faulty memory cell is addressed. Thus, the content of the memory cell can be forced to the desired (wrong) value.

## 4.3 Fault Location

From the fault definition and fault injection point of view the target system processor is presented in more detail by considering the major internal processor units. In spite of the specific characteristics of each possible target processor, it is still possible to define an abstract functional model capable of representing the possible target processor architectures. This functional target processor units considered are listed in the first column of Table 2.

For the sake of portability to other processors, Xception provides an interface for fault definition based on the above units (plus memory faults). Thus, the user can define a single set of faults and perform fault injection experiments in several microprocessors, having a common basis for comparing the results. Of course, implementations of Xception for different target processors inject faults in the same functional units but using the processor’s particular debugging and performance monitoring features.

It is worth noting that in some cases the mechanism used to emulate a fault in one particular functional unit also emulates faults in other processor units. Furthermore, the fault triggers (see next section) should also be considered when mapping software injected faults to the emulated hardware faults. For example, a fault injected in the Data Bus during an operand read emulates

Fault Locations	Fault Triggers	Fault 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 affect
Branch Processing Unit (BPU)		

**Table 2. Fault Parameters.**

faults in the Data Bus itself and in the destination register. On the other hand, a fault injected in the Data Bus during an opcode fetch can also emulate faults in the Instruction Execution Control Unit.

#### 4.4 Fault Trigger

Another important fault parameter is the trigger condition, i.e. a processor execution condition or external event that leads to the injection of the fault.

As shown in Section 4.3, Xception uses five different types of exceptions to interrupt the processor at specific moments. The occurrence of such exceptions is used to trigger the faults. Fault triggers can thus be defined both in temporal and spatial terms. The former causes a fault to be injected after a predetermined amount of time elapses (processor clock cycles) since the start of the application. A spatially defined fault is injected when the program accesses a specified memory address, either for data load/store or instruction fetch. The possible fault trigger conditions are listed in the second column of Table 2.

#### 4.5 Fault Types

The Fault Type defines exactly what is corrupted and how is that corruption performed. As the definition of the faults is related to the target functional units, the definition of fault types should take into account the actual target structure. For example, to define a fault in the processor registers it is important to know the target register map and the size of the registers.

In the case of the PowerPC the fault definition is very simple. In fact, the MPC601 address bus is 32 bits wide, all the instructions are encoded as single-word (32 bits) opcodes, and registers also are single-word sized (although the data bus is 64 bits wide). Thus, the fault type can be completely defined by a 32 bit fault mask in which the bits to be affected are set to '1' and the bits that should be left untouched set to '0'. The injection of the fault consists only in performing a specific logic operation between the fault mask and the adequate processor register or memory cell. Several bit level operation can be used: stuck-at-zero, stuck-at-one and bit flip (see third column of Table 2).

To define large quantities of faults required for automatic fault injection, the user defines the fault mask with the range of bits to be affected set to ‘1’, the number of bits to be affected simultaneously, and the operation used to corrupt the randomly selected bits from the mask. Using this process it is possible to emulate many different fault types just by selecting the appropriate bits and operations. For instance, a fault with the mask *0x000000FF*, of the bit-flip type and with the number of bits to be affected set to 1 would cause one-bit-flips in the lower byte of the affected value. A mask of *0xFFFFFFFF* of the bit-flip type and with the number of bits to be affected set to 2 would cause any two-bit faults across the whole value.

In processors where the architecture is not so regular, and instructions do not have a fixed size, such as the Intel Pentium, the user would have to define masks for 8, 16, 32 and 64 values.

#### 4.6 Data Collection and Analysis

For each injected fault, Xception collects the fault injection results and saves them in file using a spreadsheet format. After the completion of the fault injection experiment, this file can be analysed using a spreadsheet to make the postprocessing treatment that best matches the user interests. The results collected for each injected fault are described in Table 3. Errors at the application level may be detected by system built-in error detection mechanisms such as *illegal access* and *illegal instruction* and cause the application to be aborted. In addition, the kernel generates an error message to the host identifying the error condition. These messages and the Injection Context (see Table 3) are directed by the Xception kernel module to a log file (see Figure 1) to be retrieved later by the EMM. The output generated by the target application under fault injection can be compared with the output generated in the *gold run* using a general built-in compare function or using an alternative user-defined comparison program.

The Injection Context gives the user information about what was exactly affected by the fault (specially useful for time-based triggers) and contains timing information to enable the calculation of error detection latencies in the host.

Fault Injection Results	
<b>Fault Parameters</b>	Fault trigger, type, and location.
<b>Injection Context</b>	Processor state at time of fault: instruction pointer, opcode of the affected instruction and internal clock value.
<b>Execution Time</b>	Execution time of the application measured from the host.
<b>Exit Code</b>	Application's exit code (e.g. returned by the <code>exit()</code> call in the C Language ).
<b>Kernel Errors</b>	Error detected and reported by the PARIX kernel.
<b>Application output Correctness</b>	Result of comparison of the program output with the gold run output. BIGGER: Bigger size SMALLER: Smaller size DIFFERENT: Different contents EQUAL: Same size and contents NONE: No output file found EMPTY: Empty output file
<b>Error Detection Mechanisms</b>	Errors detected added by Error Detection Mechanisms (EDM) in the target.
<b>Monitoring information</b>	Reserved for Information collected from the processor performance monitoring hardware.

Table 3. Results collected by Xception

## 4.7 Fault Emulation

The injection of a fault comprises several steps. The very first step (before the actual injection of the fault) is to program a time trigger in the Decrementer, or a spatial trigger in the processor debugging logic. When the fault trigger (exception) occurs, specific registers and/or memory are manipulated and corrupted according to a predefined strategy in order to emulate a fault in the functional unit specified. Depending on the target functional unit and trigger type, a third step may be required to restore original memory/register contents. As we will see in the detailed description that follows, this last step is necessary in several situations to guarantee that real transient faults are emulated.

In this paper the injection procedure is only described in detail for three types of faults, which are the FPU, IU, and the Address bus faults triggered by an operand fetch. These cases have been selected because they are illustrative of the low level injection mechanisms used. For a detailed explanation of the other cases, the reader is referred to [35].

### 4.7.1 Faults in the Address bus when an opcode is fetched

Faults in the address bus are emulated in a different way for each trigger specified (opcode fetch, operand load and/or store, or time). Figure 2 illustrates the strategy used to inject an error when an opcode is fetched. When an exception is raised in the MPC601, the address of the causing instruction is automatically saved in special register SRR0 (Save/Restore Register) before jumping to the exception handler. This address will be used later when returning from the exception handler routine to resume execution. As shown in Figure 2, when the trigger instruction (Instruction A) is reached and Xception gains control over the processor, the register SRR0 is corrupted. Execution is then allowed to resume in Trace mode at the corrupted address. If no serious error occurs, such as an access to an invalid address, a Trace exception is raised immediately after the instruction at the corrupted address (instruction P) is executed. Xception then has to decide where to resume execution (value to store in SRR0). To accomplish this task it has to distinguish between three cases (see Figure 2): instruction P is a relative branch, an absolute branch, or is not a branch.. If it is a relative branch, then from the processor point of view the branch should be taken relatively to the address where it thinks the instruction came from, i.e.0x1000. However, if

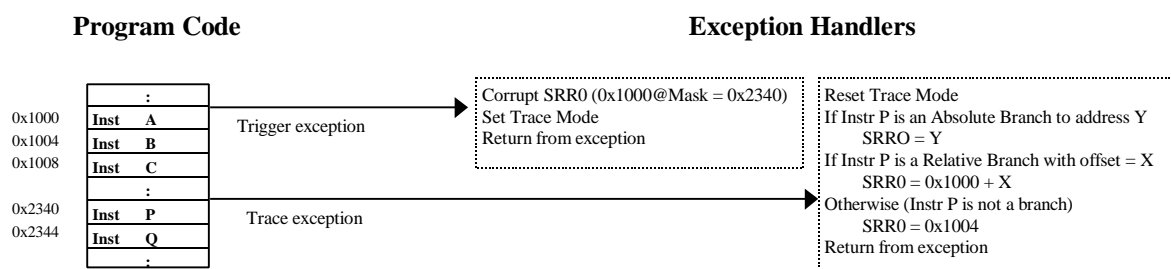


Figure 2. Transient address bus error when an opcode is fetched

the branch is absolute the address where it was (or it thinks it was) executing before is not relevant and the absolute branch is taken normally. If P is a not a branch instruction, then the internal

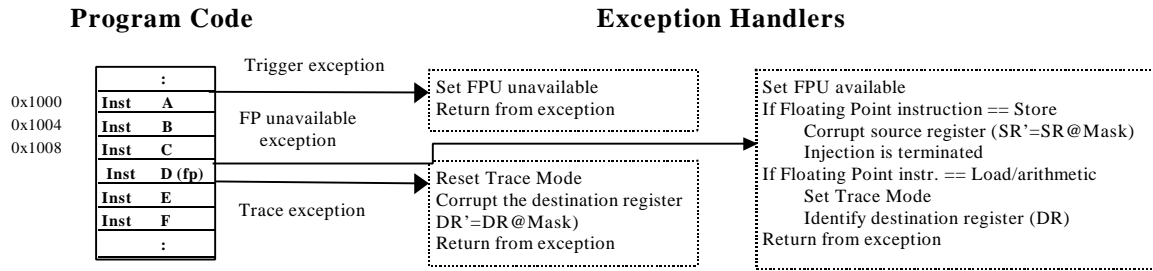


Figure 3. Transient Fault in the Floating Point Unit

IP will be incremented as usual and the instruction at 0x1004 (instruction B) is executed next. This complete procedure is necessary to correctly emulate a transient fault in the bus, outside the processor. As far as we know, previous work on this subject did not take into account such subtle issues.

#### 4.7.2 Faults in the Floating Point unit

The consequences of faults in the Floating Point Unit (FPU) are always the generation of erroneous results into the FP registers (i.e., we consider that FPU faults occur always during the use of the FPU by an FP instruction). This method is illustrated in Figure 3. After reaching the fault trigger, Xception disables the FPU and normal execution is resumed. When the processor tries to execute a floating point instruction, a floating point unavailable exception is raised and Xception takes control again. If the FP instruction is a *store*, then the source register is corrupted and execution is resumed. If it is a *load* or an FP arithmetic instruction, the destination register is identified, the instruction is executed in trace mode and the destination register is corrupted afterwards. Faults in the FPU are injected using the same method no matter the trigger type used.

#### 4.7.3 Faults in the Integer Unit

Faults in the Integer Unit (IU) may cause the generation of erroneous results into the General Purpose Registers. In this model, after reaching the fault trigger, the processor is switched to Trace mode and each subsequent instruction is decoded to check if it is an instruction that uses the IU (which are very frequent). When such instruction is found, Xception alters the contents of

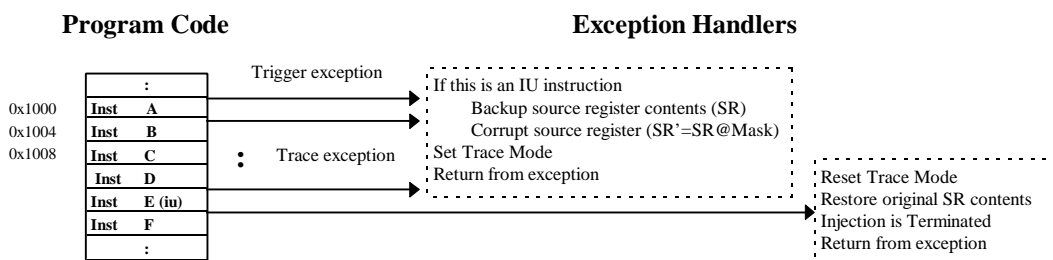


Figure 4. Transient Fault in the Integer Unit

the instruction's source register and lets the processor execute it with the wrong argument in trace mode. Afterwards, the original contents of the input register are restored and the processor is allowed to resume at full speed. Figure 4 illustrates this method. In this example the processor executed instructions A to D in trace mode until it found an integer instruction (E) where the fault was injected.

## 5. Using Xception

This Section presents empirical results on the evaluation of the impact of faults in parallel applications running in a commercial system with no particular fault handling mechanisms. The main objective is to show the detailed raw results (readouts) and the accurate fault impact and characterization that can be obtained with Xception. Aspects of the whole fault injection process such as the statistical treatment required to obtain accurate assessment of coverage [36] are not considered. Experiments have been performed in a Parsytec PowerXplorer with four nodes, running the PARIX [49] operating system. Each node contains an MPC601 for computation, a T805 transputer dedicated to the communication with the neighbour nodes, and 16 Mb of RAM. More experimental results obtained using Xception can be found in [48]

### 5.1 Benchmarks

The parallel benchmarks selected have been developed using a high-level Linda parallel programming library build on top of PARIX, called ParLin [37], as well as in C by directly using PARIX message passing primitives. The benchmarks were selected in order to be representative of real parallel applications. A short description of the benchmarks is given below:

#### 1. **p** Calculation (Linda)

Computes an approximate value of **p** by numerically calculating the area under the curve  $4/(1+X^2)$ . The area is partitioned in N strips by a Master program and each job is assigned a subset of the total strips. This jobs will be carried out by Workers that return to the Master their part of the total sum. The final calculated value for **p** is stored in a file by the master.

#### 2. **SOR Successive Overrelaxation.**

An implementation of a parallel algorithm to solve the Laplace equation over a grid [38]. The algorithm is based on the popular overrelaxation scheme with red-black ordering.

#### 3. **MATMULT - Matrix Multiplication (Linda)**

A matrix multiplication program following the master worker paradigm with ABFT(Algorithm Based Fault Tolerance) [46]. Each worker enrolled in the computation is responsible for calculating a part of the result matrix (119x119). The input matrixes contains integer values and have

an extra column and line used to calculate a checksum.

For space reasons, detailed results are only presented for the **p** Calculation application. For SOR and MATMULT applications just a summary of the main results is shown.

## 5.2 Experiment Setup

Fault sets of 3000 “typical” faults have been injected in each application affecting several functional processor units. Figure 5 shows detailed information on the faults injected in the **p** Calculation application.

<b>No. faults</b>	3000		
<b>Non injected</b>	330		
<b>Injected</b>	2670	<b>Address bus</b>	680
		<b>Data bus</b>	379
		<b>PFU</b>	559
		<b>IU</b>	349
		<b>MMU</b>	703

Figure 5. Number of faults for **p**.

One-bit-flip faults affecting the whole range of 32 bits

in a word were injected by using time based triggers. The choice of time triggers was only driven by their characteristics which assure that faults have a high probability of being injected. However, even with this kind of fault trigger some faults are not injected due to asynchronies between processors in the parallel target system. For example, as shown in Figure 5, 330 faults have not been injected because when the time for injection (counted from the start of the application) has expired the target processor had already finished its task. It should be noted that in a parallel application the load is not in general equally distributed among processors, which means that it is difficult to bound the global execution time.

The bit flip model is appropriate because previous research proved that 90% of transistor level faults in sequential circuits [39] manifest as single bit-flips. In addition physical heavy-ion fault injection experiments on the 68000 showed that 90% to 95% of all errors are single bit-flips [40].

## 5.3 Results

Table 4 to Table 7 show fault injection results for the  $\pi$  calculation. Each table presents specific information concerning the behaviour of the system under fault injection. Table 7 summarises the information by gathering the results of all tables and classifying the fault impact in three main classes: *Undetected*, *Detected*, and *No Error*. From the user perspective, the worst cases are the *Undetected*, as they represent faults which have not been detected by any means and have caused the generation of erroneous results by the application.

These results show (Table 5) that the impact of faults is highly dependent on the affected functional unit. Consider for instance, faults in the address bus: they lead to 64,7% of system crashes, i.e.abortio through timeout (see Table 5) and about 33,9% (sum first three lines of Table 4) were detected by the kernel because they caused Program, Data, or Instruction accessed exceptions.

Errors detected	Address Bus (%)	Data Bus (%)	FPU (%)	IU (%)	MMU (%)
Program exception	5,2	5,5	0,0	3,2	5,7
Data access exception	13,5	21,5	0,0	3,2	14,7
Instr. access exception	15,2	0,0	0,0	8,3	16,6
Alignment exception	0,0	0,5	0,0	0,0	0,0
Other errors	0,0	0,5	0,0	1,2	0,0
No error messages	66,1	72,0	100,0	84,1	63,0

**Table 4. Errors detected by MPC601 built-in Error Detection Mechanisms (EDM) for the p application (2670 faults)**

Output results	Address Bus (%)	Data Bus (%)	FPU (%)	IU (%)	MMU (%)
None	65,4	30,3	0,0	22,1	65,0
Empty	0,0	0,0	0,0	0,0	0,0
Equal	8,7	25,6	26,8	4,6	9,3
Different	25,9	44,1	73,2	73,4	25,8

**Table 6. Output results of the p application**

Program exit codes	Address Bus (%)	Data Bus (%)	FPU (%)	IU (%)	MMU (%)
Normal	34,6	69,7	100,0	77,9	35,0
Error in Parix calls	0,7	0,0	0,0	0,0	0,4
Error in Parlin	0,0	0,0	0,0	0,0	0,0
Timeout	64,7	30,3	0,0	22,1	64,6

**Table 5. Exit codes for the p application**

Summary	Address Bus (%)	Data Bus (%)	FPU (%)	IU (%)	MMU (%)
Undetected	25,7	43,8	73,2	72,8	25,8
No error	8,7	25,3	26,8	4,3	9,0
Detected	65,6	30,9	0,0	22,9	65,2

<b>Undetected</b>	No errors have been detected and the application produced wrong results (fatal fault)
<b>No error</b>	No errors have been detected but the application terminated normally and produced correct results
<b>Detected</b>	Some type of error have been detected (Timeout, bad exit code or MPC601 built-in Error Detection)

**Table 7. Summary of results for the p application**

Summarizing, 25,7% ended up classified as *Undetected* (Table 7). On the other hand, faults in the Data Bus did not have such a dramatic impact, as they only lead to the crash of the system in 30,3% of the cases (see Table 5). However, a larger percentage (43,8%) of *Undetected* faults (see Table 7) has been observed. This is due to the fact that some internal processor built-in error detection mechanisms that detect many address errors such as instruction access exceptions (see Table 4) are not effective in detecting data bus faults.

Another interesting result is that faults in the FPU and the IU (Integer Unit) lead to high percentages of *Undetected* cases, respectively 73,2% and 72,8% (see Table 7). This is explained by the fact that faults in these functional units only cause the generation of corrupted data into the FP or Integer registers files, and no instruction address or opcode is directly affected as in the Data/Address buses. Because the IU is also used to calculate addresses, 22,9% of the faults caused a system crash or access exceptions and were detected by the system (see Table 7). On the other hand, in the FPU, which only manipulates pure data, no error was detected at all. The errors at the FPU which did not lead to the generation of wrong results, correspond to cases where the FP instructions affected were not directly connected with the calculation of  $\pi$  (we later found that those affected FPU instructions were part of a time-of-day calculation library function).

Another issue investigated in this study concerns the dependency of the parallel system node chosen for injection in the overall effect of faults. The summary of results for the *Undetected* cases in each processor are shown in Table 8.

The large differences obtained for faults injected in different processors show that the overall effect of faults in parallel systems is highly dependent on the specific node affected by the fault. This is due to the different load distribution within each application and specific processor uses.

Undetected	Address Bus	Data Bus	FPU	IU	MMU
Processor 0	0,0	0,0	0,0	0,0	0,0
Processor 1	14,3	19,8	28,6	31,2	10,7
Processor 2	8,4	16,4	31,0	26,4	10,5
Processor 3	3,1	7,7	13,6	15,2	4,6
Total	25,7	43,8	73,2	72,8	25,8

UNDETECTED - No errors have been detected and the application produced wrong results (fatal fault)

**Table 8. Undetected cases by Processor and Functional Unit for the p application**

Summary	Address Bus (%)	Data Bus (%)	IU (%)	MMU (%)	GPRs (%)
Undetected	0,0	0,0	0,0	0,0	0,0
No error	12,1	24,4	2,7	10,6	81,9
Detected	88,0	75,6	97,3	89,4	18,1

**Table 10. Summary of results for MATMULT (2058 faults)**

Summary	With ABFT	Without ABFT
Undetected	0.0	21.8
No error	34.0	34.1
Detected	66.0	44.1

**Table 9. Coverage of Algorithm Based Fault-Tolerance in MATMULT**

Summary	Address Bus (%)	Data Bus (%)	FPU (%)	IU (%)	MMU (%)
Undetected	3.5	6.6	0.00	8.5	4.9
No error	19.8	48.2	99.4	36.2	19.7
Detected	76.7	45.2	0.6	55.3	75.4

**Table 11. Summary of results for SOR (2030 faults)**

Particularly, in this case, the Linda library maps the central repository of data (Tuple Space) in processor zero, and as there is no worker running therein, faults always have drastic consequences, but never lead to *Undetected* cases. On the other hand, processor 3 was less affected than 1 and 2 due to load balancing reasons.

Table 9 to Table 11 present a summary of the results obtained for the SOR and MATMULT applications. In the first table, the iterative nature of the SOR algorithm masked a high percentage of the faults and therefore we obtained a small number of *Undetected* cases in all functional units. The results show once more that the effects of faults are highly dependent on the functional unit where they occur.

Table 9 shows the coverage of the ABFT (Algorithm Based Fault Tolerance) mechanism embedded in MATMULT [23]. In spite of being an extremely simple method, which only implies the inclusion of an extra line and column in the result matrix, results show that the coverage was very significant. This fact is even more important because it detected all the 21,8% of faults that otherwise would cause *Undetected* wrong results.

Faults were not injected in the FPU as in this application the matrixes contain only integer values.

## 6. Future Work

The performance monitoring features of the PowerPC family are not fully explored in the current version of Xception because we started with the simplest member of the family (MPC601), which does not have special performance monitoring hardware. An upgrade of Xception for the MPC604 would use its advanced performance monitoring features, which can count many events, such as load, stores, correctly predicted branches, instruction or data cache misses, snoops, FPU

instructions, processor cycles, etc.. that can be collected by Xception for monitoring purposes. Furthermore, the performance monitoring hardware can be used in conjunction with the Run Mode and Data Access exceptions to monitor the access of specific memory addresses or the execution of specific program code after the fault.

Other topic that needs further research deals with the distribution of weights to the different functional units in order to make possible an accurate statistical analysis and the calculation of figures as the confidence level of the results. We envision the attribution of these weights by taking into account the frequency of use of the functional units by the application, along with the silicon area occupied by each of them in the processor chip. While to the latter, the information provided by the manufacturers is enough, for the former we plan the use of application assembly listing analysis (statical) as well as dynamic information collected from the performance counters (e.g. count the number of FP instructions executed by the application).

## 7. Conclusion

In this paper, a fault injection environment and monitoring named Xception is presented. Xception uses the advanced debugging and performance monitoring features existing in most of the modern processors to inject realistic faults by software, and to monitor the activation of the faults and their impact on the target system behavior in detail. This new methodology enables the definition of a general yet precise processor fault model with a wide range of fault triggers and oriented to the internal processor functional units. Furthermore, its flexibility and low overhead enables its use in a wide range of processors and machines including parallel and real time systems. In addition, Xception characteristics make it suitable for the emulation of software induced faults. Experimental results were presented to demonstrate the accuracy and potential of Xception in the evaluation of the complex systems available nowadays. These results show that the impact of faults depends heavily on the application and the specific functional unit where they occur. Particularly, faults at the FPU/IU proved to have a great impact as they were hardly detected and lead to the generation of wrong results in up to 73% of the cases for some applications.

## References

- [1] J.Arlat et al., "Fault Injection for Dependability Validation: A Methodology and Some Applications", IEEE Trans. on Soft. Eng., Vol 16, No 2, Feb. 1990, pp. 166-182.
- [2] J. Christmansson and R. Chilarege, "Generation of an Error Set That Emulates Software Faults Based on Field Data", in Proc. of FTCS-27, pp.304-313, Sendai, Japan, 1996..
- [3] H. Madeira and J.G.Silva, "Experimental Evaluation of the Fail-silent behavior in Computers without Error Masking", in Proc. of FTCS-24, pp. 350-359, Austin, USA, June 1994.

- [4] J. Karlsson, P. Lidén, P. Dahlgren, R. Johansson, and U. Gunneflo, "Using Heavy-ion Radiation to Validate Fault-Handling Mechanisms", in *IEEE Micro*, Vol. 14, No. 1, pp. 8-32, 1994.
- [5] G. Miremadi, J. Karlsson, U. Gunneflo, and J. Torin, "Two Software Techniques for On-line Error Detection", in *Proc. of FTCS-22*, 1992, pp. 328-335.
- [6] H. Madeira, M. Relá, F. Moreira, J. Silva. "RIFLE: A General Purpose Pin-Level Fault Injector", in *Proc. First European Dependable Computing Conference*, pp 199-216, Berlin, October 1994.
- [7] G. Choi and R. Iyer, "Focus: An Experimental Environment for Fault Sensitivity Analysis", *IEEE Transactions on Computers*, Vol. 41, No. 12, December 1992.
- [8] E. Jenn, J. Arlat, M. Rimén, J. Ohlsson, and J. Karlsson, "Fault Injection into VHDL Models: The MEFISTO Tool", *Proc. of FCTS-24*, pp. 336-344, Austin, TX, USA, 1994.
- [9] Z. Segall, T. Lin, "FIAT: Fault Injection Based Automated Testing Environment". In *Proc. 18<sup>th</sup> Int. Symp. Fault - Tolerant Computing.*, pp 102-107, June 1988.
- [10] R. Chillarege and N. Bowen, "Understanding Large Systems Failures - A Fault Injection Experiment", in *Proc. of 19<sup>th</sup> FTCS*, pp. 356-363, Chicago, June 1989.
- [11] S. Han, H. Rosenberg, K. Shin, "DOCTOR: an Integrated Software Fault Injection Environment", *Technical Report-University of Michigan*, 1993.
- [12] G. Kanawati, N. Kanawati, and J. Abraham, "FERRARI: A Tool for the Validation of System Dependability Properties", *FTCS-22, Digest of papers*, pp. 336-344, 1992.
- [13] W. Kao, R. K. Iyer, D. Tang, "FINE: A Fault Injection and Monitoring Environment for Tracing the UNIX System behavior under Faults", *IEEE Transactions on Software Engineering*, Vol 19, No. 11, November 1993
- [14] W. Kao, R. K. Iyer, "DEFINE: A Distributed Fault Injection and Monitoring Environment", *Workshop on Fault-Tolerant Parallel and Dist. Systems*, June, 1994.
- [15] K. Echtele, M. Leu. "The EFA Fault Injector for Fault-Tolerant Distributed System Testing", in *Workshop on Fault-Tolerant Parallel and Distributed. Systems.*, pp 28-35, 1992.
- [16] J. Carreira, Henrique Madeira, João Gabriel Silva. "Assessing the Effects of Communication Faults on Parallel Applications" in *Proc. of IPDS'95*, Erlangen, Germany, April 1995.
- [17] N.A. Kanawati, G. Kanawati, J. Abraham, "Dependability Evaluation using Hybrid Fault/Error Injection", in *Proc. IPDS'95*, pp. 224-233, Erlangen, Germany April 1995.
- [18] L. T. Young, R. K. Iyer, K. K. Goswami, and C. Alonso, "A Hybrid Monitor Assisted Fault Injection Environment" *3<sup>rd</sup> IFIP Working Conf. on Dependable Computing for Critical Apps.*, Sicily, Italy, Sept. 1992.
- [19] C.R. Yount, D.P. Siewiorek "A Methodology for the Rapid Injection of transient hardware Errors", *IEEE Transactions on Computers*, vol. 45, no. 8, pp. 881-891, 1996.
- [20] J. Guethoff, V. Sieh, "Improving the Efficiency of Fault Injection Based Dependability Evaluation", in *Proc. Of the Int. Symp. On Fault-Tolerant Computing (FTCS-25)*, Pasadena, CA,

pp.196-206, 1995.

- [21] R. Iyer and D. Rossetti, "A measurement-based model for workload dependence of CPU errors", *IEEE Trans. on Computers*, vol. C-35, pp. 511-519, June 1986.
- [22] E. Czeck and D. Siewiorek, "Effects of transient gate-level faults on program behavior", *FTCS-20*, Newcastle Upon Tyne, pp. 236-243, June 1990.
- [23] F. T. Luk, "Algorithm-Based Fault Tolerance for Parallel Matrix Solvers", *Proc. SPIE Real-Time Signal Processing VIII*, vol. 564, pp. 49-53, 1985.
- [24] G.Kanawati, N.Kanawati, J.Abraham, "EMAX: An automatic Extractor of High-Level Error Models", *AIAA Computing Aerospace Conference*, pp. 1297-1306, San Diego, CA, USA, October 1993.
- [25] E.Czeck, "Estimates of the Abilities of Software-Implemented Fault Injection to Represent Gate-Level Faults", Presented at *IEEE International Workshop on Fault and Error Injection for Dependability Validation of Computer Systems*, Gothemburg, Sweden, June 1993.
- [26] M.Rimen, J.Ohlsson, J.Torin, "On Microprocessor Error Behavior Modeling", *Proc. of the 24th Int. Symp. on Fault-Tolerant Computing (ftcs-24)*, pp. 76-85, Austin, TX, USA, 1994.
- [27] "PA-RISC 1.1 Architecture and Instruction Set Reference Manual", HP Part Number: 09/40-90039, Third Edition, February 1994.
- [28] T.Mathisen, "Pentium Secrets", *BYTE magazine*, pp. 191-192, July, 1994.
- [29] "DECchip 21064 and DECchip 21064a Alpha AXP Microprocessors Hardware Reference Manual", Order No.: EC-Q9ZUA-TE, Digital Equipment Corp., June 1994.
- [30] Joe Heinrich, "MIPS R4400 Microprocessor User's Manual", Mips Technologies, 1994
- [31] E.H Welbon, C.CChan-Nui, D.J.Shippy, and D.A.Hicks, "POWER2 Performance Monitor", *IBM Journal of Research and Development*, volume 38, No.5
- [32] "PowerPC601 RISC Microprocessor User's Manual" Motorola, July 1993
- [33] D.P. Siewiorek and R.S. Shwarz, "The Theory and Practice of Reliable Design", Digital Press, Educational Services, DEC, Bedford, Massachusetts, 1982,
- [34] P. K. Lala, "Fault Tolerant and Fault Testable Hardware Design", Prentice Hall International, NY, 1985.
- [35] J. Carreira, "Software Fault Injection in Parallel Systems", Technical Report, Univ. of Coimbra, Portugal, July 1995 (available from the web at <http://dsg.dei.uc.pt>).
- [36] D.Powell, E.Martins, J.Arlat, "Estimators for Fault Tolerance Coverage Evaluation", *IEEE Transactions on Computers*, Vol.44, No.2, February 1995.
- [37] J.G.Silva, J.Carreira, F.Moreira,"ParLin: From a Centralized Tuple Space to Adaptive Hashing".*Transputer Applications and Systems'94*, pp 91-104, IOS Press, 1994.
- [38] R.Chowdhury and P. Banerjee, "A Fault-Tolerant Algorithm for Iterative Solution of the Laplace Equation", in *Proc. Int. Conference on Parallel Processing*, pp. II-133 to III-140, 1993.

- [39] G.Ries, G.Choi, R.Iyer “Device-level Transient Fault Modeling”, Proc. of the 24th Int. Symp. on Fault-Tolerant Computing (ftcs-24), pp.76-83, Austin, TX, USA, 1994.
- [40] R.Johansson, “On Single Event Phenomena in Microprocessors”, TR162L, Dept. Of Computing Engineering, Chalmers Univ. of Technology, Gotenburg, Sweden, 1993.
- [41] Mei-Chen Hsueh, T.K.Tsai and R.K.Iyer, “Fault Injection Techniques and Tools”, IEEE Computer, vol.30, no.4, pp.75-82, April 1997.
- [42] J.Arlat, “Fault Injection for the Experimental validation of Fault-Tolerant Systems”, in Proc. Workshop Fault-Tolerant Systems, pp. 33-40, Kyoto, Japan, IEICE, Tokyo, Japan, 1992.
- [43] J.A.Clark and D.K.Pradhan, “Fault Injection - A Method for Validating Computer-System Dependability”, IEEE Computer, vol.28, no.6, pp.47-56, June 1995.
- [44] R.K.Iyer, “Experimental Evaluation”, in Proc. 25<sup>th</sup> Int. Symposium. On Fault-Tolerant Computing (FTCS-25). Special issue, pp.115-132, Pasadena, CA, USA, 1995.
- [45] 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”, in Proc. 5<sup>th</sup> IFIP Working Conference on Dependable Computing for Critical Applications (DCCA-5), pp.150-151, Urbana-Champaign, IL, USA, 1995.
- [46] K.H.Huang, J.A.Abraham, “Algorithm-based Fault Tolerance for Matrix Operations”, IEEE Transactions on Computers, vol.C-33,no.6,pp.518-528, 1984.
- [47] T.K.Tsai, R.K.Iyer, D.Jewitt, “An Approach towards Benchmarking of Fault-Tolerant Commercial Systems”, in Proc. of FTCS-26, pp.314-323, Sendai, Japan,1996.
- [48] J.G.Silva, J.Carreira, H. Madeira, D.Costa, F.Moreira, "Experimental Assessment of Parallel Systems", in Proc. of FTCS-26, pp.415-425, Japan, June 25-27, 1996.
- [49] “PARIX 1.3 -PowerPC, Software Documentation”, Parsytec Gmbh, 1994.

## Biographies:



**João Carreira** received a BS degree in Computer Science in 1991 and an MS degree in fault-tolerant distributed systems in 1995, both from the University of Coimbra, Portugal. He also worked for two years at SIEMENS SA as a software engineer in the digital telephone switching area, and as a research scientist in the Dependable Systems Group at the Department of Informatics Engineering (DEI), in Coimbra. His research work focused on dependability techniques for parallel systems, including error detection and dependability evaluation. He is currently finishing his Phd in DEI, in the area of parallel and distributed processing and also joining a Portuguese company - Critical Software Lda - to work on dependable technologies for critical systems. His current interests are on software reliability, fault tolerance, and parallel and distributed systems. He is a member of the IEEE Computer Society. João Carreira can be reached at [jcar@dei.uc.pt](mailto:jcar@dei.uc.pt)



**Henrique Madeira** is an assistant professor in the Department of Computer Engineering of the University of Coimbra, where he has been involved in the research on dependable computing since 1986. He received his BS degree in Electrotechnical Engineering and his Ph.D. in Computer Engineering from the Univer-

sity of Coimbra, Portugal, in 1983 and 1994 respectively. His main research interests focus on experimental evaluation of dependable computing systems by fault injection and behavior based error detection mechanisms. Currently, taking advantage of his experience on giving lectures on database systems, he is also doing research on the evaluation of the transaction mechanisms available in commercial database management systems. Henrique Madeira can be reached at [henrique@dei.uc.pt](mailto:henrique@dei.uc.pt)



**João Gabriel Silva** received a degree in Electrotechnical Engineering and a PhD in Informatics from the University of Coimbra, Portugal, in 1980 and 1988 respectively. Since 1988 he has been an Assistant Professor at the same University, in the Informatics Engineering Department, where he is the Head of the Scientific Committee. He also coordinates the research group on Dependable Systems, where he investigates experimental techniques for the validation of fault-tolerant systems by fault injection, behaviour based error detection, and fault-tolerant parallel and distributed programming. João Gabriel Silva can be reached at [jgabriel@dei.uc.pt](mailto:jgabriel@dei.uc.pt)