Xception: Software Fault Injection and Monitoring in Processor Functional Units[1]

João Carreira, Henrique Madeira, and João Gabriel Silva

Dep. Engenharia Informática
Pinhal de Marrocos - Polo II - Univ. Coimbra
3030 Coimbra - PORTUGAL
Email: {jcar, henrique, jgabriel}@mercurio.uc.pt

Abstract

This paper presents Xception, a software fault injection and monitoring environment. Xception uses the advanced debugging and performance monitoring features existing in most of the modern processors to inject more realistic faults by software, and to monitor the activation of the faults and their impact on the target system behaviour 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 it 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 operating system. Sets of faults can be defined by the user according to several criteria, including the emulation of faults in specific target processor functional units. Presently, Xception has been implemented on a parallel machine build around the PowerPC 601 processor running the PARIX operating system. Experiment results are presented showing the impact of faults on several parallel applications running on a commercial parallel system. It is shown that up to 73% of the faults, depending on the processor functional unit affected, can cause the application to produce wrong results. The results show that the impact of faults heavily depends on the application and the specific processor functional unit affected by the fault.

1. Introduction

The evaluation of the dependability properties of a computer system is a complex task and the growing complexity of both the hardware and software tend to make this evaluation even more difficult. The use of analytical modelling 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 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 [Arlat 90].

A popular approach consists of injecting physical faults into the target system hardware. Several methods have been used, such as pin-level fault injection [Arlat 90, Madeira 94b], heavy-ion radiation [Karlsson 94], and power supply disturbances [Miremadi 93]. 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 the fault injector tools are in general dedicated to a specific target system. Furthermore, the high complexity and the very high speed of the processors available today make the design of the special hardware required by the above techniques 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 [Madeira 94b]. 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.

[1] This work was supported by Esprit project 6731 - FTMPS "Fault Tolerant Massively Parallel Systems"

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. However, this technique usually involves an enormous development effort and can be very time consuming. Some recent examples of simulation-based fault injection tools can be found in [Choi 92, Jenn 94].

Software Implemented Fault Injection techniques (SWIFI), also known as fault emulation [Segall 88, Chillarege 89, Han 93, Young 92, Kanawati 92, Kao 94, Echtle 92] are being increasingly used as an alternative to the other methods. They basically consist in interrupting the application execution in some way (usually by inserting a software trap or executing the application in trace mode) and executing specific fault injection software code that emulates hardware faults by inserting errors in different parts of the system such as the processor registers, the memory, or the application code. The main advantages of software fault injection are the low complexity, low development effort, and low cost (no specific hardware is needed). In addition, software fault injection tools have increased portability, can be easily expanded (e.g., for new classes of faults), and do not have problems with physical and electrical interferences which are very common in physical fault injection tools.

There are limitation, however, to these approaches. They fail to inject faults in peripheral devices, specially address logic and its accuracy (ability to emulate device-level faults) as been questioned for a long time. However, some recent studies [Yount 93, Kanawati 93, Czeck93, Rimen94] with the aim of evaluating the representativity of the fault/error models used in SWIFI techniques achieved results that contradict such argues.

This paper proposes the use of the advanced debugging and performance monitoring features existing in modern processors to inject more realistic faults by software, and to monitor the activation of the faults and their impact on the target system behaviour in detail. A new software fault injection and monitoring environment, called Xception, is proposed based on this idea.

The huge complexity and integration scale of contemporary processors led the manufactures to introduce sophisticated debugging features and comprehensive performance monitoring hardware inside the processors. These new features are mainly used by sophisticated performance optimiser tools and are visible to system software.

By directly programming the debugging hardware inside the target processor, Xception can inject faults with minimum interference with the target application. Unlike previous software fault injectors, with Xception 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). The sophisticated exception triggers available in most of the modern processors allow the definition of many fault triggers (events that cause the injection of the fault), including fault triggers related with the manipulation of data. Furthermore, it is possible to monitor the activation of latent errors, such as the errors introduced in a specific memory cell, by programming the debugging hardware to cause an exception when the corrupted memory cell is addressed. All the exception code has to do in this case is to report that the corrupted memory cell has been used by the target program.

On the other hand, by assessing the performance monitoring hardware inside the processor, Xception can record detailed information on the target processor behaviour after the 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 has been accessed after the fault or if some program function has been executed.

Another important aspect is the fact that, since Xception operates at the exception handler level, and not through any service provided by the operating system, the injected faults can affect any process running on the target system including the operating system. Furthermore, it is also possible to inject faults in applications for which the source code is not available.

The target system is regarded by Xception as a system formed by the processor, memory and data/address buses. The target system can be a single computer or a node in a distributed/parallel system. In order to facilitate the task of defining sets of faults, the user can specify faults in the following internal

target units: Data Bus, Address Bus, Floating Point Unit, Integer Unit, Memory Management Unit, General Purpose Registers, Condition Code Register, Instruction Decoding, and Main Memory.

Presently, Xception has been implemented on a Parsytec parallel machine build around the PowerPC 601 processor and running the PARIX operating system (a Unix alike operating system for parallel machines). The Xception upgrade for the processor PowerPC 604 is being planed and a new version for Pentium based machines is being built.

This paper is organised as follows. Related research is discussed in section 2. Section 3 presents the processing debugging and performance monitoring features used by Xception. This section also includes a short survey of these new features in modern processors showing that the vast majority of the Xception features can be implemented in systems based on processors such as the MPC604, DEC Alpha, MIPS R4400-R10000, Pentium, HP Precision Architecture, and POWER2 Architecture. Section 4 presents the general architecture of Xception. Section 5 describes the fault model. Section 6 describes how faults are emulated by Xception at the low level. Section 7 demonstrates its capabilities and presents results obtained in preliminar experiments. Finally, Section 8 concludes the paper.

2. Related Research

Software fault injection has already been widely used in the past. One of the early approaches is FIAT [Segall 88], which enabled the corruption of task's memory image using special software. 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 in [Chillarege 89] and faults were also injected by modifying memory contents.

Another tool named DOCTOR [Han 93] 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.

An hybrid fault injection environment, called HYBRID, was proposed in [Young 92]. Faults are injected in memory via software, and extra equipment is used to trace fault activation and propagation in the target system. Thus, HYBRID solve the problem of knowing whether the faults have been activated or not, but it has the drawback of needing extra hardware.

In the FERRARI [Kanawati 92] approach, the UNIX *ptrace* function is used to corrupt the process memory image in run-time and inserting software trap instructions at the specific instruction address where a fault 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 [Kao 93] 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 [Kao 94] is an evolution of FINE that include distributed capabilities. It makes use of the program executable image modification to emulate memory faults, again by inserting software *traps* at specific memory locations in the text segment, but introducing a modified hardware clock interrupt handler to inject CPU and bus faults, including faults at the operating system level. The use of the hardware as fault trigger guarantees that these faults are always activated. Furthermore, in addition to hardware faults, DEFINE is also capable of injecting software faults, i.e. software design/implementation faults.

The injection of fault types specific to parallel and distributed systems have also been a major concern in several fault injectors such as DOCTOR [Han 93], DEFINE [Kao 94], EFA [Echtle 92] and CSFI [Carreira 95]. 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.

In spite of the large number of software fault injection works and the considerable advances proposed in the last years, existing tools still have several problems and limitations.

One of the most important problems comes from the fact that existing software fault injector tools have considerable impact on the target system behaviour, either because part or the whole of the code of the tool have to be executed in the target system (i.e., is part of the target workload) or because the target program may have to be executed in trace mode.

Previous research of different natures [Iyer 86, Czeck 90, Karlsson 94, Madeira 94a] have emphasised the impact of the workload on the performance of the fault handling mechanisms, which means that the software fault injection tools may interfere with the results. In Xception the impact in the target workload is minimal, as the exception trigger that injects the fault is programmed in the processor debugging hardware before starting the target application. Thus, the target application is not changed, no extra instructions are inserted, and the application is executed at normal speed.

Another limitation of existing software fault injection tools is related to 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 the instructions execution, and no fault triggers related with data manipulation can be defined. In [Kanawati 92] and [Kao 94] faults can also be injected by defining a temporal trigger. One advantage of this method is in the fact that the fault is always injected, as 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 due to the application execution time uncertainties. 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.

The target system monitoring is another problem of existing software fault injection tools. Monitoring is required either for detecting the activation of some faults or to collect relevant information on the fault impact. Only few proposals handle target monitoring, either by using extra instrumentation [Young 92], by using software monitors [Kao 93], or by inserting trap instructions in the adequate locations [Kao 94]. With the last method it is not possible to achieve detailed monitoring, while the other methods require extra hardware or cause great execution overhead. In Xception the use of dedicated hardware inside the processor greatly facilitates the monitoring of the target system in the presence of faults.

3. Processor debugging and performance monitoring features used by Xception

The performance monitoring and debugging features included in most of the recent processors consist mainly of performance counters and breakpoint registers. The former count user defined events such as load, store, or floating point instructions, while 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 not commonly used neither by normal applications nor by the operating system (they are mostly used by debugging and performance analysis tools), which facilitates their utilisation by Xception.

The present implementation of Xception is targeted for systems based on the PowerPC processor family, more specifically for the MPC601 [PowerPC 93]. Thus, the PowerPC will be used as a case study. At the end of this section several contemporary processors will be surveyed with the objective of

Exception Type	Causing conditions
Data access	This exception can have many specific causes: access to an invalid address, memory access not permitted, etc.. Particularly, it can occur if the address used in a load/store operation (selectable) matches the address in DABR (Data Access Breakpoint Register)
Run mode	Among other conditions, this exception is taken 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.
Floating point unavailable	Although the MPC601 have an internal FPU, it can be disabled by setting a bit in the MSR (Machine Status Register). This exception occurs when no higher priority exception exists, an attempt is made to execute an FP instruction and the FP available bit in MSR is disabled.
Decrementer	The MPC601 includes a register, named the Decrementer (DEC) 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

showing that most of the debugging and performance monitoring features required by Xception are also available in these processors.

Table 1 shows the list of exceptions types of MPC601 and MPC604 used by Xception. It is worth noting 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 address is accessed. Experiments performed by using the spatial method can 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. The Trace facility is used during the process of fault emulation to execute, when required, a very small set of instructions step-by-step. It is worth noting that Xception is not mainly based on this exception. Trace mode is used only when is strictly necessary and during very short periods, thus it does not slow down program execution.

Finally, the Floating Point unavailable exception is used to monitor the activation of FP unit faults. This is carried out by disabling floating point operations when the fault is injected in a FP register. Thus, the next FP instruction (the one that will be affected by the fault) will cause an exception, which indicate the fault activation.

The performance monitoring features of the PowerPC family are not fully explored yet 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 MPC604 will use the performance monitoring interrupt (PMI) and performance monitoring registers (PMR) of the MPC604. These performance monitoring facilities can count many events, such as correctly predicted branches, instruction or data cache misses, snoops, FPU instructions, processor cycles, etc.. The MPC604 saves the instruction address or the data address of the instruction that caused a PMI exception in additional dedicated registers and the information in all the processor registers and counters 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.

Several contemporary processor architectures have been checked to investigate the existence of the debugging and performance monitoring features required to implement the Xception on these processors.

The HP Precision architecture [HP 94] 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 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 [Mathisen 94].

The Alpha AXP architecture [DEC 94] 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 enable/disable bit for floating point instructions. All these features are similar to the ones existing in the MPC604.

The MIPS R4400-R10000 processor family [MIPS 94], 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 [Power2] includes a monitor containing several counters for counting instruction execution and data storage events 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 mechanisms used by Xception for fault injection and monitoring can be found in many other processors, which means that the concept of using the debugging and performance monitoring features for

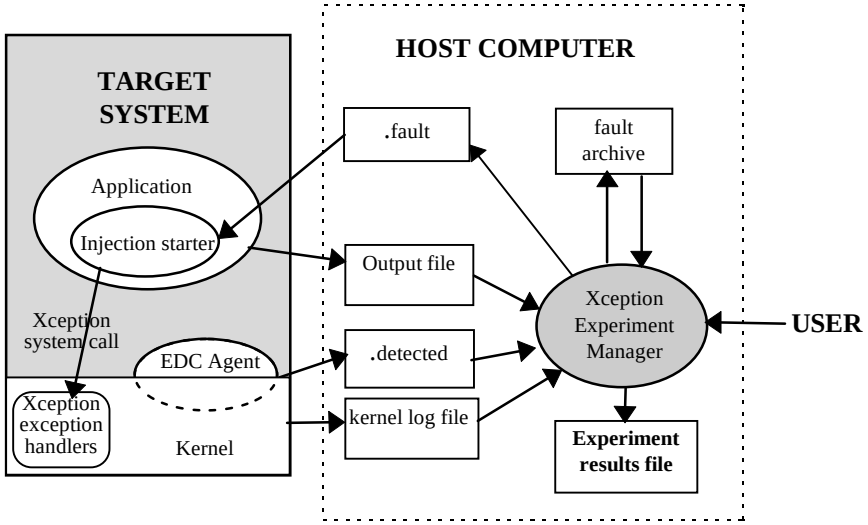

Figure 1. Xception structure

dependability evaluation by fault injection has a wide applicability.

4. Xception Architecture

4.1 Xception Modules

Xception consists of three parts: An injector module linked with the kernel of the target system, a library with functions to be called from the user application to start fault injection (or from a dedicated process if the application source code is not available), and the main module running on a host system which implements the user interface for fault definition, automatic fault injection and collection of results. The structure of Xception is shown in Figure 1.

Injection can be started from inside any process by inserting a call to an Xception function, *StartXception()* as the first statement of the code and linking it with the Xception function library. Moreover, it can be linked with C, C++ and Fortran applications. Most of the Xception modules are executed in the Xception host (presently a SUN Sparc) and the Xception code running on the target system is minimum

4.2 Xception Target System

The target system of Xception can be seen as composed by the processor, system buses and memory (the target system model is described in more detail in Section 5.1). Furthermore, it can be a single processor system or a node of a parallel machine. In fact, in the present implementation a parallel machine was used as the testbed for Xception: the Parsytec PowerXplorer. In this machine, each node contains an MPC601 for computation and a T805 transputer dedicated to the communication with the neighbour nodes. The Xception target node within the parallel machine is completely identified by an absolute number as is usual in the Parsytec architectures.

5. Fault Model

Xception was primarily designed to emulate hardware transient faults in functional units of the target processor. In fact, previous studies [Siewiorek 82, Lala 85] 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 cause 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.

5.1 Fault Location

From the fault definition and fault injection point of view the target system model presented in section 4.2 (processor, bus, and memory) can be described 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 model includes the following units:

- Instruction Execution Control Unit (IECU)
- Integer Unit (IU)
- Floating Point Unit (FPU)
- Memory Management Unit (MMU)
- Internal Data Bus (PDB)
- Internal Address Bus (PAB)
- General Purpose Registers (GPR)
- Condition Code Register (CCR)

For the sake of portability to other processors, Xception provides an interface for fault definition based in the above units (plus memory faults). In this way, the user can define a single set of faults and perform fault injection experiments in several microprocessors, thus having a common basis for comparing the results. Of course, the mechanisms used to emulate faults in the functional units are dependent on the particular features of each processor.

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 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.

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. This limitation is not very serious in many cases. It should be noted that Xception solve the problem of monitoring the activation of memory faults by programming an exception to be activated when the corrupted cell is addressed for the first time after the injection of the fault.

5.2 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 3, Xception uses five different types of exceptions to interrupt the processor in specific moments. The occurrence of such exceptions are used to trigger the faults. Fault triggers can thus be defined both in spatial and temporal terms. A temporally defined fault is 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 which can be specified in Xception are summarised bellow:

- Opcode fetch from a specified address;
- Operand load from a specified address;
- Operand store to a specified address;
- After a specified time since start-up;
- A combination of the above fault triggers.

5.3 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, bit flip, and bridging.

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. An example of fault definition parameters aimed at causing any two bit flips in the lower significant byte of a word structure (processor register or memory cell) is shown in Figure 2

Using this process it is possible to emulate many different fault types just by selecting the appropriate bits and operations.

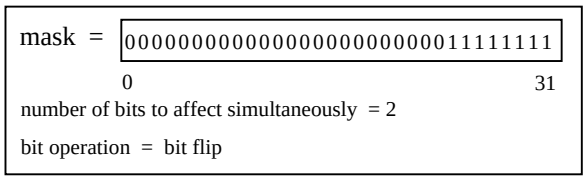

Figure 2. Sample fault parameters

6. Fault Emulation

The injection of a fault comprises several steps. The very first step (before the actual injection of the fault) is to program the fault trigger in the processor debugging logic. When the exception that injects the fault is activated, specific registers or memory content are corrupted according to the fault type and functional unit to be affected by the fault. Depending on the target functional unit and trigger type, a third step may be required to restore the original memory/register contents. This last step is necessary in several situations to guarantee that real transient faults are emulated.

For space reasons, in this paper the injection procedure is only described in detail for two types of faults, which are the FPU faults and the Address bus faults triggered by an operand fetch. These two cases have been selected because they are illustrative of the low level injection mechanisms used.

6.1 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). 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.

Figure 3 illustrates the mechanisms used to inject an error when an opcode is fetched. In this case, when the trigger instruction (Inst. A in Figure 3) 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 is executed). Xception then decides what should be the address to resume execution at (value to put in SRR0), depending on the type of erroneous instruction executed as illustrated in Figure 3. If the instruction was an absolute branch, then the branch is taken normally, if it was a relative branch, it is also taken, but the offset is relative to the original instruction (A). Finally, if it was not a branch instruction, execution continues in the instruction immediately after A. 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.

6.2 Faults in the Floating Point unit

The consequences of faults in the 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). Thus, after reaching the fault trigger, the FPU is disabled and normal execution is resumed. When the processor tries to execute a floating point instruction, a floating point unavailable exception occurs 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 identifyed, the instruction is executed in trace mode and the destination register is corrupted aftererwards. Faults in the FPU are injected using the same method no matter the trigger type used. Figure 4 illustrates this method.

7. Using Xception

7.1 Experiment Definition and Control

Xception generates any arbitrary number of faults

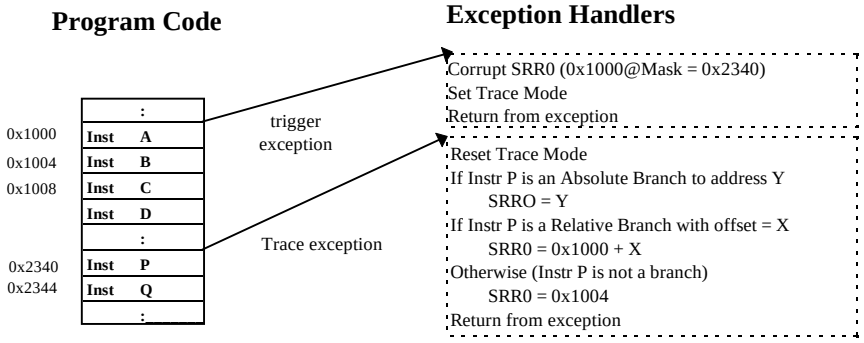

Figure 3. Transient address bus error when an opcode is fetched

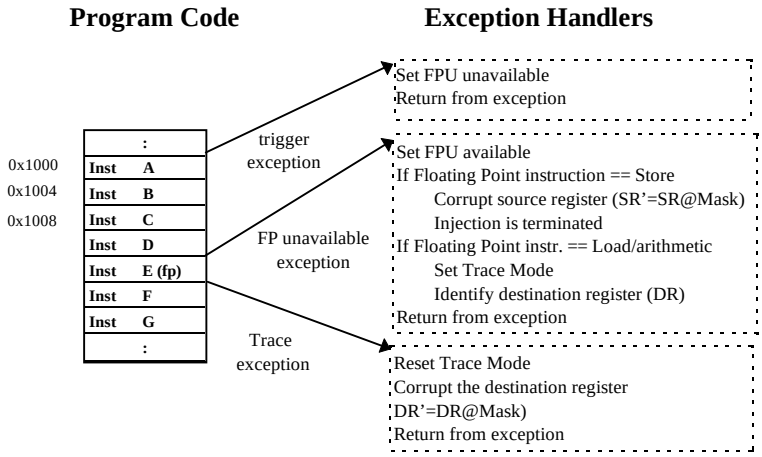

Figure 4. Transient error in the Floating Point Unit.

from a set of parameters given by the user. The user only has to provide the boundaries in which the fault parameters should vary. To help the user in this task Xception automatically executes the target application and collects some loader information, such as text limits, data limits, stack start address and size, and beginning of heap. Similar information is also provided for the kernel. This information enables the user to define spatial fault triggers much more precisely and get an higher percentage of activated faults.

As a result of the fault definition process a fault archive containing large amounts of fault descriptors is generated.

When an automatic fault injection experiment is initiated Xception runs the application once without fault injection, the so-called "gold" run, measures its execution time and backs up its output in the *.reference* file. The user is then asked to specify a Timeout upon which the application is terminated by Xception. This prevents the stall of the experiment caused by a possible hang-up of the application due to the injected fault.

During the course of the experiment the Xception main module running in the host retrieves fault descriptors from the fault archive and stores them in the *.fault* file, where they will be read by the Injection Starter. These parameters are passed by the Injection Starter to the kernel modules by means of a new system call created specifically for this purpose (see Figure 1).

At the kernel level, Xception programs the breakpoint and/or timer registers according to the fault trigger specified and lets the system activity proceed normally until the programmed exception occurs and the processor control is again acquired through the exception handlers. This procedure is repeated for every fault descriptor in the fault archive.

7.2 Data Collection and Analysis

After the injection of each fault, Xception collects several results and adds them to a file in a spreadsheet format. After the completion of the fault injection experiment the results can be analysed by consulting the file generated during the session. Using a spreadsheet the user can make the postprocessing analysis that best matches his interests.

The results collected for each injected fault are described in Table 2. 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 Injection Status messages sent by Xception modules at the kernel level are directed to a log file as shown in Figure 1, which is retrieved later by the Xception Experiment Manager Module

The output generated by the target application under fault injection can be compared with *.reference* using a general built-in compare function or alternatively, using an user-defined comparison program.

Xception guarantees that the Injection Status message is sent to the host and written in the log file immediately before the fault is injected, that is, before the exception handler returns control to the application with wrong data. This message contains timing information to enable the calculation of error detection latencies in the host.

7.3 Experimental Results

This Section presents the first experiments performed with Xception meant to demonstrate its features and show the detailed results that can be obtained. Experiments have been performed in a Parsytec Xplorer with four nodes, each one containing an MPC601, a T805 and 16 Mb of RAM, running the PARIX operating system. The goal of these experiments is to evaluate the impact of faults in parallel applications running in a commercial system with no particular fault handling mechanisms.

The parallel benchmarks selected have been developed using a high-level Linda parallel programming library build on top of PARIX, called ParLin [Silva 94], as well as in C by directly using PARIX message passing primitives. Linda is based on the abstraction of the Tuple Space, a logically shared memory that is visible to all processes in a parallel machine.

A short description of the benchmarks is given bellow:

1. π Calculation (Linda)

Computes an approximate value of **π** 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

Injection Status	Information about fault activation. If the fault is activated additional information is provided on the address and opcode of the affected instruction.
Execution Time	Execution time of the application measured from the host or the keyword "Timeout" if the application hanged-up and was terminated by the host.
Exit Code	Application's exit code (if exists) returned by the exit() call in the C Language .
Kernel Error messages	Error detected and reported by the PARIX kernel.
Application output Correctness	Result of byte-to-byte comparison of the program output with the *.reference* file. BIGGER: Bigger size SMALLER: Smaller size DIFFERENT: Same size but different contents EQUAL: Same size and contents NONE: No output file found EMPTY: Empty output file
Error Detection Results or binary events	Information on whether errors have been detected by Error Detection Mechanism (EDM) in the target or not. Space has been reserved for up to eight EDM.
Latency	Latency results associated to the EDM or binary events
Monitoring information	Information collected from the processor performance monitoring hardware (being implemented in the Xception version for the MPC604)

Table 2. Results collected by Xception

total sum. The final calculated value for π is stored in a file by the master.

2. SOR Successive Overrelaxation.

A fault tolerant version of a parallel algorithm to solve the Laplace equation over a grid [Chowdhury 93]. 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). 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.

Fault sets of 3000 "typical" faults have been injected in each application affecting several functional processor units. Two bits compensating faults affecting the all 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 are always activated.

Table 3 to Table 6 show the results provided by the statistical analysis of fault injection results for the π calculation. Each table presents specific information concerning the behaviour of the system under fault injection. The last table summarises the information by gathering the results of all tables and classifying the fault impact in three main classes: *Undetected*, *No Error*, *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 that the impact of faults is highly dependent on the affected functional unit. For instance, faults in the Address bus lead to 64,7% of system crashes (abortion through timeout). About 33,8% of these faults where detected by the kernel because they caused Program, Data, or Instruction accessed exceptions, and 25,74% were classified as *Undetected*. 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 30,34% of the cases. However, a larger percentage (43,8%) of *Undetected* (fatal fault: see Table 6) has been observed. This is due to the fact that some internal processor built-in error detection mechanisms that detect many address errors (e.g., instruction access exceptions: see Table 3) 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 73,17% and 72,78%. 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, 22,92% of the faults caused system crash or access exceptions and were detected by the system. On the other hand, in the

ERRORS DETECTED	Address Bus	Data Bus	FPU	IU	MMU
NO. OF EFFECTIVE FAULTS INJECTED	680	379	559	349	703
PROGRAM EXCEPTION	5.15%	5.54%	0.00%	3.15%	5.69%
DATA ACCESS EXCEPTION	13.53%	21.37%	0.00%	3.15%	14.65%
INSTRUCTION ACCESS EXCEPTION	15.15%	0.00%	0.00%	8.31%	16.64%
ALIGNEMENT EXCEPTION	0.00%	0.53%	0.00%	0.00%	0.00%
OTHER ERRORS	0.00%	0.52%	0.00%	1.15%	0.00%
NO ERROR MESSAGES	66.18%	72.03%	100.00%	84.24%	63.02%

Table 3. Errors detected by MPC601 built-in EDM's for the π application

PROGRAM EXIT CODES	Address Bus	Data Bus	FPU	IU	MMU
NO. OF EFFECTIVE FAULTS INJECTED	680	379	559	349	703
NORMAL	34.56%	69.66%	100.00%	77.94%	34.99%
ERROR IN PARIX SYSTEM CALLS	0.74%	0.00%	0.00%	0.00%	0.43%
ERROR IN PARLIN	0.00%	0.00%	0.00%	0.00%	0.00%
TIMEOUT	64.71%	30.34%	0.00%	22.06%	64.58%

Table 4. Exit codes for the π application

OUTPUT RESULTS	Address Bus	Data Bus	FPU	IU	MMU
NO. OF EFFECTIVE FAULTS INJECTED	680	379	559	349	703
NONE	65.44%	30.34%	0.00%	22.06%	65.01%
EMPTY	0.00%	0.00%	0.00%	0.00%	0.00%
EQUAL	8.68%	25.59%	26.83%	4.58%	9.25%
DIFFERENT	25.88%	44.06%	73.17%	73.35%	25.75%

Table 5. Output results of the π application

SUMMARY	Address Bus	Data Bus	FPU	IU	MMU
NO. OF EFFECTIVE FAULTS INJECTED	680	379	559	349	703
UNDETECTED	25.74%	43.80%	73.17%	72.78%	25.75%
NO ERROR	8.68%	25.33%	26.83%	4.30%	8.96%
ERROR	65.59%	30.87%	0.00%	22.92%	65.29%

UNDETECTED No errors have been detected and the application produced wrong results (fatal fault)
NO ERROR No errors have been detected but the application terminated normally and produced correct results (benign faults)
ERROR Some type of error have been detected (Timeout, Error Msg, bad exit code or MPC601 built-in mechanisms)

Table 6. Summary of results for the π application

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 where not directly connected with the calculation of π .

However, it should be noted that in order to obtain more correct figures about the effects of the FPU faults on the application behaviour these results should be weighted with information of the FPU usage in terms of frequency of FP instructions executed by the application. The information collected in the performance counters on the frequency of specific instructions (e.g. floating point) can be used to weight fault injection results. These features are planned for an upgrade version of Xception for the MPC604.

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* (fatal) cases in each processor are shown in Table 7.

The great 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. 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.

For space reasons, only a summary of the results obtained in the SOR and MATMULT application are presented in Table 8 and Table 9. 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

UNDETECTED	Address Bus	Data Bus	FPU	IU	MMU	Average
Processor 0	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%
Processor 1	14.26%	19.79%	28.62%	31.23%	10.67%	20.92%
Processor 2	8.38%	16.36%	30.95%	26.36%	10.53%	18.52%
Processor 3	3.09%	7.65%	13.60%	15.19%	4.55%	8.81%
Total	25.74%	43.80%	73.17%	72.78%	25.75%	48.25%

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

Table 7. Undetected cases by Processor and Functional Unit for the π application

SUMMARY	Address Bus	Data Bus	FPU	IU	MMU
UNDETECTED	3.41%	6.62%	0.00%	8.47%	4.89%
NO ERROR	19.80%	48.26%	99.46%	36.27%	19.71%
ERROR	76.79%	45.11%	0.54%	55.25%	75.41%

Table 8. Summary of results for the SOR benchmark

SUMMARY	Address Bus	Data Bus	FPU	IU	MMU	GPRs	CC Reg
UNDETECTED	0.00%	0.00%	-	0.00%	0.00%	0.00%	0.00%
NO ERROR	12.05%	24.36%	-	2.67%	10.61%	81.93%	96.95%
ERROR	87.95%	75.64%	-	97.33%	89.39%	18.07%	3.05%

Table 9. Summary of results for the MATMULT benchmark

ABFT Coverage	With ABFT	Without ABFT
UNDETECTED	0.00%	21.89%
NO ERROR	34.05%	34.05%
ERROR	65.95%	44.05%

Table 10. Coverage of Algorithm Based Fault-Tolerance in MATMULT

faults are highly dependent on the functional unit where they occur.

Table 10 shows the coverage of the ABFT (Algorithm Based Fault Tolerance) mechanism embedded in MATMULT [Luk 85]. 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 21,89% 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.

8. 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 behaviour in detail. The interference of Xception in the target application is very small as 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. Xception provides a complete set of fault triggers, including spatial and temporal fault triggers, and triggers related to the manipulation of data in memory. Furthermore, it is possible to monitor the activation of latent errors, such as the errors introduced in a specific memory cell, by programming an exception to be activated when the corrupted memory cell is addressed. Faults injected by Xception can affect any process running on the target system including the operating system, and it is possible to inject faults in applications in which the source code is not available. Sets of faults can be defined by the user according several criteria, including the emulation of faults in specific target functional units. Presently, Xception has been implemented on a Parsytec parallel machine build around the PowerPC 601 processor running the PARIX operating system. Several experiments were conducted with Xception in a system with very simple error detection mechanisms embedded in the kernel. Preliminary results obtained in these experiments show that the impact of faults depends heavily on the application and the specific functional unit where they occur. Particularly, faults at the FPU and 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

[Arlat 90] J.Arlat et al., “Fault injection for dependability validation: a methodology and some applications”, IEEE Trans. on Software Eng., Vol 16, No 2, Feb. 1990, pp. 166-182.

[Carreira 95] J. Carreira, Henrique Madeira, João Gabriel

Silva. “Assessing the Effects of Communication Faults on Parallel Applications” to be presented at IPDS’95, International Computer and Dependability Symposium, Erlangen, Germany, April 1995.

[Carreira95b] João Carreira, “Software Fault Injection in Parallel Systems”, MSc thesis, University of Coimbra, Portugal, July 1995.

[Chilarege 89] R. Chilarege ans N. Bowen, “Understanding Large Systems Failures - A fault Injection Experiment”, Proc. 19th Int. Symp. Fault-Tolerant Computing, Chicago, June, 1989, pp. 356-363.

[Choi 92] G. Choi and Ravi Iyer, “Focus: an experimental environment for fault sensitivity analysis”, IEEE Transactions on Computers, Vol. 41, No. 12, December 1992.

[Chowdhury 93] Roy-Chowdhury and P. Banerjee, “A Fault-Tolerant Algorithm for Iterative Solution of the Laplace Equation”, Proc. of the Int. Conference on Parallel Processing, 1993, pp. II-133 to III-140.

[Czeck 90] E. Czeck and D. Siewiorek, “Effects of transient gate-level faults on program behavior”, FTCS-20, Newcastle Upon Tyne, June 1990, p. 236-243.

[Czeck93] 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.

[DEC 94] “DECchip 21064 and DECchip 21064a Alpha AXP Microprocessors Hardware Reference Manual”, Order No.: EC-Q9ZUA-TE, Digital Equipment Corp., June 1994.

[Echtle 92] K. Echtle, M. Leu. “The EFA Fault Injector for Fault-Tolerant Distributed System Testing”, in Workshop on Fault-Tolerant Parallel and Dist. Systems, pp 28-35, 1992.

[Han 93] S. Han, H.Rosenberg, K.Shin, “DOCTOR: an Integrated Software Fault Injection Environment”, Technical Report-University of Michigan, 1993.

[HP 94] “PA-RISC 1.1 Achitecture and Instruction Set Reference Manual”, HP Part Number: 09/40-90039, Third Edition, February 1994.

[Iyer 86] R. Iyer and D. Rossetti, “A measurement-based model for workload dependance of CPU errors”, IEEE Trans. on Computers, vol. C-35, pp. 511-519, June 1986.

[Jenn 94] 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.

[Kanawati 92] G. Kanawati, N. Kanawati, and J. Abraham, “FERRARI: A Tool for the Validation of System Dependability Properties”, FTCS-22, Digest of papers, IEEE 1992, pp. 336-344.

[Kanawati93] G.Kanawati, N.Kanawati, J.Abraham, “EMAX: An automatic Extractor of High-Level Error Models”, AIAA Computing Aerospace Conference, San Diego, CA, October 1993, pp. 1297-1306.

[Kao 93] Wei-lun Kao, R. K.Iyer, D. Tang, “FINE: A Fault Injection and Monitoring Environment for Tracing the UNIX System behaviour under Faults”, IEEE Transactions on Software Engineering, Vol 19. No. 11, November 1993

[Kao 94] Wei-lun Kao, R. K. Iyer, “DEFINE: A Distributed Fault Injection and Monitoring Environment”, Workshop on Fault-Tolerant Parallel and Distributed Systems, June, 1994.

[Karlsson 94] J. Karlsson, P. Lidén, P. Dahlgren, R. Johansson, and U. Gunneflo, “Using Heavy-ion Radiation to Validade Fault-Handling Mechanisms”, in IEEE Micro, Vol. 14, No. 1, pp. 8-32, 1994.

[Karlsson 94] J. Karlsson, P. Lidén, P. Dahlgren, R. Johansson, and U. Gunneflo, “Using Heavy-ion Radiation to Validade Fault-Handling Mechanisms”, in IEEE Micro, Vol. 14, No. 1, pp. 8-32, 1994.

[Lala 85] P. K. Lala, “Fault Tolerant and Faut Testable Hardware Design”, Prentice Hall International, NYk, 1985.

[Luk 85] F. T. Luk, “Algorithm-based fault tolerance for parallel matrix solver”, Proc. SPIE Real-Time Signal Processing VIII, vol. 564, 1985, pp. 49-53.

[Madeira 94a] H. Madeira and J.G.Silva, “Experimental Evaluation of the Fail-silent behaviour in Computers without Error Masking”, FTCS-24 June 1994.

[Madeira 94b] H. Madeira, M.Rela, F.Moreira, J.Silva. “RIFLE: A General Purpose Pin-Level Fault Injector”, Proc. First European Dependable Computing Conference, pp 199-216, Berlin, Germany, October 1994.

[Mathisen 94] Terje Mathisen, “Pentium Secrets”, BYTE, pp. 191-192, July, 1994.

[MIPS 94] Joe Heinrich, “MIPS R4400 Microprocessor User’s Manual”, Mips Techologies, 1994

[Miremadi 92] G. Miremadi, J. Karlsson, U Gunneflo, and J. Torin, “Two Software Techniques for On-line Error Detection”, proc. of 22th Fault-Tolerant Computing Symposium, FTCS-22, 1992, p. 328-335.

[Power2] 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

[PowerPC 93] “PowerPC601 RISC Microprocessor User’s Manual“ Motorola, July 1993

[Rimen94] M.Rimen, J.Ohlsson, J.Torin, “ On Microprocessor Error Behaviour Modelling”, Proc. of the 24th Int. Symp. on Fault-Tolerant Computing (ftcs-24), pp. 76-85, Austin, TX, USA, 1994.

[Segall 88] Z.Segall, T.Lin, “FIAT: Fault Injection Based Automated Testing Environment”. In Proc.. 18th Int. Symp. Fault - Tolerant Computing., June 1988, pp 102-107.

[Siewiorek 82] D. P. Siewiorek and Robert S. Swarz, The Theory and Practice of Reliable Design, Digital Press, Educational Services, Digital Equipment Corporation, 1982, Bedford, Massachusetts.

[Silva 94] 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.

[Young 92] L. T. Young, R. K. Iyer, K. K. Goswami, and C. Alonso, “A Hybrid Monitor Assisted Fault Injection Environment” 3rd IFIP Working Conference on Dependable Computing for Critical Applications, Sicily, Italy, September 1992.

[Yount93] Charles R. Yount, “The Automatic Generation of Instruction Level Error manifestations of Hardware Faults: A New Fault-Injection Model”, Phd thesis, Carnegie Melon University, May, 1993.