CHALMERS

Fault Injection Optimization through Assembly-Level Pre-Injection Analysis

RAUL BARBOSA

Master's Thesis

ERASMUS Programme

CHALMERS UNIVERSITY OF TECHNOLOGY
Department of Computer Engineering
Göteborg 2004

ABSTRACT

This thesis describes the design and implementation of a pre-injection analysis technique aimed at reducing the cost of validation of fault-tolerant computer systems through fault injection.

Fault injection has become an important step in the process of validating and evaluating the reliability of computer systems. Its purpose is to validate fault-tolerance mechanisms and measure the dependability of these systems. GOOFI is a fault injection tool developed at the Department of Computer Engineering, Chalmers University of Technology. The pre-injection analysis technique was applied to this fault injection tool.

Using assembly-level knowledge of the computer system – Motorola's MPC565 (an implementation of the PowerPC architecture) – the pre-injection analysis produces fault-sets with a higher proportion of effective faults than random fault selection. The experimental results obtained using optimized and randomly selected fault-sets are compared. The programs executed during the fault injection experiments were an implementation of the quicksort algorithm and a complex jet engine controller. Single bit-flip faults were injected into the user registers of the MPC565 via its Nexus port (a standard debug interface for embedded applications). For the jet-engine controller workload, single-bit flips were also injected into the data, heap and stack segments of the main memory.

The pre-injection analysis yielded an increase of one order of magnitude in the effectiveness of faults, a reduction of the fault-set of two orders of magnitude in the case of registers and four to five orders of magnitude in the case of memory.

ACKNOWLEDGEMENTS

I would like to thank Professor Johan Karlsson, for making it possible for me to work in the FORCE research group and for his valuable advice while writing the thesis. Special thanks go to Assistant Professor Peter Folkesson and PhD student Jonny Vinter for the dedication, support and ideas offered throughout the project. I would also like to thank Professor Mário Rela from the University of Coimbra – my home university – for introducing me to the area of dependable computing.

INDEX

LIST OF ABBREVIATIONS

CAN	Controller Area Network
COTS	Commercial Off-The-Shelf
CR	Condition Register
ELF	Executable and Linkable Format
FI	Fault Injection
FPR	Floating Point Register
GOOFI	Generic Object Oriented Fault Injection Tool
GPR	General Purpose Register
LR	Link Register
PC	Program Counter
RCPU	RISC Central Processing Unit
RISC	Restricted Instruction Set Computer
SEU	Single Event Upset
SWIFI	Software-Implemented Fault Injection
UISA	User Instruction Set Architecture

1. INTRODUCTION

1.1. Context

Computer systems are increasingly being used in safety-critical applications such as aerospace or vehicular systems. To achieve the high dependability required by these applications, systems are designed with fault tolerance mechanisms in order to deliver correct service even in the presence of faults. Faults may, for instance, occur when processors are disturbed by high energy particles such as neutrons or heavy-ions. Such particles may sometimes interfere with the processor and cause a single event upset (SEU) – an error that typically changes the state of a single bit in the system.

Safety-critical systems must be thoroughly tested in order to validate the correctness and efficiency of their fault tolerance features. Fault injection has become an effective technique for the experimental dependability validation of computer systems. The objective of fault injection is to test fault tolerance mechanisms and measure system dependability by introducing artificial faults and errors.

A problem found during fault injection campaigns is that not all faults fulfil the purpose of disturbing the system [1]. Often 80-90% of randomly injected faults are not activated [1, 2]. A fault placed in a register just before the register is written or faults that are injected into unused memory locations are examples of faults with no possibility of activation. In most tools the location and the time for fault injection are chosen randomly from the complete fault-set, which is typically extremely large. The statistical implication of this is that the cost of obtaining appropriate confidence levels of the dependability measures becomes unnecessarily high.

To deal with this problem and reduce the cost of validation through fault injection, two main classes of analysis techniques have been proposed: pre-injection and post-injection analysis [3]. Post-injection analysis aims at predicting the outcome of fault injection experiments using the results from other experiments. Pre-injection analysis, in its turn, uses knowledge of program flow and resource usage to choose the location and time where faults should be placed, before any experiment is performed.

This thesis presents a pre-injection analysis technique that is applicable to injection of transient bit-flips into CPU user registers and memory locations. The bit-flip fault model often used in fault injection experiments to emulate the effects of single event upsets and other transient disturbances.

1.2. Purpose

The pre-injection analysis technique uses program execution information to (i) eliminate faults that have no possibility of activation and (ii) find equivalence classes among faults and inject only one representative of these. This is achieved by applying the following rule: faults should only be placed in the resources read by each instruction. A bit-flip in any resource (register or memory) will only manifest itself once this resource is read to perform an operation. Delaying the injection of the fault until the moment just before the targeted resource is read accomplishes the two objectives stated above.

The technique was implemented in the GOOFI (Generic Object-Oriented Fault Injection) [4] tool, for Nexus-based [5] fault injection. Nexus is a standard on-chip debug (OCD) interface for embedded applications suitable for fault injection. The effectiveness of the technique was assessed by comparing fault injection results with results obtained by random fault injection on the same platform. The target system is based on the Motorola MPC565 [6] – a processor from the PowerPC architecture aimed at the automotive industry and other control-intensive applications. By applying assembly-level knowledge of this architecture it is possible to identify which resources are read by each executed instruction. This information, along with the time of the fault injections, is used to define an optimized fault-set, which is stored in a database. The fault injection experiments are then conducted by selecting faults from the optimized fault-set.

1.3. Thesis Organization

The next chapter describes previous research and the state-of-the-art in pre-injection analysis. Chapter 3 gives an overview of the GOOFI-tool, the MPC565 microcontroller and the workloads executed by the target system. Chapters 4 and 5 discuss the optimization method and its implementation in the GOOFI tool. Experimental results that demonstrate the effectiveness of the optimization approach are then presented in Chapter 6, while the conclusions are presented in Chapter 7. Possible lines of future research are presented in Chapter 8.

2. STATE OF THE ART

2.1. Introduction

During the knowledge acquisition phase of this thesis a number of previous approaches to the problem of fault injection optimization were identified and characterized. Most likely all researchers in the area of fault injection have, at some moment, realized that the number of experiments required to obtain statistically valid results in some case is very large.

The resources available in computers are, usually, greater than the needs of the applications executed. This fact motivates a first optimization by injecting faults only in used resources. P. Yuste et al. [2] make, in their experiments, special care to avoid placing faults in empty (i.e. not used) memory regions. They obtained fault activation ratios of 12% and pointed out that injecting faults using a random distribution is not a time-effective approach.

Avoiding unused memory regions might be done manually by analyzing the memory map of the application and choosing the segments (stack, heap, etc.) as valid locations for fault injection. This approach is quite simple but does not consider the way resources are used along the time dimension. The following graphic of the memory reads by a jet engine controller[1] during one control loop of execution provides a picture of this fact:

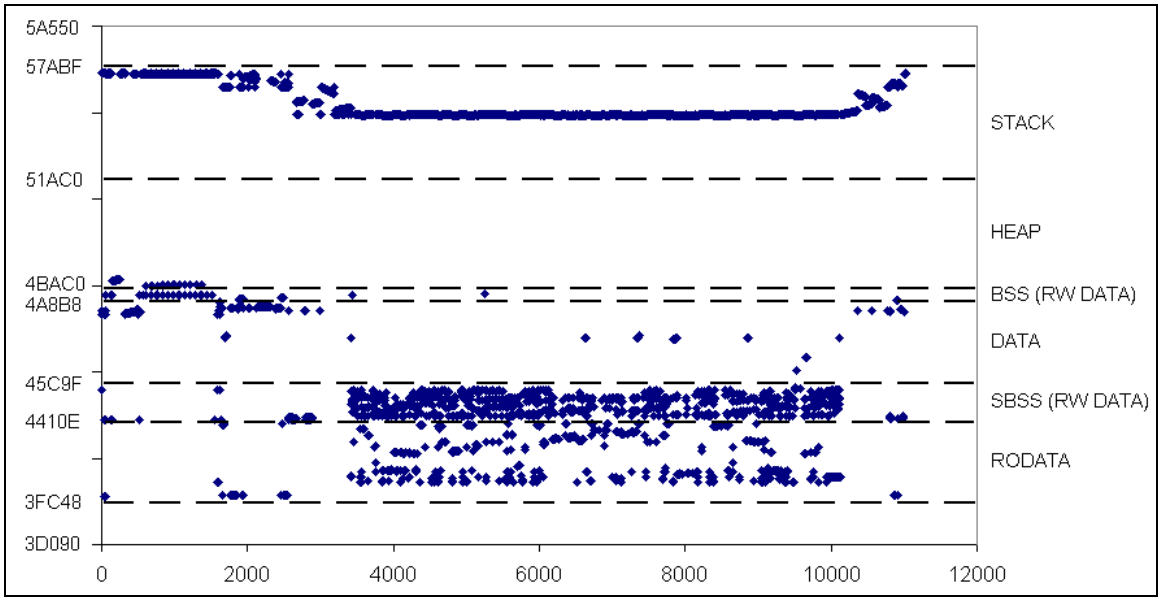


Figure 1 - Memory read accesses for the jet engine controller (1 control loop – 11028 memory accesses)

[1] The jet engine controller is one of the applications targeted in this thesis and is described in Section 3.4.

Reducing the fault-set to the memory segments presented in the graph increases the probability of fault activation. Further optimization would be achieved by considering only the memory locations within these segments that are actually used. Not only are the available resources greater than the needs of the application but also the segments are over-dimensioned. Reducing the granularity to the level of individual memory locations would yield better results, although it would be even better to use techniques that consider the dynamic usage of the resources.

During the past years two main classes of analysis techniques for reducing the cost of validating dependable systems by fault injection have been studied – pre-injection analysis and post-injection analysis. Post-injection analysis [3] aims at predicting the outcome of fault injection experiments using the results from other experiments. Pre-injection analysis, in its turn, uses knowledge of program flow and resource usage to choose the location and time where faults should be placed, before any experiment is performed.

The following sections of this chapter present some techniques focusing on pre-injection analysis developed in the past to reduce the necessary number of fault injection experiments.

2.2. Failure Acceleration

Studies conducted in the past have shown that error manifestation (rate and effects) is affected by workload [7, 8, 9]. In [10] the concept of *failure acceleration* was introduced by R. Chillarege and N. Bowen. The failure process is defined as accelerated when the fault model in not altered and:

i. The fault latency is decreased;

ii. The error latency is decreased;

iii. The probability of a fault causing a failure is increased.

Fault latency is defined as the time elapsed between the fault occurrence and its first manifestation as an error in the system. Error latency is defined as the time elapsed between the occurrence of an error and the time when the system fails to deliver correct service. The definition of these and other related terms can be found in [11].

R. Chillarege and N. Bowen achieve fault acceleration by injecting faults only on pages that are currently in use and by using a workload pushing towards the limits in CPU and I/O capacity. Their experiment had the intent to study the effects of software faults, particularly, the overlay. An overlay occurs when a program writes into an area of real storage due to an incorrect destination operand. The fault model was implemented by choosing a random page in use and setting its contents to hexadecimal ‘FF’.

The three premises that define failure acceleration are fulfilled because (i) the error condition is immediately effective thereby making the fault latency zero, (ii) the error latency is minimized by using a heavy workload (the effects of workload on error latency were studied in [9]) and because real storage is the most used virtual storage in the system, and (iii) the probability of failure is increased because the hexadecimal value 'FF' is, in the targeted architecture, an invalid opcode, an invalid branch location and generally an invalid address.

The particular case of the overlay fault model allows this approach to achieve optimization. The resource usage is considered along the time dimension but the granularity chosen for the resources (memory) is a complete page. For the single bit-flip model the page-wide granularity would have to be changed to track memory usage in greater detail and, therefore, place the faults with more precision.

2.3. Operational-Profile-Based Fault Injection

J. Güthoff and V. Sieh presented in [12] the operational-profile-based fault injection. They state that the number of fault injections into a specific system component depends on the relevance of that component for application processing. In the case of their fault model – single bit-flips affecting the processor's registers (General Purpose Registers) – the extent of fault injections into a specific register is proportional to its utilization. Register utilization is defined as the measure of the probability that an injected fault manifests itself as an error. Figure 2 (similar to the one presented in [12]) is an example of what might be obtained from experimental results:

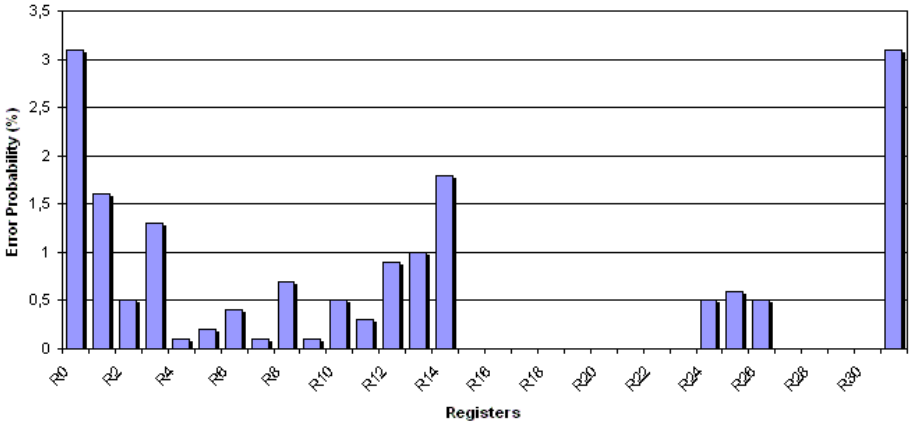


Figure 2: Estimated error probabilities per register

Considering the estimations presented in this figure it is more promising to inject faults in register R1, R14 and R31 (R0 is constant zero).

The locations for fault injection are correlated to resource utilization. Additionally, the times for fault injection are selected based on the data life-cycles. A data life-cycle starts

with the initialization of a register (write access) and ends with the last read access before the next write access. Under the single bit-flip fault model, faults need to be injected only within the data life-cycles, just before each read access.

The life-cycle approach is similar to the method presented in this thesis. Their study was conducted using a VHDL simulation of the Motorola MC88100 RISC Processor and targeting only the General Purpose Registers. After performing the optimizations the final results yielded 78% of faults without effect, 12% of faults were detected and 10% produced erroneous results. The fault-set was reduced from 21600 temporal locations to only 220.

2.4. Fault-List Collapsing

A. Benso et al. presented in [13] a set of rules with the purpose of collapsing fault-lists. The rules reduce the fault-list without affecting the accuracy of the results of fault injection campaigns by avoiding the injection of faults for which the behavior can be foreseen.

Collapsing faults in the code is achieved by eliminating faults which change the opcode of an instruction into an invalid opcode (which will trigger a processor exception) and by eliminating faults injected into the code of an instruction after the last execution of that instruction.

Collapsing faults in the data applies the concept of data life-cycles presented in the previous section. Faults injected outside data life-cycles are useless and thus removed from the fault-list. Also, faults injected in adjacent life instants on the same bit of the same variable belong to the same *equivalence class* and only one needs to be injected as a representative. Two specific rules are presented for the Instruction Register (IR) and for the Program Counter (PC). Faults injected into the IR that turn it into an illegal opcode trigger a processor exception and faults which change the PC value to point to a memory location where an illegal opcode is stored also triggers an exception in the processor.

One unexplored possibility would be to use the life-cycle concept also for faults in the code, by injecting faults only before the execution of each instruction.

Their fault injection tool adopts a simulation-based approach and exploits VHDL as the hardware description language to describe the system under evaluation – a Motorola 68040 chip.

The results computed on three workloads (*bubble sort*, *parser* and *matrix*) showed that the proposed rules can foresee the exact result of the fault injection experiment in a range between 36.6% and 54.5%.

2.5. Path-Based Injection

In [14] a technique named *path-based injection* was introduced. With this technique a fault is injected into a resource that will be used by the test program given a particular input. An *input* is the set of data that the test program processes and may include command-line arguments, contents of files, environment variables, etc.

The first requirement is, therefore, the manual derivation of an input set based upon knowledge of the test program, including documentation and knowledge of the program's high-level language code. The set of inputs should be thorough enough to allow the execution of all basic blocks of the test program at least once.

Having the input set it is necessary to discover the path of execution associated with each input. Using tracing utilities the execution paths are described in terms of a list of basic blocks that are executed due to a given input set.

The choice of which faults should be injected for each path is the final step. To simplify this step, they only considered control-flow faults that directly affect the execution of branches and faults were only injected into CPU registers. Thus, the faults that can accomplish activation in each path occur when the CPU registers used as operands for conditional branches are corrupted. This analysis is done at assembly-level since there is no dependence on any high-level language, compiler optimizations are not relevant and the access to physical registers is allowed without the need to map variables to physical registers.

Once this pre-analysis procedure is completed, path-based injection can be performed. T. Tsai et al. [14] used software-implemented fault injection (SWIFI) to demonstrate the effectiveness of path-based fault injection on a Tandem Integrity S2 computer. The fault injection campaigns using several workloads yielded error detection ratios between 12.4% and 31.6%. Undetected wrong output occurred between 26.2% and 31.8% of the experiments.

Their analysis encompassed only a subset of the instruction-set and resources (control-flow instructions and general-purpose registers) due to the complexity of the instruction-set architecture.

2.6. Workload Dependent Fault Collapsing

Working in a different line of research – fault injection for the test of fault-tolerant circuits – a set of techniques for speeding up campaigns is described in [15]. One of these techniques is *workload dependent fault collapsing*.

Intended for simulation-based fault injection in VHDL models, the proposed algorithm first executes a fault-free experiment and stores the golden run. The faults are then

injected sequentially by loading the state of the system just before fault injection time, placing the bit-flip (the most commonly used model of a SEU) and categorizing its effects. This step is achieved by using checkpoints and improves the fault injection procedure by accelerating the fault occurrence but not the failure process itself.

The effectiveness of the fault injection process is then improved by applying the workload dependent fault collapsing technique. During the golden run all read and write operations on memory elements are tracked with bit granularity. Having this log of read and write operations on each bit of each signal, all possible bit-flips are then collapsed using the following rules:

1. All bit-flips between an operation (either read or write) and a write operation are useless, and thus marked as silent;
2. All bit-flips between an operation (either read or write) and the subsequent read operation are equivalent.

Applying these rules resulted in the elimination of 74% of the faults from the complete fault-set. The checkpoint mechanism also contributes to the overall performance speed-up by reducing the length of each experiment by almost 20%.

The workload dependent fault collapsing technique is therefore suitable for the optimization of fault injection campaigns on fault-tolerant circuits. At the hardware level the technique requires the traceability of all read and write operations on each signal. The rules are similar to the ones obtained using the concept of data life-cycles at the signal-level.

3. SYSTEM DESCRIPTION

3.1. GOOFI

GOOFI (Generic Object-Oriented Fault Injection) [4] is a fault injection tool developed at the Department of Computer Engineering, Chalmers University of Technology. The goal of this tool is to perform fault injection targeting different systems. Its current plug-in based engine allows the rapid integration of new fault injection techniques and features.

The optimization technique described in this thesis was implemented in GOOFI. The fault injection campaigns performed in order to validate and evaluate the technique were, therefore, achieved with this fault injection tool. The technique is described in Chapter 4 and its implementation in GOOFI (named OFFSET) is discussed in Chapter 5.

3.2. Nexus-based Fault Injection

Recently a new fault injection technique has been developed in GOOFI which uses the Nexus [5] port to inject faults on Motorola's MPC565. Nexus is an attempt to create a standard on-chip debug (OCD) interface for embedded applications. This standard is suitable to be used for fault injection [2] since it provides read/write access to the processor's resources and code execution trace capture.

The pre-injection analysis technique was implemented to enhance the existing Nexus fault injection plug-in. The target platform for the implementation of the optimization is, therefore, the MPC565 microcontroller, described in Section 3.3.

3.3. MPC565 Microcontroller

3.3.1. System Description

The MPC565 is a microprocessor developed by Motorola that implements the PowerPC instruction standard architecture. It is aimed at the high performance automotive market as well as other control-intensive applications.

The complete computer system was based on the phyCORE-MPC565 [16] development board. It includes a 32-bit Motorola MPC565 processor, which offers a class 3 Nexus debug port. The Nexus standard describes four implementation classes. A class 3 Nexus debug port offers real-time data tracing and real-time access to memory and I/O in addition to the features of classes 1 and 2 (debugging breakpoints, program trace, etc.).

To establish a connection through this port the iSYSTEM iC3000 Active Emulator [17, 18] was used to simulate the Nexus working environment. The iC3000 emulator was, in its turn, controlled by GOOFI through winIDEA – an integrated development environment offered by iSYSTEM. GOOFI and winIDEA are running on a host PC. Figure 3 provides a schematic view of the experimental setup.

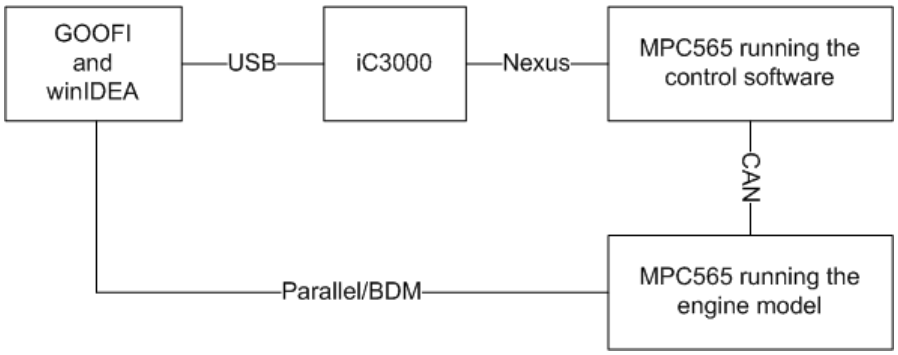


Figure 3 - Experimental setup for the control software

3.3.2. Registers

This section presents basic information about the on-chip registers available in the MPC565.

PowerPC processors have two levels of privilege: supervisor mode of operation (typically used by the operating system) and user mode of operation (used by the application software). The programming model of this architecture offers 32 general purpose registers (GPRs), 32 floating point registers (FPRs), several special-purpose registers

(SPRs) and several miscellaneous registers. Each PowerPC processor also has its own unique set of implementation-specific registers.

The MPC565 is a 32-bit implementation of the PowerPC architecture. In the MPC565, the Time Base Register (TBR) and FPRs are 64 bits. All other registers are 32 bits. The following paragraphs summarize the PowerPC registers that are available in the MPC565. The information is based on Motorola's RCPU Reference Manual [19].

General-Purpose Registers (GPRs) – The processor provides 32 user-level, general-purpose registers (GPRs). The GPRs serve as the data source or destination for all integer instructions and provide addresses for all memory-access instructions.

Floating-Point Registers (FPRs) – The processor also provides 32 user-level 64-bit floating-point registers. The FPRs serve as the data source or destination for floating-point instructions. The floating-point register file can only be accessed by the FPU.

Condition Register (CR) – The CR is a 32-bit user-level register that consists of eight four-bit fields that reflect the results of certain operations, such as move, integer and floating-point compare, arithmetic, and logical instructions, and provide a mechanism for testing and branching.

Floating-Point Status and Control Register (FPSCR) – The floating-point status and control register (FPSCR) is a user-level register that contains all exception signal bits, exception summary bits, exception enable bits, and rounding control bits needed for compliance with the IEEE 754 standard.

Machine State Register (MSR) – The machine state register (MSR) is a supervisor-level register that defines the state of the processor.

Special-Purpose Registers (SPRs) – The processor provides several special-purpose registers that serve a variety of functions, such as providing controls, indicating status, configuring the processor, and performing special operations. Some SPRs are accessed implicitly as part of executing certain instructions. All SPRs can be accessed by using the move to/from special-purpose register instructions, mtspr and mfspr.

User-Level SPRs – Three SPRs are accessible by user-level software (LR, CTR and XER). The Link Register (used to provide the branch target address and to hold the return address after branch and link instructions), the Count Register (decremented and tested automatically as a result of branch-and-count instructions) and the Integer Exception Register (contains the integer carry and overflow bits and two fields for the load string and compare byte indexed (lscbx) instruction).

Supervisor-Level SPRs – The processor contains SPRs that can be accessed only by supervisor-level software. Appendix B contains a list of these.

Development Support Registers – The processor provides a set of implementation-specific registers for development support. These are summarized in Appendix C.

The registers targeted in the fault injection campaigns described in Section 6 are the General-Purpose Registers, the Floating-Point Registers, the Condition Register and the three User-Level SPRs (Link Register, Count Register and Integer Exception Register).

3.3.3. Exceptions

The complete list of exception mechanisms available in the MPC565 is presented in Appendix D. This section presents a more detailed description of the exceptions that were most often triggered by the fault injection experiments. The description is based on the MPC565 User's Manual [6].

Checkstop (CHSTP) – The processor was configured to enter the *checkstop* state instead of taking the Machine Check Exception (MCE) itself when the MCE occurs. CHSTP does not represent an actual exception, but rather a state of the processor. The processor may also be configured to take the MCE handling routine or enter debug mode. The MCE, which, in this case, leads to the checkstop state, is caused by the following conditions:

- The accessed address does not exist.
- A data error was detected.
- A storage protection violation was detected by chip-select logic.

Alignment Exception (ALE) – The alignment exception is triggered under the following conditions:

- The operand of a floating-point load or store instruction is not word-aligned.
- The operand of a load or store multiple instruction is not word-aligned.
- The operand of lwarx or stwcx. is not word-aligned.
- The operand of a load or store instruction is not naturally aligned.
- The processor attempts to execute a multiple or string instruction.

Floating-Point Assist Exception (FPASE) – This exception occurs in the following cases:

- A floating-point enabled exception condition is detected, the corresponding floating-point enable bit in the FPSCR is set (exception enabled) and MSR[FE0] | MSR[FE1] = 1.
- A tiny result is detected and the floating point underflow exception is disabled.
- In some cases when at least one of the source operands is denormalized.

Software Emulation Exception (SEE) – An implementation-dependent software emulation exception occurs in the following cases:

- An attempt is made to execute an instruction that is not implemented.
- An attempt is made to execute an mtspr or mfspr instruction that specifies an unimplemented SPR.

External Breakpoint Exception (EBRK) – This exception occurs when an external breakpoint is asserted.

3.4. Workloads

Fault injection campaigns were conducted to evaluate error detection coverage for two different workloads: a quicksort implementation and a jet engine controller. Different campaigns targeting registers and memory, performing both random and optimized fault injection, were carried out.

The outcome of each experiment is analyzed in order perform the following classification of errors:

- **Detected Error** – All effective errors that are signaled by hardware error detection mechanisms included in the processor.
- **Wrong Output** – All effective errors that are not detected by the processor but lead to the production of wrong results.
- **Non-Effective Error** – Errors that do not affect the system execution during the chosen experiment time frame and no difference is observed between the fault injection experiment and reference values.

3.4.1. QuickSort

The quicksort workload is a recursive implementation of the well-known sorting algorithm. It sorts an array containing seven double-precision floats. Quicksort was the first workload to be tested due to its small size and simplicity.

The reference run execution takes less than two minutes and each fault injection experiment takes less than half a minute to perform.

3.4.2. Jet Engine Controller

This workload is a control application that executes in loops in order to control a jet engine. At the end of each loop the controller has to produce results and exchange information with the engine. It is significantly more complex than the quicksort program, allowing the fault-set optimization technique to be evaluated using a real-world application.

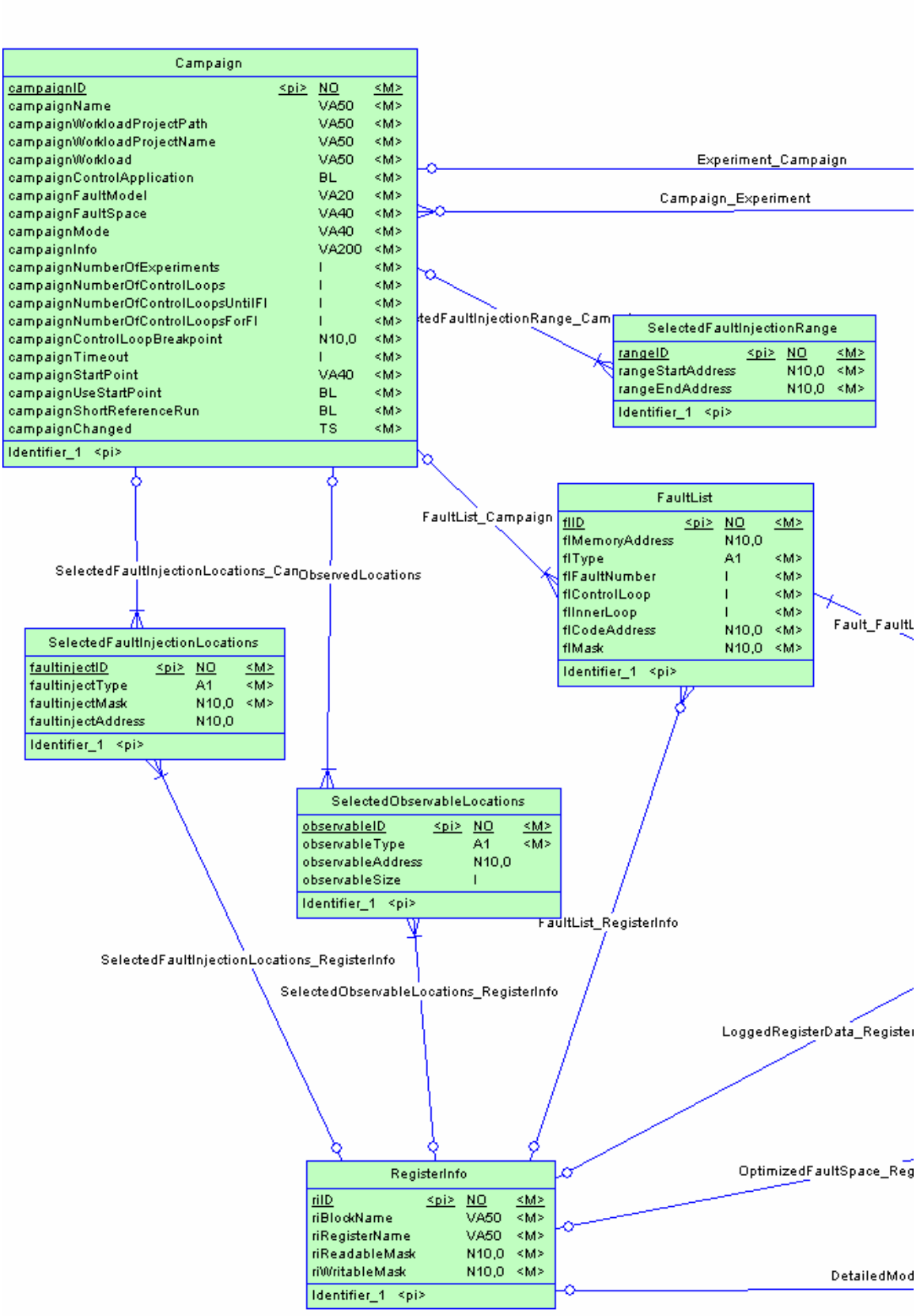
Two development boards[1] were used to create a distributed system. One of these was used to run the control software and the other to execute the model of the engine. The two boards communicate through a CAN bus. Figure 3 provides an overview of this experimental setup.

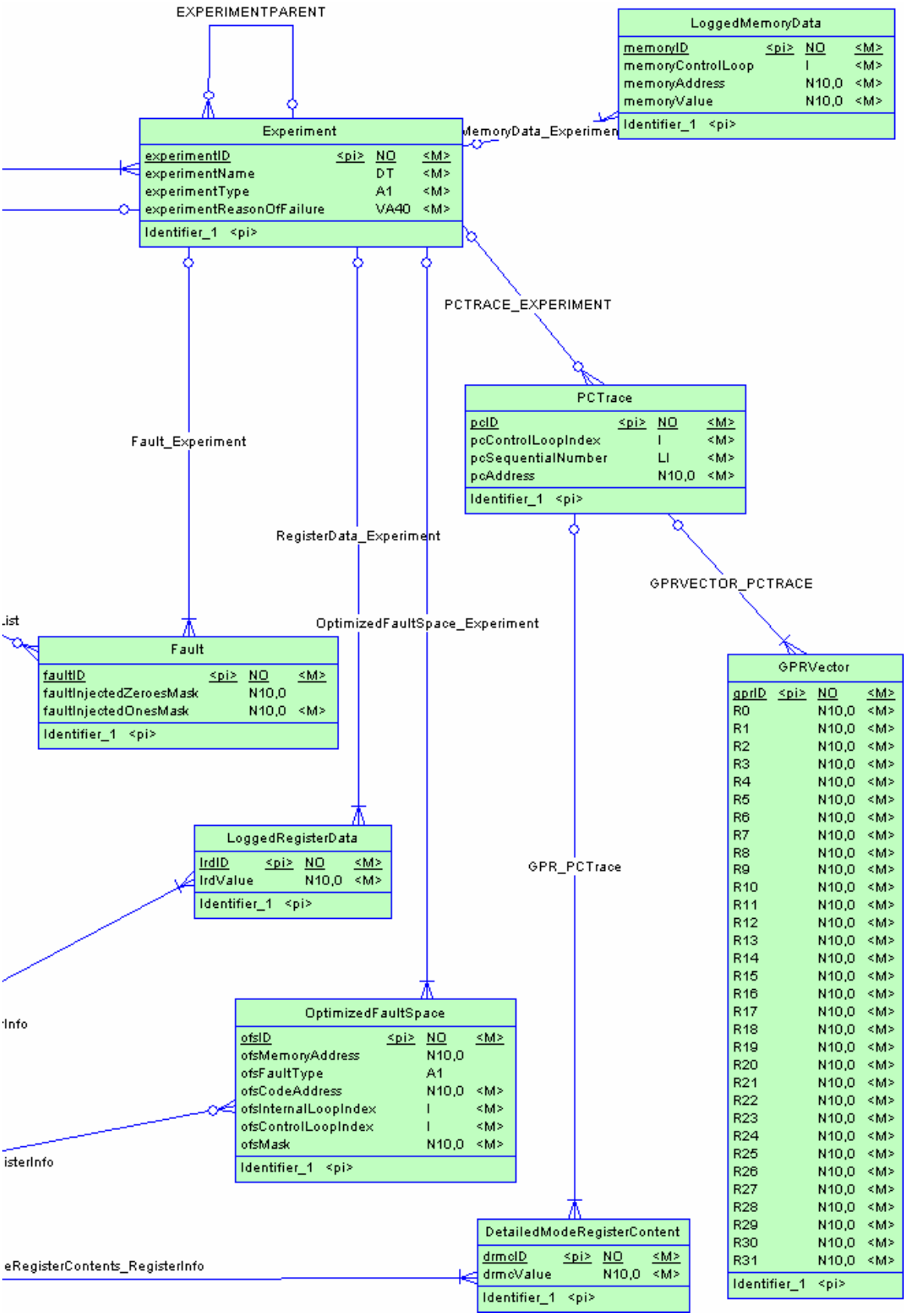
The execution of the reference run takes almost 12 hours for the ten control loops targeted for fault injection[2]. Each fault injection experiment is then performed in less than two minutes for the selected configuration (number of control loops and memory locations to be logged).

3.5. New Database Model

Part of this thesis work included renovating GOOFI's database model. The Entity-Relationship (E-R) diagram of the new database is presented in the following pages.

[1] See Section 3.3.1.
[2] The experimental setup is described in Section 6.3.1.


Campaign
campaignID <pi> NO <M>
campaignName VA50 <M>
campaignWorkloadProjectPath VA50 <M>
campaignWorkloadProjectName VA50 <M>
campaignWorkload VA50 <M>
campaignControlApplication BL <M>
campaignFaultModel VA20 <M>
campaignFaultSpace VA40 <M>
campaignMode VA40 <M>
campaignInfo VA200 <M>
campaignNumberOfExperiments I <M>
campaignNumberOfControlLoops I <M>
campaignNumberOfControlLoopsUntilFI I <M>
campaignNumberOfControlLoopsForFI I <M>
campaignControlLoopBreakpoint N10,0 <M>
campaignTimeout I <M>
campaignStartPoint VA40 <M>
campaignUseStartPoint BL <M>
campaignShortReferenceRun BL <M>
campaignChanged TS <M>
Identifier_1 <pi>
Experiment_Campaign
Campaign_Experiment
SelectedFaultInjectionRange
rangeID <pi> NO <M>
rangeStartAddress N10,0 <M>
rangeEndAddress N10,0 <M>
Identifier_1 <pi>
FaultList
flID <pi> NO <M>
flMemoryAddress N10,0
flType A1 <M>
flFaultNumber I <M>
flControlLoop I <M>
flInnerLoop I <M>
flCodeAddress N10,0 <M>
flMask N10,0 <M>
Identifier_1 <pi>
FaultList_Campaign
SelectedFaultInjectionLocations
faultinjectID <pi> NO <M>
faultinjectType A1 <M>
faultinjectMask N10,0 <M>
faultinjectAddress N10,0
Identifier_1 <pi>
SelectedObservableLocations
observableID <pi> NO <M>
observableType A1 <M>
observableAddress N10,0
observableSize I
Identifier_1 <pi>
FaultList_RegisterInfo
SelectedFaultInjectionLocations_RegisterInfo
SelectedObservableLocations_RegisterInfo
RegisterInfo
riID <pi> NO <M>
riBlockName VA50 <M>
riRegisterName VA50 <M>
riReadableMask N10,0 <M>
riWritableMask N10,0 <M>
Identifier_1 <pi>


EXPERIMENTPARENT
LoggedMemoryData
memoryID <pi> NO <M>
memoryControlLoop I <M>
memoryAddress N10,0 <M>
memoryValue N10,0 <M>
Identifier_1 <pi>
Experiment
experimentID <pi> NO <M>
experimentName DT <M>
experimentType A1 <M>
experimentReasonOfFailure VA40 <M>
Identifier_1 <pi>
PCTRACE_EXPERIMENT
PCTrace
pcID <pi> NO <M>
pcControlLoopIndex I <M>
pcSequentialNumber LI <M>
pcAddress N10,0 <M>
Identifier_1 <pi>
Fault_Experiment
RegisterData_Experiment
GPRVECTOR_PCTRACE
OptimizedFaultSpace_Experiment
Fault
faultID <pi> NO <M>
faultInjectedZeroesMask N10,0
faultInjectedOnesMask N10,0 <M>
Identifier_1 <pi>
GPRVector
gprID <pi> NO <M>
R0 N10,0 <M>
R1 N10,0 <M>
R2 N10,0 <M>
R3 N10,0 <M>
R4 N10,0 <M>
R5 N10,0 <M>
R6 N10,0 <M>
R7 N10,0 <M>
R8 N10,0 <M>
R9 N10,0 <M>
R10 N10,0 <M>
R11 N10,0 <M>
R12 N10,0 <M>
R13 N10,0 <M>
R14 N10,0 <M>
R15 N10,0 <M>
R16 N10,0 <M>
R17 N10,0 <M>
R18 N10,0 <M>
R19 N10,0 <M>
R20 N10,0 <M>
R21 N10,0 <M>
R22 N10,0 <M>
R23 N10,0 <M>
R24 N10,0 <M>
R25 N10,0 <M>
R26 N10,0 <M>
R27 N10,0 <M>
R28 N10,0 <M>
R29 N10,0 <M>
R30 N10,0 <M>
R31 N10,0 <M>
Identifier_1 <pi>
LoggedRegisterData
lrdID <pi> NO <M>
lrdValue N10,0 <M>
Identifier_1 <pi>
GPR_PCTrace
OptimizedFaultSpace
ofsID <pi> NO <M>
ofsMemoryAddress N10,0
ofsFaultType A1
ofsCodeAddress N10,0 <M>
ofsInternalLoopIndex I <M>
ofsControlLoopIndex I <M>
ofsMask N10,0 <M>
Identifier_1 <pi>
DetailedModeRegisterContent
drmcID <pi> NO <M>
drmcValue N10,0 <M>
Identifier_1 <pi>
eRegisterContents_RegisterInfo

4. FAULT-SET OPTIMIZATION METHOD

The fault-set optimization method presented in this thesis states, as mentioned before, that faults should only be placed in the resources read by each instruction. The following sections describe how this is achieved in the target system introduced in the previous chapter.

4.1. Optimization Input

In order to determine the optimized fault-set it is necessary to gather information about the code of the application and the computer system executing it. More precisely, the input required by the method is:

- Assembly code of the application;
- A vector containing the Program Counter (PC) trace over time;
- The values of the General Purpose Registers before each memory read access[1];
- The definition of which resources are read by each assembly instruction.

The **assembly code** is obtained by disassembling the executable (ELF) binaries of the application.

The **Program Counter vector** and the **values of the General Purpose Registers** are stored during the execution of the reference run. Even though the values of the GPRs are only required before each memory access, they were stored for every instruction executed to simplify implementation.

The **definitions** of which resources are read by each assembly instruction were obtained from Motorola's RISC CPU Reference Manual [19] and are presented in Appendix A.

4.2. Optimization Output

The resulting output (the optimized fault-set) consists of a list of possible locations and times for fault injection. Each element on this list contains the following information:

- Control loop index;
- Internal loop index within the control loop;

[1] The values of the GPRs are required to calculate the *effective address* for memory read instructions.

- Code address;
- The resource and bits that should be targeted.

The **control loop index** is specific for control applications which execute in cycles. It defines the cycle during which a fault should be injected. For applications like *quicksort* that do not execute in cycles, the control loop index is always set to one.

The **code address** and the **internal loop index** specify the breakpoint position inside the control loop and the number of times this breakpoint should be reached before fault injection occurs.

The **resource** specifies the register or memory location that should be injected with the fault and the bits that are to be flipped.

4.3. Performing the Optimization

Using the Program Counter trace over time, the disassembled code of the application is parsed to obtain the sequence of assembly instructions executed. Each of the instructions is then analyzed in order to determine which resources the instruction reads. The pseudo-code for this procedure is presented in Figure 4:

```
FOR i ← 1 TO number_of_instructions_executed DO
    code_address ← program_counter_vector[i]
    instruction ← instruction_at_code_address (code_address)
    instruction_read_list ← resources_read_by_instruction (instruction)
    WHILE instruction_read_list.has_elements() DO
        resource ← instruction_read_list.next_resource()
        useful_fault ← (control_loop_index, internal_loop_index, code_address, resource)
        store_in_database (useful_fault)
    ENDWHILE
ENDFOR
```

Note: the initialization of variables *control_loop_index* and *internal_loop_index* is not shown in the pseudo-code (their value is particular to each instruction).

Figure 4 - Pseudo-code for the optimization procedure

The identification of the resources read by each instruction is accomplished by applying assembly-level knowledge of the system. Appendix A presents the list of definitions used to study the MPC565 instruction-set[1].

The first step for this identification is to find the definition on the list matching the given instruction. This is done by matching the *opcode* and the *operands*. Then, by studying the possible assembly constructs, the symbols available in the *read list* of the definition are replaced by the resources actually read by the given instruction. Figure 5 illustrates this process.

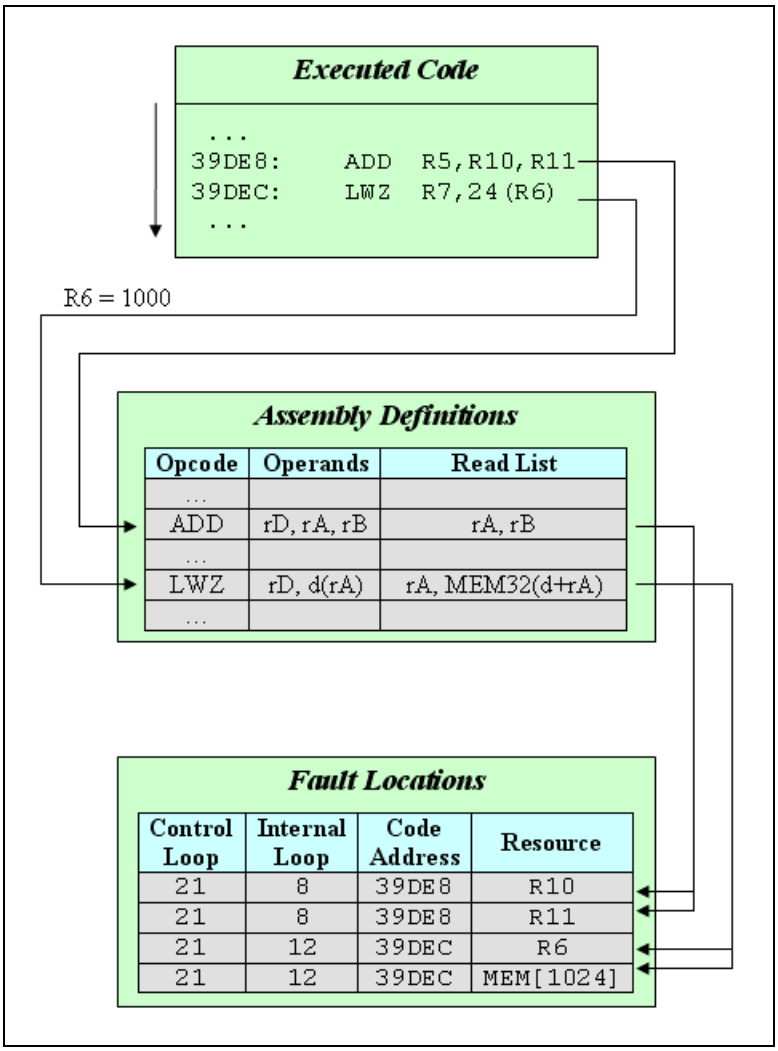


Figure 5 - Example of the optimization procedure

The instruction at address 39DE8 adds R10 to R11 and stores the result in R5. The definition for this instruction is found in the table and the read list contains rA and rB,

[1] All the instructions considered are defined by the PowerPC User Instruction Set Architecture (UISA).

respectively, R10 and R11. Since these are the two resources read by this instruction, two new lines are inserted into the fault locations for code address 39DE8 (the control loop and the internal loop are assumed to hold the values specified).

The second instruction, at address 39DEC, fetches the memory word addressed by the effective address (R6) + 24 and stores it in R7. Its definition in the table specifies rA and MEM32(d+rA), respectively, R6 and the 32-bit word at 1000[1]+24, as being read. These two resources are then inserted into the list of fault locations.

[1] The value of R6 is obtained during the reference run (see Section 4.1).

4.4. Further Optimization

Further optimization may be achieved by studying the propagation of errors after fault injection. Code execution sometimes moves the contents of a given resource onto another resource. Another equivalence class may be found when this happens since, under certain conditions, a bit-flip in the first resource is equivalent to a bit-flip in the second resource.

To simplify the description *bit-flip transitivity* is defined as the following property of operations: bit-flip transitivity occurs when a single bit-flip in one of the operands implies, at most, a single bit-flip in the result of the operation.

An example of bit-flip transitive operation is the AND instruction. This instruction performs a binary AND on the two source operands and places the result in the destination operand. A bit-flip in one of the operands implies, at most, a bit-flip in the result. Loads from memory are usually bit-flip transitive too. The contents of memory are copied onto registers. In this case, one bit-flip in the memory location results exactly in one bit-flip in the destination register.

Using this concept it is possible to further reduce the fault-set under certain conditions. This thesis only explored two particular situations. After a bit-flip transitive instruction is executed, one of four possibilities happens during the execution of subsequent instructions:

a) The source operand is written;
b) The source operand is read;
c) The destination operand is written;
d) The destination operand is read.

This thesis studied only cases a) and c). Cases b) and d) are left open for future work. The same four possibilities, but in the case of instructions that are not bit-flip transitive, are also left open.

Assume that a single bit-flip has been injected in the source operand of the bit-flip transitive instruction. If the source operand is written by a subsequent instruction, then the bit-flip in that resource will be overwritten, but it will remain in the destination operand. Thus, an equivalence class may be created over two resources that are operands of a bit-flip transitive instruction.

Similarly, if the destination operand is overwritten by a subsequent instruction, the original bit-flip had no effect on the system so far. When this situation occurs the fault has already been activated (the resource was read) but, after the execution of several

instructions, there is still only one erroneous bit in the system and it is still non-effective. An equivalence class can therefore be created over multiple reads of a resource when the error only can propagate via the last read. In this case it is suitable to inject the bit-flip just before the last read.

The following table exemplifies a situation where this optimization may be performed:

Address	*Instruction*	*Comment*
…		
4CF74:	AND R10, R15, R16	R15 is ANDed with R16 and the results are placed in R10.
4CF78:	LWZ R15, 32(R1)	R15 is overwritten with data from memory. Any bit-flip in R15 in the previous instruction was "transferred" onto R10. Thus, it is only necessary to inject faults in R10 in this equivalence class.
…		

Table 1 - Example of optimization using bit-flip transitivity

To perform this optimization it is necessary to keep track not only of read accesses on registers but also of write accesses. This is the reason why the assembly definitions presented in Appendix A contain the locations written by each instruction. The rightmost column informs whether or not the instruction is bit-flip transitive and which of the operands of this instruction manifest this property.

5. IMPLEMENTATION IN GOOFI: OFFSET

The optimization method presented in the previous chapter was implemented in the GOOFI fault injection tool. This integrated application was named OFFSET (Optimized Full Fault-Set Exploration).

It consists of two distinct modules responding to calls from GOOFI: the analysis module and the fault-selection module. The analysis module is invoked after the reference run with the purpose of performing the optimization and storing the fault-set in the database. The fault-selection module is called before each fault injection experiment in order to determine the next fault to be injected.

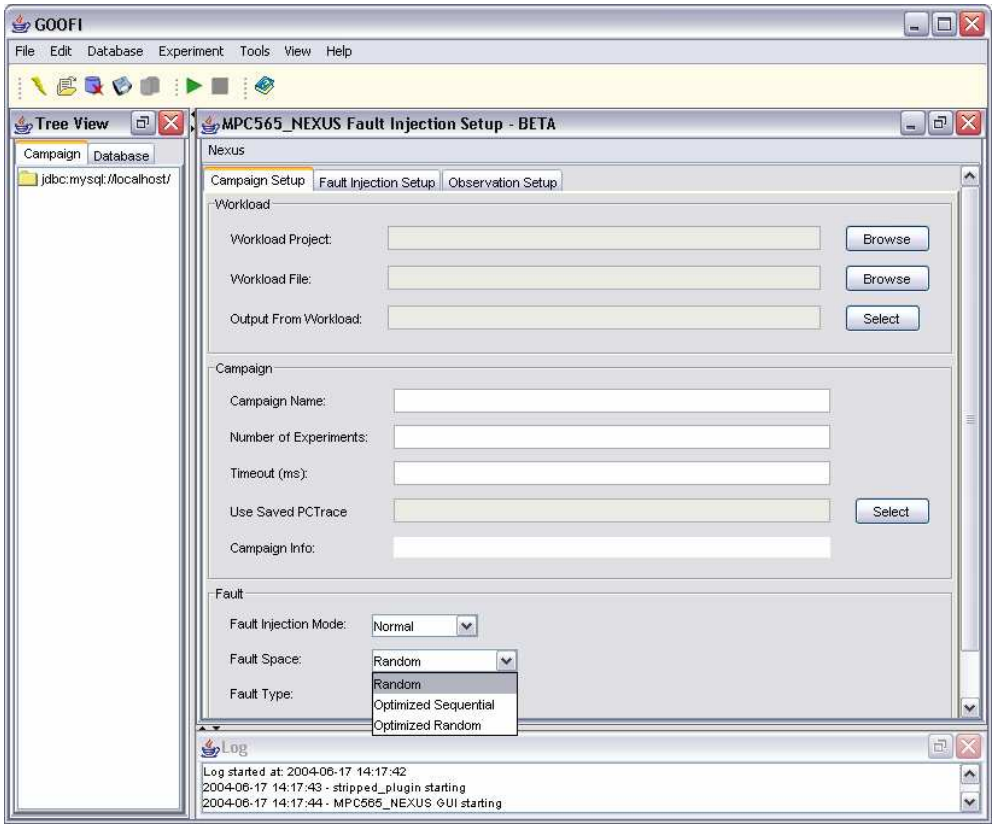


Figure 6 - OFFSET integrated with GOOFI

5.1. Analysis Module

The analysis module is responsible for optimizing the fault-set and storing the result[1] in the database. The database model presented in Section 3.5 includes the *OptimizedFaultSpace* table. This table is filled with "useful" fault locations after each reference run.

The class seen from the exterior (GOOFI) is named *AnalysisEngine*. After finishing the reference run GOOFI instantiates an AnalysisEngine object and calls the methods described in the following paragraphs:

public void loadCodeFromFile(File codeFile)

This method will open the provided file and parse it in order to obtain the code of the application. The instantiated AnalysisEngine will, from then on, have access to the instructions executed and their address.

public void loadDefinitionsFromFile(File definitionsFile)

Calling this method makes the AnalysisEngine open the file provided and load the assembly definitions from it. From this moment on the instantiated AnalysisEngine is able to use the assembly definitions in order to study the executed code.

public boolean performPreAnalysis(Connection dbConnection, String campaignName, boolean applySecondOptimization)

After calling methods loadCodeFromFile and loadDefinitionsFromFile the analysis module is ready to perform the optimization. Calling method performPreAnalysis with the given parameters carries out this step.

The parameter *dbConnection* should be a connection to the database currently used for the campaign under analysis. This campaign is identified by the parameter *campaignName*. The third parameter, *applySecondOptimization*, determines whether or not the second optimization described in Section 4.4 should be applied. With this information the AnalysisEngine is able to retrieve all required data from the database.

[1] See optimization output in Subsection 4.2.

5.2. Fault-Selection Module

The fault-selection module is used to select the fault from the optimized fault-set stored in the database before each fault injection experiment.

This module is presented in the form of an abstract class named *FaultSpace*. Any implementation of this abstract class presents the following methods:

abstract boolean nextFault();

A call to this method moves to the next fault from the current position. The first call to the method *nextFault()* makes the first fault the current fault. The second call makes the second fault the current fault, and so on.

abstract int getFaultControlLoopIndex();

Returns the *control loop index* for temporal positioning of the current fault.

abstract int getFaultInternalLoopIndex();

Returns the *internal loop index* for temporal positioning of the current fault.

abstract long getFaultCodeAddress();

Returns the *code address* for spatial positioning of the current fault.

abstract String getFaultObject();

Returns the resource that should be targeted with the current fault.

abstract long getFaultBitMask();

Returns the mask with the bit from the resource that should be targeted with the current fault.

There are currently two implementations of the abstract class FaultSpace: OptimizedFaultSpaceRandom and OptimizedFaultSpaceSequential. The first connects to the database when instantiated and subsequent calls to the method nextFault() will randomly select the next fault from the optimized fault space. The sequential version also connects to database to select the fault from the optimized fault-set but does this sequentially. This ensures that the fault-set is thoroughly studied. It should only be applied when a particular block of code (for instance one procedure) or a given resource is to be studied through fault injection since the complete fault-set is usually still too large for all faults to be injected.

The selection of which code addresses, memory locations or registers that should be injected with faults is done though the constructor of each implementation. The constructor of the random version of the OptimizedFaultSpace is as follows:

OptimizedFaultSpaceRandom(Connection dbConnection, String campaignName, int[] codeRanges, int[] memoryRanges, String[] registerList)

When the GOOFI fault injection tool instantiates an OptimizedFaultSpaceRandom object, a campaign name and a connection to the database containing this campaign should be supplied. Code ranges, memory ranges and register list are optional. The remaining methods in this class are the implementations of the methods in the abstract class FaultSpace described above. The OptimizedFaultSpaceSequential class works in a similar way.

6. EXPERIMENTAL RESULTS

6.1. Introduction

6.1.1. Summary

This chapter presents the results of fault injection campaigns targeting the two tested workloads: a quicksort implementation and a jet engine controller. The quicksort application was used as the workload during the development stages of the optimization tool. It executes 34 distinct assembly instructions (opcodes) and the complete support on the tool was available early on. When the support for all instructions executed by the controller was developed (88 assembly instructions) it was possible to test the method with this more sophisticated, real-world application. At this point the support for optimization of faults in memory was also available.

6.1.2. Targeted Registers

The processor registers were selected as spatial locations for fault injection both in the quicksort and in the jet-engine controller campaigns. Memory locations were only targeted using the jet-engine controller. The registers targeted in the random campaigns were the ones considered by the optimization method:

- General Purpose Registers (32 registers of 32 bits);
- Floating Point Registers (32 registers of 64 bits);
- Condition Register (32 bits);
- Integer Exception Register (32 bits);
- Link Register (32 bits);
- Count Register (32 bits).

These registers constitute the User Instruction Set Architecture (UISA) Register Set. The UISA registers can be accessed by either user- or supervisor-level instructions and user-level instructions can only operate on these registers.

Two limitations of winIDEA (the debugging environment) are important to mention. The floating point registers are only allowed to be injected with faults in the least significant 32 bits. These are the least significant bits of the 52-bit mantissa. The Floating Point Status And Control Register (FPSCR) was removed from the list because it is not available for fault injection also due to a limitation of winIDEA.

6.2. Quicksort

6.2.1. Introduction and Experimental Setup

This section presents the results from fault injection campaigns evaluating the quicksort application, targeting only registers through random and optimized fault injection.

During the execution of the golden-run for this application, the MPC565 processor executed 34 distinct assembly instructions (opcodes) and a total of 815 instructions.

Both campaigns targeted the registers considered by the optimization method and described in the previous section. Registers were targeted using both random and optimized fault injection, according to the fault model described in the next paragraphs.

The faults in the random campaigns were chosen using a uniform distribution for the temporal location. Then, the targeted resource is also determined using a uniform distribution. The bit that should be flipped within the resource is then chosen with a uniform distribution too. Since all registers are 32-bits[1] long, the result is a uniform distribution of faults in the non-optimized fault-set.

A different approach was used to select faults in the optimized campaigns. A fault is selected uniformly from the optimized fault-set itself[2] (temporal and spatial location at the same time). This implies for instance that the distribution of faults in resources is proportional to the representation of each resource in the fault-set. Resources with more read accesses (useful fault locations) will, therefore, be injected with more faults.

[1] Floating Point Registers are 64-bits long limited to 32-bits (see Section 6.1.2).
[2] The fault-space is the list of locations for fault injection described in Section 4.2.

6.2.2. Fault Injection Results

The optimization method eliminates faults when an equivalence class is determined or when the targeted resource is not being used. Since faults placed in unused registers are never activated and, therefore, are always non-effective, the optimization is expected to decrease the percentage of non-effective faults. The outcomes of the fault injections were classified in three classes: *non-effective*, *detected* and *wrong output*. The distribution of the outcomes is presented in Table 2.

Campaign	*# Experiments*	*Non-effective*	*Detected*	*Wrong Output*
Optimized	2791	1461 (52.3%)	744 (26.7%)	586 (21.0%)
Random	2739	2603 (95.0%)	83 (3.0%)	53 (2.0%)

Table 2 - Comparison of outcome distribution

As expected, the number of effective faults was higher for the optimized fault selection technique. The improvement is one order of magnitude. Graphical representations of the results are shown in Figures 7 and 8.

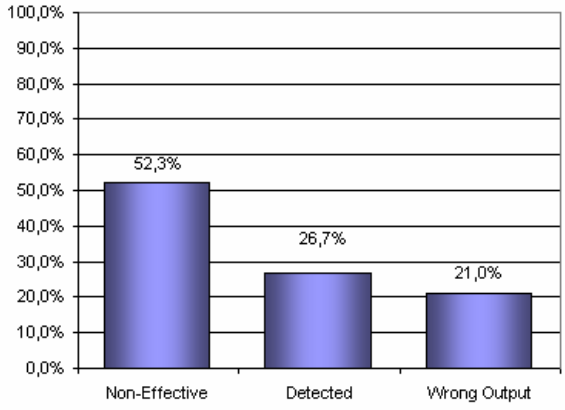


Figure 7 - Outcome of experiments in the optimized campaign

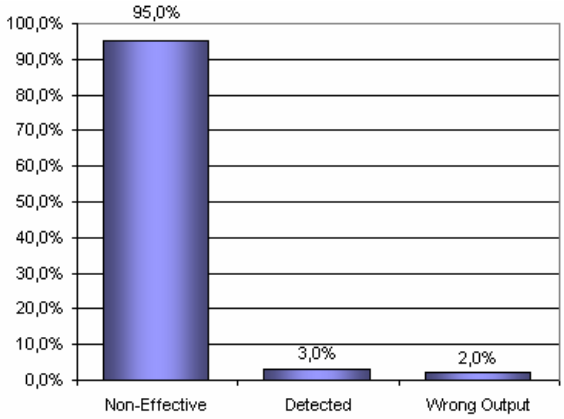


Figure 8 - Outcome of experiments in the random campaign

Table 3 shows the estimated error detection coverage. Error detection coverage is defined as the quotient between the number of detected and effective (detected or wrong output) outcomes:

Campaign	*Estimated error detection coverage (95% confidence)*
Optimized	55.9 ± 2.7%
Random	61.0 ± 8.2%

Table 3 - Error detection coverage estimations

Even though the estimated error coverage is similar for random and optimized fault injection, the 95% confidence interval for the random campaign is much wider. This is the result of the low effectiveness of random faults. In order to narrow the interval to the same size as in the optimized fault injection the number of experiments would have to be around ten times larger. More considerations on error detection coverage estimation can be found in Section 6.3 that describes the results of the experiments with the jet engine controller.

All the detected errors were detected by the exceptions provided in the MPC565 processor. The distribution among these exceptions is presented in Figure 9 for the optimized campaign, and in Figure 10 for the random campaign.

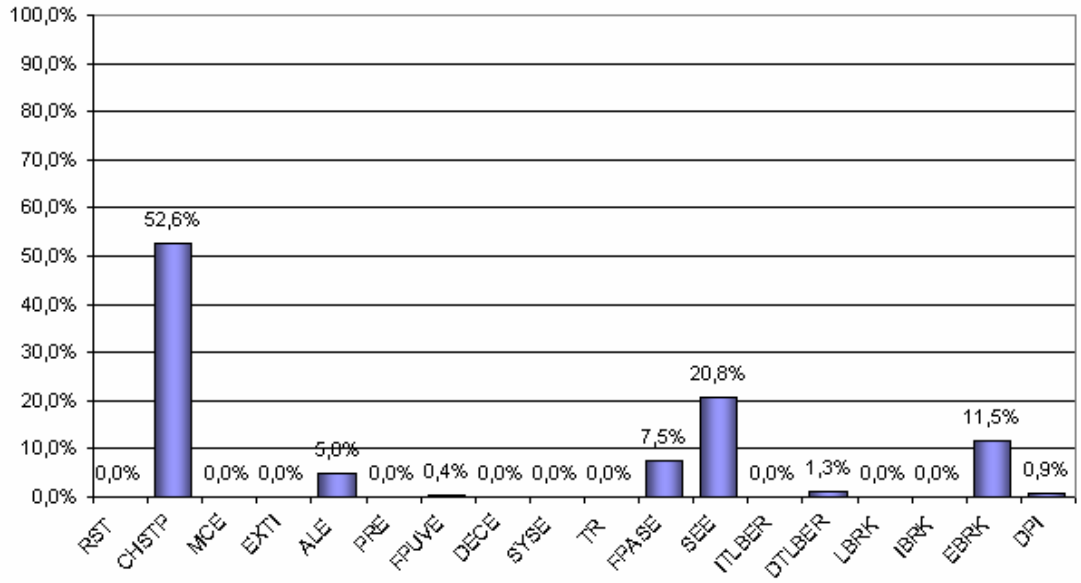


Figure 9 - Exception distribution in the optimized campaign (744 faults in registers)

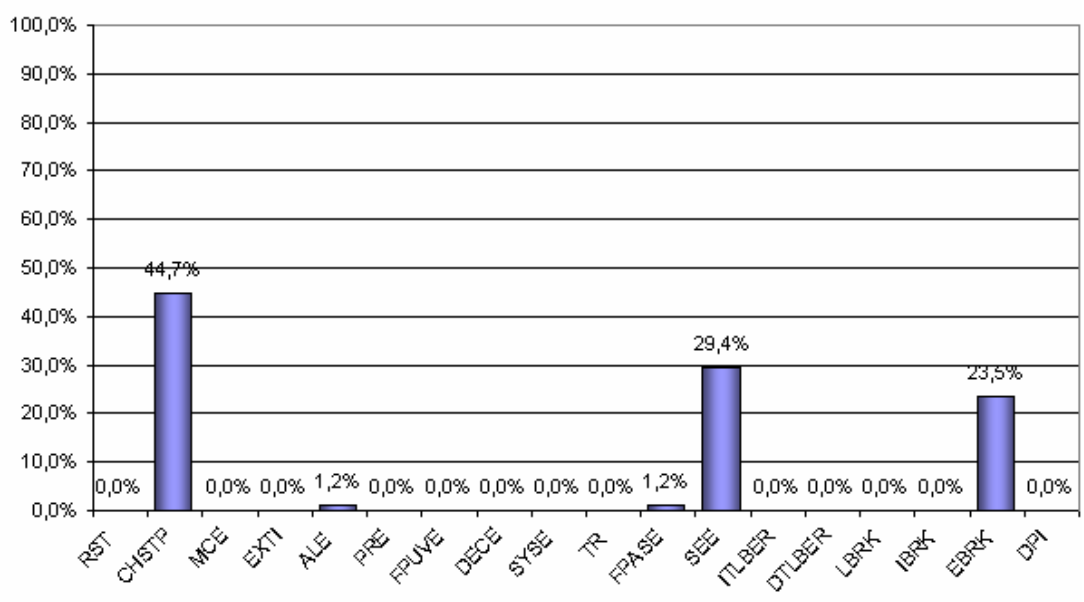


Figure 10 - Exception distribution in the random campaign (83 faults in registers)

The detection mechanisms are activated in a similar way for the random and the optimized campaign. The small number of experiments that produced detected errors in the random campaign doesn't allow strong conclusions about this fact.

It is also possible to observe that optimized fault injection activated detection mechanisms not activated with random injection, performing the same number of experiments.

6.3. Jet Engine Controller

6.3.1. Introduction and Experimental Setup

This section presents the results from fault injection campaigns evaluating the jet engine controller, targeting registers and memory, using both random and optimized fault injection.

Ten control loops of execution were chosen as possible temporal locations for fault injection. During these ten control loops the MPC565 processor executed 231.097 instructions in the reference (fault-free) run. A total of 88 different assembly instructions were executed.

The spatial locations were divided into registers and memory. The first two campaigns, described in Section 6.3.2, targeted microprocessor registers. In Section 6.3.3, the results from fault injection campaigns targeting memory (read-only data, stack, heap and other read/write data segments – sbss and bss) are described.

In the first two campaigns the registers targeted consist of the registers considered by the optimization method and previously described in Section 6.1.2.

The fault injection campaigns in memory targeted the stack, heap, sbss (read/write data), bss (read/write data) and read-only data segments of the controller. A total of around 100KB of memory were targeted as spatial locations. Faults in the code segment were not studied since the code is usually stored in ROM. The optimization would, nonetheless, be easily extended to support the optimization of faults in this segment.

Registers and memory were targeted using both random and optimized fault injection. The fault-model used is described in the following paragraphs.

The faults in the random campaigns were chosen using a uniform distribution for the temporal location. Then, the resource is also determined using a uniform distribution. The bit that should be flipped within the resource is then chosen with a uniform distribution too. Since all registers are 32-bits long, the result is a uniform distribution of faults in the non-optimized fault-set.

As mentioned before, a different approach is used to select faults in the optimized campaigns. A fault is selected uniformly from the optimized fault-set itself (temporal and spatial location at the same time). This implies that the distribution of faults in resources is proportional to the representation of each resource in the fault-set. Resources with more read accesses will, therefore, be injected with more (useful) faults.

6.3.2. Fault Injection in Registers

Applying the optimization method to the set of faults in registers resulted in the determination of 7.776.018 possible bit-flips. All the registers mentioned above are 32 bit registers[1]. The complete fault-set of these registers is obtained by flipping each bit of each register, for each instruction executed. Its size is over 500 million bit-flips. The following table summarizes these results:

Campaign	***Size of the fault-set (bit-flips)***
Optimized	7.7×10^6
Random	5.0×10^8
Ratio	1.5%

Table 4 - Comparison between fault-set sizes (registers)

The optimization reduced the fault-set to 1.5% of its original size. Faults are eliminated when an equivalence class is determined or when the resource is not being used. The percentage of non-effective faults is expected to decrease using this approach, for the reason stated in Section 6.2.2. Table 5 presents the distribution of the outcomes of faults:

Campaign	***# Experiments***	***Non-effective***	***Detected***	***Wrong Output***
Optimized	1559	964 (61.8%)	466 (29.9%)	129 (8.3%)
Random	5708	5457 (95.6%)	200 (3.5%)	51 (0.9%)

Table 5 - Comparison of outcome distribution (registers)

As expected, the effectiveness of faults increased for the optimized fault-set with respect to the random fault-set. The improvement is of one order of magnitude. Figures 11 and 12 present a graphical representation of this.

[1] Floating Point Registers are 64-bits long limited to the least significant 32-bits (see Section 6.1.2).

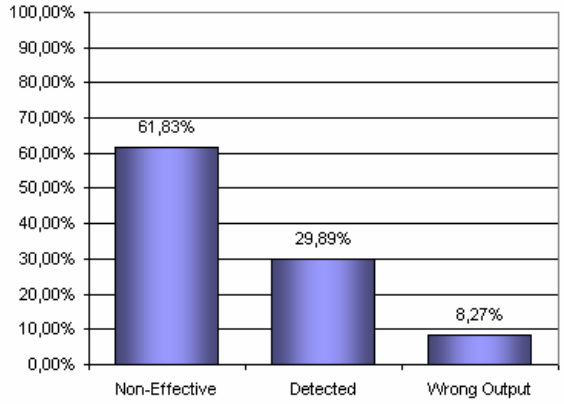


Figure 11 – Outcome of experiments in the optimized campaign targeting registers

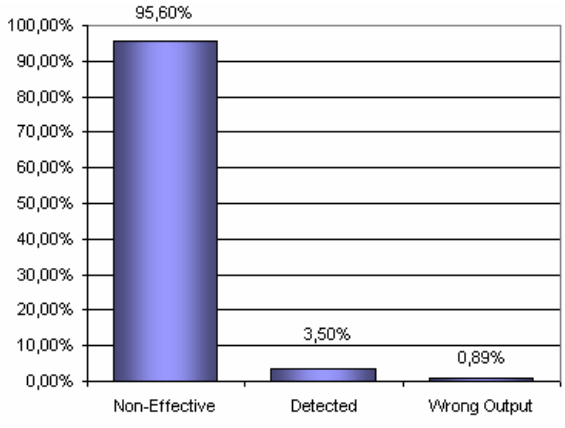


Figure 12 - Outcome of experiments in the random campaign targeting registers

The error detection coverage obtained from the two campaigns is shown in Table 6.

Campaign	***Estimated error detection coverage (95% confidence)***
Optimized	78.3 ± 3.3%
Random	79.7 ± 5.0%

Table 6 - Error detection coverage estimations (registers)

The values of the error detection coverage estimations are quite similar. This is a finding, in the sense that it was not anticipated. In the optimized campaigns the faults are only injected in the location that will activate them (at the time that the register is read). Since no weights are applied to reflect the length of the data life-cycle on the outcomes of faults, it could be expected that the error detection coverage would be skewed.

The distribution of detection among the exceptions provides further insights about the activated faults in the two campaigns, as shown in Figure 13 and 14.

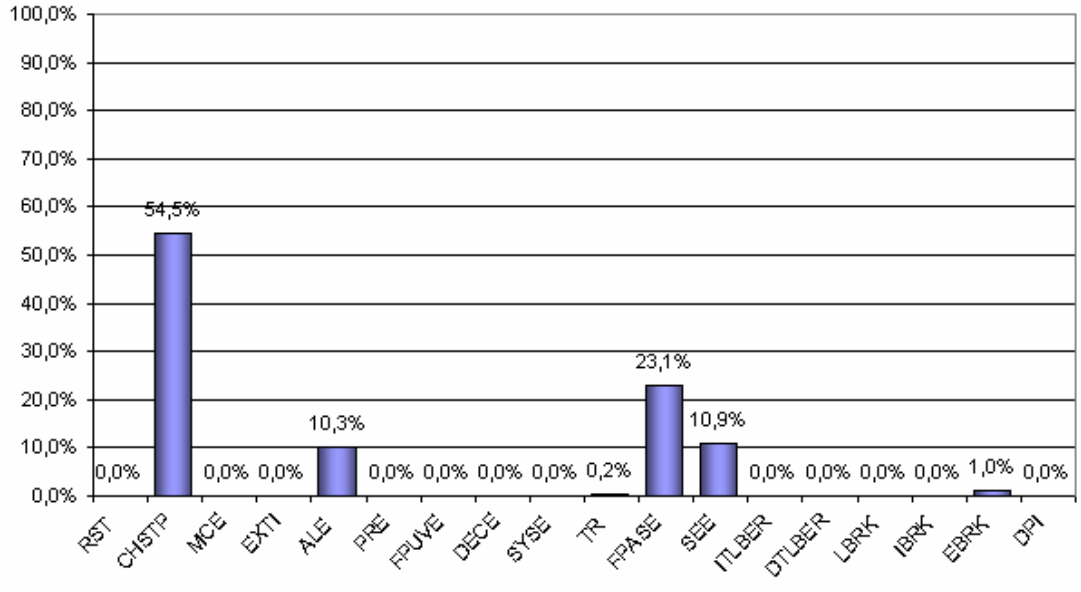


Figure 13 - Exception distribution in the optimized campaign (466 faults in registers)

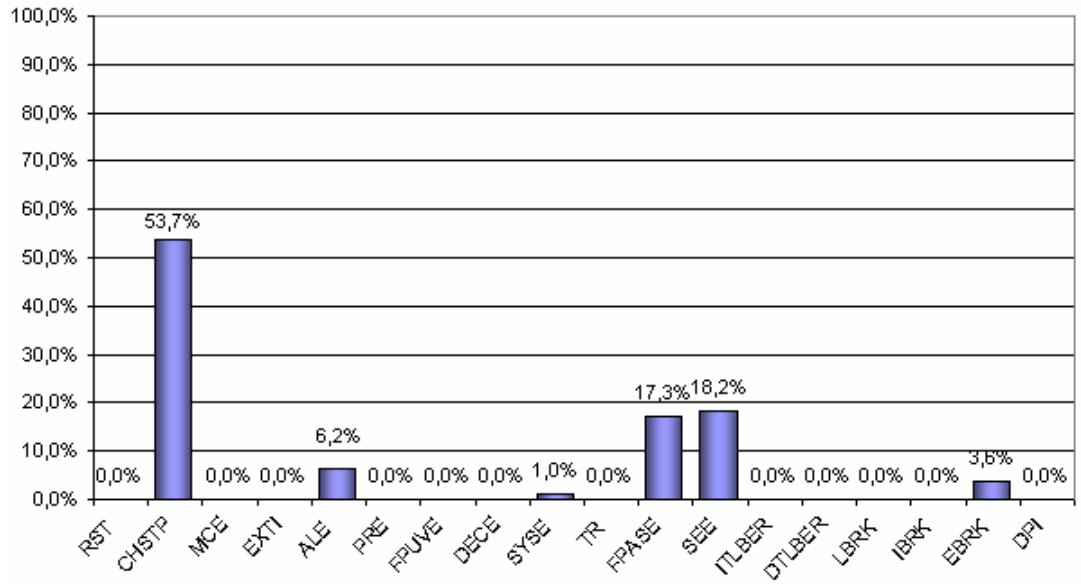


Figure 14 - Exception distribution in the random campaign (200 faults in registers)

It is possible to observe that the detection mechanisms are activated in a similar but not identical way for the random and the optimized campaigns. Different exception distributions were expected and Figures 13 and 14 provide an insight on the magnitude of these differences.

Figures 15 and 16 show the incidence of detected errors for each register injected with faults. Only the 30 registers with highest ranking are presented.

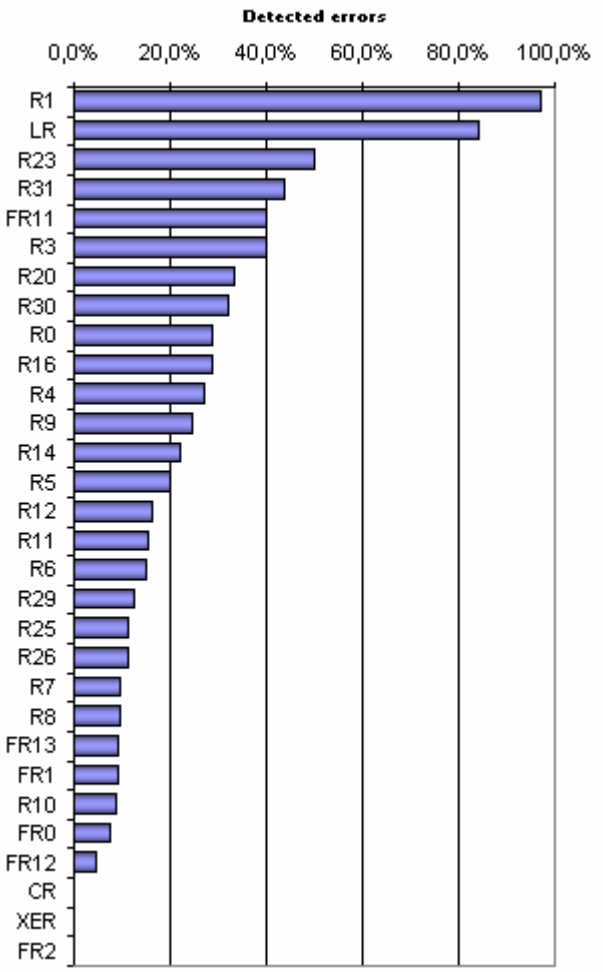


Figure 15 - Detected errors per register in the optimized campaign

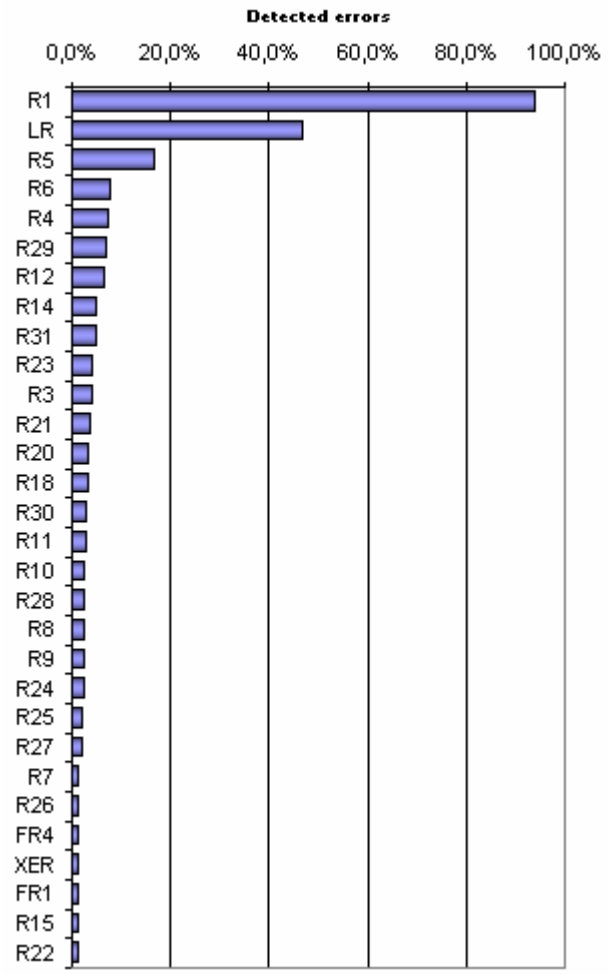


Figure 16 - Detected errors per register in the random campaign

The two registers at the top of both lists are R1 and LR, respectively, the Stack Pointer and the Link Register of this architecture. Since their contents are vital for the application, faults targeting these are expected to be detected. Figures 15 and 16 clearly illustrate the increase in the effectiveness of faults at the level of the resource (registers in this case).

It is interesting to observe that the registers are sorted differently in the two campaigns. This shows that the behavior of the system is different in the case of optimized and random (uniform) fault injection.

The faults that force the system to produce wrong output are usually the most important to study. These are faults that violate fail-silent behavior. It might therefore be useful to know which resources are responsible for this behavior and, eventually, study these in more depth. Figures 17 and 18 show the 30 resources mose likely to produce wrong results:

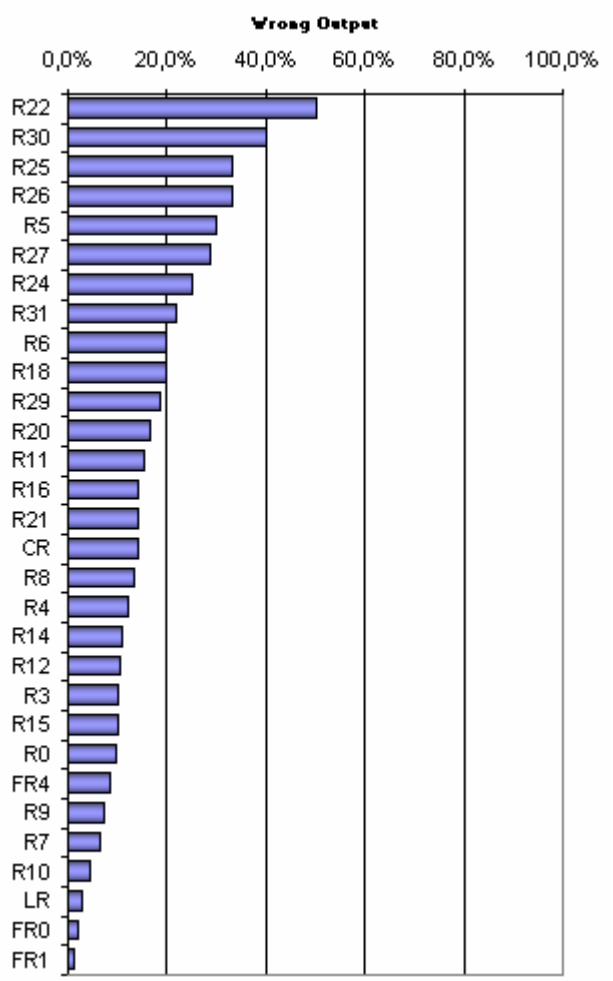


Figure 17 - Wrong output per register in the optimized campaign

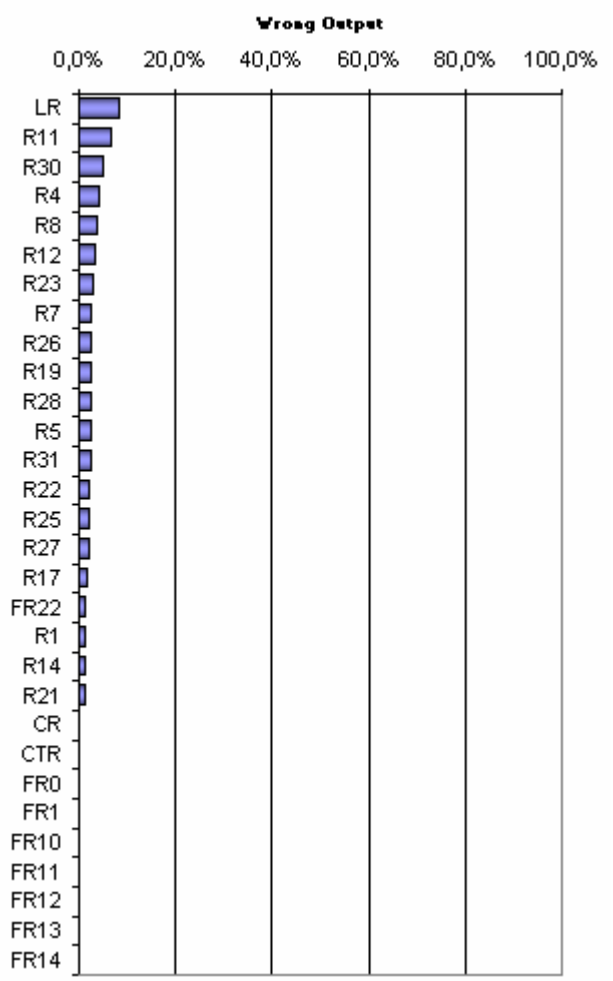


Figure 18 - Wrong output per register in the random campaign

The probability of producing a wrong output is much higher in the optimized campaign. The order of the registers is also different in the two campaigns. This shows, again, that the behavior of the system is different for the optimized campaign and the random campaign.

Figure 19 shows the 30 registers where faults are most likely to be non-effective for the optimized campaign.

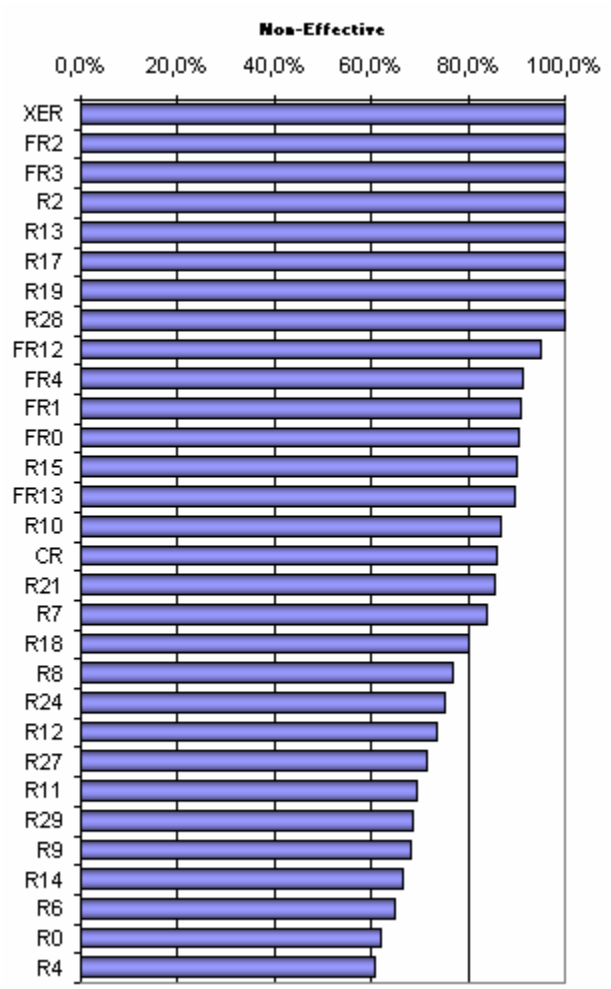


Figure 19 - Non-effective errors per register in the optimized campaign

The comparison with the random campaign is not presented since most of the registers targeted in the random campaign resulted in non-effective errors. The optimization method eliminated most of the registers that display this behavior. In the random campaign, faults injected in 34 registers always resulted in non-effective errors. Most of these faults were eliminated by the optimization method. The eight registers that are still non-effective could be the basis for future study on why activated faults are non-effective.

Finally, Figure 20 shows the distribution of faults per register for the optimized campaign. The figure clearly demonstrates the non-uniform distribution caused by the optimization. The number of faults per register is directly proportional to the number of times the register is read[1].

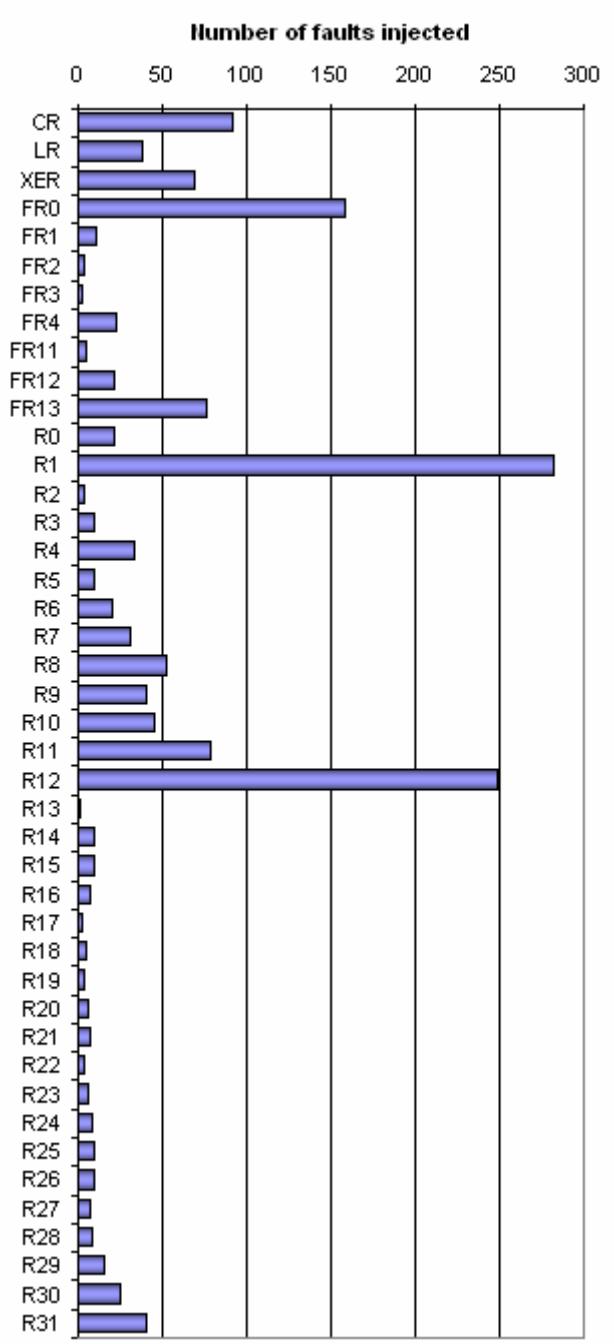


Figure 20 - Number of faults injected per register in the optimized campaign

[1] Notion of useful fault presented in Chapter 4.

6.3.3. Fault Injection in Memory

In the case of the memory fault-set 3.300.728 possible bit-flips were determined. The complete fault-set of memory is obtained by flipping each bit of each memory location used by the program, for each instruction executed. Considering a memory usage of 100KB by the jet engine controller, the size of the complete fault-set is near 200 billion bit-flips. Table 7 summarizes these results:

Campaign	***Size of the fault-set (bit-flips)***
Optimized	3.3×10^{6}
Random	1.9×10^{11}
Ratio	0.0017%

Table 7 - Comparison between fault-set sizes (memory)

The optimization reduced the size of the original fault-set five orders of magnitude. In a way similar to the registers in the previous section, faults are eliminated when an equivalence class is determined or when the resource is not being used. Again, the percentage of non-effective faults is expected to decrease using this approach. Table 8 presents the distribution of the outcomes of faults:

Campaign	***# Experiments***	***Non-effective***	***Detected***	***Wrong Output***
Optimized	2658	2150 (80.9%)	166 (6.3%)	342 (12.8%)
Random	6666	6532 (98.0%)	40 (0.6%)	94 (1.4%)

Table 8 - Comparison of outcome distribution (registers)

As expected, the effectiveness of the faults was increased. The improvement is similar to the one found in registers – one order of magnitude. Figures 21 and 22 present a graphical representation of this. Comparing these results to the results of fault injection experiments in registers it is possible to observe that the number of non-effective errors is much higher in the optimized campaign in memory than in the optimized campaign in registers.

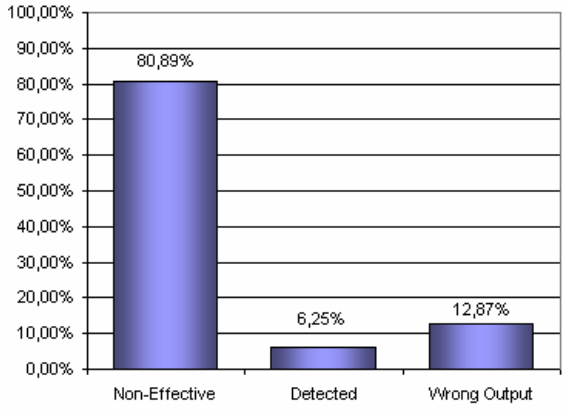


Figure 21 – Outcome of experiments in the optimized campaign targeting memory locations

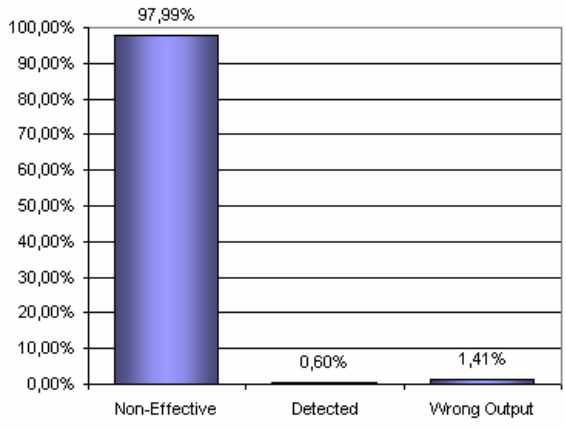


Figure 22 - Outcome of experiments in the random campaign targeting memory locations

At this point it is possible to calculate the error detection coverage – the quotient between detected and effective (detected or wrong output) – estimated by performing optimized or random fault injection in the targeted memory regions:

Campaign	***Estimated error detection coverage (95% confidence)***
Optimized	32.7 ± 4.1%
Random	29.9 ± 7.7%

Table 9 - Error detection coverage estimations (memory)

The values of the error detection coverage estimations are quite similar. In this case the estimation from the random campaign is not very reliable since the 95% confidence interval is still wide due to the small number of effective faults (2%). It is, nonetheless, a finding similar to the error detection coverage estimation in registers using both the jet engine controller and the quicksort. In the optimized campaigns the faults are only injected in the location that will activate them (at the time that the memory address is read). Since no weights are applied to reflect the length of the data life-cycle on the outcomes of faults, it could be expected that the error detection coverage would be skewed.

Figure 23 and 24 show distribution of detected errors among the exception mechanisms for the two campaigns.

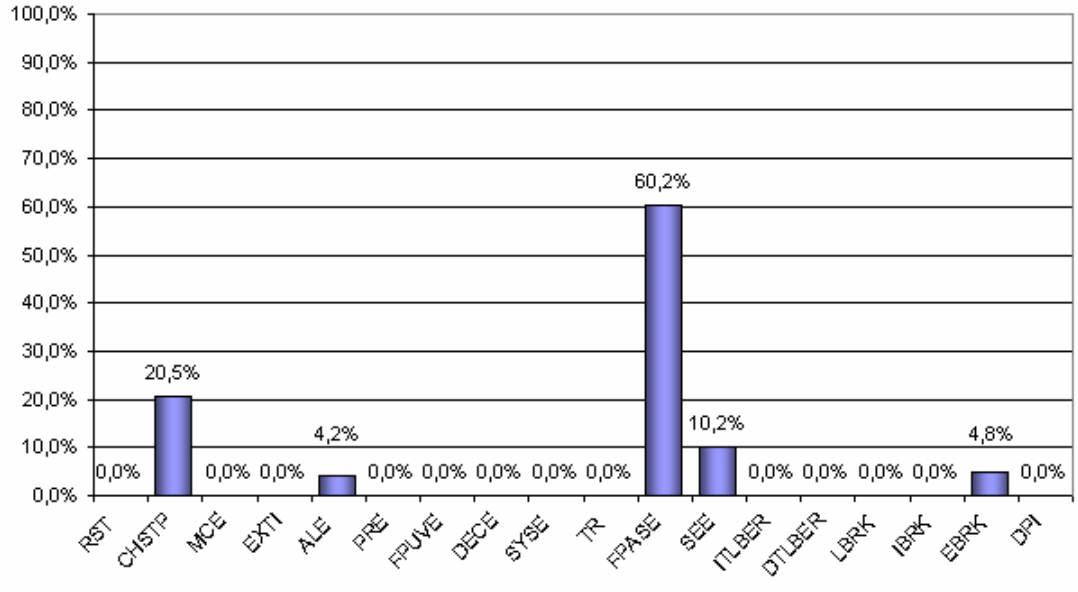


Figure 23 - Exception distribution in the optimized campaign (166 faults in memory)

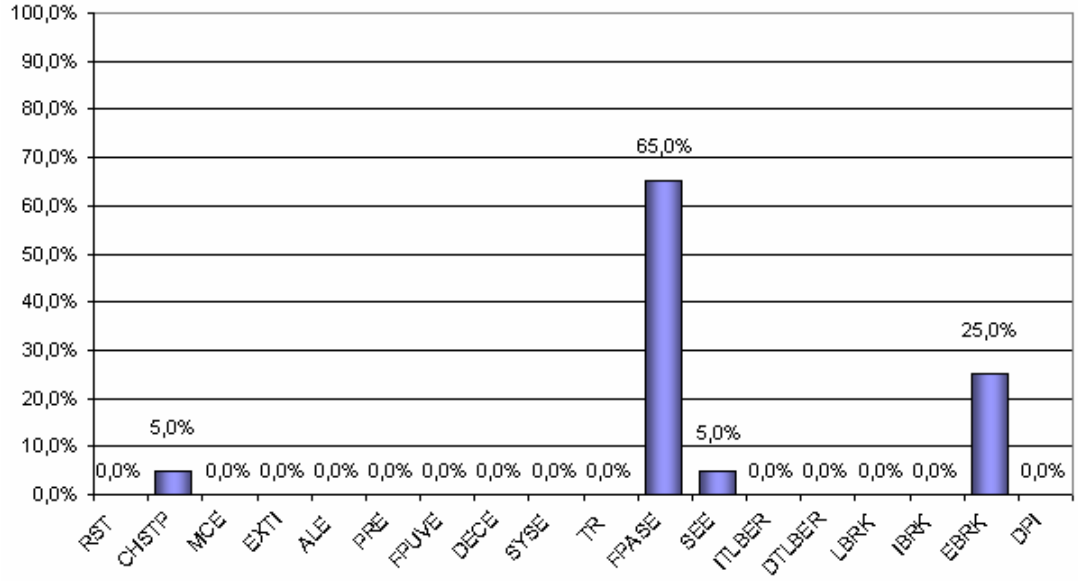


Figure 24 - Exception distribution in the random campaign (40 faults in memory)

Again, it is possible to observe that the detection mechanisms are activated in a similar but not identical way for the random and the optimized campaigns. Some differences were expected and Figures 23 and 24 provide an insight on the magnitude of these differences.

6.4. Results of Further Optimization

The second optimization strategy (presented in Section 4.4) was applied to the complete fault-set targeted in the two jet engine campaigns – memory and registers together. The fault-set was reduced 10% more. This result shows that it is possible to analyze the propagation of errors after fault injection in order to achieve further improvement in the fault activation.

It should be noted that this study was limited to the cases presented in Section 4.4 and that the instruction-set was not thoroughly examined to find all bit-flip transitive instructions. There is, therefore, room for even more improvement.

6.5. Discussion

An important observation that can be made from the results is that error detection coverage is much lower for memory faults than for register faults (30% in memory against 80% in registers). Since the exception distribution in the memory fault injection campaigns gives a much larger weight to the Floating Point Assist Exception (generic exception for floating point errors) a possible explanation is that memory is often used to store floats and that these display lower error detection coverage. At this point it isn't possible to elaborate on this possibility since only the 32 least significant bits of floating point registers can be targeted but, when stored in memory, all 64 bits are available.

An interesting observation that can be made from both workloads is that error detection coverage and exception distribution is similar in random and optimized pairs of campaigns. The error detection coverage is quite different when the workload is changed (quicksort and controller) and also when the targeted resources are changed (registers and memory) but remains similar when moving from random to optimized fault injection. The exception distribution is similar, but not identical, for the optimized and the random campaigns. The rather strong similarity shown in error detection coverage and exception distribution between the optimized and the random campaigns is an important finding. Further investigations are, however, needed to confirm that this finding is valid also for other workloads and target systems, and to explain the reasons for this behavior.

7. CONCLUSIONS

A pre-injection analysis technique was devised and developed for the GOOFI fault injection tool. It was applied to Nexus-based fault injection on a computer system based on the Motorola MPC565 processor. Using assembly-level knowledge of the processor - an implementation of the PowerPC architecture - it is possible to eliminate faults with no possibility of activation and determine equivalence classes among faults in order to reduce the fault-set.

The method and this particular implementation were studied and the experimental results of optimized fault injection were compared to the usual random approach. Two distinct applications were used as workloads for this study – a simple quicksort implementation and a more complex jet engine controller. The effectiveness of faults was increased one order of magnitude compared to random fault-injection. The fault-set was reduced two orders of magnitude for the registers and four to five orders of magnitude for the memory.

The optimization is exhaustive in the sense that all the assembly instructions used by both workloads are covered. The limitation is that not all registers in the processor can be studied through the analysis of the code. Some registers have an "asynchronous" behavior during the execution of the application code. These can be registers used by the processor as Control Registers for communication ports (Serial, CAN, etc.). There are also many Time Processor Unit (TPU) registers. The TPUs are separate modules inside the processor and fault injection in these has no effect on the applications tested. The registers targeted consist of the User Instruction Set Architecture register set and these are likely to correspond to most of the interesting resources to study through fault injection. Nonetheless, an exhaustive fault injection campaign would require fault injection also in the locations not targeted by the optimization. The future work section presents a possible way to deal with this problem.

Even though activation of faults is ensured by the proposed method (activation in the sense that the faulty resources are always utilized) not all faults result in effective errors. For the quicksort program, 52% of the faults targeting registers were still non-effective. For the jet engine controller, 62% of the faults in registers and 82% of the faults in memory remained non-effective. An increase of one order of magnitude in the fault effectiveness is attained but there are still many faults that are activated but produce no effect. This occurs when the data is used in a non-sensitive way by the code (regarding the single bit-flip model). If, for instance, a fault is injected in a register used for a comparison instruction only some of the bits in this register will actually affect the resulting flags in the Condition Register. An interesting topic for further studies would be to investigate which activated faults are non-effective and why.

One interesting finding is that the estimated error detection coverage estimation is similar in random and optimized campaigns, but quite different depending on the workload and targeted resources. All faults targeting the same bit of a given resource, before this resource is read, are considered equivalent. Since only one representative of these is injected it would be necessary to apply a weight corresponding to the number of faults in this equivalence class to obtain an accurate estimation of error detection coverage. The finding that there is actually almost no difference in the estimations might be explained by arguing that the number of experiments is still low when compared to the size of the fault-set. Another possible explanation is that resources are used by programs in a way that the number of collapsed faults in each equivalence class is not statistically correlated to the effects of the faults.

As for the exception distribution of detected errors, it is clearly different when targeting different locations (registers and memory). This was expected, since memory and registers perform different roles during the execution of a program.

An interesting finding is that the exception distribution is similar on the two workloads evaluated. This is interesting since previous research indicates a strong relationship between workload and effects of faults.

Comparing the exception distribution of detected errors in random and optimized campaigns shows that they quite similar. This is a finding similar to the error detection coverage estimation described previously.

It is possible to conclude that the outcomes of faults are highly dependent on the resource targeted. Some registers have a greater tendency to cause wrong output while some others cause detected errors more frequently. Using this finding (shown in Figures 15 to 19) would allow, for instance, performing a greater number of experiments resulting in wrong output (by targeting the resources that are more likely to observe this behavior).

A first approach at finding equivalencies among faults after propagation yielded a reduction of 10% of the already optimized fault-set. It is possible to identify instructions which propagate single bit-flips in the operands as single bit-flips in the results. These bit-flips may also be considered as an equivalence class and only one representative needs to be injected.

8. FUTURE WORK

Expanding the optimization to achieve further improvement by analyzing the propagation of bit-flips has been started in this thesis and yielded good results. It would be interesting to continue this study and see how much further is it possible to improve the fault effectiveness. This would eventually mean, for instance, that a multiple-bit flip fault should be injected to avoid the injection of several single bit-flips.

The study presented in this thesis is limited to a single target system. This implies that the results obtained are specially associated to the MPC565. The results of fault injection experiments are always strongly attached to the system under evaluation and even to the compiler [12] used. Thus, it would be interesting to implement the optimization method presented here to other microprocessors and target systems. The present implementation (OFFSET) can easily be adapted to other target systems.

In order to achieve results with the statistical properties of random fault injection it would be necessary to weight the outcome of each experiment with the number of faults in the equivalence class. Even though the results show a similarity between the results obtained by the random and optimized fault-sets, it is possible to perform optimized fault injection and still obtain results that are representative of a uniform distribution of faults.

The optimization method has not been implemented for injection into the code segment. However, the code segment was not targeted by any campaign in this thesis, since it is usually stored in ROM. The optimization method would be easily expanded by placing faults in the memory addressed by the Program Counter (adding support for MEM32[PC]).

In the future of fault injection the multiple bit-flip fault model will become more important. The processor technology is employing smaller transistors, with lower power voltages, where a single charged particle is likely to change the state of several bits. It would be appealing to extend the method presented in this thesis to improve the selection of multiple bit-flip faults.

The purpose of a fault injection campaign is sometimes to study faults that produce wrong output (the most significant faults). Correlating the locations for fault injection with the probability of producing wrong output, in a dynamic manner, would yield faster campaigns. The same could be applied to the study of effective faults (detected or wrong output). The distribution of faults would be further away from uniform but would allow a much smaller time to obtain a certain amount of results.

Using a debugger/debugging environment that supports tracing of all read/write operations on all registers during the golden-run would allow the expansion of the optimization to all registers.

During the experiments performed for this thesis there were many faults which were activated, in the sense that the targeted resource was used, but their outcome was non-effective. A manual inspection in order to determine why activated faults have no effect would be interesting.

9. REFERENCES

[1] H. Madeira and J. Silva, “Experimental Evaluation of the Fail-Silent Behavior in Computers Without Error Masking”, Proc. FTCS-24, June 1994, pp. 350-359.

[2] P. Yuste, J. Ruiz, et al., “Non-Intrusive Software-Implemented Fault Injection in Embedded Systems”, LADC 2003, LNCS 2847, 2003, pp. 23-38.

[3] J. Aidemark, P. Folkesson, and J. Karlsson, “Path-Based Error Coverage Prediction”, JETTA, Vol. 16, June 2002.

[4] J. Aidemark, J. Vinter, et al., “GOOFI: Generic Object-Oriented Fault Injection Tool”, Proc. DSN 2001, July 2001, pp. 83-88.

[5] IEEE-ISTO, "The Nexus 5001 Forum™ Standard for a Global Embedded Processor Debug Interface", 1999.

[6] Motorola Inc., “MPC565/MPC566 User’s Manual”, 2nd edition, 2003.

[7] X. Castillo and D. Siewiorek, “Workload, Performance and Reliability of Digital Computer Systems”, Proc. FTCS-11, June 1981, pp. 84-89.

[8] E. Czeck and D. Siewiorek, “Observations on the Effects of Fault Manifestation as a Function of Workload”, IEEE Transactions on Computers, Vol. 41, No. 5, May 1992, pp.559-566.

[9] R. Chillarege and R. Iyer, “The Effect of System Workload on Error Latency: An Experimental Study”, Proc. ACM SIGMETRICS 1985, August 1985, pp. 69-77.

[10] R. Chillarege and N. Bowen, “Understanding Large System Failures – A Fault Injection Experiment”, Proc. FTCS-19, June 1989, pp. 356-363.

[11] A. Avizienis, J. Laprie and B. Randell, “Fundamental Concepts of Dependability”, Technical Report 01-145, LAAS, April 2001.

[12] J. Güthoff and V. Sieh, “Combining Software-Implemented and Simulation-Based Fault Injection Into a Single Fault Injection Method”, Proc. FTCS-25, June 1995, pp. 196-206.

[13] A. Benso, M. Rebaudengo, et al., ”Fault-List Collapsing for Fault Injection Experiments”, RAMS 98, January 1998, pp. 383-388.

[14] T. Tsai, S. Upadhyaya, et al., ”Stress-Based and Path-Based Fault Injection”, IEEE Transactions on Computers, Vol. 48, No. 11, November 1999, pp. 1183-1201.

[15] L. Berrojo, I. González, et al., “New Techniques for Speeding-up Fault-injection Campaigns”, Proc. DATE 2002, March 2002, pp. 847-852.

[16] PHYTEC Technology Holding Company, http://www.phytec.com/sbc/32bit/pc565.htm, July 8th, 2004.

[17] iSYSTEM AG, http://www.isystem.com, July 8th, 2004.

[18] iC3000 Active Emulator, http://www.isystem.com/Products/Emulators/iC3000/ic3000_overview.htm, July 8th, 2004.

[19] Motorola Inc., “RISC Central Processing Unit Reference Manual”, revision 1, 1999.

APPENDIX A – POWERPC ASSEMBLY-LEVEL DEFINITIONS (MPC565)

The following table presents the definitions used to analyze the instruction-set of the MPC565 microprocessor. All the instructions considered are defined by the PowerPC User Instruction Set Architecture. This information was obtained from Motorola's RISC CPU Reference Manual [19]. These definitions cover the instructions executed by the two workloads used in this thesis.

Opcode	Operands	Read List	Write List	Transitive
ADD	rD,rA,rB	**rA,rB**	rD	
ADDI	rD,rA,SIMM	**rA**	rD	
ADDIC	rD,rA,SIMM	**rA**	rD	
ADDZE	rD,rA	**rA,XER[2]**	rD	
AND	rA,rS,rB	**rS,rB**	rA	Y,Y
AND.	rA,rS,rB	**rS,rB**	rA,CR0	
ANDC	rA,rS,rB	**rS,rB**	rA	Y,Y
ANDC.	rA,rS,rB	**rS,rB**	rA,CR0	
B	target			
BEQ-	target	**CR[2]**		
BEQ+	target	**CR[2]**		
BGE-	target	**CR[0]**		
BGE+	target	**CR[0]**		
BGT-	target	**CR[1]**		
BGT+	target	**CR[1]**		
BL	target		LR	
BLE-	target	**CR[1]**		
BLE+	target	**CR[1]**		
BLR		**LR**		
BLT-	target	**CR[0]**		
BLT+	target	**CR[0]**		
BNE-	target	**CR[2]**		
BNE+	target	**CR[2]**		
CLRLWI	rA,rS,n	**rS**	rA	
CLRLWI.	rA,rS,n	**rS**	rA,CR0	
CMPLWI	rA,value	**rA,XER[0]**	CR0	
CMPW	rA,rB	**rA,rB,XER[0]**	CR0	
CMPWI	rA,value	**rA,XER[0]**	CR0	
CROR	crbD,crbA	**CR**	CR	
CROR	crbD,crbA,crbB	**CR**	CR	
EXTSH	rA,rS	**rS[16:31]**		
EXTSH.	rA,rS	**rS[16:31]**	CR0	
FABS	frD,frB	**frB**	frD	
FADD	frD,frA,frB	**frA,frB**	frD	
FADDS	frD,frA,frB	**frA,frB**	frD	
FCMPU	crfD,frA,frB	**frA,frB**	crfD	
FCTIWZ	frD,frB	**frB**	frD	

FDIV	frD,frA,frB	**frA,frB**	frD	
FDIVS	frD,frA,frB	**frA,frB**	frD	
FMADD	frD,frA,frC,frB	**frA,frC,frB**	frD	
FMADDS	frD,frA,frC,frB	**frA,frC,frB**	frD	
FMR	frD,frB	**frB**	frD	
FMSUB	frD,frA,frC,frB	**frA,frC,frB**	frD	
FMUL	frD,frA,frC	**frA,frC**	frD	
FMULS	frD,frA,frC	**frA,frC**	frD	
FNEG	frD,frB	**frB**	frD	
FRSP	frD,frB	**frB**	frD	
FSUB	frD,frA,frB	**frA,frB**	frD	
FSUBS	frD,frA,frB	**frA,frB**	frD	
ISYNC				
LBZ	rD,d(rA)	**rA,MEM08(d+rA)**	rD	
LBZU	rD,d(rA)	**rA,MEM08(d+rA)**	rD,rA	
LFD	frD,d(rA)	**rA,MEM64(d+rA)**	frD	
LFDU	frD,d(rA)	**rA,MEM64(d+rA)**	frD,rA	
LFDX	frD,rA,rB	**rA,rB,MEM64(rA+rB)**	frD	
LFS	frD,d(rA)	**rA,MEM32(d+rA)**	frD	
LFSX	frD,rA,rB	**rA,rB,MEM32(rA+rB)**	frD	
LHA	rD,d(rA)	**rA,MEM16(d+rA)**	rD	
LHAU	rD,d(rA)	**rA,MEM16(d+rA)**	rD,rA	
LHZ	rD,d(rA)	**rA,MEM16(d+rA)**	rD	
LI	rA,value		rA	
LIS	rA,value		rA	
LMW	rD,d(rA)	**rA,rD,MEMXX(d+rA:32-rD)**	GPR[rD]	N,N,Y
LWZ	rD,d(rA)	**rA,MEM32(d+rA)**	rD	N,Y
LWZU	rD,d(rA)	**rA,MEM32(d+rA)**	rD,rA	
LWZX	rD,rA,rB	**rA,rB,MEM32(rA+rB)**	rD	
MFCR	rD	**CR**	rD	
MFLR	rD	**LR**	rD	
MFMSR	rD		rD	
MR	rA,rS	**rS**	rA	
MTLR	rA	**rA**	LR	
MTMSR	rS	**rS**	MSR	
MULLI	rD,rA,SIMM	**rA**	rD	
MULLW	rD,rA,rB	**rA,rB**	rD	
NEG	rD,rA	**rA**	rD	
OR	rA,rS,rB	**rS,rB**	rA	Y,Y
OR.	rA,rS,rB	**rS,rB**	rA,CR0	
ORI	rA,rS,UIMM	**rS**	rA	Y
RLWINM	rA,rS,SH,MB,ME	**rS**	rA	
RLWINM.	rA,rS,SH,MB,ME	**rS**	rA,CR0	
SLW	rA,rS,rB	**rS,rB[26:31]**	rA	
SRAW	rA,rS,rB	**rS,rB**	rA	
SRAWI	rA,rS,SH	**rS**	rA	
STB	rS,d(rA)	**rA,rS[24:31]**		

STBU	rS,d(rA)	**rA,rS[24:31]**	rA	
STFD	frS,d(rA)	**rA,frS**		
STFDX	frS,rA,rB	**rA,rB,frS**		
STFS	frS,d(rA)	**rA,frS**		
STH	rS,d(rA)	**rA,rS[16:31]**		
STHU	rS,d(rA)	**rA,rS[16:31]**	rA	
STMW	rS,d(rA)	**rA,rS,GPR[rS]**	MEMXX(d+rA:32-rS)	N,N,Y
STW	rS,d(rA)	**rA,rS**	MEM32(d+rA)	N,Y
STWU	rS,d(rA)	**rA,rS**	rA,MEM32(d+rA)	
SUBF	rD,rA,rB	**rA,rB**	rD	
SUBF.	rD,rA,rB	**rA,rB**	rD,CR0	
SUBFIC	rD,rA,SIMM	**rA**	rD	
SYNC				
XORI	rA,rS,UIMM	**rS**	rA	
XORIS	rA,rS,UIMM	**rS**	rA	

Table 10 - Assembly-level definitions for the MPC565 (PowerPC)

APPENDIX B – MPC565'S SUPERVISOR-LEVEL SPECIAL PURPOSE REGISTERS

SPR Number (Decimal)	Special-Purpose Register
18	DAE/Source Instruction Service Register (DSISR)
19	Data Address Register (DAR)
22	Decrementer Register (DEC)
26	Save and Restore Register 0 (SRR0)
27	Save and Restore Register 1 (SRR1)
80	External Interrupt Enable (EIE)
81	External Interrupt Disable (EID)
82	Non-Recoverable Interrupt (NRI)
272	SPR General 0 (SPRG0)
273	SPR General 1 (SPRG1)
274	SPR General 2 (SPRG2)
275	SPR General 3 (SPRG3)
284	Time Base Lower – Write (TBL)
285	Time Base Upper – Write (TBU)
287	Processor Version Register (PVR)
528	IMPU Global Region Attribute (MI_GRA)
536	L2U Region Attribute (L2U_GRA)
560	BBC Module Configuration Register (BBC_MCR)
568	L2U Module Configuration Register (L2U_MCR)
784	L2U Region Base Address Register 0 (L2U_RBA0)
785	IMPU Region Base Address Register 1 (MI_RBA1)
786	IMPU Region Base Address Register 2 (MI_RBA2)
787	IMPU Region Base Address Register 3 (MI_RBA3)
816	IMPU Region Attribute Register 0 (MI_RA0)
817	IMPU Region Attribute Register 1 (MI_RA1)
818	IMPU Region Attribute Register 2 (MI_RA2)
819	IMPU Region Attribute Register 3 (MI_RA3)
792	L2U Region Base Address Register 0 (L2U_RBA0)
793	L2U Region Base Address Register 1 (L2U_RBA1)
794	L2U Region Base Address Register 2 (L2U_RBA2)
795	L2U Region Base Address Register 3 (L2U_RBA3)
824	L2U Region Attribute Register 0 (L2U_RA0)
825	L2U Region Attribute Register 1 (L2U_RA1)
826	L2U Region Attribute Register 2 (L2U_RA2)
827	L2U Region Attribute Register 3 (L2U_RA3)
1022	Floating-Point Exception Cause Register (FPECR)

Table 11 - Supervisor-Level Special Purpose Registers

APPENDIX C – MPC565'S DEVELOPMENT SUPPORT SPECIAL PURPOSE REGISTERS

SPR Number (Decimal)	Special-Purpose Register
144	Comparator A Value Register (CMPA)
145	Comparator B Value Register (CMPB)
146	Comparator C Value Register (CMPC)
147	Comparator D Value Register (CMPD)
148	Exception Cause Register (ECR)
149	Debug Enable Register (DER)
150	Breakpoint Counter A Value and Control (COUNTA)
151	Breakpoint Counter B Value and Control (COUNTB)
152	Comparator E Value Register (CMPE)
153	Comparator F Value Register (CMPF)
154	Comparator G Value Register (CMPG)
155	Comparator H Value Register (CMPH)
156	L-bus Support Comparators Control 1 (LCTRL1)
157	L-bus Support Comparators Control 2 (LCTRL2)
158	I-bus Support Control Register (ICTRL)
159	Breakpoint Address Register (BAR)
630	Development Port Data Register (DPDR)

Table 12 - Development Support Special Purpose Registers

APPENDIX D – MPC565'S EXCEPTIONS

Mneumonic	Name	Causes
RST	System Reset Exception	Any reset pin is asserted – PORESET, HRESET, or SRESET. An internal reset is requested, such as from the software watchdog timer.
CHSTP	Checkstop	The system under study was configured to enter the Checkstop State when a machine-check exception (MCE) occurs.
MCE	Machine Check Exception	The accessed address does not exist. A data error was detected. A storage protection Violation was detected by chip-select logic.
EXTI	External Interrupt	The external interrupt exception is taken on assertion of the internal IRQ line to the RCPU.
ALE	Alignment Exception	The operand of a floating-point load or store instruction is not word-aligned. The operand of a load or store multiple instruction is not word-aligned. The operand of lwarx or stwcx. is not word-aligned.
PRE	Program Exception	A program exception occurs for several exception conditions and no higher priority exception exists.
FPUVE	Floating-Point Unavailable Exception	A floating-point unavailable exception occurs when no higher priority exception exists, an attempt is made to execute a floating-point instruction (including floating-point load, store, and move instructions), and the floating-point available bit in the MSR is disabled.
DECE	Decrementer Exception	A decrementer exception occurs when no higher priority exception exists, the decrementer register has completed decrementing, and MSR[EE] = 1.
SYSE	System Call Exception	A system call exception occurs when a system call instruction is executed.
TR	Trace Exception	A trace interrupt occurs if MSR[SE] = 1 and any instruction except rfi is successfully completed or MSR[BE] = 1 and a branch is completed.
FPASE	Floating-Point Assist Exception	A floating-point enabled exception condition is detected. The corresponding floating-point enable bit in the FPSCR (floating point status and control register) is set (exception enabled). MSR[FE0] | MSR[FE1] = 1.
SEE	Software Emulation Exception	When executing any non-implemented instruction. When executing a mtspr or mfspr instruction that specifies an un-implemented internal-to-the-processor SPR.
ITLBER	Instruction Protection Exception	The fetch access violates storage protection and MSR[IR] = 1. The fetch access is to guarded storage and MSR[IR] = 1.
DTLBER	Data Protection Error Exception	The data access violates the storage protection and MSR[DR]=1.

LBRK	Implementation-Dependent Debug Exceptions	Data breakpoint match.
IBRK	Implementation-Dependent Debug Exceptions	Instruction breakpoint match.
EBRK	Implementation-Dependent Debug Exceptions	Development port maskable request or a peripheral breakpoint.
DPI	Implementation-Dependent Debug Exceptions	Development port non-maskable request.

Table 13 - Exceptions triggered in the MPC565