Assembly-Level Pre-Injection Analysis for Improving Fault Injection Efficiency

Raul Barbosa, Jonny Vinter, Peter Folkesson, and Johan Karlsson

Department of Computer Engineering,
Chalmers University of Technology,
S-412 96 Göteborg, Sweden
{rbarbosa, vinter, peterf, johan}@ce.chalmers.se

Abstract. This paper describes a fully automated pre-injection analysis technique aimed at reducing the cost of fault injection campaigns. The technique optimizes the fault-space by utilizing assembly-level knowledge of the target system in order to place single bit-flips in registers and memory locations only immediately before these are read by the executed instructions. This way, faults (time-location pairs) that are overwritten or have identical impact on program execution are removed. Experimental results obtained by random sampling of the optimized fault-space and the complete (non-optimized) fault-space are compared for two different workloads running on a MPC565 microcontroller. The pre-injection analysis yields an increase of one order of magnitude in the effectiveness of faults, a reduction of the fault-space of two orders of magnitude in the case of CPU-registers and four to five orders of magnitude in the case of memory locations, while preserving a similar estimation of the error detection coverage.

1 Introduction

Computer systems are increasingly being used in safety-critical applications such as aerospace or vehicular systems. To achieve the high safety integrity levels 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.

In order to validate the correctness and efficiency of their fault tolerance features, safety-critical systems must be thoroughly tested. 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 commonly observed 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-space, 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 and similar problems and to 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 dependability measures using the results of fault injection 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 injected, before any experiment is performed mean.

This paper 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 is often used in fault injection experiments to emulate the effects of single event upsets and other transient disturbances usual.

The objective of the pre-injection analysis is to optimize[1] the fault-space from which the injected faults are sampled. The analysis uses program execution information to (i) eliminate faults that have no possibility of activation and (ii) find equivalence classes among faults and insert only one of these into the optimized fault-space. This is achieved by applying the following rule: *faults should only be placed in resources immediately before these are read by each instruction*. A bit-flip in any resource[2] 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. It should be noted that collapsing all faults in a given class into a single fault in the optimized fault-space may cause a bias in the estimated dependability measures (e.g. error detection coverage). One of the objectives of this research is therefore to investigate the magnitude of this bias.

The pre-injection analysis technique was implemented in the GOOFI (Generic Object-Oriented Fault Injection) [4] tool, for Nexus-based fault injection [2, 5, 6], and is also suitable for implementation in other platforms. The effectiveness of the technique was assessed by comparing fault injection results with results obtained by non-optimized fault injection on the same target system. The system is based on the Motorola MPC565 [7] – a microcontroller aimed at the automotive and other control-intensive applications based on the PowerPC architecture. 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 the optimized fault-space, which is stored in a database. The fault injection experiments are then conducted by random sampling of faults from the optimized fault-space.

[1] The word *optimize* should not suggest that the optimal fault-space is found but rather an improvement on the usual random approach. Further optimization is therefore achievable.

[2] In this paper we use the word *resource* as a common term for CPU-register, main memory locations and otherstate-elements where bit-flips may occur.

2 Related Research

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] take, in their experiments, special care to avoid placing faults in empty (i.e. not used) memory regions. They obtained 12% of effective faults and pointed out that a random sampling from an unrestricted fault-space consisting of all possible fault locations (bits) and all time points 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 dynamical usage of resources along the time dimension.

Studies conducted in the past have shown that error manifestation (rate and effects) is affected by workload [8, 9, 10]. In [11] the concept of failure acceleration was introduced by R. Chillarege and N. Bowen. They 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.

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 should be proportional to its utilization. Register utilization is defined as the measure of the probability that an injected fault manifests itself as an error. 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.

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.

In [14] T. Tsai et al. introduced a technique named path-based injection. With this technique a fault is injected into a resource that will be used by the test program, given a particular input set. After the manual derivation of the input sets, the path of execution is described in terms of a list of executed basic blocks. For each path, faults are only injected in the utilized resources.

Working in fault injection for the test of fault-tolerant circuits, using VHDL models, a set of techniques for speeding up campaigns is described by L. Berrojo et al. in [15]. One of these techniques is workload dependent fault collapsing. During the reference run (a fault-free execution in order to store the program's normal behavior) 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, at the circuit level, all possible bit-flips are then collapsed by (i) marking as silent all bit-flips between an operation (either read or write) and a write operation, and (ii) marking as equivalent all bit-flips between an operation (either read or write) and the subsequent read operation.

J. Arlat et al. [16] increased the efficiency of their fault injection experiments targeting the code segment by logging the control flow activated by the workload processes. If the randomly selected address for fault injection is not part of the log (instruction trace), then the corresponding experiment can simply be skipped (as the outcome is already known).

3 Fault-space Optimization Method

For single bit-flip fault injection, we define a fault-space to be a set of time-location pairs that determines where and when the bit-flip is injected. The time is selected from an interval during the execution of the workload selected for the experiment. The time granularity is based on the execution of machine instructions, i.e. bit-flips can only be injected between the execution of two machine instructions. The complete (non-optimized) fault-space consists of all possible time-location pairs.

The fault-space optimization method presented in this paper states that faults should only be placed in a resource immediately before the resource is read by an instruction. The following sections describe the input needed for the analysis, the output created and the optimization procedure.

3.1 Optimization Input

In order to determine the optimized fault-space it is necessary to gather information about the code of the application and the computer system executing it:

- Assembly code of the application;
- The Program Counter (PC) trace over time;
- The effective address of each memory read access;
- The definition of which resources are read by each assembly instruction.

In our experimental setup, the assembly code is textual information obtained by disassembling the executable binaries of the application, processed automatically by the optimization program. The Program Counter trace and the values of the General Purpose Registers are stored during the execution of the reference run. The effective address of each memory read access is calculated with these values. The definitions of which resources are read by each assembly instruction are built into the optimization program. These were obtained from Motorola's RISC CPU Reference Manual [17] and are available in [18].

3.2 Optimization Output

The resulting output (the optimized fault-space) consists of a list of possible locations and times for fault injection. The optimization procedure has been adapted to both one-shot applications and control applications executing in loops. Each element on the optimized fault-space contains the following information:

- Control loop index;
- Breakpoint address;
- Number of breakpoint invocations within the control loop;
- The fault injection location.

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 that do not execute in loops, the control loop index is always set to one. The **breakpoint address** specifies the breakpoint position inside the control loop and the **number of breakpoint invocations** specifies the number of times this breakpoint should be reached before fault injection.

3.3 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 1.

```
FOREACH pc_value IN program_counter_trace DO
    control_loop_index ← current_control_loop ()
    breakpoint_invocation ← breakpoint_invocations_count (pc_value)
    instruction ← instruction_at_code_address (pc_value)
    instruction_read_list ← resources_read_by_instruction (instruction)
    FOREACH resource IN instruction_read_list DO
        useful_fault ← [control_loop_index, pc_value, breakpoint_invocation, resource]
        store_in_database (useful_fault)
    ENDFOREACH
ENDFOREACH
```

Fig. 1. Pseudo-code for the optimization procedure

The most important stage (shown in bold in the pseudo-code) is the identification of the resources read by each instruction. To accomplish this, the first step is to find the definition on the list matching the given instruction. This is done by matching the *opcode* and the *operands*. Then, by examining 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 2 illustrates this process.

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, 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 index and the breakpoint invocation are assumed to hold the specified values).

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+24, as being read. The value 1000 of R6 is obtained during the reference run. The two resources along with the timings are then inserted into the fault-space.

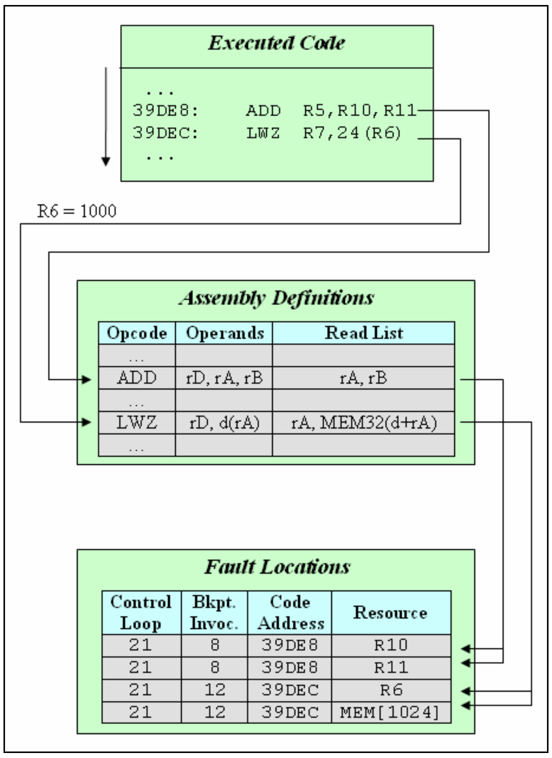

Fig. 2. Example of the optimization procedure

4 Experimental Setup

Figure 3 describes the evaluation platform used to evaluate the effectiveness of the optimization technique for experiments performed on the jet engine control software, which is one of two workloads investigated in this paper. The GOOFI fault injection

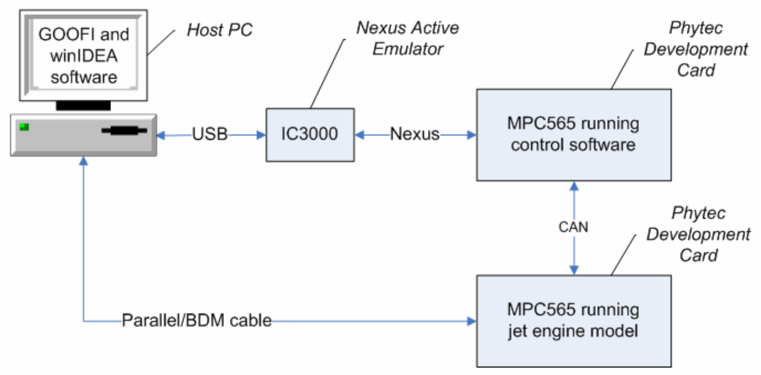

Fig. 3. Evaluation platform for the jet engine application

tool controls the experiments by using the winIDEA debugging environment [19] in conjunction with the iC3000 debugger. Faults are injected into the MPC565 microcontroller running the control software. In the case of the jet engine controller one computer board was used to run the jet engine control software and one board to execute the model of the jet engine. The experimental setup used for the other workload (an implementation of the quicksort algorithm) used only one computer board.

4.1 Fault Injection Tool

GOOFI is a fault injection tool developed at the Department of Computer Engineering, Chalmers University of Technology. It provides the ability to define and conduct fault injection campaigns on a variety of microprocessors. During each campaign GOOFI is responsible for controlling all the necessary software and hardware, and storing the acquired data into a database.

A plug-in [6] has recently 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 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 current implementation is therefore the MPC565 microcontroller. The technique may however be implemented for any microprocessor.

4.2 MPC565 Microcontroller

The MPC565 is a microcontroller 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 [20] development board. It includes a 32-bit Motorola MPC565 processor, which offers a Nexus debug port enabling real-time trace of program and data flow.

To establish a connection through this port the iSYSTEM iC3000 Active Emulator [21, 22] was used to access the Nexus working environment. The iC3000 emulator was, in its turn, controlled by GOOFI via winIDEA – an integrated development environment offered by iSYSTEM. GOOFI and winIDEA are executing on the same host PC.

4.3 Workloads

Fault injection campaigns were conducted to evaluate the optimization technique using two different workloads: a sort program using the quicksort algorithm and a jet engine controller. Different campaigns targeting registers and data memory, using both optimized and non-optimized fault selection, were carried out. The technique is fully implemented in the sense that all the assembly instructions executed by the workloads are analysed and all registers and data memory locations where optimization is achievable with this method are considered. The outcome of each fault injection experiment was classified into one of the following categories:

- **Detected Error** – All effective errors that are signalled 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.

4.3.1 Quicksort

The quicksort workload is a recursive implementation of the well-known sorting algorithm. It sorts an array containing seven double-precision floats.

The reference run execution takes two minutes during which the processor is being stepped and all the required data is obtained. The optimization procedure takes 20 seconds to complete. Each fault injection experiment takes less than half a minute to perform. During the execution of the reference run for this application, the MPC565 processor executed 34 distinct assembly instructions (opcodes) and a total of 815 instructions.

4.3.2 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 (sensor values from the engine and actuator commands from the controller). It is significantly more complex than the quicksort program, allowing the fault-space optimization technique to be evaluated using a real-world application.

The execution of the reference run takes almost 12 hours. The optimization procedure takes 10 minutes to complete. 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).

Forty control loops of execution were logged during each experiment. From these, ten loops (21 to 30) were chosen as possible temporal locations for fault injection (corresponding to 50ms of real-time execution of the controller). During these ten control loops, in the reference run, the MPC565 processor executed 231.097 instructions. A total of 88 different assembly instructions (opcodes) were executed.

4.4 Fault Model and Fault Selection

The fault model applied is the single bit-flip model of the effects of transient faults. The technique assumes this model as the basis for optimization.

The faults in the non-optimized campaigns were chosen using a uniform distribution. In the case of the optimized campaigns the faults are selected randomly from the optimized fault-space itself (the list of temporal and spatial locations for fault injection described in Section 3.2). This implies that the distribution of faults in resources is proportional to the representation of each resource in the optimized fault-space.

Microprocessor 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 non-optimized campaigns are the ones considered by the optimization method and shown in Table 1.

Table 1. Registers targeted for optimization

General Purpose Registers (32 registers of 32 bits)	Condition Register (32 bits)
Floating Point Registers (32 registers of 64 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. User-level instructions are limited to this register set while supervisor-level instructions have access to other, special purpose registers (SPRs).

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), targeted by the optimization, is also not available for fault injection.

The fault injection campaigns in memory targeted the stack, heap and all other read/write and read-only data segments of the controller. A total of 100KB of memory were targeted as spatial locations.

The analysis of faults in the code segment was still not implemented and was therefore not studied. The optimization is easily extendable to support faults in the code segment by targeting, in each instruction, the 32-bit memory contents addressed by the Program Counter. This would be equivalent to the analysis performed in [16] by using the instruction trace.

5 Experimental Results

5.1 Fault Injection in Registers

Table 2 shows the distribution of the outcomes of faults in the fault injection campaigns targeting microprocessor registers for both the quicksort and the jet engine controller workloads. The quicksort campaigns include approximately the same number of experiments. For the non-optimized jet engine controller campaign, a much higher number of experiments had to be performed in order to increase the confidence in the results.

Table 2. Distribution of outcomes of fault injection in registers

Campaign		*# Exp.*	*Non-effective*	*Detected*	*Wrong Output*
Quicksort	*Random*	2739	2603 (95.0%)	83 (3.0%)	53 (2.0%)
	Optimized	2791	1461 (52.3%)	744 (26.7%)	586 (21.0%)
Jet Engine Controller	*Random*	5708	5457 (95.6%)	200 (3.5%)	51 (0.9%)
	Optimized	1559	964 (61.8%)	466 (29.9%)	129 (8.3%)

The percentage of *effective* faults (*detected* or *wrong output*) increases from 5.0% using non-optimized fault selection to 47.7% choosing faults from the optimized fault-space when targeting the quicksort workload. In the jet engine controller this increase is from 4.4% to 38.2%. The improvement in the effectiveness of faults is, therefore, one order of magnitude.

Table 3 shows the estimated error detection coverage obtained in each campaign. We here define error detection coverage as the quotient between the number of detected and the number of effective faults.

Table 3. Error detection coverage estimations (registers)

Campaign		*Estimated error detection coverage (95% confidence)*
Quicksort	*Random*	61.0 ± 8.2%
	Optimized	55.9 ± 2.7%
Jet Engine Controller	*Random*	79.7 ± 5.0%
	Optimized	78.3 ± 3.3%

The values of the error detection coverage estimations are quite similar whether applying non-optimized or optimized fault selection. 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 detected errors were signalled by the exceptions provided in the MPC565 processor. The distribution among these exceptions is presented in Figures 4 and 5 for the quicksort campaigns, and in Figures 6 and 7 for the jet engine controller campaigns.

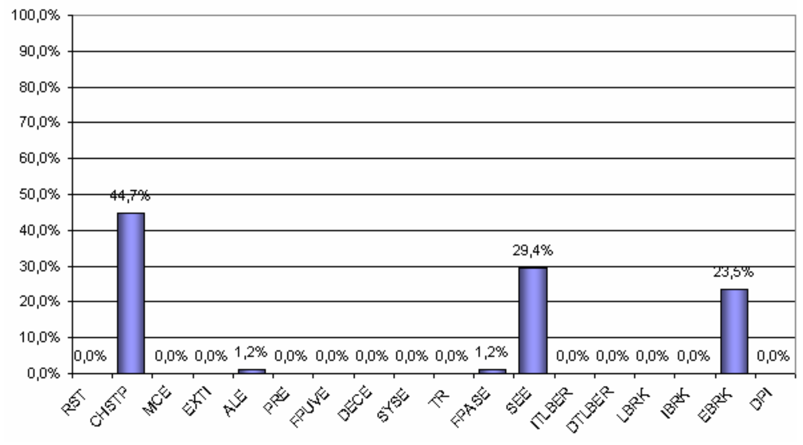

Fig. 4. Exceptions in the quicksort non-optimized campaign (83 faults in registers)

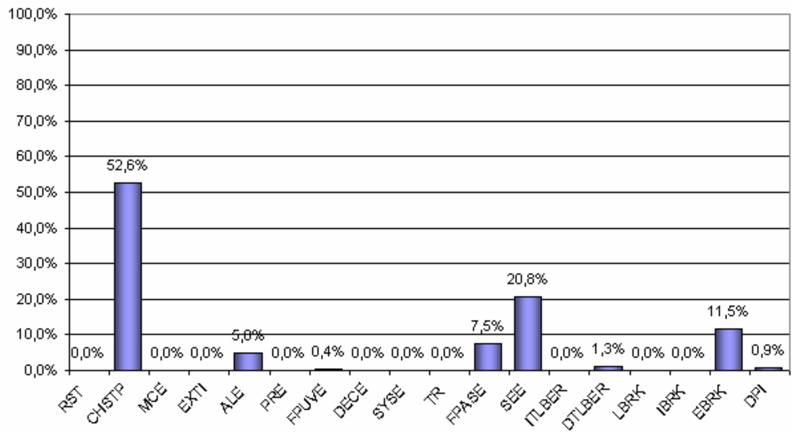

Fig. 5. Exceptions in the quicksort optimized campaign (744 faults in registers)

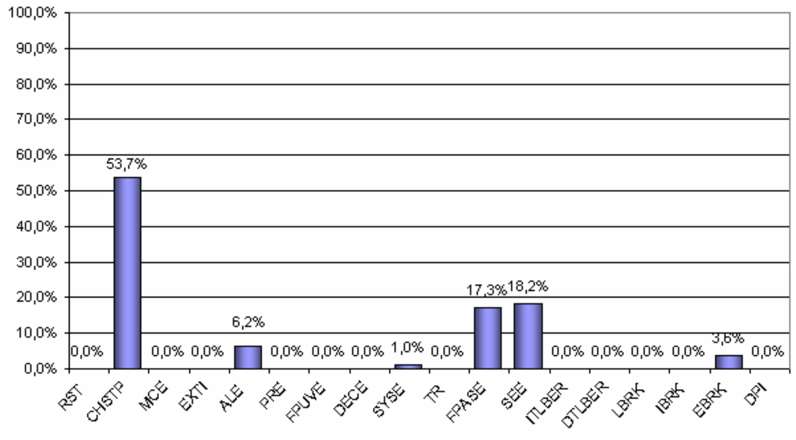

Fig. 6. Exceptions in the jet engine controller non-optimized campaign (200 faults in registers)

It is possible to observe that the detection mechanisms are activated in a similar but not identical way for the non-optimized and the optimized campaigns. Figures 4 to 7 provide an insight on the magnitude of the differences between non-optimized and optimized fault selection. A brief description follows of the most frequently activated exceptions.

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, for instance, when the accessed memory address does not exist.

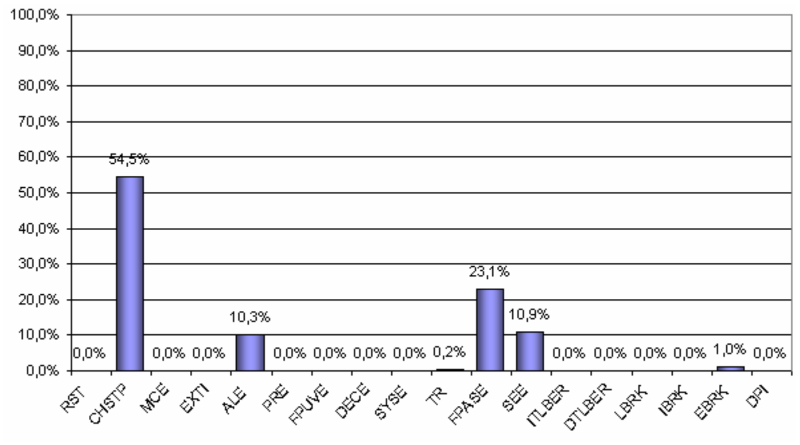

Fig. 7. Exceptions in the jet engine controller optimized campaign (466 faults in registers)

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 Floating Point Status And Control Register (FPSCR) is set (exception enabled);
- 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 Special Puspose Register (SPR).

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

Figure 8 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.

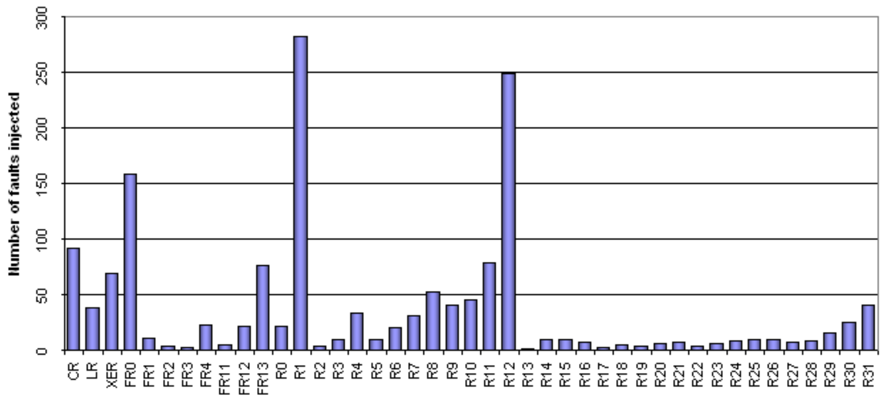

Fig. 8. Number of faults injected per register

5.2 Fault Injection in Memory

Fault injection in memory locations was performed only for the jet engine controller. Table 4 shows the distribution of the outcomes of faults for both non-optimized and optimized fault selection.

Table 4. Distribution of outcomes of fault injection in memory

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

The effectiveness of faults increases from 2.0% using non-optimized fault selection to 19.1% choosing faults from the optimized fault-space. The improvement in the effectiveness of faults is one order of magnitude, similar to one obtained for faults in microprocessor registers.

Table 5 shows the error detection coverage estimations obtained with non-optimized and optimized fault selection.

Table 5. Error detection coverage estimations (memory)

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

We here observe a similar pattern to that observed for microprocessor registers, where the error detection coverage estimation using non-optimized or optimized fault selection is quite similar. In this case the estimation from the non-optimized campaign is not very accurate since the 95% confidence interval is still wide due to the small number of effective faults (2%).

Figures 9 and 10 show the distribution of detected errors among the exception mechanisms for the two campaigns.

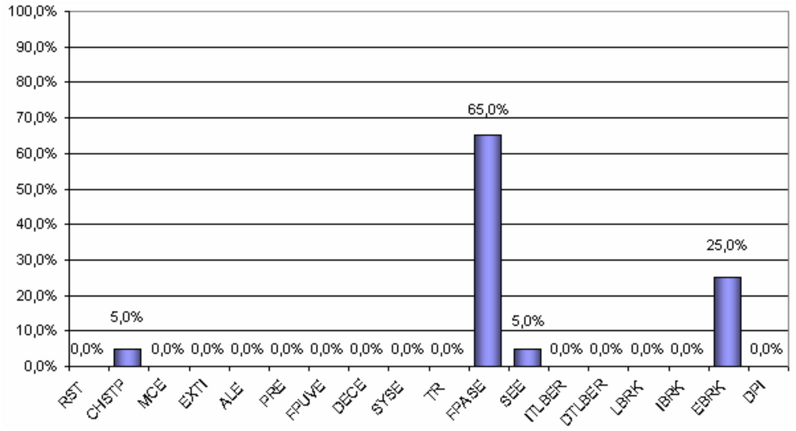

Fig. 9. Exceptions in the jet engine controller non-optimized campaign (40 faults in memory)

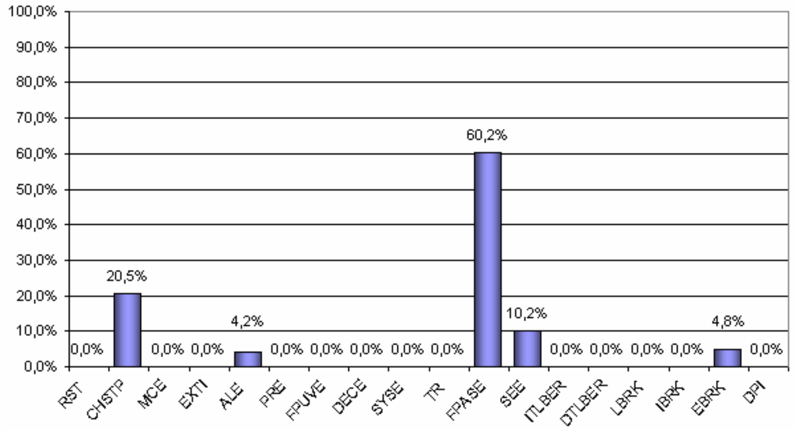

Fig. 10. Exceptions in the jet engine controller optimized campaign (166 faults in memory)

Again, it is possible to observe that the detection mechanisms are activated in a similar but not identical way for the non-optimized and the optimized campaigns.

5.3 Fault-space Considerations

Applying the optimization method to the fault-space of registers for the jet engine controller resulted in the determination of 7.7×10^6 distinct time-location pairs for bit-flips. All the targeted registers are 32 bit registers[3]. The complete non-optimized fault-space of these registers is obtained by flipping each bit of each register, for each instruction executed. This results in a set containing over 500 million bit-flips. Table 6 summarizes these results.

Table 6. Comparison between fault-space sizes (registers)

Campaign		*Size of the fault-space (time-location pairs for bit-flips)*
Jet Engine Controller	*Random*	5.0×10^8
	Optimized	7.7×10^6
Ratio		1.5%

In the case of the memory fault-space 3.3×10^6 possible time-location pairs for bit-flips were determined using optimized fault selection. The complete fault-space 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 for data by the jet engine controller, the size of the complete fault-space is near 200 billion bit-flips.

Table 7. Comparison between fault-space sizes (memory)

Campaign		*Size of the fault-space (time-location pairs for bit-flips)*
Jet Engine Controller	*Random*	1.9×10^{11}
	Optimized	3.3×10^6
Ratio		0.0017%

6 Conclusions and Future Work

The study presented in this paper shows the efficiency of eliminating faults with no possibility of activation and determining equivalence classes among faults. A comparison with traditional non-optimized fault selection in the complete fault-space shows an order of magnitude increase in the effectiveness of faults. The fault-space itself is reduced two orders of magnitude for the registers and four to five orders of magnitude for the memory. Even though these fault-spaces are still quite large when targeting the complete execution of programs, the exhaustive evaluation of small enough sub-routines against all possible bit-flips becomes possible.

All faults targeting the same bit of a given resource, before this resource is read, are considered equivalent. This way, only one representative of these faults is injected. To obtain an accurate estimation of the error detection coverage (or any other dependability measure) it would be necessary to apply a weight corresponding to the

[3] Floating Point Registers are 64-bits long limited by winIDEA to the least significant 32-bits.

number of faults in each equivalence class. However, the error detection coverage estimated by the optimized fault selection is found to be quite similar to the coverage estimated by non-optimized fault selection.

The analysis of assembly constructors limits the technique to the UISA Register Set. Using a debugger/debugging environment that supports tracing of all read operations on all registers during the reference run would allow the fault-space of all registers to be optimized.

Even though activation of faults is ensured by the optimization technique (activation in the sense that the faulty resources are always utilized) not all faults result in effective errors. This occurs when the data is used in a non-sensitive way by the code (regarding the single bit-flip model). An interesting topic for further studies would be to investigate which activated faults are non-effective and why.

The outcome of a fault is highly dependent on the targeted resource. Faults in some registers were observed to have a greater tendency to cause wrong output while faults in other registers cause detected errors more frequently. This motivates a possible evolution in fault selection by using the results of previous fault injection experiments to select the faults that should be injected next (a combination of pre-injection and post-injection analysis). It would be possible to achieve a faster evaluation of specific error detection mechanisms by injecting faults in the resources that are more likely to activate them.

In the future of fault injection the multiple bit-flip fault model may become more important. Microprocessor 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 paper to improve the selection of multiple bit-flip faults.

A research line orthogonal to the optimization of fault-spaces is the improvement of the path coverage obtained during fault injection campaigns (i.e. consider different control flow decisions and the associated fault-spaces). The presented pre-injection analysis and such a path coverage analysis are complementary and could eventually be combined.

There is still room for further optimization by analyzing the error propagation. When a bit-flip is copied from one resource onto another and the first resource is overwritten, the fault in the new location is equivalent to the fault in the first location. The implementation of an analysis taking advantage of this has been started and preliminary results show additional improvement.

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, L. Lemus, P. Gil, "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, P. Folkesson, J. Karlsson, "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. D. Skarin, J. Vinter, P. Folkesson and J. Karlsson, "Implementation and Usage of the GOOFI MPC565 Nexus Fault Injection Plug-in", Tech. Report No. 04-08, Dept. of Comp. Eng., Chalmers University of Technology, Göteborg, Sweden, 2004.
7. Motorola Inc., "MPC565/MPC566 User's Manual", 2nd edition, 2003.
8. X. Castillo and D. Siewiorek, "Workload, Performance and Reliability of Digital Computer Systems", Proc. FTCS-11, June 1981, pp. 84-89.
9. 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.
10. 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.
11. R. Chillarege and N. Bowen, "Understanding Large System Failures – A Fault Injection Experiment", Proc. FTCS-19, June 1989, pp. 356-363.
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, L. Impagliazzo, P. Marmo, "Fault-List Collapsing for Fault Injection Experiments", RAMS 98, January 1998, pp. 383-388.
14. T. Tsai, M.-C. Hsueh, H. Zhao, Z. Kalbarczyk, R. Iyer, "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, F. Corno, M. Reorda, G. Squillero, L. Entrena, C. Lopez, "New Techniques for Speeding-up Fault-injection Campaigns", Proc. DATE 2002, March 2002, pp. 847-852.
16. J. Arlat, J.-C. Fabre, M. Rodríguez, F. Salles, "Dependability of COTS Microkernel-Based Systems", IEEE Transactions on Computers, Vol. 51, No. 2, February 2002, pp. 138-163.
17. Motorola Inc., "RISC Central Processing Unit Reference Manual", revision 1, 1999.
18. R. Barbosa, J. Vinter, P. Folkesson and J. Karlsson, "Fault Injection Optimization through Assembly-Level Pre-Injection Analysis", Tech. Report No. 04-07, Dept. of Comp. Eng., Chalmers University of Technology, Göteborg, Sweden, 2004.
19. winIDEA – iSystem's Integrated Development Environment, http://www.isystem.se/products/ide.htm, January 24th, 2005.
20. PHYTEC Technology Holding Company, http://www.phytec.com/sbc/32bit/pc565.htm, January 24th, 2005.
21. iSYSTEM AB, http://www.isystem.se, January 24th, 2005.
22. iC3000 Active Emulator, http://www.isystem.se/products/emulators.htm#three, January 24th, 2005.