A Comparison of Simulation Based and Scan Chain Implemented Fault Injection

Peter Folkesson, Sven Svensson, Johan Karlsson

Laboratory for Dependable Computing
Department of Computer Engineering
Chalmers University of Technology
S-412 96 Göteborg, Sweden

Abstract

*This paper compares two fault injection techniques: scan chain implemented fault injection (SCIFI), i.e. fault injection in a physical system using built in test logic, and fault injection in a VHDL software simulation model of a system. The fault injections were used to evaluate the error detection mechanisms included in the Thor RISC microprocessor, developed by Saab Ericsson Space AB. The Thor microprocessor uses several advanced error detection mechanisms including control flow checking, stack range checking and variable constraint checking. A newly developed tool called FIMBUL (*Fault Injection and Monitoring using BUilt in Logic*), which uses the Test Access Port (TAP) of the Thor CPU to do fault injection, is presented. The simulations were carried out using the MEFISTO-C tool and a highly detailed VHDL model of the Thor processor. The results show that the larger fault set available in the simulations caused only minor differences in the error detection distribution compared to SCIFI and that the overall error coverage was lower using SCIFI (90-94% vs. 94-96% using simulation based fault injection).*

Keywords: *fault injection, experimental validation, concurrent error detection, boundary scan, VHDL*

1 Introduction

Fault injection has become an established method for testing and evaluating the fault handling capabilities of fault-tolerant and fail-safe systems [1]. Techniques for fault injection fall into two categories (i) fault injection in software simulation models of systems, and (ii) fault injection into physical systems, i.e. prototypes or actual systems.

Techniques in the two categories complement each other as they are used in different phases of the design process. One advantage of simulation-based fault injection is that it can be employed early in the design process allowing early detection of design faults, which thus reduces the cost for correcting such faults. It also provides a high degree of controllability and observability. The main drawback to simulation-based fault injection is the time overhead involved in simulations, which puts practical limitations on the amount hardware and software activity that can be simulated.

Fault injection in physical systems is important because it tests the actual implementation of fault handling mechanisms. However, techniques for injecting faults in physical systems, such as pin-level fault injection [2][3] or software implemented fault injection (SWIFI) [4][5][6] provides limited controllability and observability. Also, these techniques may not be able emulate the effects of all faults since they suffer from a lack of physical reachability.

One way of improving reachability as well as observability and controllability in the evaluation of physical systems is to use scan-chain implemented fault injection (SCIFI). This technique injects faults via the built-in test logic, i.e. boundary scan chains and internal scan chains, present in many modern VLSI circuits.

To this end, we have developed a tool called FIMBUL (*F*ault *I*njection and *M*onitoring using *BU*ilt-in *L*ogic) for fault injection via the test access port (TAP) of microprocessors. We have used the tool to evaluate the error detection mechanisms included in the Thor microprocessor, a 32 bit stack oriented RISC processor developed by Saab Ericsson Space AB [7].

The Thor microprocessor is primarily intended for embedded systems in space applications. The objective of the fault injection experiments was to assess Thor's capability of handling single event upsets caused by ionizing particles in the space environment. To model the effects of such faults we injected single bit-flips into the internal state elements (flip-flops and latches) of Thor. The fault locations were selected randomly among the 2250 state elements that are accessible via the TAP of Thor. We also conducted fault injection experiments using a detailed VHDL simulation model of Thor in order to assess the

accuracy of the SCIFI technique. The simulation model allowed us to inject faults into all of the 3971 state elements included in Thor. This way, we could investigate the differences in the results obtained by the two fault injection techniques.

In the next section of this paper we describe the architecture of the Thor microprocessor and its error detection mechanisms. In Section 3 an overview of the FIMBUL tool and the VHDL simulation tool is presented. Section 4 describes the experimental set-ups, while Section 5 presents the results of the experiments. Finally, a discussion and conclusions are presented in Section 6.

2 Overview of the Thor processor

The Thor microprocessor is designed and sold by Saab Ericsson Space AB. It is a 32-bit RISC processor based on a stack-oriented instruction set architecture and is intended for embedded real-time applications with high dependability requirements. The Thor processor has on-chip support for real-time processing specifically for the Ada language including task scheduling and dispatch, communication between tasks, time handling, accurate delays and fast interrupt handling. The support for dependability includes several internal error detection mechanisms and the possibility to run the processor in a pair configuration with a comparator function (master/slave comparator configuration).

2.1 Architecture

The stack-oriented instruction set architecture gives compact machine code due to frequent use of an implicit operand addressing mode. The drawback of requiring the operands to be located at the stack top, present in regular stack architectures, is eliminated by the implementation of a stack relative addressing mode. Because of the compact machine code Thor requires less main memory than conventional microprocessors. This reduces weight and power consumption, which is important in space applications. A reduction of memory size also improves reliability since memory is particularly vulnerable to single event upsets.

A feature of a stack based architecture is the high locality of data references. This property results in a high hit rate of the data cache. Another consequence of a stack based architecture is that fast context switching can be performed since only a stack pointer needs to be updated.

In Figure 1 the main blocks of the processor are depicted. The instruction pipeline is found in the middle of the figure. It consists of an *Instruction Fetch* stage (*IF*) including a four instruction word prefetch buffer, an *Address Generation* stage (*AG*), an *Operand Fetch* stage (*OF*) incorporating a 128 byte direct mapped write-back data cache (32 words) and an *EXecute* stage (*EX*). An instruction enters the pipeline in the IF stage and is then passed from one stage to the next each system clock cycle except when a pipeline stall occurs. This can happen for example when the EX stage is performing a multicycle operation or the EX stage is waiting for a write operation to slow memory.

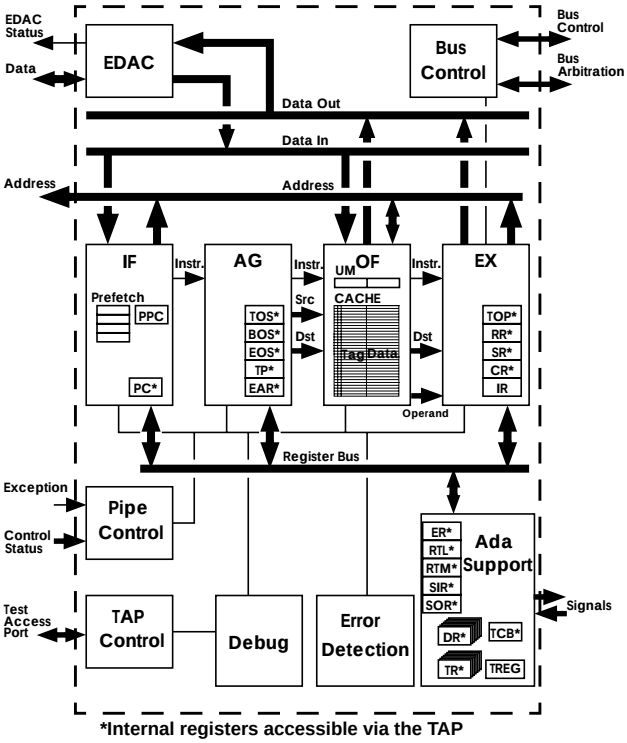

Figure 1: Block diagram of the Thor processor.

Actions for the control of the pipe are managed by the *Pipe Control* block. All memory bus accesses requested from different blocks of the processor are managed by the *Bus Control* block. Task handling is implemented in the *Ada Support* block, which can administer up to 8 tasks and also features a 64-bit real-time clock for scheduling purposes.

The error detection mechanisms are controlled and supervised by the *Error Detection* block. Most of these mechanisms are implemented in the pipeline blocks but the error detection and correction of data transfers between Thor and external memory is implemented in a separate block, the *EDAC* block.

The *TAP control* and *Debug* blocks implements the test and debug facilities. The TAP conforms to the IEEE 1149.1 standard for boundary scan. It provides access to the 101 chip-pin signals via a boundary scan-chain, as well as all memory elements in the cache and 18 internal registers (marked with an asterisk in Figure 1) via an internal scan-chain. The TAP also gives access to a debug scan-register, which allows the user to set break-points. When a break-point is reached, the processor halts and the values of the memory elements in the scan-chain can be read or

written via the TAP. The user can program the debug scan-register to halt the processor when it accesses a particular address or a range of addresses, uses a particular data value or a range of data values, executes a specific instruction, or at a specific time determined by the real-time clock. This feature provides very powerful support for fault injection.

2.2 Error detection mechanisms

The error detection mechanisms supported by Thor can be divided into *control flow checking*, *run-time checks*, *memory data checks* and *master/slave comparator operation*.

The *control flow checking* mechanism is a signature monitoring scheme that checks the control flow of the program execution. For each basic block (i.e. branch free interval) in the program, a signature is calculated using the arithmetic sum of the opcodes of the instructions in the basic block. At run-time a dedicated monitor calculates the signature from the instructions fetched by the processor. This signature is compared by a special NOP instruction, executed at the end of the basic block, with the correct signature which is provided as an operand to the NOP instruction by the compiler. For a general description of control flow checking techniques see [8].

The *run-time checks* are implemented as hardware exceptions in Thor, see Table 1. Some of the run-time checks are commonly found in other microprocessors such as illegal opcode detection, division by zero check, etc. Some checks, however, are Thor specific. These are the data error, constraint error and storage error exceptions. Data errors occur when the *memory data check mechanism* detected an error. This mechanism uses a (39,32) modified Hamming code to correct single-bit errors and detect double-bit errors in the main memory. Constraint errors occur when the checks made by the run-time assertion instructions Compare Lower Limit (CLL) and Compare Upper Limit (CUL) fail. These can be used to check index bounds of arrays or variable value ranges. The storage error exception is used for detecting attempts to access memory outside a task's designated stack area in user mode. The exceptions numbered 8 to 14 provides hardware support for exceptions required by the Ada language.

The *master slave comparator operation*, which detects errors by comparing the results from two Thor microprocessors, is suitable for applications requiring 100% coverage of internal CPU errors.

The goal of this study was to evaluate the effectiveness of the control flow checking mechanism and the run-time checks with respect to single event upsets occurring inside the CPU. These are the primary mechanisms for detecting CPU faults when Thor is used in a single processor configuration. The memory data check mechanism and master/slave comparator configuration were not evaluated.

No	Exception	Corresponding Ada exception	Description
1	BUS ERROR	-	Bus time-out on external memory access
2	ADDRESS ERROR	-	Chip input signal *AE (Address Exception)* is asserted, or an operand effective address was larger than 2Gbyte
3	DATA ERROR	-	Chip input signal *DE (Data Exception)* is asserted
4	INSTRUCTION ERR.	-	Attempt to execute a privileged instruction in user mode or an illegal instruction
5	JUMP ERROR	-	Attempt to jump, call or return to a target address outside memory address space
6	-	-	Reserved
7	-	-	Reserved
8	CONSTRAINT ERROR	CONSTRAINT ERROR	A constraint of a CLL or CUL instruction not satisfied
9	ACCESS CHECK	CONSTRAINT ERROR	Attempt to follow a null pointer
10	STORAGE ERROR	STORAGE ERROR	Attempt to access memory outside the task's stack in user mode
11	OVERFLOW CHECK	NUMERIC ERROR	Overflow of signed integer and float arithmetic operations
12	UNDERFLOW CHECK	NUMERIC ERROR	Underflow or denormalized result of float arithmetic operations
13	DIVISION CHECK	NUMERIC ERROR	Divide by zero for integer division Divide by ±0 for float division
14	ILLEGAL OPER.	NUMERIC ERROR	Illegal operation for float and double arithmetic instructions involving 0 and ∞
15	TASKING ERROR	TASKING ERROR	Reserved for future use

Table 1: Hardware exceptions in Thor

3 The fault injection tools

This section provides an overview of FIMBUL and the MEFISTO-C tool. The latter was used for the simulation based fault injections experiments using the VHDL model of Thor.

3.1 The FIMBUL tool

FIMBUL (*F*ault *I*njection and *M*onitoring using *BU*ilt in *L*ogic) is a tool that can inject faults via the TAP facilities of the Thor CPU. Transient faults can be injected into any of the locations accessible by the boundary and internal scan-chains of the Thor CPU using the bit-flip fault model. The points in time when a fault is injected can be chosen by setting break-points using the Thor debug scan register. The selection of when and where to inject a fault can be made either randomly or non-randomly within the limitations of the Thor hardware.

Figure 2 shows an overview of FIMBUL. The FIMBUL software consists of a fault injection program written in GNU Ada and several analysis programs written mainly in the Perl programming language which all execute on a UNIX workstation. The hardware consists of a Thor evaluation board [9], featuring the Thor CPU and memory circuits, installed on a Sun UNIX workstation using an SBus interface. All communication between FIMBUL and the Thor CPU is performed via UNIX device drivers.

There are three phases involved in conducting fault

injection campaigns using the FIMBUL tool: the *set-up*, *fault injection* and *analysis* phases.

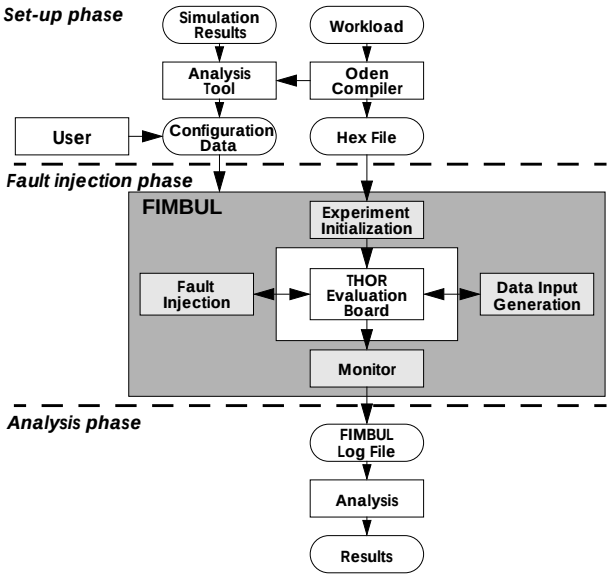

Figure 2: FIMBUL overview

3.1.1 The set-up phase. In the set-up phase, the workload chosen for fault injection experiments is analysed either manually or by an analysis tool in order to produce configuration data for the experiments. The configuration data contains information about when and where to inject faults and the total number of faults to be injected during the fault injection campaign, i.e. the number of *fault injection experiments* to perform. The configuration data also contains information about the termination conditions for the fault injection experiments. A fault injection experiment can be terminated either when a time-out value has been reached, an error has been detected or the execution of the workload ends, whichever comes first. When the execution of the workload should end is also defined in the configuration data. The workload either consists of an infinite loop that should be executed a certain number of times or a program that terminates by itself.

The data input generation module of FIMBUL (see Figure 2) is used for generating input data to the workload and obtaining the results produced by the workload, e.g. acting as the sensors and actuators of the target system. In order for the data input generation module to know where to store the input data and where to look for the results produced by the workload, pointers to the memory locations of the input and output data within the target system are also given in the configuration data. Using FIMBUL to simulate the environment of the target system in this way is optional but in order to identify system failures (see Section 3.1.3), FIMBUL must know where to look for the results produced by the workload.

The configuration data also contains the operation mode for the FIMBUL fault injection module. There are four operation modes:

- *Normal*: The CPU state is logged only when a fault injection experiment terminates. This is the normal operation mode of FIMBUL.
- *Normal reference*: The same as normal mode but no fault injection is performed.
- *Detail*: The CPU state is logged after each instruction executed since fault injection. This produces an execution trace allowing the error propagation to be analysed in detail.
- *Detail reference*: Produces an execution trace of the whole workload execution without injecting any faults. This is compared with execution traces obtained in detail mode in order to study the impact of errors.

The CPU state logged always includes the contents of the Thor scan-chains and the contents of the memory locations where the results produced by the workload are stored, together with information about when and where any faults were injected.

3.1.2 The fault injection phase. In the fault injection phase, the configuration data is read and interpreted by FIMBUL. The Thor evaluation board is initialized and the workload is downloaded to the evaluation board.

If FIMBUL operates in normal mode, a reference execution of the workload without performing any fault injection is made first, i.e. FIMBUL operates in normal reference mode first. The Thor evaluation board is then reset and the execution of the first fault injection experiment begins.

The configuration data contains information about when a fault should be injected. A break-point is set according to this information and the CPU starts executing the workload. When the break-point condition has been fulfilled, execution of the workload stops and fault injection takes place. Fault injection is made by reading the contents of the boundary and internal scan-chains of the Thor CPU, inverting the bits stated in the configuration data and writing back the fault injected scan-chains to Thor. The execution then starts from where the CPU was halted and continues until the termination condition stated in the configuration data is reached. The CPU state is then logged and the Thor evaluation board is reset and another fault injection experiment begins.

When FIMBUL operates in detail mode, fault injection is performed using the procedure described for the normal mode operation. The only differences are that the CPU executes each workload instruction step-by-step after fault injection has been made and that the CPU state is

logged after each step instead of only once when the fault injection experiment terminates.

If FIMBUL operates in detail reference mode, no fault injection is made and the CPU executes each workload instruction step-by-step until the termination condition given in the configuration data is reached. The CPU state is logged after each workload instruction executed.

3.1.3 The analysis phase. In the analysis phase, the data logged in the fault injection phase is analysed to obtain dependability measures about the target system. The scripts used to calculate the measures must be developed specifically for the target system. For the evaluation of Thor, the analysis scripts compare the CPU states logged during normal mode operation with the CPU state produced in the normal reference mode to identify:

- *Detected errors*: The fault injection experiments where an error was detected by an error detection mechanism.
- *System failures*: The fault injection experiments where incorrect results were produced by the workload or the violation of any timeliness requirements were made, e.g. the results were delivered to late.
- *Latent errors*: The fault injection experiments where differences in the CPU states logged at the end of each experiment and the normal reference mode are observed but which could not be identified as either detected errors or system failures.
- *Overwritten errors*: The fault injection experiments where no differences between the CPU states logged at the end of each experiment and normal reference mode are observed.

The detected errors are further classified into errors detected by either the control flow checking mechanism or each of the run-time checks of the Thor CPU (see Section 2.2), or as *other errors*, i.e. when the information about which mechanism that detected the error is unavailable.

Due to a lack of observability using the FIMBUL tool on the Thor processor (2612 of 3917 state elements are observable using FIMBUL), some errors can not be classified with absolute certainty. These are the errors detected by the control flow mechanism and the system failures where the timeliness requirements are violated.

The following errors generate the same observable behaviour of the Thor CPU as a control flow error:

- An exception occurred during exception processing.
- HALT instruction in the EX stage.

The FIMBUL analysis scripts try to classify these errors as either control flow errors or other errors, but may not always make the correct classifications.

The timeliness requirement used in this study is that a correct result should not be delayed more than 50 clock cycles, otherwise it is classified as a system failure. FIMBUL uses the built in real-time clock of the Thor CPU for detecting if the timeliness requirement is violated. However, the contents of the real-time clock registers may sometimes be incorrect due to fault injection. An error which should be classified as a latent error may therefore be incorrectly classified as a system failure.

3.2 The MEFISTO-C tool

MEFISTO-C is a tool developed at Chalmers University of Technology for conducting fault injection experiments using VHDL simulation models. The tool is an improved version of the MEFISTO tool which was developed jointly by LAAS-CNRS and Chalmers [10]. (A similar tool called MEFISTO-L has been developed at LAAS-CNRS). MEFISTO-C uses the Vantage Optium VHDL simulator and injects faults via simulator commands in variables and signals defined in the VHDL model. It offers the user a variety of predefined fault models as well as other features to set-up and automatically conduct fault injection campaigns on a network of UNIX workstations.

4 Experimental set-up

The FIMBUL tool was implemented on a 50 MHz Sun Sparc Classic workstation with a Thor evaluation board connected to the SBus interface. The Thor evaluation board used a 12.5 MHz clock and was equipped with 512 KB RAM. The workstation was connected via a local network to a file server where all results were stored.

For the simulation experiments we used a structural VHDL model of the Thor CPU and a 512 KB RAM. The Thor model was highly detailed with all internal state elements available as signal objects. The simulation experiments were conducted by MEFISTO-C using five 70 MHz Sun Sparc 5 workstations, each having its own 1 Gbyte disk space for storing results.

The fault injection experiments were carried out using two different workloads. One is a digital control application (Powerpos) and the other is an implementation of the quick-sort algorithm (Qsort). It is well known that the workload have a substantial impact on the results of a fault injection campaign [11][12][13]. The digital controller was chosen because it represents an application which is common in embedded systems. The quick sort application was chosen because it has been commonly used in fault injection experiments, and thus provides an opportunity for comparing our results with those obtained in other experiments. Both workloads were written in Ada and compiled by the Oden Ada compiler system [14], which has been developed specifically for the Thor processor.

The remainder of this section provides detailed information about the workloads and the fault injection campaigns, as well as a short comparison of the FIMBUL and the MEFISTO-C tools.

4.1 The Powerpos workload

The task of the Powerpos workload is to control the transfer of an object to a position given by a *set-point value*. This is done by controlling an actuator that applies a force on the object. The digital controller acquires data from sensors measuring speed and position of the object (the state variables of the system), and calculates new control values for output to the actuator.

By using an emulation model of the system environment including the controlled object, a close to authentic representation of the state data (sensor data) is achieved. The emulation of the system environment is done by the data input generation module in FIMBUL, and by VHDL code in the simulation model when using MEFISTO-C.

The workload is based on a repeated program sequence wherein Thor reads *state data* (represented by sensor measurements) from the controlled object and a *set-point value* from the control input, calculates the value of the *control signal* and sends it to the actuator, thereby closing the control loop. The workload is designated *task 1* of the eight tasks supported on-chip. To force Thor to perform a scheduling operation, a delay instruction is inserted at the end of the control loop. This operation invokes a dummy task, *task 0*. After the specified delay has passed, *task 1* is allocated the CPU again and the next execution of the control loop starts.

The size of the Powerpos workload is 900 bytes and it utilizes 33 of the 80 instructions of the Thor CPU. The execution time is 2350 cycles, which corresponds to three repetitions of the control loop. Fault injection is performed during the second iteration of the control loop.

4.2 The Qsort workload

The sort workload (Qsort) implements a recursive quick-sort algorithm. It sorts a list containing seven data elements of the Ada predefined type *float*. The size of this workload is 756 bytes and it utilizes 27 of the 80 instructions of the Thor CPU. The execution time is 1755 cycles and fault injection is performed in any of these cycles.

4.3 Fault injection campaigns

Table 2 summarizes the fault injection campaigns conducted to evaluate the error detection mechanisms in Thor. As already mentioned, the fault model used was single bit-flips in latches and flip-flops (state elements). The injected faults were selected randomly by sampling the fault space using a uniform sampling distribution. We define the fault space as the Cartesian product $F = L \times T$, where L is the set of all fault locations (state elements) and T the set of all time points when faults can be injected.

The time resolution is one machine instruction for FIMBUL and one clock cycle in the simulations. Since most instructions are executed in one clock cycle, the difference in time resolution is small. It should be noted that the injected faults sets are identical for campaign 3 and 4b. This allowed us to verify that the tools produced identical results when the same faults were injected. To verify that the responses indeed were identical, it was necessary to use the detailed operation mode for FIMBUL.

Campaign No.	Tool	Workload	Operation mode (FIMBUL only)	Fault space	
				L (No. of fault locations)	T (No. of time points)
1	FIMBUL	Powerpos	Normal	2250	~700 (instructions)
2	MEFISTO-C	Powerpos	-	3971	~800 (clock cycles)
3	FIMBUL	Qsort	Detailed	2250	~1300 (instructions)
4a	MEFISTO-C	Qsort	-	3971	~1800 (clock cycles)
4b	MEFISTO-C	Qsort	-	2250	~1300 (instructions)

Table 2: Fault injection campaigns

Each of the fault injection campaigns consists of up to six sub-campaigns, in which faults were injected into one functional block of Thor. That is, we used stratified sampling [15] by dividing the L set into six strata. The Thor CPU was divided into the following functional blocks, or strata: The IF stage, AG stage, CACHE, EX stage and Ada Support block (ADA). A sixth strata, unavailable for fault injection using the FIMBUL tool, called the MISC block (containing the Pipe Control block, Bus Control block, Error Detection block, EDAC block and the OF stage without the cache), was used in campaign 2 and 4a.

4.4 Comparison of the fault injection tools

	FIMBUL (Scan chain implemented fault injection)	MEFISTO-C (Simulation based fault injection)
No. of fault injection locations (state elements)	2250	3971
Time resolution for fault injection	one instruction (= one clock cycle for most instructions)	one clock cycle, or less
No. of observable state elements	2612	3971
Performance (faults injected per second)	Normal mode: 0.25 faults/s Detail mode: 0.003 faults/s	0.002 faults/s per workstation used

Table 3: Comparison of FIMBUL and MEFISTO-C

Table 3 summarizes some of the key characteristics of the two fault injection tools. It is shown that FIMBUL in normal mode is much faster than MEFISTO-C (0.25 faults injected per second vs. 0.002 faults/s for each workstation participating in the simulations). When FIMBUL executes in detail mode, the performance of the two techniques is similar. The maximum time resolution for FIMBUL is one instruction as Thor only can be halted between two instructions. The time resolution for MEFISTO-C is user definable during the campaign set-up. We decided to use a time resolution of one clock cycle.

5 Results

In this section we first describe the error classification scheme and the formulas used for calculating confidence intervals. We then present the results of fault injection campaign 1, 2, 3 and 4a in which the two tools used different fault sets, as the faults were sampled from the entire fault space available in each tool. Finally, we compare the results of campaign 3 and 4b, in which identical fault sets were used for the two tools.

5.1 Error classification

In the subsequent sections, the results from the fault injection campaigns are presented using tables where the errors are divided into several error categories. The errors are divided into two main categories: “Non Effective Errors” and “Effective Errors”. The “Non Effective Errors” are further divided into “Latent Errors” and “Overwritten Errors”. The “Effective Errors” are further divided into errors leading to system failures (“System Failure”) or detected errors. The detected errors are those detected by an exception (“Exception *n*” categories), detected by the control flow mechanism (“Control Flow Errors”), or otherwise detected (“Other Errors”), as explained in Section 3.1.3. The error coverage (“Coverage”) is defined as the percentage of errors that are either detected or non effective, i.e. overwritten or latent.

The observed relative frequency of the error categories, for each functional block, or strata, of the Thor CPU are presented in the results. In addition, the weighted total relative frequency, $\hat{p}_e$, is given for each error category. Let $\hat{p}_{el}$ denote the observed relative frequency for error category *e*, in strata *l*. $\hat{p}_{el} = \frac{n_{el}}{n_l}$, where n_{el} is the observed number of errors in category *e* for strata *l*, and n_l is the total number of fault injections in strata *l*. The weighted total relative frequency using *L* strata is then given as [15]

$$\hat{p}_e = \sum_{l=1}^{L} \frac{N_l}{N} \cdot \hat{p}_{el}$$

where N_l is the size of strata *l* and $N=N_1+N_2+...+N_L$ is the total population size.

The corresponding 95% confidence intervals for all the weighted total relative frequencies are also given. Based on the normal approximation we can derive an approximate 95% confidence interval for p_e, the overall probability for error category *e*, as

$$p_e = \hat{p}_e \pm 1.96 \cdot \sqrt{\sum_{l=1}^{L} \left(\frac{N_l}{N}\right)^2 \cdot \frac{\hat{p}_{el}(1-\hat{p}_{el})}{n_l - 1}}$$

5.2 Results obtained using different fault sets

Table 4 shows the results obtained using the Powerpos workload (campaign 1 and 2). The results show that the address error-, variable constraint check-, and control flow error mechanisms are the most efficient mechanisms of the Thor processor. The CACHE block, containing the 1824 state elements of the data cache, is the functional block most sensitive to fault injection, i.e. the CACHE block has the lowest coverage (except for campaign 1 where the EX block has the lowest coverage). This suggests that a parity protection of the data cache would improve the overall error coverage. The coverage is 94% using FIMBUL and 96% using MEFISTO-C, which means that the SCIFI technique gives a more pessimistic coverage estimation.

The percentage of effective errors is higher using SCIFI than using simulation-based fault injection (30% vs. 24%) but only 12 error detection mechanisms were triggered using the SCIFI technique compared to all 14 mechanisms using the simulation-based technique. The differences between the techniques are most prominent for the IF block and least prominent for the ADA block. The largest differences are observed for the address error mechanism (exception 2) (15% using FIMBUL vs. 8% using MEFISTO-C) and for the control flow error mechanism (1.19% vs. 2.91%).

Table 5 shows the results obtained using the Qsort workload (campaign 3 and 4a). Again, the coverage is lower using FIMBUL than using MEFISTO-C (90% vs. 94%) and the CACHE block is the functional block most sensitive to fault injection. The results also show that the address error-, variable constraint check- and control flow error mechanisms are among the most triggered mechanisms, similar to what could be observed using the Powerpos workload.

Again, the percentage of effective errors is higher using SCIFI than using simulation-based fault injection (43% vs. 31%). The number of error detection mechanisms triggered is also lower using SCIFI (10 using FIMBUL vs. all 14 mechanisms using MEFISTO-C). The differences are most prominent for the IF block and least prominent for the ADA block, just as for the Powerpos workload, but the differences between the total amount of latent and overwritten errors observed is lower for the Qsort workload. Again, when looking at the weighted total, the only major differences are the percentage of the address error mechanism (exception 2) (18% using FIMBUL vs. 11% when using the extended fault set of MEFISTO-C) and the percentage of control flow errors (0.57% vs. 2.55%) (excluding the differences in the percentage of latent and overwritten errors obtained for each technique).

Several differences between the workloads can be observed. The variable constraint checks (exception 8) and

Fault injection tool	FIMBUL						MEFISTO-C						
Part of CPU fault injected (no. of state elements)	IF (30)	AG (146)	CACHE (1824)	EX (90)	ADA (160)	**Weighted total (2250)**	IF (233)	AG (206)	CACHE (1824)	EX (482)	ADA (320)	MISC (906)	**Weighted total (3971)**
Latent Errors	*0.22%*	*32.39%*	*22.98%*	*26.98%*	*92.17%*	**28.37% (±0.44%)**	*0.00%*	*15.37%*	*11.65%*	*12.07%*	*51.98%*	*13.11%*	**14.79% (±0.80%)**
Overwritten Errors	*0.00%*	*16.10%*	*46.45%*	*53.81%*	*5.76%*	**41.26% (±0.51%)**	*50.66%*	*37.48%*	*57.95%*	*68.64%*	*47.45%*	*74.76%*	**60.75% (±1.18%)**
Total (Non Effective Errors)	***0.22%***	***48.49%***	***69.43%***	***80.78%***	***97.94%***	**69.63% (±0.47%)**	***50.66%***	***52.85%***	***69.60%***	***80.71%***	***99.43%***	***87.88%***	**75.54% (±1.05%)**
Exception 1 (Bus Error)	*0.00%*	*0.00%*	*0.00%*	*0.00%*	*0.00%*	**0.00%**	*0.00%*	*0.00%*	*0.00%*	*0.00%*	*0.00%*	*0.24%*	**0.05% (±0.04%)**
Exception 2 (Address Error)	*39.22%*	*12.64%*	*16.42%*	*1.51%*	*0.00%*	**14.72% (±0.37%)**	*13.41%*	*10.54%*	*13.30%*	*0.57%*	*0.00%*	*4.00%*	**8.42% (±0.74%)**
Exception 3 (Data Error)	*0.00%*	*0.00%*	*0.00%*	*2.41%*	*0.00%*	**0.10% (±0.03%)**	*0.00%*	*0.00%*	*0.00%*	*1.07%*	*0.00%*	*0.00%*	**0.13% (±0.07%)**
Exception 4 (Instruction Error)	*42.46%*	*1.08%*	*2.66%*	*0.53%*	*0.43%*	**2.84% (±0.17%)**	*7.62%*	*1.55%*	*2.30%*	*0.71%*	*0.23%*	*091%*	**1.90% (±0.35%)**
Exception 5 (Jump Error)	*0.22%*	*0.81%*	*0.30%*	*0.08%*	*0.00%*	**0.30% (±0.06%)**	*0.50%*	*0.00%*	*0.45%*	*0.57%*	*0.00%*	*0.16%*	**0.34% (±0.15%)**
Exception 8 (Constraint Check)	*0.65%*	*0.81%*	*1.39%*	*2.86%*	*0.00%*	**1.30% (±0.12%)**	*2.15%*	*1.21%*	*4.25%*	*1.79%*	*0.23%*	*0.79%*	**2.55% (±0.43%)**
Exception 9 (Access Check)	*0.00%*	*0.00%*	*0.00%*	*0.00%*	*0.00%*	**0.00%**	*0.17%*	*0.86%*	*0.00%*	*0.00%*	*0.00%*	*0.00%*	**0.05% (±0.04%)**
Exception 10 (Storage Error)	*0.00%*	*31.49%*	*0.53%*	*0.00%*	*0.00%*	**2.47% (±0.14%)**	*4.47%*	*23.49%*	*0.30%*	*0.29%*	*0.11%*	*0.44%*	**1.75% (±0.24%)**
Exception 11 (Overflow)	*0.00%*	*0.09%*	*0.27%*	*0.53%*	*0.00%*	**0.25% (±0.05%)**	*0.66%*	*0.00%*	*0.30%*	*0.43%*	*0.00%*	*0.12%*	**0.26% (±0.13%)**
Exception 12 (Underflow)	*0.00%*	*0.00%*	*0.01%*	*0.00%*	*0.00%*	**0.01% (±0.01%)**	*0.00%*	*0.00%*	*0.00%*	*0.00%*	*0.00%*	*0.04%*	**0.01% (±0.02%)**
Exception 13 (Division Check)	*0.00%*	*0.00%*	*0.12%*	*0.30%*	*0.00%*	**0.11% (±0.04%)**	*0.00%*	*0.00%*	*0.25%*	*0.21%*	*0.00%*	*0.08%*	**0.16% (±0.11%)**
Exception 14 (Illegal Operation)	*1.29%*	*1.98%*	*0.53%*	*0.38%*	*0.00%*	**0.59% (±0.08%)**	*1.16%*	*2.94%*	*0.60%*	*1.00%*	*0.00%*	*0.48%*	**0.73% (±0.20%)**
Control Flow Errors	*15.52%*	*0.00%*	*1.21%*	*0.00%*	*0.00%*	**1.19% (±0.11%)**	*14.57%*	*0.69%*	*0.80%*	*9.21%*	*0.23%*	*2.26%*	**2.91% (±0.33%)**
Other Errors	*0.43%*	*2.20%*	*0.07%*	*0.00%*	*0.69%*	**0.25% (±0.05%)**	*0.33%*	*4.84%*	*1.40%*	*0.64%*	*0.00%*	*0.28%*	**1.05% (±0.26%)**
System Failure	*0.00%*	*0.40%*	*7.05%*	*10.63%*	*0.95%*	**6.23% (±0.26%)**	*4.30%*	*1.04%*	*6.45%*	*2.79%*	*0.00%*	*2.34%*	**4.14% (±0.53%)**
Total (Effective Errors)	***99.78%***	***51.51%***	***30.57%***	***19.22%***	***2.06%***	**30.37% (±0.47%)**	***49.34%***	***47.15%***	***24.43%***	***19.29%***	***0.57%***	***12.12%***	**24.46% (±1.05%)**
Coverage	***100.00%***	***99.60%***	***92.95%***	***89.37%***	***99.05%***	**93.77% (±0.26%)**	***95.70%***	***98.96%***	***93.55%***	***97.21%***	***100.00%***	***97.66%***	**95.86% (±0.53%)**
Total no. of faults injected	***464***	***2223***	***27066***	***1327***	***2325***	**33405**	***604***	***579***	***2000***	***1400***	***883***	***2524***	**7990**

Table 4: Results of campaign 1 and 2

the address error mechanism (exception 2) occur more frequently for Qsort than for Powerpos (7%-9% vs. 1%-3% for the constraint checks and 11%-18% vs. 8%-15% for the address error mechanism). The percentage of effective errors is 31%-43% for the Qsort workload and only 24%-30% for the Powerpos workload. The total error coverage is 90%-94% for the Qsort workload and 94%-96% for the Powerpos workload.

5.3 Results obtained using identical fault sets

Table 6 shows the results obtained using the fault injection campaigns 3 and 4b where identical fault sets were used by the two tools. These campaigns were conducted to verify that the tools produced the same results for identical faults.

A total of 4626 faults were injected. For 386 faults, the tools produced different results.

The reasons for these discrepancies are given in Table 7. Here, the number of experiments for which the results differ are given and categorized as follows:

- *Classification*: The errors which are classified differently due to the lower observability available in FIMBUL, e.g. due to the observability problems described in Section 3.1.3, or due to the fact that the time-out for ending the simulations was reached before the error could be detected.
- *Implementation*: The experiments which differ because of differences between the Thor VHDL model and the actual chip. These discrepancies are most notably caused by different exceptions being signalled by the RET (return) instruction when returning to an invalid address. Exception 5 (“Jump Error”) is signalled in the VHDL model instead of exception 2 (“Address Error”) for the actual chip.
- *Tool*: The discrepancies that exist due to different operation of the tools. The faults may be injected at slightly different points in time, or the CPU states may not be exactly equal during fault injection. The most common discrepancy of this type is the reset of the IF block prefetch buffer that occurs when the CPU is halted for fault injection using FIMBUL. This causes the instruction flow to be transferred immediately to an erroneous location if a fault is injected into the program counter (PC), while the correct instructions in the buffer will be executed in the simulations. The state of the UM buffer, i.e. the buffer containing the next dirty cache word to be written to memory, may be also differ after fault injection since it is emptied during available free bus cycles. These cycles may not be equal in the Thor VHDL model and the actual CPU when a fault has been injected.
- *Unknown*: We have not been able to identify the reason for these discrepancies.

The percentage of experiments belonging to each category of the total number faults injected are also given in Table 7. It shows that the most common causes for the discrepancies are the limitations in the observability for the SCIFI technique (only 2612 of all 3971 locations are observable) causing some errors to be classified as overwritten instead of latent. These discrepancies account for 5% of all experiments. The percentage of discrepancies due to implementation differences between the VHDL model and the actual chip is 0.3%. The most serious discrepancies, those due to different operation of the tools, account for about 3% of all experiments. The results clearly demonstrate that the VHDL model is an accurate, although not perfect, representation of the real Thor CPU.

Fault injection tool	FIMBUL						MEFISTO-C						
Part of CPU fault injected (no. of state elements)	IF (30)	AG (146)	CACHE (1824)	EX (90)	ADA (160)	**Weighted total (2250)**	IF (233)	AG (206)	CACHE (1824)	EX (482)	ADA (320)	MISC (906)	**Weighted total (3971)**
Latent Errors	*0.00%*	*65.58%*	*4.91%*	*19.38%*	*95.41%*	**15.80% (±0.91%)**	*2.48%*	*32.82%*	*14.65%*	*38.32%*	*66.51%*	*18.14%*	**22.73% (±0.91%)**
Overwritten Errors	*0.00%*	*0.00%*	*47.64%*	*57.82%*	*4.00%*	**41.22% (±2.04%)**	*45.05%*	*35.75%*	*38.95%*	*41.55%*	*33.49%*	*69.53%*	**45.99% (±1.18%)**
Total (Non Effective Errors)	***0.00%***	***65.58%***	***52.56%***	***77.20%***	***99.40%***	**57.02% (±2.04%)**	***47.52%***	***68.57%***	***53.60%***	***79.87%***	***100.00%***	***87.67%***	**68.72% (±1.12%)**
Exception 1 (Bus Error)	*0.00%*	*0.00%*	*0.00%*	*0.00%*	*0.00%*	**0.00%**	*0.00%*	*0.00%*	*0.00%*	*0.14%*	*0.00%*	*0.16%*	**0.05% (±0.04%)**
Exception 2 (Address Error)	*36.95%*	*3.34%*	*21.17%*	*1.63%*	*0.00%*	**17.93% (±1.67%)**	*15.02%*	*2.94%*	*19.70%*	*0.36%*	*0.00%*	*3.30%*	**10.88% (±0.84%)**
Exception 3 (Data Error)	*0.00%*	*0.00%*	*0.00%*	*2.61%*	*0.00%*	**0.10% (±0.05%)**	*0.00%*	*0.00%*	*0.00%*	*1.29%*	*0.00%*	*0.00%*	**0.16% (±0.07%)**
Exception 4 (Instruction Error)	*58.62%*	*0.45%*	*1.70%*	*0.00%*	*0.00%*	**2.19% (±0.53%)**	*4.62%*	*0.17%*	*1.90%*	*0.50%*	*0.00%*	*0.75%*	**1.38% (±0.31%)**
Exception 5 (Jump Error)	*0.00%*	*0.00%*	*0.59%*	*0.00%*	*0.00%*	**0.48% (±0.31%)**	*0.50%*	*0.52%*	*1.10%*	*0.43%*	*0.00%*	*0.20%*	**0.66% (±0.22%)**
Exception 8 (Constraint Check)	*0.49%*	*3.43%*	*10.29%*	*7.82%*	*0.00%*	**8.88% (±1.24%)**	*12.54%*	*3.45%*	*9.95%*	*3.58%*	*0.00%*	*2.83%*	**6.57% (±0.65%)**
Exception 9 (Access Check)	*0.00%*	*0.45%*	*0.13%*	*0.16%*	*0.00%*	**0.14% (±0.15%)**	*0.00%*	*1.38%*	*0.05%*	*0.00%*	*0.00%*	*0.04%*	**0.10% (±0.07%)**
Exception 10 (Storage Error)	*0.00%*	*25.56%*	*0.00%*	*0.00%*	*0.00%*	**1.66% (±0.17%)**	*4.79%*	*21.24%*	*0.10%*	*0.72%*	*0.00%*	*0.31%*	**1.59% (±0.22%)**
Exception 11 (Overflow)	*0.00%*	*0.00%*	*0.00%*	*0.00%*	*0.00%*	**0.00%**	*0.17%*	*0.00%*	*0.00%*	*0.50%*	*0.00%*	*0.04%*	**0.08% (±0.05%)**
Exception 12 (Underflow)	*0.00%*	*0.00%*	*0.00%*	*0.00%*	*0.00%*	**0.00%**	*0.00%*	*0.00%*	*0.00%*	*0.29%*	*0.00%*	*0.00%*	**0.03% (±0.03%)**
Exception 13 (Division Check)	*0.00%*	*0.00%*	*0.00%*	*0.00%*	*0.00%*	**0.00%**	*0.00%*	*0.00%*	*0.00%*	*0.50%*	*0.00%*	*0.00%*	**0.06% (±0.04%)**
Exception 14 (Illegal Operation)	*0.49%*	*1.17%*	*0.79%*	*0.33%*	*0.00%*	**0.73% (±0.36%)**	*1.49%*	*1.21%*	*0.85%*	*0.57%*	*0.00%*	*0.35%*	**0.69% (±0.21%)**
Control Flow Errors	*2.96%*	*0.00%*	*0.66%*	*0.00%*	*0.00%*	**0.57% (±0.33%)**	*11.06%*	*0.35%*	*0.65%*	*9.53%*	*0.00%*	*1.88%*	**2.55% (±0.31%)**
Other Errors	*0.00%*	*0.00%*	*0.13%*	*0.00%*	*0.00%*	**0.11% (±0.15%)**	*0.17%*	*0.00%*	*0.00%*	*0.29%*	*0.00%*	*0.39%*	**0.13% (±0.07%)**
System Failure	*0.49%*	*0.00%*	*11.99%*	*10.26%*	*0.60%*	**10.18% (±1.33%)**	*2.15%*	*0.17%*	*12.10%*	*1.43%*	*0.00%*	*2.08%*	**6.34% (±0.68%)**
Total (Effective Errors)	***100.00%***	***34.42%***	***47.44%***	***22.80%***	***0.60%***	**42.98% (±2.04%)**	***52.48%***	***31.43%***	***46.40%***	***20.13%***	***0.00%***	***12.33%***	**31.28% (±1.12%)**
Coverage	***99.51%***	***100.00%***	***88.01%***	***89.74%***	***99.40%***	**89.82% (±1.33%)**	***97.85%***	***99.83%***	***87.90%***	***98.57%***	***100.00%***	***97.92%***	**93.66% (±0.68%)**
Total no. of faults injected	***203***	***1107***	***1526***	***614***	***1176***	**4626**	***606***	***579***	***2000***	***1396***	***872***	***2547***	**8000**

Table 5: Results of campaign 3 and 4a

Reason		IF	AG	CACHE	EX	ADA	Total	Percentage of total number of faults injected
Classification	Observability lower for FIMBUL (overwritten instead of latent)	0	0	155	12	55	222	4.80%
	Observability lower for FIMBUL (result late (System Failure) instead of latent)	0	0	0	5	7	12	0.26%
	Observability lower for FIMBUL (Control Flow Error instead of Other Error)	0	0	3	0	0	3	0.06%
	Simulation ended prematurely using MEFISTO-C (result late (System Failure) instead of detected error)	0	0	1	0	0	1	0.02%
Implementation	RET instruction implementation differs	0	2	12	0	0	14	0.30%
	UDE* signal implementation differs	0	0	0	1	0	1	0.02%
Tool	Fault injection time differs	1	0	13	4	0	18	0.39%
	Prefetch buffer reset after fault injection using FIMBUL	101	0	0	0	0	101	2.18%
	UM buffer emptied at slightly different times	0	0	9	0	0	9	0.19%
	Unknown	1	1	3	0	0	5	0.11%
	Total number of discrepancies	**103**	**3**	**196**	**22**	**62**	**386**	**8.34%**

Table 7: Reasons for discrepancies between campaign 3 and 4b

6 Conclusions

We presented a new tool for scan chain implemented fault injection (SCIFI) called FIMBUL. The tool was used for evaluation of the error detection mechanisms included in the Thor microprocessor. Since Thor has been designed specifically for space applications, our primary goal was to study the effects of single event upsets caused by ionizing particles in the space environment. To model the effects of these faults FIMBUL injected single bit-flips into internal state elements that are accessible via Thor's test access port (TAP). The bit-flips were injected randomly using a uniform fault distribution.

To assess the accuracy of the SCIFI technique we also conducted simulation-based fault injection experiments using the MEFISTO-C tool and a detailed VHDL model of Thor. MEFISTO-C can inject faults into all of the 3971 state elements of Thor, while FIMBUL can inject faults into 2250 state elements.

Only minor differences in the distribution of detected errors were observed for the two techniques. The results show a total error coverage of 90%-94% (depending on the workload) using FIMBUL compared to 94%-96% using MEFISTO-C. To verify that the two fault injection tools produced the same results for identical faults, we conducted special fault injection campaigns using identical fault sets. Only about 3% of the faults injected gave dissimilar results due to different operation of the tools. Most of these discrepancies were caused by a reset of Thor's prefetch buffer when using FIMBUL.

The most efficient error detection mechanisms of the Thor processor were the address error-, variable constraint check- and control flow checking mechanisms. The results also indicate that adding a parity-bit to the data cache of Thor would significantly increase the overall error coverage.

Our experiments showed that the SCIFI technique can be more than 100 times faster than simulation-based fault injection, and yet produce very similar results. Thus we have demonstrated that SCIFI can be a cost-effective and accurate technique for evaluating error handling mechanisms. Our results suggest that the SCIFI technique gives a more pessimistic coverage estimation than the simulation-based technique. However, additional fault injection experiments are needed to corroborate that this observa-

Fault injection tool	FIMBUL						MEFISTO-C					
Part of CPU fault injected (no. of state elements)	IF (30)	AG (146)	CACHE (1824)	EX (90)	ADA (160)	**Weighted total (2250)**	IF (30)	AG (146)	CACHE (1824)	EX (90)	ADA (160)	**Weighted total (2250)**
Latent Errors	*0.00%*	*65.58%*	*4.91%*	*19.38%*	*95.41%*	**15.80% (±0.91%)**	*0.00%*	*65.58%*	*15.40%*	*19.71%*	*100.00%*	**24.64% (±1.49%)**
Overwritten Errors	*0.00%*	*0.00%*	*47.64%*	*57.82%*	*4.00%*	**41.22% (±2.04%)**	*9.36%*	*0.00%*	*37.22%*	*59.93%*	*0.00%*	**32.70% (±1.97%)**
Total (Non Effective Errors)	***0.00%***	***65.58%***	***52.56%***	***77.20%***	***99.40%***	**57.02% (±2.04%)**	***9.36%***	***65.58%***	***52.62%***	***79.64%***	***100.00%***	**57.34% (±2.04%)**
Exception 1 (Bus Error)	*0.00%*	*0.00%*	*0.00%*	*0.00%*	*0.00%*	**0.00%**	*0.00%*	*0.00%*	*0.00%*	*0.00%*	*0.00%*	**0.00%**
Exception 2 (Address Error)	*36.95%*	*3.34%*	*21.17%*	*1.63%*	*0.00%*	**17.93% (±1.67%)**	*35.47%*	*3.16%*	*19.86%*	*1.30%*	*0.00%*	**16.83% (±1.63%)**
Exception 3 (Data Error)	*0.00%*	*0.00%*	*0.00%*	*2.61%*	*0.00%*	**0.10% (±0.05%)**	*0.00%*	*0.00%*	*0.00%*	*2.44%*	*0.00%*	**0.10% (±0.05%)**
Exception 4 (Instruction Error)	*58.62%*	*0.45%*	*1.70%*	*0.00%*	*0.00%*	**2.19% (±0.53%)**	*13.30%*	*0.45%*	*1.70%*	*0.00%*	*0.00%*	**1.59% (±0.53%)**
Exception 5 (Jump Error)	*0.00%*	*0.00%*	*0.59%*	*0.00%*	*0.00%*	**0.48% (±0.31%)**	*0.00%*	*0.18%*	*1.18%*	*0.00%*	*0.00%*	**0.97% (±0.44%)**
Exception 8 (Constraint Check)	*0.49%*	*3.43%*	*10.29%*	*7.82%*	*0.00%*	**8.88% (±1.24%)**	*39.90%*	*3.52%*	*10.88%*	*7.82%*	*0.00%*	**9.89% (±1.27%)**
Exception 9 (Access Check)	*0.00%*	*0.45%*	*0.13%*	*0.16%*	*0.00%*	**0.14% (±0.15%)**	*0.00%*	*0.45%*	*0.07%*	*0.16%*	*0.00%*	**0.09% (±0.11%)**
Exception 10 (Storage Error)	*0.00%*	*25.56%*	*0.00%*	*0.00%*	*0.00%*	**1.66% (±0.17%)**	*0.00%*	*25.47%*	*0.00%*	*0.00%*	*0.00%*	**1.65% (±0.17%)**
Exception 11 (Overflow)	*0.00%*	*0.00%*	*0.00%*	*0.00%*	*0.00%*	**0.00%**	*0.00%*	*0.00%*	*0.00%*	*0.00%*	*0.00%*	**0.00%**
Exception 12 (Underflow)	*0.00%*	*0.00%*	*0.00%*	*0.00%*	*0.00%*	**0.00%**	*0.00%*	*0.00%*	*0.00%*	*0.00%*	*0.00%*	**0.00%**
Exception 13 (Division Check)	*0.00%*	*0.00%*	*0.00%*	*0.00%*	*0.00%*	**0.00%**	*0.99%*	*0.00%*	*0.00%*	*0.00%*	*0.00%*	**0.01% (±0.02%)**
Exception 14 (Illegal Operation)	*0.49%*	*1.17%*	*0.79%*	*0.33%*	*0.00%*	**0.73% (±0.36%)**	*0.00%*	*1.17%*	*0.85%*	*0.33%*	*0.00%*	**0.78% (±0.38%)**
Control Flow Errors	*2.96%*	*0.00%*	*0.66%*	*0.00%*	*0.00%*	**0.57% (±0.33%)**	*0.99%*	*0.00%*	*0.66%*	*0.00%*	*0.00%*	**0.54% (±0.33%)**
Other Errors	*0.00%*	*0.00%*	*0.13%*	*0.00%*	*0.00%*	**0.11% (±0.15%)**	*0.00%*	*0.00%*	*0.00%*	*0.00%*	*0.00%*	**0.00%**
System Failure	*0.49%*	*0.00%*	*11.99%*	*10.26%*	*0.60%*	**10.18% (±1.33%)**	*0.00%*	*0.00%*	*12.19%*	*8.31%*	*0.00%*	**10.21% (±1.33%)**
Total (Effective Errors)	***100.00%***	***34.42%***	***47.44%***	***22.80%***	***0.60%***	**42.98% (±2.04%)**	***90.64%***	***34.42%***	***47.38%***	***20.36%***	***0.00%***	**42.66% (±2.04%)**
Coverage	***99.51%***	***100.00%***	***88.01%***	***89.74%***	***99.40%***	**89.82% (±1.33%)**	***100.00%***	***100.00%***	***87.81%***	***91.69%***	***100.00%***	**89.79% (±1.33%)**
Total no. of faults injected	***203***	***1107***	***1526***	***614***	***1176***	**4626**	***203***	***1107***	***1526***	***614***	***1176***	**4626**

Table 6: Results of campaign 3 and 4b

tion is valid also for other workloads. More research is also needed to investigate the use of SCIFI for other microprocessors and other types of circuits. In the near future we will extend the FIMBUL tool and add support for injection of permanent- and semi-permanent faults.

Acknowledgements

This work was partially supported by Saab Ericsson Space AB and the Swedish National Board for Industrial and Technical Development (NUTEK). We wish to express our gratitude to Magnus Legnehed, Stefan Asserhäll, Torbjörn Hult, and Roland Pettersson of Saab Ericsson Space AB for their support, valuable suggestions and technical assistance. We would also like to thank Prof. Jan Torin for his encouragement and many valuable comments. Thanks are also due Joakim Ohlsson and Marcus Rimén for their support concerning the MEFISTO-C tool.

References

[1] Iyer, R.K., "Experimental Evaluation," in *Special Issue of Proc. 25th Int. Symp. on Fault-Tolerant Computing (FTCS-25)*, Pasadena, CA, USA, 1995.

[2] J. Arlat, M. Aguera, L. Amat, Y. Crouzet, J.-C. Fabre, J.-C. Laprie, E. Martins and D. Powell, "Fault Injection for Dependability Validation — A Methodology and Some Applications", *IEEE Transactions on Software Engineering*, 16 (2), pp.166-82, February 1990.

[3] H. Madeira, M. Rela, F. Moreira and J. G. Silva, "RIFLE: A General Purpose Pin-level Fault Injector", in *Proc. 1st European Dependable Computing Conf. (EDCC-1)*, (Berlin, Germany), pp.199-216, Springer-Verlag, 1994.

[4] Z. Segall, T. Lin, "FIAT: Fault Injection Based Automated Testing Environment", in *Proc 18th Annual IEEE International Symposium on Fault-Tolerant Computing*, pp. 102-107, June 1988.

[5] G. Kanawati, N. Kanawati and J. Abraham, "FERRARI: A Tool for the Validation of System Dependability Properties", in *Proc 22nd Annual IEEE International Symposium on Fault-Tolerant Computing*, pp. 336-344, 1992.

[6] Carreira J., Madeira H. and Silva J. G., "Xception: Software Fault Injection and Monitoring in Processor Functional Units", in *Proc 5th International Working Conference on Dependable Computing For Critical Applications (DCCA-5)*, pp. 135-149, September 1995.

[7] Saab Ericsson Space AB, *Microprocessor Thor, Product Information*, September 1993.

[8] Mahmood A., et al, "Concurrent Error Detection Using Watchdog Processors - A Survey", *Transactions on Computers*, vol. 37, No. 2, February 1988, pp. 160-174.

[9] Saab Ericsson Space AB, *Workstation Board Specification*, Doc. no. TOR/TNOT/0015/SE, February 1993.

[10] Jenn E., Arlat J., Rimén M., Ohlsson J., Karlsson J., "Fault Injection into VHDL Models: The MEFISTO Tool", in *Proc. 24th Annual IEEE International Symposium on Fault-Tolerant Computing*, FTCS-24, pp. 66-75, Austin, TX, USA, June 1994.

[11] Chillarege R., Iyer R. K., "Measure-Based Analysis of Error Latency", *Transactions on Computers*, vol. C-36, No. 5, May 1987, pp. 529-537.

[12] Czeck E. W., Siewiorek D. P., "Observations on the Effects of Fault Manifestation as a Function of Workload", *Transactions on Computers*, vol. 41, No. 5, May 1992, pp. 559-565.

[13] Gunneflo U., et al., "Evaluation of Error Detection Schemes Using Fault Injection by Heavy-ion Radiation", in *Proc. 19th Annual IEEE International Symposium on Fault-Tolerant Computing*, Chicago, IL, USA, June, 1989, pp. 340-347.

[14] Saab Ericsson Space AB, *Users Guide for ODEN Ada Compiler System*, Doc. no. TOR/TNT/0010/SE, April 1996.

[15] Powell D., et al., "On Stratified Sampling for High Coverage Estimations", in *Dependable Computing - EDCC-2, Lectures Notes in Computer Science, vol. 1150*, Springer-Verlag, Berlin, Germany, 1996, pp. 37-54.