Considering Workload Input Variations in Error Coverage Estimation

Peter Folkesson and Johan Karlsson

Laboratory for Dependable Computing, Department of Computer Engineering,
Chalmers University of Technology, S-412 96 Göteborg, Sweden
{peterf, johan}@ce.chalmers.se

Abstract. The effects of variations in the workload input when estimating error detection coverage using fault injection are investigated. Results from scan-chain implemented fault injection experiments using the FIMBUL tool on the Thor microprocessor show that the estimated error non-coverage may vary by more than five percentage units for different workload input sequences. A methodology for predicting error coverage for a particular input sequence based on results from fault injection experiments with another input sequence is presented. The methodology is based on the fact that workload input variations alter the usage of sensitive data and cause different parts of the workload code to be executed different number of times. By using the results from fault injection experiments with a chosen input sequence, the error coverage factors for the different parts of the code and the data are calculated. The error coverage for a particular input sequence is then predicted by means of a weighted sum of these coverage factors. The weight factors are obtained by analysing the execution profile and data usage of the input sequence. Experimental results show that the methodology can identify input sequences with high, medium or low coverage although the accuracy of the predicted values is limited. The results show that the coverage of errors in the data cache is preferably predicted using data usage based prediction while the error coverage for the rest of the CPU is predicted more favourably using execution profile based prediction.

1 Introduction

Since fault-tolerant computer systems are often used for protecting large investments and even lives, validating such systems before they are being put to use is imperative. Experimental validation methods such as fault injection have become particularly attractive for estimating the dependability of computer systems [8]. As fault-tolerant systems typically contain several mechanisms for detecting and handling errors in order to avoid system failures, it is of particular interest to measure the efficiency of these mechanisms, i.e., to measure the error detection coverage.

It is well known that the workload program has significant impact on the dependability measures obtained from fault injection experiments [4], [6]. Thus, the program must be carefully chosen when evaluating fault-tolerant systems. The ideal is to use the real program that will be used during actual operation of the system.

The results of fault injection experiments may also be affected by the workload input sequence [3]. Fault injection experiments conducted on a Motorola MVME162 board executing a parser evaluating algebraic expressions clearly show the impact of the input domain on estimated dependability measures [2]. Thus, both the program and the input sequence to the program must be considered when validating fault-tolerant systems.

We have used the FIMBUL tool [5] to investigate the effects of workload input variations on fault injection results. We show, for three different programs, that variations in input domain significantly affects the estimated error coverage. A solution to the problem of accurately estimating error coverage when the input domain varies could be to perform fault injection experiments for many input sequences. However, conducting fault injection experiments for a large number of input sequences is very time consuming. To speed up the validation process, a methodology for predicting the error coverage for a particular input sequence based on fault injection results obtained for another input sequence is presented.

The remainder of the paper is organized as follows. The experimental set-up used for investigating the effects of workload input variations on fault injection results is described in Sect. 2. In Sect. 3, results obtained from fault injection experiments clearly demonstrate that the input sequence affects the estimated error coverage. The methodology for predicting error coverage is presented in Sect. 4 and applied on three different workloads in Sect. 5. Finally, the conclusions of this study are given in Sect. 6 together with a discussion about the advantages/disadvantages of the methodology and the need for further research.

2 Experimental set-up

The target system for the fault injection experiments was the Thor microprocessor, a 32 bit stack oriented RISC processor developed and sold by Saab Ericsson Space AB [10]. The Thor processor is primarily intended for embedded systems in space applications and includes several advanced error detection mechanisms to support fault tolerance. Access to the internal logic of the CPU is provided via internal and boundary scan chains using a Test Access Port (TAP).

The FIMBUL (Fault Injection and Monitoring Using Built-in Logic) tool was used to conduct the fault injection experiments on Thor. FIMBUL is able to inject faults into the internal logic of Thor via scan-chain implemented fault injection (SCIFI). This technique can be more than 100 times faster than e.g. simulation based fault injection, and yet produce very similar results [5].

This section gives a short overview of Thor and FIMBUL, and a description of how they were configured during the experiments.

2.1 The Thor processor

The Thor processor uses a stack oriented instruction set architecture which gives compact machine code due to frequent use of an implicit operand addressing mode. A stack based architecture also provides a high locality of data references resulting in

high hit rates of the data cache and fast context switching since only the stack pointer needs to be updated.

A block diagram of the Thor chip is given in Fig. 1. A four stage instruction pipeline is found in the middle of the figure. The pipeline consists of an *Instruction Fetch* stage (*IF*), an *Address Generation* stage (*AG*), an *Operand Fetch* stage (*OF*) incorporating a 32 word (128 byte) direct mapped write-back data cache, 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, e.g., when the EX stage is performing a multicycle operation or waits for a write operation to slow memory.

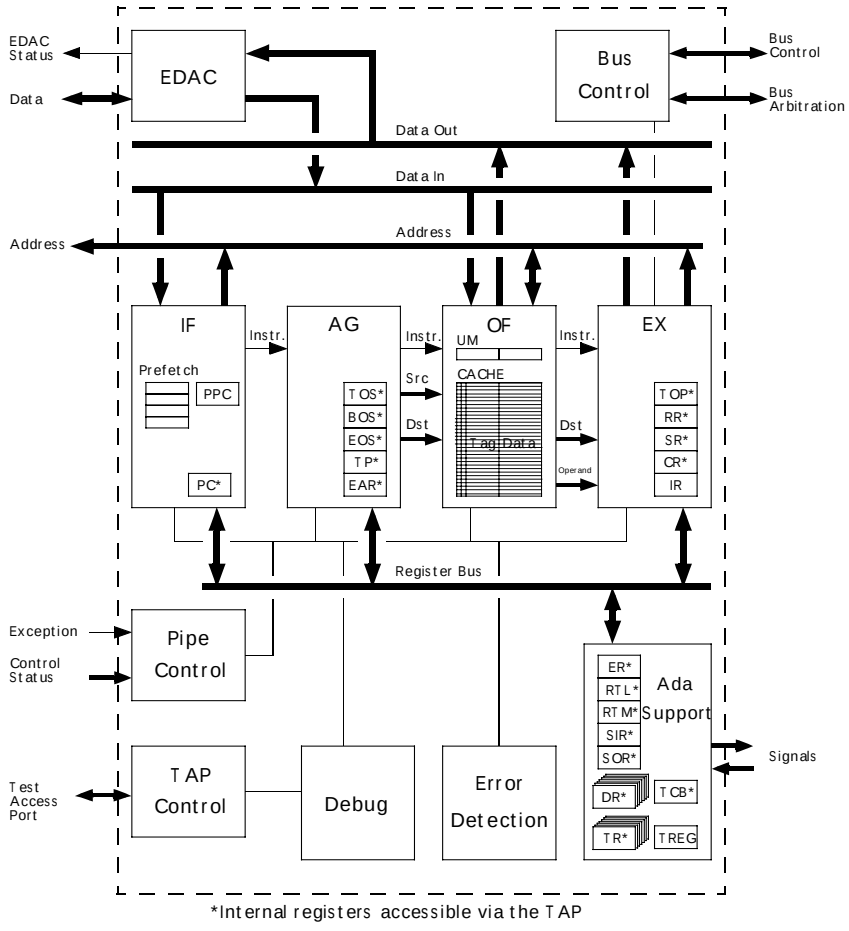


Fig. 1. Block diagram of the Thor processor

On-chip support for real-time processing specifically for the Ada language is available on Thor and is handled by the *Ada support* block. This includes task scheduling and dispatch, communication between tasks, time handling, accurate delays and fast interrupt handling.

The support for fault tolerance includes several internal error detection mechanisms. The error detection mechanisms are controlled and supervised by the *Error Detection* block and consist of *control flow checking*, *run-time checks*, *memory data checks* and *master/slave comparator operation*. In this study, we consider estimation of the error detection coverage of the run-time checks and control flow checking mechanism. The run-time checks include those which are commonly found in other microprocessors (division by zero checks etc.), as well as Thor specific checks such as index array bound checks. The control flow checking mechanism uses a signature monitoring scheme that checks the control flow of the program execution during run-time (see [9] for a general description of this technique).

The *TAP control* and *Debug* blocks implements the test and debug facilities. The TAP conforms to the IEEE 1149.1 standard for boundary scan [7]. It provides access to 101 chip-pin signals via a boundary scan-chain, and all the memory elements in the cache and 18 internal registers (marked with an asterisk in Fig. 1) via an internal scan-chain. The TAP also gives access to a debug scan-register, which allows the user to set breakpoints. When a breakpoint condition is fulfilled, the processor halts and the values of the memory elements in the scan-chains can be read or written via the TAP. This feature provides very powerful support for fault injection.

2.2 The FIMBUL tool

The test and debug facilities included in the Thor processor is used for fault injection by the FIMBUL tool. Transient bit-flip faults can be injected into any of the locations accessible by the boundary and internal scan-chains of the Thor CPU. The points in time for fault injection are chosen by programming the debug scan-register to halt the processor when a particular address is accessed.

The tool uses a Thor evaluation board [11], featuring the Thor CPU, memory circuits and I/O ports, installed on a Sun UNIX workstation using an SBus interface. All communication between FIMBUL and the Thor CPU is performed via UNIX device drivers. The software needed to communicate with Thor in order to inject faults and collect data is executed on the workstation. The workstation is also used for workload generation and data analysis.

There are three phases involved in conducting fault injection campaigns using FIMBUL: the *set-up*, *fault injection* and *analysis phases*. The workload chosen for fault injection experiments is analysed in the set-up phase to produce configuration data for the experiments. The configuration data contains all the information needed to perform fault injection experiments, e.g. when and where to inject faults, the number of faults to inject and which workload to use. The configuration data also determines the operation mode for the FIMBUL fault injection module. There are four operation modes: *normal*, *normal reference*, *detail* and *detail reference*. In normal mode, the CPU state, i.e. the contents of the Thor scan-chains, is logged when a fault injection experiment terminates. In detail mode, the CPU state is logged after each

instruction executed since fault injection allowing the error propagation to be analysed in detail. No fault injection is performed in the corresponding reference modes to obtain data from fault free executions.

The configuration data is read and interpreted by FIMBUL in the fault injection phase. After initializing the Thor evaluation board, the workload is downloaded and FIMBUL starts operating in reference mode. Then, the first fault injection experiment is performed according to the information given in the configuration data. Fault injection is made by programming the debug-scan register to halt the processor when an address given in the configuration data is accessed. The contents of the boundary and internal scan-chains of Thor are then read and the bits stated in the configuration data are inverted and written back to the CPU. The workload execution is then resumed (the CPU state is now logged after each instruction executed since fault injection if FIMBUL operates in detail mode). Workload execution continues until a time-out value has been reached, an error has been detected or the program finishes its execution, whichever comes first. Then, the CPU state is logged and the Thor evaluation board is reinitialized and another fault injection experiment begins.

The data logged in the fault injection phase is analysed in the analysis phase to obtain dependability measures about the target system. The dependability measures obtained include the percentage of detected, latent and overwritten errors as well as the percentage of faults leading to incorrect results, i.e. the error non-coverage.

2.3 Experiments conducted

FIMBUL executed on a 50 MHz Sun Sparc Classic workstation equipped with a Thor evaluation board. The evaluation board used 512 KB RAM and was clocked with 12.5 MHz. The workstation used its own 2 GB disk space for storing results and managing the experiments. FIMBUL operated in normal mode since most of the experiments were focused on measuring error non-coverage. Using this set-up, FIMBUL injected approximately one fault every two seconds. The detail reference mode was sometimes used for further investigation, e.g. studying the impact of input domain variations on workload execution.

The faults injected were single bit-flips in the internal state elements (latches and flip flops). Single bit-flip faults were selected to model the effects of Single Event Upsets, which are common in the space environment. The FIMBUL tool is capable of injecting faults into 2250 of the 3971 internal state elements of Thor (see Fig. 1). The data cache of Thor contains 1824 of the injectable state elements while 426 injectable state elements are located in the other parts of Thor, hereafter called *register part* or *registers*. 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.

Three different workloads written in the Ada language were considered. One is an implementation of the Quicksort algorithm. It uses recursion to sort an array 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 close to a few thousand clock cycles depending on the initial sort

order of the elements. The second workload is an implementation of the non-recursive Shellsort algorithm. Again, a seven element array of the Ada predefined type float is sorted. The size of the workload is 600 bytes and it utilizes 29 of the 80 instructions of Thor. Execution time is a bit lower than for the Quicksort workload, but again varies depending on the initial sort order of the elements. The third workload implements an algorithm solving the Towers of Hanoi puzzle. The size is 1724 bytes and it utilizes 27 different instructions. Execution time varies around five thousand clock cycles depending on the input sequence used.

Seven elements were sorted by the sort workloads in the experiments. Twenty-five initial permutations of these seven elements were chosen among all the 5040 possible permutations as input sequences for the workloads. These are alphabetically denoted *A*-*Y*, where the sort order varies from *A*="the seven elements are already sorted" to *Y*="the elements are sorted backwards". Seven different input sequences were used for the Towers of Hanoi workload, see Sect. 5.3.

3 Results for the Quicksort workload

The results for the Quicksort workload, obtained when injecting 4000 faults for each of the input sequences *A*-*Y*, are given in Fig. 2. The observed error non-coverage is shown for each input sequence. Major differences can be observed. Only 7.80% non-covered errors were obtained for input sequence *N* while 13.67% non-covered errors were obtained for input sequence *Y* with the corresponding 95% confidence intervals for the two sequences of ±0.94% and ±1.10% respectively. Thus, error non-coverage was estimated with a difference of more than five percentage units. These results clearly demonstrate that the input sequence affects the error detection coverage.

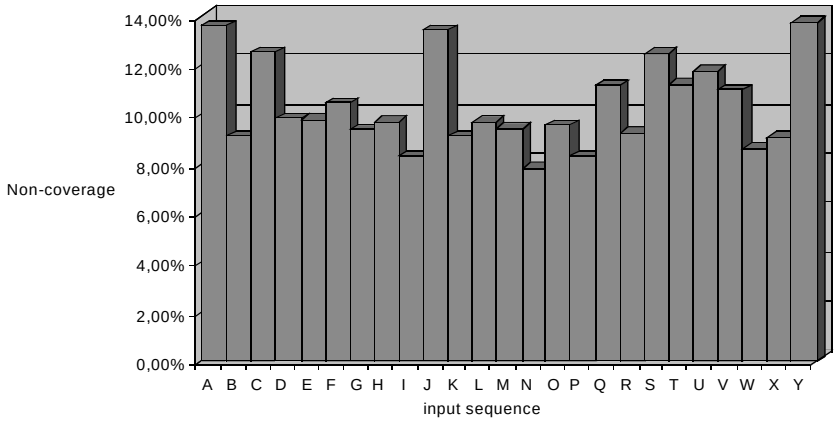


Fig. 2. Error non-coverage for Quicksort using different input sequences.

4 A methodology for predicting error coverage

The goal of a fault injection experiment is usually to provide a single error coverage factor which can be used in an analytical model for calculating the reliability, availability or safety of the target system. Such a single measure of error detection coverage, *c*, can be obtained as a weighted sum of the coverage factors obtained for different input sequences:

$$c = \sum_{i=1}^{n} c_i \cdot w_i \tag{1}$$

where c_i is the coverage estimated for input sequence *i* and w_i is the weight factor for input sequence *i*. Clearly w_i is the probability of occurrence for input sequence *i* during real operation.

There are several practical problems in estimating the coverage factor this way. First, the probability distribution of the input sequences must be established, which could be a difficult task for many applications. Second, the number of input sequences could be extremely large and it would therefore be impossible to perform fault injection experiments to estimate the coverage factor for each input sequence.

A practical solution to these problems would be to use a manageable number of input sequences which, for example, are selected based on educated guesses about the input sequence distribution. Even if the number of input sequences is reduced to, say, between 10 and 100, it is quite time consuming to conduct fault injection experiments to estimate the error coverage (or non-coverage) for every input sequence. To speed up this process, we propose a methodology for predicting the error coverage for a particular input sequence based on the results from fault injection experiments with another *base-* or *reference-*, input sequence. Thus, the goal of our research is to find efficient ways to estimate the c_i values in (1). Estimation of the weight factors, w_i, is another problem which is not addressed in this paper.

Two different techniques for predicting the error coverage are proposed. One is based on the fact that workload input variations cause different parts of the workload code to be executed different number of times. The other takes into account that the input variations alter the usage of data.

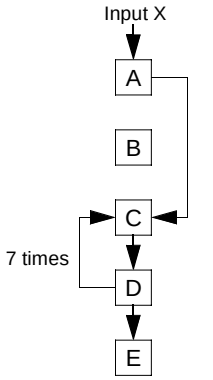
4.1 Execution profile based prediction

One reason for the observed coverage variations is that the *execution profile* of a program varies for different input sequences. A program can be divided into basic blocks [1], and each basic block is executed a different number of times depending on the input sequence. An example of how the workload execution profile may vary for two different input sequences is given in Fig. 3.

Let $P_{nce,i}$ denote the probability that a fault results in a non-covered error, given that the fault is activated during execution of basic block *i*. It is reasonable to assume that the probabilities $P_{nce,a}$ and $P_{nce,b}$ for two basic blocks *a* and *b* varies, but that the probability $P_{nce,i}$ is constant for each basic block *i* for all input sequences since the

activity of the system during execution of the basic block is the same regardless of input sequence used, i.e. the same instructions are always executed.

Assume that $P_{nce,C} > P_{nce,n}$ for all blocks $n \in \{A, B, D, E\}$ in Fig. 3. The non-coverage is then higher for a system that processes input *X* than input *Y* (assuming faults activated with equal probability for all points in time during the execution of the workload) since the proportion of the time spent executing block *C* is 14*8/(7*1+3*0+14*8+9*8+3*1)*100%=58% for input *X* and only 14*3/(7*1+3*1+14*3+9*3+3*1)*100%=53% for input *Y*.



Block	No. of clock cycles	No. of times executed for input X	No. of times executed for input Y
A	7	1	1
B	3	0	1
C	14	8	3
D	9	8	3
E	3	1	1

Fig. 3. Workload execution profile varies for different input data.

The execution profile based prediction technique calculates the predicted non-coverage $\bar{c}_{ep}$ for a particular input sequence *p* using the following equation:

$$\bar{c}_{ep} = \sum_{i=1}^{n} P_{nce,i} \cdot w_{ep,i} \tag{2}$$

where the basic blocks of the workload are numbered 1 to *n*. $P_{nce,i}$ is estimated using fault injection experiments for the base input sequence as

$$\hat{P}_{nce,i} = \frac{n_{nce,i}}{n_{e,i}} \tag{3}$$

where $n_{nce,i}$ is the observed number of faults activated when block *i* is executing resulting in non-covered errors, and $n_{e,i}$ is the total number of faults activated during execution of block *i*.

$w_{ep,i}$ is the weight factor for block *i*, i.e. the proportion of faults activated during execution of block *i* for input sequence *p* out of the total number of faults activated for input sequence *p*. If all faults are assumed to be activated with an equal probability for all points in time, the weight factor can be estimated as the proportion of the whole execution time spent executing block *i*, for input sequence *p*. Assume that block *i* executes for k_i clock cycles that the workload executes for l_p clock cycles for input

sequence *p*. $w_{ep,i}$ is then calculated as $w_{ep,i} = \frac{k_i}{l_p} \cdot x_{p,i}$ where $x_{p,i}$ is the number of executions of block *i* for input sequence *p*.

4.2 Data usage based prediction

Another reason for variations in error detection coverage is that different input sequences lead to different usage of data. (We here use the term data in a generic sense covering all types of data used by a program, including the program counter, hardware and user stack pointers, pointers used by the application program, application data, etc.) Error detection coverage clearly varies for different classes of data items. For example, errors in application data are less likely to affect control flow or memory access behaviour, than errors affecting application pointers or the program counter. Thus, when error detection relies on control flow checking and memory access checking, the error detection coverage is lower for errors that affect application data compared to errors that affect application pointers or the program counter. The amount of application data used by a program varies for different input sequences, which leads to variations in error detection coverage.

Based on these observations, a technique for predicting error coverage, or non-coverage, for a particular input sequence can be proposed. The different types of data items used by a program are divided into *n* classes, such that the error detection coverage is similar for the data items in a given class. As an approximation, we assume that the error detection coverage is the same for all data items in each class. Clearly, the classification must be made such that this approximation is valid.

The data usage based prediction technique calculates the predicted non-coverage $\bar{c}_{dp}$ for a particular input sequence *p* using the following equation:

$$\bar{c}_{dp} = \sum_{i=1}^{n} P_{ncd,i} \cdot w_{dp,i} \tag{4}$$

where the data classes are numbered 1 to *n*. $P_{ncd,i}$ is the probability that a fault in a data item in class *i* leads to a non-covered error, estimated using the fault injection experiments for the base input sequence as

$$\hat{P}_{ncd,i} = \frac{n_{ncd,i}}{n_{d,i}} \tag{5}$$

where $n_{ncd,i}$ is the observed number of faults injected in data items in class *i* resulting in non-covered errors, and $n_{d,i}$ is the total number of faults injected into data items in class *i*. The technique relies on the assumption that $P_{ncd,i}$ is approximately constant for different input sequences.

$w_{dp,i}$ is the weight factor for data class *i* for input sequence *p*. The weight factor is calculated as the percentage of the fault space $F = L \times T$ (see Sect. 2.3) containing data in class *i* during the execution using input sequence *p*. The weight factor is obtained by investigating the state space of the microprocessor during a single fault free run of the application program using input sequence *p*.

5 Applying the methodology

5.1 Predictions for the Quicksort workload

The FIMBUL tool is not always able to identify the basic block executing when a non-covered error is activated. We therefore assume, as an approximation, that this block is the same as the one executing when the fault is injected. This is a reasonable approximation for the register part since most of the faults injected in the register part resulting in non-covered errors have short activation latencies. However, the approximation is inadequate for the data cache since the fault injected cache lines are often used during execution of basic blocks other than the one executing when the fault is injected. Data usage based prediction is better suited for predicting the effects of faults injected into the data cache. To demonstrate how the two prediction techniques work for different parts of the processor, the results are presented separately for the register part and the data cache.

Table 1 shows the execution profiles of the Quicksort workload for the input sequences used in this study. Several differences can be observed between the profiles, e.g. block C93 is executed between one and nine times depending on the input sequence and block CF6 is not executed at all for input sequence *A* and up to five times using other input sequences.

For each basic block, the right-most column of Table 1 gives the estimated probability that a fault injected into the register part results in a non-covered error, given that the fault is activated during execution of the basic block. The predictions are based on 1500 faults injected into the register part during execution of each basic block for input sequence *M*. (The Quicksort workload contains 21 basic blocks, therefore a total of 31500 faults were injected). The last two rows of Table 1 show the error non-coverage predicted using equation (2) vs. the observed error non-coverage when injecting 4000 faults into the Thor register part for each input combination.

The results show that the execution profile based prediction technique correctly predicts that input sequences *F*, *J*, *T* and *Y* should have the highest error non-coverage and *A* the lowest. However, the predicted non-coverage is sometimes much higher than the observed non-coverage, e.g. for sequences *A*, *D* and *P*. One reason for the discrepancies between the predicted and observed non-coverage may be the low number of non-covered errors observed leading to a low confidence in the results. The 95% confidence intervals for the observed non-coverage values are around ±0.2%. Another reason may be that the assumption that the block executing when a fault is activated is the same as the one executing when the fault is injected, sometimes is inaccurate.

The results of execution profile based prediction on the register part is also shown in the left diagram in Fig. 4. The right diagram in Fig. 4 shows the results of data usage based prediction on the register part. Again, input sequence *M* was chosen as the base input sequence. The data used by the workload was divided into six classes. One class consists of the elements to be sorted, another of pointers to the elements to be sorted, three classes contains a particular value of the status register and the sixth class contains all other data.

Table 1. Execution profiles for different input sequences to the Quicksort workload. Thor registers fault injected.

Block start-address	No. of instr. (k_i)	No. of executions per input sequence ($x_{p,i}$)																									$\hat{P}nce, i$ input M
		A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	
C46	8	3	7	5	6	6	7	7	5	6	7	7	5	6	7	6	7	7	8	6	8	7	7	7	7	6	1.19%
C4F	3	3	7	5	6	6	8	7	5	6	7	8	5	6	7	6	7	7	8	6	8	7	7	7	7	6	2.01%
C52	26	3	6	4	5	5	5	6	5	5	4	6	4	5	6	5	6	5	6	5	6	4	5	6	6	3	0.07%
C6D	12	8	12	11	14	12	9	10	13	12	9	11	8	14	15	11	14	15	9	10	10	8	8	11	14	8	0.40%
C7B	9	5	4	5	7	5	1	2	6	4	2	3	2	5	6	4	6	5	1	3	1	2	1	3	4	2	0.07%
C87	12	8	13	10	11	11	14	14	11	15	8	13	15	16	16	12	14	14	14	12	15	9	12	14	15	8	0.47%
C93	9	5	5	4	4	4	6	6	4	7	1	5	9	7	7	5	6	4	6	5	6	2	5	6	5	2	0.00%
C9D	5	3	8	6	7	7	8	8	7	8	7	8	6	9	9	7	8	10	8	7	9	7	7	8	10	6	0.27%
CA3	21	3	7	5	6	6	7	7	5	6	7	7	5	6	7	6	7	7	8	6	8	7	7	7	7	6	2.04%
CBA	27	3	7	5	6	6	7	7	5	6	7	7	5	6	7	6	7	7	8	6	8	7	7	7	7	6	1.39%
CD8	8	3	7	5	6	6	7	7	5	6	7	7	5	6	7	6	7	7	8	6	8	7	7	7	7	6	0.07%
CE1	5	3	7	5	6	6	7	7	5	6	7	7	5	6	7	6	7	7	8	6	8	7	7	7	7	6	0.40%
CE7	8	3	6	5	5	6	7	7	5	6	7	7	5	6	7	6	6	6	8	6	7	7	7	6	7	6	0.00%
CF0	5	3	8	6	7	7	8	8	7	8	7	8	6	9	9	7	8	10	8	7	9	7	7	8	10	6	0.00%
CF6	4	0	2	2	2	2	3	2	2	3	3	2	2	4	3	2	2	5	2	2	3	3	2	2	4	3	0.07%
CFC	5	3	6	4	5	5	5	6	5	5	4	6	4	5	6	5	6	5	6	5	6	4	5	6	6	3	0.98%
D02	5	1	2	2	3	2	2	2	3	1	2	3	1	2	3	2	3	3	2	2	2	1	1	3	2	1	0.00%
D09	5	3	6	4	5	5	5	6	5	5	4	6	4	5	6	5	6	5	6	5	6	4	5	6	6	3	0.60%
D0F	5	1	3	1	1	2	2	3	1	3	1	2	2	2	2	2	2	1	3	2	3	2	3	2	3	1	0.00%
D16	4	3	6	4	5	5	5	6	5	5	4	6	4	5	6	5	6	5	6	5	6	4	5	6	6	3	0.00%
D24	13	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0.20%
Predicted non-coverage		0.60%	0.69%	0.66%	0.66%	0.68%	0.72%	0.69%	0.63%	0.65%	0.78%	0.70%	0.65%	0.63%	0.65%	0.68%	0.67%	0.68%	0.73%	0.68%	0.72%	0.78%	0.74%	0.69%	0.66%	0.78%	
Observed non-coverage		0.28%	0.61%	0.52%	0.49%	0.54%	0.70%	0.59%	0.58%	0.52%	0.68%	0.66%	0.51%	0.59%	0.64%	0.62%	0.47%	0.61%	0.60%	0.64%	0.74%	0.66%	0.64%	0.63%	0.68%	0.71%	

The results in Fig. 4 show that the data usage based prediction method fails to predict whether the error non-coverage is high or low for a particular input sequence, while the execution based prediction technique correctly identifies the input sequences with the highest, or lowest, error non-coverage (although the predicted value is sometimes too high, probably due to the reasons discussed above).

The results using data usage based prediction on the data cache are shown in Table 2. An analysis of the Quicksort workload shows that the data most sensitive to fault injection in the data cache are the elements to be sorted. Two data classes were therefore used for the data usage based prediction technique. One class contains the elements to be sorted while the other class contains all other data. Table 2 gives the weight factors for these two classes for each input sequence. The right-most column gives the estimated probability that a fault injected into a cache line containing the data in the class will lead to a non-covered error. The estimations are based on examination of 351 fault injected cache lines leading to non-covered errors (out of a total of 3019 injections) for input sequence *M*.

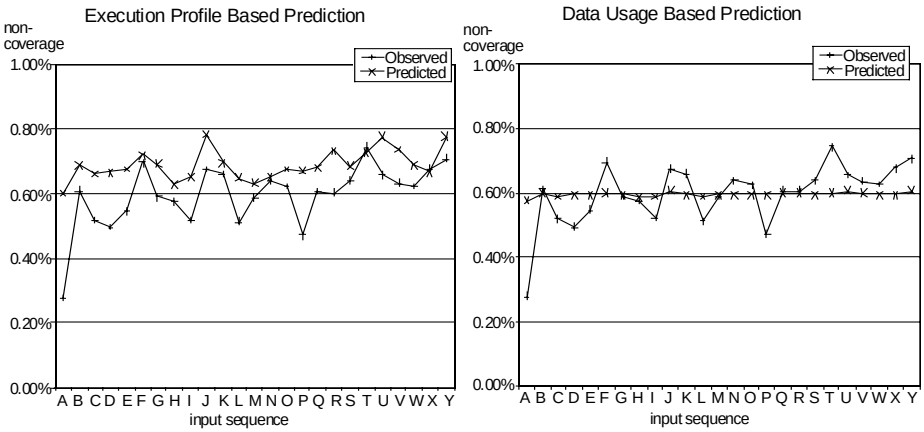


Fig. 4. Predicted vs. observed non-coverage for Quicksort. Thor registers fault injected.

Table 2 also gives the error non-coverage, predicted using equation (4), as well as the observed values estimated using 3000 faults injected into the data cache for each input sequence. A comparison shows that the predicted values are generally lower than the observed values, but that the relative differences between the predicted values correspond well to the relative differences between the observed values. This shows that the technique is capable of pointing out the input sequences with the highest (or lowest) error non-coverages.

Table 2. Weight factors and non-coverage for different input sequences to Quicksort. Thor data cache fault injected.

Critical data class	Weight factors for the data classes for each input sequence ($w_{dp,i}$) (%)																								$\hat{P}ncd, i$ input M
	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	
Elements to sort	24.72	20.20	23.81	20.81	21.49	22.00	20.93	21.06	21.25	24.95	21.06	20.68	20.78	20.93	21.37	20.34	22.73	21.28	22.99	22.41	23.75	22.28	21.10	20.47	26.37	43.05%
Other data	75.28	79.80	76.19	79.19	78.51	78.00	79.07	78.94	78.75	75.05	78.94	79.32	79.22	79.07	78.63	79.66	77.27	78.72	77.01	77.59	76.25	77.72	78.90	79.53	73.63	3.39%

Predicted non-cov. (%)	13.19	11.40	12.83	11.64	11.91	12.11	11.69	11.74	11.82	13.28	11.74	11.59	11.63	11.69	11.86	11.46	12.40	11.83	12.50	12.27	12.81	12.22	11.75	11.51	13.85
Observed non-cov. (%)	16.87	11.39	15.84	12.05	12.12	12.91	11.52	11.99	10.35	16.45	11.47	11.75	11.63	9.60	11.85	10.24	13.78	11.21	15.32	13.70	14.43	13.40	10.73	11.06	16.79

In addition, we have also used execution profile based prediction on the data cache. In this case, 1000 faults were injected for each basic block. (A total of 21000 faults were injected).

A comparison of the results obtained when using execution profile based and data usage based prediction on the data cache is shown in Fig. 5 (input sequence *M* was used as the base sequence for both prediction techniques). The diagrams show the observed vs. predicted error non-coverage for the different input sequences *A*-*Y* for each prediction technique. The observed values are estimated using 3000 faults injected for each input sequence.

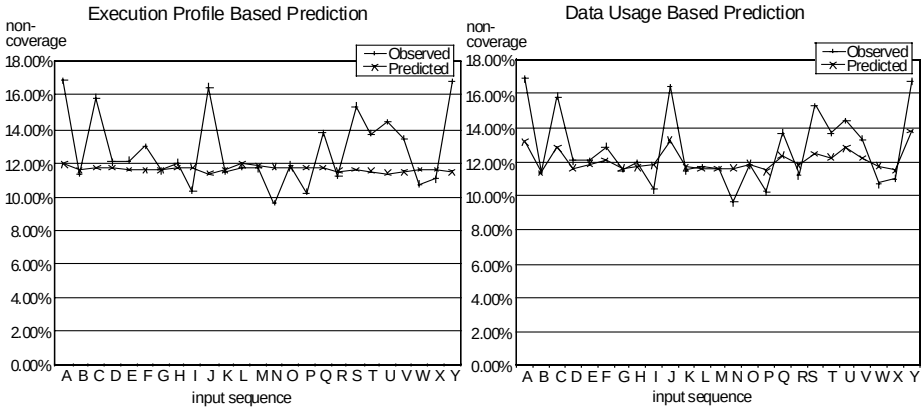


Fig. 5. Predicted vs. observed non-coverage for Quicksort. Thor data cache fault injected.

The results show that the data usage based prediction technique succeeds in identifying the input sequences with the highest non-coverage (*A*, *C*, *J*, *S*, *U* and *Y*) while the execution profile based prediction technique is clearly inadequate.

5.2 Predictions for the Shellsort workload

To verify the validity of the observations made in Sect. 5.1, a second workload was investigated. Fig. 6 shows the observed vs. predicted error non-coverage for various input sequences using the Shellsort workload when fault injecting the data cache. The observed error non-coverage is based on 2000 injected faults for each input sequence. Also in these experiments, the input sequence *M* was used as the base sequence for both prediction methods. The left diagram shows the results for execution profile based prediction based on a total of 4000 faults injected during execution of the whole workload. The right diagram shows the results for data usage based prediction based on examination of 744 injected faults leading to non-covered errors.

In contrast to the results for the Quicksort workload, the observed non-coverage varies much less for the different input sequences. Both prediction methods correctly point out input sequence *A* as having the lowest non-coverage but do not find any input sequence with an exceptionally high non-coverage.

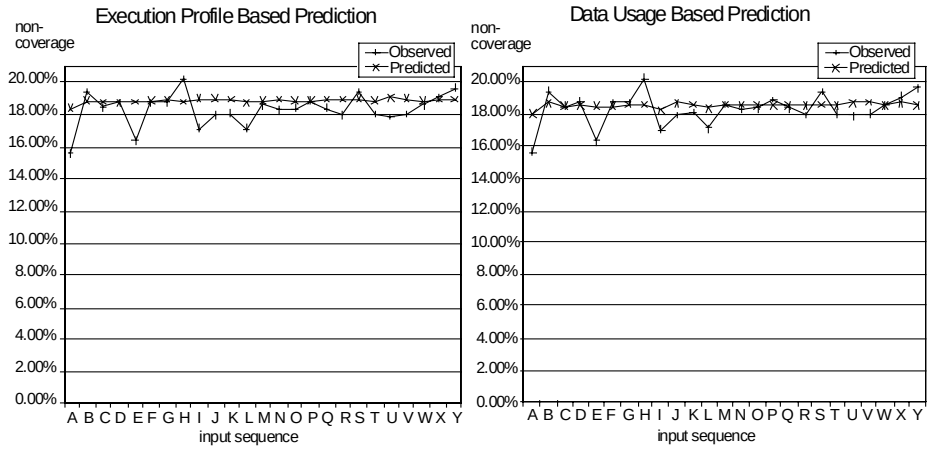


Fig. 6. Predicted vs. observed non-coverage for Shellsort. Thor data cache fault injected.

Fig. 7 shows the observed vs. predicted error non-coverage using the Shellsort workload when fault injecting the register part. The observed error non-coverage is based on 4000 faults injected for each input sequence. Execution profile based prediction is based on a total of 4000 injected faults. Data usage based prediction is based on examination of 435 injected faults leading to non-covered errors.

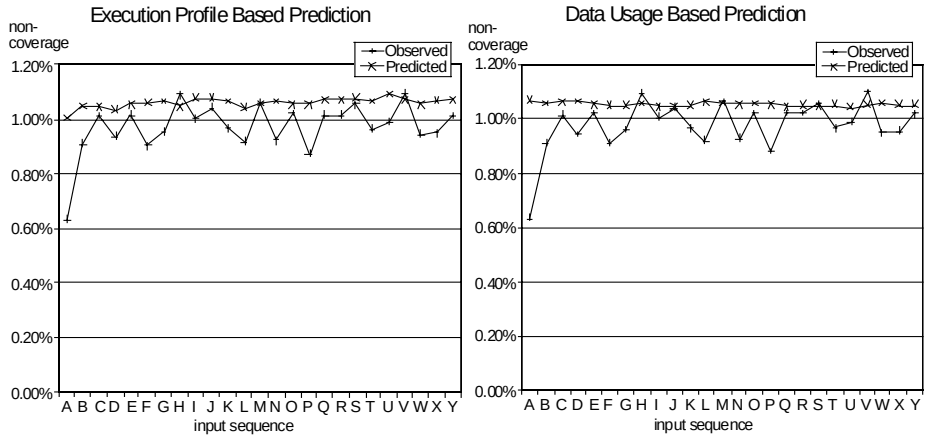


Fig. 7. Predicted vs. observed non-coverage for Shellsort. Thor registers fault injected.

The observed non-coverage differences are much smaller, just as for the data cache, and neither method finds any input sequence with an exceptionally high non-coverage. Only the execution profile based prediction method correctly identifies input sequence *A* as having the lowest error non-coverage.

5.3 Predictions for the Towers of Hanoi workload

The methodology described in Sect. 4 was applied on a third workload. The workload chosen is an implementation of an algorithm solving the Towers of Hanoi puzzle, see Fig. 8.

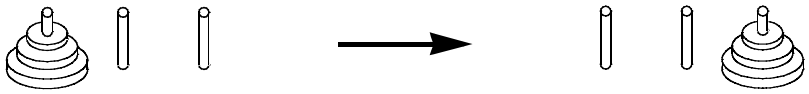


Fig. 8. Towers of Hanoi puzzle.

The purpose is to move the three discs of one tower stick to another by dropping the discs on any of the three available tower sticks. A larger disc may never be placed on top of a smaller one. The implementation uses a 3x3 floating point matrix containing non-zero values, representing the discs, and zero values, representing no disc. The locations of these values in the matrix determine the current configuration of the three towers. The algorithm recursively solves the problem of moving the discs to the right-most tower stick.

Seven different tower configurations (*A*-*G*) were used as the initial input sequences for the algorithm, see Fig. 9. Input sequence *A* was chosen as the base sequence.

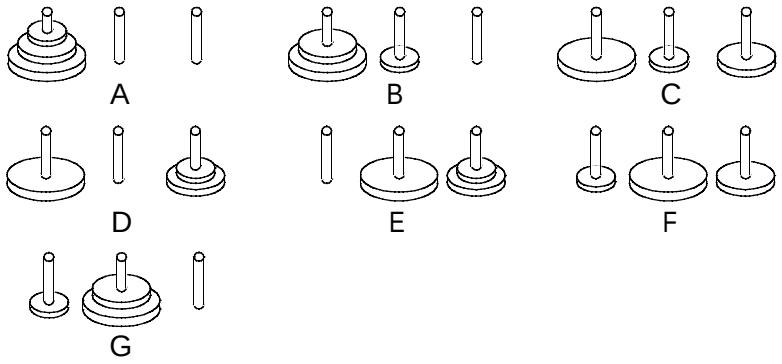


Fig. 9. Input sequences for Towers of Hanoi.

Fault injection results show that almost all of the non-covered errors were activated when fault injecting the cache. The faults injected into the register part are therefore neglected in these experiments and only the data usage based prediction technique was used. The critical data was found to be the zeros and floating point values representing the discs. 50 out of 167 non-covered errors were due to faults injected into zeros while 100 were due to faults injected into disc values out of a total of 10000 faults injected into the cache.

The error non-coverage predicted using equation (4) is given for each of the input sequences *A*-*G* in Table 3. The predicted error non-coverage is highest for input sequences *E* and *F*. The last row of Table 3 shows the error non-coverage estimated

using around 7500 faults injected into all available fault locations for each of the input sequences. The results show that the observed error non-coverage is indeed highest for input sequences *E* and *F* (2.55% vs. 2.71%).

Table 3. Predicted non-coverage for Towers of Hanoi. Only data cache considered.

	Input sequence						
	A	B	C	D	E	F	G
Zero value data usage	0.10668	0.09737	0.10985	0.11450	0.17536	0.13475	0.10909
Disc data (non-zero value) usage	0.05083	0.05524	0.05638	0.05395	0.05346	0.06404	0.06381
Other data usage	0.84249	0.84739	0.83377	0.83155	0.77118	0.80121	0.82710
Zero data non-coverage	0.047	-	-	-	-	-	-
Disc data non-coverage	0.197	-	-	-	-	-	-
Other data non-coverage	0.0020	-	-	-	-	-	-
Predicted non-coverage (data cache only)	1.67%	1.71%	1.79%	1.77%	2.03%	2.05%	1.93%
Observed non-coverage (whole CPU)	1.87%	1.38%	1.84%	1.67%	2.55%	2.71%	1.79%

6 Conclusion

This paper investigated the impact of workload input domain on fault injection results. The effects of varying the workload input when estimating error coverage using the FIMBUL fault injection tool on the Thor processor was examined. The results show that the estimated error non-coverage can vary more than five percentage units for different input sequences. This clearly demonstrate that the workload input domain should be considered and carefully chosen when estimating error coverage.

The problem of accurately estimating error coverage could be solved by performing several fault injection campaigns using different input sequences. Since this would be very time consuming, a methodology to speed up the process was presented. The methodology involves predicting error coverage for different input sequences based on fault injection experiments conducted using another input sequence.

Two different techniques for predicting error coverage were proposed. One technique uses the fact that workload input variations cause different parts of the workload code, i.e. basic blocks, to be executed. The other technique takes into account that workload input variations alter the usage of data. Prediction is made by calculating coverage factors for each basic block or sensitive data based on fault injection results for a single input sequence. The error coverage for a particular input sequence is then predicted by means of a weighted sum of these coverage factors. The weight factors are obtained by analysing either the execution profile or the data usage profile of the input sequence using a single fault free run of the program.

Another way of estimating error coverage could be to carry out fault injection campaigns where the input sequence is selected randomly for each injected fault. The selection should be made according to the input sequence distribution. However, many of the advantages associated with the prediction methodology would be lost. Identifying the basic blocks and data most sensitive to faults can be useful when

trying to improve the error coverage of the target system. The prediction based methodology can also be useful for identifying the input sequences with the lowest (worst case) error coverage. Also, there is no need to perform any new fault injection experiments if the input sequence distribution for the target system is altered.

Results show that error coverage for the register part of Thor, i.e. all parts of the CPU except the cache, is predicted more favourably using execution profile based prediction, while error coverage for the data cache is predicted more favourably using data usage based prediction. Since it is not always possible to identify the basic block which activates a non-covered error using FIMBUL, this block is assumed to be the same as the one executing when the fault is injected. This approximation requires the data usage based prediction technique to be used on the data cache since faults injected into the data cache have longer activation latencies than faults injected into the register part.

The results also show that although data usage based prediction fails to predict the actual error coverage, it can be useful for finding the input sequences with the most extreme error coverages, particularly when the error coverage differences are prominent. These input sequences could then be used in fault injection campaigns to estimate the real coverage values.

More research is needed to refine the methodology. In this paper, the input sequence used as the base input sequence was selected arbitrarily. It may be more favourable to select it according to certain criteria, e.g. an input sequence that causes all of the basic blocks of the workload code to be executed and as many critical data items to be used as possible. Perhaps several input sequences should be used as base input sequences for certain workload programs.

A method for identifying the critical data items for the data usage based prediction technique needs to be developed. The data belonging to various data classes were easily identifiable for the workloads used in this study since the same values were always used (albeit in a different sort order) for each input sequence, often enabling identification by simply studying the values. For other workloads the corresponding sensitive variables would sometimes have to be identified, e.g. by tracing the values and locations of the variables during execution of the workload.

Simulation based fault injection would probably allow the error coverage of the whole CPU to be predicted using execution profile based prediction only, since the higher observability available in the simulations should enable identification of the basic blocks activating the non-covered errors. This would completely eliminate the need for the data usage based prediction technique as well as the approximations made about which block that activates a non-covered error. The accuracy of the predicted values should thereby improve.

Although the methodology managed to identify input sequences with high, medium or low non-coverage for the workloads used in this study, more research is needed to determine whether the methodology is applicable to other workloads and other systems. The workloads used in this study had low complexity and performed similar floating point array manipulations but nevertheless provided a good starting point for our research. Larger programs can easily be investigated with FIMBUL since the SCIFI technique is very fast. Such programs typically consist of a number of subprograms, e.g. subroutines for sorting etc., and error coverage for each of these subroutines could probably be used to estimate the total error coverage.

In addition to these issues, the possibility to expand the methodology for estimating not only overall error coverage, but the coverage of specific error detection mechanisms as well, should also be investigated.

Acknowledgements

We wish to thank Magnus Legnehed, Stefan Asserhäll, Torbjörn Hult, and Roland Pettersson of Saab Ericsson Space AB as well as Prof. Jan Torin for their support of this research.

References

1. A. Aho, R.Sethi, and J. Ullman, "Compilers: Principles, Techniques and Tools", Reading, MA: Addison Wesley, 1985.
2. A. Amendola, L. Impagliazzo, P. Marmo, and F. Poli, "Experimental Evaluation of Computer-Based Railway Control Systems", in *Proc. 27th Int. Symp. on Fault-Tolerant Computing (FTCS-27)*, pp. 380-384, (Seattle, WA, USA) June 1997.
3. C. Constantinescu, "Using multi-stage and stratified sampling for inferring fault-coverage probabilities", *IEEE Transactions on Reliability*, 44 (4), pp. 632-639, 1995.
4. E. Czeck, and D. Siewiorek, "Observations on the Effects of Fault Manifestation as a Function of Workload", *IEEE Transactions on Computers*, 41 (5), pp. 559-566, May 1992.
5. P. Folkesson, S. Svensson, and J. Karlsson, "A Comparison of Simulation Based and Scan Chain Implemented Fault Injection", in *Proc. 28th Int. Symp. on Fault-Tolerant Computing (FTCS-28)*, pp. 284-293, (Munich, Germany) June 1998.
6. U. Gunneflo, J. Karlsson, and J. Torin, "Evaluation of Error Detection Schemes Using Fault Injection by Heavy-ion Radiation", in *Proc. 19th Int. Symp. Fault-Tolerant Computing (FTCS-19)*, pp. 340-347, 1989.
7. Test Technology Technical Committee of the IEEE Computer Society, *IEEE standard test access port and boundary-scan architecture*, USA 1990.
8. R. K. Iyer, "Experimental Evaluation", in *Special Issue of Proc. 25th Int. Symp. on Fault-Tolerant Computing (FTCS-25)*, Pasadena, CA, USA, 1995.
9. Mahmood A., et al, "Concurrent Error Detection Using Watchdog Processors - A Survey", *Transactions on Computers*, vol. 37, No. 2, February 1988, pp. 160-174.
10. Saab Ericsson Space AB, *Microprocessor Thor, Product Information*, September 1993.
11. Saab Ericsson Space AB, *Workstation Board Specification*, Doc. no. TOR/TNOT/0015/SE, February 1993.