Path-Based Error Coverage Prediction

Joakim Aidemark, Peter Folkesson, and Johan Karlsson
Department of Computer Engineering
Chalmers University of Technology
S-412 96 Göteborg, Sweden

{aidemark, peterf, johan}@ce.chalmers.se

Abstract

We present an analytical technique that uses fault injection data for estimating the coverage of concurrent error detection mechanisms in microprocessors. A major problem in such estimations is that the coverage depends on the program executed by the microprocessor as well as the input sequence to the program. We propose a method that predicts the error coverage for a specified input sequence based on fault injection data obtained for another input sequence. Our results show that post-injection analysis is a promising approach for reducing the cost of coverage estimation.

Key words: Error detection coverage, fault injection, analytical coverage estimation.

1. Introduction

Computer systems are increasingly being used in applications that require high dependability. To be confident that these systems deliver a correct service, they need to be validated. Both analytical and experimental techniques are required to fully validate a dependable system. Fault injection [10] is an important technique for experimental validation of dependable systems. It is used to study systems in the presence of faults to identify dependability weaknesses. It is also used for estimating the coverage of concurrent (on-line) error detection mechanisms. Error detection coverage is an important parameter for calculating reliability, availability or safety of a system [2].

In this paper we focus on the estimation of coverage for hardware implemented error detection mechanisms in microprocessors. For most software and hardware implemented error detection mechanisms, the coverage does not depend only on the implementation of the detection mechanism, but also on the program executed by the system [5], as well as the input sequence to the program [1], [4], [6], [7]. Therefore, it is important to use a representative *workload* when estimating error detection coverage by fault injection.

Achieving representativeness poses a problem since a system may have a variety of usage profiles depending on its mode of operation. Thus, fault injection experiments need to be conducted with different programs and/or input sequences.

In this paper we propose a technique aimed at reducing the cost of estimating coverage for different input sequences. The main idea is to analyse fault injection data obtained for a given input sequence, in order to estimate the coverage for another input sequence.

The analysis is based on an estimation of the coverage for each *basic block,* i.e., branch free intervals, in the program. The total coverage is then calculated as a weighted sum of these coverages, where the weight factors are determined by the basic block usage for the input sequence for which the estimation is made. This is an example of *post-injection analysis*.

Post-injection analysis and *pre-injection analysis* are two main classes of analysis techniques that have been proposed for reducing the cost of validating dependable systems by fault injection. Pre-injection analysis is made before any fault injection is performed to focus the injections to specific parts of the fault space [3], [8], [15]. In post-injection analysis, the results from fault injection experiments are used to predict the outcome of other experiments in order to speed up the validation process [5].

We have previously investigated two post-injection techniques for predicting error coverage, called Execution Profile Based Prediction and Data Usage Based Prediction [6]. These techniques predict error detection coverage from results obtained by scan-chain implemented fault injection. The new post-injection prediction technique presented in this paper, called Path-Based Error Coverage Prediction, is intended for simulation-based fault injection and utilizes the high observability available in simulations.

Path-Based Error Coverage Prediction can be used in the following way. Assume that we want to identify those input sequences that give extremely high, or low, error detection coverage among a given set of input sequences. (The set of input sequences of interest is typically determined by studying the usage of the evaluated system.) Instead of conducting a fault injection campaign for each input sequence, we conduct a single fault injection campaign for an arbitrary input sequence and then apply Path-Based Error Coverage Prediction to rank the input sequences according to error detection coverage. Once the ranking is done, we can conduct fault injection campaigns with the "interesting" input sequences in order to accurately determine the coverage figures of interest.

This procedure significantly reduces the time it takes to identify input sequences with extremely high or low coverage, as prediction is much faster than conducting fault injection campaigns. The prediction uses information about the usage of the program's basic blocks for the input sequence for which the prediction is made. This information is collected during a single fault-free simulation of the program execution. The time needed for the prediction is essentially determined by the time it takes to run this simulation. This time is approximately equal to the time it takes to make one fault injection experiment, i.e. to observe the effect of a single fault.

We have used this technique for estimating the coverage of several hardware implemented runtime checks included in the Thor microprocessor [14], which has been designed for use in highly dependable space applications. We injected single bit-flips inside the microprocessor to emulate the effects of Single Event Upsets.

The next section describes necessary definitions. In Section 3, the prediction technique is presented. The experimental set-up used for the evaluation of the prediction technique is described in Section 4. The results of the evaluation are presented in Section 5. Finally, the conclusions are given in Section 6.

2. Definitions

Errors can be classified into *non-effective* and *effective errors*. The non-effective errors are errors that are either latent or overwritten (they have no effect on the system behaviour). Effective errors correspond to errors, which are either detected by the error detection mechanisms of the system or lead to incorrect results being produced.

We define the *total error coverage* as the (un-conditional) probability that an error is either detected or non-effective. Some fault injection techniques, such as radiation based fault injection [12], do not allow estimation of the total error coverage because they do not provide adequate observability of non-effective errors.

However, such techniques can be used to estimate the *effective error coverage*, which is defined as the conditional probability that an error is detected given that the error is effective.

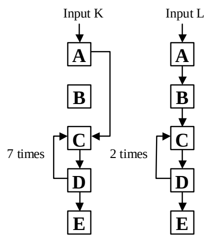
Non-covered errors are effective errors that 1) escape the error detection mechanisms and 2) produce wrong results.

The following distinction between *faults* and *errors* is made. A fault is an event, in our case the occurrence of a bit-flip inside the microprocessor, which immediately leads to an error; an error is a perturbation of the internal state of the microprocessor. An error may become *overwritten* in either the location where the fault was injected or, after propagation, in another location.

An injected fault is *activated* when the corresponding error first propagates. For example, if a fault is injected in a data cache, the fault is activated when the incorrect data is first used by the CPU.

3. Path-Based Error Coverage Prediction

The Path-Based Error Coverage Prediction technique is an enhanced version of the Execution Profile Based Prediction technique presented in [6]. It predicts the error coverage for an arbitrary input sequence based on the results from fault injection experiments with another input sequence, called the *base sequence*. The technique is based on the fact that the execution path of a program varies for different input sequences. Depending on the input sequence, different parts of the program are executed different number of times. This is illustrated in Fig. 1, where a program is divided into basic blocks denoted A to E. Each basic block is executed a different number of times depending on the input sequence.



Block (*i*)	No. of times executed for input K ($x_{K,i}$)	No. of times executed for input L ($x_{L,i}$)
A	1	1
B	0	1
C	8	3
D	8	3
E	1	1

Fig. 1. Workload execution path differs for different input data

The Execution Profile Based Prediction technique relies on the approximation that the basic block executing when a fault is activated is the same as the block executing when the fault is injected. The Path-Based Error Coverage Prediction technique, however, uses precise information about which block is executing during activation of the injected fault.

Let $P_{nc,i}$ denote the probability that a fault results in a non-covered error, given that the fault is activated during execution of block i. The prediction is based on the assumption that $P_{nc,i}$ is constant for all input sequences. This is motivated by the fact that the activity of the system during execution of a basic block is the same regardless of the input sequence used, i.e. the same instructions are always executed in each basic block. However, this probability is likely to vary for different basic blocks in a program.

The error coverage for an input sequence p, c_p, can be calculated using the following equation for a workload program containing n basic blocks:

$$c_p = 1 - \sum_{i=1}^{n} P_{nc,i} \cdot w_{p,i} \qquad (3.1)$$

Where $w_{p,i}$ is a weight factor for block i, corresponding to 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. $P_{nc,i}$ is estimated from fault injection experiments conducted with the base input sequence as:

$$\hat{P}_{nc,i} = \frac{n_{nc,i}}{n_i} \qquad (3.2)$$

Where $n_{nc,i}$ is the observed number of faults activated when block i is executing resulting in non-covered errors, and n_i is the total number of faults activated during execution of block i. The weight factor $w_{p,i}$ is estimated using the following equation:

$$w_{p,i} = \frac{n_{p,i}}{n_p} \qquad (3.3)$$

Where $n_{p,i}$ is the number of faults activated during execution of block i for input sequence p and n_p is the total number of faults activated for input sequence p. Now, $n_{p,i}$ is unknown, but can be estimated based on the number of activated faults for the base sequence as:

$$\hat{n}_{p,i} = \frac{n_i}{x_i} \cdot x_{p,i} \qquad (3.4)$$

Where n_i is the number of faults activated during execution of block i for the base input sequence, and x_i is the number of times block i is executed for the base input sequence. Thus, n_i divided by x_i gives the number of activations per execution in block i for the base input sequence. This is multiplied with $x_{p,i}$, which is the number of times block i is executed for input sequence p, thereby giving an estimation of $n_{p,i}$, the number of faults activated during execution of block i for input sequence p. This definition requires that all basic blocks are executed at least once for the chosen input sequence, otherwise additional input sequences must be chosen for the measurements. Identifying the input sequence(s) that uses all basic blocks can be performed through test coverage measures such as block coverage [9].

n_p can be estimated as the sum of the estimated number of activated faults for all the n basic blocks as:

$$\hat{n}_p = \sum_{j=1}^{n} \frac{n_j}{x_j} \cdot x_{p,j} \qquad (3.5)$$

Where n_j is the number of faults activated during execution of block j for the base input sequence, $x_{p,j}$ is the number of times block j is executed for input sequence p, and x_j is the number of times block j is executed for the base input sequence.

By combining Equations 3.1-3.5, the predicted error coverage for input sequence p, c_p, can be estimated from fault injection experiments conducted using a chosen base input sequence as:

$$c_p = 1 - \sum_{i=1}^{n} \left(\frac{n_{nc,i}}{x_i} \cdot \frac{x_{p,i}}{\sum_{j=1}^{n} \frac{n_j}{x_j} \cdot x_{p,j}} \right) \qquad (3.6)$$

When the total error coverage is predicted using Eq. 3.6, the activation of faults leading to unpropagated latent or overwritten errors have to be defined. For faults leading to unpropagated latent errors, we define the fault to be activated when it is injected. For faults leading to unpropagated overwritten errors, the fault is activated when the corresponding error is overwritten.

4. Experimental set-up

In this section we describe the experimental set-up used to evaluate the prediction techniques, the fault model and the fault locations, as well as the error classification used for the presentation of the results.

Target system: The experiments have been conducted on the Thor processor, a microprocessor developed by SAAB Ericsson Space AB. Thor is a 32-bit Reduced Instruction Set Computer (RISC) with a four-stage pipeline and 128 byte data cache. Thor features a stack-oriented instruction set architecture.

There are several error detection mechanisms implemented in the Thor processor such as *run-time checks*, *control flow checking* and *main memory error checking*. The run-time checks, described in Table 1, include those, which are commonly found in other microprocessors, e.g., division by zero or overflow checks, as well as Thor specific checks such as constraint checks of, for example, array indices.

Table 1. Hardware exceptions in Thor

Exception	Description
Bus error	Bus time out of external memory
Address error	Operand effective address larger than 2 Gbyte
Data error	Chip input signal DE (Data Exception) is asserted by the EDAC
Instruction error	Illegal instruction or trying to execute a privileged instruction in user mode
Jump error	Attempt to jump, call or return to a target address outside memory address space
Constraint error	A constraint of run-time assertion instructions is not satisfied
Access error	Attempt to follow a null pointer
Storage error	Attempt to access memory outside the task's stack in user mode
Overflow check	Overflow of signed integer and float arithmetic operations
Underflow check	Underflow or denormalized result of float arithmetic operations
Division check	Division by zero
Illegal operation	Illegal operation for float and double arithmetic instruction involving 0 and ∞

Control flow checking is used to check that the execution of machine instructions follows the control flow of the program. The main memory checking mechanisms were not included in our experiments since no faults were injected into the main memory.

Fault injection tool: The experiments were performed using MEFISTO-C, which is an elaboration of the MEFISTO tool [10]. MEFISTO-C injects faults in VHDL models by utilizing simulator commands that affect signals. In this study, the Vantage Optium^{TM} simulator was used.

Workload: The workload program is based on an Ada package implementing a recursive quick-sort algorithm, which sorts seven data elements of the predefined Ada type *float*. Fault injection was performed during the actual sorting of the seven data elements. After the sorting was finished, the result was written to memory.

Fault model: The fault model used was the single bit-flip fault model. The faults were injected by changing the logical value of single state elements from zero to one or from one to zero. This fault model is reasonably accurate for SEUs (Single Effect Upsets), caused by e.g. heavy ion radiation in the space environment or by neutrons at high altitude [13].

Fault locations: Faults were injected in 4410 internal state elements (flip-flops and latches) of the Thor microprocessor. These state elements are divided in three parts; *Registers*, *Cache* and *Other registers* (811, 1824 and 1775 locations respectively).

Error classification: The MEFISTO-C tool is able to observe both non-effective and effective errors. The effective errors are divided into three categories:

- *Detected errors:* Errors that were detected by the Thor error detection mechanisms (See Table 1).
- *Other errors:* Errors that caused the CPU to hang, i.e., the processor stopped executing instructions. These errors were not detected by the CPU's internal detection mechanisms. (They would have been detected by an external watchdog timer, but such a mechanism was not implemented in the VHDL model.)
- *Non-covered errors:* Errors that were not detected by the error detection mechanisms causing a failure to occur, i.e., the sort-algorithm produced an incorrect result.

5. Applying and evaluating Path Based Error Coverage Prediction

The Path Based Error Coverage Prediction Technique was evaluated by investigating 24 input sequences, denoted *a–x*, corresponding to 24 randomly chosen permutations of seven elements to be sorted by the quick-sort program. A fault-free simulation of the program execution was performed for each of the input sequences. The execution paths for the input sequences were obtained by analyses of the simulation traces. This allowed the usage of the basic blocks of the program to be derived for each input sequence.

Fault injection campaigns were performed for the three input sequences denoted *x*, *i* and *h*. Table 2 shows a summary of the results obtained for the three fault injection campaigns. These results show that the input sequence has a significant impact on the estimated error coverage as the coverage varies between 92% and 96%. Around 70 % of the errors are non-effective errors. One reason for the high percentage of non-effective errors is that faults were injected into many unused locations of the CPU resulting in latent errors.

Table 2. Results of fault injection experiments

Name	Input sequence *h*	Input sequence *i*	Input sequence *x*
Number of injected faults	13928	12831	13682
Non-effective errors	71,17 ± 0,75 %	67,69 ± 0,81%	69,16 ± 0,77%
Effective errors			
Detected errors	24,15 ± 0,71%	24,09 ± 0,74%	23,64 ± 0,71 %
Other errors	0,41 ± 0,11%	0,35 ± 0,10%	0,33 ± 0,10 %
Non-covered errors	**4,27 ± 0,34%**	**7,87 ± 0,47%**	**6,87 ± 0,42 %**

Results from the fault injection campaign for input sequence *x* were analysed in detail to obtain the number of faults activated for each basic block and information about whether the faults led to non-covered errors or not. This information together with the basic block usage for input sequence *a-x* allowed the total error coverage for each of the input sequences to be calculated according to Eq. 3.6. The results of these predictions are shown in Fig. 2. The figure shows that input sequences *i* and *p* have the highest predicted total error non-coverage while input sequence *o* have the lowest.

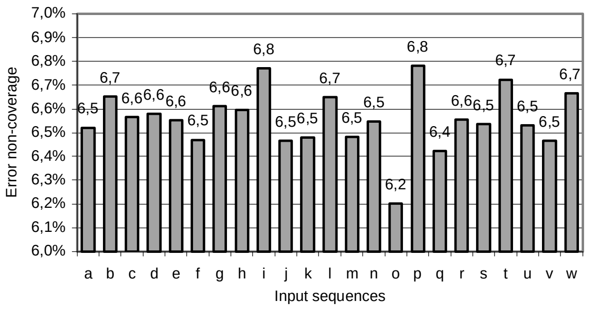


Fig. 2. Predicted total error non-coverage using input sequence *x* as the base sequence.

In order to evaluate the accuracy of the predictions, we used each of the input sequences *x*, *i* and *h* as the base sequences to predict the coverage of the other two sequences. That is, we used *x* to predict the coverage for *i* and *h*, and *i* to predict the coverage for *x* and *h*, etc. This allowed us to make in total six predictions that can be compared with the coverage obtained by fault injection.

5.1. Predicting effective error coverage

The results of using Path Based Error Coverage Prediction to predict the effective error coverage are shown in Figs. 3 to 5. The figures show a comparison of the predicted effective error non-coverage and the observed effective error non-coverage for the various parts of the CPU (Registers, Cache, Other) as well as the entire CPU (All).

The results show that the predictions are biased towards the non-coverage for the base sequence. Fig. 3 shows that the predicted non-coverage is higher than the observed non-coverage for both *x* and *h*, when *i* is used as the base sequence. The reason is that *i* has a higher observed non-coverage than both *x* and *h*. Fig. 5 shows that using *h* as the base sequence under-estimates the non-coverage for both *x* and *i*, since *h* has a lower non-coverage than the other two.

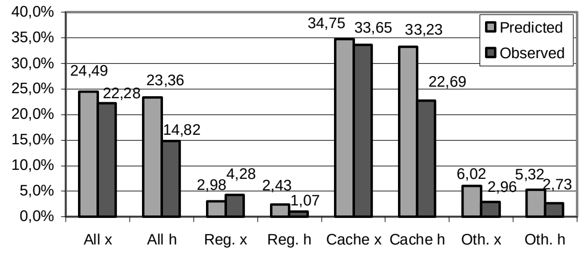


Fig. 3. Predicted vs observed effective error non-coverage using input sequence *i* as base sequence.

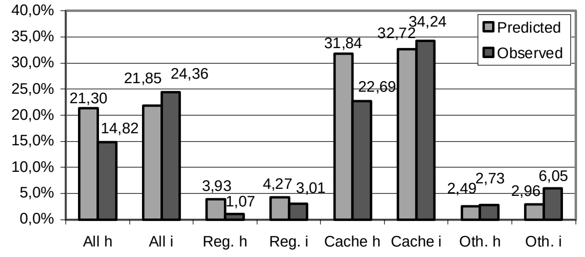


Fig. 4. Predicted vs observed effective error non-coverage using input sequence *x* as base sequence

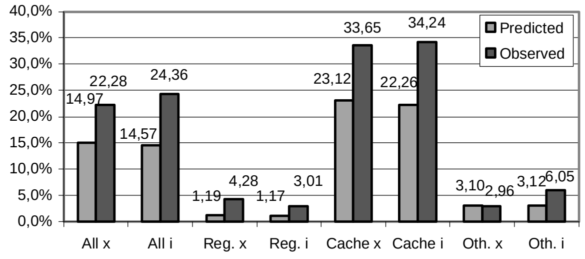


Fig. 5. Predicted vs observed effective error non-coverage using input sequence *h* as base sequence.

However, the primary goal is not to provide accurate estimations, but rather to conduct a ranking of the non-coverage for different input sequences.

Fig. 3 shows that the predictions correctly rank the coverage for input sequence *x* to be higher than the coverage for input sequence *h*, when *i* is used as the base sequence. Fig. 4 shows that the predictions also manage to correctly rank the coverages for input sequences *h* and *i*, when *x* is used as the base sequence. However, Fig. 5 shows that the ranking is not correct when input sequence *h* is used as the base sequence. This is not surprising since the observed effective error coverage for input sequence *i* and *x* are very similar; the difference is only ~2 percentage points for faults injected into the whole CPU and ~0.6 percentage points for faults injected into the cache. Whereas using base sequences *i* or *x*, the difference between the non-coverage to be predicted is more than 8 percentage points.

5.2. Predicting total error coverage

The results of predicting total error coverage are shown in Figs. 6 to 8. Our prediction technique is able to identify the input sequence with the highest total error non-coverage using base sequence *i* or *x*. However, similar to what was observed in Section 5.1, the prediction is incorrect when base sequence *h* is used since the difference in total error coverage for sequence *i* and *x* is not significant enough.

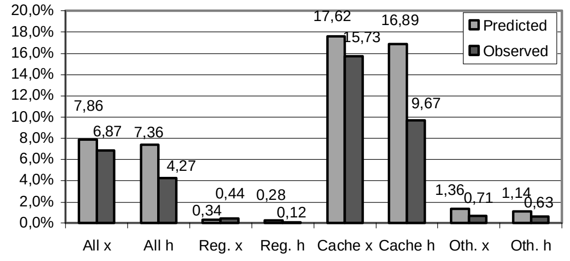


Fig. 6. Predicted vs observed total error non-coverage using input sequence *i* as base sequence.

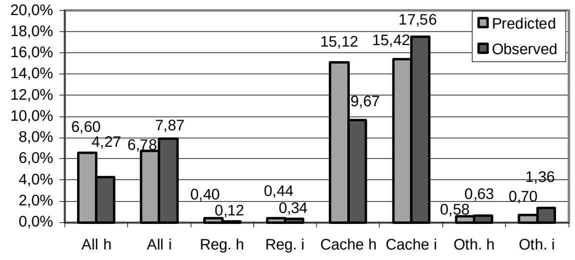


Fig. 7. Predicted vs observed total error non-coverage using input sequence *x* as base sequence.

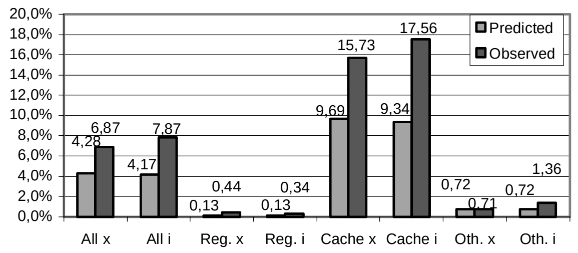


Fig. 8. Predicted vs observed total error non-coverage using input sequence *h* as base sequence.

6. Conclusions

A post-injection prediction technique aimed at reducing the cost of estimating error coverage was proposed. The technique, called Path-Based Error Coverage Prediction, uses fault injection data obtained for a given input sequence to estimate the error coverage for other input sequences. The need to consider input sequence variations when estimating the error coverage using fault injection has been shown in previous studies and was also verified in this study. Our results show that the estimated total error coverage for various input sequences to a quick-sort program executing on the Thor microprocessor may vary between 92% and 96%.

A comparison of predicted and observed results using three fault injection campaigns conducted with different input sequences to the quick-sort program showed that the technique is able to identify the input sequence with the highest or lowest error coverage provided that the difference in actual coverage is significant.

Our experiments also gives an indication of the time savings to be expected using Path-Based Error Coverage Prediction. The time needed to estimate the error coverage for a workload with a particular input sequence using fault injection (3000 experiments) on a 300 Mhz Sun workstation is in the order of one hundred hours, while the time needed to predict the error coverage using an automated tool is in the order of minutes.

Although the study shows promising results, we would like to stress that this research still is in an early stage. More fault injection experiments are needed to show if the technique can be generally applied on other workloads and target systems. The prediction technique should also be further refined, e.g., the error coverage for each basic block can probably be estimated more accurately using a mean value calculated from several fault injection experiments. The statistical analysis of the predictions, i.e. calculation of confidence intervals etc., needs further investigation. We also need to find the reason for the bias in the predictions and try to find a way of eliminating it. To further reduce the cost of coverage estimations, our technique should be combined with pre-injection analysis in order to better control the fault injections so that the coverage estimations for the individual basic blocks could be made more accurate and less time consuming.

References

1. A.M. Amendola, L. Impagliazzo, P. Marmo, F Poli, “Experimental evaluation of computer-based railway control systems” *Digest of Papers, Twenty-Seventh Annual Int. Symp. on Fault-Tolerant Computing*, 1997, pp 380 –384.
2. T.F. Arnold, “The Concept of Coverage and Its Effects on the Reliability Model of a Repairable System”, *IEEE Trans. Computers*, vol. C-22, pp. 251-254, Mar 1973.
3. A. Benso, M. Rebaudengo, L. Impagliazzo, P. Marmo, “Fault-list collapsing for fault-injection experiments”, *Proceedings Annual Reliability and Maintainability Symposium*, 1998, pp. 383 –388.
4. C. Constantinescu, “Using multi-stage and stratified sampling for inferring fault-coverage probabilities”, *IEEE Trans. on Reliability*, vol. 44, pp. 632 –639, Dec. 1995.
5. E. Czeck, and D. Siewiorek, “Observations on the Effects of Fault Manifestation as a Function of Workload”, *IEEE Trans. on Computers*, vol. 41, pp. 559-566, May 1992.
6. P. Folkesson and J. Karlsson, “Considering Workload Input Variations in Error Coverage Estimation”, *Proc. Third European Dependable Computing Conf.*, 1999, pp. 171-188.
7. P. Gawkowski, J. Sosnowski, “Analyzing fault effects in fault insertion experiments”, *Seventh Int. On-Line Testing Workshop*, 2001, pp 21–24.
8. J. Guthoff, V. Sieh, “Combining software-implemented and simulation-based fault injection into a single fault injection method” *Digest of Papers, Twenty-Fifth Int. Symp. on Fault-Tolerant Computing*, 1995, pp. 196-206.
9. J.R. Horgan, S. London, and M.R. Lyu, “Achieving software quality with testing coverage measures”, *IEEE Computer*, vol. 27, pp. 60-69, Sept. 1994.
10. R.K. Iyer, “Experimental Evaluation”, *Special Issue of Proc. Twenty-Fifth Int. Symp. on Fault-Tolerant Computing*, 1995, pp. 115-130.
11. E. Jenn, J. Arlat, M. Rimén, J. Ohlsson, and J. Karlsson, “Fault Injection into VHDL Models: The MEFISTO Tool”, *Proc. 24th Int. Symp. on Fault-Tolerant Computing*, 1994, pp. 66-75.
12. J. Karlsson, P. Lidén, P. Dahlgren, R. Johansson and U. Gunneflo, “Using heavy-ion radiation to validate fault-handling mechanisms", *IEEE Micro*, vol. 14, pp. 8-23, Feb. 1994.
13. G. C. Messenger, "Collection of Charge on Junction Nodes From Ion Tracks", *IEEE Trans. on Nuclear Science*, vol. 29, pp. 2024-2031, Dec. 1982.
14. *Saab Ericsson Space AB, Microprocessor Thor, Product Information*, September 1993.
15. T.K. Tsai, M.-C. Hsueh, H. Zhao, Z. Kalbarczyk, R.K. Iyer, “Stress-based and path-based fault injection”, *IEEE Trans. on Computers*, vol. 48, pp 1183–1201, Nov. 1999.