Technical Report

FALCON: Rapid Statistical Fault Coverage Estimation for Complex Designs

Shahrzad Mirkhani and Jacob A. Abraham

UT-CERC-12-04

May 30, 2012

Computer Engineering Research Center
Department of Electrical & Computer Engineering
The University of Texas at Austin

201 E. 24th St., Stop C8800
Austin, Texas 78712-1234

Telephone: 512-471-8000
Fax: 512-471-8967
http://www.cerc.utexas.edu

FALCON: Rapid Statistical Fault Coverage Estimation for Complex Designs

Shahrzad Mirkhani, and Jacob A. Abraham

Computer Engineering Research Center
The University of Texas at Austin, Austin, TX 78712
Email: {shahrzad, jaa}@cerc.utexas.edu

Abstract—**FALCON (FAst fauLt COverage estimatioN) is a scalable method for fault grading which uses local fault simulations to estimate the fault coverage of a large system. The generality of this method makes it applicable for any modular design. Our analysis shows that the run time of our algorithm is related to the number of gates and the number of IOs in a module, while fault simulation run time is related to the total number of gates in the system. We have measured fault coverage for OR1200 and IVM processors and compared the results with fault simulation performed by a commercial tool. We have also compared our results with fault sampling. Our results show that for large designs FALCON works faster (one order of magnitude) compared to fault simulation. It also has smaller error rate compared to fault sampling when the size of design under test grows.**

I. Introduction

Fault grading has been studied for half a century [28]. The complexity of fault grading makes it time-consuming for today's large designs. Design-for-test techniques can reduce the complexity of fault grading.

Although scan based methods, like full-scan, are widely used in chip testing, they cannot eliminate the need for functional testing. There are several papers which compare structural and functional test methods [19][16][17][30]. These papers address issues for structural testing like test quality, test vector re-usability, and test application time.

However, due to growing design size and complexity and decreasing feature lengths, other problems in structural testing show up. Small delay defects, capacitive coupling between circuit lines, and power droops are some design related issues that might not be addressed effectively by structural testing. Since functional tests are applied in the system with normal mode operation, they are able to address the issues related to process variation. For example, they are more efficient in detecting small delay defects compared to scan-based designs. In addition, since in functional testing the circuit is operating at its normal mode, the frequency goals of the design is completely satisfied and also there are no issues like potential overkill issues which can be seen in scan tests.

Therefore, to be able to test the designs using functional test vectors, there is still a need for measuring coverage for application level tests after manufacturing. For example, in microprocessors, these tests are instruction level tests which are applied to the the whole processor chip while it is operating in its normal mode inside the system. Industrial designs have used such techniques as their test techniques [23] [7].

Proposed methods attacking the fault grading problem can be categorized into three major groups: fault simulation, fault emulation, and coverage estimation (statistical methods). Approaches to fault simulation include basic gate-level algorithms [1] and several hybrid methods. These methods either combine basic methods (like PROOFS [20]), or they use different levels of abstraction to speed up the simulation process [13][27][18]. Although these methods are faster than the traditional gate-level methods, they are still not scalable. To overcome this problem, high-level (example, RTL) fault models [29] have been proposed. These methods scale well with the size of the design, but they lack a precise correlation with the fault models used to evaluate the coverage of manufacturing test sequences. In general, in a large design with a large set of test vectors, system fault simulation seems to be nearly impossible [9].

Fault emulation was proposed in the 90s [31]. The main drawback of fault emulation is that the design under test should be fully synthesizable, which makes emulation not applicable in early design stages. Also, fitting large designs into the emulation hardware might not be feasible.

Another solution for fault grading, proposed first in the 80s, is to estimate the coverage using some data from good simulation and/or the circuit structure [4][8][14]. These methods are based on fault sampling [2], test vector sampling [10], and gate-level statistical analysis using data from design simulation (called STAFAN) [12][15]. The initial work was mainly on the Single Stuck-At (SSA) fault model. Later, fault models were expanded to path delay faults and also sequential circuits [11][24][6]. Some high-level testability measurements were also introduced [25] and an extension of STAFAN to RTL components was proposed [26]. Since most of the statistical methods use good simulation data, their run-time is comparable to the run-time of good simulation, which makes them scalable. Apart from their scalability, these methods (like STAFAN) introduce some parameters (for re-convergent fanouts for example) which should be determined empirically. The results have been shown only for small circuits. There is also a commercial tool [5] based on statistical fault analysis which uses testability measurements and sets the faults with 0 probability as *undetectable*. Then it fault simulates the design with the rest of the faults. The error in this tool is claimed to be not more than 10%, but it still depends on the fault simulation on the whole system.

In this paper, we present FALCON, a coverage estimation method for SSA faults in modular designs, which overcomes some of the above problems. Our method is similar to [15] in that both are based on functional block observability calculation. However our approach is different, since their approach is an extension to STAFAN and has the same drawbacks as the STAFAN method. Our method eliminates the need for defining empirical parameters. However it needs more simulations (local simulations) to determine a more realistic measure for fault propagation. These local fault simulations make our method applicable for industrial designs with an acceptable error range.

FALCON uses data from stand-alone fault simulations to estimate the detection probability of each fault in the whole system. Therefore, it reduces the complexity of fault grading from a function of G to a function of g, where g is the number of gates inside a module and G is the number of gates in the whole system. This method is applicable to both combinational and sequential designs. As another feature, users are also able to apply this method on early stage designs when all the modules may not be at the gate level of abstraction.

This can help test engineers start the test process shortly after the design process starts and since it is based on functional test vectors, it can re-use test vectors from the design verification process. Since this methodology uses the manufacturing fault models, the resulting coverage is completely correlated with gate-level fault coverage.

We have applied FALCON on two processors, OR1200 [22] and IVM [21]. OR1200 has around 40 thousand gates and 2 thousand sequential elements, while IVM has around 5 million gates and more than 100 thousand sequential elements. We have injected around 75 thousand faults in OR1200 and 320 thousand fault in IVM. Our experiments show that FALCON works much faster than fault simulation and it estimates the coverage more accurately than the fault sampling method [2] with a confidence of 0.998 [3]. Our contributions to this work include the following.

- To our knowledge, the idea of divide-and-conquer for coverage estimation of modular designs has not been proposed before. The existing methods (like STAFAN) use gate granularity rather than module granularity.
- We have developed a fully automated environment for our coverage estimation method using a commercial fault simulator, a commercial logic simulator, and a few Perl scripts to feed these tools the proper testbenches.
- We have measured the results on a small-size (around 40K gates) and a large-size testcase (around 5M gates). Almost all of the previous methods have been shown for relatively trivial designs (around a few thousand gates).

The structure for the rest of this paper is as follows. In Section II, we discuss our approach. Section II-A and Section II-B describe the methodology. The following subsections (II-C, II-D, and II-E) describe the details for each step of our estimation method. In Section III, we show our experimental results. Finally, in Section IV, a brief run-time analysis is discussed.

II. Coverage Estimation Methodology

A. Overview

We have developed a coverage estimation method for large modular designs, where a module can be combinational or sequential. A module boundary can be the HDL (like Verilog) modules in a hierarchical design. However, if the design is flattened by the synthesis tool, the partitioning algorithm to make partitions for the design is not difficult. Any boundary which includes a reasonable number of gates can be used in this method.

In this paper, the SSA fault model has been used. However, this method can be applied to any desired fault model. The main idea of our approach is to accurately estimate, from the fault grading results on a standalone module, the coverage when that module is embedded in a larger system. This is accomplished by estimating how each *module* can propagate an error from one of its inputs to one of its outputs. We also calculate the probability of the presence of errors on the outputs of each module-under-test (MUT). The latter factor gives us an idea of how many errors will be activated on the boundaries of a MUT, while the former factor helps us find how many of these activated faults can be propagated through other modules in the whole system. Combining these two, we can estimate how many errors can reach the system's primary outputs.

The following are a few terms used frequently in this paper.

- MUT (module-under-test): A module in the system which is the target for fault grading. We perform fault grading module by module.
- Detection probability table: A table indicating the probability of each fault in a MUT to be present at one of the outputs of that MUT. There is one detection probability table per each MUT.
- Propagation table: A table indicating the possibility of error propagation from one input of a module to one of its outputs. There is one propagation table for each module in the design.
- Local test vector set: The set of stimuli at the inputs of a module, when applying the test vector set to the primary inputs of the system.
- Stand-alone fault simulation: The process of fault simulating a module, separated from the system, with its corresponding local test vector set.
- Local fault dictionary: The resulting fault dictionary when performing stand-alone fault simulation on a module.

B. Algorithm Steps

This section describes our methodology step by step using an example. The details of each step will be discussed in the following sections. We start with a modular design and an input test sequence whose fault coverage needs to be determined.

- Step 1: Given a test vector set, we simulate the whole system to generate local test vectors for each module. This step is done by a commercial logic simulator.

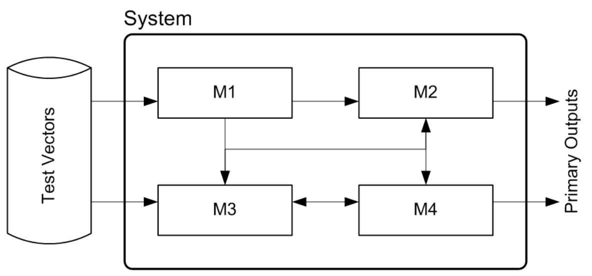


Fig. 1. System block-diagram

A system with 4 modules and a set of test vectors is shown in Figure 1, while Figure 2 shows the system after applying this step.

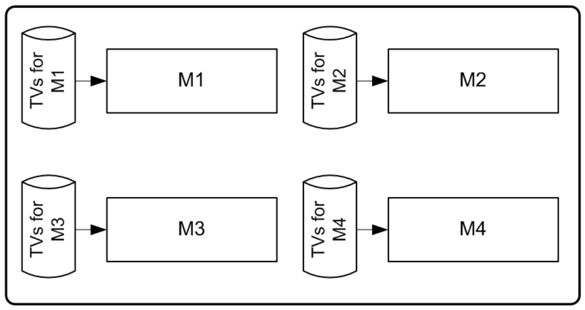
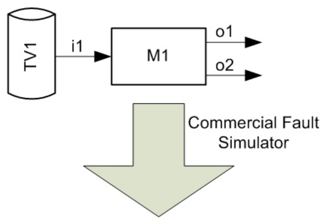


Fig. 2. Module with local test vectors

- Step 2: Now that we have local test vectors, we perform stand-alone fault simulation for each MUT ($M1$ in our example). Note that faults are not dropped during this process, because we want to measure the probability of each fault detection. Therefore, the more a fault is detected on a MUT output, the more probable it can be detected on a system primary output. In this step, the results are stored in local fault dictionaries. This step is done by a commercial fault simulator (shown in Figure 3).
- Step 3: Using the local fault dictionaries from step 2, detection probability tables are generated for each MUT and propagation tables are generated for all modules in the system (Figure 4). Note that for modules which are not in MUT set, we still need to generate propagation tables. We will discuss this in more details in Section II-D. This step is done by a Perl script.
- Step 4: We generate a statistical model using module interconnections in our design, propagation tables for each module in the design, and detection probability tables for each MUT (Figure 5). By simulating this statistical model with a commercial simulator, we are able to estimate the fault coverage of each MUT in the whole system. We will describe the probability calculation formula in Section II-E.



	Fault	o_1 (0/1)	o_1 (1/0)	o_2 (0/1)	o_2 (1/0)
	f_1	10	2	0	0
i1 s@0	f_2	25	0	1	0
i1 s@1	f_3	13	0	2	0
	…				
	f_k	1	10	4	0
	…				

Fig. 3. Local fault dictionary

Propagation Table for M1

Input	Output	0/1 -> 0/1	0/1 -> 1/0	1/0 -> 0/1	1/0 -> 1/0
i1	o1	0.5	0	1	0
i1	o2	0.076	0	0.038	0

Detection Probability Table for M1

Fault	o_1 (0/1)	o_1 (1/0)	o_2 (0/1)	o_2 (1/0)
f_1	0.2	0.04	0.0	0.0
f_2	0.5	0.0	0.02	0.0
f_3	0.26	0.0	0.04	0.0
…				
f_k	0.02	0.2	0.08	0.0
…				

Fig. 4. Propagation and detection probability tables

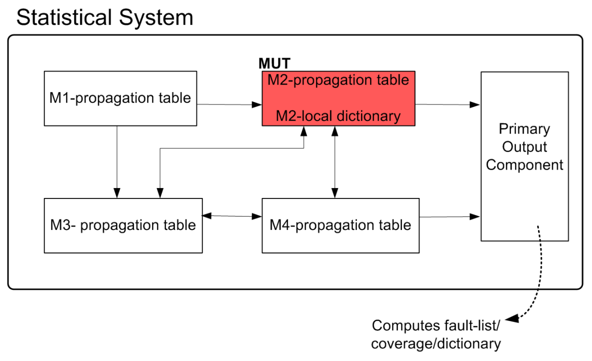


Fig. 5. Statistical system block diagram

C. Detection Probability Tables

As discussed above, this table indicates the detection probability for each fault on each output of a MUT. It is implemented as a 3-dimensional array; the first dimension represents the fault number, the second dimension represents the MUT output number, and the third dimension is either 0 or 1. Zero represents a 0/1 value and one represents a 1/0 value (a line with $v/\bar{v}$ value shows that an error has reached that line and inverted the value of that line from v to $\bar{v}$).

TABLE I
INTERPRETATION OF INDEX VALUES IN PROPAGATION TABLE

Index Value	representation
0	$0/1 \to 0/1$
1	$0/1 \to 1/0$
2	$1/0 \to 0/1$
3	$1/0 \to 1/0$

The value of each element in this table, e.g. $det\_prob\_table[f][o][v]$, is calculated as below.

$$det\_prob\_table[f][o][v] = \frac{\#\ of\ times\ f\ is\ detected\ on\ o\ with\ value\ v/\bar{v}}{\#\ of\ test\ vectors} \quad (1)$$

As an example, fault #10 is detected on the 5th output of a MUT in the stand-alone fault simulation process, 4 times with value 0/1 and 11 times with value 1/0. Suppose our test vector set contains 100 test vectors, then the detection probability table includes

$det\_prob\_table[10][5][0] = 0.04$ and $det\_prob\_table[10][5][1] = 0.11$

D. Propagation Tables

This table calculates the ability of a module to propagate an error from each of its inputs to each of its outputs. As discussed in Section II-A, this table is generated using the local fault dictionaries of each module. However, if we do not have this module at the gate level, we can generate this table by simulating this module stand-alone (with its local test vectors) and inject the module's input stuck-at-0 (stuck-at-1) faults by putting a constant 0 (1) instead of the value of that input. The number of simulations will be $2 \times i$ where i is the number of module inputs. Since this is done on a high-level module, the simulation cost is not that high. In another case, if we have the gate-level of abstraction for a module but we do not want to perform fault grading for this module, we can inject only the faults for this module's primary inputs and perform stand-alone fault simulation.

Similar to detection probability tables, propagation tables are also implemented as a 3-dimensional array. The first dimension is the input number, the second dimension is the output number, and the third one is between 0 and 3. Value 0 for this dimension shows the propagation probability of a 0/1 value from an input to a 0/1 value to an output. Table I shows the interpretation of other values for this dimension.

The value of each element of this array is calculated to be the *propagation factor* from an input to an output. If value $v/\bar{v}$ can be propagated through output o, it means that fault $i-sa-\bar{v}$ is detected on output o. Therefore, we calculate propagation factors from fault simulation as below:

$$prop\_table[i][o][k] = \frac{\#\ of\ times\ i-sa-\bar{v}\ detected\ on\ o\ with\ w/\bar{w}\ effect}{\#\ of\ times\ i-sa-\bar{v}\ activated}$$

Note that we do not divide the numerator by the number of test vectors. This is because whenever we use this factor in our calculations, the error has been already propagated through the input of this module. Therefore, we only need to use a definition similar to conditional probability (i.e. the probability of an error propagation given that error is activated).

Also, we do not call this factor as propagation *probability*. This is because it can happen that the number of errors detected is more than the number of error activations and this factor becomes greater than 1. This can happen in modules with sequential feedback paths.

E. Detection Probability Function

Suppose we have generated a propagation table for each module in the system and we have generated detection probability table for our MUT. Now, using these tables, we want to calculate the detection probability of each fault on system primary outputs. For this purpose, we need to define a function that accepts the detection probability values of module's inputs and calculates the detection probability values of that module's outputs using propagation factors of that module.

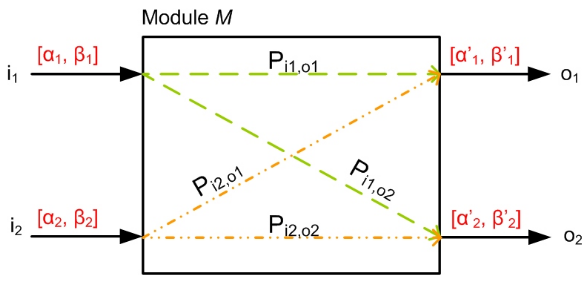


Fig. 6. A sample for detection probability function

Suppose an error is propagated through more than one input of a module. This case happens usually since we always have fanouts in our design. In our example in Figure 6, suppose an error has reached input i_1 and i_2 with 0/1 probability values equal to α_1 and α_2, and 1/0 probability values equal to β_1 and β_2, respectively. In this case, it is easier to calculate the probability of *absorption* of an error from *ALL* inputs through an output and then negate this absorption probability to reach the propagation probability from either of inputs to that output. This idea is a realization of the following probability formula (suppose A and B are independent events),

$$P(A \cup B) = 1 - (1 - P(A)) \times (1 - P(B)) \quad (2)$$

Note that we are adding some error by assuming that the two events are independent. Because in reality, two inputs of a module can affect each other in error propagation (i.e. the error can be masked). Since we are only dealing with system re-convergent fanouts and intra-module re-convergent fanouts are taken care by stand-alone fault simulations, we expect only a small amount of error due to this assumption in our estimation method. This is validated by our experimental results discussed in Section III.

Using Formula 2, the detection probability of the error reached i_1 and i_2 on o_1 with value 0/1 can be calculated as,

$$\begin{aligned} det\_prob[o_1][0] &= \alpha_1' \\ &= 1 - \quad [(1-\alpha_1 \times P_{i_1,o_1}^{0/1\to 0/1}) \times (1-\beta_1 \times P_{i_1,o_1}^{1/0\to 0/1}) \\ &\qquad \times (1-\alpha_2 \times P_{i_2,o_1}^{0/1\to 0/1}) \times (1-\beta_1 \times P_{i_2,o_1}^{1/0\to 0/1})] \end{aligned}$$

The other values for detection probability of the outputs can be calculated in a similar way. A general formula for o_1 with value 0/1, when an error reaches N inputs is,

$$det\_prob[o_1][0] = 1 - \prod_{n=1}^{N} (1-\alpha_n \times P_{i_n,o_1}^{0/1\to 0/1}) \times (1-\beta_n \times P_{i_n,o_1}^{1/0\to 0/1})$$

F. Fault Detection Metric

Now that we can calculate the detection probabilities of a fault on each line in the system, we need to know a way to determine which value (or ranges of values) should be determined as **detected** and which ones should be considered as **not detected**. In other words, we need a metric for our fault coverage.

Using our statistical system and statistical simulation environment (Section II-G), we calculate the detection probability for MUT faults from the outputs of each MUT through the primary outputs of the system. Since detection probability is defined as in Equation 1, and we define our detection threshold as,

$$detection\_threshold = \frac{1}{\#\ of\ test\ vectors}$$

This threshold means that the fault is detected one time when applying our test vector set to our design. Therefore, if a detection probability value at a system primary output is greater than or equal to this value, it should be counted as a detected fault.

Using our detection probability function and our defined detection threshold, we can estimate the fault coverage of the system for each MUT. Due to our detection probability definition in Equation 1, the output of our statistical system shows the detection of the faults in the system **as if they are not dropped**.

G. Statistical System and Simulation

After we build propagation tables and detection probability tables, it is time to calculate the detection probability for each line in our design using the detection probability function discussed in Section II-E. Note that if the top module of the system (the module we are building our statistical system from) has some glue logic, we wrap it inside a dummy module and generate propagation tables for this dummy module as well. We have done this in one of our testcases. We generate our statistical system (in Verilog) following the steps below.

- Replace every module in the system with its propagation table.
- Add a detection probability table to the MUT.
- Connect these high-level models as they were connected in the original design.
- Change the signal type to a type which accepts the detection probability for both 0/1 and 1/0 values (e.g. a two element array of type *real*).

Given the above statistical system (along with a library containing detection probability function), and our commercial simulator, the detection probabilities of interconnections and system primary outputs can be calculated. For coverage calculation, detection probabilities on primary outputs are compared with our defined detection threshold.

III. Experimental Results

We have developed scripts for generating local test vectors and testbenches for stand-alone fault simulation to be able to apply our method on designs. Figure 7 shows the flow of our estimation methodology.

We have applied FALCON on two CPU designs, OR1200 which is a RISC processor and IVM which is an implementation of alpha processor. These CPUs are Verilog designs which were synthesized with the TSMC 180nm technology library. Table II shows some characteristics for each test case. We ran our experiments on an Intel® Xeon® X5670, 2.93GHz processor, with 72GB of memory, and 12 cores (with hyper threading).

As discussed in previous sections, FALCON estimates the presence of each fault on each output of a design. This can be considered as a statistical fault dictionary. To show the accuracy of our estimation method, we have performed fault simulation on a sub-set of faults for OR1200 without fault dropping and averaged the appearance of each fault on each primary output. On the other hand, we have applied our method on the same sub-set of faults and measured the detection probability of each fault on each primary output. An example is shown in Figure 8 (fault simulation) and Figure 9 (fault estimation) for the faults in ALU module in OR1200. As it can be seen, these two measurements are very close to each other, which means FALCON is able to prepare statistical data about fault detection rather than outputing only a coverage number. This data can be used for purposes like fault diagnosis. Other estimation methods, like fault sampling, do not output any other data but the fault coverage. However, in this paper, we have compared our results with the results of fault simulation **with** fault dropping, which is not a fair comparison for FALCON. But we have done this comparison to show its speed and accuracy.

We have performed traditional fault simulation and fault sampling (using the same commercial fault simulator that we use in our method for standalone fault simulations) on the whole system, measured their run-time and coverage, and we have compared the run-time and fault coverage of our estimation method with these results. As discussed above, fault simulation and fault sampling processes are done with fault dropping.

In fault sampling method, a sample of faults from the fault list is selected and fault simulated. Based on the fault coverage

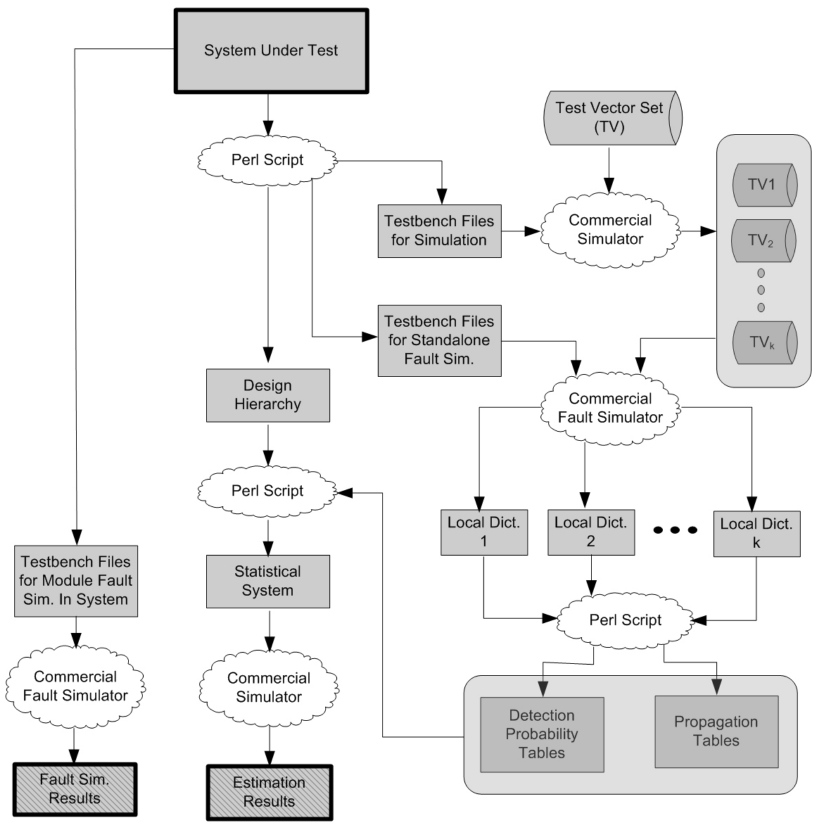


Fig. 7. Coverage estimation and fault simulation measurement flow

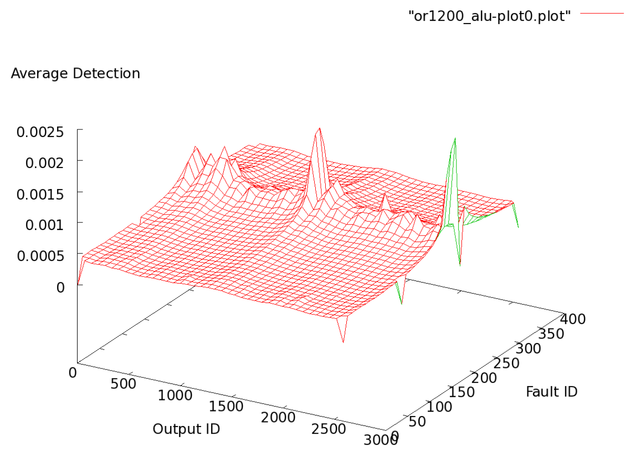


Fig. 8. Average detection in fault simulation

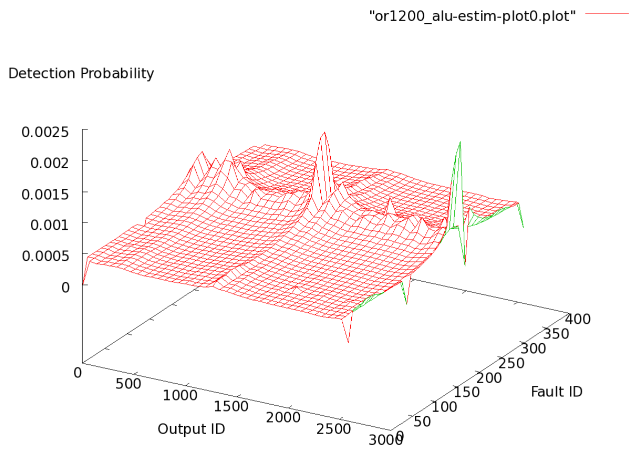


Fig. 9. Detection probability in coverage estimation

from these sampled faults, the fault coverage for the whole system is calculated using a formula. This method gives the user a range of fault coverage with a level of confidence. Based on a few experiments on both designs, we have found the best sample size as 10% of the whole faults and we have found that the confidence of 0.998 (known as 3σ) gives the best answer for this method. For example, for OR1200 design, with 1024 test vectors and a sample of 7512 faults (10%), fault sampling method gives us a range equal to [28.4, 30.9] with a confidence of 0.998. This means that with a probability of 0.998, the real fault coverage (for the whole system) is between 28.4% and 30.9%.

TABLE II
TESTCASE CHARACTERISTICS

design name	approx. size (gates)	MUT modules	# of inputs	memory elements	analyzed faults
OR1200	40,000	13	387	2,000	75,129
IVM	5,000,000	18	836	100,000	320,912

Our experimental results show that fault sampling coverage range does not match the real coverage in several cases. In cases that the calculated coverage matches the real fault coverage, we have put 0% error in our tables and diagrams. For the cases that the real coverage is not in the calculated range, we have calculated the error as the difference between the real fault coverage and the coverage in the middle of the range.

In the following sections, we will discuss our experiments on two case studies using some tables and diagrams.

A. *OR1200 Case Study*

For OR1200 case study, we have applied different sizes of test vectors to the CPU. These test vectors are random vectors. Fault coverages are shown in Table III. The first column shows the number of test vectors, while columns 2, 3, and 4 show the coverage results for fault simulation, fault sampling, and our coverage estimation method, respectively. As discussed above, in column 3, a range of fault coverage is shown. Column 5 indicates the error between our estimation method and fault simulation method. We have measured our error as **the number of mis-calculated faults over the total number of faults**. That is why the difference between fault coverages shows a smaller number than the error shown in the fifth row of Table III. The sixth row of this table shows the error between fault sampling method and traditional fault simulation method (columns 2 and 3). As discussed above, the error is defined as 0% if the real coverage is in the range of the calculated coverage. The next two columns in this table shows the number of misdetected faults (rather than percentage) in coverage estimation and fault sampling methods, respectively. The last column shows the difference between the number of misdetected faults between fault sampling method and our estimation method. As it can be seen, this number is relatively high in the first three cases.

Table IV shows the run-time results for fault simulation, fault sampling, and our coverage estimation method for the runs whose coverages shown in Table III. The first column of this table shows the number of test vectors. The second, third, and forth columns show the run-times of fault grading for fault simulation, fault sampling, and our coverage estimation method, respectively. In column 5, speed-up in run-time between our coverage estimation and fault simulation has been shown. This speed-up factor is calculated by dividing the time spent in fault simulation method by the time spent in all the steps of coverage estimation (i.e. column 2 divided by column 4). All run-times are shown in *seconds*. We have measured the run-time speedup between FALCON and fault simulation. As it can be seen in this table, fault sampling works faster than

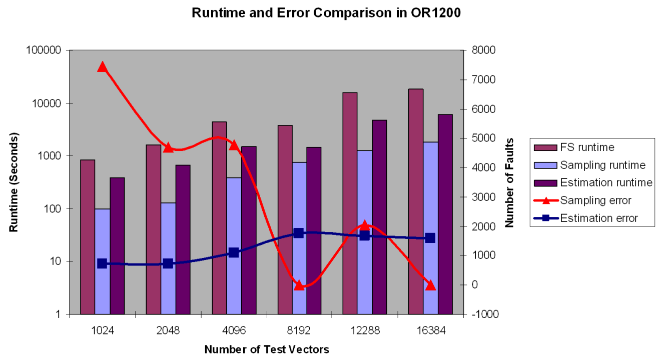


Fig. 10. OR1200 run-time results and misdetected faults

our method, but the error of sampling method is more than our estimation in most cases as shown in Table III. Also, when the design size grows, fault sampling method calculates less accurate results compared to our estimation method. However, the sampling method run-time is still comparable with our estimation method. This can be seen in the IVM test case in the next section (Tables V and VI).

We have summarized our results in Figure 10. This figure shows the run-time for fault simulation, fault sampling and coverage estimation in logarithmic scale (shown with bars). Also, sampling error and coverage estimation error are shown in this figure with lines. These errors are shown by the number of miscalculated faults. As it can be seen in this figure, fault sampling method has the fastest run-time when the number of test vectors are increased. It can be seen that our estimation method also grows more slowly than the traditional fault simulation. For IVM test case, FALCON works faster than fault sampling method. This is while we only use a small subset of faults to simulate. We believe that FALCON will run even faster than fault sampling with smaller error rates if we inject more faults in our design.

B. *IVM Case Study*

In IVM test case, we have applied random test vectors, which are valid instructions. As you can see in Table V, we have run fault simulation, fault sampling, and coverage estimation methods on this test case for 50, 200, 500, 1000, 2000, and 5000 clock cycles.

Also in this case, we have chosen a subset of faults for this processor and we have not simulated all the faults. This is because the fault simulation process could not be finished in a reasonable time even for a small number of cycles when all the faults are injected in the circuit. Similar to OR1200 case (Section III-A), we have measured results for coverage and run-time for fault simulation, fault sampling, and coverage estimation. Fault coverage results are shown in Table V and run-time results can be found in Table VI.

As it can be seen in Table VI, in this test case, fault sampling takes longer times than our coverage estimation method and as it shows in Table V, the error between fault sampling and fault simulation is higher than the error between our coverage

TABLE III
FAULT COVERAGE RESULTS FOR OR1200

# of Test Vectors	Fault Sim. Coverage(%)	Fault Sampl. Coverage (%)	FALCON Coverage (%)	FALCON Error (%)	Sampl. Error (%)	FALCON misdetected	Sampl. misdetected	Error Difference
1,024	39.57	[28.4, 30.9]	38.99	0.96	9.92	722	7453	6731
2,048	42.99	[35.4, 38.1]	42.51	0.97	6.24	729	4689	3960
4,096	45.82	[38.1, 40.8]	46.65	1.46	6.37	1097	4786	3689
8,192	53.69	[52.4, 55.1]	54.39	2.34	0	1759	0	-1759
12,288	57.63	[59.0, 61.7]	58.42	2.24	2.72	1683	2044	361
16,384	60.1	[59.0, 61.7]	61.41	2.11	0	1586	0	-1586

TABLE IV
RUN-TIME RESULTS FOR OR1200

# of Test Vectors	Fault Sim. Run-time	Fault Sampl. Run-time	FALCON Run-time	Speedup (Fault Sim. time/FALCON time)
1,024	846	100	386	2.19
2,048	1,608	130	679	2.37
4,096	4,341	387	1,488	2.92
8,192	3,788	751	1,470	2.58
12,288	15,608	1275	4,672	3.34
16,384	18,578	1792	6,125	3.03

TABLE V
FAULT COVERAGE RESULTS FOR IVM

# of Test Vectors	Fault Sim. Coverage(%)	Fault Sampl. Coverage (%)	FALCON Coverage (%)	FALCON Error (%)	Sampl. Error (%)	FALCON misdetected	Sampl. misdetected	Error Difference
50	15.9	[19.6, 20.9]	15.3	0.82	4.35	2,632	13,960	11,328
200	21.8	[26.6, 29.1]	19.93	2.01	6.05	6,451	19,416	12,965
500	41.4	[47.4, 49.0]	39.46	2.1	6.8	6,740	21,823	15,083
1,000	45.0	[50.6, 52.2]	43.44	1.89	6.4	6,066	20,539	14,473
2,000	49.5	[55.51, 57.08]	48.4	1.21	6.8	3,884	21,823	17,939
5,000	53.4	[57.91, 59.48]	51.88	1.81	5.3	5,809	17,009	11,200

TABLE VI
RUN-TIME RESULTS FOR IVM

# of Test Vectors	Fault Sim. Run-time	Fault Sampl. Run-time	FALCON Run-time	Speedup (Fault Sim. time/FALCON time)
50	13,841	3,273	610	22.6
200	30,057	6,370	1,232	24.3
500	77,833	11,665	4,762	16.34
1,000	111,984	13,762	8,910	12.5
2,000	243,780	52,851	18,795	12.97
5,000	477,859	61,188	40,090	11.91

estimation method and fault simulation (Table V).

Similar to OR1200 case, we have shown run-times, fault coverages, and coverage errors for fault simulation, fault sampling, and coverage estimation in IVM processor. Figure 11 shows the run-times for the three methods and errors in coverage for fault sampling and coverage estimation. The run-times are shown in logarithmic scale and the error is shown by the number of faults.

As it can be seen in Figure 11 for both OR1200 and IVM cases, the run-time in our method, due to its scalability, grows in a slower rate than fault simulation. Also, it can be seen that our method runs faster than fault sampling with the growth of the design size with less error rate.

As it can be shown in Figure 11, fault coverages calculated by fault sampling in IVM case study is always more than the real fault coverage, while fault coverage estimated by FALCON is always less than the real fault coverage. This can be counted as an advantage for our method since it estimates the coverage in a more conservative and pesimistic way.

As another advantage of FALCON, we can determine which faults are detected on which outputs. This is useful when the user needs more data than a simple coverage number (e.g. fault diagnosis).

As we can see in the above two test cases, the run-time of our estimation method grows faster than fault sampling, however it is still comparable to fault sampling for large designs. The main reason for this grow rate in coverage estimation is that we do not drop the faults during our process, which can have its own applications. In cases that we do not need the results without fault dropping, we can divide our test vector set into sub-sets of test vectors and apply our method step-by-step for each sub-set of test vector. In each step, we can drop the detected faults. This way, we reduce the time of our stand-alone fault simulations.

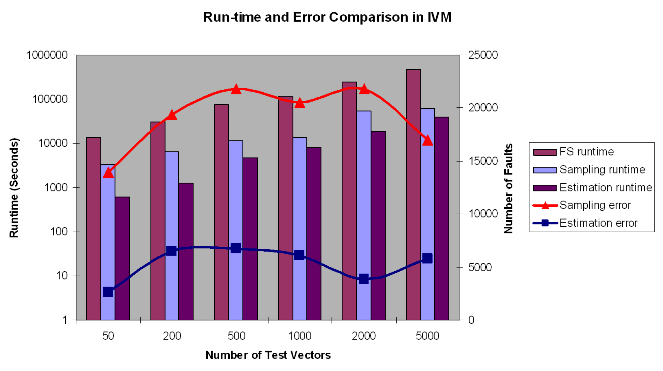


Fig. 11. IVM run-time results and misdetected faults

From our experimental results, we can say that fault sampling is a great method for estimation fault coverage for small to medium designs. It is still a good way to roughly estimate fault coverage for larger designs. However, this method does not provide data other than fault coverage. On the other hand, FALCON works a lot faster than fault simulation. Although it works slower than fault sampling for small to medium designs, it becomes faster than fault sampling for larger deisgns. In addition, FALCON provides more information about fault detection which can be useful during the test process.

IV. Run-time Analysis

In this section, we discuss a trivial run-time complexity analysis for our estimation method and compare it with run-time analysis of fault simulation.

We can use the following symbols in our analysis:

- M: number of modules in the system
- T: number of test vectors
- f_m: number of faults in an MUT m
- G: number of gates in the system
- g_m: number of gates in MUT m
- $i_{max} \times o_{max}$: maximum module input/output product

Using the above definitions, the complexity of each step of our algorithm can be expressed as follows, supposing we are estimating the fault coverage of module m in our system:

- Good simulation: $O(G \times T)$
- Stand-alone fault simulation: $O(g_m \times f_m \times T)$
- Propagation table making: $O(I \times T)$,where I is total number of module inputs
- Detection probability table making: $O(f_m \times T)$
- Making testbenches: $O(M)$ (this usually takes only a few seconds)
- Statistical simulation: $O(f_m\prime \times M \times i_{max} \times o_{max})$, where $f_m\prime$ is the number of detected faults in stand-alone fault simulation. In worst case $f_m = f_m\prime$

All of the above should be done for coverage estimation of module m. Therefore, the runtime of coverage estimation can be written as:

$$O(G \times T + g_m \times f_m \times T + I \times T + f_m \times T + M + f_m \times M \times (i_{max} \times o_{max})) \quad (3)$$

If we want to inject all of the faults in our MUT, the number of faults are linearly related to the number of gates. Therefore, we can replace f by g in formula 3. We also can remove $I \times T + M$ part since it is negligible compared to other parts. The statistical simulation part ($g_m \times M \times (i_m \times o_m)$) is not negligible if all faults propagate through all inputs of every module. Since every fault usually affects a limited part of the design, it will propagate through a few of the module paths. Therefore, using $i_{max} \times o_{max}$ in our formula is unrealistic since this term can be easily replaced by a small constant. On the other hand, M is also a relatively small number and the whole product of $M \times i_{max} \times o_{max}$ can be replaced by a constant. As a result, the estimation run-time can be written as:

$$Estimation\ run\ time = O(G \times T + g_m^2 \times T + g_m) \quad (4)$$

which can be written as:

$$Estimation\ run\ time = O(G \times T + g_m^2 \times T) \quad (5)$$

Equation 5 shows that the run-time of FALCON mostly depends on the time of good simulation and the time of local fault simulation for module m.

On the other hand, the complexity of fault simulation for a module with g_m gates can be written as:

$$FS\ run\ time = O(f_m \times G \times T) = O(g_m \times G \times T) \quad (6)$$

If we compare each part of equation 5 with equation 6, we can see that fault simulation run-time is proportional to $g_m \times G$, while coverage estimation run-time is proportional to G or g_m^2 and in each case coverage estimation is a smaller number.

An Example: Suppose we have a design with 5 million gates (G) and 10,000 test vectors (T). If we want to perform coverage estimation on a module with 50,000 gates (g_m), then we have the following run-time estimations for coverage estimation and fault simulation (based on equations 5). Since in complexity analysis we deal with the biggest exponent, we can write:

$coverage\ \ estimation\ \ run\ \ time = SomeConstant \times max\{5 \times 10^{10}, 25 \times 10^{12}\}$

or:

$coverage\ estimation\ run\ time = SomeConstant \times 25 \times 10^{12}$

On the other hand, based on equation 6, we can write:

$fault\ simulation\ run\ time = SomeConstant \times 5 \times 10^6 \times 5 \times 10^4 \times 10^4$

which can be written as:

$fault\ simulation\ run\ time = SomeConstant \times 25 \times 10^{14}$

As we can see, our estimation method can work around **100** times faster than fault simulation. For example, if coverage estimation takes a few minutes, we can expect *hours* for fault simulation or if FALCON takes an hour, we can expect *days* for fault simulation.

Due to the above analysis, if g_m is close to G (which means g_m is a big module in the design), our estimation method will be as time-consuming as fault simulation. If we have such modules in the design, we need to break them down into smaller modules and apply the algorithm on these smaller modules. Fortunately, with today's hierarchical designs, every module has its own sub-modules. Therefore, we can use the sub-modules of large modules under test as our new modules under test and apply our technique hierarchicaly to the design.

V. Conclusions

We have developed a scalable and modular technique for estimating fault coverage (FALCON). Currently, this method is evaluated for single-stuck-at faults, but the techniques can be extended to any fault model. Our experimental results show that for large designs, we can reach orders of magnitude improvements in time with a very small amount of error. Our estimation method works the best when each module in the design is a few times smaller than the whole design. For large modules, we can simply break them into smaller modules and use these modules in our estimation system instead. FALCON works on both combinational and sequential modules. This method can be used even before completing the design, when we do not have every module at the gate level of abstraction. Future work will include analysis for error bounds.

References

[1] M. Abramovici, M. Breuer, and A. Friedman. *Digital systems testing and testable design*. IEEE press New York, 1990.

[2] V. Agrawal. Sampling techniques for determining fault coverage in lsi circuits. volume 5, pages 189–202, 1981.

[3] V. Agrawal. Fault sampling revisited. volume 7, pages 32–35. IEEE, 1990.

[4] V. Agrawal, S. Bose, and V. Gangaram. Upper bounding fault coverage by structural analysis and signal monitoring. In *VLSI Test Symposium, 2006. Proceedings. 24th IEEE*, pages 6–pp. IEEE, 2006.

[5] H. Bhatnagar. Verifault-xl user's guide.

[6] S. Bose and V. Agrawal. Estimating stuck fault coverage in sequential logic using state traversal and entropy analysis. In *Test Conference, 2007. ITC 2007. IEEE International*, pages 1–10. IEEE, 2007.

[7] A. Carbine and D. Feltham. Pentium (r) pro processor design for test and debug. In *Test Conference, 1997. Proceedings., International*, pages 294–303. IEEE, 1997.

[8] C. Chen and N. Soong. A statistical model for fault coverage analysis. In *VLSI Test Symposium, 1991.'Chip-to-System Test Concerns for the 90's', Digest of Papers*, pages 227–232. IEEE.

[9] G. Ganapathy and J. Abraham. Hardware acceleration alone will not make fault grading ulsi a reality. In *Test Conference, 1991, Proceedings., International*, page 848. IEEE, 1991.

[10] K. Heragu, V. Agrawal, and M. Bushnell. Facts: fault coverage estimation by test vector sampling. In *VLSI Test Symposium, 1994. Proceedings., 12th IEEE*, pages 266–271. IEEE, 1994.

[11] K. Heragu, V. Agrawal, and M. Bushnell. Statistical methods for delay fault coverage analysis. pages 166–170, 1995.

[12] S. Jain and V. Agrawal. Stafan: An alternative to fault simulation. In *Papers on Twenty-five years of electronic design automation*, pages 475–480. ACM, 1988.

[13] S. Karthik, M. Aitken, L. Martin, S. Pappula, B. Stettler, P. Vishakantaiah, M. d'Abreu, and J. Abraham. Distributed mixed level logic and fault simulation on the pentium (r) pro microprocessor. pages 160–166. IEEE, 1996.

[14] V. Kim, T. Chen, and M. Tegethoff. Fault coverage estimation for early stage of vlsi design. In *VLSI, 1999. Proceedings. Ninth Great Lakes Symposium on*, pages 105–108. IEEE, 1999.

[15] H. Ma and A. Sangiovanni-Vincentelli. Mixed-level fault coverage estimation. In *Proceedings of the 23rd ACM/IEEE Design Automation Conference*, pages 553–559. IEEE Press, 1986.

[16] P. Maxwell, R. Aitken, V. Johansen, and I. Chiang. The effect of different test sets on quality level prediction: When is 80% better than 90%. In *Proc. Int. Test Conf*, pages 358–364, 1991.

[17] P. Maxwell, I. Hartanto, and L. Bentz. Comparing functional and structural tests. In *Test Conference, 2000. Proceedings. International*, pages 400–407. IEEE, 2000.

[18] W. Meyer and R. Camposano. Fast hierarchical multi-level fault simulation of sequential circuits with switch-level accuracy. In *Proceedings of the 30th international Design Automation Conference*, pages 515–519. ACM, 1993.

[19] B. Murray and J. Hayes. Testing ics: Getting to the core of the problem. volume 29, pages 32–38. IEEE, 1996.

[20] T. Niermann, W. Cheng, and J. Patel. Proofs: A fast, memory-efficient sequential circuit fault simulator. volume 11, pages 198–207. IEEE, 1992.

[21] U. of Illinois at Urbana-Champaign. Ivm processor. http://www.crhc.illinois.edu/ACS/tools/ivm/about.html.

[22] OpenCores. Opencores webpage. http://opencores.org/openrisc,or1200.

[23] P. Parvathala, K. Maneparambil, and W. Lindsay. Frits-a microprocessor functional bist method. In *Test Conference, 2002. Proceedings. International*, pages 590–598. IEEE, 2002.

[24] W. Qiu, X. Lu, J. Wang, Z. Li, D. Walker, and W. Shi. A statistical fault coverage metric for realistic path delay faults. In *VLSI Test Symposium, 2004. Proceedings. 22nd IEEE*, pages 37–42. IEEE, 2004.

[25] C. Ravikumar and H. Joshi. Hiscoap: a hierarchical testability analysis tool. page 272. Published by the IEEE Computer Society, 1995.

[26] C. Ravikumar, G. Saund, and N. Agrawal. A stafan-like functional testability measure for register-level circuits. In *Test Symposium, 1995., Proceedings of the Fourth Asian*, pages 192–198. IEEE, 1995.

[27] D. Saab, R. Mueller-Thuns, D. Blaauw, J. Rahmeh, and J. Abraham. Hierarchical multi-level fault simulation of large systems. volume 1, pages 139–149. Springer, 1990.

[28] S. Seshu and D. N. Freeman. The diagnosis of asynchronous sequential switching systems. volume EC-11, pages 459–465, 1962.

[29] P. Thaker, V. Agrawal, and M. Zaghloul. Register-transfer level fault modeling and test evaluation techniques for vlsi circuits. International Test Conference, 2000.

[30] A. Vij. Good scan= good quality level? well, it depends. In *Test Conference, 2002. Proceedings. International*, page 1195. IEEE, 2002.

[31] R. Wieler, Z. Zhang, and R. McLeod. Emulating static faults using a xilinx based emulator. In *fccm*, page 0110. Published by the IEEE Computer Society, 1995.