A Concurrent Fault Simulation for Crosstalk Faults in Sequential Circuits *

Marong Phadoongsidhi
Dept. of Electrical & Computer Engineering
University of Wisconsin - Madison
U.S.A.
phadoong@ece.wisc.edu

Kim T. Le
School of Computing
University of Canberra
Australia
kim.le2@ise.canberra.edu.au

Kewal K. Saluja
Dept. of Electrical & Computer Engineering
University of Wisconsin - Madison
U.S.A.
saluja@engr.wisc.edu

Abstract

Existing principle for crosstalk fault simulation requires the storage of waveform representation at each node in the circuit throughout a time frame. At the end of each time frame a pair of waveforms, one belonging to an aggressor node, and one depicting a victim node, is inspected. If the fault is captured, it will be simulated until it is either detected or the test vectors are exhausted. This fault detection method can require a prohibitive amount of computation time for a large sequential circuit with high number of possible fault pairs to be tested. With our simulation technique introduced in this paper, these operations can be processed concurrently for many faults. The fault list dynamically adjusts itself during the simulation to accommodate fault injection and fault dropping. Experimental results on ISCAS'89 benchmark circuits show that a substantial improvement in CPU time over a conventional method is achieved with a trade-off in the amount of memory consumed.

1. Introduction

Due to a rapid progress in a VLSI fabrication technology, crosstalk coupling-induced faults in digital circuits have recently been receiving substantial attention by the research community and the industry. Higher capacitance coupling along with shorter signal transition time in today's high-performance circuits are prime suspects for the increasing role of a crosstalk coupling effect. Aggressive circuit scaling makes the undesirable effect of interconnects capacitive coupling even more pronounced. All of these factors put a crosstalk phenomenon in high performance processors as one of the most serious problems that needs to be addressed during the design phase as well as during the testing period.

Crosstalk coupling refers to an effect of signal transitions on some lines (dubbed *aggressors*) inducing unwanted transitions on the other lines (*victims*). Crosstalk may cause a slow-down or speed-up in signal propagation, or may generate pulses on victim lines. Theoretical analysis and experimental results show that the effect of crosstalk-induced coupling varies with device size (e.g., strength of line drivers, load and coupling capacitances), timing (rise/fall time, transition occurring time, etc), and fabrication process parameters [3]. For crosstalk pulses, the amplitude can attain a value greater than $Vdd/2$ [1]. Because of aggressive clocking schemes and small noise margins of modern designs, this magnitude is more than large enough for these pulses to alter the operation of circuits.

In case of a crosstalk-inducing pulse on a victim being a clock line, a logic value may be latched into one or more flip-flops before the end of a clock cycle. If the value which is prematurely captured is identical to the value to be latched at the beginning of the next clock period, the circuit will continue to function correctly. However, if this is not the case, wrong logic values will appear at the outputs or at the pseudo primary inputs (flip-flops). If flip-flops capture these incorrect values, these may lead to an eventually erroneous outcome of the circuit.

Although crosstalk faults may arise in a circuit without causing any malfunction, faults that are perhaps sources of malfunction should be detected early in the design cycle so that appropriate layout modifications can be made to address the problems. A crosstalk fault simulator can prove to be a valuable tool for circuit designers to strategically verify their work throughout a design phase in order to minimize or avoid costly layout revisions as dictated by crosstalk couplings.

For larger circuits, the number of all possible combination of aggressors and victims lines can be daunting, given the computational resources test engineers might have at their disposal. Nevertheless, information about devices and interconnects topology extracted from the layout can be utilized to produce an accurate picture of sites where crosstalk couplings are inevitable or otherwise, giving a more manageable number of suspect faults that needs to be tested. In addition, some fault reduction technique, such as the one proposed in [5], may be used to cope with an immense amount of fault permutations typical of complex circuits. Also, a crosstalk fault simulator can be integrated into a

*This work was completed when Prof. Le was at UW-Madison

Proceedings of the 11 th Asian Test Symposium(ATS'02)
1081-7735/02 $17.00 © 2002 IEEE

IEEE COMPUTER SOCIETY

conventional test generation or a diagnosis system [7] to create a more complete VLSI testing suite.

The main purpose of this paper is to give an efficient implementation of a crosstalk-pulse fault simulator. The paper is organized as follows. We first visit a previous design of a crosstalk fault simulator and study its underlying principle as well as its weaknesses. Next, we propose ideas of how our implementation of the simulator is capable of addressing those issues. In Sections 3, we present the outline of our simulator and the simulation algorithm with an example to illustrate the proposed algorithm. Experimental results and relevant discussions are presented in Section 4. We finally conclude the paper in Section 5.

2. Previous Work and Contributions

A simulator for crosstalk-pulse faults in sequential circuits was developed and presented in [4]. A simple update of this simulator with modifications suitable for crosstalk fault diagnosis tool is described in [7]. The construct of a 3-value crosstalk fault simulator fundamentally lies in the way changes in logic values are represented at a node (i.e. gate/flip-flop outputs, primary inputs and outputs) in each time frame. Logic values of a node at different time steps are stored as a bitmap (Figure 1) to signify the logic waveform at the node for a particular time frame. Upon the conclusion of each time frame, a pair of waveforms from an aggressor node and a victim node (i.e., fault pair) is examined in order to determine whether the pair can be considered as a crosstalk pulse fault candidate. If a pulse-generating condition is met, re-simulation of this time frame with this particular fault injected to the circuit will be necessary in order to determine if the injected crosstalk fault will be detected. This process is repeated for each time frame until either the fault of interest is detected or the test vectors are exhausted.

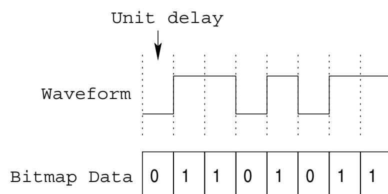

Figure 1. Waveform of Logic Values

To keep memory usage reasonable, the conventional simulator utilizes bitmap data only for nodes belonging to the two fan-in cones. The first cone roots back from the output of an aggressor toward primary inputs, and the second cone originates from the data input line of the victim flip-flop back to the primary inputs. Due to this arrangement, the simulator must start from the first vector of a test sequence for every fault being tested as data calculated for the simulation of one fault cannot be reused to simulate another fault. For a larger circuit with a significant amount of faults to be tested, the cost measured by the amount of CPU time can quickly become intolerable if only one fault can be injected per pass of a test vectors sequence.

To combat excessive amount of computation time required by the conventional crosstalk-pulse fault simulator, we develop a new simulator which employs an algorithm capable of processing multiple faults in each time frame. Even thought the amount of memory consumed by the simulator (which is a function of the size of a fault batch to be simulated concurrently) can be high if the fault batch is chosen to be large, the reduction in total execution time can be significant. As the experiments suggest, the trade-off between memory usage and CPU time is well worthwhile.

In the following section, the new crosstalk-induced pulse fault and the algorithm associated with it will be discussed in detail. An example of how the algorithm works will also be given to assist interested readers in understanding the simulator.

3. Fault Simulator and Simulation Algorithm

We will give a high level description of our concurrent crosstalk fault simulator and the outline of the processes involved. For a more elaborate view of a conventional bitmap-based crosstalk fault simulator, readers are advised to refer to [4] and [7]. Next, in sections 3.2 and 3.3, two operations that are of essence for the crosstalk fault simulation are described in detail. A simple example to illustrate key simulation ideas are given in section 3.4.

3.1. Simulator Overview

A crosstalk fault simulator can be thought of as a regular sequential circuit logic simulator with fault injection and fault detection capability built into it. In a regular logic simulator, each node in the circuit only needs to be allocated a memory space to store a logical value of 0, 1, or an unknown value U. Straightforward modification of a logic simulator can be made to convert a logic simulator to a fault simulator for stuck-at type faults.

Unfortunately, the situation is more complex in case of a crosstalk-pulse fault simulator. Transitions (waveforms) at each node of a circuit need to be recorded for inspection at the end of each time frame. A *time frame* is composed of a finer-grain time units called time steps. A *time step* corresponds to how long it takes for a logic value to propagate from an input to an output of a gate. For this experiment, we set this value to be one (i.e. a unit delay), but it can be altered to adapt to a different gate delay model. The number of time steps in a time frame equals to the maximum number of times a particular signal must travel through gates, from a primary (or pseudo-primary) input before it reaches

IEEE COMPUTER SOCIETY

a primary (or pseudo-primary) output. This number therefore corresponds to maximum depth of the circuit.

Integer-type storage (strictly speaking, 2 integer buffers per node for a 3-value logic) is assigned to each node. Since in most architectures an integer is 32-bit long, the maximum number of levels a circuit can have is limited by this number (although *long int* or a concatenation of integers can be used to extend this limit to 64 or more). For a unit delay time, the change in logic value throughout the time frame is illustrated in Figure 1.

Each node in the circuit is assigned two sets of buffers. The first set, *normal buffers*, is used to store waveforms during the fault-free logic simulation for each time frame. The second set, dubbed *fault buffers*, is allocated for storing waveforms during faulty machine simulation. The number of fault buffers allotted to each node corresponds to the *batch size*, the number of faults that can be simulated on a single pass of the test sequence. The batch size is related to the number of nodes that act as aggressors of the current simulation run, and is a product of the number of aggressor and the number of flip-flops (victims). Although other combinations are possible, our simulator takes an aggressor as either a primary input, primary output, or output of a gate in the circuit. The victim can only be a flip-flop (a clock line to a flip-flop, to be precise).

For each simulation iteration, the simulator goes through all the test vectors until all faults of the current batch are detected or the test vectors are exhausted. Hence, the number of iterations is inversely proportional to the size of the fault batch that is set per iteration. Fewer numbers of iterations implies less CPU time required for fault simulation, although the memory consumption grows as a function of gates in the circuit for each fault added to the batch.

For each time frame, the simulator starts with a logic simulation of the fault-free circuit concurrently with faulty circuit simulation. At the end of the time frame, the simulator checks primary outputs for any fault detection. Also, *state* (i.e., primary and pseudo-primary outputs) of each faulty machines is examined. If a faulty machine state converges to a good machine state, this faulty machine will be deactivated, and will be re-considered during the fault-capturing check (section 3.2).

It will become clearer in the following sections the importance of logic waveforms at node buffers in fault capturing and fault simulation.

3.2. Fault Capturing Criteria

At the end of a fault-free simulation for each time frame, each aggressor-victim pair from the fault batch is checked in order to determine if it has met the condition that would create a valid pulse on a victim flip-flops clock line. A pulse on the victim line is generated (via crosstalk-coupling effect) by a transition of a logic level on an aggressor. For the pulse to be valid, the transition to a different logic value on an aggressor needs to hold on to the new level long enough before the next transition can take place (if any). This duration is specified as a *pulse width* (PD in Figure 2 and 3). For our purpose, this value is set to one unit, although it can be altered to suit the technology. For flip-flops that capture their inputs at a positive edge of a clock signal, there are two conditions in which a valid pulse on a victims clock line is generated:

[Pulse-Generating Condition 1] During the time when the clock signal is at a logical 1 (i.e, the *on* duty phase), a 1→0 (*down*) transition (at time T_{ON}) on an aggressor will induce a down transition on the victims clock line. After a period of previously defined pulse width, the signal on the clock line will restore itself, thus creating a pulse (Figure 2). The negative transition on an aggressor has to be at least one unit of a pulse width before the clock enters the off duty phase. If this criterion is not met, then a down transition on an aggressor will not generate a pulse on the victim clock line

[Pulse-Generating Condition 2] During the time when the clock signal is at a logical 0 (i.e, the *off* duty phase), a 0→1 (*up*) transition (at time T_{OFF}) on an aggressor will induce an up transition on the victims clock line. This up transition on the aggressor must be at least one unit of a pulse width before the clock enters the on duty phase (Figure 2).

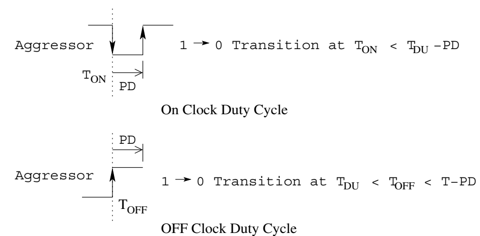

Figure 2. Pulse Generating Conditions

If *either* one of the above conditions is satisfied, a spurious pulse is said to be validly generated on the clock line of a victim flip-flop. Thus, a crosstalk-induced positive transition on the victim line will capture the input value of the flip-flip as its output value. However, in order to determine the captured value, the following constraint must not be violated:

[Timing Constraint] If the input of a flip-flop is held to a constant logic value for at least a time period which satisfies the flip-flop's *setup* and *hold* time, a crosstalk-induced pulse on the flip-flop's clock line will cause the output of the flip-flop to acquire the same logic value as its input.

Although one of the pulse-generating conditions is satis-

IEEE COMPUTER SOCIETY

fied, it is possible that the faulty value does not propagate to the flip-flop's output. Specifically, the capturing becomes redundant if the logic value at the flip-flop's input is identical to the value at the flip-flop's output at the time the capturing occurs.

[Fault Capturing Criteria] The fault due to a crosstalk-inducing pulse on a flip-flop's clock line is said to be *captured* if and only if the following two conditions hold:

1. Either one of the two pulse-generating conditions is satisfied.
2. Logic value at the flip-flop's input is *different* than the value at the flip-flop's output.

If the Timing Constraint is not violated, the fault is considered *actual-captured*. Otherwise, the fault is considered *potential-captured* (Figure 3).

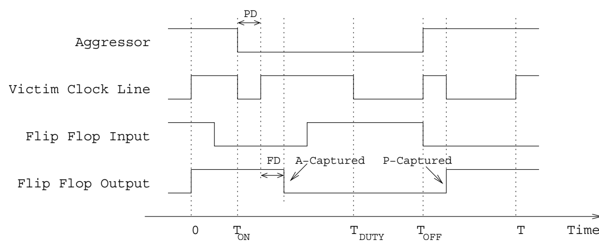

Figure 3. Fault Capturing

3.3. Fault Simulation and Detection

If the fault of interest is captured according to the criteria outlined in the previous section, the machine corresponding to this fault will be activated. First, the faulty machine is created by replication of the good circuit (i.e., node buffers associated with this fault are assigned values from corresponding buffers belong to the good circuit). To complete the activation of our new *copy* of faulty circuit, the faulty machine's victim buffer *from* the time step at which the capturing takes place are overwritten by the captured value. This procedure is called *fault injection*.

The current time frame of this faulty circuit is re-simulated (along with other faulty circuits in case there is more than one fault being captured). At the end of time frame, primary outputs of each faulty circuit are compared to those belong to a normal circuit. If *each* primary output of a faulty circuit matches the corresponding primary output of normal circuit, it is said that the fault is *not detectable* in this time frame. This faulty circuit will thus be simulated along with the normal circuit in the next time frame. However, if there is *at least one* mismatch between the primary outputs of the normal and faulty circuit, the fault is said to be *detected*. If this fault has been marked as an actual-capture, it is recorded as being actual-detected, and is dropped from the simulation. Otherwise, the fault is recorded as potential-detected before being re-labeled as *not-detected*, hoping that the fault will be actual-capture in the future (which could eventually lead to a true detection).

It is important to note that if at the end of the time frame re-simulation the *state* of a faulty circuit becomes identical to that of a good circuit, then it becomes unnecessary to further simulate this faulty machine. Once the *state reconvergence* happens, the faulty circuit's behavior is indistinguishable from the good circuit. Thus, the faulty machine can be deactivated until the next capturing by this crosstalk fault.

Figure 4 summarizes the fault simulation algorithm as described in the previous sections.

3.4. Example

The following example will help clarify the key ideas of the algorithm. Figure 5 depicts a part of a circuit in which there are nine aggressors and one victim. Assuming that the fault batch size equals to the number of aggressors, each node is therefore allocated one normal buffer and nine fault buffers. In this case, a good circuit and 9 other faulty circuits can be concurrently simulated at a time. Fault buffer No.1 is assigned to fault *A/J*, fault buffer No.2 to *B/J*, and so on. Assume that there are 4 test vectors to be used, the simulation will be run for 4 time frames.

At time frame Tf_0, fault *B/J* (2) and *E/J* (5) satisfy the actual and potential fault capturing criteria, respectively. Hence, fault buffer No.2 and No.5 of each node are activated and setup. After re-simulation of time frame Tf_0, *E/J* is detected while *B/J* is not. Being potentially captured, fault *E/J* will be recorded as potentially detected fault before its status is changed to not-detected. As a result, fault buffer No.5 will be deactivated, and will not be involved in the simulation of the next time frame. However, since *B/J* has been an actual-capture, fault buffer No.2 will be processed during the simulation of the next time frame. At the end of time frame Tf_2, fault *B/J* has yet to be detected, and the simulation proceeds to the next time frame. At the end of time frame Tf_3, another fault (*D/J*) meets the actual capturing criterion, hence fault buffer No.4 of each node is activated, and this faulty circuit is re-simulated. *D/J* is not detected but remains activated together with *B/J* at the end of Tf_2 time frame re-simulation. Finally, *B/J* is detected in time frame Tf_3, and is therefore dropped from the simulation.

4. Experimental Results

We implemented our concurrent fault simulator in the C++ programming language. All the experiments are conducted on an Intel P4 1.8 GHz Linux workstation, with

IEEE COMPUTER SOCIETY

```
Suspect_List = {All crosstalk faults to be simulated}
A_capture_list = A_detect_list = φ
P_capture_list = P_detect_list = φ
n = 0

while [TFn not exceeding test sequence length] {
   simulate good circuit
   for each [fault Fi in A_capture_list] {
     simulate Fi
     if [Fi == detected] {
        remove Fi from A_capture_list
        add Fi to A_detect_list
     }
     else if [STATE(Fi) == STATE(good circuit)] {
         remove Fi from A_capture_list
         put Fi back to Suspect_list
     }
   }
   for each [fault Fj in P_capture_list] {
     simulate Fj
     if [Fj == detected] {
         copy Fj to P_detect_list, Suspect_list
         remove Fj from P_capture_list
     }
     else if [STATE(Fj) == STATE(good circuit)] {
         remove Fj from P_capture_list
         put Fj back to Suspect_list
     }
   }
   for each [fault Fk in Suspect_list] {
     check the capture of Fk
     if [Fk == Actual_captured] {
         remove Fk from Suspect_list
         add Fk to A_capture_list
         re-simulate Fk
         if [Fk == detected]{
            remove Fk from A_capture_list
            add Fk to A_detect_list
         }
         else if [STATE(Fk) == STATE(good circuit)] {
            remove Fk from A_capture_list
            put Fk back to Suspect_list
         }
     }
     if [Fk == Potential_captured] {
         remove Fk from Suspect_list
         add Fk to P_capture_list
         re-simulate Fk
         if [Fk == detected] {
            copy Fk to P_detect_list, Suspect_list
            remove Fk from P_capture_list
         }
         else if [STATE(Fk) == STATE(good circuit)] {
            remove Fk from P_capture_list
            put Fk back to Suspect_list
         }
     }
   }
   increment n
}
```

Figure 4. Fault Simulation Algorithm

512MB of physical memory. We ran our simulator on several ISCAS'89 [2] benchmark circuits, using vectors provided by FASTEST [6] as test sequences. For comparison purpose, we ran the same benchmark circuits and test sequences on the conventional crosstalk fault simulator originally used in [4] and updated in [7]. Table 1 summarizes the information about benchmark circuits and test sequences used in the experiments. Table 2 and 3 present experimental results for the conventional and our new concurrent fault simulators. Column 2 of the tables shows the total number of crosstalk aggressor-victim pairs simulated. The third and forth columns give the number of faults potentially and actually detected by the fault simulations. The sums of faults detected are given in column 5. Column 6 shows the percentage of the total faults detected with respect to the number of faults included in the simulation (as specified in the second column). The execution time of each circuit is shown in column 7. The memory usage is also provided for our concurrent simulator (memory consumption of the conventional simulator is not included as it never exceeds 3 MB, hence it will not give a meaningful comparison to our simulator). For circuits s5378 and s35932, the simulation was done on a set of 100 randomly selected faults to ensure that the simulation be completed within a reasonable amount of time.

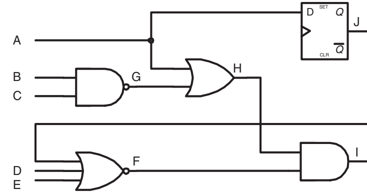

Figure 5. Sample Circuit

Table 1. Circuits and Test Vectors Information

Circuit	No. PI	No. PO	No. FF	No. Gates	Max Level	Test Seq. Length
s298	3	6	14	119	9	105
s344	9	11	15	159	11	82
s349	9	11	15	161	20	94
s382	3	6	21	158	9	77
s386	7	7	6	159	11	108
s444	3	6	21	181	11	66
s820	18	19	5	289	10	15
s953	16	23	29	395	16	13
s1196	14	14	18	529	24	351
s1238	14	14	18	508	22	342
s1488	8	19	6	653	17	158
s1494	8	19	6	647	17	128
s5378	35	49	179	2779	25	907
s35932	35	320	1728	16065	29	497

We compare some aspects of the conventional and our new simulator in Table 4. Column 2, 3, and 4 show the number of potential, actual, and total faults detected as ratios of the conventional simulator to the concurrent simulator. The ratios of fault coverage as described previously is given in column 5. Finally, the ratio of the CPU time of the old simulator to the new one is displayed in the last column.

We notice that the fault coverage in our simulator is generally lower than the conventional crosstalk fault simulator in smaller circuits, although the coverages are similar for s344 and s386. Yet on bigger circuits, better or similar fault coverage is obtained by our new simulator. We believe the discrepancy of the number of detected faults stems from a

IEEE COMPUTER SOCIETY

number of fundamental differences in the way the two simulators determine how faults are captured and detected. The old simulator assumes that a positive crosstalk pulse could capture a fault at any point during the whole clock period (i.e., the clock can be visualized as having close to zero duty cycle). In our case, the clock duty cycle is 50%, thus a positive pulse can possibly lead to fault capture only during the *off* period of the clock cycle. Another difference is due to timing assumption used in each simulator. The old simulator has a flip-flop hold time of 2 units whereas in our case, the hold time and setup time is one unit delay each. These two differences affect the number of faults captured. Moreover, there is a difference in the definition of potential-detection used by each simulator.

Nevertheless, the concurrent crosstalk fault simulator shows major improvement in the execution time over the original simulator in all circuits. The reduction in computation time ranges from a factor of 26 in s349 to 907 in s35932 (nearly 3 orders of magnitude).

Table 2. Conventional Crosstalk Fault Sim.

Circuit	No.of Faults	No.of P-Det	No.of A-Det	No.of Tot.Det	Coverage (%)	CPU Time (s)
s298	1792	495	505	1000	55.80	34.24
s344	2700	370	1190	1560	57.77	34.59
s349	2715	370	1239	1609	59.26	39.59
s382	3507	437	380	817	23.29	69.21
s386	1038	109	124	233	22.44	36.48
s444	3990	422	222	644	16.14	85.38
s820	1630	412	489	901	55.27	266.15
s953	12586	115	167	282	2.24	132.57
s1196	10026	1333	5076	6409	63.92	3310.07
s1238	9648	1256	4404	5660	58.66	3705.65
s1488	4080	139	3404	3543	86.83	938.77
s1494	4044	162	3224	3386	83.72	900.34
s5378	100	32	28	60	60.00	1309.5
s35932	100	0	100	100	100	13462

Table 3. Concurrent Crosstalk Fault Sim.

Circuit	No.of Faults	No.of P-Det	No.of A-Det	Tot. Det	Cov. (%)	CPU (s)	Mem (MB)
s298	1792	272	256	528	29.46	1.29	39
s344	2700	501	943	1444	53.48	1.09	58
s349	2715	441	797	1238	45.60	1.32	58
s382	3507	28	0	28	0.79	0.38	78
s386	1038	110	108	218	21.00	0.21	21
s444	3990	544	733	1277	32.00	1.67	87
s820	1630	452	613	1065	65.34	0.93	33
s953	12586	213	410	623	4.94	0.93	256
s1196	10026	908	3929	4837	48.24	103.54	198
s1238	9648	1059	3947	5006	51.89	93.73	191
s1488	4080	279	3480	3759	92.13	9.03	81
s1494	4044	335	3343	3678	90.95	8.24	80
s5378	100	16	28	44	44.00	14.01	8
s35932	100	0	98	98	98.00	14.840	45

Note that, in our simulator, the size of fault batch that can be processed per iteration is adjustable. In smaller circuits, the batch size can be made equal to the number of target faults. Consequently, only a single iteration is required to process all target faults without excessive memory consumption, thus minimizing the CPU time. However, on circuits of significant size, a careful balance between speed and memory requirement needs to be considered to achieve an optimal performance.

Table 4. Concurrent vs Conventional sim.

Circuit	Ratio P-Det (new/old)	Ratio A-Det (new/old)	Ratio Tot. Det (new/old)	Ratio Cov. (new/old)	Ratio CPU (old/new)
s298	0.55	0.51	0.53	0.53	26.54
s344	1.35	0.79	0.93	0.93	31.73
s349	1.19	0.64	0.77	0.77	29.99
s382	0.06	0.00	0.03	0.03	182.13
s386	1.01	0.87	0.94	0.94	173.71
s444	1.29	3.30	1.98	1.98	51.13
s820	1.10	1.25	1.18	1.18	286.18
s953	1.85	2.46	2.21	2.21	142.55
s1196	0.68	0.77	0.75	0.75	31.97
s1238	0.84	0.90	0.88	0.88	39.54
s1488	2.01	1.02	1.06	1.06	103.96
s1494	2.07	1.04	1.09	1.09	109.26
s5378	0.5	1.0	0.73	0.73	93.47
s35932	1.0	0.98	0.98	0.98	907.14

5. Conclusions

In this paper we proposed a concurrent simulation technique for detection of crosstalk-induced pulse faults. The new simulation algorithm introduced leads to a significant reduction in simulation time with higher memory consumption as a trade-off. Experimental results on ISCAS'89 benchmark circuits show from 1 to nearly 3 orders of magnitude reduction in CPU time.

References

[1] M. A. Breuer and S. K. Gupta. Process aggravated noise (pan): New validation and test problems. *Proc. Int. Test Conf.*, pages 914–923, 1996.

[2] F. Brglez, D. Bryan, and K. Kozminski. Combinational profiles of sequential benchmark circuits. *Int. Symp. of Circuits & Systems*, pages 1929–1934, May 1989.

[3] W. Y. Chen, S. K. Gupta, and M. A. Breuer. Analytic models for crosstalk delay and pulse analysis under non-ideal inputs. *Proc. Int. Test Conf.*, pages 809–818, 1997.

[4] N. Itazaki, Y. Idomoto, and K. Kinoshita. A fault simulator method for crosstalk faults in synchronous sequential circuits. *Proc. 26th Fault-Tolerant Computing Symp.*, pages 38–43, 1996.

[5] K. J. Keller, H. Takahashi, K. K. Saluja, and Y. Takamatsu. On reducing the target fault list of crosstalk-induced delay faults in synchronous sequential circuits. *Proc. Int. Test Conf.*, pages 568–577, 2001.

[6] T. P. Kelsey, K. K. Saluja, and S. Y. Lee. An efficient algorithm for sequential circuit test generation. *IEEE Trans. on Computers*, pages 1361–1371, 1993.

[7] H. Takahashi, M. Phadoongsidhi, Y. Higami, K. K. Saluja, and Y. Takamatsu. Simulation-based diagnosis for crosstalk faults in sequential circuits. *Proc. 10th Asian Test Symposium*, pages 63–68, 2001.