Testing the Path Delay Faults for ISCAS85 Circuit c6288

Wangqi Qiu D. M. H. Walker

Department of Computer Science
Texas A&M University
College Station TX 77843-3112
Tel: (979) 862-4387
Fax: (979) 847-8578
Email: {wangqiq, walker}@cs.tamu.edu

Abstract

It is known that the ISCAS85 circuit c6288 contains an exponential number of paths and more than 99% of the path delay faults are untestable. Most ATPG tools which can efficiently handle other circuits fail on c6288. In this paper the logic structure of c6288 is studied and the main features which cause false paths are revealed. A heuristic which significantly helps the path delay fault test generation for this circuit is presented. Experimental results show that our methodology is able to efficiently generate testable paths for c6288.

1. Introduction

Delay test detects timing defects and ensures that the design meets the desired performance specifications. Under the path delay fault model [1] a circuit is considered faulty if the delay of any of its paths exceeds the specification time. Because the number of paths in a circuit (and therefore the number of path delay faults) can be exponential in the number of gates, usually only the longest paths, which have the maximum delays, are tested. These paths are also called critical paths because they dominate the performance of the circuit.

A path is said to be testable if a rising/falling transition can propagate from the primary input to the primary output associated with the path, under certain sensitization criteria [1][2][3][4][5]. If a path is not testable, it is called an untestable or false path [6]. For example, in Figure 1, path *a-c-d* is a false path under the single-path sensitization criterion, because to propagate a transition through the AND gate requires line *b* to be logic 1 and to propagate the transition through the OR gate requires line *b* to be logic 0. In this paper the terms "untestable" and "false" are used interchangeably.

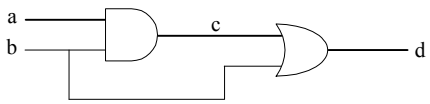


Figure 1. A circuit with a false path *a-c-d*.

Many automatic test pattern generation (ATPG) methods [7][8][9][10] for path delay faults first generate a list of long structural paths and then check their testability. Unfortunately, in most of the ISCAS85 circuits the longest structural paths are untestable. This fact results in the inefficiency of these methods. In order to increase the efficiency, NEST [11] generates paths in a nonenumerative way, which can handle a large number paths simultaneously, but it is only effective in highly testable circuits, where large numbers of path delay faults are testable. It only generates one testable path for c6288. DYNAMITE [12] improved the efficiency for poorly testable circuits, but still results in numerous aborts in the test generation for c6288. RESIST [13] exploits the fact that many paths in a circuit have common subpaths and sensitizes those subpaths only once, which reduces repeated work and identifies large sets of untestable paths. Moreover, for the first time this research identified 99.4% of all path delay faults in circuit c6288 as either testable or untestable. However, the test generation for c6288 is very slow. It took 1 122 CPU hours to find 12 592 paths on a SPARC IPX 28MIPS machine. For comparison, RESIST generated 86 250 testable paths for c7552, the largest ISCAS85 circuit, within 2 457 seconds on the same machine. It can be seen that there is a big gap between the test generation efficiency for c6288 and the other ISCAS85 circuits.

Some ATPG tools, such as RESIST, are able to generate the longest testable paths throughout the entire circuit, which are termed *global longest paths*, for c6288. But to our knowledge, no tool has successfully generated the longest paths through a particular gate or line for this circuit. In this paper it is shown that for c6288, it is much harder to generate the longest testable paths through a particular gate/line than to generate the global longest paths. However, testing the longest paths through a particular gate/line is important because it can detect the smallest extra delay on that gate/line, due to some spot defects, such as resistive opens or shorts. In this paper we present a heuristic which significantly helps the path generation for c6288. The experimental results show that with our heuristic, a path generator similar to [14] is able to efficiently generate the longest paths through each gate in c6288, with a small number of aborts.

The remainder of the paper is organized as follows. In Section 2 the logic structure of c6288 is studied and the

main features which result in false paths are revealed. Section 3 describes the heuristic and path generator we use. In Section 4 experimental results are shown and analyzed. Section 5 concludes the paper with directions for future research.

2. Logic Structure of c6288

ISCAS85 circuit c6288 is a 16×16 multiplier. Figure 2 [15] shows its structure. The circuit contains 240 adders, among which 16 are half adders, which are shaded in Figure 2. $P_{31}...P_0$ are the 32-bit outputs. Each floating line, including P_0, is fed by an AND gate, whose inputs are connected to two primary inputs.

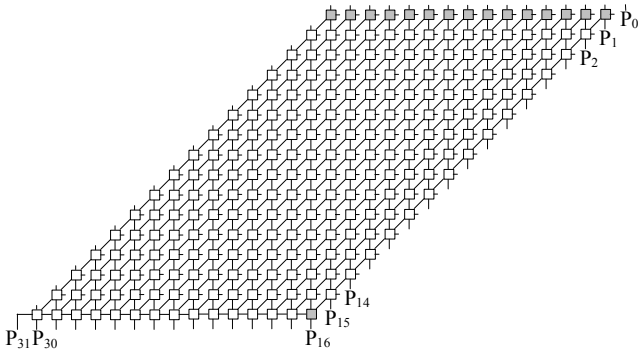
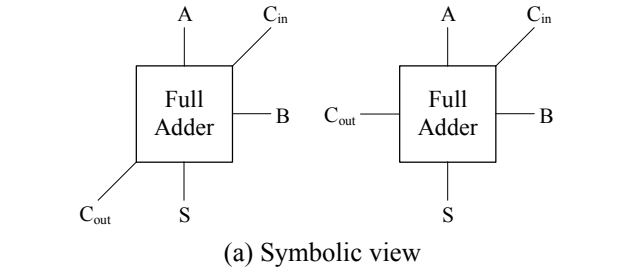
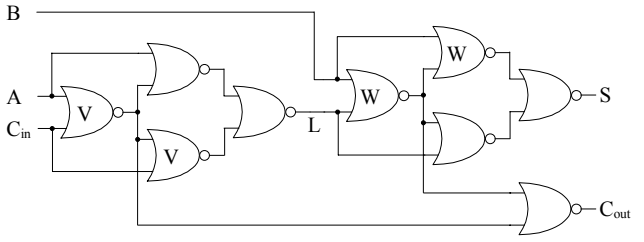


Figure 2. ISCAS85 c6288 16×16 multiplier.

Figures 3(a) and 3(b) [15] show the symbolic and schematic view of a full adder in c6288, respectively. The 15 top-row half adders in Figure 2 lack the C_{in} input. Each of them has two inverters at locations V in Figure 3(b). The single half adder in the bottom row lacks the B input, and it has two inverters at locations W.



(a) Symbolic view



(b) Schematic view

Figure 3. Full adder module in c6288.

Table I shows the delays between all the input-output pairs in the full adder, assuming every gate has one unit delay and no sensitization constraint is considered. In other words, the numbers are the structural length between the input-output pairs.

Table I. Maximum delays between input-output pairs in a full adder.

	A	B	C_{in}
S	6	3	6
C_{out}	5	2	5

The paths highlighted in Figure 4 are the longest structural paths in the circuit. These paths are all from primary input A_{15} or B_0, to output P_{30}. Each of these paths contains 124 gates, and must include the longest structural path from input A or C_{in} to output C_{out} in at least one adder.

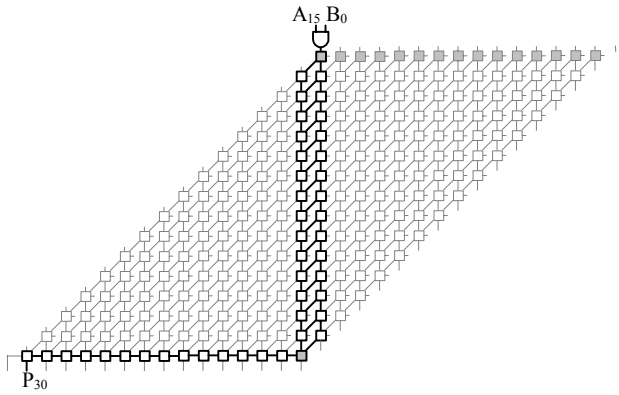
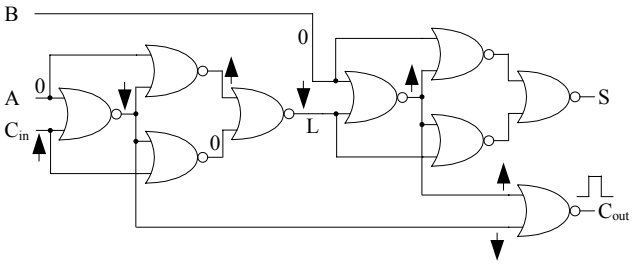
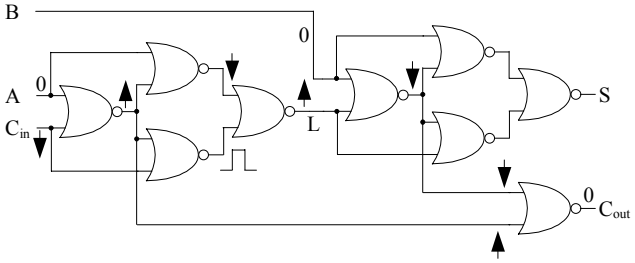


Figure 4. Longest structural paths in c6288.



(a) A rising transition at C_{in}



(b) A falling transition at C_{in}

Figure 5. Finding a robust test for the longest paths from input C_{in} to output C_{out} in a full adder.

However, there is no robust test for the longest structural paths from input C_{in} to output C_{out} in the full adder. Figures 5(a) and 5(b) show the signal propagation and necessary assignments if a rising or falling transition is applied to C_{in}. Similarly, the longest structural paths from input A to output C_{out} are not robustly testable either.

Therefore all the 5-gate paths from A or C_{in} to C_{out} are not robustly testable. But the 4-gate paths from A or C_{in} to C_{out} are robustly testable, if the first gate is bypassed. Figure 6 shows an example.

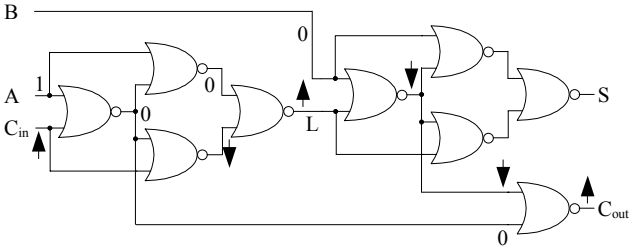


Figure 6. A longest robustly testable path from input C_{in} to output C_{out} in a full adder.

Thus, when a robustly testable path goes from A or C_{in} to C_{out} through a full adder or the half adder in the bottom row, its delay must be at least one gate less than that of a longest structural path. It can be seen from Figure 2 that this case must happen in one full adder and the half adder in the bottom row. Therefore, the delay of a longest robustly testable path is at least two gates less than that of a longest structural path, which contains 124 gates. With similar analysis, the delay of a longest non-robustly testable path is at least one gate less than that of a longest structural path. The reason is, if the path goes through a longest structural path from A (or C_{in}) to C_{out} in a full adder, there must be a glitch at C_{out}, according to Figure 5. This glitch cannot propagate through the longest structural paths from A (or C_{in}) to C_{out} in the half adder in the bottom row. Therefore, the non-robustly testable path must bypass one gate before it reaches the half adder, or bypass one gate in the half adder. Then a glitch can be generated at the output C_{out} of the half adder and propagated to the primary output P_{30}, without bypassing any more gates.

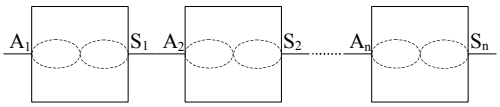


Figure 7. An adder network example.

The reason why traditional path generators fail on c6288 is because all the longest structural paths, which have 124 gates, are not testable. Because of the reconvergence within the adder modules (*local reconvergence*) and in the adder network (*global reconvergence*), the number of longest structural paths is exponential in the number of gates. For example, in an adder network shown in Figure 7, there are n full adders and the output S of the ith adder is connected to the input A of the $(i+1)$th adder $(0<i<n)$. The dotted curves represent the four reconverged longest structural paths between input A and output C_{out} within an adder. In this network, there are 2^{2n} longest structural paths, which have equal length, between the input A_1 and the output S_n. If they are all false paths, to prove so many longest structural paths are untestable consumes too much CPU time for traditional path generators.

The path generator used in [14] is able to find the global longest testable path in c6288. The reason is that it uses a *dynamic dominator* heuristic [16]. This tool grows paths from primary inputs. It can be seen in Figure 8 that when a path reaches line H it must go through line L, and when it reaches line P it must go through line S, assuming the path attempts to extend to the farthest primary output from that point. It is said that line L dynamically dominates line H and line S dynamically dominates line P. Therefore, if the path is blocked somewhere and the conflict is irrelevant to the decision of going through the upper NOR gate or the lower NOR gate, all the paths through the series of dynamic dominators can be proven untestable. Figure 9 shows this case in the multiplier. Then the path generation backtracks to g_3 immediately and extends to a shorter path P_2, instead of trying all the equal length paths from g_1 to g_4.

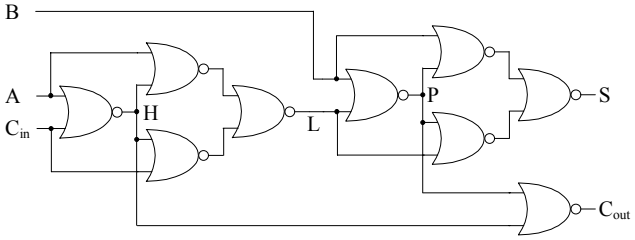


Figure 8. Dynamic dominators in a full adder.

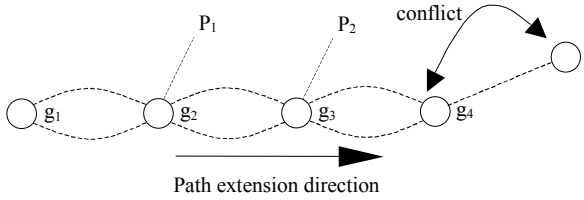


Figure 9. Dominator heuristic.

However, the limitation of this heuristic is that it only solves the problem of exponential number of paths due to the local reconvergence within an adder. It does not solve the problem due to the global reconvergence in the adder network. Fortunately, as shown in Figure 4, there is not much global reconvergence which causes backtracks, before a global longest testable path is generated. This assumes the local reconvergence problem is solved so that there are few backtracks within an adder.

Unfortunately, if we attempt to generate the longest testable paths through some gates/adders in c6288, e.g. the black one in Figure 10, the global reconvergence problem is much worse. The longest structural paths are highlighted in Figure 10. As discussed above, whenever a robustly testable path passes from the input A or C_{in} to the output C_{out} in an adder, it must bypass at least one gate. So the delay of a longest robustly testable path through a particular adder is several gates less than that of a longest structural path through this adder, depending on which

column the adder is located in. For example, a longest robustly testable path through the black adder in Figure 10 should bypass at least 6 gates. This results in more false paths which are longer than the longest testable paths.

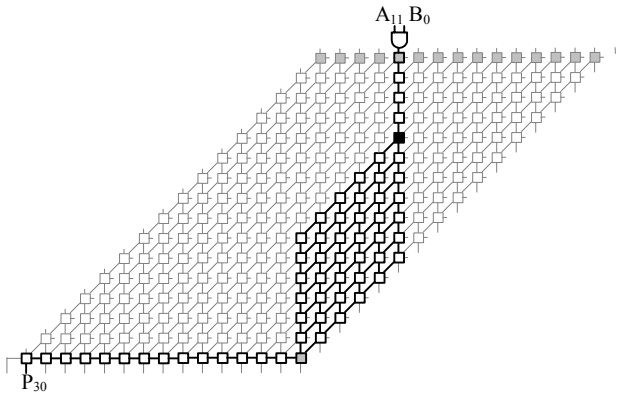


Figure 10. Longest structural paths through a particular adder.

3. Heuristic

The analysis in the previous section shows that local conflicts are the fundamental reason for false paths in c6288, and this is true for most circuits [3]. The reconvergence which does not cause conflicts, either local or global, exacerbates the problem by introducing an exponential number of backtracks for a single local conflict. But this type of reconvergence does not cause the false path problem by itself. Therefore, it is helpful to identify the local conflicts as early as possible to avoid the expensive search in the non-solution space (more reconvergence causes more expensive search in the non-solution space).

The path generator used in [14] includes a preprocessing phase. In this phase the maximum structural distance from each gate to primary outputs is computed, without considering any logic constraint. This value is termed the *PERT delay*. In the path generation phase, *partial paths* are initialized from primary inputs. A partial path is a path which originates from a primary input but has not reached any primary output. A value called *esperance* [3] is associated with a partial path. The esperance is the sum of the length of the partial path and the PERT delay from its last node to a primary output. In other words, the esperance of a partial path is the upper bound of its delay when it reaches a primary output. In each iteration of the path generation phase, the partial path with the highest esperance value is extended by adding one gate. If the last gate of the partial path has more than one fanout, the partial path splits. The esperance value associated with each new partial path is updated. Then the constraints to propagate the transition on the added gate, such as non-controlling side input values, are applied. If there are any conflicts, the whole search space which contains the partial path is trimmed off.

However, a drawback in this path generator is that it only detects the conflict between the constraints at the newly added gate and the gates already in the partial paths. For the remaining search space, it simply uses PERT delay values. Since the PERT delays are computed without considering any logic constraint, it is not able to detect a local conflict in the unexplored search space until the partial path grows to that site. This results in too many backtracks if the case shown in Figure 10 occurs.

The basic idea of our heuristic is to exclude untestable subpaths due to local conflicts when computing the PERT delay for a gate. We call the new values *Smart-PERT* delays, or *S-PERT*. Because some untestable subpaths are not included in the S-PERT computation, a gate's S-PERT delay is always less than or equal to its PERT delay. Moreover, compared to the PERT delay, the S-PERT delay is closer to the delay of the longest testable path from that gate to a primary output.

A gate's PERT delay can be computed using its fanout gates' PERT delays. If the unit delay model is used, PERT(g_i) = max {PERT(g_j) | g_j is a fanout gate of g_i} + 1. Figure 11(a) shows an example, assuming PERT(g_3) = 8 and PERT(g_4) = 6 are known. In this example, PERT(g_0) = 10 is computed using PERT(g_1) and PERT(g_2).

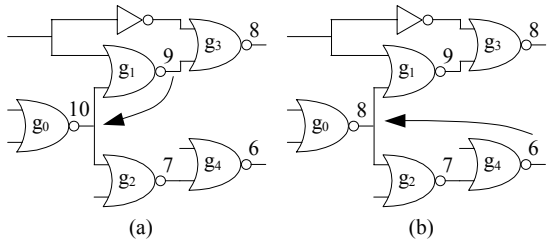


Figure 11. Computation of PERT delay (a) and S-PERT delay (b).

When S-PERT(g_i) is computed, a user-defined variable *S-PERT depth* is used. If the S-PERT depth is set to d, then S-PERT(g_i) is computed using S-PERT(g_j) where g_j is d gates from g_i in g_i's fanout tree. For example, in Figure 11(b), if d is set to 2, then S-PERT(g_0) is computed using S-PERT(g_3) and S-PERT(g_4).

The heuristic works as follows. Suppose S-PERT(g_i) is being computed. G = {g_j | g_j is d gates from g_i in g_i's fanout tree}, and G is sorted by S-PERT(g_j) in decreasing order. The heuristic pops the first gate g_j in G and attempts to propagate a transition from g_i to g_j. If there is no conflict (the transition successfully reaches g_j, with all the constraints applied), S-PERT(g_i) is set to S-PERT(g_j) + d. Otherwise, it pops the second gate in G and repeats the same procedure. In Figure 11(b), for example, at first the heuristic tries to propagate a transition from g_0 to g_3, but finds it is impossible to set the side inputs of g_2 and g_3 both to non-controlling values. Then it tries g_4 and does not meet any conflict. So S-PERT(g_0) is 8. It is obvious that increasing the S-PERT depth can make the S-PERT delays closer to the delay of the longest testable path from that

gate to a primary output, but its cost increases exponentially. Therefore, there must be some trade-off.

Since most conflicts are local, with S-PERT delays, the path generation is well guided to a testable path, with many fewer conflicts, because most of the non-solution space is trimmed off during the preprocessing phase.

4. Experimental Results

A path generation tool using our heuristic has been implemented in Visual C++ and run on Windows 2000 with a 2.2 GHz Pentium 4 processor and 256 MB memory. The unit delay model is used in the experiments.

In our experiments we consider only robustly testable paths, because a robust test requires the on-path signals have different initial and final logic values. Therefore the test is comparable to a transition test [17]. If neither the slow-to-rise nor the slow-to-fall transition fault at a gate/line is detectable, it is not possible to generate a robustly testable path through that gate/line. Therefore the transition fault coverage for c6288 is a good reference to see how many gates/lines have testable paths but our tool is not able to generate the longest through them (aborts).

Because a transition test can be composed by pairing the stuck-at-0 and stuck-at-1 vectors [18], the transition fault coverage achieved by exhaustively pairing all stuck-at vectors gives the percentage of the gates/lines which have at least one robustly testable path. For c6288, the percentage is 99.19% [19]. In other words, 0.81% of the gates have no transition test.

Table II. Results for generating the K longest robustly testable paths through each gate for c6288.

K	# of Paths	Aborts (%)	CPU Time (m:s)
1	726	1.28	30:53
2	1 463	1.28	31:33
3	2 194	1.49	37:33
4	2 942	1.49	38:00
5	3 679	1.49	38:10
6	4 416	1.49	38:16
7	5 127	1.49	38:30
8	5 828	1.49	38:33
9	6 529	1.57	40:26
10	7 224	1.57	40:33

Table II shows the results for generating the K longest paths through each gate for c6288. In the experiments we set the S-PERT depth to 6. Very little benefit was observed for larger S-PERT depths. Column 2 shows the number of generated paths. Column 3 shows the percentage of aborted gates. In the experiments the path generator gives up if the Kth testable path is not found after 50 000 iterations. An iteration contains a set of operations in which a partial path is popped and one more gate is added. The CPU time is shown in column 4. Only a few of gates which have testable paths passing through get aborted, compared to the transition fault coverage. For example, when K=1, the percentage is 0.47% (1.28% – 0.81%). Also it can be seen that most CPU time is spent on finding the first testable path through each gate. It is known that many longest testable paths have equal length. So after the first one is found, it does not cost much to find more. On the other hand, when the abort percentage increases (K=3 and K=9), the CPU time slightly jumps. It indicates that the CPU time spent on the aborts cannot be neglected, even if the abort rate is low.

5. Conclusions and Future Work

The ISCAS85 circuit c6288 has an exponential number of false paths. The path generation for this circuit using existing ATPG tools is very inefficient, compared to other circuits. In this paper we have studied the logic structure of c6288, and found the features which result in numerous false paths. We have proposed a novel heuristic which efficiently generates the K longest paths through each gate for c6288. Compared to the transition fault coverage, only a few of gates get aborted using this heuristic.

In this work we set the S-PERT delay depth to a fixed value. However, if the value is too small, the path generator cannot exclude enough local conflicts, and if it is too large, its cost is too high. Our future work is to make the heuristic automatically learn how large the S-PERT delay depth should be for different gates.

Acknowledgements

This research was funded by the Semiconductor Research Corporation under contract 2000-TJ-844 and the National Science Foundation under contract CCR-1109413.

References

[1] G. L. Smith, “Model for Delay Faults Based Upon Paths,” *IEEE Int’l Test Conf.*, Philadelphia, PA, Oct. 1985, pp. 342-349.

[2] C. J. Lin and S. M. Reddy, “On Delay Fault Testing in Logic Circuits,” *IEEE Trans. on Computer-Aided Design*, vol. 6, no. 9, Sept. 1987, pp. 694-701.

[3] J. Benkoski, E. V. Meersch, L. J. M. Claesen and H. D. Man, “Timing Verification Using Statically Sensitizable Paths,” *IEEE Trans. on Computer-Aided Design*, vol. 9, no. 10, Oct. 1990, pp. 1073-1084.

[4] P. McGeer and R. K. Brayton, “Efficient Algorithms for Computing the Longest Viable Path in a Combinational Network,” *ACM/IEEE Design Automation Conf.*, Las Vegas, NV, June 1989, pp. 561-567.

[5] H. Chang and J. A. Abraham, “VIPER: An Efficient Vigorously Sensitizable Path Extractor,” *ACM/IEEE Design Automation Conf.*, Dallas, TX, June 1993, pp. 112-117.

[6] J. J. Liou. A. Krstic, Li-C. Wang and K. T. Cheng, “False-Path-Aware Statistical Timing Analysis and Efficient Path Selection for Delay Testing and Timing Validation,” *ACM/IEEE Design Automation Conf.*, New Orleans, LA, June 2002, pp. 566-569.

[7] W. N. Li, S. M. Reddy and S. K. Sahni, “On Path Selection in Combinational Logic Circuits,” *IEEE Trans. on Computer Aided Design*, vol. 8, no. 1, Jan. 1989, pp. 56-63.

[8] A. K. Majhi, V. D. Agrawal, J. Jacob and L. M. Patnaik, "Line Coverage of Path Delay Faults," *IEEE Trans. on VLSI Systems*, vol. 8, no. 5, Oct. 2000, pp. 610-613.

[9] A. Murakami, S. Kajihara, T. Sasao, R. Pomeranz and S. M. Reddy, "Selection of Potentially Testable Path Delay Faults for Test Generation," *IEEE Int'l Test Conf.,* Atlantic City, NJ, Oct. 2000, pp. 376-384.

[10] Y. Shao, S. M. Reddy, I. Pomeranz and S. Kajihara, "On Selecting Testable Paths in Scan Designs," *IEEE European Test Workshop*, Corfu, Greece, May 2002, pp. 53-58.

[11] I. Pomeranz, S. M. Reddy and P. Uppaluri, "NEST: A Nonenumerative Test Generation Method for Path Delay Faults in Combinational Circuits," *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems*, vol. 14, no. 12, Dec. 1995, pp. 1505-1515.

[12] K. Fuchs, F. Fink and M. H. Schulz, "DYNAMITE: An Efficient Automatic Test Pattern Generation System for Path Delay Faults," *IEEE Trans. on Computer-Aided Design*, vol. 10, no. 10, Oct. 1991, pp. 1323-1355.

[13] K. Fuchs, M. Pabst and T. Rossel, "RESIST: A Recursive Test Pattern Generation Algorithm for Path Delay Faults Considering Various Test Classes," *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems*, vol. 13, no. 12, Dec. 1994, pp. 1550-1562.

[14] J. A. Bell, "Timing Analysis of Logic-Level Digital Circuits Using Uncertainty Intervals," *M. S. Thesis*, Department of Computer Science, Texas A&M University, 1996.

[15] M. Hansen, H. Yalcin and J. P. Hayes, "Unveiling the ISCAS-85 Benchmarks: A Case Study in Reverse Engineering," *IEEE Trans. on Design and Test*, vol. 16, no. 3, July-Sept. 1999, pp. 72-80.

[16] T. Kirkland and M. R. Mercer, "A Topological Search Algorithm for ATPG," *ACM/IEEE Design Automation Conf.*, June 1987, Miami, FL, pp. 502-508.

[17] Z. Barzilai and B. K. Rosen, "Comparison of AC Self-Testing Procedures," *IEEE Int'l Test Conf.*, Philadelphia, PA, Oct. 1983, pp. 89-94.

[18] J. Waicukauski, E. Lindbloom, B. K. Rosen and V. S. Iyengar, "Transition Fault Simulation," *IEEE Design & Test of Computers*, vol. 4, no. 5, April 1987, pp. 32-38.

[19] X. Liu, M. S. Hsiao, S. Chakravarty and P. J. Thadikaran, "Novel ATPG Algorithms for Transition Faults," *IEEE European Test Workshop*, Corfu, Greece, May 2002, pp. 47-52.