An Efficient Algorithm for Finding the K Longest Testable Paths Through Each Gate in a Combinational Circuit

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

Testing the K longest paths through each gate (KLPG) in a circuit detects the smallest local delay faults under process variation. In this work a novel automatic test pattern generation (ATPG) methodology to find the K longest testable paths through each gate in a combinational circuit is presented. Many techniques are used to significantly reduce the search space. The results on the ISCAS benchmark circuits show that this methodology is very efficient and able to handle circuits with an exponential number of paths, such as c6288.

1. Introduction

Delay testing detects small manufacturing defects which do not cause functional failure but affect the speed of integrated circuits. The path delay fault model [1] is the most conservative of any of the classical models for delay faults because a circuit is considered faulty if the delay of any of its paths exceeds the specification time. The main problem with this model is the large number of paths in real circuits. To overcome this problem, some test methods only cover a subset of paths, e.g. the global longest paths in a circuit [2][3], or the longest path through each gate [4][5][6][7][8]. A delay fault caused by a local defect, such as a resistive open or short, can only be detected by testing a path through it, and testing the longest path through it can detect the smallest *local delay fault*. The quality of a test set is defined as how close the minimum actually detected delay fault sizes are to the minimum possibly detectable fault sizes [9], according to the gate delay fault model [10].

The problem of finding the longest path through each gate or line in a circuit has been extensively studied [4][5][6][7][8]. However, some delay defects are distributed on a path, such as variation in transistor channel length across a chip. The faults caused by this kind of defects are termed *global delay faults* [11]. Timing and power optimization tends to compress the distribution of path delays in a circuit, so many paths are close to the maximum delay [12]. Moreover, because process variation occurs everywhere in an integrated circuit, even for a single gate, it is hard to determine which path is the *actual longest path* passing through it. Therefore, testing only one path through each gate cannot guarantee the detection of the smallest local delay faults. Testing the K longest paths through a fault site increases the fault detection probability under process variation, because it increases the probability that the actual longest path is tested. 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][13][14][15]. If a path is not testable, it is called an untestable or false path [16]. For example, in Figure 1, path *a-c-d* is a false path under the single-path sensitization criterion [13], 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.

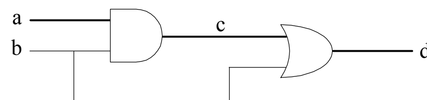

Figure 1. A circuit with a false path *a-c-d*.

Earlier research [4][5][6][7] on generating the longest path through each gate is either inefficient or fails to guarantee the testability of the generated paths. The inefficiency comes from the fact that most prior work lists many long structural paths first, then checks their testability. If there are subcircuits with a large number of paths passing through them, all the long paths listed are similar and it is possible that none of them is testable.

Many ATPGs for the global longest path generation were studied to see if they could be extended to solve the problem of finding the K longest testable paths through each gate (KLPG). A fast ATPG tool NEST [17] 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. DYNAMITE [18] is very efficient in poorly testable circuits, but in highly testable circuits many faults are treated separately, which results in huge memory

 0-7803-8106-8/03 $17.00 Copyright 2003 IEEE

consumption and so is not practical for large circuits. RESIST [19] 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 the path delay faults as either testable or untestable in circuit c6288, which is known for having an exponential number of paths. However, the test generation for c6288 is still slow. RESIST took 1 122 CPU hours to find 12 592 paths on a SPARC IPX 28 MIPS machine. Recent research [8] presented an efficient method to extend the RESIST algorithm to the problem of finding a set of longest testable paths that cover every gate. This method takes advantage of the relations between the longest paths through different gates, and guarantees their testability. However, partly because this work assumes a unit delay model, there is no obvious way to extend it to handle the problem of finding the K longest testable paths through each gate, and this method fails when applied to c6288, indicating that the advantages in the RESIST algorithm may not be easily applied to generating paths through a certain gate.

A timing analysis tool [3] presents another method to efficiently identify the global longest testable paths in a combinational circuit. Instead of generating many long structural paths and checking their testability, this tool grows paths from the primary inputs. In each iteration a new gate is added and the constraints are applied to that gate. Then instead of assigning logic values on one or more primary inputs to satisfy the constraints on the newly added gate, as done in VIPER [15], direct implications, which are more efficient, are applied to find local conflicts. If conflicts exist, the whole search space which contains the already-grown series of gates is trimmed off. This technique is called implicit false path elimination [13][20]. Some other false path elimination techniques, such as forward trimming and dynamic dominators, are also applied in this tool to identify false paths earlier. This tool is efficient and able to handle c6288.

In this paper, we present an algorithm which extends this method to generate the KLPG test set for a combinational circuit. It inherits the framework of [3] but aims at particular gates one by one. This algorithm also takes advantage of the relations between the long paths through different gates, which are revealed in previous research [8] and extended in this work, to reduce the search space and avoid repeated work. Analysis shows that these relations are very easy to apply in this framework and their usefulness is observed in the experiments. In this work we also determined that initially applying the global longest path generation could cover some gates very quickly. When the efficiency of more global paths declines, the path generation aiming at individual gates is performed. Experimental results show that this 2-phase path generation saves 5-10% in execution time.

The remainder of the paper is organized as follows: Section 2 describes our method to generate the K longest testable paths through a particular gate. Section 3 describes some false path elimination methods which can be applied in the path generation to reduce the search space and avoid repeated work. In Section 4 experimental results are shown and analyzed. We performed experiments on the ISCAS85 and the full scan versions of the largest ISCAS89 benchmark circuits under the robust [1][2] and non-robust [1] sensitization criteria. Instead of using the unit delay model, we extracted buffer-to-buffer nominal delays from circuit layouts [21], which makes the experiments more realistic. Section 5 concludes with directions for future research.

2. Path Generation

2.1. Preprocessing

Before the path generation, some topology information is collected to help guide the path generation process and trim the search space. First, the min-max delay from each gate to any primary output is computed without considering any logic constraint (PERT delay). A gate's min-max PERT delay can be simply computed using its fanout gates' min-max PERT delays and the rising/falling buffer-to-buffer delays between gates.

In addition to the PERT delays, the earliest and latest possible rising/falling transition times on the input and output lines for each gate are computed, assuming that a transition at any primary input can only occur at time zero. This procedure is similar to the PERT delay computation, with complexity linear in the circuit size. This information is useful under some sensitization criteria [14][15] because transitions can occur only within the earliest/latest range. When propagating a transition which transits to a controlling value through a gate, if a side input cannot have a transition before the on-path transition happens, the final value on that side input can be either a controlling or non-controlling value (the initial value must still be non-controlling), without blocking the on-path transition. Without this information the search procedure may unnecessarily require the final value on the side input to be non-controlling.

2.2. Path Store

To find the K longest testable paths through gate g_i, a path store is established for the path generation. In the path store, many *partial paths*, which may become the K longest testable paths through gate g_i, are stored. A partial path is a path which originates from a primary input but has not reached a primary output. Figure 2 shows an example. The partial path starts from primary input g_0, and ends at gate g_i. At the beginning, the path store contains $2n_{\mathrm{PI}}$ partial paths, where n_{PI} is the number of primary inputs. There are 2 partial paths from each primary input, representing a rising or falling transition at that primary input. Each partial path has only one node (a primary

input) initially. A partial path grows when one more gate is added to it. When a partial path reaches a primary output, it becomes a *complete path*.

A value called *esperance* [13] is associated with a partial path. The min-max esperance is the sum of the length of the partial path and the min-max PERT delay from its last node to a primary output. In other words, the max esperance of a partial path is the upper bound of its delay when it grows to a complete path, and the min esperance is the lower bound if the complete path is testable. In Figure 2, suppose the length of the partial path $g_0 \ldots g_i$ is 5, and the PERT delays between g_i and primary outputs g_r, g_s, g_t are 10, 8, 6, respectively. The min-max esperance of partial path $g_0 \ldots g_i$ is 11/15.

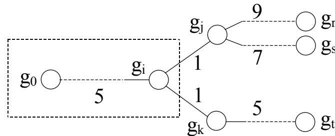

Figure 2. A partial path and its esperance.

The partial paths are sorted by max esperance. Every time the path generator selects the partial path with the largest max esperance. Potentially this partial path will grow to a complete path with maximum delay.

2.3. Path Generation

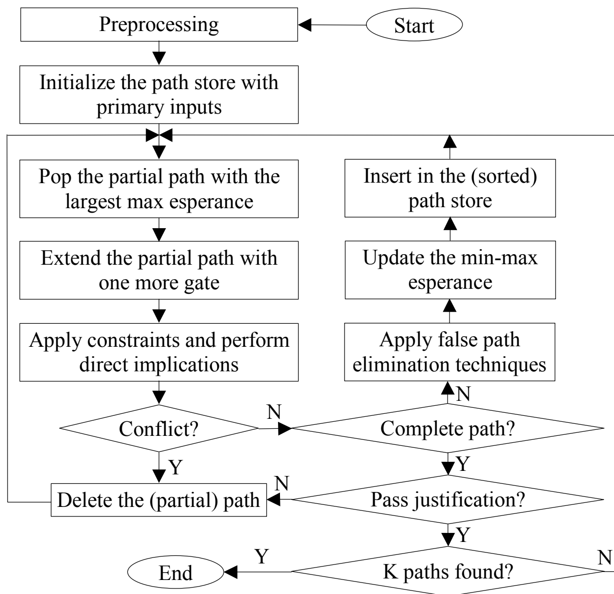

Figure 3. Path generation algorithm.

Figure 3 is the algorithm of finding the K longest testable paths through gate g_i. Before the path generation for gate g_i, all gates which are not in g_i's fanin or fanout cone are identified because when a partial path grows, it is impossible for any of these gates to be added (otherwise the partial path has no chance to pass through g_i). But these gates are still useful because they are related to constraints, such as side input constraints of a gate on the path which is being searched. Figure 4 shows the search space.

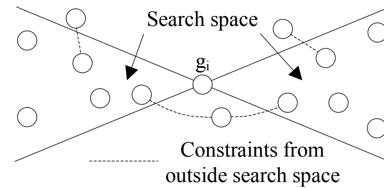

Figure 4. The search space for a path through gate g_i.

Each iteration of the path generation begins by popping the first partial path from the path store, which has the largest max esperance. The partial path is extended by adding a fanout gate which contributes to the largest max esperance. For example, in Figure 5, the partial path $g_0 \ldots g_i$ is extended by adding gate g_j because extending to g_j could potentially keep the max esperance. If the partial path has more than one extendable fanout, it must be saved in another copy and in the copy the already tried fanout must be marked "blocked" or "tried". Then the copy gets its esperance updated and is pushed into the path store. For example (Figure 5), since gate g_i has 2 fanouts, and extending the partial path to g_j may result in false paths later, the partial path $g_0 \ldots g_i$ must be saved because extending it to gate g_k may get a longer testable path. And because fanout g_j has been tried, in the copy the min-max esperance becomes 11/11.

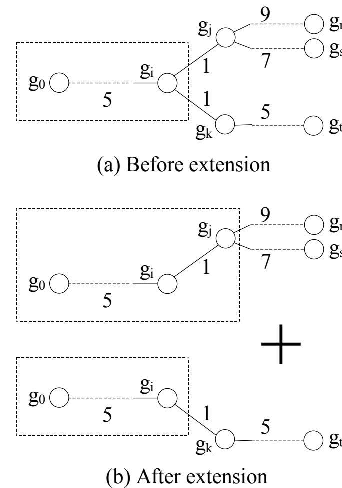

Figure 5. Extending a partial path.

After the partial path is extended ($g_0 \ldots g_i g_j$ in Figure 5) the constraints to propagate the transition on the added gate (g_j) are applied. Under the non-robust sensitization criterion [1], non-controlling final values on the side inputs are required. Under the robust sensitization criterion [1][2], in addition to non-controlling final values, the side inputs must remain non-controlling if the on-path input has a transition to the controlling value. Then direct implications are used to propagate the constraints throughout the circuit. A direct implication on a gate is one where an input or

output of that gate can be directly determined from the other values assigned to that gate. Figure 6 shows some examples of direct implications on an AND gate. The values in boxes are implied from the existing values, which are not in boxes. Figure 6(a) is an example of forward implication, and (b)(c) are examples of backward implications. If a conflict happens during direct implications, the partial path is false. In other words, any path including this partial path is a false path. For example (Figure 5), if extending partial path $g_0 \ldots g_i$ to gate g_j results in a conflict (Figure 7 shows an example), both path $g_0 \ldots g_r$ and $g_0 \ldots g_s$ are determined to be false. Therefore, the partial path is deleted from the path store so that the whole search space which contains this partial path is trimmed off. Previous research [13] showed that most false paths can be eliminated by direct implications, and this is also observed in our experiments.

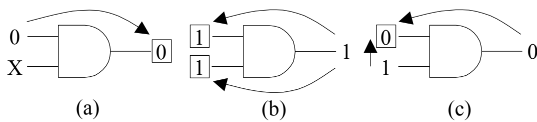

Figure 6. Examples of direct implications.

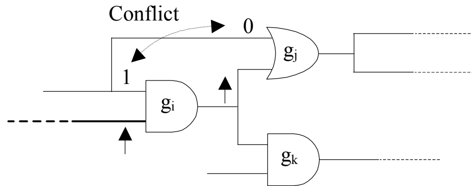

Figure 7. Conflict after applying direct implications.

If the extended partial path reaches a primary output, it becomes a complete path. In this case, a final justification process (a FAN [22] style decision tree based justification algorithm) is performed on the path. One reason to do final justification is to find a vector pair which sensitizes this path. The other reason is that some false paths do not have any conflict in the direct implications during the growth procedure. Figure 8 shows an example [13]. Suppose both AND gates need their side inputs to be logic 1, and direct implications stop at the two OR gates because neither input must be logic 0 or 1, assuming the algorithm does not know the two inputs are tied together. This path does not fail the direct implications but it is a false path. Since most false paths cannot pass the direct implications, it is not often that a complete path fails the final justification.

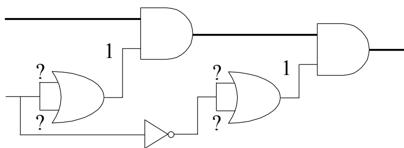

Figure 8. A path which passes direct implications but fails final justification.

If the extended partial path is not a complete path, some false path elimination techniques, which will be discussed in detail in the next section, are applied to it, to more efficiently prevent the new partial path from becoming a false path. Then the min-max esperance of the partial path is updated and it is inserted into the path store. Since its max esperance may decrease and min esperance may increase after extension, it may not be inserted at the top of the path store. If this happens, in the next iteration another partial path will be selected for extension. For example (Figure 5), after extending partial path $g_0 \ldots g_i$ to gate g_j, the min-max esperance changes from 11/15 to 13/15. If path $g_j \ldots g_r$ is blocked, which means path $g_0 \ldots g_r$ is a false path, after applying the false path elimination techniques, the min-max esperance is 13/13.

Because each partial path consumes memory, and the path store cannot have an infinite size, when the number of partial paths exceeds the path store size limit, some partial paths with low max esperance are removed from the path store. The maximum esperance of the removed partial paths is recorded. In the future any partial path with max esperance below that value must be removed from the path store because it may not truly be one of the K longest testable paths. Therefore it may happen that K paths have not been found when the path store is empty. However, since a partial path is represented as a sequence of gates, usually it consumes less than 1 KB memory. Thus the path store can have a large number of partial paths so that in most cases the algorithm does not abort unless none of the structural paths through the gate is testable.

The path generation iteration does not stop until the K longest testable paths through gate g_i are found or the path store is empty. Since the K longest testable paths through different gates may overlap, every time a new path is generated, it must be checked to see if it has already been generated during the path generation for another gate.

3. Refined Implicit False Path Elimination

3.1. Forward Trimming

When a partial path grows, in some cases after the constraints are applied and the direct implications are performed, the possibility to continue extending the partial path to the primary outputs may be reduced. In the extreme case there is no way to continue extending the partial path (completely blocked).

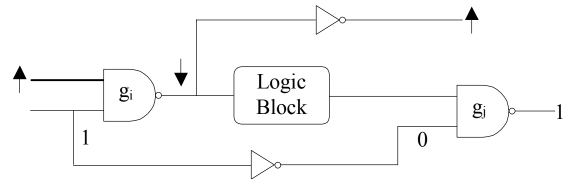

Figure 9. Application of forward trimming.

Figure 9 shows an example where the path through the logic block cannot propagate to the output [3]. In this example the partial path has grown to the NAND gate g_i, and the side input must be a logic 1 if the single-path sensitization constraints [13] are considered. This value

propagates forward through the inverter and becomes a controlling value on one of the inputs of the NAND gate g_j. This condition prevents propagation from g_i to g_j through any paths within the logic block. With *forward trimming* the entire logic block is trimmed off and the search process is guided toward an unblocked path (the upper inverter) earlier. Without forward trimming the search process might be much less efficient. Since the paths through the logic block are not blocked until the final gate g_j, the search process may attempt to traverse through the logic block until it reaches g_j and learn that the path is blocked. This may be done for each possible path through the logic block.

During the circuit preprocessing the PERT delays are computed assuming no path is blocked. With the growth of a partial path, more and more information is known because more constraints are applied. Forward trimming recomputes the min-max PERT delay from the end of the partial path, based on the structure and the current value assignments. In this case blocked extension choices are not considered in the computation and false paths can be eliminated earlier so that the search can be guided more accurately toward a testable path. If the search space is partially trimmed off, the partial path still has a chance to become a testable complete path, but its esperance may be reduced according to its PERT delay reduction. In the next iteration of path generation, a more promising partial path may be selected.

3.2. Smart-PERT Delay

If the PERT delays are used, a local conflict in the unexplored search space is not detected until the partial path grows to that site, because the PERT delays are computed without considering any logic constraint.

We have developed a heuristic 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*. For simplicity only maximum PERT and S-PERT delays are discussed. 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 10(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).

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 10(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 10(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_1 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.

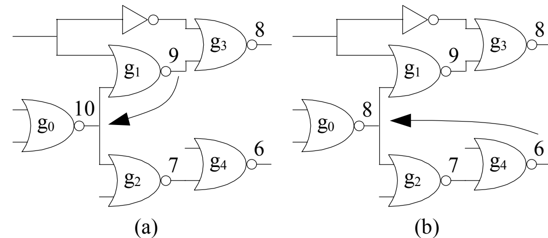

Figure 10. Computation of PERT delay (a) and S-PERT delay (b).

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.

The usefulness of this technique is highly dependent on the structure of the circuit. The most benefit would be derived from a path with d gates, each of which has fanout f and each fanout reconverges at a later gate (Figure 11). This results in f^d possible paths which must be traversed. The worst case is that all of them are false paths but conflicts do not occur until the partial path grows very long. With the use of S-PERT delays, the path generation extends to a shorter structural path P_2 (Figure 11) because it has larger esperance, and the traversal of all the false paths with equal length from g_1 to g_4 is avoided. As our experimental results show, this technique helps exclude many false paths in circuit c6288.

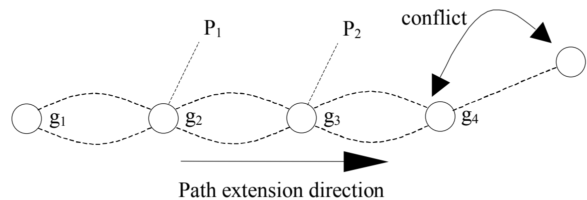

Figure 11. A circuit with exponential number of false paths.

3.3. Relations between Gates

There are tight relations between long testable paths through different gates because a long testable path through a gate is possibly a long testable path through another gate. Some rules in previous research [8] are extended in this work and they are very easy to apply to the path generation algorithm.

Each gate in the circuit has two arrays: $L_{ub}[1\ldots K]$ and $L_{lb}[1\ldots K]$, which indicate the upper and lower bound of the lengths of the K longest testable paths through this gate. The two arrays are sorted. Initially the values in $L_{ub}[1\ldots K]$ are all set to the length of the longest structural path through the gate, and the values in $L_{lb}[1\ldots K]$ are all 0.

When the K longest testable paths are found for gate g_i, $L_{ub}[1\ldots K]$ and $L_{lb}[1\ldots K]$ for g_i are updated and both of them are set to the actual lengths of the K longest testable paths. Suppose a newly found path for gate g_i also contains gate g_j, which means this path passes through both g_i and g_j. If the length of this path is greater than that of a previously found path for g_j, $L_{lb}[1\ldots K]$ for g_j is updated by inserting a link to the newly found path and deleting the link to the shortest path found. This process may increase the values in $L_{lb}[1\ldots K]$ for all the gates contained in the newly found path. Figure 12 shows an example. Assuming K=3, at some point $L_{lb}[1\ldots 3]$ for gate g_j is {22,18,15}, which means the lengths of the 3 longest paths through g_j found by the path generator are at least 22, 18 and 15. Suppose the length of a newly found path for gate g_i is 20. Then $L_{lb}[1\ldots 3]$ for gate g_j is updated to {22,20,18}.

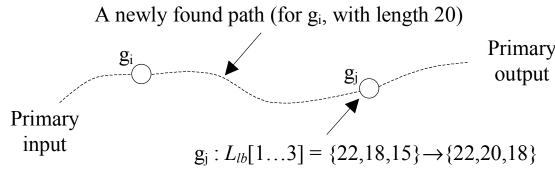

Figure 12. Updating $L_{lb}[1\ldots K]$.

On the other hand, the values in $L_{ub}[1\ldots K]$ for some gates decrease when a new path is found. Suppose gate g_i has f fanin gates, and $\cup_{\text{fanin}}L_{ub}[1\ldots K]$ indicates the union of the $L_{ub}[1\ldots K]$ arrays of its fanin gates and it is sorted in decreasing order. The upper bound of the lengths of the K longest testable paths through gate g_i cannot exceed the first K values in $\cup_{\text{fanin}}L_{ub}[1\ldots K]$, because all the paths through g_i must also pass through one of its fanin gates. Figure 13 shows an example. Assuming K=3, and gate g_i has two fanin gates, with $L_{ub}[1\ldots 3]$={17,16,11} and {20,18,12}. Then the values in $L_{ub}[1\ldots 3]$ for g_i must be no more than {20,18,17}. The same analysis can be performed using the fanout gates or absolute dominators [3] of gate g_i.

As more paths are found, the values in $L_{ub}[1\ldots K]$ and $L_{lb}[1\ldots K]$ for gate g_i, for which the path generation has not been performed, get closer. If they are close enough, say the difference is less than 1%, it can be assumed that the K longest testable paths for gate g_i have been found so that the path generation for it can be skipped. Many gates can be skipped if a unit delay model is used [8].

If gate g_i cannot avoid path generation, during its path generation process, if the max esperance of the partial path being processed is less than its $L_{lb}[v]$ ($1<v\leq K$) value, then the first v paths already found for g_i are proved to be the v longest paths through g_i, because the partial paths in the path store have no chance to grow to a complete path with larger length. So $L_{ub}[u]$ (u=1,…,v) are updated accordingly (set to $L_{lb}[u]$). On the other hand, when the vth longest path through gate g_i is being searched, and the min esperance of a partial path is greater than $L_{ub}[v]$, the partial path can be deleted immediately because when it becomes a complete path, this path is either a false path or already found.

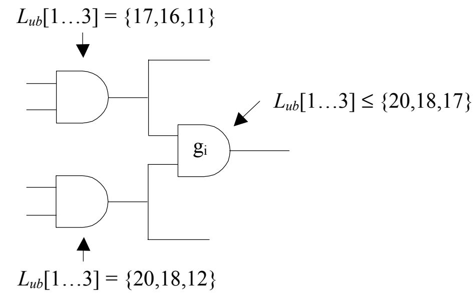

Figure 13. Updating $L_{ub}[1\ldots K]$.

The $L_{ub}[1]$ values of other gates can also be taken advantage of during the path generation process for gate g_i. Suppose a partial path grows to gate g_j. If the max esperance of the partial path is greater than the $L_{ub}[1]$ value for g_j, it must be reduced to $L_{ub}[1]$ for g_j, because it is impossible for this partial path to grow to a complete path with length greater than $L_{ub}[1]$ for g_j, otherwise this path would also be a testable path through g_j, which invalidates $L_{ub}[1]$ for g_j. When the partial path continues to grow, its min esperance may increase, and if it becomes larger than its max esperance, the partial path is deleted because it must eventually grow to a false path.

3.4. Global Longest Path Generation

The *global longest paths* are the longest paths throughout the circuit, regardless of which gates they pass through. The global longest path generation algorithm is a slight modification to the path generation algorithm for a particular gate (Figure 3). If no gate is eliminated from being added to a partial path (gates which are not in gate g_i's fanin and fanout cones are eliminated for the path generation for g_i), the complete paths generated from the path generation are the global longest paths.

The advantage of finding the global longest paths is: If there are p global longest paths covering gate g_i, these paths must be the p longest paths through g_i. Therefore, at the beginning, the global longest path generation can cover many gates. For comparison, if no global longest path generation were performed, gate g_i would also get some potential longest path through it during the path generation

process for other gates, but in most cases they would not be "verified" until the path generation for g_i is performed.

However, as the global longest path generation finds more paths, the possibility that more gates get covered falls. The worst case is that almost all the global long paths only pass through a very small subset of the gates. Therefore, at the beginning the global longest path generation is useful but after a certain number of paths it is necessary to apply the path generation aiming at individual gates.

In this work a 2-phase strategy is used: Run the global longest path generation until no more gates benefit. Then run the path generation for individual gates which are not fully covered during the global path generation.

The benefits from the global longest path generation are not only that it drops some gates from the individual path generation, it also speeds up the individual path generation for the gates which are not dropped. Suppose the length of the last generated global longest path is L. During the individual path generation all the partial paths with min esperance greater than L can be removed because all the testable paths whose length is greater than L have already been generated. This technique is especially useful when a circuit contains many long false paths.

4. Experimental Results

A path generation tool has been implemented in Visual C++ and run on Windows 2000 with a 2.2 GHz Pentium 4 processor and 256 MB memory. Buffer-to-buffer nominal delays are extracted from circuit layouts [21] and used in the experiments.

In our experiments, K=1 means the path generation tries to find the longest path (one path) with either rising or falling transition on the target gate output. It does not cover both slow-to-rise and slow-to-fall transition faults for the target gate output. Recent research [23] shows that most delay faults are due to resistive opens that affect both transitions. Therefore a resistive open fault can be detected by testing either transition.

Table 1 shows the results for generating the five longest paths through each gate (K=5) for the ISCAS85 and the full scan versions of the largest ISCAS89 benchmark circuits, under the robust and non-robust sensitization criteria. Under each criterion, the number of testable paths generated (not including the paths which fail final justification) and the execution time are listed. The size of the path store was set to 3 000. Experiments showed that for most circuits, the K longest testable paths for more than 90% of the gates can be found using a path store with this size even when K=50. In the experiments a gate is aborted if the path store is empty or the number of iterations exceeds 50 000 (in each iteration a gate is added to a partial path). In the experiments the number of backtracks in the final justification process is set to 100, because it is observed in our experiments that more than 90% of the generated paths can be justified within this number of backtracks and a higher backtrack limit provides little benefit. The number of gates in each circuit is given in the second column in Table 1. Clearly the upper bound of the total number of generated paths is K times the number of gates in the circuit. It can be seen that the actual number is much smaller than the upper bound, indicating that many gates share the same paths. For all the circuits except c6288, the five longest testable paths through each gate were found or it was proved that the number of testable paths through a gate is less than K, e.g. there is no transition fault test for the gate, or the number of structural paths through the gate is less than K/2. For circuit c6288, our algorithm aborts on 20 gates (out of 2 416 gates) which have transition tests [24]. There will be more discussion on c6288 at the end of this section.

Table 1. Path generation results for generating the five longest paths (K=5) through each gate.

Circuit	# Gates	Robust		Non-Robust	
		# Testable Paths	CPU Time (m:s)	# Testable Paths	CPU Time (m:s)
c432	160	235	0:01	252	0:01
c499	202	423	0:01	430	0:01
c880	383	737	0:02	737	0:02
c1355	546	828	0:04	835	0:06
c1908	880	1 287	0:25	1 269	0:33
c2670	1 269	2 124	0:13	2 251	0:19
c3540	1 669	2 764	0:58	2 796	0:56
c5315	2 307	3 921	0:26	4 101	0:23
c6288	2 416	3 679	38:10	3 458	32:29
c7552	3 513	5 462	0:59	5 815	1:16
s9234	5 597	5 002	1:17	5 283	1:31
s13207	7 951	7 643	1:52	7 711	2:10
s15850	9 772	8 957	2:39	9 377	2:31
s38417	22 179	25 175	6:43	29 446	8:09
s38584	19 253	28 435	10:49	31 095	11:23

Table 2 shows the execution time if some of the implicit false path elimination techniques are not used, assuming K=5 and the robust sensitization criterion is used. The results assuming the forward trimming (FT) or smart-PERT (SP) technique is not used are listed in columns 2 and 3. Column 4 lists the execution time if neither the relations between gates nor the global longest path generation (R&G) is used. Column 5, using all the techniques, is copied from Table 1 for comparison. It can be seen that the forward trimming technique significantly reduces the execution time; the smart-PERT technique is not very useful for most circuits, and using it sometimes results in slightly longer execution time, but it significantly helps c2670 and c6288. The reason is because in most cases, the local conflicts do not cause an exponential number of false paths and these local conflicts can be efficiently removed by forward trimming. Moreover, for most circuits the cost of using the smart-PERT technique is greater than the benefits, while for circuits with many reconvergences and long untestable paths, such as c2670

and c6288, applying this technique results in huge benefits. Without it, c6288 cannot even finish the path generation in a reasonable time (12 hours).

If neither the relations between gates technique nor the global longest path generation (R&G) is used, the path generation slows down significantly for some circuits. But if only one of the techniques is not used, only a 5-10% slowdown is observed. This phenomenon indicates that the benefits from the two techniques greatly overlap for most circuits. If we look at c2670 and c6288, it can be found that most global long structural paths are untestable. If the global longest path generation is applied, these untestable paths are recognized and during the path generation for a particular gate, none of these paths need to be recognized again. Similarly, if the relations between gates are applied, these long untestable paths are also recognized only once. For a short untestable path *p*, the effects are limited because this path may be a long path through gate g_i, assuming all the paths through g_i are short paths, but it is not likely to be a long path through another gate g_j. Thus, during the path generation for g_j, it is very possible that the K longest testable paths through it have already been found before considering path *p*. Therefore, the more untestable global long paths, the more benefits from applying the two techniques.

Table 2. Execution time comparison, assuming some of the implicit false path elimination techniques are not used (K=5, robust).

Circuit	CPU Time (m:s)			
	No FT	No SP	No R&G	Table 1
c432	0:01	0:01	0:02	0:01
c499	0:02	0:01	0:01	0:01
c880	0:04	0:02	0:03	0:02
c1355	0:08	0:04	0:08	0:04
c1908	2:25	0:23	0:36	0:25
c2670	0:21	1:07	0:43	0:13
c3540	3:49	0:57	1:45	0:58
c5315	0:37	0:26	0:46	0:26
c6288	58:13	×	62:38	38:10
c7552	1:21	0:55	1:42	0:59
s9234	1:48	1:11	5:04	1:17
s13207	2:24	2:49	4:13	1:52
s15850	3:12	2:33	6:25	2:39
s38417	8:20	6:35	15:26	6:43
s38584	14:31	10:42	20:09	10:49

"×" means the path generation did not finish within 12 hours.

Another interesting phenomenon is that the forward trimming technique is more efficient for the ISCAS85 circuits while the relations between gates technique and the global longest path generation (R&G) are more efficient for the ISCAS89 circuits. The reason is because in most ISCAS89 circuits more than half of the gates are inverters and buffers. Since these gates have only one input, no path gets blocked if a logic value is assigned to their inputs. Thus the efficiency of the forward trimming technique decreases. However, the efficiency of the R&G increases because the inverters and buffers have only one fanin gate. If the K longest testable paths have been found for the only fanin gate and all these paths go through the inverter or buffer, which is very likely, then the path generation for this inverter or buffer can be skipped.

From the analysis it can be concluded that the forward trimming and smart-PERT techniques reduce the search space and guide the path generation more accurately to generate a testable path, while the relations between gates and the global longest path generation mainly help avoid repeated work.

Table 3 shows the execution time when K is increased. It can be seen that the execution time is approximately linear in K, and the execution time increases much more slowly than K. For c7552, for example, there are 1 196 paths generated when K=1 and 19 888 paths when K=20. The number of generated paths when K=20 is more than 16 times that of K=1, but the execution time increases less than 87%. The reason is that when the first longest path through a gate is generated, in the path store there are many partial paths almost reaching a primary output. Therefore the cost of generating more paths is small. Large K values, such as 500, have been tried and it is observed that the execution time is still linear in K. However, in practice K would not be that large because the test set would be too large.

Table 3. Execution time for different K values (robust).

Circuit	CPU Time (m:s)			
	K=1	K=5	K=10	K=20
c432	0:01	0:01	0:01	0:02
c499	0:01	0:01	0:01	0:01
c880	0:02	0:02	0:03	0:03
c1355	0:04	0:04	0:05	0:06
c1908	0:22	0:25	0:26	0:28
c2670	0:12	0:13	0:14	0:16
c3540	0:46	0:58	1:06	1:28
c5315	0:22	0:26	0:29	0:36
c6288	30:53	38:10	40:33	45:04
c7552	0:45	0:59	1:06	1:24
s9234	1:07	1:17	1:29	1:45
s13207	1:30	1:52	2:21	3:19
s15850	2:06	2:39	3:17	4:15
s38417	5:30	6:43	7:48	9:47
s38584	7:09	10:49	13:16	16:44

The ISCAS85 circuit c6288 is a special case because it contains an exponential number of false paths. Figure 14 [25] shows the structure of circuit c6288, which is a 16×16 bit multiplier. The circuit contains 240 adders, among which 16 are half adders, which are shaded. $P_{31}...P_0$ 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 15 [25] shows the structure of a full adder in c6288 and Figure 16 [25] shows the block diagram. The 15 top-row half adders in Figure 14 lack the C_{in} input. Each of

them has two inverters at locations *V* in Figure 15. The single half adder in the bottom row lacks the *B* input, and it has two inverters at locations *W*.

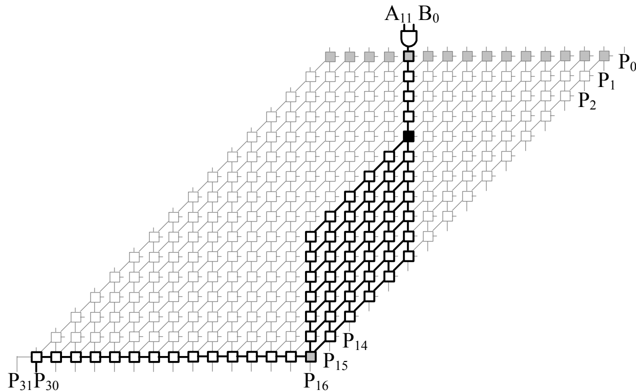

Figure 14. ISCAS85 circuit c6288 16×16 multiplier.

The longest structural paths through a particular gate or adder in c6288, e.g. the black one in Figure 14, are highlighted. All these paths have equal length and include the longest structural paths within the adders. However, the longest structural paths from the input *A* or C_{in} to the output C_{out} in the adders are not robustly testable [26]. This is the main feature which causes most false paths in this circuit. In our experiments the S-PERT depth is set to 6 and this type of local conflict is identified in the preprocessing phase. However, this value is set manually after looking into the circuit structure of c6288. We are developing a heuristic which is able to automatically and dynamically set the S-PERT depth.

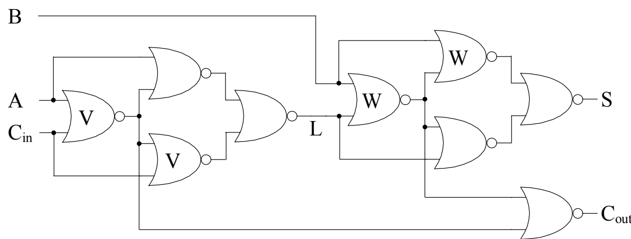

Figure 15. Full adder module in c6288 (schematic).

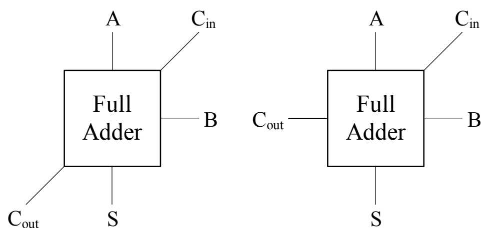

Figure 16. Full adder module in c6288 (symbolic).

5. Conclusions and Future Work

We have proposed a novel algorithm which generates the K longest testable paths through each gate (KLPG test set) in a combinational circuit. This was achieved by growing partial paths from the primary inputs, and during the procedure, implicit false path elimination techniques are used to trim the search space and guide the search more accurately toward a testable path. Based on the fact that many gates share long paths, the relations between gates and the global longest path generation are efficiently used to reduce repeated work. Experimental results show that the path generation is very efficient, and to our knowledge, this tool is faster than any existing tools solving similar problems, and it is the first tool that efficiently generates the longest testable path through each gate in c6288.

A process variation model is being developed for the path generation. So far we assume that there is no correlation between path delays. This model results in large K values for some gates if all the possible longest paths must be identified (the maximum K value can be 500+ if ±10% delay variation is assumed). Our preliminary results show that after path pruning using process correlation, the average number of possible longest paths per gate is 2-3 and almost all of these are in the top 10 nominal paths, so path generation time can be greatly reduced by incrementally generating and pruning paths.

Recent research also focuses on the path delay variation caused by capacitive coupling [27][28][29]. The delay of a path may be larger than its nominal delay due to capacitive coupling. Sometimes the delay increase cannot be ignored, and with technology scaling, the effect is becoming more significant. Considering that only a small fraction of lines have necessary assignments, in the justification process for a path, when a justification goal can be achieved by different means, the choice which causes larger path delay due to capacitive coupling is selected. It has also been observed that the vector pairs generated by our tool contain many “don’t care” bits. Therefore the vectors can be compacted or proper transitions can be assigned on these free bits to increase the path delay.

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

[4] 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.

[5] 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.

[6] 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.

[7] 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.

[8] M. Sharma and J. H. Patel, "Finding a Small Set of Longest Testable Paths that Cover Every Gate," *IEEE Int'l Test Conf.,* Baltimore, MD, Oct. 2002, pp. 974-982.

[9] V. Iyengar, B. K. Rosen and I. Spillinger, "Delay Test Generation 1 – Concepts and Coverage Metrics," *IEEE Int'l Test Conf.*, Washington, DC, Sept. 1988, pp. 857-866.

[10] J. L. Carter, V. S. Iyengar and B. K. Rosen, "Efficient Test Coverage Determination for Delay Faults," *IEEE Int'l Test Conf.*, Washington, DC, Sept. 1987, pp. 418-427.

[11] G. M. Luong and D. M. H. Walker, "Test Generation for Global Delay Faults," *IEEE Int'l Test Conf.,* Washington, DC, Oct. 1996, pp. 433-442.

[12] T. W. Williams, B. Underwood and M. R. Mercer, "The Interdependence Between Delay-Optimization of Synthesized Networks and Testing," *ACM/IEEE Design Automation Conf.*, San Francisco, CA, June 1991, pp. 87-92.

[13] 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.

[14] 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.

[15] 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.

[16] 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.

[17] 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*, vol. 14, no. 12, Dec. 1995, pp. 1505-1515.

[18] 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.

[19] 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*, vol. 13, no. 12, Dec. 1994, pp. 1550-1562.

[20] R. Stewart and J. Benkoski, "Static Timing Analysis Using Interval Constraints," *IEEE Int'l Conf. on Computer-Aided Design*, Santa Clara, CA, June 1991, pp. 308-311.

[21] Z. Li, X. Lu, W. Qiu, W. Shi and D. M. H. Walker, "A Circuit Level Fault Model for Resistive Opens and Bridges," *IEEE VLSI Test Symp.*, Napa Valley, CA, April-May 2003, pp. 379-384.

[22] H. Fujiwara and T. Shimono, "On the Acceleration of Test Generation Algorithms," *IEEE Trans. on Computers*, vol. 32, no. 12, Dec. 1983, pp. 215-222.

[23] B. R. Benware, R. Madge, C. Lu and R. Daasch, "Effectiveness Comparisons of Outlier Screening Methods for Frequency Dependent Defects on Complex ASICs," *IEEE VLSI Test Symp.*, Napa Valley, CA, April-May 2003, pp. 39-46.

[24] 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.

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

[26] W. Qiu and D. M. H. Walker, "Testing the Path Delay Faults for ISCAS85 Circuit c6288," *IEEE Int'l Workshop on Microprocessor Test and Verification*, Austin, TX, May 2003, pp. 38-43.

[27] W. Y. Chen, S. Gupta and M. Breuer, "Test Generation for Crosstalk-Induced Delay in Integrated Circuits," *IEEE Int'l Test Conf.,* Atlantic City, NJ, Sept. 1999, pp. 191-200.

[28] B. Choi and D. M. H. Walker, "Timing Analysis of Combinational Circuits Including Capacitive Coupling and Statistical Process Variation," *IEEE VLSI Test Symp.,* Montreal, Canada, April 2000, pp. 49-54.

[29] A. Krstic, J. J. Liou, Y. M. Jiang and K. T. Cheng, "Delay Testing Considering Crosstalk-Induced Effects," *IEEE Int'l Test Conf.*, Baltimore, MD, Oct. 2001, pp. 558-567.