

Memory Efficient ATPG for Path Delay Faults¹

Wangning Long*, Zhongcheng Li, Shiyuan Yang*, and Yinghua Min

CAD Lab., ICT, Chinese Academy of Science, Beijing, 100080

* Automation Department, Tsinghua University, Beijing, 100084

Abstract

A memory efficient test pattern generator for path delay faults, DTPG, is presented in this paper, which uses the efficient path identifier to represent a path. A compact bit table, path information table, is proposed to store test information efficiently. Furthermore, DTPG is capable of identifying functional sensitizable paths, which account for large percent of paths in many circuits. The experimental results show that DTPG is memory efficient. It generates tests for C3540 with 57 million paths and preserves the testability information for all paths. Experimental results show the influence of stepwise mandatory sensitization, multiple backtrace, and backtracking limits on the cpu time consumed by delay test generation process.

1 Introduction

Delay faults have been of major concern for high-performance ICs. The path delay fault model is considered practical and acceptable. Some test generation systems for path delay faults have been presented [1-3].

In some circuits, the number of paths is too large. For instance, C3540 of the ISCAS'85 circuits has 28.7 millions of physical paths, *i.e.* 57.4 millions of logical paths. Therefore path representation is vital to test generation for path delay faults. Path tree is used in DYNAMITE [2]. However too much space is needed to store the path tree. About 115 M bytes memory, for example, is required only for storage of the leaf nodes for C3540. Lack of efficient path representation method, DYNAMITE is unable to deal with large circuits such as C3540.

RESIST [3] is an efficient path delay test generation system benefited from a recursive test pattern generation algorithm. However no path information is preserved during the test generation procedure.

For a non-robust testable path, validatable non-robust test generation [1][4] or high quality test generation [5] are used. Besides, many paths in a circuit are non-robust untestable but irredundant, which are referred as functional sensitizable paths [6], for which a primitive test [7][8] or some resynthesis methods [7][4] may be used to improve the testability. All these methods need to know whether a path is robust, non-robust or functional sensitizable and

require preserving path information.

Path identifier is proposed in [9] as an efficient numeric representation of physical paths and further developed in [10] to represent logical paths. A path delay fault simulator based on path identifier is presented in [11], which is memory efficient and fast.

In this paper, a memory efficient test pattern generator for path delay faults, DTPG, is developed by using path identifier. In the system, methodologies of path sensitization in FAN [12], 10-valued and 3-valued logic in DANAMITE [2] are adopted. DTPG has advantages over the previous ATPG systems in following three aspects:

- (1) it uses path identifier to represent a logical path, and uses path information table to store the path information which can be preserved even after the testing process;
- (2) it is able to identify not only robust and non-robust testable paths, but also functional sensitizable paths. For a sensitizable path, it provides a sensitizing vector pair;
- (3) it is able to deal with the circuits with large number of paths such as C3540 in reasonable time and space complexity.

2 Path Identifier and Path Information Table

In this section, the path identifier is formally summarized based on [9], and then a compact bit table, path information table, is presented to store path information. Combinational circuit composed of simple gates, which are AND, OR, NAND, NOR and NOT gates, is concerned.

2.1 Path Identifier

A combinational circuit can be described by a **forward network**. Assuming the orders of primary inputs, primary outputs, and each fanout branches are given. Let the primary input set

$$PI = \{ I_1, I_2, \dots, I_m \} \quad (1)$$

and the primary output set

$$PO = \{ O_1, O_2, \dots, O_n \} \quad (2)$$

Each gate or primary input or primary output corresponds to a node. Physical connections correspond to an edge with the direction from *PI* to *PO*. Let the node set be *V*, edge set *E*. For each edge, there is a unique source node, and a unique terminal node. For each node *v* in *V*, the number of incoming edges is called its **incoming degree**, denoted by *IN(v)*, and the number of outgoing edges, **outgoing degree** *EN(v)*. Suppose *e* is an incoming edge of *v*, *EN(v)* is also called outgoing degree of *e*, *EN(e)=EN(v)* for convenience.

¹ This work was supported by National Natural Science Foundation of China under grant No. 69473024.

A **physical path** is any alternating sequence of nodes and edges

$$PP = \{ v_0, e_0, v_1, e_1, \dots, v_{k-1}, e_{k-1}, v_k \} \quad (3)$$

where v_0 in PI , v_k in PO , e_{i-1} is an incoming edge of v_i , and e_i is an outgoing edge of v_i for $1 \leq i \leq k$. We call v_0 the **origin** of path PP , and v_k its **destination**, denoted by $v_0 = I(PP)$ and $v_k = J(PP)$. Any **subpath** from v_i to v_j ($1 \leq i \leq k$, $1 \leq j \leq k$, $i \leq j$) is the alternating sequence of nodes and edges from v_i to v_j . A path can be specified only by its node sequence for simplicity. A function $N(e)$ is defined to be the number of subpaths from e to all POs. Obviously

$$N(e) = 1, \quad \text{if } e \text{ is an incoming edge of } v \in PO;$$

$$N(e) = N(h), \quad \text{if } EN(e) = 1, \text{ and } h \text{ is the successor of } e;$$

$$N(e) = N(h_1) + \dots + N(h_r), \quad \text{if } EN(e) > 1, \text{ and } h_1, \dots, h_r$$

are the successors of e .

For each edge e which is an outgoing edge of v , the **edge index** $D(e)$ is defined as follows:

$$(i) \text{ If } EN(v) = 1, \text{ then } D(e) = 0;$$

(ii) If $k = EN(v) > 1$, and e_1, \dots, e_k are outgoing edges of v , then

$$D(e_1) = 0;$$

$$D(e_i) = N(e_1) + \dots + N(e_{i-1}), \quad 2 < i \leq k$$

For each primary input I_i in PI , the **index** $D(I_i)$ is defined as follows:

$$D(I_1) = 0;$$

$D(I_i) = D(I_{i-1}) + N(e_1) + \dots + N(e_k)$, if $1 < i \leq n$, and e_1, \dots, e_k are all outgoing edges of I_{i-1} .

Then, for the path

$$PP = \{ v_0, e_0, v_1, e_1, \dots, v_{k-1}, e_{k-1}, v_k \}$$

the **path identifier** is

$$\lambda(PP) = D(v_0) + D(e_0) + \dots + D(e_{k-1}) \quad (4)$$

The following theorem is given with the proof omitted.

Theorem 1: If the primary inputs of a combinational circuit are $\{I_1, \dots, I_n\}$, and the fanout branches of I_n are $\{e_1, \dots, e_m\}$, then equation (4) gives a **one-to-one correspondence** between all physical paths and the integers from 0 to $(N_{PP} - 1)$, where N_{PP} is the number of physical paths of the circuit, and $N_{PP} = D(I_n) + N(e_1) + \dots + N(e_m)$.

For delay testing, a **logical path** $LP = \{U, PP\}$ is associated with the physical path PP , where $U \in \{U_0, U_1\}$, where U_0 represents a falling transition at v_0 , and U_1 a rising transition. The **logical path identifier** is defined as follows:

$$Q(LP) = \begin{cases} \lambda(PP) & \text{if } U = U_0 \\ \lambda(PP) + N_{PP} & \text{if } U = U_1 \end{cases}$$

where N_{PP} represents the total number of physical paths.

It is easy to find the path identifier for a given path according to (4). Conversely, a procedure to find the corresponding path for a given path identifier is given as follows:

find_logic_path(θ_p)

int θ_p ; /* path identifier */

```
{
  if ( $\theta_p \geq N_{PP}$ )
    { path[0] =  $U_1$ ;
       $\lambda = \theta_p - N_{PP}$ ; }
}
```

```
else { path[0] =  $U_0$ ;
```

```
       $\lambda = \theta_p$ ; }
```

```
for(  $i = 2$ ;  $i \leq \text{number\_of\_PIs}$ ;  $i++$  )
```

```
  if (  $\lambda < D(I_i)$  ) break;
```

```
  path[2] =  $I_{i-1}$ ;
```

```
   $\lambda = \lambda - D(I_{i-1})$ ;
```

```
   $e = I_{i-1}$ ;
```

```
   $pn = 1$ ; /* the path length */
```

```
  while(1)
```

```
    /* When the outgoing degree of  $e$  is 1 */
```

```
    while (  $EN[e] == 1$  &&  $e \notin PO$  )
```

```
      {  $e = \text{the\_successor\_of\_}e$ ;
```

```
        path[ $++pn$ ] =  $e$ ; }
```

```
  if( $e \in PO$ ) break;
```

```
  /* When the outgoing degree of  $e > 1$  */
```

```
  for( $i=2$ ;  $i \leq EN[e]$ ;  $i++$ )
```

```
    {  $a = \text{fo}[e].i$ ; /* Take the  $i$ th fanout branch. */
```

```
      if (  $\lambda < D(a)$  ) break; }
```

```
  path[ $++pn$ ] =  $\text{fo}[e].(i-1)$ ;
```

```
   $\lambda = \lambda - D(\text{fo}[e].(i-1))$ ;
```

```
}
```

```
return( $pn$ ); /* return the path length. */
```

```
}/* End of find_logic_path() */
```

2.2 Path Information Table

To utilize the efficiency of path identifier, a kind of bit table, path information table, is proposed in this paper to store the test information for all paths. Two path information tables, *Try_table* and *Testable_table*, are used in DTPG. *Try_table* stores the information of whether a given path is accessed or not, and *Testable_table* stores the information of whether it is robust (non-robust, or functional) testable.

$$\text{Try_table}[i] = \begin{cases} 0 & \text{if the } i\text{th path never tried} \\ 1 & \text{if tried} \end{cases}$$

$$\text{Testable_table}[i] = \begin{cases} 0 & \text{if the } i\text{th path is untestable} \\ 1 & \text{if testable} \end{cases}$$

The length of path information table is equal to N_{LP} , the number of logical paths. Each table is represented by $(N_{LP}/32 + 1)$ integers in a 32-bit operating system. When C3540 with 57 M logical paths is concerned, for example, each table occupies a memory space about 1.8 M integers, which is affordable for a general workstation.

The path information representation by using the bit tables can be accessed very efficiently. For instance, during delay test generation, very often it is necessary to access many consecutive paths, say N consecutive paths are robust untestable, which needs to access N leaf nodes one by one if path tree [2] is used. What we need in our case is just to set the consecutive N bits of *Try_Table* to be 1, and that of *Testable-table* to be 0. In 32-bit operating system, suppose the paths from a 1 to an N need to be set. If $a_N - a_1 \geq 32$, then, in the low end, set 1 bit $((32 - a_1) \bmod 32)$ times; in the high end, set 1 bit $(a_N \bmod 32)$ times, and the other

($\lfloor a_N/32 \rfloor - \lfloor a_1/32 \rfloor - 1$) integers are put to be -1, which is implemented in the function, `mark_bit_table(i_2, i_1 , Untestable)` (Refer to next section).

3 DTPG--A Test Generation System for Path Delay Faults

By using the path identifier and path information table, a test generation system for path delay faults, DTPG, is developed to identify robust, non-robust and functional delay testable paths, and to generate the sensitization pattern pair for testable paths.

DTPG is developed based on SABATPG [13]. Some conventional techniques in ATPG for stuck-at faults, such as implication, multiple backtrace [12] and static learning [14] are adopted. In addition, 10-valued logic and stepwise path sensitization [2] are also used. The path identifier technique is used throughout the program. It enables us to deal with the huge number of paths efficiently in both CPU time and memory space.

10-valued logic is used for robust and functional testing. 3-valued logic is used for non-robust testing. The values required to be assigned at the side-input [6] in each case is listed in Table 1. Suppose vector pair $T = \langle v_1, v_2 \rangle$ is applied. S_0 means it is always at 0. U_0 means it stabilizes at 0 under v_2 , but may be uncertain before stabilizing. S_1 and U_1 are similar to S_0 and U_0 respectively. $X_1 = \{S_1, U_1\}$, $X_0 = \{S_0, U_0\}$, $UX_1 = \{S_1, U_1, U_0\}$, $UX_0 = \{S_0, U_0, U_1\}$.

Table 1 The value requirement at side-input

Gate Type	Robust		Non-robust	Functional	
	Rising	Falling	Both	Rising	Falling
AND/NAND	X_1	S_1	X_1	X_1	UX_1
OR/NOR	S_0	X_0	X_0	UX_0	X_0

The main frame of the program is as follows:

```

delay_test_main()
{
    pre_processing();
    for(p_identifier=0; p_identifier < NLP; p_identifier++)
        if ( Try_table[p_identifier] == 0 ) {
            ok_flag = find_test_for_a_path(p_identifier);
        }
    final_report();
} /* End of delay_test_main() */

find_test_for_a_path(p_identifier)
int p_identifier;
{
    path_length = find_logic_path(p_identifier);
    recover_path_signal(p_identifier);
    conflict_flag = No;
    /* Phase 1 */
    for ( i1=1; i1 < path_length; i1++) {
        conflict_flag = implication();
        if ( conflict_flag == Yes ) break;
    }
}

```

```

/* Phase 2a ( if some conflicts occur during phase 1 ) */
if ( conflict_flag == Yes ) {
    i2 = backward_path_sensitization();
    mark_bit_table(i2, i1, Untestable);
    return (Fail);
}
/* Phase 2b ( if no conflict occurs during phase 1 ) */
while ( unjust_table != Null || conflict_flag == Yes ) {
    if ( conflict_flag == No )
        ok_flag = multi_backtrace();
    else ok_flag = Fail;
    if ( ok_flag == Fail ) {
        ok_flag = back_tracking();
        if ( ok_flag == Fail ) {
            mark_bit_table( 1, path_length, Untestable );
            if ( aborted_flag == Yes )
                number_aborted++;
            return ( Fail );
        }
    }
    conflict_flag = implication();
}
justify_head_line();
save_test_vector();
mark_bit_table( 1, path_length, Testable );
return (Successful);
} /* end of find_test_for_a_path() */

```

In `delay_test_main()`, `pre_processing()` is to do some initial processing. Then, for every untried logical path, `find_test_for_a_path()` is to generate a test for the path delay fault. If the path is testable, a test vector pair is generated, which is represented by a 10-logic or 3-logic vector. Next we will adopt an efficient parallel simulator [11][15] to find all other paths which the vector pair can test. If the path is untestable, an untestable sub-path will be found and then all the paths including the sub-path will be marked as tried and untestable in the path information table.

`find_test_for_a_path()` is a main procedure of DTPG, where **phase 1** and **phase 2a** are forward and backward stepwise mandatory sensitization procedure [2][6] respectively, by which most untestable paths are found. If a conflict is found during phase 1, say at the circuit line $path[i_1]$, then phase 2a is carried out backward from $path[i_1]$. The backward stepwise mandatory sensitization is continued until a conflict is found again, say at $path[i_2]$. The sub-path from $path[i_2]$ to $path[i_1]$ is called a prime segment [6]. Then the function `mark_bit_table(i_2, i_1 , Untestable)` is called to set all the bits as 1 in the `Try_table` correspondent to the paths including the sub-path, while the bits in `Testable_table` remain unchanged.

The function `mark_bit_table(i_2, i_1 , Untestable)` is efficient when $i_1 < path_length$, because the difference between the largest and smallest index of the sub-paths above $path[i_1]$ can be calculated easily and just for one time. When $i_2 > 1$ and $i_1 < path_length$, only backward depth-first search for the sub-paths below $path[i_2]$ and their index is

Table 2 Results of DTPG for Non-robust Testable Paths of ISCAS'85 Circuit

Circuit Name	N_{PP}	N_{LP}	Mem. by pi-tbl (int)	$maxb$	# of T-paths	# of ab.	fault coverage	U-B by [6]	U-B of ours	CPU (min.)
C181	929	1858	118	50	1312	0	70.61%	-----	70.61%	4.4 (s)
C432	291826	583652	36.48K	2048	15855	24	2.72%	2.89%	2.72%	6.83
C499	397888	795776	49.74K	2048	367521	223	46.18%	46.21%	46.21%	433.5
C880	8642	17284	1083	50	16652	0	96.34%	96.34%	96.34%	2.27
C1355	4173216	8346432	521.66K	512	1039005	71299	12.45%	13.30%	13.30%	1627.0
C1908	729057	1458114	91.14K	2048	355168	4173	24.36%	24.36%	24.64%	551.5
C2670	679960	1359920	85.00K	108	130626	70460	9.61%	9.61%	14.78%	699.6*
C3540	28676671	57353342	3.585M	108	1196642	29511	2.09%	2.15%	2.14%	1224.4
C5315	1341305	2682610	167.67K	2048	342117	440	12.75%	13.17%	12.77%	171.0
C7552	725494	1452988	90.81K	1024	276996	1458	19.06%	19.46%	19.16%	556.1*

* Tested on Sun Sparc Classic Workstation.

The others are on Sun Sparc 10 Workstation.

needed. However, in [6], the forward depth-first search for the sub-paths above $path[i]$ is also needed. This demonstrates again the advantage of using path identifier to represent a path in a path delay test generation program.

After the forward stepwise mandatory sensitization (phase 1) is taken along a path, if no conflict occurs, **phase 2b** is taken to identify if the path is testable and generate the vector pairs for the testable paths. In phase 2b, when some unjustified lines exist, multiple backtrace [12] is called repeatedly until a head-line is set as a final objective. Implication is then carried out. If some conflicts still exist in the implication after multiple backtrace, backtracking [12] is then necessary. If conflict occurs after all possible back-tracking is tried, then the path is untestable. If no conflict is found and no unjustified line exists, then the test is successfully generated. If the number of backtracking exceeds a given limit, the function returns with an abort flag, and the number of aborted times increase 1.

4 Experimental Results and Remarks

4.1 Experimental Results for Robust, Non-robust and Functional Testable paths

To show the efficiency of DTPG, some experimental results of test generation for non-robust, robust, and functional testable path delay faults are given in Table 2-4. In the experiments, the time needed to deal with the path information table can be omitted compared to the test generation process, and therefore it is not mentioned in the following discussion.

Table 2 shows the experimental results of test generation for non-robust testable path delay faults, which includes robust testable path delay faults, for ISCAS'85 benchmark circuits. In the table, ' N_{PP} ' stands for number of physical paths; ' N_{LP} '--number of logical paths; 'Mem. by pi-tbl (int)'--the memory (scaled in integers) used by the two path information tables; ' $maxb$ '--maximum number of back-trackings; '# of T-paths'--number of robust or non-robust testable path delay faults; '# of ab.'--number of aborted faults; 'fault coverage(%)'--percentage of non-robust

testable faults to the total number of logical path delay faults; 'U-B by [6] (%)'--the upper bound listed in [6]; 'U-B of ours(%)'--percentage of the sum of testable and aborted faults to the total; 'CPU'--CPU time (minutes) consumed by the whole test generation process in SUN SPARC 10 or Sun Sparc Classic workstation. It is noted that the percentage of robust or non-robust testable faults is less than or equal to the upper bound given by [6], but very close to it. It is also noted that the memory space needed to store the path information is very compact. For C3540, for example, only 3.59M integers are needed for the two path information tables in a 32-bit operation system.

Table 3 shows the experimental results of the robust delay testable faults. The meaning of the items is similar to that in Table 2. Comparing Table 2 and 3, we can obtain the percentage of non-robust testable but robust untestable faults.

To show the ability of DTPG to identify functional sensitizable paths, a simplified experiment of functional test generation is carried out for 7 ISCAS'85 circuits, where 5000 random paths are used and the number of max back-tracking is set to 100. The results are listed in Table 4. Ten-value logic implication is used for the functional sensitizing procedure, and the value required at the side inputs

Table 3 Results of DTPG for Robust Testable Paths of ISCAS'85 Circuit

Circuit Name	$maxb$	# of T-paths	# of ab.	fault coverage	U-B of ours	CPU (min.)
C181	1024	1160	0	62.43%	62.43%	6.4 (s)*
C432	2048	3618	596	0.62%	0.72%	69.18
C499	50	112122	171417	14.09%	35.63%	869.7
C880	2048	16080	3	93.03%	93.05%	5.8
C1355	1024	14797	17128	0.18%	0.38%	1071.3
C1908	40	84209	217904	5.78%	20.72%	530.1*
C2670	2048	15012	3234	1.10%	1.34%	181.5
C3540	128	77698	30237	0.14%	0.19%	1145.2
C5315	120	43458	45440	1.62%	3.31%	393.5
C7552	300	82705	8343	5.69%	6.27%	436.5

* Tested on Sun Sparc 10 Workstation.

The others are on Sun Sparc Classic Workstation.

of a gate along a given path is shown in Table 1. We have noticed that 10-value logic implication for functional delay test generation is not as efficient as for robust test generation. In Table 4, the upper bound of functional testable paths given by [6] is also listed for reference. It shows that our results is less than and close to the upper bounds. All the experiment is carried out on Sun Sparc Classic workstation.

Table 4 Results of functional sensitizable paths ($maxb=100$ and 5000 random paths are used)

Circuit Name	# of T-paths	# of ab.	fault coverage	U-B by [6]	CPU (secs)
C432	703	1023	14.06%	35.75%	235
C499	3339	205	66.78%	69.95%	487
C880	4862	89	97.24%	99.06%	139
C1355	844	83	16.89%	18.81%	692
C1908	2831	523	56.62%	67.21%	904
C5315	471	575	9.42%	21.95%	411
C7552	1349	222	26.98%	31.22%	758

4.2 The influence of Stepwise Mandatory Sensitization, Multiple Backtrace and Backtracking on CPU Time of DTPG

To show the influence of stepwise mandatory sensitization, multiple backtrace and backtracking on CPU time of DTPG, some experiments are carried out for non-robust delay testing on Sun SparcStation 10. The results are given in Table 5-7, where non-robust testable paths are referred to both robust and non-robust testable paths.

Table 5 lists the test results of stepwise mandatory sensitization, for which only phase 1 and phase 2a in `find_test_for_a_path()` are carried out while phase 2b is deleted from the program. In Table 5, ' N_{NRUT} ' means the number of non-robust untestable paths identified by the stepwise mandatory sensitization process; ' N_{NRUT} / N_{LP} '--the ratio of the number of non-robust untestable paths to the total number of logical paths; 'LU-B R_{NRUT}^l '--the lower upper bound of (N_{NRUT} / N_{LP}) , which is calculated from the data in Table 2 by formula $(N_{LP} - N_T - N_{ab}) / N_{LP}$, where N_T is the number of testable paths, and N_{ab} is the number of aborted paths; 'HU-B R_{NRUT}^h '--the higher

Table 5 The influence of stepwise mandatory sensitization on DTPG

Circuit Name	N_{NRUT}	$R_{NRUT} = N_{NRUT} / N_{LP}$	LU-B R_{NRUT}^l	HU-B R_{NRUT}^h	R_{NRUT} / R_{NRUT}^h	CPU (secs)
C432	562339	96.35%	97.28%	97.28%	99.04%	250.3
C499	428032	53.79%	53.79%	53.82%	99.94%	2181
C880	597	3.45%	3.66%	3.66%	94.26%	48
C1355	7236128	86.70%	86.70%	87.55%	99.03%	6743
C1908	1097155	75.24%	75.36%	75.64%	99.47%	3362
C2670	1133903	83.38%	85.22%	90.39%	92.24%	1642
C3540	55947709	97.55%	97.86%	97.91%	99.63%	32554
C5315	2323599	86.62%	87.23%	87.25%	99.28%	4427
C7552	1167528	80.35%	80.84%	80.94%	99.27%	3978

upper bound of (N_{NRUT} / N_{LP}) , with the calculation method similar to that for LU-B except by formula $(N_{LP} - N_T) / N_{LP}$; ' R_{NRUT} / R_{NRUT}^h '--the percentage of the number of non-robust testable paths identified by stepwise mandatory sensitization process to all non-robust untestable (NRUT) paths, where the later is represented by the higher upper bound.

Table 5 shows that for most of the ISCAS'85 circuits, at least 99% of non-robust untestable paths are identified by the stepwise mandatory sensitization process, and at least 92% of the other two are identified. On the other hand, compared to the whole ATPG, the time spent by stepwise mandatory sensitization process is quite limited. For C1908, for example, it spends only 3362 seconds to identify 75.24% paths in the circuit as non-robust untestable, which account for at least 99.47% of total NRUT paths; while it needs 33092 seconds to generate test for the testable paths when $maxb=2048$. This implies two things: (1) the stepwise mandatory sensitization process is efficient in identifying non-robust untestable paths; and (2) the CPU time of the ATPG program is dominated by process of generating test for the testable paths.

To generate a test vector pair for a non-robust testable path, multiple backtrace is necessary. To show the influence of multiple backtrace on DTPG, phase 2b in the function of `find_test_for_a_path()` is carried out after no conflict is found by phase 1, and multiple backtrace is performed without backtracking, i.e., $maxb$ is set to 0. The results are listed in Table 6, where '# left after phase 1' means the number of unjustified paths after phase 1. From Table 6, we notice that for 5 of the 8 circuits, more than half of the unjustified paths after phase 1 are identified by multiple backtrace without backtracking. However it spends longer time than the stepwise mandatory sensitization process, especially when $maxb > 0$. For C7552, for example, 4427 seconds are spent by the stepwise mandatory sensitization process, while $(9988 - 4427 = 5561)$ seconds are spent in phase 2b with multiple backtrace when $maxb=0$. The gain is that 192525 testable paths are identified and their test vector pairs are generated, which account for 67.44% of the unjustified paths after phase 1.

Although for most circuits, a large percent of unjustified paths after stepwise mandatory sensitization process can be identified only by multiple backtrace when $maxb=0$, we have to spend more time to identify the rest paths. To show the influence of backtracking on delay testing, DTPG is

Table 6 Non-robust test results when $maxb=0$

Circuit Name	# left after phase 1	# of T-paths	# of ab.	fault coverage	CPU (secs)
C432	21313	15515	5802	2.66%	295
C499	367753	147481	220263	18.53%	6470
C880	16687	15681	1006	90.73%	84
C1355	1110304	219768	890536	2.63%	18426
C1908	360959	122968	237991	8.43%	7207
C2670	226017	125070	100947	9.20%	3433
C5315	359011	229696	129315	8.56%	7084
C7552	285460	192525	92935	13.25%	9988

run for three ISCAS'85 circuits when $maxb=0$, 50 and 2048 respectively. The test results are listed in Table 7. For C499, for example, DTPG spends 6470 seconds with 220263 aborted paths when $maxb=0$ and 15954 seconds when $maxb=50$, which is 2.47 times of that when $maxb=0$. However the number of aborted paths is 45282 when $maxb=50$, which is only 20.56% of that when $maxb=0$. When $maxb$ is enlarged to 2048, the CPU time increases to 26007, and the number of aborted paths left is just 223. However, it is noticed from C1908 that the enlargement of $maxb$ does not necessarily reduce the number of aborted paths significantly.

Table 7 The influence of backtracking on DTPG

Circuit Name	$maxb = 0$		$maxb = 50$		$maxb = 2048$	
	# ab.	CPU (s)	# ab.	CPU (s)	# ab.	CPU (s)
C432	5802	295	271	362	24	410
C499	220263	6470	45282	15954	223	26007
C1908	237991	7208	9594	14079	4173	33092

5 Conclusion

This paper presents a memory efficient test pattern generator for path delay faults, DTPG, which employs the path identifier [9] to represent circuit paths. A bit table, path information table, is proposed to store the test information efficiently for all paths in a circuit. With the path identifier and the path information table, DTPG generates robust and non-robust tests for larger circuits than that by previous schemes, such as DYNAMITE [2]. It also identifies functional sensitizable paths, which account for large percent of paths in many circuits. We believe that path information is applicable to other application, such as primitive fault test generation [7][8], resynthesis of a circuit [7][4], and high quality test generation [5].

We also show the influence of stepwise mandatory sensitization, multiple backtrace, and backtracking on CPU time spent by test generation programs. We note that for most circuits, 99% or more non-robust untestable paths can be identified by stepwise mandatory sensitization process. In addition, backtracking is the most time consuming step, and large backtrack limit does not necessarily increase the fault coverage significantly.

6 Acknowledgment

This work was done when W.N. Long visited the Center for Fault-Tolerant Computing, CAD Lab of ICT, Chinese Academy of Science. The authors would like to thank Dr. Xiaoqing Wen for his assistance in the experiments.

References

- [1] Reddy, C. J. Lin, and S. Patil, "An Automatic Test Pattern Generator for the Detection of Path Delay Faults", In: Proc. of ICCAD, pp.284-287, 1987.
- [2] K. Fuchs, F. Fink, and M. H. Schulz, "DYNAMITE: An Efficient Automatic Test Pattern Generation System for Path Delay Faults", IEEE Trans. on CAD, Vol.10, No.10, pp1323-1335, Oct. 1991.
- [3] K. Fuchs, M. Pabst, T. Rossel, "RESIST: A Recursive Test Pattern Generation Algorithm for Path Delay Faults", In: Proc. European Design Automation Conf., pp316-321, 1994.
- [4] S. Devadas and K. Keutzer, "Validatable Non-robust Delay-Fault Testable Circuits", IEEE, Trans. on CAD, Vol.11, No.12, pp1559-1573, Dec., 1992.
- [5] K.-T. Cheng, A. Krstic, H.-C. Chen, "Generation of High Quality Tests for Robustly Untestable Path Delay Faults", IEEE Trans. on Computer Vol.45, No.12, pp.1779-1392, Dec., 1996.
- [6] K.-T. Cheng, H.-C. Chen, "Classification and Identification of Non-robust Untestable Path Delay Faults", IEEE Trans. on CAD, Vol.15, No.8, Aug. 1996.
- [7] W. Ke and P. R. Menon, "Synthesis of Delay-Verifiable Combinational Circuits", IEEE Trans. on Computers, Vol.44, No.2, pp.213-222 Feb. 1995.
- [8] A. Krstic, K.-T. Cheng, and S. T. Chakradhar, "Identification and Test Generation for Primitive Faults", In: Proc. of International Test Conference, 1996.
- [9] I. Pomeranz, L. N. Reddy, and S. M. Reddy, "SPADES: A Simulator for Path Delay Faults in Sequential Circuits", In: Proc. of European Conf. on Design Automation, pp.428-432, Sept., 1992.
- [10] H. Wittmann, M. Henfling, "Efficient Path Identification for Delay Testing--Time and Space Optimization", In: Proc. of European Design & Test Conf., pp.513-517, Feb., 1994.
- [11] M. C. Lin, J. E. Chen, and C. L. Lee, "A Fast and Memory efficient Path Delay Fault Simulator", In: Proc. of European Design & Test Conf., pp.508-512, Feb., 1994.
- [12] H. Fujiwara and T. Shimono, "On the Acceleration of Test Generation Algorithms", IEEE Trans. on Computers, vol.C-32, pp.1137-1144, Dec. 1983.
- [13] Zhongcheng Li, Yuqi Pan, Yinghua Min, "SABATPG--A Structural Analysis Based Automatic Test Generation System", Science in China (Series A), Vol.37, No. 9, pp104-114, Sept. 1994.
- [14] M. H. Schulz, E. Trischler, and T. M. Sarfert, "SOCRATES: A Highly Efficient Automatic Test Pattern Generation System", IEEE T-CAD, pp.126-137, Jan. 1988.
- [15] M. H. Schulz, K. Fuchs, and F. Fink, "Parallel pattern Fault simulation of path delay faults", In: Proc. of 26th DAC, pp357-363, June 1989.