RESIST: A Recursive Test Pattern Generation Algorithm for Path Delay Faults

Karl Fuchs
Siemens Mobile Radio Networks
81359 Munich, Germany

Michael Pabst
Siemens Transportation Systems
38126 Braunschweig, Germany

Torsten Rössel
Siemens Corporate R & D
81739 Munich, Germany

Abstract

This paper presents RESIST, *a recursive test pattern generation (TPG) algorithm for path delay fault testing of scan-based circuits. In contrast to other approaches, it exploits the fact that many paths in a circuit have common subpaths.* RESIST *sensitizes those subpaths only once, reducing the number of value assignments during path sensitization significantly. In addition, our procedure identifies large sets of redundant path delay faults without enumerating them.* RESIST *is capable of performing TPG for all path delay faults in all ISCAS-85 and ISCAS-89 circuits. For the first time, results for all path delay faults in circuit c6288 are presented. A comparison with other TPG systems revealed that* RESIST *is significantly faster than all previously published methods.*

1 Introduction

In synthesized circuits, delay testing becomes mandatory since many paths have a propagation delay close to the maximum circuit delay [1]. The path delay fault model is the most comprehensive model for real delay defects. Recent research has solved some problems associated with this model. In [2] it is shown that complete path delay fault testability is not always necessary to guarantee circuit performance, i.e., test patterns for a small fraction of testable path delay faults may provide a sufficient test quality. In addition, synthesis for delay fault testability can increase the delay fault coverage. A remaining problem, however, is the determination of a test set that detects all robustly testable path delay faults (RPDFs) in practical circuits. Since most methods only find a small fraction of all RPDFs with reasonable effort, there is still need for efficient TPG tools.

TPG for path delay faults is either based on branch-and-bound algorithms [3,4,5,6,7,8,9] or on Boolean algebraic methods [10,11]. No conventional TPG algorithm that targets a specific fault can cope with the large number of paths existing in practical circuits. Today's fastest TPG methods are TSUNAMI-D [11], NEST [9], and DYNAMITE [8]. These methods can handle large path sets but they have certain limitations.

TSUNAMI-D uses reduced ordered binary decision diagrams (ROBDDs) to represent constraint functions that have to be satisfied by a delay test. This approach, however, requires post-processing steps to check for robustness. Since an exhaustive check may be prohibitively cpu intensive, TSUNAMI-D only provides a conservative estimate of the robust path delay fault coverage.

NEST determines pairs of signals such that a maximum number of paths between the signals can be simultaneously tested. In order to sensitize the paths between two selected signals, test generation objectives are established. Once the objectives are satisfied, the fault simulation method from [12] is used to estimate how many faults are actually detected. As mentioned in [9], the algorithm is most effective in highly testable circuits but may fail in those poorly testable. A comparison between NEST and our algorithm reveals that NEST also provides only a conservative estimate of the robust path delay fault coverage for the ISCAS-85 and ISCAS-89 circuits.

DYNAMITE applies improved redundancy identification techniques. All selected paths are stored in a path tree. A stepwise path sensitization procedure identifies sets of redundant path delay faults without enumerating them. This method is very effective in poorly testable circuits but many faults have to be treated separately in those highly testable. A limitation of this approach is the path tree which is impractical for large circuits. Because of limited memory resources, the set of all path delay faults must usually be partitioned into many subsets.

Our algorithm, named RESIST (recursive selection and sensitization technique for path delay faults), overcomes the limitations of previously published methods. It is a cost-effective way to generate delay tests for a large number of path delay faults in both poorly testable and highly testable circuits.

In contrast to DYNAMITE, RESIST needs no path tree. Therefore, no fault set partitioning is required. As with DYNAMITE, many path delay faults are concurrently identified as redundant. For redundancy identification, however, no path tree traversal is needed. To efficiently handle highly testable circuits, we improved the path sensitization step. RESIST exploits the fact that faults are dependent in the path delay fault model, i.e., paths in a circuit usually have sections in common. In order to eliminate the repetition of work, RESIST executes the sensitization step for common subpaths only once. This technique significantly decreases the number of value assignments during path sensitization. Compared to conventional methods, a speed-up factor that grows

linearly with the circuit depth is obtained. Experimental results revealed that RESIST is significantly faster than TSUNAMI-D, NEST, and DYNAMITE.

Section 2 gives some definitions. Section 3 presents the basic idea and implementation of RESIST. Section 4 contains experimental results on TPG for the ISCAS-85 and ISCAS-89 circuits and Section 5 contains our conclusions.

2 Basic Definitions

2.1 Structural and Functional Paths

A structural path $P = (x_0, ..., x_n)$ starts at a primary input (PI) x_0 and ends at a primary output (PO) x_n. P consists of several structural subpaths $S_1, ..., S_m$. Each subpath leads from a PI or a fanout stem to a PO or a fanout stem. To simplify the following discussion, we will assume that there are no XOR- or XNOR-gates in the circuit. Then, two functional paths $P_r = (x_0^r, ..., x_n^{t_n})$ and $P_f = (x_0^f, ..., x_n^{t_n})$ are associated with a structural path P (rising transition (r) or falling transition (f) at x_0). The transitions $t_i \in \{r, f\}$ at all other signals x_i, $1 \leq i \leq n$, are uniquely determined by the gates (inverting or non-inverting) along path P. Signals $x_0, ..., x_n$ are called **on-path signals**. All other inputs of the gates along path P are referred to as **off-path signals**.

2.2 Conflicts, Updates, Unjustified Lines

Logic $\mathcal{A}_{10}$ proposed in [5] is used for TPG of robust tests. A robust test remains valid even if there are multiple path delay faults in the circuit. The **basic values** ($\mathcal{B}_{10}$) and **composite values** ($\mathcal{C}_{10}$) of this logic are shown in Fig. 1. In particular, 0s and 1s denote the stable values 0 and 1. $0\bar{s}$ is either a static zero hazard or a falling transition and $1\bar{s}$ is either a static one hazard or a rising transition. A composite value $c \in \mathcal{C}_{10}$ is given by $c = \{b_1, ..., b_n\}$ where $b_1, ..., b_n \in \mathcal{B}_{10}$ represent basic values. The relation between a value $v \in \mathcal{A}_{10}$ and its basic values can be described by a mapping $\beta : \mathcal{A}_{10} \rightarrow 2^{\mathcal{B}_{10}}$, where $\beta(v) = \{b_1, ..., b_n\}$ if v is a composite value and $\beta(v) = \{b\}$ if v is a basic value.

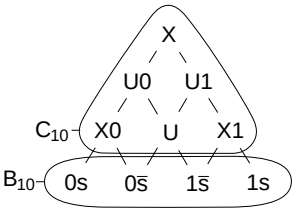

Figure 1: Hasse diagram of the 10-valued logic $\mathcal{A}_{10}$.

Let us assume that a signal x already has a value $v_i \in \mathcal{A}_{10}$ and the test generator tries to assign value $w_i \in \mathcal{A}_{10}$ to x. Depending on these values, two cases are possible. If the two values v_i and w_i have no basic value in common, i.e.,

$$\beta(v_i) \cap \beta(w_i) = \emptyset,$$

then there is a **conflict** that must be resolved by backtracking. Otherwise, if $\beta(v_i)$ and $\beta(w_i)$ intersect, there is a value $v_{i+1} \in \mathcal{A}_{10}$ with $\beta(v_{i+1}) = \beta(v_i) \cap \beta(w_i)$. Obviously, value v_{i+1} represents the basic values of the intersection. If $\beta(v_{i+1}) \neq \beta(v_i)$, the test generator assigns the new value v_{i+1} to x which is called the **update** of v_i by w_i. For example, the update of U0 by X1 is $1\bar{s}$ since $\beta(1\bar{s}) = \beta(\text{U0}) \cap \beta(\text{X1})$ (see Fig. 1).

Let G be a gate and let $v_i, v_j \in \mathcal{A}_{10}$ be the values at its inputs i and j. Furthermore, let "$\circ$" be the gate function of G and let v_y be the value at output y. y is called an **unjustified line**, if

$$\beta(v_y) \quad \subset \quad \beta(v_i \circ v_j).$$

2.3 TPG Status

The values at unjustified lines are satisfied via a backtrack search. Tentative value assignments are performed at head lines [13]. Each time an assignment leads to a conflict, backtracking is performed. The set of unjustified lines as well as the signal updates must be stored to continue the search after backtracking. Several stacks are used for that purpose. These stacks determine the TPG status $TS = (US, VS(x_0), ..., VS(x_N))$. US denotes the unjustified line stack. An entry of stack US is a set of unjustified lines. $VS(x_0), ..., VS(x_N)$ are the value stacks for all signals $x_0, ..., x_N$ in the circuit. A value stack $VS(x_i) = (...(((v_0), v_1), v_2), ..., v_k)$, $0 \leq i \leq N$, consists of the updates $v_1, v_2, ..., v_k$ of signal x_i. The initial value v_0 is X which is assigned to each signal in the initialization step of the TPG process. The value stacks are bounded by the maximum number of updates (3 for logic $\mathcal{A}_{10}$), and the unjustified line stack is bounded by the number of PIs.

3 The Test Generation Algorithm

The TPG process can be divided into path sensitization and line justification. The path sensitization step consists of assigning values to all on-path and off-path signals and performing all implications of these assignments. After all mandatory value assignments have been carried out, a target path is said to be sensitized. The logic values of all unjustified lines must then be justified. This is done via a backtrack search. The TPG process is finished when there are no more unjustified lines.

The complexity of line justification may be exponential in the number of PIs. The complexity of path sensitization directly depends on the number of selected paths. In the following we describe a path sensitization strategy which significantly reduces the computation time required for this step.

3.1 Basic Idea of RESIST

Path delay faults are dependent since many paths have common subpaths. Therefore, treating each path delay fault separately is inappropriate. To illustrate this, let us use the circuit example C_n from [9] (see Fig. 2). Only the bold paths starting from PI y_0 and ending at PO y_n will be considered. There are $2 \cdot n$ different structural subpaths $S_{i,i+1}^k$, $0 \leq i < n$, $1 \leq k \leq 2$, between consecutive fanout stems. Each path from y_0 to y_n consists of n subpaths $S_{i,i+1}^k$. Altogether, there are 2^n different paths from y_0 to y_n.

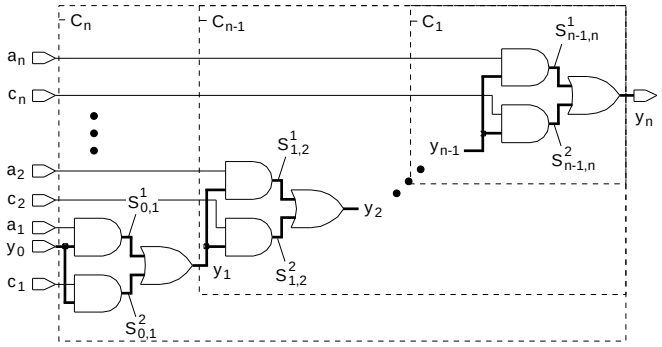

Figure 2: Circuit C_n from [9].

A conventional test pattern generator (CTPG) sensitizes each path separately. Hence, it performs $S(n) := n \cdot 2^n$ subpath sensitization steps (SPS) in circuit C_n. To reduce the number of SPS, RESIST uses a different strategy.

Let us consider the subpaths $S^1_{0,1}$ and $S^2_{0,1}$ from PI y_0 to fanout stem y_1. Both subpaths are included in 2^{n-1} paths from y_0 to y_n. Hence, sensitizing $S^1_{0,1}$ and $S^2_{0,1}$ only once reduces the number of SPS from $S(n)$ to

$$2 \cdot \left[1 + (n-1) \cdot 2^{n-1}\right] \quad = \quad 2 + 2 \cdot S(n-1),$$

where $S(n-1) := (n-1) \cdot 2^{n-1}$ is the number of SPS performed by CTPG in circuit C_{n-1}. Applying the same principle to all subcircuits $C_{n-1}, ..., C_1$ (see Fig. 2), the number of SPS becomes

$$2 + 2 \cdot [2 + 2 \cdot [...[2 + 2 \cdot S(1)]...]] =$$

$$2^1 + ... + 2^{n-1} + 2^{n-1} \cdot S(1) = \sum_{i=1}^{n} 2^i = 2^{n+1} - 2,$$

since $S(1) = 2$. Compared with CTPG, the number of SPS is reduced by $\frac{n \cdot 2^n}{2^{n+1}-2} = \frac{n}{2-2^{-n+1}} \approx \frac{n}{2}$. If $n = 100$, for example, the reduction factor is 50. The reduction factor increases with an increasing number of fanout branches ($fanout > 2$) converging to n.

Performing the sensitization step for a common subpath S only once requires that the TPG status is updated at fanout stems. Both, mandatory value assignments for subpath sensitization and the corresponding unjustified lines have to be stored in order to exploit the TPG status for all paths including S.

3.2 Pseudo-Code of RESIST

The pseudo-code of our test generation procedure is depicted in Fig 3. RESIST is called twice for each PI x_0, where t_0 is first initialized to a rising transition and second to a falling transition. The procedure consists of path selection, path sensitization, and line justification.

Starting at a PI, the netlist of the circuit is traversed in a depth-first manner. Path sensitization is performed during the netlist traversal. The path sensitization step is completed if the actual signal is a PO. In this case a path from PI x_0 to PO x_n has been selected and sensitized. A backtrack search is then used

```
RESIST( x_i, t_i ) {
    /* x_i: on-path signal, t_i: transition at x_i. */
    if ( x_i is a PO ) {
(1)   satisfy_unjustified_lines();   /* Backtrack search. */
    } else {
      if ( fanout(x_i) > 1 ) {
(2)     implication();
        V_imp := {(x, v_k) | v_k is the last update assigned
                             to x during implication};
        U := actual set of unjustified lines;
        if ( no conflict ) {
(3)       foreach ( (x, v_k) ∈ V_imp ) { push(v_k, VS(x)); }
(4)       push(U, US);
        } else {
          foreach ( (x, v_k) ∈ V_imp ) {
(5)         assign value top( VS(x)) to signal x; }
          return;
      } }

(6)   foreach ( branch signal b_j of x_i ) {
        G_i := gate with input signal b_j;
        I(G_i) := set of input signals of G_i;
        x_{i+1} := output signal of G_i;

        if ( G_i is a buffer or AND/OR-gate ) {
(7)       assign_gate_values( x_i, I(G_i)\{x_i}, t_i, type of G_i );
          V_gate = {(x, v_k) | v_k is the update assigned to x
                               during path sensitization at G_i};
          if ( no conflict ) {
            foreach ( (x, v_k) ∈ V_gate ) { push(v_k, VS(x)); }
            t_{i+1} := t_i;
            RESIST( x_{i+1}, t_{i+1} );   /* Recursive call. */

            /* Restore TPG status. */
(8)         foreach ( (x, v_k) ∈ V_gate ) {
(9)           pop( VS(x));
(10)          assign value top( VS(x)) to signal x;
            }
            if ( fanout(x_i) > 1 ) {
              if ( b_j is the last processed branch ) {
(11)            foreach ( (x, v_k) ∈ V_imp ) {
(12)              pop( VS(x));
(13)              assign value top( VS(x)) to signal x; }
(14)            pop( US);
              } else {
(15)            actual set of unjustified lines := top( US);
              }
            }
          } else {
(16)        foreach ( (x, v_k) ∈ V_gate ) {
(17)          assign value top( VS(x)) to signal x; }
          }

        } else if ( G_i is an inverter or NAND/NOR-gate ) {
          /* Similar to above. */
        } else if ( G_i is a XOR/XNOR-gate ) {
          /* Similar to above. */
        } } }
}
```

Figure 3: Pseudo-code of RESIST.

for line justification (step (1)). The backtrack search is guided by a multiple backtrace procedure which is based upon the principles described in [13]. After gen-

erating a test or proving the fault as redundant, the path selection and sensitization process is continued.

In the sequel we assume that the actual signal x_i is no PO. We distinguish between two cases: 1) x_i is a fanout stem, i.e. $fanout(x_i) > 1$, and x_i does not branch, i.e. $fanout(x_i) = 1$. To refer to gate input and output signals we use the following notation: b_j denotes a branch signal of x_i, G_i is the gate with input signal b_j, $I(G_i)$ is the set of inputs of G_i, and x_{i+1} is the output of G_i. Note that x_i has only one branch if $fanout(x_i) = 1$. Since the next steps of RESIST are similar for all considered gate types, we will suppose that G_i is a non-inverting gate (buffer or AND/OR-gate).

Each time a fanout stem is encountered, the current TPG status $TS = (US, VS(x_0), ..., VS(x_N))$ has to be updated. Initially, the TPG status TS = (empty stack, ..., empty stack). Let us assume that S denotes the selected subpath from the last encountered fanout stem x_j or from PI x_0 to the actual fanout stem x_i. The value assignments to the on-path signals (except x_i) and off-path signals of S have already been done in previous calls of RESIST. In step (2) we complete the sensitization of subpath S by performing all implications of these assignments. Assuming that no conflict occurred in step (2), the last update of a signal included in set $\mathcal{V}_{\text{imp}}$ and the actual set $\mathcal{U}$ of all unjustified lines are stored in TS (see steps (3) and (4) in Fig. 3). Note that several updates may be assigned to a signal x during the implication step but only the last one has to be stored in the value stack $VS(x)$. After updating the TPG status, $top(TS) := (top(US), top(VS(x_0)), ..., top(VS(x_N)))$ represents the unjustified lines of the sensitized subpath $S = (x_0^{t_0}, ..., x_i^{t_i})$ and the actual signal assignments in the circuit. Hence, entry $top(TS)$ is common for all paths that start at fanout stem x_i. If a conflict occurred in step (2), all paths including subpath S are identified as untestable. Consequently, path selection and sensitization is stopped at x_i. Before continuing the path selection and sensitization process at the last encountered fanout stem x_j or at PI x_0, each signal x with $(x, v_k) \in \mathcal{V}_{\text{imp}}$ gets its previous value which is stored in $top(VS(x))$ (see step (5)).

After all implications are done and the TPG status is stored, path selection is continued in step (6). While steps (2)-(5) are only executed for fanout stems, the next steps are required for all signals. For each branch signal b_j of x_i, a new subpath $(x_i^{t_i}, x_{i+1}^{t_{i+1}})$ is first selected and subsequently sensitized in step (7). Table 1 shows the values which are assigned to on-path signal x_i and to the off-path signals $I(G_i) \setminus \{x_i\}$ depending on the transition t_i and the type of G_i. If no conflict occurs in step (7), all signal updates (see set $\mathcal{V}_{\text{gate}}$ in Fig. 3) are stored in the corresponding value stacks and RESIST is invoked with the gate output x_{i+1} and the transition at that signal. After processing all subpaths in the output cone of signal x_{i+1}, the TPG status has to be restored.

First, the value assignments for subpath sensitization are undone. In steps (8)-(10), the corresponding value stacks are popped and the signals get their previous values. Next, we distinguish two cases:

1) $fanout(x_i) > 1$ **and b_j was the last processed branch**: Since all branches of x_i have been processed, the updates of TS performed at x_i in steps (3) and (4) are no longer required. Hence, the corresponding stacks of TS are popped and the value assignments of the implication step have to be undone. Each signal x with $(x, v_k) \in \mathcal{V}_{\text{imp}}$ gets its previous value $top(VS(x))$ (see steps (11)-(14) in Fig. 3).

2) $fanout(x_i) > 1$ **and there are still unprocessed branches**: $top(TS)$ is still required for the unselected subpaths starting at x_i. Hence, only the actual set of unjustified lines is restored, i.e., $top(US)$ becomes the unjustified line set (see step (15)).

If a conflict occurs in step (7), all paths including subpath $S = (x_0^{t_0}, ..., x_{i+1}^{t_{i+1}})$ are identified as redundant. Hence, path selection and sensitization is stopped and each signal x with $(x, v_k) \in \mathcal{V}_{\text{gate}}$ gets its previous value (see steps (16) and (17)).

Finally, path selection and sensitization is continued at the last encountered fanout stem x_j or at PI x_0. RESIST stops when all paths in the output cone of PI x_0 have been processed. Note that it is also possible to target only maximum length paths by taking the nominal gate delays into account.

Off-Path Signals					On-Path Signals	
AND/NAND		OR/NOR		XOR/XNOR		
rising	falling	rising	falling		rising	falling
X1	1s	0s	X0	0s/1s	$1\overline{s}$	$0\overline{s}$

Table 1: Logic values assigned to on-path and off-path signals during path sensitization.

Until now, we assumed that there are no XOR/XNOR-gates in the circuit. If the actual signal x_i is the input of a XOR/XNOR-gate, two recursive calls of RESIST must be performed (assigning 0s and 1s to the off-path signal, see Table 1). Subpath $(x_i^r, x_{i+1}^{r/f})$ or $(x_i^f, x_{i+1}^{f/r})$ is sensitized in the one case and $(x_i^r, x_{i+1}^{f/r})$ or $(x_i^f, x_{i+1}^{r/f})$ in the other case. In order to exploit the TPG status for both subpaths, the TPG status must also be updated at input signals of XOR/XNOR-gates.

4 Experimental Results

RESIST was implemented in 18 000 lines of C code. For evaluation, we used the ISCAS-85 and ISCAS-89 benchmark circuits. We compared our results with those obtained with DYNAMITE [8], TSUNAMI-D [11], and NEST [9]. Table 2 gives the number of all path delay faults, the number of robustly tested faults, and the required CPU time. Note that only circuits with a large number of path delay faults ($> 10\,000$) are listed. Aborted faults occurred in circuits c432, c499, c1355, c1908, c3540, and c6288.

Circuit Name	All Path Delay Faults			Tested Path Delay Faults			CPU time [sec.]			
	RESIST, DYNAM.	TSUN.-D	NEST	RESIST, DYNAM.	TSUN.-D	NEST	RESIST [1]	DYNAM. [2]	TSUN.-D [2]	NEST [3]
s713	43624	43624	43624	1184	1184	181	3	15	20	105
s1423	89452	89452	89452	28696	23220	465	92	315	926	176
s5378	27084	27084	27084	18656	18248	2035	76	198	296	523
s9234	489708	-	489708	21389	-	2079	175	472	-	6789
s13207	2690738	2690638	-	27603	27484	-	167	1984	4286	-
s15850	329476092	-	-	182673	-	-	1478	86040	-	-
s35932	394282	394282	-	21783	21655	-	636	1135	439	-
s38417	2783158	-	-	598062	-	-	9819	26392	-	-
s38584	2161446	2161442	-	92239	90146	-	607	10636	4790	-
	RESIST	-	NEST	RESIST	-	NEST	RESIST [1]	-	-	NEST [3]
c432	583652	-	-	3722*	-	-	137	-	-	-
c499	795776	-	-	132684*	-	-	3327	-	-	-
c880	17284	-	17284	16083	-	768	58	-	-	30
c1355	8346432	-	8346432	22624*	-	78	2473	-	-	4502
c1908	1458114	-	1458112	97588*	-	442	21224	-	-	246
c2670	1359768	-	1359756	15218	-	790	318	-	-	3826
c3540	57353342	-	56531748	88378*	-	842	5224	-	-	24389
c5315	2682610	-	2682610	81435	-	1668	1524	-	-	9705
c6288	$1.98 \cdot 10^{20}$	-	$1.98 \cdot 10^{20}$	12592*	-	1	1122 h	-	-	45739
c7552	1452986	-	1452986	86250	-	2410	2457	-	-	24145

[1]CPU time on a SUN SPARC IPX (28 Mips). [2]CPU time on a DEC 5000 (25 Mips).
[3]CPU time on a SUN SPARC 2 (28 Mips).
- No results are available for these circuits. * Circuits with aborted faults.

Table 2: Results of RESIST, DYNAMITE [8], TSUNAMI-D [11], and NEST [9] for TPG of robust tests.

The data show that RESIST is significantly faster than DYNAMITE, especially in circuits with a large number of path delay faults like s13207, s15850, and s38584. For example, in circuit s15850, RESIST is nearly two orders of magnitude faster than DYNAMITE.

A comparison with TSUNAMI-D reveals that RESIST is faster for circuits s713, s1423, s5378, s13207, and s38584 while TSUNAMI-D is slightly faster for circuit s35932. As can be seen from the number of tested faults, TSUNAMI-D provides only a conservative estimate of the robust path delay fault coverage. For example, in circuit s1423, 5476 faults are not proven to be robustly testable by TSUNAMI-D.

NEST gives an even lower fault coverage. In all listed circuits, it detected significantly less faults. In some poorly testable ISCAS-85 circuits (c1355, c1908, c3540, c6288), the tested faults differ by more than two orders of magnitude. Although RESIST gave a higher fault coverage and used no fault simulator to accelerate the TPG process, it is considerably faster than NEST except in circuits c880, c1908, and c6288. The higher CPU times required in these circuits are a consequence of the much larger number of tested faults.

Detailed results on TPG for circuit c6288 are depicted in Fig. 4. We invoked RESIST for each PI separately. Fig. 4 shows the number of tested, aborted, and proven redundant faults as well as the required CPU time for each output cone (OC) of a PI. Note that the 32 OCs are ordered in terms of increasing number of paths starting at the corresponding PIs. Besides a backtrack limit of 10 a total time limit of 500 000 seconds was used.

RESIST detected 12 592 testable faults while NEST found only one testable fault. $1.96789 \cdot 10^{20}$ faults were proven as redundant. From the remaining faults, 1 132 540 were aborted because of the backtrack limit and about $1.09755 \cdot 10^{18}$ were aborted because of the time limit.

As can be seen in Fig. 4, the graphs for CPU-time and for the number of aborted faults are similar. In four cases (OC_{22}, OC_{21}, OC_{20}, OC_{19}) the time limit was exceeded resulting in numerous aborted faults. In the remaining 28 OCs, relatively few faults are aborted because large sets of faults were proven redundant rapidly. The remaining aborted faults occurred due to the backtrack search limit. The best results were obtained in OC_{17} which has $7.34 \cdot 10^{19}$ faults. Only 7.3 seconds were needed for TPG with no aborted fault.

5 Conclusions

We presented RESIST that is capable of performing TPG for all path delay faults in the complete ISCAS benchmark set. We proposed an efficient sensitization technique which sensitizes common subpaths only once. Our method resulted in a substantial decrease in the number of subpath sensitization steps. The sensitization procedure has been shown to give a speed-up factor that grows linearly with the circuit depth. Compared to conventional approaches, RESIST is capable of detecting a significantly larger number of path delay faults in less time. Especially noticeable is the fact that RESIST can handle circuit c6288 which

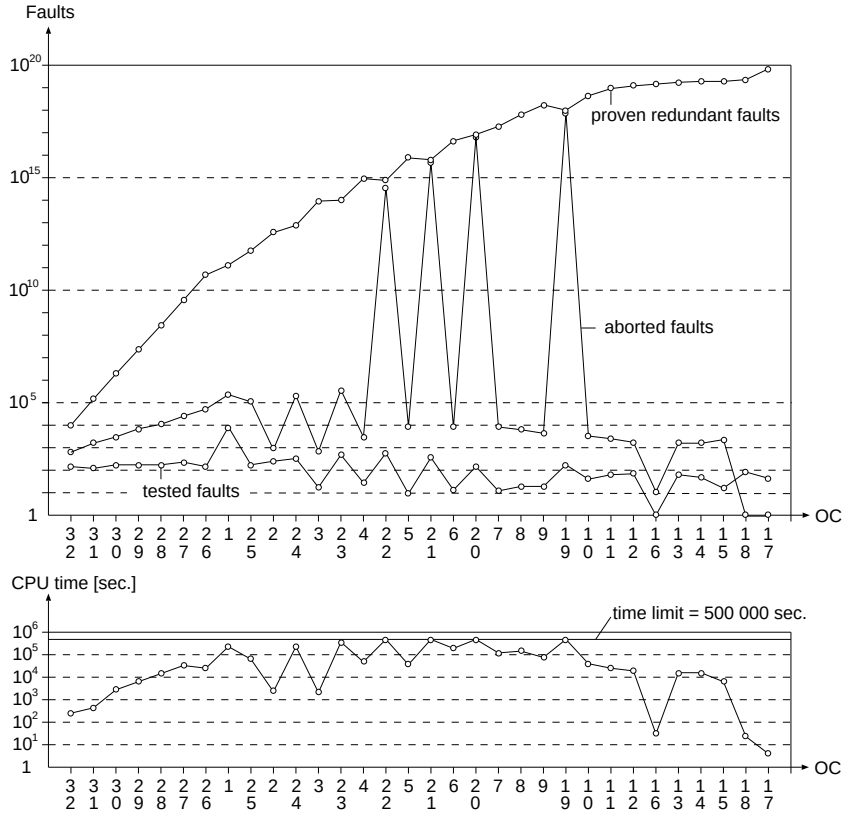

Figure 4: Tested, aborted, proven redundant path delay faults, and CPU time required for TPG of robust tests in the output cones of c6288.

has about $1.98 \cdot 10^{20}$ path delay faults. In 1122 CPU hours it identified 99.4% of all faults either as testable or redundant. The average run time of $2 \cdot 10^{-14}$ seconds per fault indicates that RESIST is very efficient.

References

[1] T. W. Williams, B. Underwood, and M. R. Mercer. The Interdependence Between Delay-Optimization of Synthesized Networks and Testing. *Proc. 28th DAC*, pp. 87 – 92, June 1991.

[2] W. K. Lam, A. Saldanha, R. K. Brayton, and A. L. Sangiovanni-Vincentelli. Delay Fault Coverage and Performance Tradeoffs. *30th DAC*, pp. 446 – 452, June 1993.

[3] S. M. Reddy, C. J. Lin, and S. Patil. An Automatic Test Pattern Generator for the Detection of Path Delay Faults. *IEEE Int. Conf. on CAD*, pp. 284 – 287, Nov. 1987.

[4] C. J. Lin and S. M. Reddy. On Delay Fault Testing in Logic Circuits. *IEEE Trans. on CAD*, Vol. CAD-6, No. 5, pp. 694 – 703, Sept. 1987.

[5] M. H. Schulz, K. Fuchs, and F. Fink. Advanced Automatic Test Pattern Generation Techniques for Path Delay Faults. *19th FTCS*, pp. 44 – 51, June 1989.

[6] K. T. Cheng, S. Devadas, and K. Keutzer. Robust Delay-Fault Test Generation and Synthesis for Testability under a Standard Scan Design Methodology. *Proc. 28th DAC*, pp. 80 – 86, June 1991.

[7] 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. CAD-10, No. 10, pp. 1323 – 1335, Oct. 1991.

[8] K. Fuchs, H. C. Wittmann, and K. J. Antreich. Fast Test Pattern Generation for All Path Delay Faults Considering Various Test Classes. *3rd Europ. Test Conf.*, pp. 89 – 98, April 1993.

[9] I. Pomeranz, S. M. Reddy, and P. Uppaluri. NEST: A Non-Enumerative Test Generation Method for Path Delay Faults in Combinational Circuits. *30th DAC*, pp. 439 – 445, June 1993.

[10] P. C. McGeer et al. Timing Analysis and Delay-Fault Test Generation using Path-Recursive Functions. *IEEE Int. Conf. on CAD*, pp. 180 – 183, Nov. 1991.

[11] D. Bhattacharya, P. Agrawal, and V. D. Agrawal. Delay Fault Test Generation for Scan/Hold Circuits using Boolean Expressions. *Proc. 29th DAC*, pp. 159 – 164, June 1992.

[12] I. Pomeranz and S. M. Reddy. An Efficient Non-Enumerative Method to Estimate Path Delay Fault Coverage. *Proc. IEEE Int. Conf. on CAD*, Nov. 1992.

[13] H. Fujiwara and T. Shimono. On the Acceleration of Test Generation Algorithms. *IEEE Trans. on Comp.*, Vol. C-32, No. 12, pp. 1137 – 1144, Dec. 1983.