

RL-TR-97-230
Final Technical Report
October 1997



DELAY FAULT AND STUCK-AT FAULT TEST GENERATION USING MULTIPROCESSING

Syracuse University

Chien-Hsing Wu and Carlos R.P. Hartmann

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

DTIC QUALITY INSPECTED 4

19980310 145

**Rome Laboratory
Air Force Materiel Command
Rome, New York**

This report has been reviewed by the Rome Laboratory Public Affairs Office (PA) and is releasable to the National Technical Information Service (NTIS). At NTIS it will be releasable to the general public, including foreign nations.

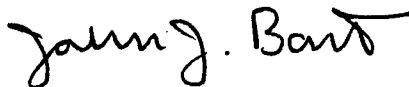
RL-TR-97-230 has been reviewed and is approved for publication.

APPROVED:



WARREN H. DEBANY, JR.
Project Engineer

FOR THE DIRECTOR:



JOHN J. BART, Chief Scientist
Electromagnetics & Reliability Directorate

If your address has changed or if you wish to be removed from the Rome Laboratory mailing list, or if the addressee is no longer employed by your organization, please notify RL/ERDA, 525 Brooks Road, Rome, NY 13441-4505. This will assist us in maintaining a current mailing list.

Do not return copies of this report unless contractual obligations or notices on a specific document require that it be returned.

REPORT DOCUMENTATION PAGE			Form Approved OMB No. 0704-0188	
<small>Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington Headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188), Washington, DC 20503.</small>				
1. AGENCY USE ONLY (Leave blank)		2. REPORT DATE October 1997		3. REPORT TYPE AND DATES COVERED Final Aug 94 - Dec 96
4. TITLE AND SUBTITLE DELAY FAULT AND STUCK-AT FAULT TEST GENERATION USING MULTIPROCESSING			5. FUNDING NUMBERS C - F30602-94-1-0005 PE - 62702F PR - 4600 TA - A0 WU - A4	
6. AUTHOR(S) Chien-Hsing Wu and Carlos R.P. Hartmann				
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Syracuse University School of Computer Information Science Suite 2-120/CST Syracuse NY 13244-4100			8. PERFORMING ORGANIZATION REPORT NUMBER N/A	
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES) Rome Laboratory/ERDA 525 Brooks Road Rome NY 13441-4505			10. SPONSORING/MONITORING AGENCY REPORT NUMBER RL-TR-97-230	
11. SUPPLEMENTARY NOTES Rome Laboratory Project Engineer: Warren H. Debany, Jr./ERDA/(315) 330-2922				
12a. DISTRIBUTION AVAILABILITY STATEMENT Approved for Public Release; Distribution Unlimited.			12b. DISTRIBUTION CODE	
13. ABSTRACT (Maximum 200 words) Digital logic circuits must be tested to assure their correct behavior at the desired clock rate. This report describes an algorithm for generating tests for path delay faults; these faults are models of the faulty switching behavior of digital circuits. The path delay fault test generation system developed here is based on an extension to the SIXteen valued Maximized Propagation Lowered Enumeration (SIMPLE) algorithm, which was originally developed for stuck-at fault test generation. The extension of SIMPLE resulted in a powerful path delay fault test generator with the ability to identify nearly every nonrobustly-detectable fault in a circuit without resorting to the enumeration phase. A parallel implementation of the test generator was developed using the Parallel Virtual Machine (PVM) communication package.				
14. SUBJECT TERMS parallel processing, multiprocessing, delay faults, fault detection, test generation, digital logic circuits			15. NUMBER OF PAGES 64	
			16. PRICE CODE	
17. SECURITY CLASSIFICATION OF REPORT UNCLASSIFIED	18. SECURITY CLASSIFICATION OF THIS PAGE UNCLASSIFIED	19. SECURITY CLASSIFICATION OF ABSTRACT UNCLASSIFIED	20. LIMITATION OF ABSTRACT UL	

Abstract

Ascertaining proper operation of digital circuits requires verification not only of the correct functional operation but of the correct operation at the desired clock rates as well. The classical gate-level fault model has been the *stuck-at fault model* where the effects of physical failures are described by the inputs and outputs of logic gates permanently stuck at logic value 0 or 1. We have implemented an algorithm called SIMPLE (**SIXteen valued, Maximized Propagation Lowered Enumeration approach to test generation**) that generates test patterns for single stuck-at faults in combinational circuits.

Failures causing logic circuits to malfunction at desired clock rates or to not meet timing specifications are called *delay faults*. The well-known *path delay fault model* features the advantageous capability of modeling distributed failures in a circuit, which are typically caused by statistical variations in the manufacturing process. We have also implemented an algorithm (**Path Delay Fault Test Generator**) for generating robust tests for path delay faults which is based on a new 64-valued logic system.

It is well known that test generation processes for stuck-at faults and path delay faults are very time consuming. We have employed the power of parallel processing in this project to speed up the performance of test generation for these two classes of faults. The experimental results presented in this report clearly show the efficiency of our straightforward and yet effective parallelization schemes for both of our algorithms.

The experiments for the test generation for path delay faults also show an interesting and surprising fact that almost all the faults in the fault set of nearly every target combinational circuit are not robustly testable and each of these faults can be identified within a very short time. This phenomenon suggests that a lot of circuits are not designed with features to enhance the robust testability for the path delay fault model. Design methodologies that increase the robust testability of at least the timing-critical functional paths must be applied for the automatic test generators to be applicable to the problem of testing the robust path delay faults.

Table of Contents

1.

1. Introduction.....	1
2. Stuck-at Fault Test Generation.....	2
2.1 SIMPLE.....	3
2.1.1 Pre-Processing Phase.....	5
2.1.2 Propagation Phase.....	11
2.1.3 Forward and Backward Implications.....	14
2.1.4 Enumeration Phase.....	18
2.2 Parallelization of SIMPLE.....	20
2.3 Experimental Results for SIMPLE.....	21
3. Delay Fault Test Generation.....	21
3.1 Hardware Model and Robust Tests for Path Delay Faults.....	23
3.2 Logic System and Requirements for Robust Tests.....	25
3.3 Forward and Backward Implication Procedures.....	27
3.4 Derivation of Static Learning Table.....	29
3.5 Algorithm Outline and Test Compaction.....	30
3.6 Experimental Results for the Sequential PDFTG.....	32
3.7 Parallelization of PDFTG.....	33
4. Discussion.....	37
Reference.....	38

List of Tables

Stuck-at Fault Test Generation

Table 1: Forward implication tables for AND and XOR gates.....	4
Table 2: Forward implication table for NOT gate.....	4
Table 3: Contrapositive implications at net m_1	9
Table 4: (L_2, G) combinations that yield useful contrapositive assertions	10
Table 5: Backward implication table for AND gate ⁰	17
Table 6: Rules to calculate the controllability in SCOAP.....	19
Table 7: Required values at the off-path sensitizing inputs.....	27
Table 8: Forward implication tables for AND and XOR gate	28
Table 9: Forward implication table for NOT gate.....	28
Table 10: Backward implication table for AND gate.....	28
Table 11: Backward implication table for XOR gate	28
Table 12: Correspondence of the elements in the basic set and the 3-bit sequence.....	29
Table 13: Contrapositive implications at net N_1	30
Table 14: Results from Fault List Generation.....	32
Table 15: Results from Test Generation with Compaction.....	33
Table 16: Results from Test Generation without Compaction ⁰	34
Table 17: Timing results for parallel version of PDFTG ⁰	35

List of Figures

Stuck-at Fault Test Generation

<i>Figure 1: Contrapositive Implication</i>	<i>7</i>
<i>Figure 2 Gate decomposition</i>	<i>16</i>
<i>Figure 3. Timing results for c432.....</i>	<i>21</i>
<i>Figure 4. Timing results for c6288.....</i>	<i>21</i>

Delay Fault Test Generation

<i>Figure 5. Hardware Model for Delay Fault Testing</i>	<i>24</i>
<i>Figure 6. Timing results for c432.....</i>	<i>35</i>
<i>Figure 7. Timing results for c499.....</i>	<i>35</i>
<i>Figure 8. Timing results for c880.....</i>	<i>36</i>
<i>Figure 9. Timing results for c1355.....</i>	<i>36</i>
<i>Figure 10. Timing results for c1908.....</i>	<i>36</i>
<i>Figure 11. Timing results for c2670.....</i>	<i>36</i>
<i>Figure 12. Timing results for c3540.....</i>	<i>36</i>
<i>Figure 13. Timing results for c5315.....</i>	<i>36</i>
<i>Figure 14. Timing results for c6288.....</i>	<i>36</i>
<i>Figure 15. Timing results for c7552.....</i>	<i>36</i>

1. Introduction

The problem of testing circuits is gaining more and more importance as rapid strides are being made in VLSI technology. Testing is useful both before and after fabrication of circuits. Testing before fabrication ensures that a circuit design meets the intended specifications and is free of functional or logic errors. The more crucial goal of testing is to detect faulty devices after fabrication. Increasing circuit complexity has an adverse effect on testing by increasing testing time (and hence cost), test pattern generation and evaluation time and, of course, the sheer amount of data that has to be handled.

Testing consists of applying a series of input patterns to a circuit and observing its response. This is then compared with the expected response of the circuit to verify correct operation. If there is a discrepancy, an *error* is said to have occurred and its physical cause is denoted as a *fault*. Thus test generation is closely related to fault modeling - the mapping of physical defects to errors. Faults affecting the logic function of combinational circuits are called *functional faults*. The classical gate-level fault model has been the stuck-at fault model, where the effects of faults are described by the inputs and outputs of logic gates permanently stuck at 0 or 1. It is well known that much of the work in the testing field has been in terms of this model. Faults causing logic circuits to malfunction at desired clock rates, or not meeting timing specifications are called *delay faults*. Delay fault testing has been gaining considerable importance with the increased susceptibility to manufacturing defects that increase circuit delays.

We have developed an algorithm called SIMPLE (Sixteen valued, Maximized Propagation Lowered Enumeration approach to test generation) for detecting single stuck-at faults in combinational circuits [1, 2]. This algorithm is based on a 16-valued logic system and introduces some novel approaches to making test pattern generation more efficient.

We have also developed an algorithm (Path Delay Fault Test Generator) [3] for generating robust test for path delay faults which is based on a new 64-valued logic system. This

logic system is obtained by extending the 16-valued logic system to consider all possible stable and hazardous values that can occur at a net in the context of two-pattern testing. Inherent in this scheme is a test compaction procedure which exploits the availability of choices in the values of certain nets in order to construct a two-pattern test that will robustly test other functional paths along with the target path. We have shown that when generating a robust test for a path delay fault, we also generate, without any additional computation, a test for detecting single stuck-at faults at nets along this path.

It is well known that test generation processes for stuck-at faults and delay faults are very time consuming. We have employed the power of parallel processing in this project to speed up the performance of test generation. The communication package that we chose for the parallel implementations of the test generation algorithms is PVM [34], the Parallel Virtual Machine, which is a software system that permits a network of heterogeneous Unix computers to be used as a single large parallel computer. This heterogeneous environment provided by PVM permits us to run our parallel versions of the test generation algorithms on machines ranging from SUN workstations to CM5 and CRAY machines.

This report is organized as follows. Section 2 discusses our stuck-at fault test generation algorithm, SIMPLE, and the experimental results. Section 3 summarizes the path delay fault test generation algorithm, PDFTG, and the experimental results. The conclusion of the project is presented in Section 4.

2. Stuck-at Fault Test Generation

The generation of test patterns for combinational circuits has been long recognized by researchers as a well-defined mathematical problem that belongs to the class of NP-complete problems [12, 15]. Several Automatic Test Pattern Generation (ATPG) algorithms for detecting stuck-at faults in combinational circuits exist in the literature [9,

11, 13, 16, 18, 19, 20, 21, 22]. SIMPLE, an ATPG algorithm based on a 16-valued logic system, is proposed in [1]. This algorithm introduces some novel approaches to making test generation more efficient.

Two prototype implementations of SIMPLE were developed in C: one is the sequential version and the other the parallel version using the PVM software package. In section 2.1 we summarize the principles behind SIMPLE. The strategy used in the implementation of the parallel version of SIMPLE is described in section 2.2. Section 2.3 summarizes the simulation results, which confirm the findings shown in [4].

2.1 SIMPLE

In this section we give a concise description of the SIMPLE [1] algorithm for detecting single stuck-at faults in combinational circuits that contain NOT, AND, NAND, OR, NOR, XOR, and XNOR gates.

Testing generation involves considering the value of a net in the good and the faulty circuit. This can be done by representing the value of a net as an ordered pair (b_g, b_f) where b_g (b_f) is the value of the net in the good (faulty) circuit [17]. Thus the value of a net is one of the elements of the set $U = \{(0, 0), (0, 1), (1, 0), (1, 1)\}$. In the process of generating tests, it might not be possible to uniquely specify the value of a net as one of the elements of U . However, we may already know that a net cannot assume one or more of these values. We incorporate this information by defining the value of a net as one of the 16 subsets of U . We denote 16 sets as $\emptyset, 0, 1, D, \bar{D}, 0/1, 0/D, 1/D, 0/\bar{D}, 1/\bar{D}, D/\bar{D}, 0/1/D, 0/1/\bar{D}, 0/D/\bar{D}, 1/D/\bar{D}$, and $0/1/D/\bar{D}$ where $0 = \{(0, 0)\}$, $1 = \{(1, 1)\}$, $D = \{(1, 0)\}$, $\bar{D} = \{(0, 1)\}$, and “/” denotes set union operations. Note that $U = 0/1/D/\bar{D}$. The value \emptyset needs to be included to reflect the situation when two or more constraints require disjoint values on a net. These 16 values are equivalent to the elements of the logic system developed by Akers [5] to provide a tool for test generation. Tables 1 and 2 represent the

AND, XOR, and NOT functions in our 16-valued system for the values $0, 1, D, \bar{D}$. The complete table for all the 15 non- \emptyset values can be easily constructed from the given tables by using the set union operation. The tables for all the other functions can be obtained from the three tables. Note that any logic function with \emptyset as one of its arguments will yield \emptyset as a result.

Table 1: Forward implication tables for AND and XOR gates

AND	0	1	D	\bar{D}	XOR	0	1	D	\bar{D}
0	0	0	0	0	0	0	1	D	\bar{D}
1	0	1	D	\bar{D}	1	1	0	\bar{D}	D
D	0	D	D	0	D	D	\bar{D}	0	1
\bar{D}	0	\bar{D}	0	\bar{D}	\bar{D}	\bar{D}	D	1	0

Table 2: Forward implication table for NOT gate

NOT	0	1	D	\bar{D}
	1	0	\bar{D}	D

Using this notation we define a sensitized net as one whose value is either D, \bar{D} , or D/\bar{D} . Furthermore, if all the nets along a path in the circuit are sensitized, then the path is said to be sensitized. This 16-valued system exploits the linearity of XOR/XNOR gates during test generation. It also allows us to characterize all restrictions that are imposed by a fault as well as the particular circuit path chosen in order to propagate its effect. There are three distinct phases in the SIMPLE algorithm:

(i) Pre-processing phase: In this phase we construct a set of trees based on the interdependence of circuit nets. Among other things, this forest will be used to easily identify which circuit nets *must* be sensitized by any test. The experiments for collecting useful contrapositive assertions are also conducted at the fanout stem (FOS) nets of the circuit in this phase. The contrapositive assertions can later be applied to determine net

values that would otherwise not be able to be obtained by only the forward and backward implication procedures.

(ii) Propagation phase: In this phase we deliberately sensitize a single path from the fault site to a PO and find all the resulting deterministic implications, namely, the forward, backward, and contrapositive implications. In this process other paths may also be sensitized. Path selection is the only choice made in this phase -- implications are based on all the constraints that *must* be satisfied in order to sensitize the chosen path. This is possible because of the completeness of the 16-valued system and the use of deterministic implication rules.

(iii) Enumeration phase: In general, the test cube constructed by the propagation phase will not yield a test -- particularly because no arbitrary choices were made other than the selection of the sensitized path. Thus there may be gates whose input net values contain combinations capable of desensitizing the chosen path. In this phase we use an enumeration procedure to select values for the primary inputs (PIs) so that such combinations can never occur.

2.1.1 Pre-Processing Phase

Construction of Dominator Forest

The importance of identifying nets that *must* be sensitized for a fault to be detected was first highlighted by Akers [5] and later by Fujiwara and Shimono [11]. As pointed out in TOPS [16], the concept of graph dominators [23] can be used to identify the nets which must be sensitized to detect a fault. In the context of test generation, we term the set of dominators of a net m as the set of all nets in the circuits which lie on every path from net m to any PO. By definition, net m is a dominator of itself; however, for ease of notation we define $D(m)$ as the set of all dominators of m except m itself. To account for multiple-output circuits, the concept of a dominator tree can be extended to that of a forest. We present here a procedure to construct this forest for a given circuit.

We construct a set of trees such that every signal line of the circuit corresponds to a node in one of the trees in the forest. We start by creating as many trees as there are primary outputs (POs), such that each PO corresponds to a root of a tree. However, new trees may be created during the procedure. Thereafter, each node which has not been marked as a leaf is inspected and the tree construction is continued as follows:

(i) If the node m_i being considered corresponds to the output line of a logic gate G_i in the circuit, then every input line of G_i becomes a child of this node m_i . If the input line is a PI, then it is marked as a PI leaf. If the input line is a FOB, then it is marked as a FOB leaf.

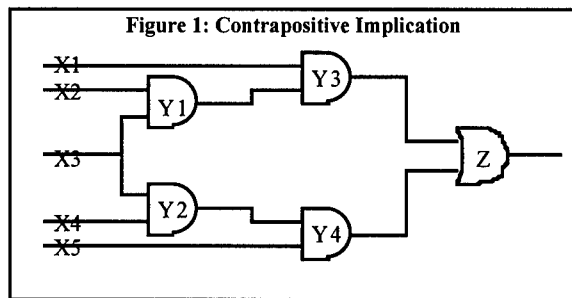
(ii) If the node m_i being inspected is a FOS, then wait until all the fanout branches (FOBs) of this FOS have been marked as FOB leaves. Find the first common ancestor of all these FOB leaves by traversing the tree(s) from these leaves to the root(s) of the tree(s). The necessary and sufficient condition for these FOB leaves to have a common ancestor is that they belong to the same tree. If such an ancestor exists, then mark m_i a child of this ancestor node. If it does not, then start a new tree with m_i as a root. In either case, mark m_i as an FOS node -- if it is also a PI, then it must be marked as a PI leaf also.

The above procedure is continued until every line of the circuit becomes a node in some tree of the forest.

The root of any tree in the constructed forest is either a PO or a FOS. If any tree has a single node, then this node must correspond to a PI which is also a FOS. The set $D(m)$ contains all the nodes encountered when traversing the tree (containing node m) from m to the root.

Collecting Contrapositive Logical Assertions

In this section we give a summary of how using the contrapositive assertions of implications performed during the Pre-processing Phase can be used as an effective speed-up technique. The use of contrapositive logical identity to reduce the search space was first suggested by Schulz, et. al., in SOCRATES [21].



The contrapositive of the logic expression $P \Rightarrow Q$ is the equivalent expression $\sim Q \Rightarrow \sim P$. Referring to the circuit of Fig. 1 we notice that $X_3 = 0 \Rightarrow Z = 0$. Hence the contrapositive would yield $Z = 1 \Rightarrow X_3 = 1$. However, if we require the value 1 at Z given that all other nets have the value 0/1, no deterministic change would be implied by the backward implication procedure alone. Note that in some cases a backward implication will yield the information provided by the contrapositive implication. For example, $X_3 = 0 \Rightarrow Y_4 = 0$ yields $Y_4 = 1 \Rightarrow X_3 = 1$. However, a backward implication of $Y_4 = 1$ yields a 1 at X_3 , X_4 , and X_5 . Hence it is useful to identify the conditions under which a backward implication cannot yield the information provided by a contrapositive assertion. In such cases we may store this information for possible use later in the test generation process.

The procedure presented in SOCRATES can only be used to backward imply the value 0 or 1 because it is assumed in [21, Section 4.3] that the “injected target fault is not located in this part of the circuit and the effects of the target fault can not propagate to it as well.” Furthermore, as mentioned in the learning procedure of [21, Fig. 5], the 0 and 1 implications are performed for all the nets of the circuits. SIMPLE employs an improved learning procedure that performs the 0 and 1 implications for only the FOS nets of the

circuits, and thus the number of implications performed and the number of assertions stored for future use are fewer than those of SOCRATES. The information obtained by our procedure is sufficient to generate the information that can be obtained by performing the **0** and **1** implications for the remaining nets because of the deterministic nature of our backward implication procedure. Furthermore, these stored **0** and **1** implications from FOS nets are sufficient to generate the useful contrapositive assertions for all 16 values of our logic system and for all nets of the circuit as long as the logical behavior of the corresponding portion of the faulty circuit is identical to that of the good one. Thus our use of contrapositive assertions will not be limited, as SOCRATES is, only to nets that are unaffected by the fault.

In our 16-valued system, assume that the forward implication of a value L_1 at net m_1 with **0/1/D/ \bar{D}** at all other nets yields the value L_2 at net m_2 . Thus when we require a value $L'_2 \subseteq (\mathbf{0/1/D/\bar{D}} - L_2)$ at net m_2 , then the value of net m_1 cannot contain any element of L_1 . To obtain the implication for all possible values of L_1 , we need only to perform implications for each individual element of **0/1/D/ \bar{D}** . Thus the procedure to obtain the implications for the 16-valued system, henceforth referred to as **16-VP**, would be to set the value of net m_1 to each of the values **0**, **1**, **D** and **\bar{D}** , one at a time and with **0/1/D/ \bar{D}** at all other nets, and observe the implied value at net m_2 . Each such implication is referred to as a **16-VP** “experiment.” We will show that the information yielded by **16-VP** can be obtained from a simpler procedure that utilizes a 3-valued (**0**, **1**, **0/1**) logic system. In this procedure, which we denote as **3-VP** “experiment,” we set the value of an FOS net m_1 to each of the values **0** and **1** one at a time and with **0/1** at all other nets, and observe the implied value at net m_2 .

We now illustrate by an example that the results obtained from the **16-VP** experiments on a FOS net m_1 can be deduced from those of the corresponding **3-VP** experiments on the same FOS net. Consider the situation where a **0** and a **1** at net m_1 yields a **0** and **0/1**, respectively, at net m_2 , the direct implication yielded from this experiment and the contrapositive of this implication are shown below.

$$m_1 = 0 \rightarrow m_2 = 0 \text{ ----- (1)}$$

$$m_2 = 1 \rightarrow m_1 = 1 \text{ ----- (2), contrapositive of direct implication (1) from 3-VP experiment}$$

Now if we are required to set net m_2 to $\mathbf{D} = (1, 0)$ during test generation, by the application of contrapositive assertion (2) to each value of net m_2 under the good and faulty circuit, we know that net m_1 must be set to either $(1, 0)$ or $(1, 1)$, which is equivalent to the set $\mathbf{1/D}$ in our 16-valued logic system. As another example, if the required value at net m_2 is $\overline{\mathbf{D}} = (0, 1)$ during test generation, then net m_1 can only contain elements in the set $\mathbf{1/\overline{D}}$, or equivalently $\{(1, 1), (0, 1)\}$. The following table summarizes the implied values at net m_1 for each different direct implication from the **3-VP** experiments and each singleton value as a requirement at net m_2 .

Table 3: Contrapositive implications at net m_1

Basic value at net m_2	$m_1 = 0 \rightarrow m_2 = 0$	$m_1 = 0 \rightarrow m_2 = 1$	$m_1 = 1 \rightarrow m_2 = 0$	$m_1 = 1 \rightarrow m_2 = 1$
0	$0/1/D/\overline{D}$	1	$0/1/D/\overline{D}$	0
1	1	$0/1/D/\overline{D}$	0	$0/1/D/\overline{D}$
D	$1/D$	$1/\overline{D}$	$0/\overline{D}$	$0/D$
\overline{D}	$1/\overline{D}$	$1/D$	$0/D$	$0/\overline{D}$

The above table is derived under the assumption that the results of the **3-VP** experiments conducted for the faulty circuit are the same as those of the good one. Therefore, we should avoid applying the contrapositive rules that are derived from **3-VP** experiments whose results will be affected by the presence of the current target stuck-at fault.

We now discuss the condition under which the information yielded by a contrapositive assertion cannot be obtained by a deterministic backward implication alone and hence should be stored for future use. Consider the situation where a singleton value L_1 at net m_1 yields, using **3-VP**, a singleton value L_2 at the output net m_2 of a gate G . The corresponding contrapositive assertion should be stored if and only if the value L_2 can be obtained at the output of G by setting all its input to non-controlling values. Consequently, Table 4 shows the L_2 and G combinations for which this implication

should be stored for future use. In general, for the cases that satisfy the (L_2, G) combinations given in Table 4, we will not be able to drop L_1 from the set of all possible values at net m_1 when we require a value $L'_2 \subseteq (0/1/D/\overline{D} - L_2)$ at net m_2 by using only the forward and backward implication procedures.

Table 4: (L_2, G) combinations that yield useful contrapositive assertions

L_2	Gate type			
0	OR	NAND	XOR	XNOR
1	NOR	AND	XOR	XNOR

Selection of *pdcf*

The selection of the primitive D-cube of failure (*pdcf*) in DALG [19] may involve arbitrary choices which can result in mistaken decisions causing costly backtracking. We avoid this problem by introducing a fictitious gate G_f at the site of the fault. If the fault is at net n , then we connect net n_f to all signal lines which were previously connected to net n . If the fault site is a FOB which is identified by net n and net n_f , then the G_f is inserted in this FOB only. Accordingly, the unique *pdcf* depends only on the kind of stuck-at fault.

	n	n_f
fault site <i>s-a-0</i>	1	D
fault site <i>s-a-1</i>	0	\overline{D}

Token Assignment

The goal of this step is to identify which circuit nets cannot be affected by the fault. In order to convey this information, we associate with every net a Boolean token. This token is TRUE if and only if there exists a path from n_f to any PO which passes through this net. These tokens can be computed by a single forward pass through the circuit.

2.1.2 Propagation Phase

In this phase we sensitize a single path from net n_f to a PO, however, other paths may also get sensitized. In a manner analogous to DALG [19] we use test cubes whose entries reflect the current values of all nets during any stage of test generation. The entries of any test cube, tc_k , are elements of our 16-valued system.

We initialize this phase by constructing tc_1 in the following manner:

1. set nets n and n_f to the values specified by the *pcdf*.
2. Assign D/\overline{D} to all nets belonging to the set $D(n)$.
3. Set all nets with FALSE token, except net n , to $0/1$.
4. Assign $0/1/D/\overline{D}$ to all unassigned nets of the test cube.

For each test cube tc_k generated at any stage of our algorithm, we find its corresponding *deterministic* test cube, $d(tc_k)$. We define a $d(tc_k)$ as one in which no entry can be changed without making an arbitrary choice for one or more net values. That is, all unique implications of the net values must be considered. The derivation of $d(tc_k)$ thus involves the application of the forward, backward, and contrapositive implications to the current state, represented by tc_k , of the circuit under test generation. Rules for the contrapositive implication are summarized in the previous section for the Pre-Processing Phase. Rules for forward and backward implication procedures are given in the following section after we present the overall procedure for this phase.

If $d(tc_1)$ cannot be constructed because contradictions were encountered, then there exists no test for the fault. Otherwise we have a sensitized path from n_f to all the FOB nets of the first FOS node (which could be n itself!) encountered in traversing the appropriate tree of the dominator forest from n to the root. If there is no FOS encountered, then we have a sensitized path from n to the PO corresponding to the root of the tree.

At this point we have to select one of the FOB nets, say the FOB net from net m_1 to net m_2 (denoted as $m_1 \rightarrow m_2$), to extend the sensitized path. In this implementation of SIMPLE, the selection of FOB nets is guided by the observability measure introduced in COP [6], which is summarized in the following subsection. To obtain tc_2 we should sensitize all nets belonging to the set $D(m_1 \rightarrow m_2) - D(n)$ by intersecting their values in $d(tc_1)$ with D/\overline{D} . If any empty intersection results, then the sensitized path cannot be extended through net m_2 and alternative paths should be investigated. Note that this step implicitly performs the equivalence of the X-path check [13] while setting up the gate outputs that should be sensitized. As stated earlier, we should then construct $d(tc_2)$. If contradictions occur while constructing $d(tc_2)$, then an alternate path must be selected. Otherwise we have a sensitized path from n_f at least to the FOB nets corresponding to the next FOS net or some PO. The process of extending the sensitized path by selecting a FOB net, constructing a tc_k and its corresponding $d(tc_k)$ continues until the algorithm establishes a sensitized path p_i from the fault site to any PO in a deterministic test cube $d(tc_j)$. This test cube, $d(tc_j)$, is denoted as $T_f(p_i)$. If contradictions occur during this path extending process, then alternate paths should be investigated. If all possible paths result in contradictions, then no test exists. Note that all possible single paths need not be explicitly investigated to arrive at this conclusion.

$T_f(p_i)$ represents all the constraints that must be imposed to sensitize path p_i . Since the backward implication rule does not make any arbitrary choices, there may be gates where the output value is a proper subset of the value implied by the input values, i.e., the input values include combination(s) that will desensitize path p_i . We define the output nets of such gates as *variant* nets. If a net is not a variant it is defined to be *invariant*. If there are no variant nets in $T_f(p_i)$, then we have already obtained a test for the fault. Otherwise the enumeration phase must be invoked to determine a test by converting all the variant nets in $T_f(p_i)$ to invariant ones.

Observability

To define the observability measure introduced in COP [6], we first need to define the controllability measure that it is based on. Both measures are based on a probabilistic approach that uses the simplifying assumption that logic signal probabilities are statistically independent.

For every PI, we set $C^0(PI) = C^1(PI) = 0.5$. Also, for any net m , $C^0(m) = 1 - C^1(m)$. Let G be a gate with input nets i_1, i_2, \dots, i_n , and the output net m . To express $C^0(m)$ in terms of $C^0(i_j)$ and $C^1(i_j)$, for $j \in \{1, 2, \dots, n\}$, we first define N^0 as the set of logic patterns that, when applied to the inputs of G , set net m to the logic value 0. For $\alpha = (\alpha_1, \alpha_2, \dots, \alpha_n) \in N^0$ define p_j , $1 \leq j \leq n$ as follows:

$$p_j = \begin{cases} C^0(i_j), & \text{if } \alpha_j = 0 \\ C^1(i_j), & \text{if } \alpha_j = 1 \end{cases} \quad \text{----- (3)}$$

$C^0(m)$ can now be defined in terms of p_1, p_2, \dots, p_n as follows:

$$C^0(m) = \sum_{\alpha \in N^0} \prod_{j=1}^n p_j \quad \text{----- (4)}$$

If net m_l is a fanout branch whose corresponding stem is net m , then $C^0(m_l) = C^0(m)$ and $C^1(m_l) = C^1(m)$.

Now we are in the position of defining $OB(m)$, the observability measure of net m . For every PO we define $OB(PO) = 1$. Now consider gate G . Let S_j be the set of logic patterns that, when applied to the input $i_1, i_2, \dots, i_{j-1}, i_{j+1}, \dots, i_n$, sensitize the net m to a change in the input i_j . Then

$$OB(i_j) = OB(m) \times \sum_{B \in S_j} \prod_{\substack{l=1 \\ l \neq j}}^n p_l \quad \text{----- (5)}$$

Finally, if net m_1, m_2, \dots, m_r are fanout branches of a fanout stem m , then

$$OB(m) = 1 - \prod_{l=1}^r (1 - OB(m_l)) \quad \text{----- (6)}$$

For any two nets m_1 and m_2 , if $OB(m_1) > OB(m_2)$, then net m_1 is “easier” to observe than net m_2 . Thus, this measure of observability tends to increase with the *ease of observing* a net.

2.1.3 Forward and Backward Implications

In a $d(tc_k)$ all deterministic implications (i.e. making no arbitrary choices) of all entries of the test cube tc_k are fully considered.

To construct $d(tc_1)$ from tc_1 we perform, in addition to contrapositive implications, backward and forward implications of all nets whose values in tc_1 are different from 0/1 and 0/1/D/ \bar{D} and all other nets whose values change during this implication process. In the general case, when we are constructing $d(tc_k)$ from tc_k , we start by considering the forward and backward implications of the nets whose values in tc_k are different from those in the last successfully constructed deterministic test cube. During the construction of $d(tc_k)$ from tc_k , if a backward or forward implication request results in a new value L'_j for any net m_j of the circuit, then we should update the corresponding net entry L_j by setting it to $L_j \cap L'_j$. If this intersection yields the empty set, then $d(tc_k)$ cannot be constructed.

In order to obtain $d(tc_k)$ the process of forward and backward implications continues until no more changes occur in the values associated with any net. Note that this process is guaranteed to terminate in a finite number of steps because we are performing set intersections on finite sets.

The rules for constructing deterministic test cubes must include the provision for appropriately handling the values of nets associated with fanout points. We now present the rules for forward and backward implication.

Forward Implication

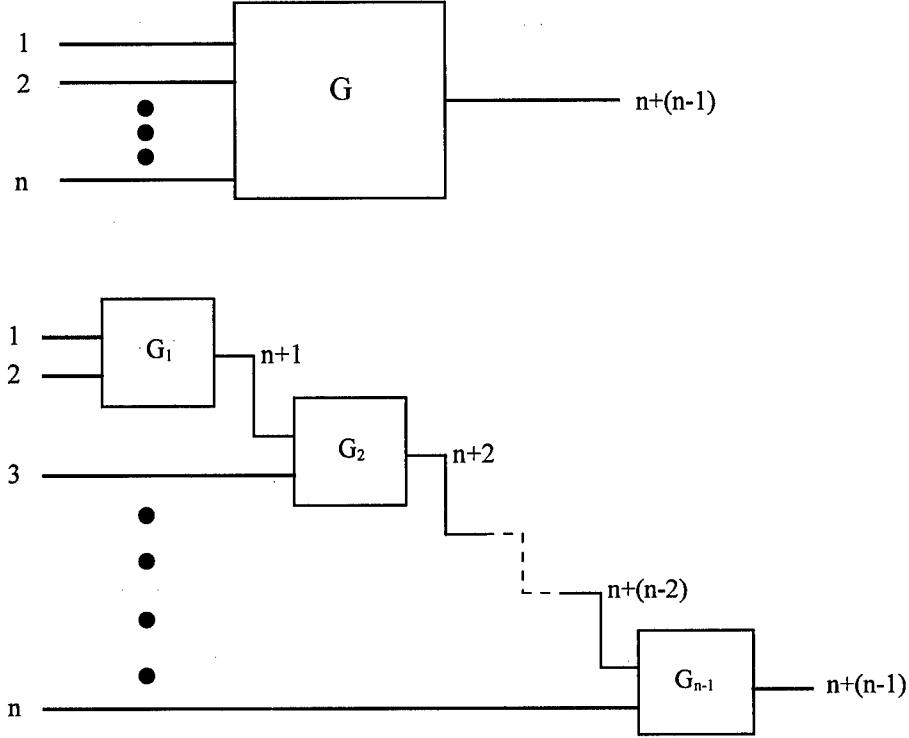
The process of forward implication of the values associated with every net is done with the help of Tables 1 and 2. These tables are generalizations of the truth tables of the

respective gates. Note that these tables are sufficient because OR, NOR, NAND, and XNOR functions can be derived by appropriately using these tables. For gates with more than two inputs, the method adopted in SIMPLE is similar to that used by Akers [5]. We view every gate as being constructed out of two-input gates and use the existing values of a gate to generate a new value for the output. An n -input ($n > 2$) gate is decomposed into a cascade of $n-1$ two-input gates, as shown in Fig. 2. If the n -input gate is a NAND (NOR) gates, then G_1, G_2, \dots, G_{n-2} are AND (OR) gates and G_{n-1} (which sources the output) is a NAND (NOR) gate. This decomposition is performed only for the propagation of logic values; faults are considered only on the $n+1$ signal lines with the original n -input gate.

Suppose we are performing forward implications due to changes(s) in input(s) of a gate G whose output is net m . Let L_O be the “old” set of values associated with net m in the test cube prior to forward implication being performed. Let L_N be the “new” value obtained at net m by using the new values of the inputs of G . Net m is then set to $L_O \cap L_N$ unless $L_O \cap L_N = \emptyset$, which would imply a contradiction. Four other situations are possible:

1. $L_O = L_N$. No further action is needed for this forward implication.
2. $L_O \subset L_N$ (proper subset). We now have to consider the forward implication of the value of L_N at net m on all gates driven by G .
3. $L_O \supset L_N$. We now have to perform a backward implication of the value L_O at net m . This may result in further changes in the inputs of gate G .
4. Otherwise. Both forward and backward implications of the value $L_O \cap L_N$ at net m should be performed.

Figure 2 Gate decomposition



Backward Implication

The process of backward implication involves determining the changes required at the inputs of a gate to satisfy a requested change at the output. A change in the value of a net means that one or more of the possible values associated with the net has been deleted. Consequently, an input change can be made only if the deleted value can never be used with the existing values at the other inputs to generate any of the requested output value(s).

A general set of backward implication rules can be derived in terms of the input values and the requested output value. However, in a manner similar to that presented in [5] we consider each multiple-input gate as a cascade to two-input gates. The backward implication rules for a two-input AND gate is shown in Table 5.

Table 5: Backward implication table for AND gate¹

AND	0	1	D	\bar{D}
0	0/1/D/ \bar{D}	\emptyset	\emptyset	\emptyset
1	0	1	D	\bar{D}
D	0/ \bar{D}	\emptyset	1/D	\emptyset
\bar{D}	0/D	\emptyset	\emptyset	1/ \bar{D}

¹ row header: existing value at one input, column header: requested value at the output

Note that the element \emptyset has been included in this table to detect an unsatisfiable backward implication request. The complete table for all 15 non- \emptyset values is obtained by the set union operation. The resulting table is equivalent to that proposed by Akers [5]. To perform backward implication for a two-input AND gate, we reference the table using the requested value at the output and the existing value at one input to generate the value of the other input. Since the XOR gate is linear, its forward implication table in Table 1 can be used for backward implication also. Regardless of the type of gate in question, the value generated by the appropriate table must be intersected with the existing value of the input. Analogously, the new value of the input and the requested value of the output must now be used to generate the new value of the other input. For example, consider a two-input gate whose input values are L_1 and L_2 . If the requested value of the output of the gate is L_G , then we use L_G and L_1 to determine the new value L'_2 of the second input and then L'_2 and L_G to determine the new value L'_1 of the first input.

As stated before, any gate with more than two inputs is represented as a cascade of two-input gates. Consider an n -input gate G represented as a cascade of $n-1$ two-input gates G_1, G_2, \dots, G_{n-2} and G_{n-1} , with net numbers as shown in Fig. 2. Assume that the values at nets $1, 2, \dots, n$ are X_1, X_2, \dots, X_n respectively. We first use forward implication of these values to compute Y_1, Y_2, \dots, Y_{n-2} , the values of nets $n+1, n+2, \dots, n+(n-2)$ respectively. Then using the value Z , which is the required value at the output of gate G , we apply the backward implication rules for gate G_{n-1} to obtain Z_{n-2} and X'_n , the new values of nets $n+(n-2)$ and n respectively. Having done that, we proceed backwards and apply the backward implication rules for all the gates, one at a time, ending with gate G_1 . Since the

binary operation represented by any non-inverting logic gate is associative, the order in which the input X_i are cascaded is irrelevant.

It is shown in [1] that the above procedure will stabilize in a single pass, unlike the approach followed in [5] which may require several passes.

2.1.4 Enumeration Phase

The goal of this phase is to obtain a test by specifying the unassigned PIs in $T_f(p_i)$ such that all nets are invariant and have values that are subset of their corresponding values in $T_f(p_i)$.

We choose an unassigned PI I_1 in $T_f(p_i)$ and assign a logic value (0 or 1) to it, thereby creating a new test cube which we denote by $tc_f(p_i, 1)$. Now we find its corresponding deterministic test cube $d(tc_f(p_i, 1))$ and update its list of variant nets (note that new variant nets may be created). However, if $d(tc_f(p_i, 1))$ cannot be obtained due to some contradiction, then we complement the entry for I_1 in $tc_f(p_i, 1)$ and construct its corresponding $d(tc_f(p_i, 1))$. If this also leads to a contradiction, then there exists no test corresponding to $T_f(p_i)$. If we are successful in constructing $d(tc_f(p_i, 1))$, we now assign a logic value to some other unassigned PI I_2 , thereby creating $tc_f(p_i, 2)$. As before, we must construct $d(tc_f(p_i, 2))$ and update its list of variant nets. This procedure is continued and we traverse the decision tree, in a manner analogous to PODEM [13], until one of the following two conditions occur:

- The list of variant nets corresponding to some $d(tc_f(p_i, j))$ becomes empty. This indicates the values of the PIs in $d(tc_f(p_i, j))$ represent test(s) for the fault.
- The decision tree is exhausted, i.e., no test exist.

In this implementation of SIMPLE, the selection of PIs is performed by a backtrace procedure that is guided based on the controllability measure proposed in SCOAP [14]. A

short description of how to calculate this measure is given in the following subsection. The description of the backtrace procedure is taken from [10]. During the backtrace procedure, objectives are successfully transferred from gate outputs to gate inputs until a PI is reached. This transfer of objectives is performed using the “easy/hard” heuristic described as follows. When the current objective is to set the output of a gate to a logic value that can be achieved by setting one of its inputs to a *controlling* value (0 for OR/NOR, 1 for AND/NAND), an input which is identified as the “easiest” to control (according to the measure being used) is chosen. On the contrary, if such objective can only be achieved by setting all the inputs of the gate to a *non-controlling* value (0 for OR/NOR, 1 for AND/NAND), then an input which is identified as the “hardest” to control is chosen. This is done so that an early determination of the inability to satisfy an objective will save the time that would be wasted in attempting to set the remaining inputs of the gate. If the current objective is the output of an XOR/XNOR gate, an input which is “easiest” to control is selected.

Controllability

SCOAP associates with every net m two integers denoted by $C^0(m)$ (0-controllability) and $C^1(m)$ (1-controllability). For every PI, we set $C^0(PI) = C^1(PI) = 1$. Now let G be a gate with n input nets i_1, i_2, \dots, i_n , and the output net m . Table 6 shows how to calculate $C^0(m)$ and $C^1(m)$ as a function of the 0-controllabilities and 1-controllabilities of these n inputs.

Table 6: Rules to calculate the controllability in SCOAP

Gate type	$C^0(m)$	$C^1(m)$
AND	$1 + \min_{j \in \{1, 2, \dots, n\}} \{C^0(i_j)\}$	$1 + \sum_{j=1}^n C^1(i_j)$
OR	$1 + \sum_{j=1}^n C^0(i_j)$	$1 + \min_{j \in \{1, 2, \dots, n\}} \{C^1(i_j)\}$
XOR	$1 + \min\{C^0(i_1) + C^0(i_2), C^1(i_1) + C^1(i_2)\}$	$1 + \min\{C^0(i_1) + C^1(i_2), C^1(i_1) + C^0(i_2)\}$

^{||} Only for 2-input XOR gate

Finally, if net m_1 is a fanout branch whose corresponding stem is net m , then $C^0(m_1) = C^0(m)$ and $C^1(m_1) = C^1(m)$.

For any two nets m_1 and m_2 , if $C^0(m_1) < C^0(m_2)$ ($C^1(m_1) < C^1(m_2)$), then we say that m_1 is “easier” to control than net m_2 with respect to logic value 0(1). Thus, this measure of controllability increases with the *difficulty of controlling* a net.

2.2 Parallelization of SIMPLE

In this section we describe the approach used to parallelize our sequential implementation of SIMPLE.

Simulation results indicated that more than 95% of the running time of our sequential implementation of SIMPLE was spent in the enumeration phase. Thus we parallelized only the enumeration phase of our algorithm.

Assume that there are $n = 2^k$ processors available in the computing environment during the execution of the parallel version of SIMPLE. As described in Section 2.1.4 the enumeration procedure searches the input pattern space by assigning logic values to the undetermined PIs one at a time. We can parallelize this search procedure by partitioning the input pattern space into 2^k subspaces with equal number of patterns and assigning each subspace to a processor. The partitioning can be done by selecting k undetermined PIs, and each subspace is identified by a unique input combination on these k PIs. After partitioning the pattern space, all the 2^k processors can simultaneously search their assigned subspaces using the enumeration procedure in Section 2.1.4. The selection of the k PIs is guided by the controllability measure described in the section for the enumeration phase.

2.3 Experimental Results for SIMPLE

In this section we present some representative experimental results for our parallel implementation of SIMPLE on the ISCAS '85 benchmark circuits [7]. The experiments were conducted on a network of Sun workstations. Below shows the timing performance of our parallel program using C432 and C6288 as the target circuits for test generation. Fault collapsing techniques [8] were employed to reduce the size of the stuck-at fault sets for the benchmark circuits. To measure the speedup performance, the simulation run for each of the circuits was repeated with different number of processors involved; i.e., 1, 2, 4, 8, 16, and 32 processors. The performance of this parallel implementation was measured in process time, which, in addition to the theoretical efficiency of the parallelization scheme, is also affected by practical considerations such as the different CPU speeds of the involved machines and process assignment policy in PVM. However, the results clearly show again the effectiveness of this parallel implementation reported in [4].

Fig. 3: Timing Results for c432

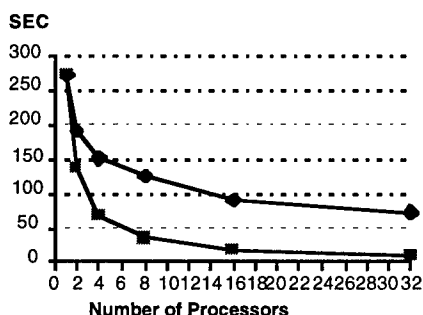
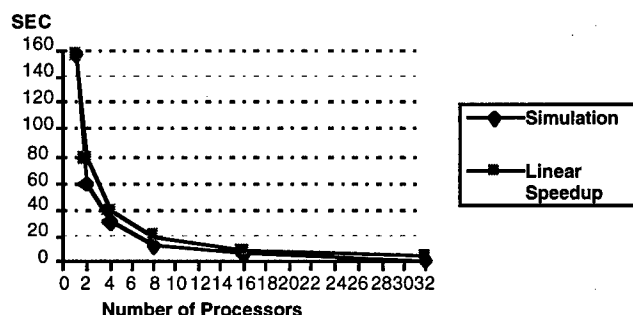


Fig. 4: Timing Results for c6288



3. Delay Fault Test Generation

Ascertaining proper operation of digital circuits requires verification not only of the correct functional operation but of the correct operation at the desired clock rates as well.

Failures causing logic circuit to malfunction at desired clock rates or to not meet timing specifications are called *delay faults*. Delay fault testing is gaining considerable importance with the increased susceptibility to manufacturing defects that increase circuit delays. Moreover, logic designs optimized for gate count, area, or power tend to have too many gates on critical paths, thus making them susceptible to delay faults. Two different delay fault models, the *gate delay fault model* and the *path delay fault model*, have been proposed in the literature. The gate delay fault model has been introduced to model those defects that cause an actual propagation delay through a distinct gate to exceed its worst-case specification [24]. Formerly, it was also referred to as the *transition fault model*, which merely allows a qualitative consideration of gate delay faults of large size [25]. Since being restricted to large-sized delay faults is neither sufficient nor satisfactory, manifold research activities have recently been undertaken in order to explicitly consider the actual size of gate delay faults during test generation and fault simulation [26].

In contrast, in order to overcome the main deficiency associated with the gate delay fault model, the path delay fault model features the advantageous capability of modeling distributed failures, which are typically caused by statistical variations in the manufacturing process [27]. In addition, it is extremely useful for circuit designs based on statistical timing, since those circuits are known to have non-zero probability for the occurrence of delay faults, even when all gate delays are within their specified worst case ranges. The path delay fault model therefore is a more realistic and useful model for delay faults.

For the discussion in the following sections, we now define a few terms related to path delay fault test generation. A path delay fault can be specified by a *functional path* which indicates the structural path from a PI to a PO in the circuit and the desired transitions along that path. We call an input of a gate on a path delay fault an *off-path sensitizing input* if it is not on the structural path that constitutes that fault. The fault universe for the path delay fault model, which comprises all the functional paths in the circuit under test, may grow exponentially with the circuit depth; therefore, delay testing based on

path delay fault model must focus on a tractable subset of the fault universe. The most popular approach to this problem suggests concentrating on the longest paths in the circuit under test in terms of normal delays.

Our approach is independent of any specific delays in the circuit, except in the selection of the target fault set (i.e., the “longest” paths). In this report, we employ the *Unit Gate Delay Model* in our calculations. In this model, every gate and every PI contributes a unit delay (in the normal circuit) to a signal transition that propagate through it. Arbitrarily, we consider “unit delay” to be 1 ns. In this way, our results are independent of any particular cell library or circuit layout. Of course, for any case where the actual rising and falling delay values are known, these values can be incorporated into our delay calculations.

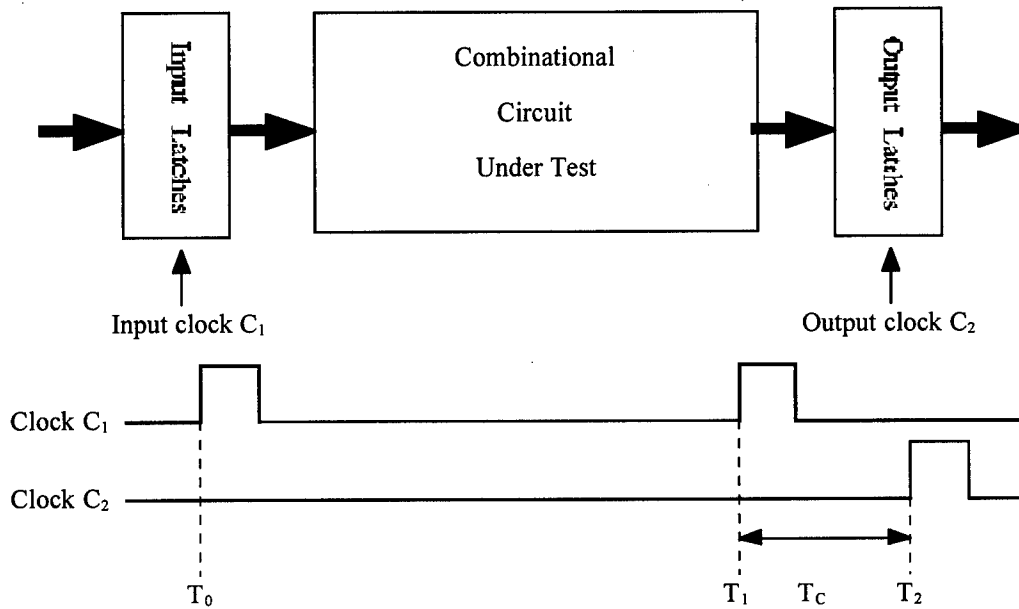
The remaining sections the PDFTG algorithm are organized as follows. Section 3.1 illustrates the hardware model for delay fault testing and categorizes the test patterns for path delay faults. Sections 3.2 to 3.4 describe the major components of our PDFTG algorithm. An algorithm outline and a brief discussion of the compaction procedure are presented in section 3.5. The experimental results for the sequential and parallel implementation of the PDFTG algorithm are shown in section 3.6 and 3.7 respectively.

3.1 Hardware Model and Robust Tests for Path Delay Faults

It is well know that, except when dealing with dynamic logic, testing delay faults requires two patterns rather than a single one as in the case of stuck-at fault testing. Taking the generally accepted hardware model [29, 30] illustrated in Fig. 5 as the basis for further discussion, we assume that the initialization vector V_1 is loaded into the input latches at time T_0 . Subsequently, after all signals of the circuit under test have been allowed to stabilize under V_1 , the propagation vector V_2 is applied to the PIs of the circuit at time T_1 by pulsing clock C_1 . Finally, the logic values at the POs are sampled into the output latches at time $T_2 = T_1 + T_c$ by activating clock C_2 , where T_c represents the system clock

interval at the desired functional clock rate. We assume that the input latches are glitchless, namely, there are no static hazards, and the combinational circuit under test generation is represented by a circuit network with the basic gates AND, NAND, OR, NOR, XOR, XNOR, and NOT as the components.

Figure 5. Hardware model for delay fault testing



The test patterns for the delay faults can be categorized into three classes: *hazard-free*, *robust* [29] and *non-robust* [30] test patterns. A hazard is created at an output of a gate when two or more inputs change their values simultaneously, and the change in one input has reverse polarity in comparison with another input. A hazard-free test of a path introduces no transitions on the off-path sensitizing inputs. Hence, a circuit fails a hazard-free test if and only if it contains the delay fault. The problem with this class is that it is rarely the case that there exists a hazard-free test for a delay fault. Robust testing relaxes the hazard-free restriction on the off-path sensitizing inputs while still maintaining the property that the faults detected by them are not invalidated by delays along other paths. In other words, if a circuit contains a path delay fault then that circuit will fail the

application of a robust test pattern for that fault. Finally, a test pattern of a delay fault is called a non-robust test if it provokes the desired transitions on that path and all off-path sensitizing inputs assume non-controlling final values. Such a test can be invalidated by delays along other paths.

Based on the hardware model and the path delay fault model discussed above, we concentrated our efforts on the task of devising and implementing an algorithm that, given a subset P of the path delay fault universe of a circuit under test, finds a robust two-pattern test $\langle V_1, V_2 \rangle$ for every robustly testable fault in P . The test generation problem for a specific functional path can now be viewed as a search problem in the two-pattern space, defined by the PIs, for a pattern so that the requirements of robustness and propagation of the transitions specified by the functional path are satisfied.

3.2 Logic System and Requirements for Robust Tests

Delay fault testing involves considering the values of a net provoked by the input pattern pair V_1 and V_2 which comprise the two-pattern test. This can be done by representing the value of a net as an ordered pair (b_1, b_2) where b_1 and b_2 are the values of the net in response to the input pattern V_1 and V_2 respectively. Using this representation the value of a net can be one of the elements of the set $\{(0, 0), (0, 1), (1, 0), (1, 1)\}$. Note that element $(0, 1)$ indicates a *rising* transition and $(1, 0)$ a *falling* transition at a net in the circuit. The distinction between transitional and non-transitional values is essential for the dynamic test compaction procedure mentioned at the beginning of this report. The presence of hazards, in the context of delay fault testing, requires that we make the distinction between stable and hazardous values. Consequently, instead of the four basic elements of the above set we will now have six basic values of a net. These are the elements in the basic set $B = \{(0, 0)_s, (0, 0)_h, (0, 1), (1, 0), (1, 1)_s, (1, 1)_h\}$, the subscripts s and h indicate whether a value is *stable* or *hazardous* respectively. We use the following notation to represent the basic set $B = \{0_s, 0_h, \bar{D}, D, 1_s, 1_h\}$, where \bar{D}, D represent $(0, 1)$

and (1, 0) respectively. In the process of generating test patterns for a circuit, it might not be possible to uniquely specify the value of a net as one of the elements in the basic set. However, we may have already known that a net cannot assume one or more of the values in the basic set. We incorporate this information by defining the *value* of a net as a subset of B, thus, there are 64 possible values a net can assume in our logic system, which are elements in the power set of the basic set $P(B)$. The basic set itself hence represents the *totally unknown value* in our logic system.

The robust property for a test for a path delay fault indicates that the presence of the target fault will be detected by an application of the test pattern even in the presence of the effects of other gate transition delays in the circuits. The value of the off-path sensitizing inputs to the gates on the target path should be restricted during test generation so that the above robustness property can be satisfied. Take an AND gate with inputs A and B, and output C as an example. Suppose a falling transition (\bar{D}) at input A is required on the functional path under test generation, and there is another falling transition at the off-path sensitizing input B. If the transition at input B occurs before that at input A, the falling transition at output C will be triggered by the one at input B, rather than at input A. Thus, in this case the off-path sensitizing input B needs to be set to 1_s . On the other hand, if we are trying to propagate a rising transition (\bar{D}) through an AND gate, then since the \bar{D} at the output is triggered by the slowest rising transition at the inputs, we could allow the off-path sensitizing inputs to have any elements in $1_s/1_p/\bar{D}$. Similar results for OR gate can be derived from the principle of duality. However, analysis of propagation of both rising and falling transition from an input of XOR gate to the output shows that the off-path sensitizing inputs must have either 1_s or 0_s , depending on the required transition at the output. Table 7 summarizes the required values at the off-path sensitizing inputs for the robust test.

Table 7: Required values at the off-path sensitizing inputs

Input transition on the function path	AND NAND	OR NOR	XOR XNOR
D	1 _s	0 _s /0 _h /D	0 _s /1 _s
\overline{D}	1 _s /1 _h / \overline{D}	0 _s /0 _h	0 _s /1 _s

3.3 Forward and Backward Implication Procedures

As mentioned above, test generation for a path delay fault of a circuit involves a search in the pattern space defined by the PIs. Our PDFTG algorithm directs the search for a robust test by using deterministic as well as heuristic techniques based on those for SIMPLE. To guide the search deterministically, local forward and backward implication are executed whenever possible. As emphasized and substantiated by the work in [10, 11], the efficiency of any deterministic ATPG for stuck-at faults, as well as for path delay faults, depends strongly upon the power of its implication procedures, whose basic task consists of the immediate assignment of uniquely determined values to the corresponding nets. As shown in [31, 32], the local implication procedures owe their power to the logic system employed in the algorithm. The authors of [31, 32] introduced a criterion called *completeness*, which, for a given basic set of logic values B , allows an efficient determination of a minimal subset of the power set of the basic set $P(B)$ as a logic system for test generation while maximizing the implication power. The completeness criterion essentially defines the ability to express all possible results of all the basic logic functions, for the purpose of test generation, using exactly the elements in a given logic value set. This criterion thus does not take into account the possible values generated from backward implication of all the basic logic functions. The proposed 64-valued logic system is complete in terms of both the forward and backward implications since it is exactly the power set of the basic set.

Below are the tables for the forward and backward implication used in the PDFTG algorithm, with input values from the basic set $\{0_s, 0_h, \overline{D}, D, 1_s, 1_h\}$. We call implications

resulted from the applications of forward and backward implication tables *direct implications*.

The implication procedure is identical to the one for SIMPLE, except the tables used.

Table 8: Forward implication tables for AND and XOR gate

AND	0 _s	0 _h	1 _s	1 _h	D	\overline{D}	XOR	0 _s	0 _h	1 _s	1 _h	D	\overline{D}
0 _s	0 _s	0 _s	0 _s	0 _s	0 _s	0 _s	0 _s	0 _s	0 _h	1 _s	1 _h	D	\overline{D}
0 _h	0 _s	0 _h	0 _h	0 _h	0 _h	0 _h	0 _h	0 _h	0 _h	1 _h	1 _h	D	\overline{D}
1 _s	0 _s	0 _h	1 _s	1 _h	D	\overline{D}	1 _s	1 _s	1 _h	0 _s	0 _h	\overline{D}	D
1 _h	0 _s	0 _h	1 _h	1 _h	D	\overline{D}	1 _h	1 _h	1 _h	0 _h	0 _h	\overline{D}	D
D	0 _s	0 _h	D	D	D	0 _h	D	D	D	\overline{D}	\overline{D}	0 _h	1 _h
\overline{D}	0 _s	0 _h	\overline{D}	\overline{D}	0 _h	\overline{D}	\overline{D}	\overline{D}	\overline{D}	D	D	1 _h	0 _h

Table 9: Forward implication table for NOT gate

NOT	0 _s	0 _h	1 _s	1 _h	D	\overline{D}
	1 _s	1 _h	0 _s	0 _h	\overline{D}	D

Table 10: Backward implication table for AND gate

AND	0 _s	0 _h	1 _s	1 _h	D	\overline{D}
0 _s	0 _s /0 _h /D/ \overline{D} /1 _s /1 _h	\emptyset	\emptyset	\emptyset	\emptyset	\emptyset
0 _h	0 _s	0 _h /D/ \overline{D} /1 _s /1 _h	\emptyset	\emptyset	\emptyset	\emptyset
1 _s	0 _s	0 _h	1 _s	1 _h	D	\overline{D}
1 _h	0 _s	0 _h	\emptyset	1 _s /1 _h	D	\overline{D}
D	0 _s	0 _h / \overline{D}	\emptyset	\emptyset	1 _s /1 _h /D	\emptyset
\overline{D}	0 _s	0 _h /D	\emptyset	\emptyset	\emptyset	1 _s /1 _h / \overline{D}

|| row header: existing value at one input, column header: requested value at the output

Table 11: Backward implication table for XOR gate

XOR	0 _s	0 _h	1 _s	1 _h	D	\overline{D}
0 _s	0 _s	0 _h	1 _s	1 _h	D	\overline{D}
0 _h	\emptyset	0 _s /0 _h	\emptyset	1 _s /1 _h	D	\overline{D}
1 _s	1 _s	1 _h	0 _s	0 _h	\overline{D}	D
1 _h	\emptyset	1 _s /1 _h	\emptyset	0 _s /0 _h	\overline{D}	D
D	\emptyset	D	\emptyset	\emptyset	0 _s /0 _h	1 _s /1 _h
\overline{D}	\emptyset	\overline{D}	\emptyset	\overline{D}	1 _s /1 _h	0 _s /0 _h

|| row header: existing value at one input, column header: requested value at the output

3.4 Derivation of Static Learning Table

In addition to the local implication procedure discussed above, as in SIMPLE, PDFTG also utilized the concept of static learning by the application of the *contrapositive logical identity* introduced in [21] to restrict the search of a test pattern for a functional path.

From the discussion of the basic value set for our logic system, we know that it is the requirement that we make the distinction between the stable and hazardous values that leads us to the formation of the basic set of six elements. The basic set is exactly the same as that used in the analysis of static hazard [28, 33], which can be represented by 3-bit sequences. Following is a table for the correspondence between the 3-bit sequences and the elements in our basic set.

Table 12: Correspondence of the elements in the basic set and the 3-bit sequence

Value	Sequence(s)	Meaning
0_s	000	Static 0
0_h	010	Static 0-hazard
1_s	111	Static 1
1_h	101	Static 1-hazard
\mathbf{D}	$\{100, 110\} = 1x0$	Falling transition
$\overline{\mathbf{D}}$	$\{001, 011\} = 0x1$	rising transition

The first bits in the sequences indicate, in terms of our hardware model, the values at time T_1 . The second bits are the values between time T_1 and T_2 , and the third bits show the values at time T_2 . Notice that 0_h corresponds to the sequence 010, rather than 0x0, since in our case 0x0 means the set $0_s/0_h$. The value 1 at the second bit of the sequence for 0_h actually denotes the possibility of value 1 occurring between T_1 and T_2 . The view of elements in the basic set as 3-bit sequences, and the assumption that the circuit under test generation is combinational, suggest that we use the results of the **3-VP** experiment, as defined in the discussion of SIMPLE, to find the contrapositive implications for our 64-valued logic system. As an example, suppose that we do a **3-VP** experiment with the values at net N_1 set to 0 and the value at net N_2 is 0 after the implication. The direct

implication obtained from the experiment and the contrapositive of this implication are shown below.

$$N_1 = 0 \rightarrow N_2 = 0 \text{ ----- (7)}$$

$$N_2 = 1 \rightarrow N_1 = 1 \text{ ----- (8), contrapositive of direct implication (7) from 3-VP experiment}$$

Now if we are required to set net N_2 to 1_s during test generation, by the application of contrapositive implication (8) to each bit in the corresponding bit sequence 111 for the value 1_s , we know that net N_1 must also be set to 1_s . As another example, if the required value at net N_2 is $D = 1x0$ during test generation, then N_2 can only contain elements in $1xx = 1_s/1_h/D$, where $1xx$ is obtained from the application of rule (8) to $1x0$. The following table summarizes the implied value at net N_1 for each direct implication from the **3-VP** experiments and each element in the basic set as a requirement at net N_2 .

Table 13: Contrapositive implications at net N_1

Basic value at net N_2	$N_1 = 0 \rightarrow N_2 = 0$	$N_1 = 0 \rightarrow N_2 = 1$	$N_1 = 1 \rightarrow N_2 = 0$	$N_1 = 1 \rightarrow N_2 = 1$
0_s	$0_s/0_h/D/\bar{D}/1_s/1_h$	1_s	$0_s/0_h/D/\bar{D}/1_s/1_h$	0_s
0_h	$0_h/D/\bar{D}/1_s/1_h$	$1_s/1_h$	$0_s/D/\bar{D}/1_s/1_h$	$0_s/0_h$
1_s	1_s	$0_s/0_h/D/\bar{D}/1_s/1_h$	0_s	$0_s/0_h/D/\bar{D}/1_s/1_h$
1_h	$1_s/1_h$	$0_h/D/\bar{D}/1_s/1_h$	$0_s/0_h$	$0_s/0_h/D/\bar{D}/1_h$
D	$D/1_s/1_h$	$\bar{D}/1_s/1_h$	$0_s/0_h/\bar{D}$	$0_s/0_h/D$
\bar{D}	$\bar{D}/1_s/1_h$	$D/1_s/1_h$	$0_s/0_h/D$	$0_s/0_h/\bar{D}$

3.5 Algorithm Outline and Test Compaction

We now present the outline of our algorithm for generating a robust test pattern for a functional path and discuss the way test compaction is incorporated during test generation.

1. Initialization - Initialize the nets on the functional path to the required transitions and the off-path sensitizing inputs to the values required to robustly propagate the transitions. Set all the other nets to the unknown value.
2. Implication - Use forward and backward implications, along with the contrapositive implications we found in the static learning procedure, to determine the values of other nets in the circuit. If an inconsistency occurs, then the fault is not robustly testable.
3. Enumeration - Justify any variant nets in the circuit by invoking the enumeration procedure that is identical to that for SIMPLE.
4. If we exhaust the test pattern space defined by those PIs after step 2, then the fault is not robustly testable. If there are no variant nets left, then the pattern at the PIs constitutes a robust test pattern for the current path delay fault.

When the algorithm succeeds in finding a test pattern for the current target fault, often there still remain unassigned PIs. We further process the other faults with the constraints found for the current target fault remaining at the PIs. The algorithm attempts to include as many additional faults as possible until it exhausts the set of unprocessed faults. As noted in the discussion for the path delay fault model, the fault universe is often very large. The situation remains even if we only take a subset of the fault universe as the fault set for test generation. In order to effectively cope with the typically huge number of paths in the fault set, we have adopted a path tree structure [32] to store and examine the fault set efficiently. The basic idea for this stems from the fact that a path delay fault can be specified in two parts, one for the structural path in the circuit it is on, the other for the required transitions on the structural path. Thus parts of paths that are common to many paths from a specific PI can be compactly stored as a tree structure rooted at that PI.

3.6 Experimental Results for the Sequential PDFTG

With the incorporation of test compaction in the algorithm, the test generation process does not need to rely on a fault simulator to identify additional testable path delay faults after creating a test pattern for the current target fault. To assess the efficiency aspects of the algorithm, we have performed robust test generation for the ISCAS '85 standard benchmark circuits on a Sun "Ultra 1" workstation with one Sparc CPU using the sequential version of the algorithm. The subset of the path delay fault universe for each benchmark circuit was constructed by selecting all the functional paths whose lengths, in terms of unit delay, are greater than or equal to a threshold value. The table below depicts the characteristics of the fault sets of the benchmark circuits selected for the test generation experiments.

Table 14: Results from Fault List Generation

Circuit Name	Number of Faults	Number of Selected Faults	Maximum Path Length	Path Length Threshold
C432	583652	200000 ⁱⁱ	18	16
C499	795776	249984 ⁱⁱ	12	12
C880	17284	16194	25	12
C1355	8346432	150000 ⁱⁱ	25	25
C1908	14588114	98144	41	36
C2670	1359920	103360	33	30
C3540	57353342	59840	48	45
C5315	2682610	60940	50	45
C6288	overflow	27000 ⁱⁱ	125	125
C7552	1452988	91664	44	38

ⁱⁱ The total number of faults whose length are not less than the threshold is greater than the number of selected faults

Table 15 shows the results of this experiment. A few remarks on the meaning of some columns are followed: column "Redundant Paths" indicates the number of functional paths that are not robustly testable, column "Compacted Tests" represents the number of test patterns generated for all the testable paths.

Table 15: Results from Test Generation with Compaction

Circuit Name	Size of Test Set	Redundant Paths	Redundancy Ratio	Testable Paths	Compacted Tests	Compaction Ratio	Time (sec) U
C432	200000	199978	99.9%	22	6	27.27%	153.880
C499	249984	229504	91.8%	20480	10240	50%	7079.802
C880	16194	1201	7.42%	14993	1392	9.28%	1077.470
C1355	150000	150000	100%	0	0	NA	34.710
C1908	98144	98048	99.9%	96	94	97.92%	1222.600
C2670	103360	103360	100%	0	0	NA	379.680
C3540	59840	59840	100%	0	0	NA	277.760
C5315	60940	60940	100%	0	0	NA	174.930
C6288	27000	27000	100%	0	0	NA	47.590
C7552	91664	91664	100%	0	0	NA	2870.620

U in terms of CPU sec

For all the circuits considered here, the algorithm completed the process of test generation and compaction in remarkably small amounts of CPU time, in comparison with the time for stuck-at fault test generation. However, it is noticeable that the redundancy ratio, which is the ratio of the number of robustly redundant path delay faults to the total number of processed faults, is very high in almost all of the benchmark circuits; only C880 has a ratio less than 90%. This phenomenon suggests that these circuits are not designed with features to enhance robust testability for the path delay fault model. Nevertheless, the compaction ratio, which is the ratio of the number of generated test patterns to the number of robustly testable paths, shows that the algorithm generated a small set of test patterns efficiently for each circuits with low redundancy ratio. It would be interesting to compare the timing results with those in [31, 32]. However, since the criteria to select the target faults in our experiment are different from those in [31, 32] we can not make the comparison.

3.7 Parallelization of PDFTG

Another experiment for the sequential version of PDFTG, without the compaction procedure on the same sets of path delay faults, was conducted on the same Sun workstation to analyze the distribution of the test generation time of the selected faults.

The results summarized in Table 16 showed that, contrary to the cases of stuck-at fault test generation, the total execution time for each of the target circuits was not determined mainly by a very small fraction of the target fault set. Therefore, it might be more efficient to parallelize PDFTG by partitioning the target fault list instead of the search space as in the parallelization scheme for SIMPLE. This experiment also shows an interesting and surprising fact: not only are almost all the faults in the fault set of nearly every ISCAS '85 circuit not robustly testable, but also each of these faults can be identified within a very short time. Design methodologies that increase the robust testability of at least the timing-critical functional paths must be applied for the automatic test generators to be applicable to the problem of testing the robust path delay faults.

Table 16: Results from Test Generation without Compaction[¶]

Circuit Name	Size of Test Set	Redundant Paths	Max Tm for Redundant Paths	Total Process Time	Time for Redundant Paths	Timing Ratio
C432	200000	199978	0.02	154.280	154.230	99.9%
C499	249984	229504	0.02	401.550	347.280	86.5%
C880	16194	1201	0.09	67.860	32.130	47.3%
C1355	150000	150000	0.01	40.510	40.510	100%
C1908	98144	98048	10.61	1221.200	1220.690	99.9%
C2670	103360	103360	0.02	378.940	378.940	100%
C3540	59840	59840	0.02	279.910	279.910	100%
C5315	60940	60940	0.02	172.810	172.810	100%
C6288	27000	27000	0.01	47.179	47.179	100%
C7552	91664	91664	0.08	3052.440	3052.440	100%

[¶] in terms of CPU sec

A parallel version of PDFTG based on the scheme of partitioning the target fault lists was implemented using the PVM communication package. The timing results of the experiment for this parallel PDFTG on the ISCAS '85 circuits are presented in Table 17, and the performance of this parallel scheme is illustrated in the speedup graphs from Figs. 6-15. Each row in Table 17 shows the execution time of the parallel PDFTG for the corresponding circuit using the same fault set as in the experiment for the sequential program. The performance of this parallel implementation was measured in real time,

which can be affected by the intrinsic efficiency of the parallelization scheme as well as practical considerations such as the communication overhead and the load on each machine. In the multi-user, heterogeneous computing environment that was available for this experiment, it was impossible to completely isolate or even average out the effects of these factors. However, the results clearly show the effectiveness of this parallel implementation.

Table 17: Timing results for parallel version of PDFTG

# of processors	1	2	4	8	12	16
C432	2432.000	1344.000	640.000	448.000	320.000	256.000
C499	27618.537	4352.000	1920.000	1088.000	896.000	576.000
C880	701.718	341.301	233.539	108.499	78.518	50.368
C1355	850.355	388.251	247.761	223.455	209.913	192.000
C1908	12756.544	5238.482	2192.936	1352.781	910.995	842.949
C2670	3274.054	1410.807	1107.352	569.559	361.558	344.342
C3540	4074.841	1530.686	948.049	403.330	347.612	310.847
C5315	1554.499	1090.279	666.914	346.219	329.274	270.903
C6288	1589.463	746.037	451.827	317.276	299.518	271.827
C7552	18914.391	10838.726	8161.644	3010.675	2513.987	1975.421

U in terms of sec

Fig. 6: Timing Results for c432

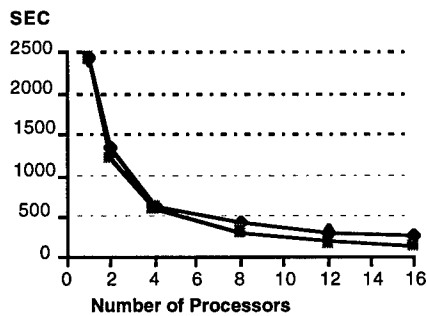


Fig. 7: Timing Results for c499

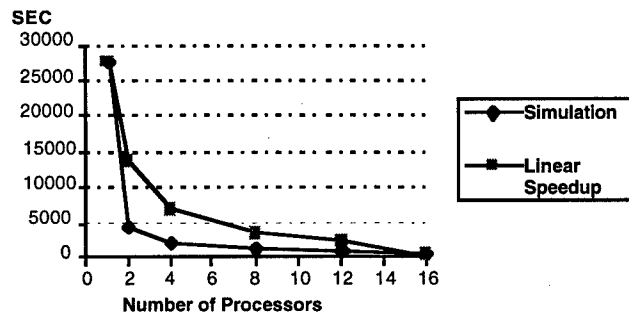


Fig. 8: Timing Results for c880

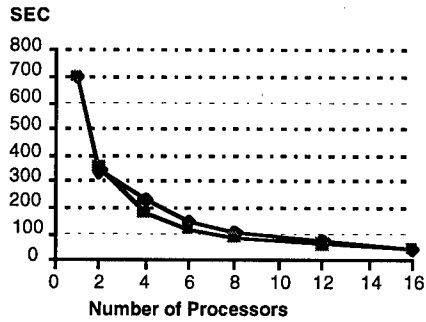


Fig. 9: Timing Results for c1355

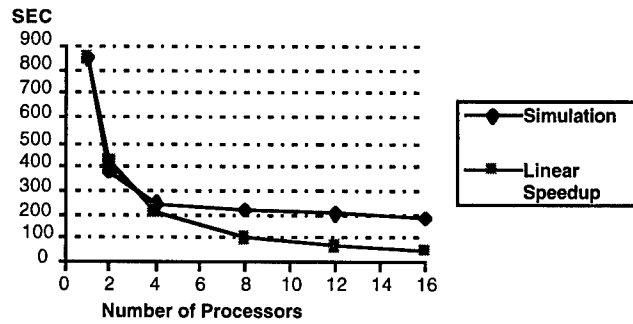


Fig. 10: Timing Results for c1908

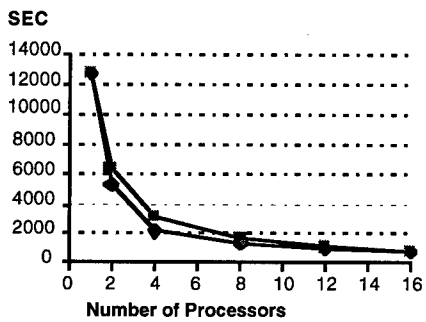


Fig. 11: Timing Results for c2670

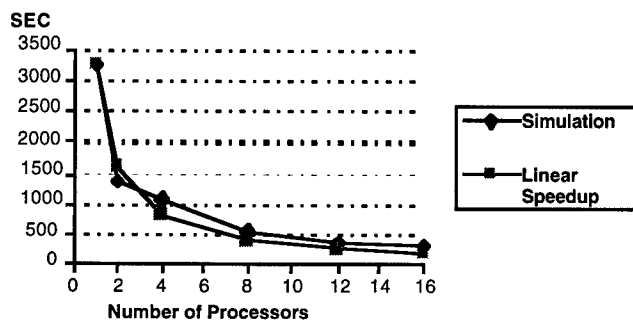


Fig. 12: Timing Results for c3540

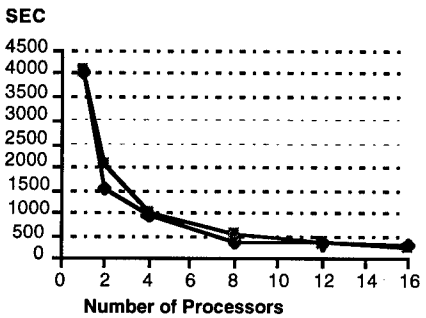


Fig. 13: Timing Results for c5315

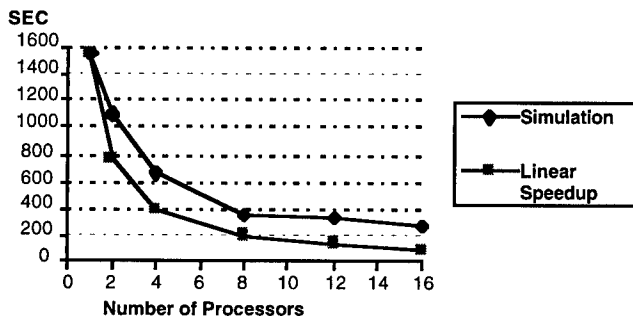


Fig. 14: Timing Results for c6288

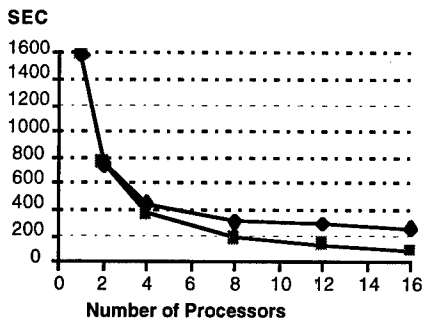
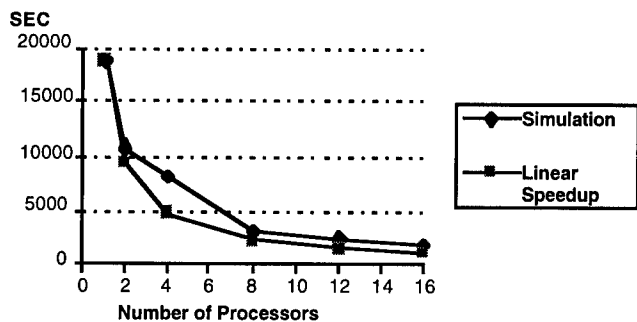


Fig. 15: Timing Results for c7552



4. Discussion

Both stuck-at fault and delay fault testing have been gaining importance due to the ever-increasing VLSI circuit complexity. It is also well known that the test generation processes for stuck-at faults and delay faults are very time-consuming. In this report we have given a concise description of SIMPLE which employ various algorithmic and heuristic techniques, such as contrapositive implication and testability measures, to enhance the performance of test generation for stuck-at faults in combinational circuits. The power of deterministic implication was fully exploited through our 16-valued logic system. The efficiency of our simple and yet effective parallelization scheme was demonstrated again from our experimental results. This near-linear speedup can be attributed to the conclusion of our analysis of the behavior of the sequential implementation: the enumeration phase is responsible for more than 95% of the execution time for the faults that are hard to generate tests for. In turn, the test generation process spends a high percentage of its time for these “hard” faults; most of them are in fact redundant. In addition, our parallelization scheme incurs no communication overhead among the processors.

Our algorithm to generate test patterns for path delay faults followed the strategies and techniques similar to those used in SIMPLE. Furthermore, the ability of test compaction was incorporated into the algorithm to take advantage of the freedom provided by the unassigned input after constructing a test for the current target fault. The logic system in this algorithm was defined to maximize the inference power of both the forward and backward implications. The static learning procedure was revised to take into account the properties of our logic system. The performance of this algorithm has been demonstrated by the impressive results of the experiments on the ISCAS '85 benchmark circuit. The experiments also show an interesting and surprising fact that almost all the faults in the fault set of nearly every target combinational circuit are not robustly testable and each of these faults can be identified within a very short time. This phenomenon suggests that a

lot of circuits are not designed with features to enhance the robust testability for the path delay fault model. Design methodologies that increase the robust testability of at least the timing-critical functional paths must be applied for the automatic test generators to be applicable to the problem of testing the robust path delay faults.

The parallelization scheme of partitioning the target fault list, instead of the search space, among the processors for the algorithm has showed its effectiveness from the speedup figures for the experiments of our parallel implementation of PDFTG on the ISCAS '85 benchmark circuits.

These two parallel implementations of our test generation algorithms for stuck-at faults and path delay faults, together with the versatile PVM communication package, provide us with an efficient prototype of a test generation system that exploits the parallel processing power in the common heterogeneous computer network environment.

Reference

- [1] A. M. Ali and C. R. P. Hartmann, "A Sixteen-Valued Algorithm for Test Generation in Combinational Circuits," Technical Report SU-CIS-91-18, School of Computer and Information Science, Syracuse University, N.Y. 13244, June 1991.
- [2] A. M. Ali , "Use of 16 valued Logic System in Combinational Circuit Testing," Ph.D. Dissertation, Department of Electrical and Computer Engineering, Syracuse University, N.Y. 13244, August 1990.
- [3] A. M. Ali and C. R. P. Hartmann, "A Novel Approach to Delay Fault Testing," *Proceedings of Second Annual Symposium on Communications, Signal Processing, Expert System and ASIC VLSI Design*, Greensboro, North Carolina, March 1991.
- [4] C. R. P. Hartmann and Dennis C. Y. Shiau, "Digital Test Generation Using Multiprocessing," Final Technical Report RL-TR-95-163, US Air Force Rome Laboratory (RL/ERDA), Griffiss AFB, NY 13441, September 1995.

- [5] Sheldon B. Akers, "A Logic System for Fault Test Generation," *IEEE Transactions on Computers*, vol. c-25, pp. 620-630, June 1976.
- [6] F. Brglez, P. Pownall, and R. Hum, "Applications of Testability Analysis: From ATPG to Critical Path Tracing," *IEEE International Test Conference*, pp. 705-712, 1984.
- [7] Franc Brglez and Hideo Fujiwara, "A Neutral Netlist of 10 Combinational Benchmark Circuits and a Target Translator in Fortran," *Proceedings of the IEEE International Symposium on Circuits & Systems*, pp. 663-698, June 1985.
- [8] W. H. Debany, et. at., "Fault coverage Measurement for Digital Microcircuits," MIL-STD-883 Test Procedure 5012, US Air Force Rome Laboratory (RL/ERDA), Griffiss AFB, NY 13441, December 18 1989 (Notice 11) and July 27 1990 (Notice 12).
- [9] Charles W. Cha, William E. Donath, and Fusun Ozguner, "9-V Algorithm for Test Pattern Generation of Combinational Digital Circuits." *IEEE Transactions on Computers*, vol. c-27, pp. 193-209, March 1978.
- [10] S. J. Chandra and J. H. Patel, "Experimental Evaluation of Testability Measures for Test Generation," *IEEE Transactions on Computer-Aided Design*, vol. 8, no. 1, pp. 93-98, January 1989.
- [11] H. Fujiwara and T. Shimono, "On the Acceleration of Test Generation Algorithms," *IEEE Transactions on Computers*, vol. c-32, pp. 1137-1144, December 1983.
- [12] H. Fujiwara and S. Toida, "The Complexity of Fault Detection: An Approach to Design for Testability," *Proceedings of the 12th International Symposium on Fault Tolerant Computing*, pp. 101-108, June 1982.
- [13] Prabhakar Goel, "An Implicit Enumeration Algorithm to Generate Tests for Combinational Logic Circuits," *IEEE Transactions on Computers*, vol. c-30, no. 3, pp. 215-222, March 1981.

- [14] L. H. Goldstein and E. L. Thigpen, "SCOAP: Sandia Controllability/Observability Analysis Program," *Proceedings of the 17th ACM/IEEE Design Automation Conference*, pp. 190-196, 1980.
- [15] O. H. Ibarra and S. K. Sahni, "Polynomially Complete Fault Detection Problems," *IEEE Transactions on Computers*, vol. c-24, pp. 242-259, March 1975.
- [16] Tom Kirkland and M. Ray Mercer, "A Topological Search Algorithm for ATPG," *Proceedings of the 24th ACM/IEEE Design Automation Conference*, pp. 502-508, 1987.
- [17] P. Muth, "A Nine-Valued Circuit Model for Test Generation," *IEEE Transactions on Computers*, vol. c-25, no. 6, pp. 630-636, June 1976.
- [18] Janusz Rajski and Henry Cox, "A Method of Test Generation and Fault Diagnosis in Very Large Combinational Circuits," *IEEE International Test Conference*, pp. 932-943, 1987.
- [19] J. P. Roth, W. G. Bouricius, and O. R. Schneider, "Programmed Algorithms to Compute Tests to Detect and Distinguish Between Failures in Logic Circuits," *IEEE Transactions on Computers*, vol. c-16, pp. 567-579, October 1967.
- [20] Donald M. Schuler, Ernst G. Ulrich, Thomas E. Baker, and Susan P. Bryant, "Random Test Generation Using Concurrent Logic Simulation," *Proceedings of the 12th Design Automation Conference*, pp. 261-267, 1975.
- [21] M. H. Schulz, et. al., "SOCRATES: A Highly Efficient Automatic Test Pattern Generation System," *Proceedings of the International Test Conference*, pp. 1016-1026, September 1987.
- [22] Michael H. Schulz and Elisabeth Auth, "Advanced Automatic Test Pattern Generation and Redundancy Identification Techniques," *Proceedings of the 18th Symposium on Fault-Tolerant Computing*, pp. 30-35, 1988.
- [23] R. Tarjan, "Finding Dominators in Directed Graphs," *SIAM Journal of Computing*, vol. 3, no. 11, pp. 62-89, 1974.

- [24] Z. Barzilai and B. K. Rosen, "Comparison of AC Self-Testing Procedure," *Proceedings of the IEEE International Test Conference*, pp. 89-91, October 1983.
- [25] S. Koeppe, "Modeling and Simulation of Delay Faults in CMOS Logic Circuits," *Proceedings of the IEEE International Test Conference*, pp. 530-536, September 1986.
- [26] A. K. Pramanick and S. M. Reddy, "On the Detection of Delay Faults," *Proceedings of the IEEE International Test Conference*, pp. 845-856, September 1988.
- [27] J. J. Shedletsky and J. D. Lesser, "An Experimental Delay Test Generator for LSI Logic," *IEEE Transactions on Computers*, vol. c-29, pp. 235-248, March 1980.
- [28] M. Abramovici, et. al., "Digital Systems Testing and Testable Design," Computer Science Press.
- [29] G. L. Smith, "Model for Delay Faults Based upon Paths," *Proceedings of the IEEE International Test Conference*, pp. 342-349, September 1985.
- [30] S. M. Reddy, et. al., "On Delay Testing in Logic Circuits," *Proceedings of the IEEE International Conference on Computer Aided Design*, pp. 148-151, 1986.
- [31] M. H. Schulz, et. al., "Advanced Automatic Test Pattern Generation Techniques for Path Delay Faults," *Proceedings of the IEEE International Conference on Computer Aided Design*, pp. 44-51, 1989.
- [32] M. H. Schulz, et. al., "Dynamite: An Efficient Automatic Test Pattern Generation System for Path Delay Faults," *IEEE Transactions on Computer-Aided Design*, vol. 10, no. 10, pp. 1323-1335.
- [33] J.P. Hayes, "Uncertainty, Energy, and Multiple-Valued Logics," *IEEE Transactions on Computers*, vol. c35, no. 2, pp. 107-114, February 1986.
- [34] Al Geist, et. al., "PVM: Parallel Virtual Machine - A Users Guide and Tutorial for Networked Parallel Computing," MIT Press.

MISSION OF ROME LABORATORY

Mission. The mission of Rome Laboratory is to advance the science and technologies of command, control, communications and intelligence and to transition them into systems to meet customer needs. To achieve this, Rome Lab:

- a. Conducts vigorous research, development and test programs in all applicable technologies;
- b. Transitions technology to current and future systems to improve operational capability, readiness, and supportability;
- c. Provides a full range of technical support to Air Force Material Command product centers and other Air Force organizations;
- d. Promotes transfer of technology to the private sector;
- e. Maintains leading edge technological expertise in the areas of surveillance, communications, command and control, intelligence, reliability science, electro-magnetic technology, photonics, signal processing, and computational science.

The thrust areas of technical competence include: Surveillance, Communications, Command and Control, Intelligence, Signal Processing, Computer Science and Technology, Electromagnetic Technology, Photonics and Reliability Sciences.