RL-TR-95-163
Final Technical Report
September 1995

DIGITAL TEST GENERATION USING MULTIPROCESSING

Syracuse University

Carlos R.P. Hartmann and Dennis C.Y. Shiau

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

Rome Laboratory
Air Force Materiel Command
Griffiss Air Force Base, 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-95-163 has been reviewed and is approved for publication.

APPROVED:

WARREN H. DEBANY, JR., Ph.D., P.E.
Project Engineer

FOR THE COMMANDER:

JOHN J. BART
Chief Scientist, Reliability Sciences
Electromagnetics & Reliability Directorate

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY (Leave Blank)

2. REPORT DATE
September 1995

3. REPORT TYPE AND DATES COVERED
Final Mar 91 - Jun 92

4. TITLE AND SUBTITLE
DIGITAL TEST GENERATION USING MULTIPROCESSING

5. FUNDING NUMBERS
C - F30602-91-D-0001, Task 7
PE - 62702F
PR - 2338
TA - 01
WU - P8

6. AUTHOR(S)
Carlos R.P. Hartmann and Dennis C.Y. Shiau

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Syracuse University
School of Computer & Information Science
Suite 4-116, 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 Rd
Griffiss AFB Ny 13441-4505

10. SPONSORING/MONITORING AGENCY REPORT NUMBER
RL-TR-95-163

11. SUPPLEMENTARY NOTES
Rome Laboratory Project Engineer: Warren H. Debany, Jr., Ph.D., P.E./ERDA/(315) 330-2922

12a. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release; distribution unlimited.

12b. DISTRIBUTION CODE

13. ABSTRACT (Maximum 200 words)

The SIxteen valued Maximized Propagation Lowered Enumeration (SIMPLE) algorithm is modified to execute efficiently within a parallel or multiprocessing environment. SIMPLE was developed to generate test vectors for stuck-at faults in digital logic circuits and executes on a scalar processor. Implementation of the multiprocessing version was not completed, but simulations showed that speedup was nearly linear with the number of processors for detectable and undetectable faults.

14. SUBJECT TERMS
Parallel processors, Multiprocessors, Fault detection, Test generation, Digital logic circuits

15. NUMBER OF PAGES
60

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	UL

NSN 7540-01-280-5500

Standard Form 298 (Rev. 2 89)
Prescribed by ANSI Std. Z39-18
298-102

Contents

List of Tables

List of Figures

1 Introduction

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 [10, 13]. Several Automatic Test Pattern Generation (ATPG) algorithms for detecting stuck-at faults in combinational circuits exist in the literature [5, 7, 9, 11, 15, 17, 18, 20]. SIMPLE, an ATPG algorithm based on a 16-valued logic system, is proposed in [2]. This algorithm introduces some novel approaches to making test generation more efficient.

Two prototype implementations of SIMPLE were developed in C. The first program is written for a sequential architecture computer, and the other for parallel.

In Section 2, we describe our test pattern generation system. In Section 3 a short description of SIMPLE is given for completeness. The strategy used in the implementation of this parallel version of SIMPLE is described in Section 4. In Section 5 we give simulation results, and discuss them in Section 6.

2 Automatic Test Pattern Generation System

Our *Automatic Test Pattern Generation (ATPG) System* constructs test patterns to detect all detectable single stuck-at faults in a given combinational circuit, and identifies the undetectable faults, that is, single stuck-at faults for which no test exists. Since we are interested only in constructing test patterns for single stuck-at faults, it is understood that a "fault" is a "single stuck-at fault". A stuck-at-x, $x \in \{0, 1\}$, will be denoted by $s - a - x$.

Given a circuit, an reduced fault list, $\mathcal{L}_r$, is generated. Each fault in $\mathcal{L}_r$ is given to our test pattern generation algorithm (SIMPLE) which either constructs a test pattern to detect the given fault, or identifies that no such a pattern exists. After all the faults in list $\mathcal{L}_r$ are given to SIMPLE, the system calculates the fault coverage and produces a fault dictionary. Fig. 1 shows a block diagram for our ATPG system.

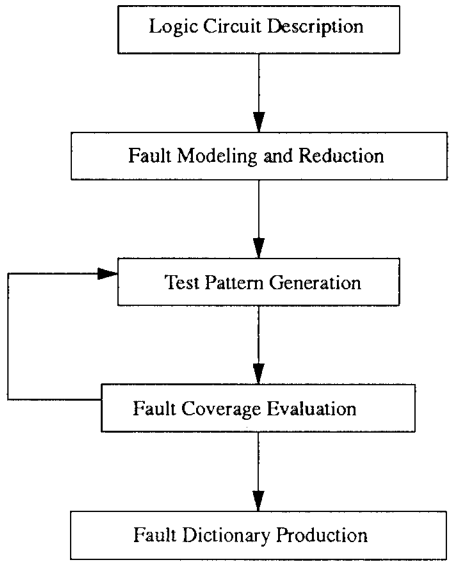


Figure 1: Procedure for test generation.

2.1 Input-Output Format and Data Structure

The ISCAS '85 benchmark circuits [4] are used as our test data. The ISCAS '85 benchmark circuits are ten combinational networks provided to researchers at the 1985 International Symposium on Circuits and Systems to be used as data for comparison of the performance of different ATPG systems. The ISCAS '85 netlist format was distributed on magnetic tape along with a FORTRAN translator that would generate netlists in a few different formats. Although a new translator is now available which produces a netlist in a format that is easier to be read, we use a translator, written by Dong-Liang Jan and Kuo-Kuei Ho (J-H Translator) [14], which is more suitable for our program.

Fig. 2 shows one of the ISCAS '85 benchmark circuits, known as "c17." The original format for this circuit is given below:

```
ISCAS '85 netlist format:

1   1gat inpt 1  0   >sa1
```

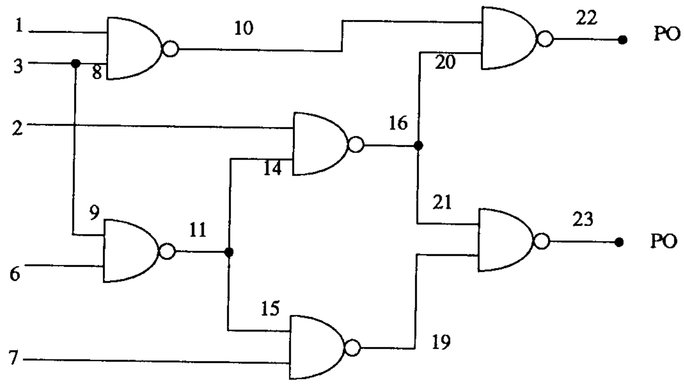


Figure 2: ISCAS '85 Benchmark Circuit C17.isc.

```
2     2gat inpt 1 0 >sa1
3        3gat inpt 2  0 >sa0  >sa1
8     8fan from  3gat >sa1
9     9fan from  3gat  >sa1
6       6gat inpt 1 0 >sa1
7     7gat inpt 1 0 >sa1
10       10gat nand 1 2 >sa1
  1        8
11       11gat nand 2 2 >sa0  >sa1
  9        6
14       14fan from  11gat >sa1
15       15fan from  11gat >sa1
16       16gat nand 2 2 >sa0 >sa1
  2       14
20       20fan from   16gat >sa1
21       21fan from     16gat >sa1
```

```
19        19gat nand 1 2 >sa1
 15         7
22        22gat nand 0        2 >sa0  >sa1
 10       20
23        23gat nand 0 2 >sa0  >sa1
 21        19
```

The format given by the J-H Translator which is the input format for our programs is as follows:

For an m-input circuit, where $m \leq N_{PI}$ and $N_{PI} \in \{100, 1000\}$, the primary inputs (PIs) are numbered from 1 to m. The output of gates are numbered using a leveling rule. That is, the number assigned to the output of a gate is always greater than the number(s) assigned to its input(s). Also, the gate number (G) is always the same as the number associated with its output net. The nets which are output of gates are numbered with $N_{PI}+1$,$N_{PI}+2$,..., $N_{PI}+M$, where M is the number of gates in the circuit. The first line in the format indicates how many PIs the circuit has. Thus, the first line is:

$$m\ PI$$

A gate whose input are numbered n_1, n_2,..., n_s is indicated in this format:

$$n_1\ n_2\ \ldots\ n_s\ \ type\_of\_gate$$

Gates types are AND, NAND, OR, NOR, XOR, XNOR, BUFFER, and NOT. If net n is a primary output (PO), it is indicated by

$$n\ \ PO$$

The output generated by the J-H Translator for the circuit in Fig. 2 is as follows:

```
5 pi
1 3 nand
```

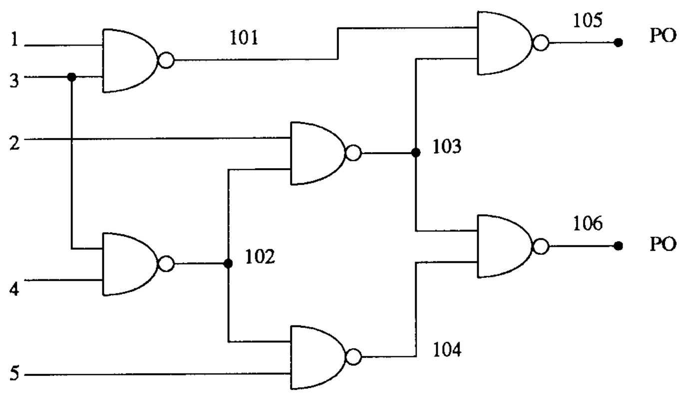


Figure 3: Net numbers assigned by J-H Translator for c17.isc.

```
3 4 nand

2 102 nand

102 5 nand

101 103 nand

105 po

103 104 nand

106 po
```

Fig. 3 gives the circuit of Fig. 2 with the net numbers assigned by the J-H Translator.

While the original format gives the circuit description and a fault list, the format given by the J-H Translator gives only the circuit description. Thus we must create a fault list. In this list a fault is identified as follows:

- n FOS $s-a-x$

  identifies the fault net n $s-a-x$, where net n is a fanout stem.

- n PO $s-a-x$

  identifies the fault net n $s-a-x$, where net n is a PO.

Now, let net n_1 be a PI or the output of gate G_1 which is not a PO. Furthermore, assume that net n_1 is connected to an input of gate G_2 whose output is net n_2.

- n_1 n_2 $s-a-x$

  identifies the fault:

  1. net n_1 $s-a-x$ if net n_1 is not a fanout stem.
  2. fanout branch connecting net n_1 to gate G_2 $s-a-x$ if net n_1 is a fanout stem.

2.2 Construction of the Fault List

In general we want to construct a test set that detects all possible single stuck-at faults in a combinational circuit. In a circuit, $\mathcal{C}$, with n signal lines there are $2n$ possible single stuck-at faults. Thus the initial fault list, $\mathcal{L}$, may contain $2n$ faults. However, we can reduce the cardinality of $\mathcal{L}$ based on the functional equivalence concept.

Let $\mathcal{C}$ be a circuit that realizes a function $Z(X)$. In the presence of a fault α (β) this circuit realizes $Z_\alpha(X)(Z_\beta(X))$.

Definition: Two faults α and β are said to be *functional equivalent* if and only if $Z_\alpha(X) = Z_\beta(X)$.

To decide if two faults are equivalent may be very time-consuming. However, some equivalent faults can be easily identified. To reduce the cardinality of $\mathcal{L}$ we use the procedure proposed in [8] which identifies equivalent faults based on gate fault equivalence only. This procedure is easily to be implemented and its time complexity is $O(n)$, but it does not identify all the equivalent faults.

Before indicating which faults belong to the reduced fault list $\mathcal{L}_r$, we introduce the following definition:

Stuck-at faults	Type of logic line in logic model
$s-a-1$	Every input of multiple-input AND or NAND gates
$s-a-0$	Every input of multiple-input OR or NOR gates
$s-a-0, s-a-1$	Every input of multiple-input components that are not AND, OR, NAND, or NOR gates
$s-a-0, s-a-1$	Every logic line that is a FOS
$s-a-0, s-a-1$	Every logic line that is a PO of type II

Table 1: Faults comprising list $\mathcal{L}_r$.

Definition: A PO of type I is a PO which is directly connected to a FOS or connected to a fanout stem through single-input gates only. A PO of type II is a PO which is not of type I.

All single permanent stuck-at faults specified in Table 1 belong to $\mathcal{L}_r$,

Table 2 gives the number of faults in $\mathcal{L}_r$ for the benchmark circuits [4], where the faults in $\mathcal{L}_r$ were identified using the above procedure.

3 SIMPLE Algorithm

3.1 Introduction

In this section we give a concise description of SIMPLE (SIxteen valued, Maximized Propagation Lowered Enumeration approach to test generation) [2], for detecting single stuck-at-faults in combinational circuits that contain NOT, AND, NAND, OR, NOR, XOR and XNOR gates. This algorithm is based on a 16-valued logic system and introduces some novel approaches to making test pattern generation more efficient.

Test 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 [16]. 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

Circuit Name	Circuit Function	Total Gates	Input Lines	Output Lines	Cardinality of $\mathcal{L}_r$
C432	Priority Decoder	160 (18 EXOR)	36	7	518
C499[1]	ECAT	202 (104 EXOR)	41	32	758
C880	ALU and Control	383	60	26	940
C1355[1]	ECAT	546	41	32	1574
C1908	ECAT	880	33	25	1879
C2670	ALU and Control	1193	233	140	2641
C3540	ALU and Control	1669	50	22	3428
C5315	ALU and Selector	2307	178	123	5248
C6288	16-bit Multiplier	2406	32	32	7744
C7552	ALU and Control	3512	207	108	7438

[1]Circuits C499 and C1355 are functionally equivalent. All EXOR gates of C499 have been expanded into their 4-NAND gate equivalents in C1355.

Table 2: Table 1 of ISCAS '85 benchmark circuit descriptions.

AND	$\mathbf{0}$	$\mathbf{1}$	$\boldsymbol{D}$	$\overline{\boldsymbol{D}}$
$\mathbf{0}$	$\mathbf{0}$	$\mathbf{0}$	$\mathbf{0}$	$\mathbf{0}$
$\mathbf{1}$	$\mathbf{0}$	$\mathbf{1}$	$\boldsymbol{D}$	$\overline{\boldsymbol{D}}$
$\boldsymbol{D}$	$\mathbf{0}$	$\boldsymbol{D}$	$\boldsymbol{D}$	$\mathbf{0}$
$\overline{\boldsymbol{D}}$	$\mathbf{0}$	$\overline{\boldsymbol{D}}$	$\mathbf{0}$	$\overline{\boldsymbol{D}}$

Table 3: AND Table.

Variable	$\mathbf{0}$	$\mathbf{1}$	$\boldsymbol{D}$	$\overline{\boldsymbol{D}}$
Complement	$\mathbf{1}$	$\mathbf{0}$	$\overline{\boldsymbol{D}}$	$\boldsymbol{D}$

Table 4: NOT Table.

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 these 16 sets as ϕ, $\mathbf{0}$, $\mathbf{1}$, $\boldsymbol{D}$, $\overline{\boldsymbol{D}}$, $\mathbf{0/1}$, $\mathbf{0}/\boldsymbol{D}$, $\mathbf{1}/\boldsymbol{D}$, $\mathbf{0}/\overline{\boldsymbol{D}}$, $\mathbf{1}/\overline{\boldsymbol{D}}$, $\boldsymbol{D}/\overline{\boldsymbol{D}}$, $\mathbf{0/1}/\boldsymbol{D}$, $\mathbf{0/1}/\overline{\boldsymbol{D}}$, $\mathbf{0}/\boldsymbol{D}/\overline{\boldsymbol{D}}$, $\mathbf{1}/\boldsymbol{D}/\overline{\boldsymbol{D}}$, and $\mathbf{0/1}/\boldsymbol{D}/\overline{\boldsymbol{D}}$ where $\mathbf{0}=\{(0,0)\}$, $\mathbf{1}=\{(1,1)\}$, $\boldsymbol{D}=\{(1,0)\}$, $\overline{\boldsymbol{D}}=\{(0,1)\}$ and "/" denotes set union. Note that $U=\mathbf{0/1}/\boldsymbol{D}/\overline{\boldsymbol{D}}$. The value ϕ needs to be included to reflect the situation when two or more constraints require disjoint values on a net. For example, if at some step of the algorithm a net has the value $\mathbf{0/1}/\overline{\boldsymbol{D}}$, then this net cannot have the value $\overline{\boldsymbol{D}}$, either because this value will desensitize the path that the algorithm is trying to sensitize, or because it is inconsistent with the assignment of the PIs. These 16 values are equivalent to the elements of the logic system developed by Akers [1] to provide a tool for test generation. Tables 3, 4, and 5 represent the AND, NOT, and XOR functions in our 16-valued system for the values $\mathbf{0}$, $\mathbf{1}$, $\boldsymbol{D}$, and $\overline{\boldsymbol{D}}$. The complete table for all the 15 non-ϕ values can be easily constructed from the given tables by using the set union operation. The tables for all other logic functions can be obtained from these three tables. Note that any logic function with ϕ as one of its arguments will yield ϕ as a result.

Using this notation we define a sensitized net as one whose value is either $\boldsymbol{D}$,

XOR	0	1	D	$\overline{D}$
0	0	1	D	$\overline{D}$
1	1	0	$\overline{D}$	D
D	D	$\overline{D}$	0	1
$\overline{D}$	$\overline{D}$	D	1	0

Table 5: XOR Table.

$\overline{D}$, or $D/\overline{D}$. 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, and the particular circuit path chosen in order to propagate its effect.

There are three distinct phases in the algorithm presented here:

(*i*) Pre-processing phase (§3.2). 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.

(*ii*) Propagation phase (§3.3). In this phase we deliberately sensitize a single path from the fault site to a PO and find all the resulting deterministic forward and backward implications. In the process other paths may 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 (§3.4). 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 path chosen to be sensitized. 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 PIs so that such combinations can never occur.

To illustrate the above phases of our algorithm we will construct a test pattern for the fault net 3 $s - a - 0$ in the circuit of Fig. 4.

3.2 Pre-processing Phase

3.2.1 Construction of Dominator Forest

The importance of identifying nets that *must* be sensitized for a fault to be detected was first highlighted by Akers [1] and later by Fujiwara and Shimono [9]. As pointed out in TOPS [15], the concept of graph dominators [21] 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 circuit 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 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 FOBs corresponding to this FOS have been marked as FOB leaves. Find the immediate 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

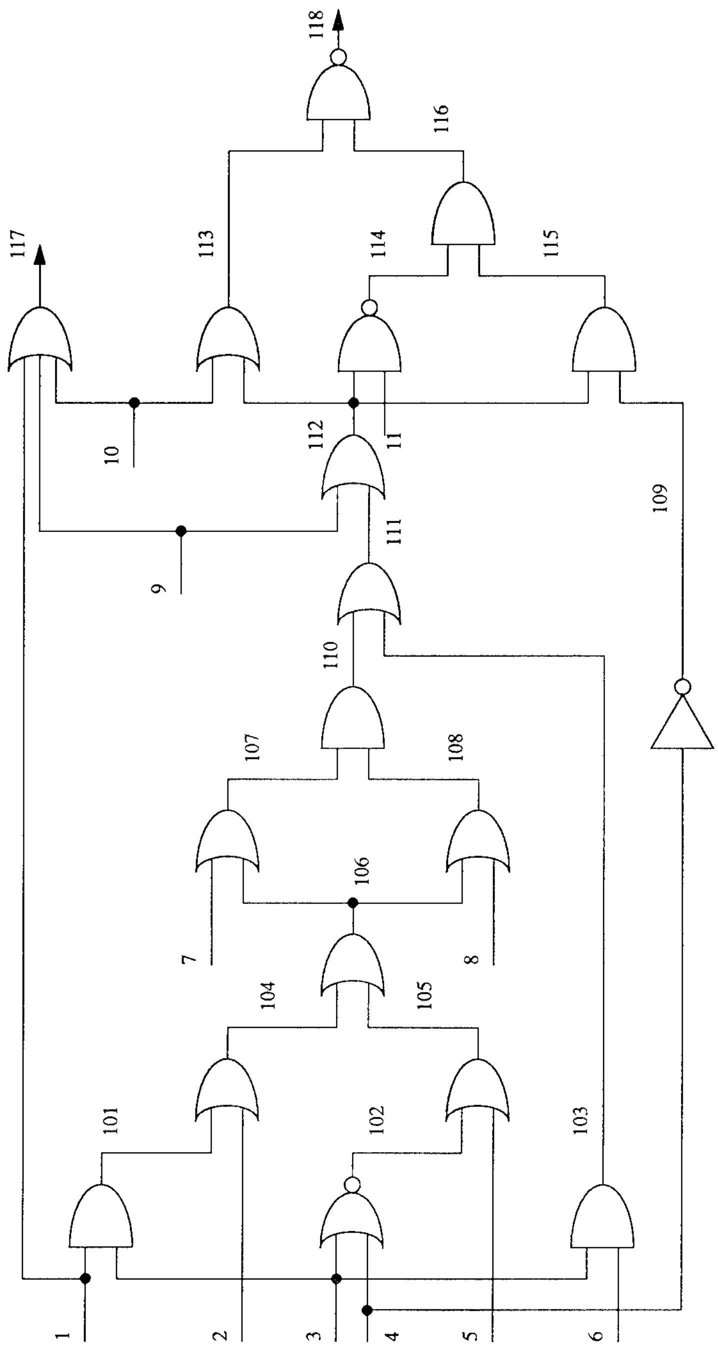


Figure 4: An example circuit.

common ancestor is that they belong to the same tree. If such an ancestor exists, then make 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 (in which m is a node) from m to the root.

The dominator forest for the circuit in Fig. 4 is shown in Fig. 5.

Recall that FOBs are not numbered in our description of the circuit. In the dominator forest they are identified by the number associated with its corresponding FOS followed by a "B" (for branch).

3.2.2 Selection of pdcf

The selection of the primitive D-cube of failure (***pdcf***) in DALG [18] 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 introduce G_f between net n and a newly created net n_f as shown in Fig. 6A. We now 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_1, then the G_f is inserted in this FOB as shown in Fig. 6B. Accordingly, the unique ***pdcf*** depends only on the kind of stuck-at fault.

	n	n_f
fault site s-a-0	1	D
fault site s-a-1	0	$\overline{D}$

Thus in our example we will modify the circuit in Fig. 4 to include the gate shown in Fig. 6C.

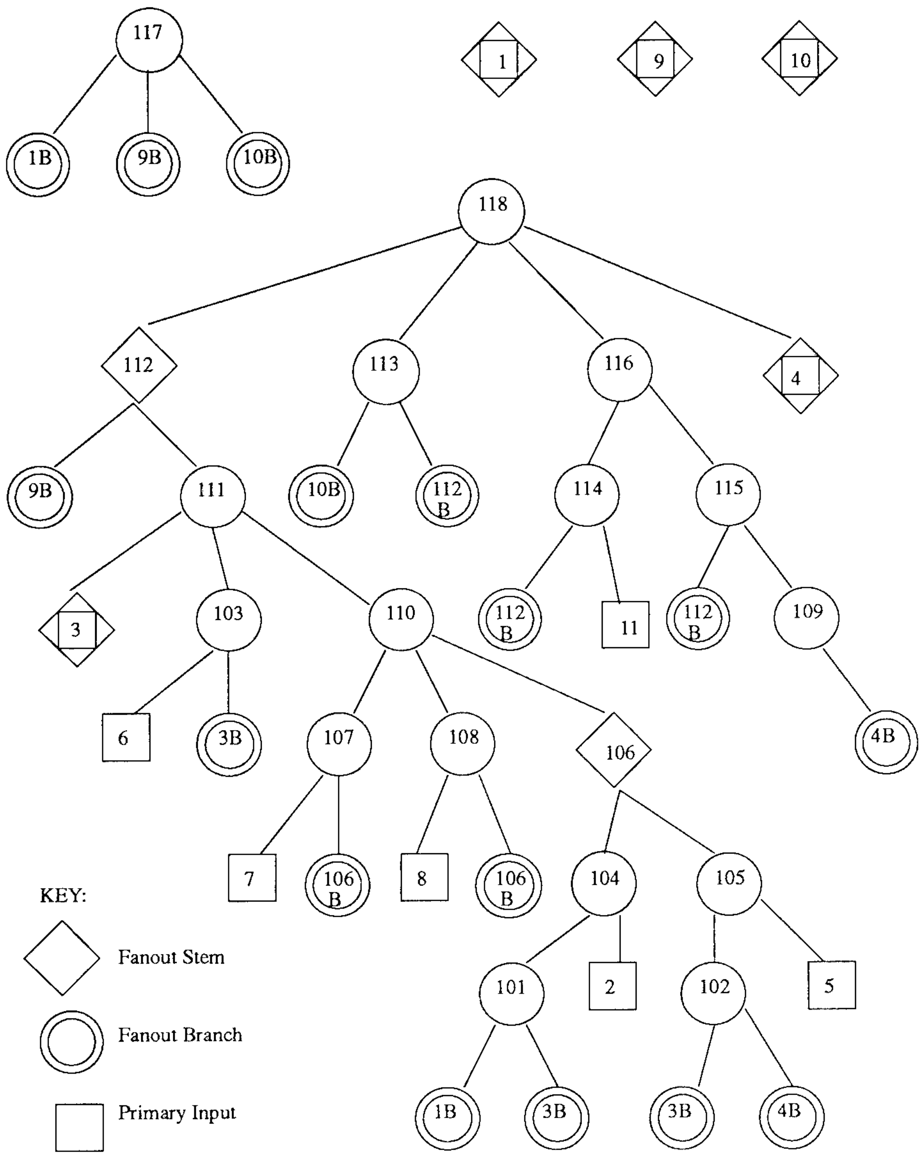


Figure 5: Dominator forest for circuit of Fig. 4.

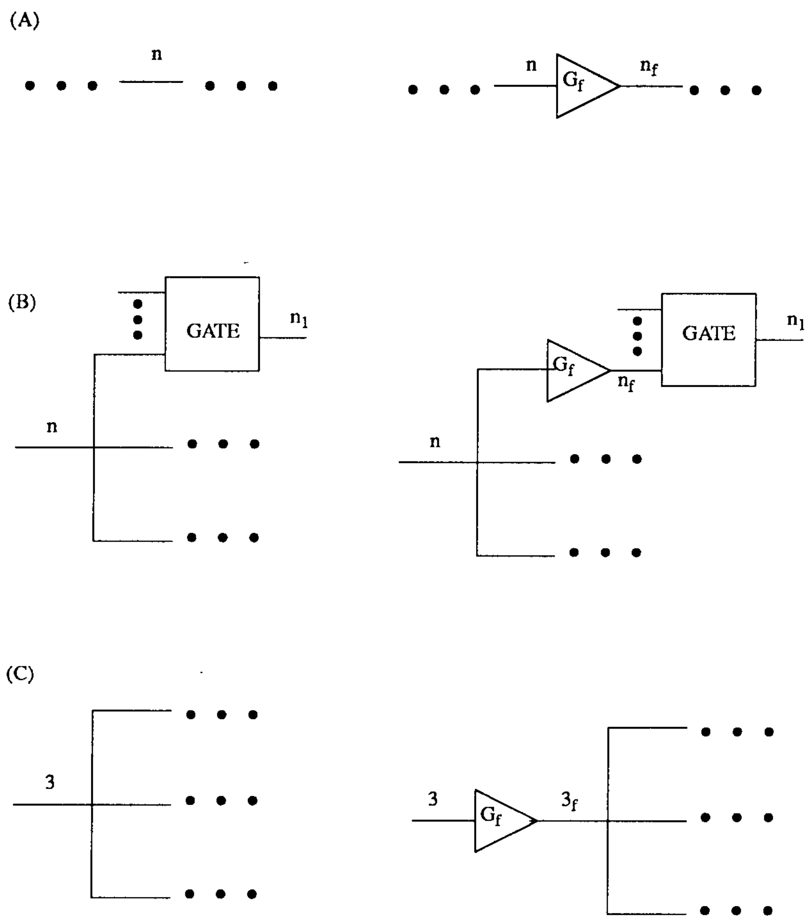


Figure 6: (A) Fictitious gate. (B) Fictitious gate for FOBs. (C) Fictitious gate for net 3 s-a-0 in circuit of Fig. 4.

Nets with TRUE Token	3_f, 101, 102, 103, 104, 105, 106, 107, 108, 110, 111, 112, 113, 114, 115, 116, 118
Nets with FALSE Token	1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11,109, 117

Table 6: Token assignment for net 3 $s-a-0$ in Fig. 4.

3.2.3 Token Assignment

The goal of this stage is to identify which circuit nets can or 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. Table 6 shows the Boolean token assignment for our example.

3.3 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 [18] 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, $\boldsymbol{tc}_k$, are elements of our 16-valued system.

We initialize this phase by constructing $\boldsymbol{tc}_1$ in the following manner:

1. Set nets n and n_f to the values specified by the $\boldsymbol{pdcf}$.

2. Assign $\boldsymbol{D/\overline{D}}$ to all nets belonging to the set $D(n)$.

3. Set all nets with FALSE tokens, except net n, to $\mathbf{0/1}$.

4. Assign $\mathbf{0/1/}\boldsymbol{D/\overline{D}}$ to all unassigned nets of the test cube.

In our example $D(3) = \{111, 112, 118\}$, and the resulting $\boldsymbol{tc}_1$ is given below where only nets whose entries are different from $\mathbf{0/1}$ and $\mathbf{0/1/}\boldsymbol{D/\overline{D}}$ are shown.

3	3_f	111	112	118
1	$\boldsymbol{D}$	$\boldsymbol{D/\overline{D}}$	$\boldsymbol{D/\overline{D}}$	$\boldsymbol{D/\overline{D}}$

For each test cube $\boldsymbol{tc}_k$ generated at any stage of our algorithm we find its corresponding "deterministic" test cube, $\boldsymbol{d}(\boldsymbol{tc}_k)$. We define a $\boldsymbol{d}(\boldsymbol{tc}_k)$ 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. Rules for forward and backward implication procedures to be used in constructing $\boldsymbol{d}(\boldsymbol{tc}_k)$ from $\boldsymbol{tc}_k$ are given in Appendix A. If in any $\boldsymbol{d}(\boldsymbol{tc}_j)$ we have a sensitized path $\boldsymbol{p}_i$ from the fault site to any PO, then the enumeration phase is invoked. This test cube, $\boldsymbol{d}(\boldsymbol{tc}_j)$, is denoted as $\boldsymbol{T}_f(\boldsymbol{p}_i)$. The $\boldsymbol{d}(\boldsymbol{tc}_1)$ for our example is shown below. Only the entries for nets whose values are different from those in $\boldsymbol{tc}_1$ are listed. In fact, for each cube that we construct only the entries whose values are different from those in the preceding one will be explicitly shown.

9	101	102	103	104	105	110
$\mathbf{0}$	$\mathbf{0}/\boldsymbol{D}$	$\mathbf{0}/\overline{\boldsymbol{D}}$	$\mathbf{0}/\boldsymbol{D}$	$\mathbf{0}/\mathbf{1}/\boldsymbol{D}$	$\mathbf{0}/\mathbf{1}/\overline{\boldsymbol{D}}$	$\mathbf{0}/\boldsymbol{D}/\overline{\boldsymbol{D}}$

113	114	115	116
$\mathbf{1}/\boldsymbol{D}/\overline{\boldsymbol{D}}$	$\mathbf{1}/\boldsymbol{D}/\overline{\boldsymbol{D}}$	$\mathbf{1}/\boldsymbol{D}/\overline{\boldsymbol{D}}$	$\mathbf{1}/\boldsymbol{D}/\overline{\boldsymbol{D}}$

If $\boldsymbol{d}(\boldsymbol{tc}_1)$ 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 corresponding to 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_f to the PO corresponding to the root of the tree. In our example, since net 3 is an FOS we have sensitized paths only until reaching its FOB nets.

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 by net $m_1 \rightarrow m_2$), to extend the sensitized path. To obtain $\boldsymbol{tc}_2$ we should sensitize all nets belonging to the set $D(m_1 \rightarrow m_2) - D(n)$ by intersecting their values in $\boldsymbol{d}(\boldsymbol{tc}_1)$ with $\boldsymbol{D}/\overline{\boldsymbol{D}}$. If any empty intersection results, then the sensitized path cannot be extended through net m_2 and alternate paths should be investigated. Note that this step implicitly performs the equivalent of the X-path check [11] while setting up the gate outputs that should be sensitized. As stated earlier, we would then construct $\boldsymbol{d}(\boldsymbol{tc}_2)$. If contradictions occur while constructing $\boldsymbol{d}(\boldsymbol{tc}_2)$, 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.

There are many strategies to select a FOB to extend the sensitized path. We use the observability measure introduced in COP [3].

A short description of this measure is given in Appendix B. In Table 7 we give the observability values according to COP for the circuit shown in Fig. 4. Since net $3 \rightarrow 103$ for the circuit shown in Fig. 4 has the highest observability among the observabilities of the three FOBs of net 3, we extend the sensitized path in our example through this branch. We use $D(3 \rightarrow 103) - D(3) = \{103\}$ so that net 103 has the value $\boldsymbol{D}$ in $\boldsymbol{tc}_2$. In the resulting $\boldsymbol{d}(\boldsymbol{tc}_2)$ shown below we have sensitized paths up to the FOB nets of net 112.

6	110	111	112	113	114	115	116
1	$\boldsymbol{0/D}$	$\boldsymbol{D}$	$\boldsymbol{D}$	$\boldsymbol{1/D}$	$\boldsymbol{1/\overline{D}}$	$\boldsymbol{1/\overline{D}}$	$\boldsymbol{1/\overline{D}}$

The process of extending the sensitized path by selecting a FOB net, constructing a $\boldsymbol{tc}_k$ and its corresponding $\boldsymbol{d}(\boldsymbol{tc}_k)$, continues until we reach a PO and have constructed $\boldsymbol{T}_f(\boldsymbol{p}_i)$. If contradictions occur, then alternate paths should be investigated. If all possible paths give contradictions, then no test exists. Note that all possible single paths need not be explicitly investigated to arrive at this conclusion. Proceeding with our example, we extend the sensitized path through the net $112 \rightarrow 114$ since observability of this branch is the highest among the observabilities of all the three branches. Since $D(112 \rightarrow 114) - D(112) = \{114, 116\}$, the $\boldsymbol{tc}_3$ shown below results.

114	116
$\boldsymbol{\overline{D}}$	$\boldsymbol{\overline{D}}$

The $\boldsymbol{d}(\boldsymbol{tc}_3)$ constructed from $\boldsymbol{tc}_3$ is shown below:

10	11	113	114	116	117	118
1	**1**	**1**	$\boldsymbol{\overline{D}}$	$\boldsymbol{\overline{D}}$	**1**	$\boldsymbol{D}$

Net number	Observability	Net number	Observability
1	0.262234	105	0.065250
2	0.048937	106	0.173999
3	0.049522	107	0.182308
4	0.494403	108	0.182308
5	0.048937	109	0.486018
6	0.017738	110	0.196096
7	0.025637	111	0.261461
8	0.025637	112	0.522923
9	0.289910	113	0.276087
10	0.260536	114	0.512073
11	0.486018	115	0.512073
101	0.032625	116	0.974560
102	0.032625	117	1.000000
103	0.035475	118	1.000000
104	0.065250		

Table 7: Observability for the circuit in Fig. 4.

We now have a sensitized path (say $\boldsymbol{p}_1$) from 3_f to a PO, and thus $\boldsymbol{d}(\boldsymbol{tc}_3)$ is $\boldsymbol{T}_f(\boldsymbol{p}_1)$. $\boldsymbol{T}_f(\boldsymbol{p}_i)$ represents all the constraints that *must* be imposed to sensitize path $\boldsymbol{p}_i$. 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 $\boldsymbol{p}_i$. We define the output nets of such gates as **variant** nets. If a net is not variant it is defined to be **invariant**. In our example the only variant net with respect to $\boldsymbol{T}_f(\boldsymbol{p}_1)$ is net 112.

If there are no variant nets in $\boldsymbol{T}_f(\boldsymbol{p}_i)$, then we have already obtained a test for the fault. Otherwise the enumeration phase must be invoked to determine a test.

3.4 Enumeration Phase

The goal of this phase is to obtain a test by specifying the unassigned PIs in $\boldsymbol{T}_f(\boldsymbol{p}_i)$ such that all nets are invariant and have values that are subsets of their corresponding values in $\boldsymbol{T}_f(\boldsymbol{p}_i)$.

We choose an unassigned PI I_{l_1} in $\boldsymbol{T}_f(\boldsymbol{p}_i)$ and assign a logic value (0 or 1) to it, thereby creating a new test cube which we denote by $\boldsymbol{tc}_f(\boldsymbol{p}_i, \mathbf{1})$. Now we find its corresponding deterministic test cube $\boldsymbol{d}(\boldsymbol{tc}_f(\boldsymbol{p}_i, \mathbf{1}))$ and update its list of variant nets (note that new variant nets may be created). However if $\boldsymbol{d}(\boldsymbol{tc}_f(\boldsymbol{p}_i, \mathbf{1}))$ cannot be obtained due to some contradiction, then we complement the entry for I_{l_1} in $\boldsymbol{tc}_f(\boldsymbol{p}_i, \mathbf{1})$ and construct its corresponding $\boldsymbol{d}(\boldsymbol{tc}_f(\boldsymbol{p}_i, \mathbf{1}))$. If this also leads to a contradiction, then there exists no test corresponding to $\boldsymbol{T}_f(\boldsymbol{p}_i)$. If we are successful in constructing $\boldsymbol{d}(\boldsymbol{tc}_f(\boldsymbol{p}_i, \mathbf{1}))$ we now assign a logic value to some other unassigned PI I_{l_2}, thereby creating $\boldsymbol{tc}_f(\boldsymbol{p}_i, \mathbf{2})$. As before, we must construct $\boldsymbol{d}(\boldsymbol{tc}_f(\boldsymbol{p}_i, \mathbf{2}))$ and update its list of variant nets. This procedure is continued and we traverse the decision tree, in a manner analogous to PODEM [11], until one of the following two conditions occurs:

- The list of variant nets corresponding to some $\boldsymbol{d}(\boldsymbol{tc}_f(\boldsymbol{p}_i, \boldsymbol{j}))$ becomes empty. This indicates the values of the PIs in $\boldsymbol{d}(\boldsymbol{tc}_f(\boldsymbol{p}_i, \boldsymbol{j}))$ represent test(s) for the fault.

- The decision tree is exhausted, i.e., no test exists.

For the sake of completeness we denote $\boldsymbol{T_f(p_i)}$ as $\boldsymbol{d(tc_f(p_i, \mathbf{o}))}$.

We now continue with our example for the fault net 3 $s-a-0$ in the circuit of Fig. 4 Thus, the algorithm must assign logic values to unassigned PIs in order to construct a test. There are many strategies to select such a PI. In this implementation of SIMPLE this selection is made based on the controllability measure proposed in SCOAP [12]. A short description of how to calculate this measure is given in Appendix B.

Our initial objective is to make net 110 an invariant net, which corresponds to setting one of the inputs of the gate whose output is net 110 to the value 0. A PI assignment which has a good chance of helping to achieve this objective is selected using a backtrace procedure. The description of this procedure is taken from [6]. 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 AND/NAND, 1 for OR/NOR), 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.

In Table 8 we give the controllability values for 0 and 1 obtained using SCOAP [12] for all the nets in the circuit of Fig. 4. Following the backtrace procedure described above, we set net 4 to 1, obtaining $\boldsymbol{tc_f(p_i, \mathbf{1})}$:

Net number	CY(0)	CY(1)	Net number	CY(0)	CY(1)
1	1	1	105	4	2
2	1	1	106	9	3
3	1	1	107	11	2
4	1	1	108	11	2
5	1	1	109	1	1
6	1	1	110	12	5
7	1	1	111	15	4
8	1	1	112	17	2
9	1	1	113	19	2
10	1	1	114	4	2
11	1	1	115	4	2
101	2	3	116	5	5
102	2	3	117	4	2
103	2	3	118	8	6
104	4	2	G_f	1	1

Table 8: Controllability for the circuit in Fig. 4.

4
1

We now obtain $\boldsymbol{d}(\boldsymbol{tc}_f(\boldsymbol{p}_i, \mathbf{1}))$: which is shown below

102	105	106	107	108	109	115
0	**0/1**	**$0/1/D$**	**$0/1/D$**	**$0/1/D$**	**0**	**1**

However, net 110 is still a variant net in $\boldsymbol{d}(\boldsymbol{tc}_f(\boldsymbol{p}_i, \mathbf{1}))$, so the backtrace procedure starts again at net 110, and sets PI 5 to 0. Thus $\boldsymbol{tc}_f(\boldsymbol{p}_i, \mathbf{1})$ is

5
0

and $\boldsymbol{d}(\boldsymbol{tc}_f(\boldsymbol{p}_i, \mathbf{1}))$ is

105
0

We need to continue this PI assignment since net 110 is still a variant net in $\boldsymbol{d}(\boldsymbol{tc}_f(\boldsymbol{p}_i, \mathbf{2}))$. By continuing with the PI assignment procedure, nets 1, 2, and 8 are set to 0 before a deterministic test cube with no variant nets is constructed. Thus, the following test has been constructed:

1	2	3	4	5	6	7	8	9	10	11
0	**0**	**1**	**1**	**0**	**1**	**0/1**	**0**	**0**	**1**	**1**

4 Parallel Version

In this section we describe the approach used to parallelize our sequential implementation of SIMPLE. Assume that there are $n = 2^k$ processors available in the parallel computer being used during simulation, where k is the number of PIs that will be assigned in the enumeration phase.

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.

In the enumeration phase we must assign logic value to the PIs. In the sequential implementation this assignment is done one PI at a time; in the parallel version we assign logic values simultaneously to k PIs. Thus, 2^k instances are created, and each instance is given to one processor which executes the sequential version of the algorithm. The selection of the k PIs is guided by the controllability measure described in Appendix B.1.

5 Simulation Results

Circuit Name	Cardinality of $\mathcal{L}_r$	Undetectable faults in $\mathcal{L}_r$	Average Time per fault (sec.)	Maximum time (sec.)	Fault coverage (%)
C432	518	4	1.894	339.26	99.23
C499	758	8	0.227	0.39	98.94
C880	940	0	0.299	0.41	100.00
C1355	1574	8	0.581	0.73	99.49
C1908	1879	9	0.653	1.35	99.52
C2670	2641	116	7.521	17466.20	95.61
C3540	3428	137	7.297	14237.89	96.00
C5315	5248	49	5.262	2177.70	99.06
C6288	7744	34	43.93	9878.39	99.56
C7552	7438	143	47.19	10087.19	98.08

Table 9: Experimental results for the ISCAS '85 Benchmark Circuits.

In this section we give the simulation results for the ISCAS '85 benchmark circuits. Table 9 summarizes the results achieved on a Sun/Sparc workstation by our implementation of the sequential version of SIMPLE. In order to obtain statistics for all of the faults, we attempt to find tests for all the faults in $\mathcal{L}_r$. (Normally, fault simulation is used in conjunction with ATPG. As tests are generated, additional faults that are detected are eliminated from consideration.) This program has found tests for all detectable faults and has identified all undetectable ones. For some circuits the number of undetectable faults given in Table 9 is different from the one given in [19, Table 1]. This is because the reduced fault set being considered may be different. In this table the average time per fault was obtained by dividing the total execution time by the cardinality of $\mathcal{L}_r$. The maximum time given in Table 9 is the maximum execution time, for any fault, taken by the program either to find a test for the fault or to identify that the fault is undetectable.

The simulation results achieved on a CM-5 (MIMD architecture) by our parallel implementation of SIMPLE for some faults are given in figures 7 to 18. In these figures we have plotted the time taken to find a test for these faults or to prove that no test exists when n processors are available. We also have plotted the quantity T_1/n where T_1 is the time taken by the algorithm when only 1 processor is available. Since we have a maximum number of 32 processors, the simulation results are given for $n = 1, 2, 4, 8, 16$, and 32.

6 Discussion

In this report we have given a short description of SIMPLE which is the central part of our ATPG system. We have described the measures used in the implementation of the sequential version of SIMPLE, and the approach used in our implementation of the parallel version of SIMPLE. We have presented the simulation result for the ISCAS '85 benchmark circuits.

These simulation results reveal that, even though our implementation of the sequential version of SIMPLE does not use any of the speed-up techniques proposed in [2, 20], this program found tests for all detectable faults or proved that such tests do not exist in a "reasonable" time. This is due to the strength of the 16-valued system coupled with our forward and backward implication rules. The inclusion of these speed-up techniques in our program would considerably reduce the search space. As a consequence, we expect a speed-up of at least two orders of magnitude in the time taken by the algorithm to find tests for hard-to-detect faults or to prove that no test exist for undetectable faults. It is well-known that the ATPG algorithms proposed in the literature do not lead themselves to an efficient parallel implementation. However, the simulation results obtained using our implementation of the parallel version of SIMPLE indicate that for hard-to-detect faults an almost linear speed-up is obtained by this implementation. The reasons for such a speed-up are:

(i) We have parallized only the enumeration phase that is responsible for more than 95% of the runtime of our algorithm in the sequential version of SIMPLE.

(ii) There are almost no communication among the processors.

(iii) No processor is idle when the program starts to run.

Our implementation does not use processors that become idle when the program is running. We expect an even better speed-up if we were to use the processors that became idle to further divide the search space among them. We remark here that any speed-up in the sequential version of SIMPLE would be reflected in the parallel version.

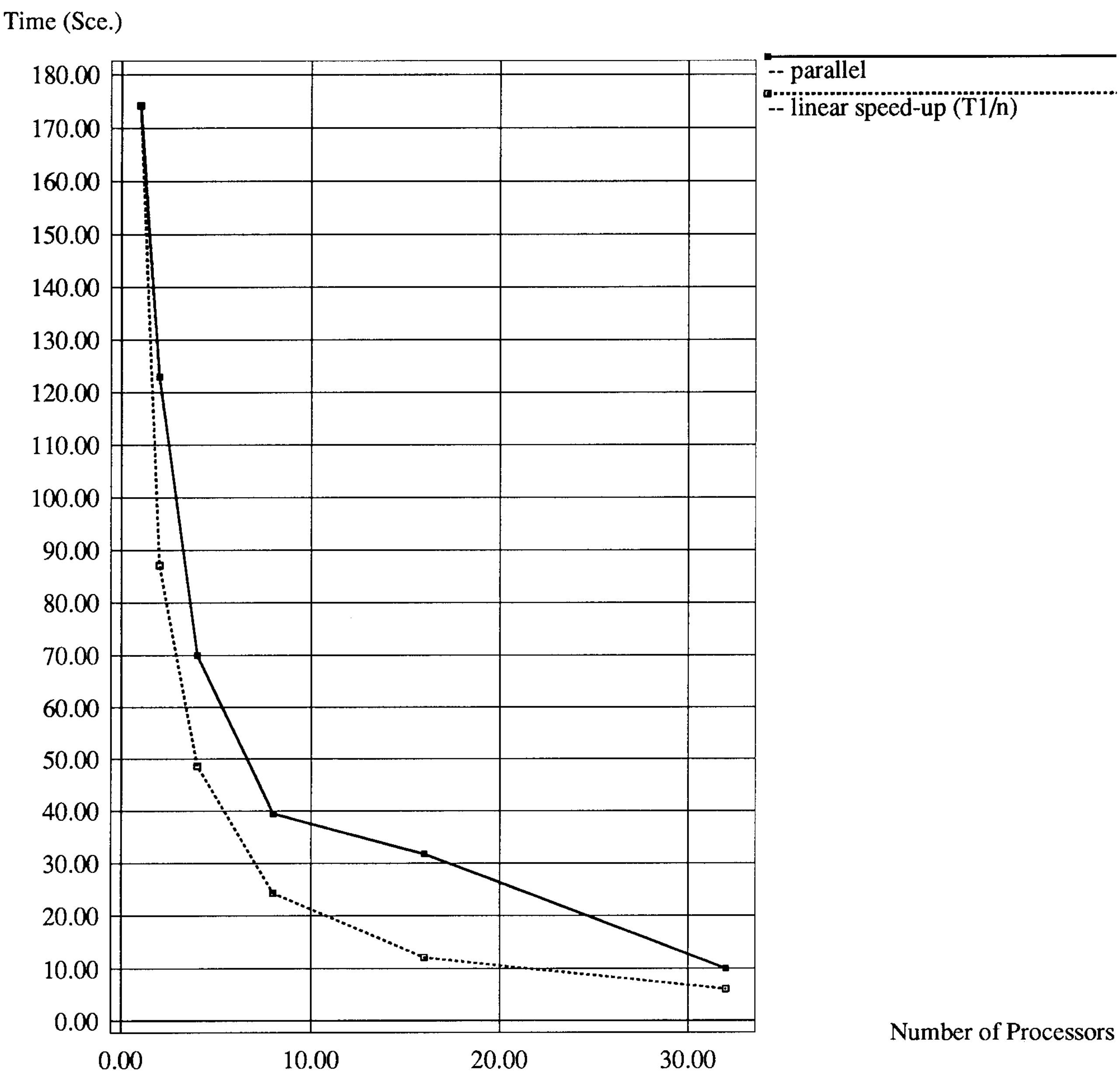


Figure 7: CM-5 Simulation Result for FOB net (167→246) s-a-1 in cc432.

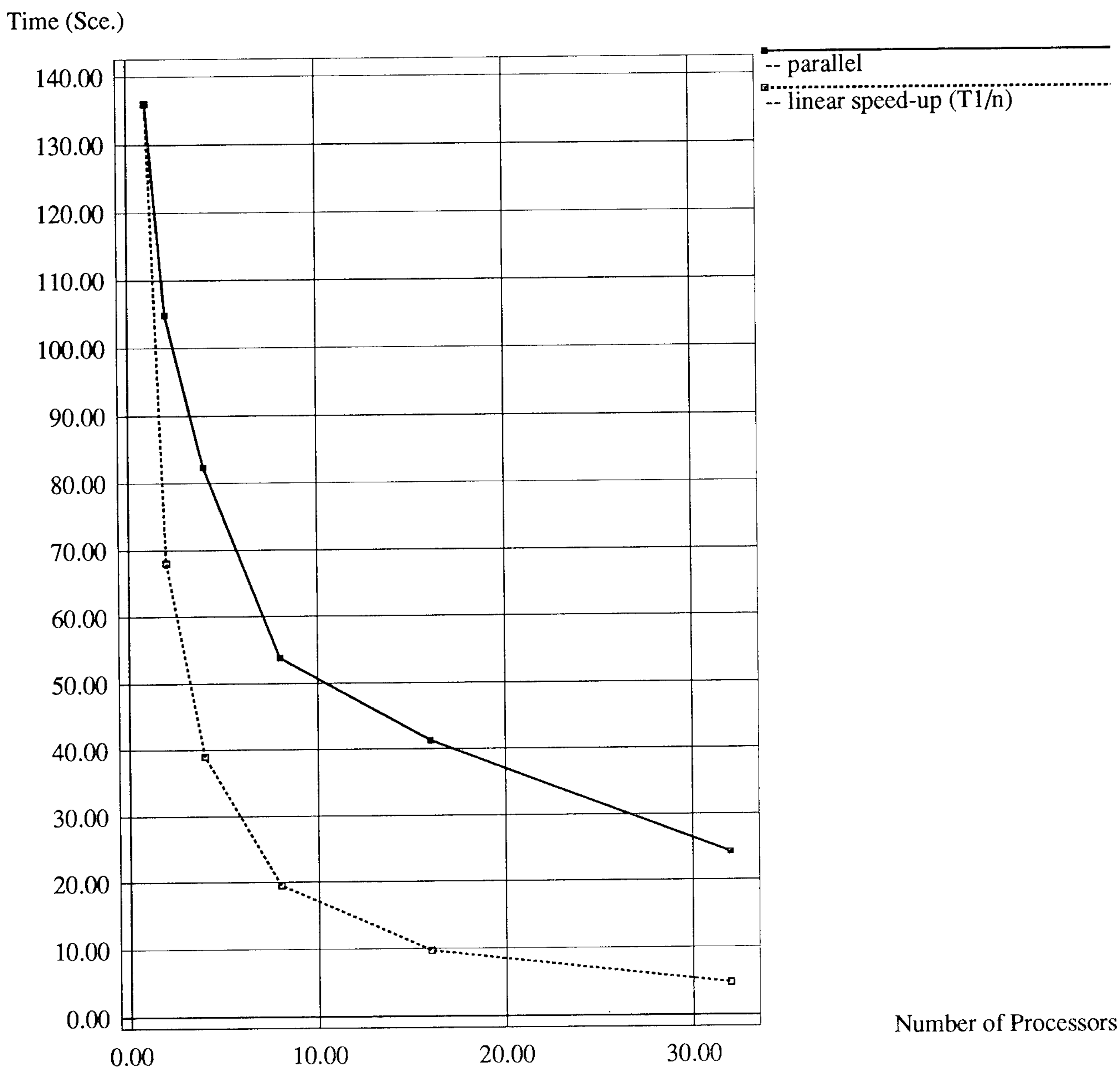


Figure 8: CM-5 Simulation Result for FOB net (216→246) s-a-1 in cc432.

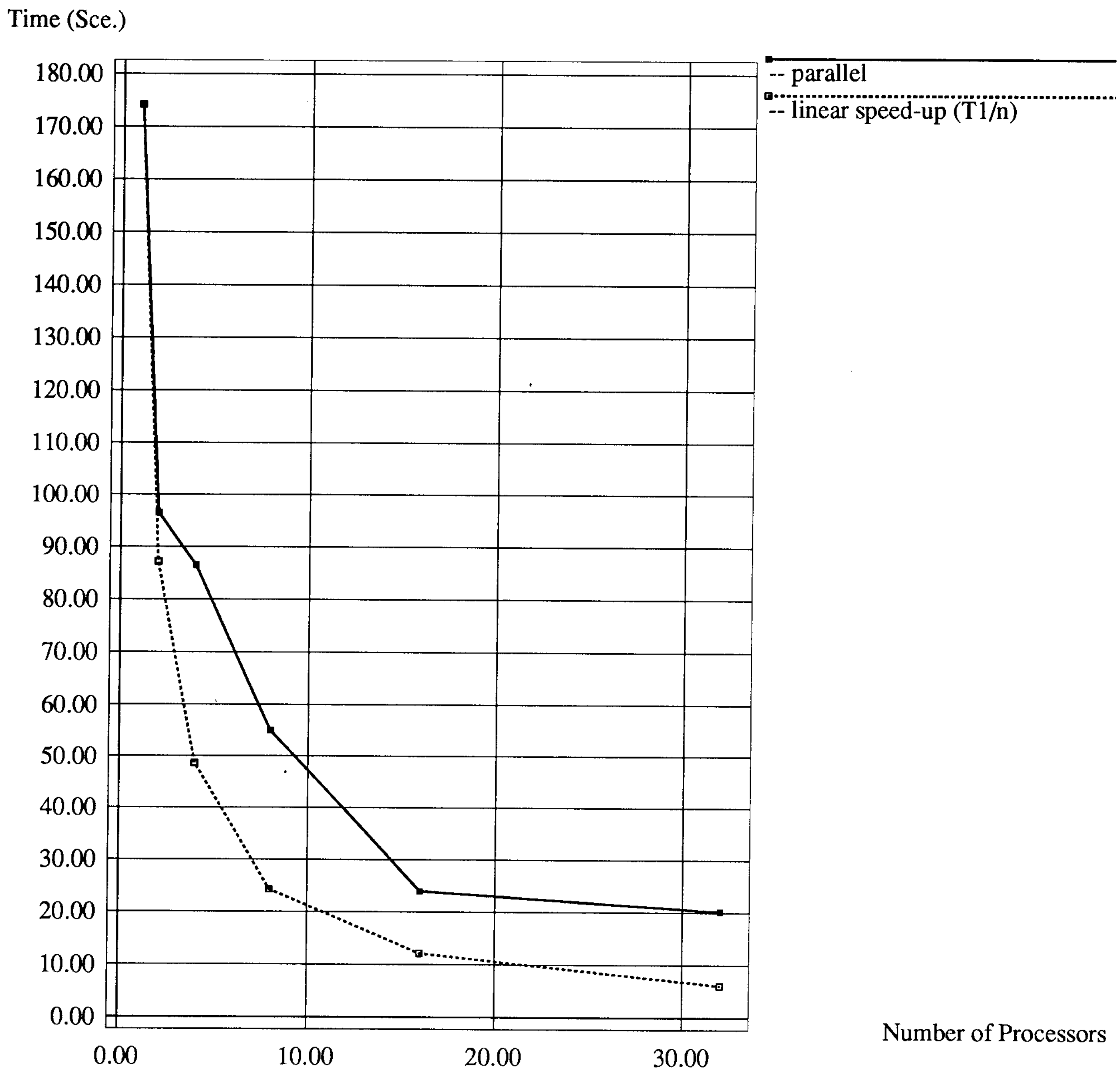


Figure 9: CM-5 Simulation Result for FOB net (237→246) *s-a-*1 in cc432.

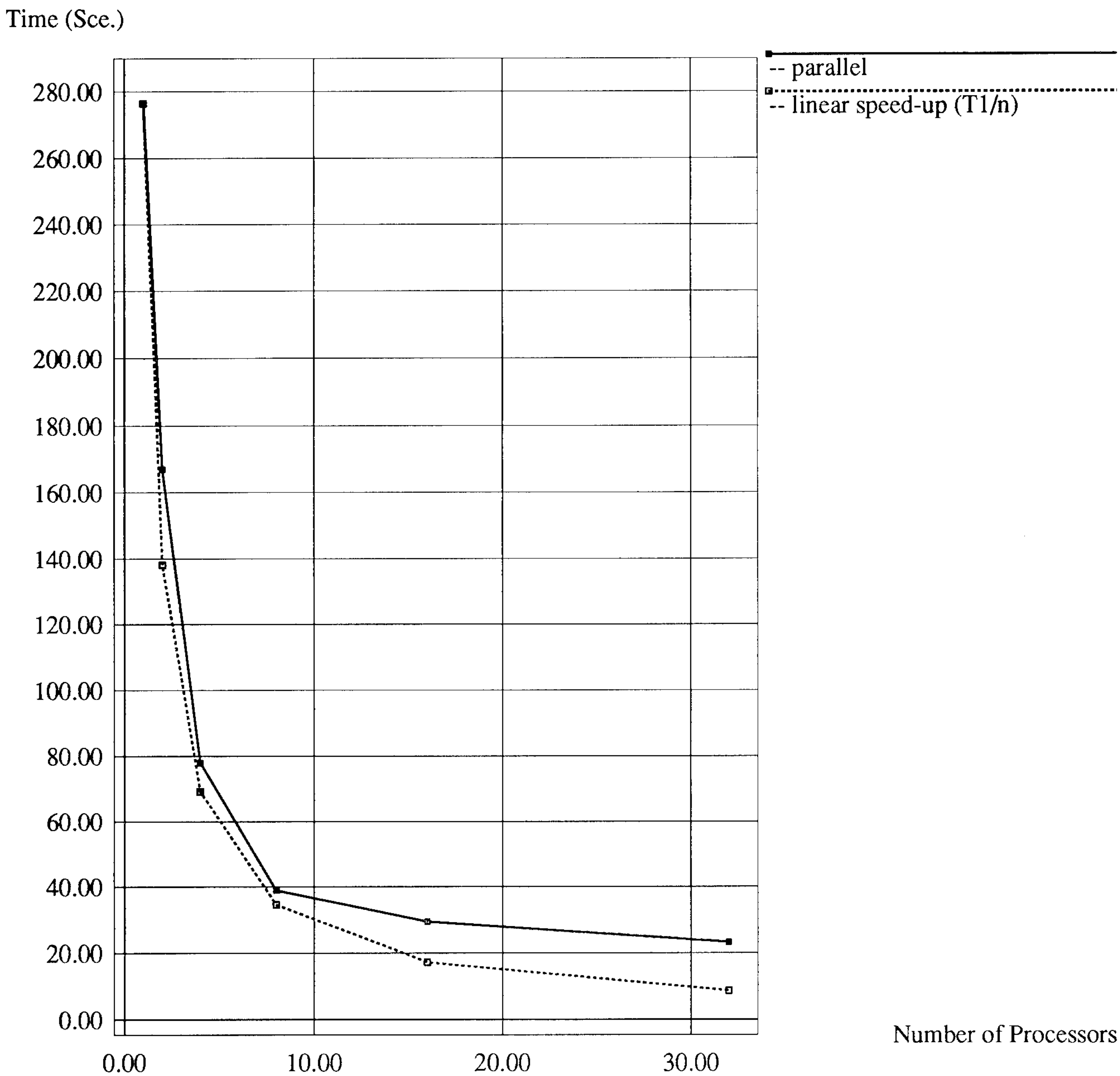


Figure 10: CM-5 Simulation Result for FOB net (1383→1632) *s-a*-1 in cc2670.

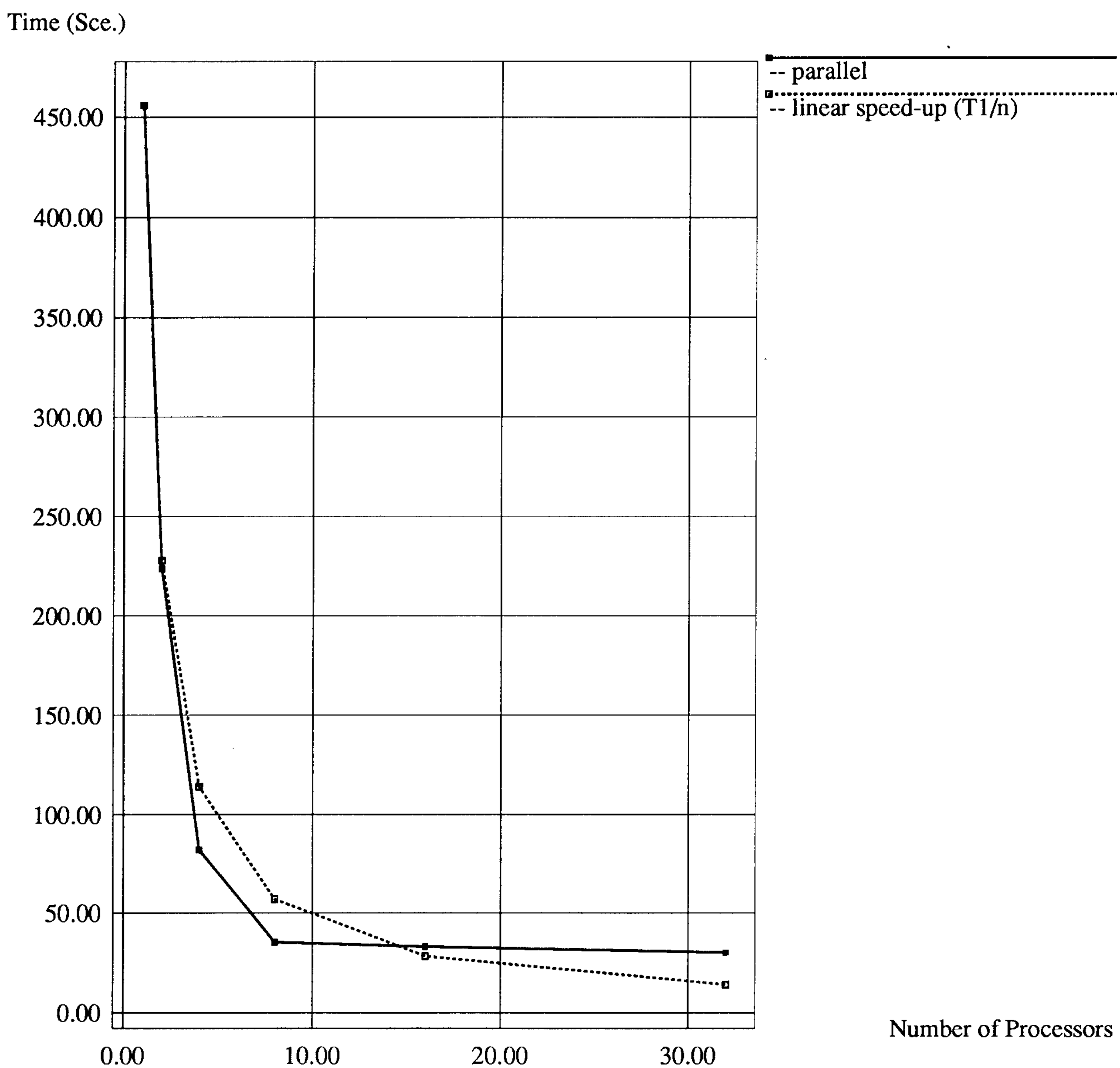


Figure 11: CM-5 Simulation Result for FOB net (1337→1633) *s-a-*1 in cc2670.

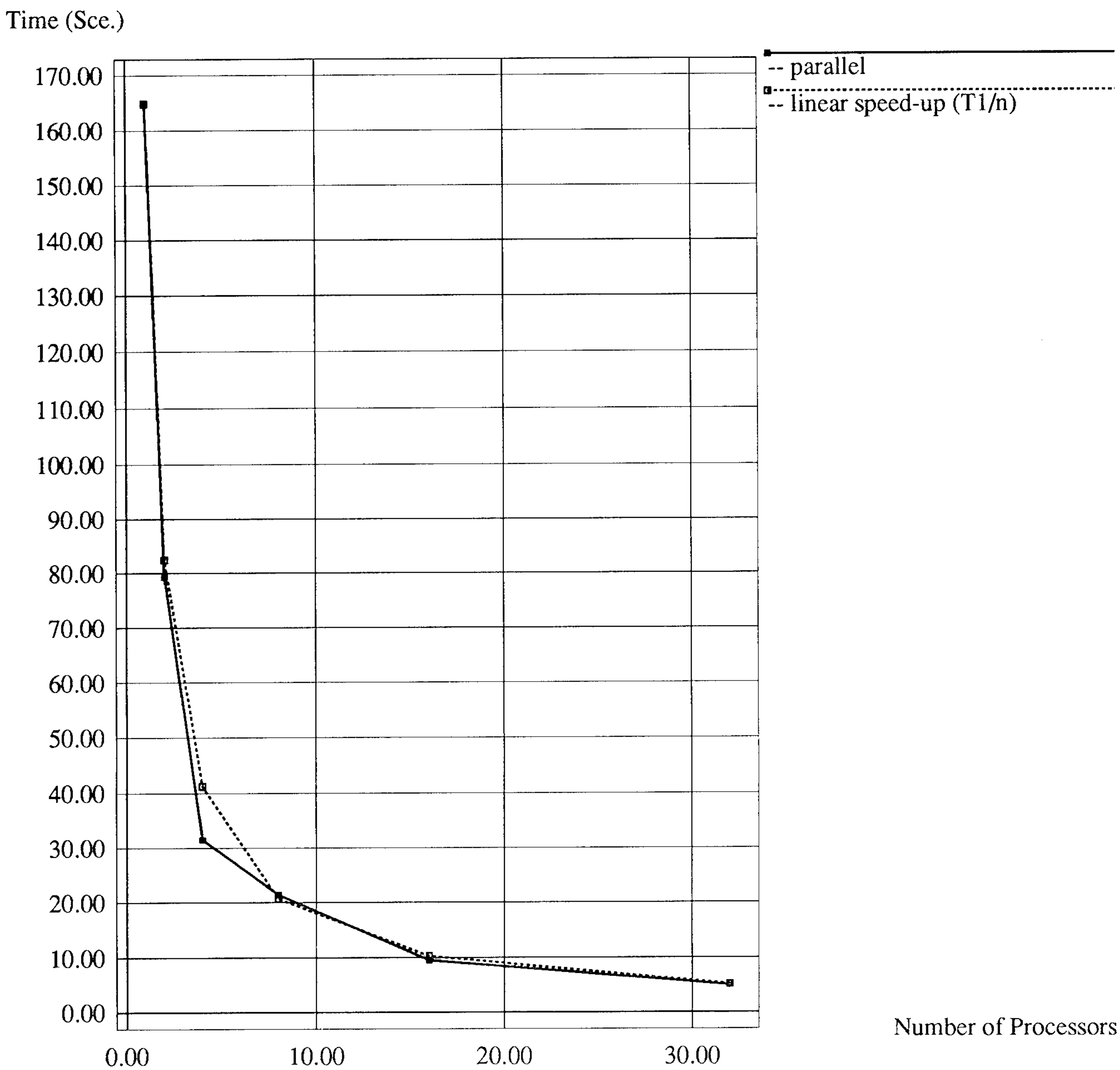


Figure 12: CM-5 Simulation Result for FOS net 1942 s-a-0 in cc2670.

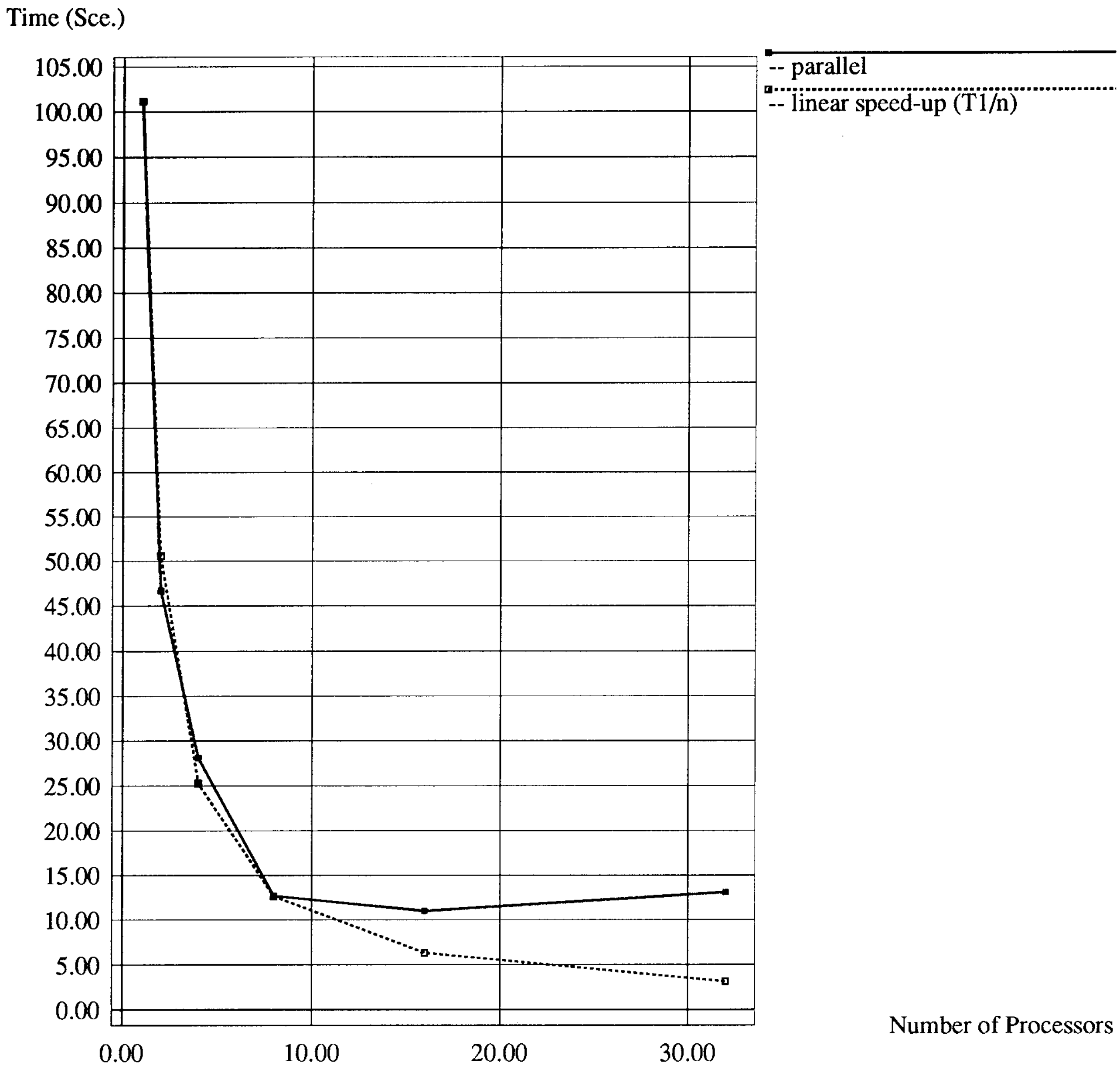


Figure 13: CM-5 Simulation Result for FOB net (137→1859) *s*-*a*-1 in cc5315.

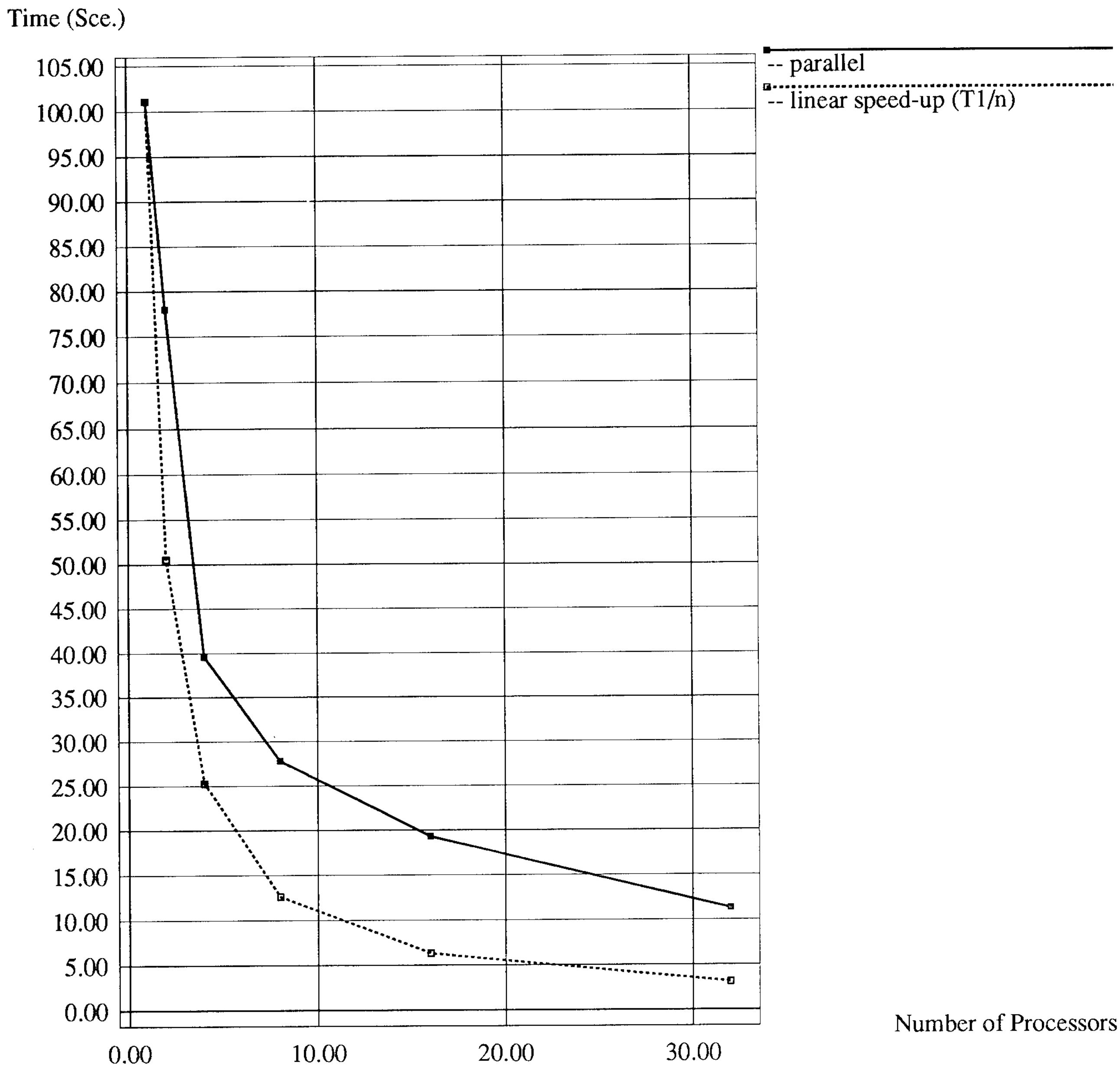


Figure 14: CM-5 Simulation Result for FOB net (1752→1859) s-a-1 in cc5315.

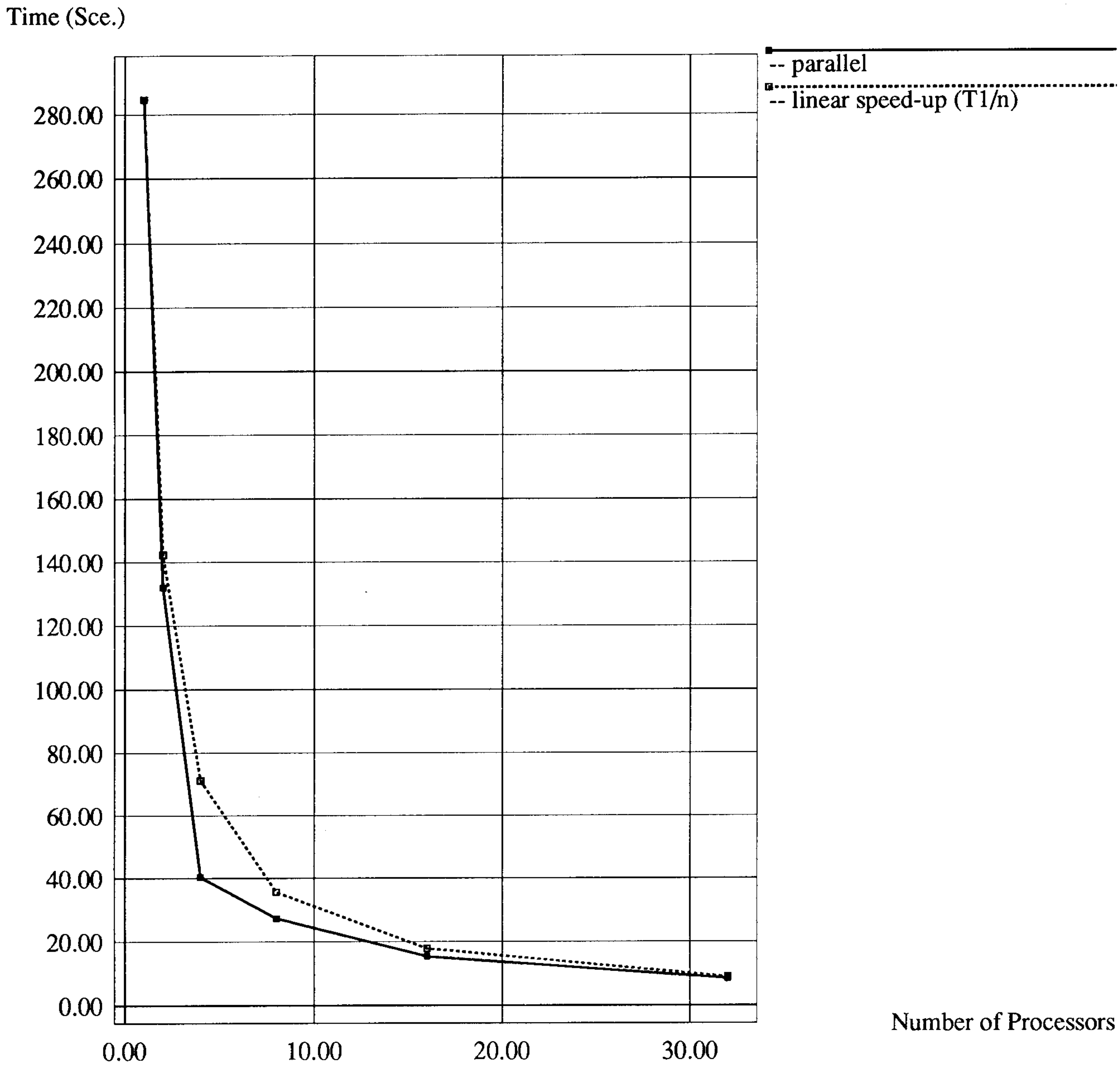


Figure 15: CM-5 Simulation Result for FOB net (2590→3055) *s-a-1* in cc5315.

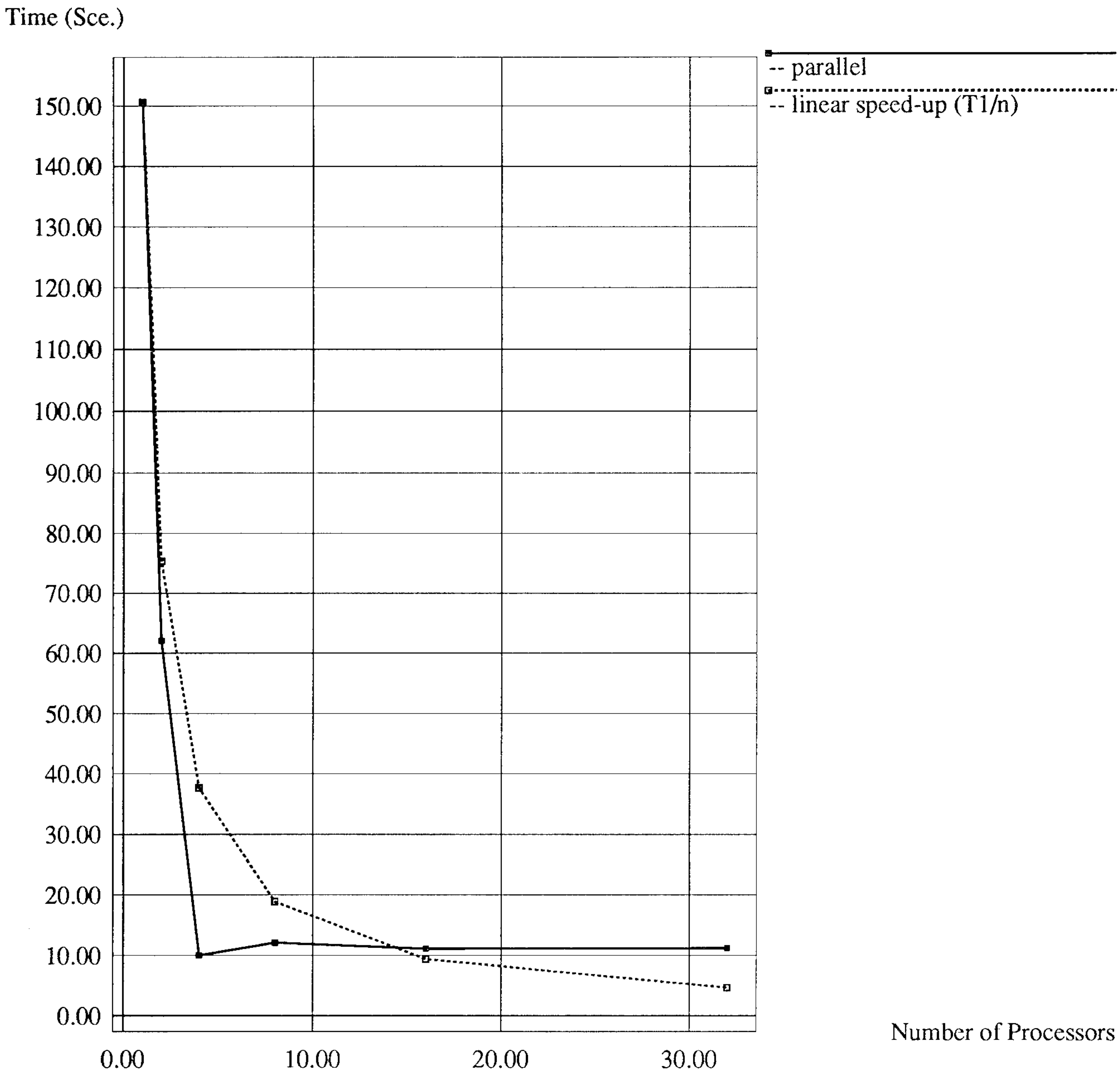


Figure 16: CM-5 Simulation Result for FOB net (2415→3137) *s-a-1* in cc7552.

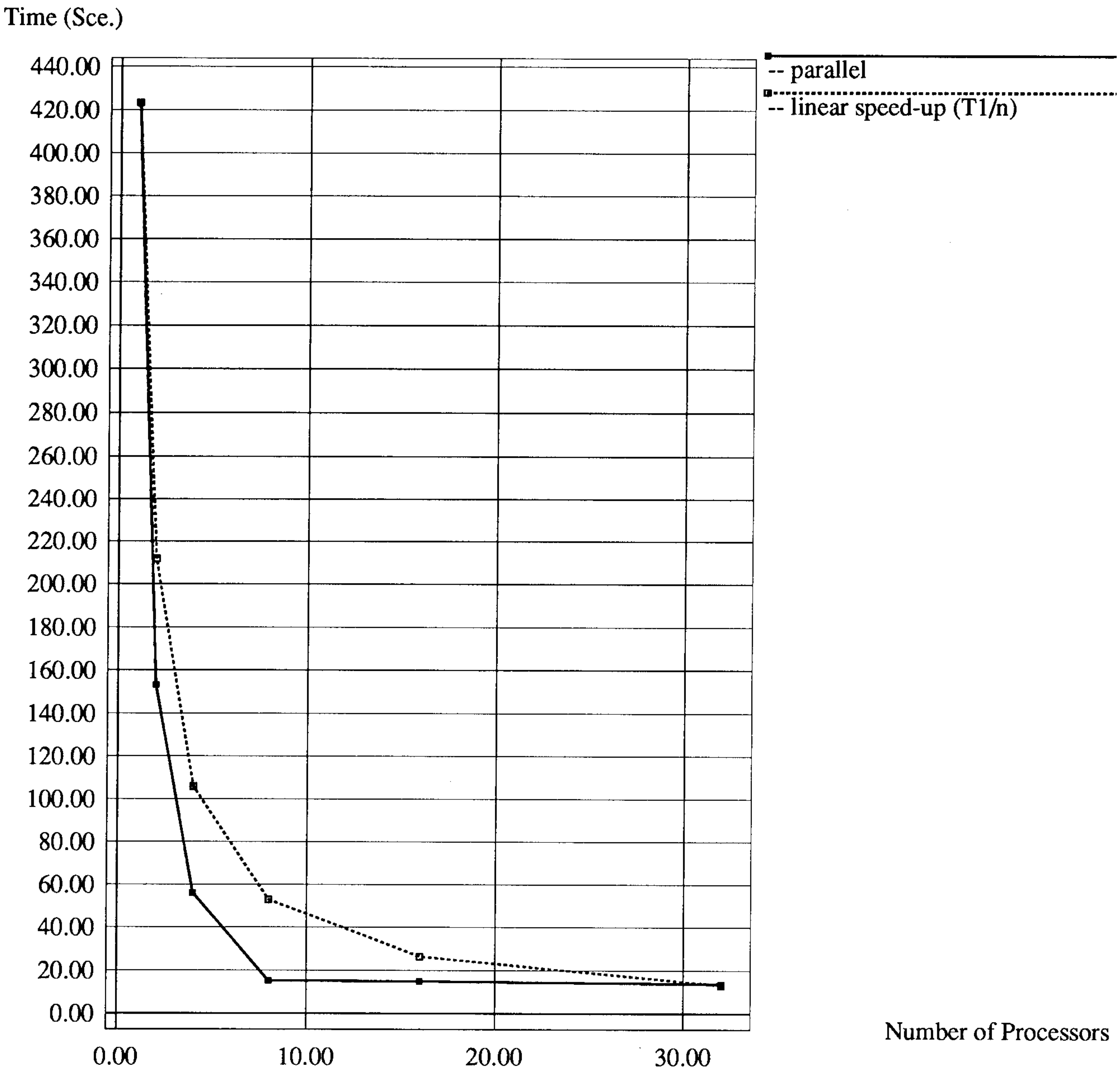


Figure 17: CM-5 Simulation Result for FOB net (2415→3142) s-a-1 in cc7552.

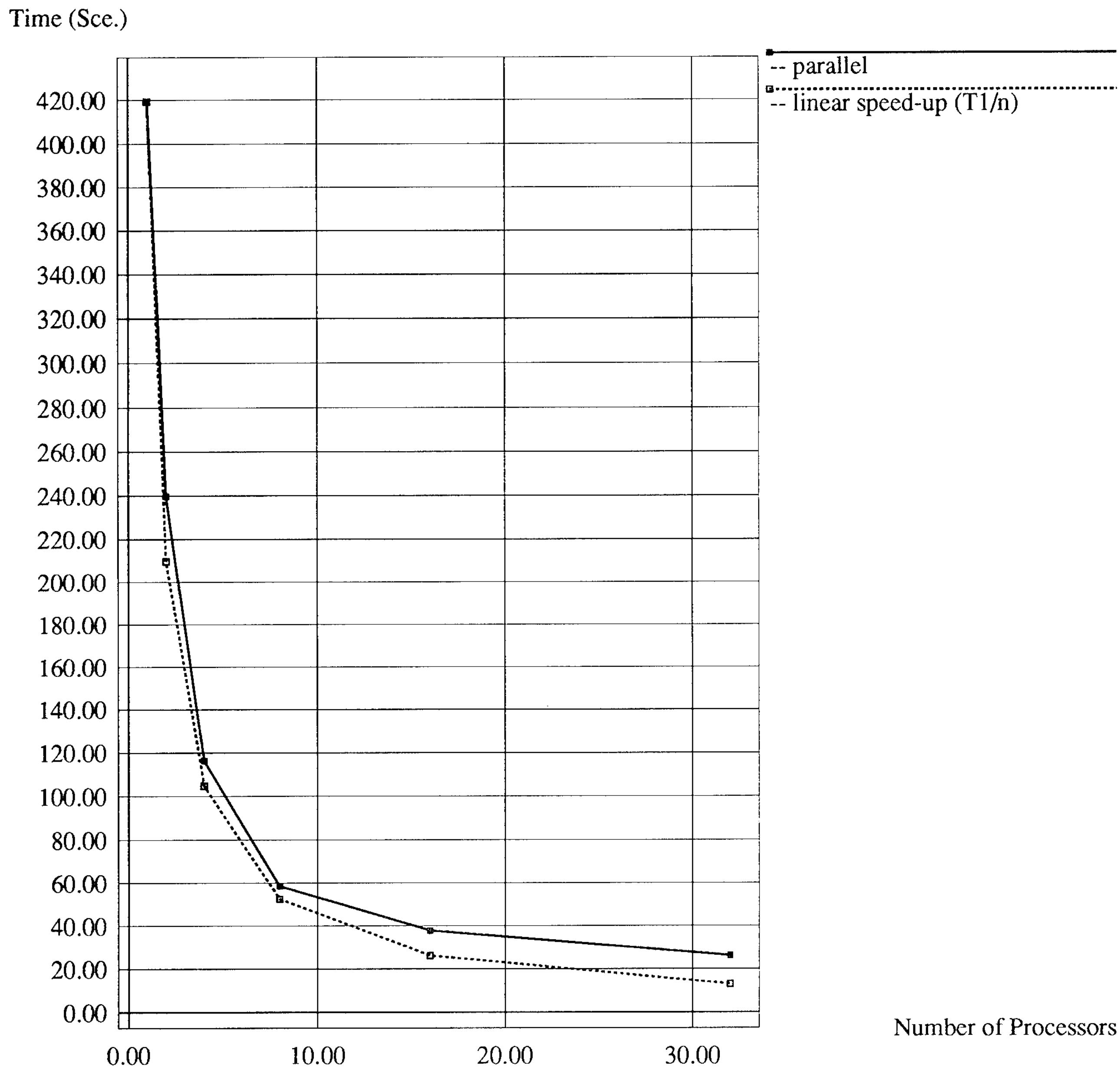


Figure 18: CM-5 Simulation Result for FOB net (3786→3866) s-a-1 in cc7552.

Appendix

A Construction of Deterministic Test Cubes

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

To construct $\boldsymbol{d(tc_1)}$ from $\boldsymbol{tc_1}$ we perform backward and forward implications of all nets whose values in $\boldsymbol{tc_1}$ are different from $\mathbf{0/1}$ and $\boldsymbol{0/1/D/\overline{D}}$ and all other nets whose values change during this implication process. In the general case, when we are constructing $\boldsymbol{d(tc_k)}$ from $\boldsymbol{tc_k}$, we start by considering the forward and backward implications of the nets whose values in $\boldsymbol{tc_k}$ are different from those in the last successfully constructed deterministic test cube. During the construction of $\boldsymbol{d(tc_k)}$ from $\boldsymbol{tc_k}$, 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 $\boldsymbol{d(tc_k)}$ cannot be constructed.

In order to obtain $\boldsymbol{d(tc_k)}$ 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.

A.1 Forward Implication

The process of forward implication of the values associated with every net is done with the help of Tables 3, 4, and 5. These tables are a generalization of the truth tables of the respective gates. For gates with more than two inputs the method adopted is similar to that used by Akers [1]. We view every gate as being constructed out of two-input gates and use the existing values at the inputs 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. 19. If the n-input gate is a NAND (NOR) gates, then $G_1, G_2, ..., G_{n-2}$ 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 associated with the original n-input gate.

Note that the three tables are sufficient because OR, NOR, and NAND functions can be derived by appropriately using Tables 3 and 4, and Tables 4 and 5 can be used to generate the XNOR function.

Suppose we are performing forward implications due to change(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. $L_N \subset L_O$ (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. $L_O \subset L_N$. 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. $L_O \not\subseteq L_N$ and $L_N \not\subseteq L_O$. Both forward and backward implications of the value $L_O \cap L_N$ at net m should be performed.

A.2 Backward Implication

The process of backward implication involves determining the changes required at the inputs of a gate in order to satisfy a requested change at the output. A change in

* / **	0	1	D	$\overline{D}$
0	$0/1/D/\overline{D}$	$\emptyset$	$\emptyset$	$\emptyset$
1	0	1	D	$\overline{D}$
D	$0/\overline{D}$	$\emptyset$	$1/D$	$\emptyset$
$\overline{D}$	$0/D$	$\emptyset$	$\emptyset$	$1/\overline{D}$

* Requested Output

** Existing value at one input

Table 10: Backward Implication for a 2-input AND gate.

the value of a net means that one or more of the possible values associated with the net has been deleted. In that sense 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 [1] we consider each multiple-input gate as a cascade of two-input gates. The backward implication rules for a two-input AND gate is shown in Table 10.

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 [1]. 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, Table 5 can be used for backward implication also. Thus Tables 4, 5, and 10 can be used to perform backward implication for any two-input gate. 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 to generate the new 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, $G_2, \ldots, G_{n-2}$ and G_{n-1}, with net numbers as shown in Fig. 19. Assume that the values at nets $1, 2, \ldots, n$ are $X_1, X_2, \ldots, X_n$ respectively. We first use forward implication of these values to compute $Y_1, Y_2, \ldots, Y_{n-2}$, the values of nets $n+1, n+2, \ldots, n+(n-2)$ 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 logic gate is associative, the order in which the inputs X_i are cascaded is irrelevant.

It is shown in [2] that the above procedure will stabilize in a single pass, unlike the approach followed in [1] which may require several passes.

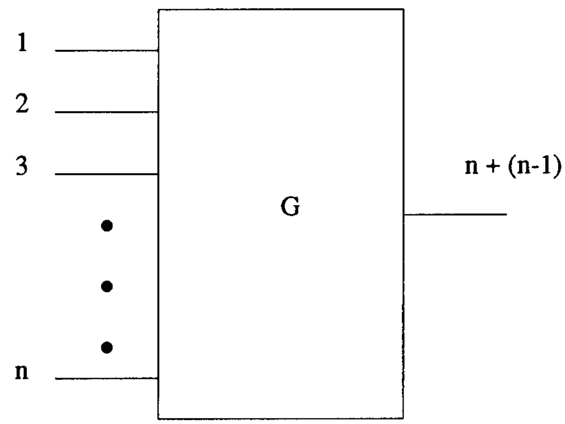
B Measure for Controllability and Observability

In this appendix we give a short description of the controllability and observability measure used in our implementation of SIMPLE.

The controllability measure used was that proposed in SCOAP [12], and the observability in COP [3]. The descriptions of these measures are taken from [6].

B.1 Controllability

With every net n SCOAP associates 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,



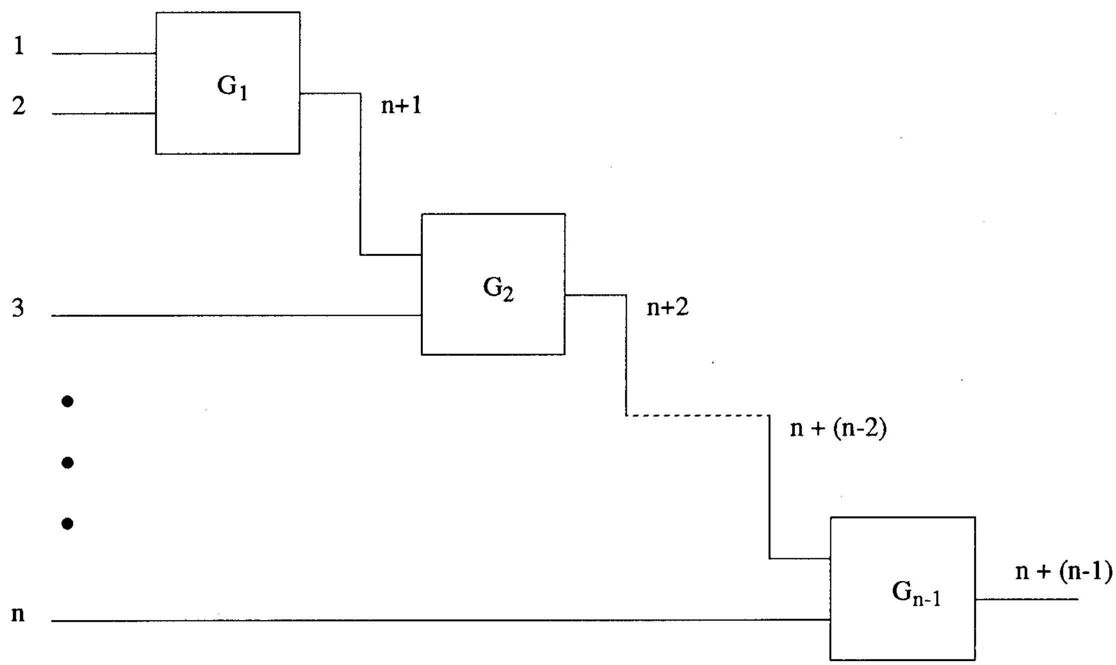


Figure 19: Gate decomposition.

gate type	$C^0(m)$	$C^1(m)$
AND	$1 + \min_{j\in\{1,2,\ldots,n\}}\{C^0(i_j)\}$	$1 + \sum_{j=1}^{n} C^1(i_j)$
OR	$1 + \sum_{j=1}^{n} C^0(i_j)$	$1 + \min_{j\in\{1,2,\ldots,n\}}\{C^1(i_j)\}$
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.

Table 11: Rules to calculate the controllability in SCOAP.

let G be a gate with n inputs nets $i_1, i_2, ..., i_n$, and output net m. Table 11 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.

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 $C^0(m_1) < C^0(m_2)$ ($C^1(m_1) < C^1(m_2)$) we say that m_1 is "easier" to control then net m_2 with respect to logic value 0 (1). Thus, this measure of controllability increases with the *difficulty of controlling* a net.

B.2 Observability

To define the observability measure introduced in COP we first need to define the controllability measure that it is based on. Both measures are based on a simplistic probabilistic approach. The description of these measures is taken from [6].

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

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

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

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

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

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 inputs $i_1, i_2, ..., i_{j-1}, i_{j+1}, ..., i_n$, sensitize the net m to a change in the input i_j. Then

$$OB(i_j) = OB(m) \times \sum_{B \in S_j} \prod_{\substack{\ell=1 \\ \ell \neq j}}^{n} p_\ell.$$

Finally, if net $m_1, m_2, ..., m_r$ are fanout branches corresponding to fanout stem m', then

$$OB(m') = 1 - \prod_{\ell=1}^{r} (1 - OB(m_\ell)).$$

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 increases with the *ease of observing* a net.

References

[1] Sheldon B. Akers, "A Logic System for Fault Test Generation," *IEEE Transactions on Computers*, vol. c-25, pp. 620–630, June 1976.

[2] A. M. Ali and C. R. P. Hartmann, "SIMPLE: A New Approach to Combinational Circuit Testing," Technical Report, Syracuse University, Syracuse, NY, 1989.

[3] 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.

[4] Franc Brglez and Hideo Fujiwara, "A Neutral Netlist of 10 Combinational Benchmark Circuits and a Target Translator in Fortran," *IEEE International Symposium on Circuits & Systems*, pp. 663–698, June 1985.

[5] Charles W. Cha, William E. Donath, and Füsun Özgüner, "9-V Algorithm for Test Pattern Generation of Combinational Digital Circuits," *IEEE Transactions on Computers*, vol. c-27, pp. 193–209, March 1978.

[6] 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.

[7] Wu-Teng Cheng, "Split Circuit Model for Test Generation," *Proceedings of the* 25^{th} *ACM/IEEE Design Automation Conference*, pp. 96–101, 1988.

[8] W.H. Debany, K.A. Kwiat, H.B. Dussault, M.J. Gorniak, A.R. Macera, and D.E. Daskiewich, "Fault Coverage Measurement for Digital Microcircuits," MIL-STD-883 Test Procedure 5012, US Air Force Rome Laboratory (RL/ERDA), Griffiss AFB, NY 13441, 18 December 1989 (Notice 11) and 27 July 1990 (Notice 12).

[9] H. Fujiwara and T. Shimono, "On the Acceleration of Test Generation Algorithms," *IEEE Transactions on Computers*, vol. c-32, pp. 1137–1144, December 1983.

[10] H. Fujiwara and S. Toida, "The Complexity of Fault Detection: An Approach to Design for Testability," *Proceedings of the* 12^{th} *International Symposium on Fault Tolerant Computing*, pp. 101–108, June 1982.

[11] Prabhakar Goel, "An Implicit Enumeration Algorithm to Generate Tests for Combinational Logic Circuits," *IEEE Transactions on Computers*, vol. c-30, pp. 215–222, March 1981.

[12] L. H. Goldstein and E. L. Thigpen, "SCOAP: Sandia Controllability/Observability Analysis Program," *Proceedings of the* 17^{th} *ACM/IEEE Design Automation Conference*, pp. 190–196, 1980.

[13] O. H. Ibarra and S. K. Sahni, "Polynomially Complete Fault Detection Problems," *IEEE Transactions on Computers*, vol. c-24, pp. 242–259, March 1975.

[14] Dong-Liang Jan and Kuo-Kuei Ho, "Translator for ISCAS '85 Netlist Format," *Private Communication*, June 1991.

[15] Tom Kirkland and M. Ray Mercer, "A Topological Search Algorithm for ATPG," *Proceedings of the* 24^{th} *ACM/IEEE Design Automation Conference*, pp. 502–508, 1987.

[16] P. Muth, "A Nine-Valued Circuit Model for Test Generation," *IEEE Transactions on Computers*, vol. c-25, pp. 630–636, June 1976.

[17] 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.

[18] J. P. Roth, W. G. Bouricius, and P. 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.

[19] 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.

[20] Michael H. Schulz, Erwin Trischler, and Thomas M. Sarfert, "Socrates—A Highly Efficient Automatic Test Pattern Generation System," *Proceedings of the IEEE International Test Conference*, pp. 1016–1026, 1987.

[21] R. Tarjan, "Finding Dominators in Directed Graphs," *SIAM Journal of Computing*, vol. 3, no. 11, pp. 62–89, 1974.

Rome Laboratory

Customer Satisfaction Survey

RL-TR-________

Please complete this survey, and mail to RL/IMPS, 26 Electronic Pky, Griffiss AFB NY 13441-4514. Your assessment and feedback regarding this technical report will allow Rome Laboratory to have a vehicle to continuously improve our methods of research, publication, and customer satisfaction. Your assistance is greatly appreciated.
Thank You

____

____

Organization Name:______________________________(Optional)

Organization POC: ______________________________(Optional)

Address:____

1. On a scale of 1 to 5 how would you rate the technology developed under this research?

5-Extremely Useful   1-Not Useful/Wasteful

Rating_____

Please use the space below to comment on your rating. Please suggest improvements. Use the back of this sheet if necessary.

2. Do any specific areas of the report stand out as exceptional?

Yes___ No______

If yes, please identify the area(s), and comment on what aspects make them "stand out."

3. Do any specific areas of the report stand out as inferior?

Yes___ No___

If yes, please identify the area(s), and comment on what aspects make them "stand out."

4. Please utilize the space below to comment on any other aspects of the report. Comments on both technical content and reporting format are desired.

☆U.S. GOVERNMENT PRINTING OFFICE: 1995-604-027-28029