A Parallel Implementation of Fault Simulation on a Cluster of Workstations

Except where reference is made to the work of others, the work described in this thesis is my own or was done in collaboration with my advisory committee. This thesis does not include proprietary or classified information.

Kyunghwan Han

Certificate of Approval:

Fa Foster Dai
Associate Professor
Electrical and Computer Engineering

Soo-Young Lee, Chair
Professor
Electrical and Computer Engineering

Chwan-Hwa John Wu
Professor
Electrical and Computer Engineering

Stephen L. McFarland
Acting Dean
Graduate School

A Parallel Implementation of Fault Simulation on a Cluster of Workstations

Kyunghwan Han

A Thesis

Submitted to

the Graduate Faculty of

Auburn University

in Partial Fulfillment of the

Requirements for the

Degree of

Master of Science

Auburn, Alabama
December 15, 2006

A Parallel Implementation of Fault Simulation on a Cluster of Workstations

Kyunghwan Han

Permission is granted to Auburn University to make copies of this thesis at its discretion, upon the request of individuals or institutions and at their expense. The author reserves all publication rights.

Signature of Author

Date of Graduation

Thesis Abstract

A Parallel Implementation of Fault Simulation on a Cluster of Workstations

Kyunghwan Han

Master of Science, December 15, 2006
(B.S., Sungkyunkwan University, 2003)

81 Typed Pages

Directed by Soo-Young Lee

Parallel simulation on a cluster workstations is one method by which fault simulation time for large circuits can be reduced significantly. To get near-linear speedups from parallel processing, parallelization methods should result in an even computational load distribution among processors in a cluster workstations. Fault simulation can be parallelized by partitioning fault list, the test vector or both. In the thesis, parallel fault simulation algorithm called PAUSIM has been developed. This algorithm consists of logic simulation and two steps of fault simulation for sequential logic circuits. Compared to the other algorithms, PAUSIM-CY avoids redundant work by a judicious task decomposition. Also, it adopts a cyclic fault partitioning method based on the LOG partitioning and local redistribution, resulting in a well-balanced load distribution. The parallel implementations were done using the MPI library on a cluster of workstations. The results show a significant speed-up by PAUSIM-CY over other existing parallel algorithms.

Style manual or journal used Journal of Approximation Theory (together with the style known as “aums”). Bibliograpy follows van Leunen’s *A Handbook for Scholars.*

Computer software used The document preparation package TEX (specifically LATEX) together with the departmental style-file `aums.sty`.

Table of Contents

List of Figures viii

List of Tables xi

1 Introduction 1
- 1.1 Background . . . 1
- 1.2 Review of Previous Work . . . 3
- 1.3 Problem Definition . . . 4
- 1.4 Objective . . . 4

2 Circuit Testing 5
- 2.1 Fault Modeling . . . 5
- 2.2 Test Pattern Environment . . . 9
- 2.3 Sequential Methods . . . 10
  - 2.3.1 Word-parallel Fault Simulation . . . 11
  - 2.3.2 Deductive Fault Simulation . . . 12
  - 2.3.3 Concurrent Fault Simulation . . . 13
  - 2.3.4 Differential Fault Simulation . . . 15
  - 2.3.5 PROOFS . . . 17
  - 2.3.6 HOPE . . . 20
  - 2.3.7 PARIS and PSF . . . 21
- 2.4 Parallel Methods . . . 21
  - 2.4.1 Circuit Parallelism . . . 22
  - 2.4.2 Algorithmic Parallelism . . . 22
  - 2.4.3 Fault Parallelism . . . 23
  - 2.4.4 Pattern Parallelism . . . 25
  - 2.4.5 Fault and Pattern Parallelism . . . 26
- 2.5 AUSIM . . . 28
  - 2.5.1 Program Configuration . . . 28
  - 2.5.2 Algorithm Analysis . . . 29

3 Parallelization 34
- 3.1 PAUSIM-BL . . . 34
  - 3.1.1 Task Decomposition . . . 34
  - 3.1.2 Fault Partitioning . . . 36
  - 3.1.3 Test Vector Partitioning . . . 36
  - 3.1.4 Procedure . . . 37

3.2 PAUSIM-CY 41
3.2.1 Load Balancing 41
3.2.2 Procedures 41
3.3 Implementation 43

4 Performance 47
4.1 Experimental Environment 47
4.2 Results 48

5 Conclusion 67

Bibliography 68

List of Figures

1.1 Typical fault simulator. 2

2.1 Definition of fan-out stem. 5

2.2 Definition of fault. 6

2.3 An Example of a single stuck-at fault. 7

2.4 An example of an level order. 8

2.5 An example of an output cone. 9

2.6 An example of parallel fault simulation. 12

2.7 An example of deductive fault simulation. 13

2.8 Fault-lists in concurrent fault simulation. 14

2.9 An example of differential fault simulation 16

2.10 PROOFS algorithm. 18

2.11 An example of PROOFS fault simulation. 19

2.12 Test sequence partitioning in SPITFIRE-0 25

2.13 Partitioning in SPITFIRE-1 26

2.14 Partitioning in SPITFIRE-2 28

2.15 Sequential structure of AUSIM 30

2.16 Data Structure of AUSIM 31

2.17 An example of gate evaluation of AUSIM. 32

3.1 Task decomposition in PAUSIM-BL 35

3.2 Fault partitioning in PAUSIM-BL for 4 processors where each fault group is distinguished by a different grey-scale 37

3.3 Test vector set partitioning 38

3.4 Flowchart of the master processor in PAUSIM-BL 39

3.5 Flowchart of slave processors in PAUSIM-BL 40

3.6 Example of fault partitioning for s27 benchmark circuit for 5 processors . 42

3.7 Task decomposition in PAUSIM-CY1 43

3.8 Communication among 4 processors in logic simulation in PAUSIM-CY1. V: Test vector, O: Logic simulation result 45

3.9 Communication among 4 processors in the first step of fault simulation in PAUSIM-CY1. F: Fault, U: Undetected fault, D: Undetected fault 45

3.10 Communication among 4 processors in the second step of fault simulation in PAUSIM-CY1 46

4.1 Execution time for 4 processors - small circuit 50

4.2 Execution time for 4 processors - large circuit 50

4.3 Execution time for 8 processors - small circuit 51

4.4 Execution time for 8 processors - large circuit 51

4.5 Execution time for 16 processors - small circuit 52

4.6 Execution time for 16 processors - large circuit 52

4.7 Execution times for PAUSIM-SF 54

4.8 Execution times for PAUSIM-BL 54

4.9 Execution times for PAUSIM-CY0 55

4.10 Execution times for PAUSIM-CY1 55

4.11 Speedup for PAUSIM-SF 56

4.12 Speedup for PAUSIM-BL 56
4.13 Speedup for PAUSIM-CY0 57
4.14 Speedup for PAUSIM-CY1 57
4.15 Efficiency for PAUSIM-SF 58
4.16 Efficiency for PAUSIM-BL 58
4.17 Efficiency for PAUSIM-CY0 59
4.18 Efficiency for PAUSIM-CY1 59
4.19 Workload distribution: s3271 benchmark circuit, PAUSIM-SF 63
4.20 Workload distribution: s3271 benchmark circuit, PAUSIM-BL 63
4.21 Workload distribution: s3271 benchmark circuit, PAUSIM-CY0 64
4.22 Workload distribution: s3271 benchmark circuit, PAUSIM-CY1 64
4.23 Workload distribution: s3271 benchmark circuit, PAUSIM-SF 65
4.24 Workload distribution: s3271 benchmark circuit, PAUSIM-BL 65
4.25 Workload distribution: s3271 benchmark circuit, PAUSIM-CY0 66
4.26 Workload distribution: s3271 benchmark circuit, PAUSIM-CY1 66

List of Tables

4.1 Fault coverage statistics using 1600 random vectors on a single processor . 47

4.2 Execution time (seconds) and speedups using 1600 random vectors on multiprocessor 48

4.3 Mean and standard deviation execution times on eight processors for PAUSIM-SF, PAUSIM-BL, PAUSIM-CY0 and PAUSIM-CY1. The unit for time is second. s1196, s1423 and s1512 61

4.4 Mean and standard deviation execution times on eight processors for PAUSIM-SF, PAUSIM-BL, PAUSIM-CY0 and PAUSIM-CY1. The unit for time is second. s3271 and s5378 62

Chapter 1

Introduction

Once a digital circuit is designed and fabricated, the circuit needs to be tested for the potential presence of physical defects or faults. The objective of a fault simulation algorithm is to find the fraction of total faults (also referred to as the fault coverage) that are detected by a given set of input vectors. Especially, fault simulation is essential to designing a high fault coverage Built-in Self Test (BIST) becoming popular for VLSI testing.

1.1 Background

In the simplest form of testing, a fault is injected into a logic circuit by setting a line or a gate to a faulty value (1 or 0), and then the effects of the fault are simulated using zero-delay logic simulation. Most fault simulation algorithms are typically of $O(n^2)$ time complexity, where n is the number of lines in the circuit. Studies have shown that there is little hope of finding a linear time fault simulation algorithm [3].

Figure 1.1 shows a typical fault simulator [1]. The block C() is the fault-free circuit and blocks C(f_1) through C(f_n) are copies of the same circuit with faults f_1 through f_n permanently inserted. The good circuit (fault-free circuit) and the faulty circuits are simulated for each test vector. If the output responses of a faulty circuit differ from those of the good circuit, then the corresponding fault is detected, and the fault can be dropped from the fault list, speeding up simulation of subsequent test vectors. A fault simulator can be run in a stand-alone mode to grade a given set of test vectors,

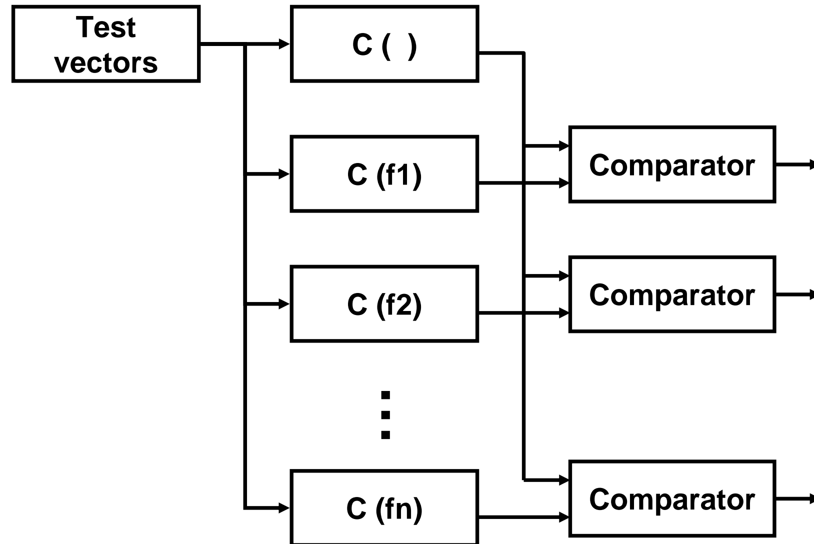


Figure 1.1: Typical fault simulator.

or interfaced with a test vector generator to reduce the number of faults that must be explicitly targeted by the test vector generator. In a random vector environment, the fault simulator helps in evaluating the fault coverage of a set of random vectors. In either environment, fault simulation can consume a significant amount of time, especially in random vector testing in which millions of vectors may have to be simulated. While many methods have been suggested for efficient fault simulation to evaluate the fault coverage of an enormous amount of test patterns, parallel processing can be utilized to reduce the fault simulation time greatly.

1.2 Review of Previous Work

To parallelize simulation, one may exploit circuit parallelism, algorithm parallelism, data parallelism, or a combination of them.

In a circuit-parallel approach [19], a circuit is partitioned into several parts. Each part is assigned to a processor. When a processor needs the information on a circuit node or line that is not in its own part, it must communicate with the processor that has the information. Hence, a large amount of communication is required. Circuit parallelism has the advantage that each processor needs to store the circuit description and temporary structures only for a fraction of the circuit, hence requiring a smaller space of memory.

In an algorithm-parallel approach [24][25], the simulation is carried out in a pipeline mode. That is, all gates at each logic level form a stage of a pipeline. A processor is assigned to each stage. The different test vectors are then pipelined through the logic circuit. In the case of a sequential circuit, since test vectors have to be applied in sequence, it is not possible to exploit pattern-parallel approach. Also, due to the presence of feedback paths through memory elements like flip-flops, speedups may be severely limited.

In a data-parallel approach [17][26][29], the simulation data are partitioned into disjoint sets and each set assigned to a processor. Each processor executes the entire algorithm and simulates the entire circuit. Fault parallelism is relatively simple to exploit. The fault list is partitioned among processors, and each processor performs fault simulation on the entire circuit for its own fault list with the complete set of test vectors. It is possible to obtain an almost linear speedup. The problem is that for each partition

of faults. Depending on the partitioning of the faults, the faults of each partition for a test vector may not be uniform across all partitions. In pattern parallelism, the given input test vectors are decomposed into subsets. Each processor gets a copy of the entire circuit, the fault set and a subset of the test vectors. Each processor performs fault simulation with its subset of test vectors. For sequential circuits, the future behavior of the circuit depends on the past input vectors, thus, limiting applicability of this approach.

1.3 Problem Definition

Parallel computing is becoming an increasingly cost-effective and affordable means for providing high computing power and represents a challenge to costly conventional supercomputers. For example, a cluster of workstations (COWs) can be easily configured as a high performance computing platform. Therefore, it is worthwhile to investigate efficient ways to utilize a COWs for time-consuming circuit testing.

1.4 Objective

The objective of this study is to develop parallel fault simulation algorithms that can be efficiently executed on a COWs in order to maximize speedup. This is achieved by judiciously partitioning the fault and test vector spaces, minimizing redundant computation and better balancing the load distribution among workstations (processors).

Chapter 2

Circuit Testing

In this chapter, the general issues of VLSI testing are briefly described, introducing the important concepts and terms.

2.1 Fault Modeling

Fan-out stem : A signal that branches to multiple places, each of which is called a fan-out branch. The source of those branches is called stem or fan-in.

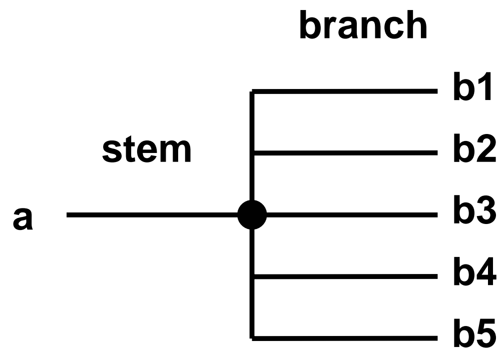


Figure 2.1: Definition of fan-out stem.

Fault: A defect in electronic system is the unintended difference between the implemented hardware and its intended design. A representation of a defect at the abstract

function level is called a fault. That is, a defect and a fault are the imperfections at the hardware and function levels, respectively.

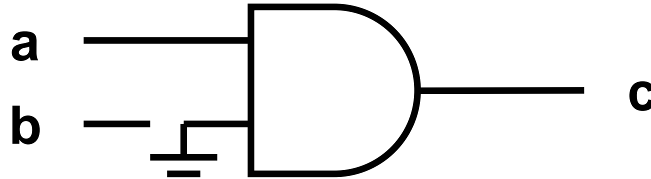


Figure 2.2: Definition of fault.

A simple digital system in Figure 2.2 consists of an AND gate, two input terminals, a and b, and an output terminal c. But, the connection between b and the gate is left unconnected and the second input of the gate is grounded. The functional output of this system, as implemented, is $c = 0$, instead of the correct intended output $c = ab$. For this system, the defect is a short to ground, and the fault is single b stuck at logic 0.

Stuck-at Fault: The type of fault described above is modeled by assigning a fixed (0 or 1) value to a signal line in the circuit. A signal line is an input or an output of a logic gate or a flip-flop. The most popular forms are the single stuck-at faults, i.e., two faults per line, *stuck-at-1* (s-a-1 or sa1) and *stuck-at-0* (s-a-0 or sa0).

Figure 2.3 illustrates a single stuck-at fault [1]. A stuck-at-1 fault as marked at the output of the OR gate means that the faulty signal remains 1 irrespective of the input state of the OR gate. It shows that the normal (faulty) value at the output is applied to the AND2 gate as 0 (1). When the input vector (1100) is applied as a test vector for the s-a-1 fault, it is easy to see that the normal and faulty outputs are different. The circuit

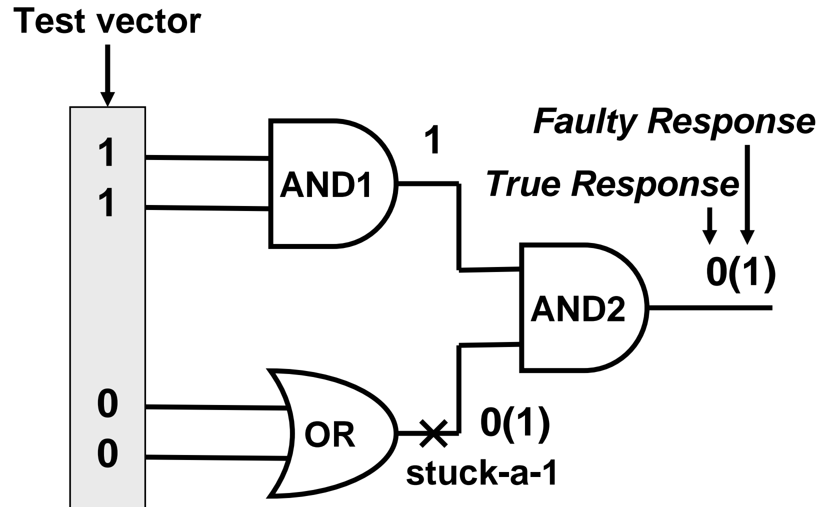


Figure 2.3: An Example of a single stuck-at fault.

in Figure 2.3 has seven lines, each of which is the potential site for a single stuck-at fault. Hence, the number of possible faults is 14.

Level and Output Cone of Circuit: Each gate in a circuit can be assigned a level, which represents the maximum distance (in gates) from a primary input (PI) to the gate. In Figure 2.4, the levels of gates are shown in circles. A level i gate is one for which at least one of its inputs is from a level *i-1* gate. For example, PIs *G0*, *G1*, and *G2* have the distance of 0 and therefore are assigned a level of 0. Accordingly, the top two inputs of gate *G5*, the top input of gate *G6*, and the inputs to inverters *G3* and *G8* are labelled with a level of 0. When all of the inputs of a gate are labelled, the gate and its outputs are labelled with the maximum of its input levels plus 1. That is, the gates

G3 and G8 are given a level of 1. Then, G4 is labeled with a level of 2, G5 and G6 with a level of 3. Finally, G7 is assigned a level of 4.

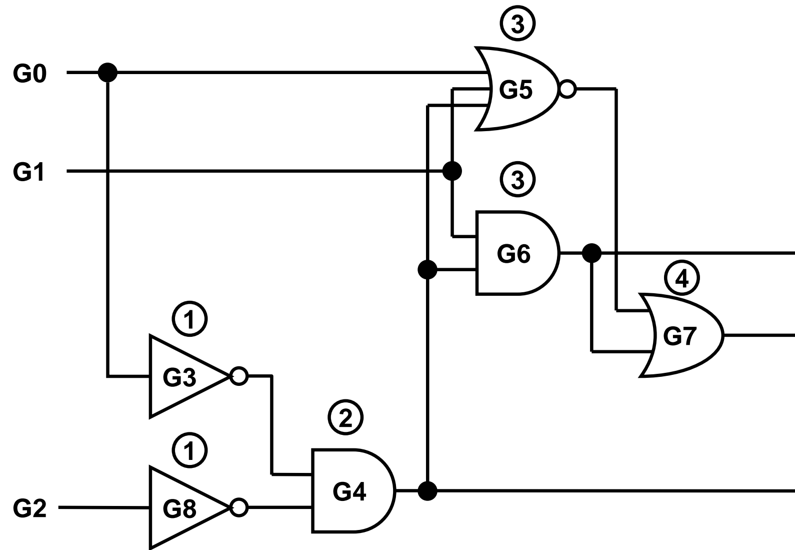


Figure 2.4: An example of an level order.

Each gate can be assigned a level by parsing the circuit once from the PI's to the primary outputs (PO's). A path is an alternating sequence of wires and gates. Signals are propagated from the inputs of a circuit to the outputs along one or more paths. A gate g is in the output cone *OC(*O_i*)* of a circuit PO O_i if there is a path from the output of g to the PO O_i. More formally, a gate g is in the output cone *OC(*O_i*)* of the PO O_i if its output is the PO O_i or at least one of the gates on the fan-out of g is in *OC(*O_i*)*. Figure 2.5 shows the output cone of an output O_i [33]. The faults from the two gates, *G1* and *G2*, are propagated on the same path to the output. The cone to which a gate in a circuit belongs is determined by a simple depth-first search from each PO. Both level

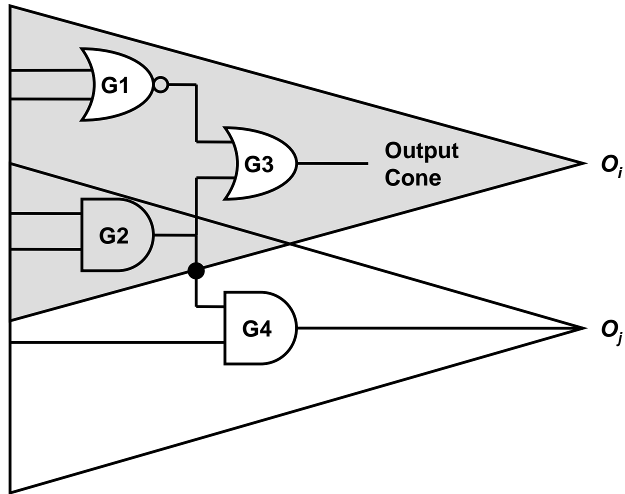


Figure 2.5: An example of an output cone.

and cone identifications can be carried out in time proportional to the number of gates in the circuit.

2.2 Test Pattern Environment

Fault simulation algorithms may be used in two different environments. One is a deterministic test generation environment and the other is a random pattern test generation environment. In the former, a specific algorithm named Automatic Test-Pattern Generation (ATPG) is used to generate a test vector for every fault in a circuit. The ATPG selects a fault from the fault list and tries to generate a test vector for the fault. If the ATPG is successful in generating a test vector, fault simulation is carried

out on the entire fault list and the fault simulator finds out the additional faults that are detectable by the same test vector. But, the ATPG is an NP-complete problem and can be computationally very expensive. In fact, in some complex circuits, the use of such algorithms is no longer feasible or practical.

In the latter, a fault simulator determines if test vectors lead to the detection of a target fault and evaluates the fault coverage of a set of random test vectors. The random pattern environment uses a relatively inexpensive pseudo-random test pattern generator to generate test vectors. The fault simulator randomly selects test vectors, runs fault simulation and determines which faults are detected. Since the random pattern testing may have to be simulated for a large number of vectors, fault simulation can be very time-consuming. Thus, parallel processing can be employed to reduce the fault simulation time significantly.

2.3 Sequential Methods

The task of fault simulation is to determine for each fault in a given list whether the simulation of the faulty and fault-free circuits differ in any primary output. While a fault simulator can be built in a straightforward manner from any logic simulator, the resulting performance would be low due to significant duplicated computation. As a result, various techniques to reduce duplicated computation have been developed. These approaches may be classified by the manner in which they compute and store the good and faulty circuit states. The most popular approaches are word-parallel, deductive, concurrent, differential, and proofs fault simulation algorithms. In this section, these and other single processor simulation algorithms are reviewed.

2.3.1 Word-parallel Fault Simulation

Several algorithms have been proposed for sequential circuit fault simulation, most of which are targeted at single stuck-at faults in synchronous sequential circuits. Three-valued (0, 1, X) simulation is generally carried out, and no reset is assumed. Word-parallel simulation [4] utilizes bit-oriented logic operations to perform many of gate evaluations simultaneously. If one word consists of *32* bits on a computer, *32* gate evaluations can be performed at a time, where one bit is used for good circuit. The word-parallel simulation can be either fault-parallel or pattern-parallel. The former simulates the good circuit and *31* fault classes with one input vector at a time by assigning a bit to each fault case. The latter simulates *32* input patterns for one fault at a time by assigning a bit to each test vector case. In the word-parallel fault simulation [4], 31 faulty circuits are simulated in parallel with the good circuit. Faults are packed statically into fault groups, and all test vectors are applied to the circuit for a given fault group. Then, the process is repeated for each group of 31 faults. Fault detection is done by comparing the good and faulty circuit values of the primary outputs. Fault dropping is not possible in this algorithm; therefore, the bit space for a faulty circuit is wasted once the fault is detected.

Figure 2.6 shows a circuit that is being simulated for three faults, c stuck-at-0, f stuck-at-1 and g stuck-at-0, with a four-bit word. To simulate the fault-free and three faulty circuits in parallel, the signal on each line is expressed as one word. The state of each bit represents the signal value in the fault-free and faulty circuits. When a vector $(a, b) = (1, 1)$ is applied, the output of the circuit with c s-a-0 (the second bit) and g s-a-0 (the fourth bit) differs from that of the fault-free circuit (the first bit). Hence, those

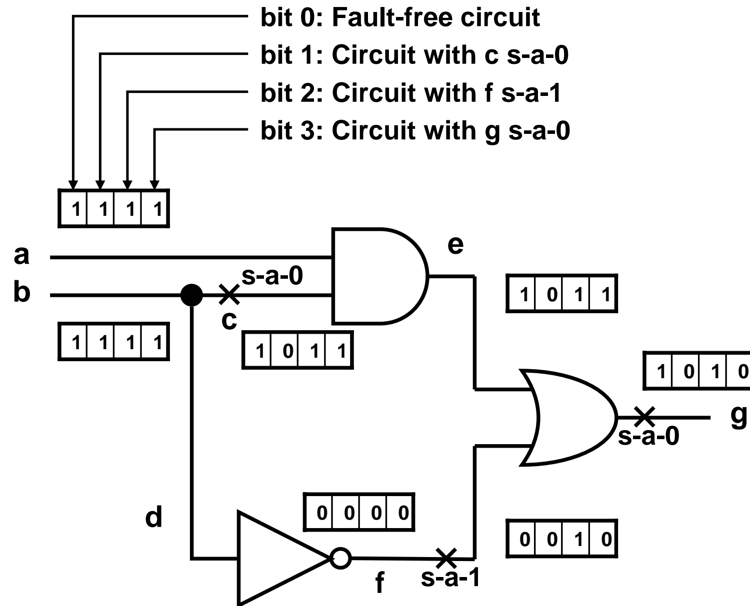


Figure 2.6: An example of parallel fault simulation.

faults are detected. The other fault (the third bit), which produces the same output as that by the fault-free circuit, is not detected by this vector.

2.3.2 Deductive Fault Simulation

In deductive fault simulation [5], an event-driven algorithm is used, and processing an event involves simulating the good circuit and propagating lists of active faults for a given test vector. Every node in the circuit may have a large list of active faulty circuits associated with it, and fault propagation is done using set operations on the lists of active faulty circuits at the inputs of a gate. However, since an event-driven algorithm

is used, fault propagation is done only if one of the active fault lists at the inputs of a gate has changed since the previous time frame.

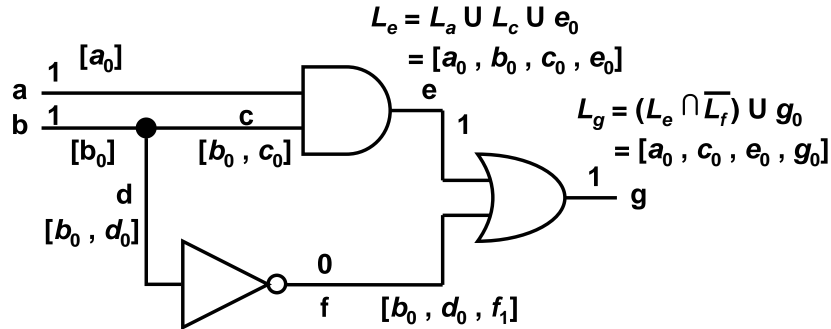


Figure 2.7: An example of deductive fault simulation.

An example of deductive fault simulation is shown in Figure 2.7 [1]. The vector (1, 1) is applied to the circuit. First, logic simulation is carried out to determine all signal values. Next, the s-a-0 and s-a-1 faults on all lines a through g are simulated. The lists of primary inputs just contain the respective s-a-0 faults that are active at the inputs. Their fault lists are denoted as sets, $L_a = [a_0]$ and $L_b = [b_0]$. Fault lists for fan-outs c and d are obtained by adding their locally active faults to the fault list L_b of the stem. The fault lists for e, f and g are obtained by fault propagation. When the fault propagation is completed, four faults a s-a-0, c s-a-0, e s-a-0, and g s-a-0 are detected by the input vector (1, 1).

2.3.3 Concurrent Fault Simulation

Concurrent fault simulation [6][7] is similar to the deductive fault simulation, but fault lists are propagated by evaluating individual gates, and only active faulty circuits

are simulated, which reduces the execution time. Timing information can easily be incorporated, and function-level modules can be handled. However, more memory is required to store the fault lists. Fault dropping is straightforward in both deductive and concurrent fault simulations. The performance of a concurrent fault simulator can be improved if it is restricted to synchronous sequential circuits [8]. Further improvements in performance have been achieved with a parallel concurrent approach [9].

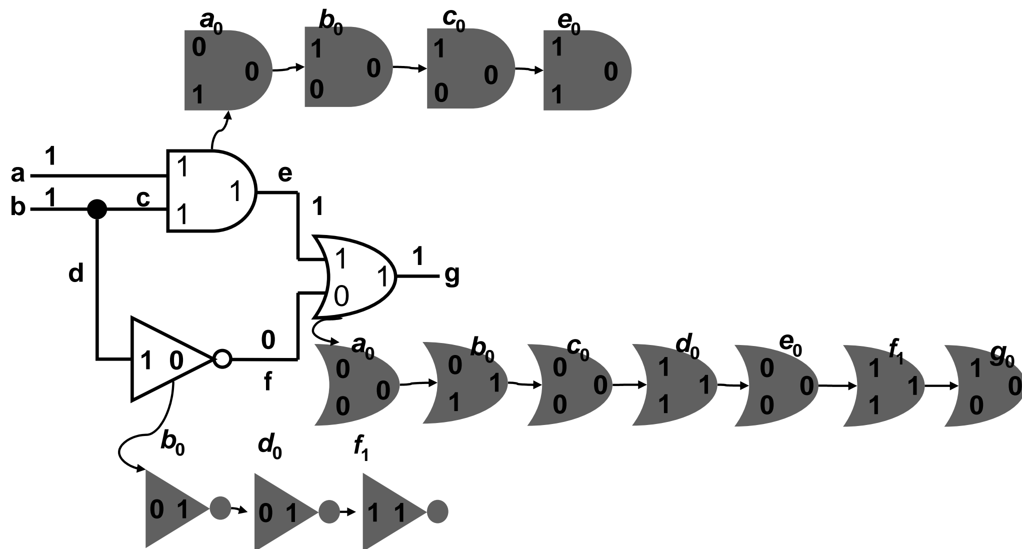


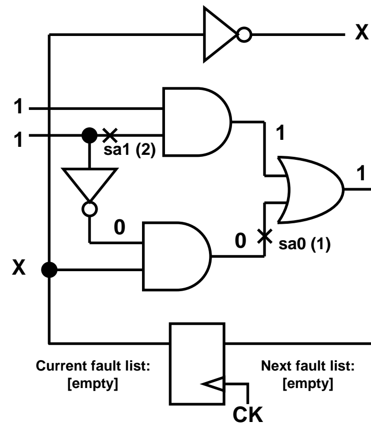
Figure 2.8: Fault-lists in concurrent fault simulation.

Figure 2.8 shows that all stuck-at faults are concurrently simulated for an input vector (1, 1) [1]. The subscript notation is used for faults. Thus, fault b_0 means b s-a-0 fault. Faults are modeled on all gate inputs, primary inputs a and b, and primary output g. To each good gate, a set of bad gates in grey shade with the corresponding fault name is attached in a linked-list structure. Signal values at the input and output of each gate are written inside the gate. At the primary output g, any bad gate whose output differs

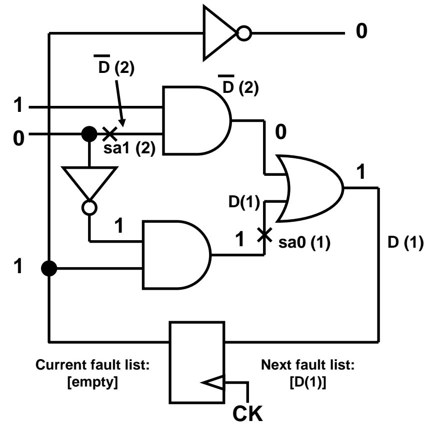
from that of the good gate indicates fault detection. Thus, faults a_0, c_0, e_0, and g_0 are detected by the test vector (1, 1). In the deductive simulator the fault list is for a signal and contains only the faults that affect that signal. In the concurrent simulator, the fault lists are for a gate and faults that affect the inputs of that gate are included in the list. Fault lists in a concurrent simulator are, therefore, comparatively longer.

2.3.4 Differential Fault Simulation

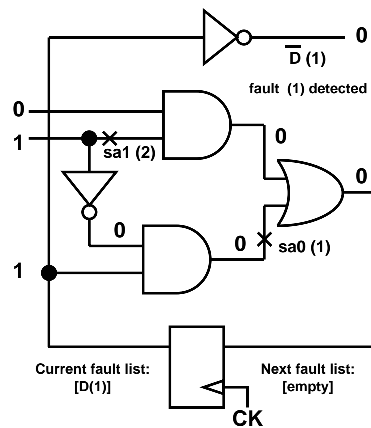
Differential fault simulation algorithm was proposed for synchronous sequential circuits [10], where only differences between the current and previous faulty circuits are simulated. The memory requirement is low, since only a single copy of node values and differences between the succesive faulty circuits in the flip-flops are stored. Fault dropping is more difficult, however, since simulation of each faulty circuit depends on the previous faulty circuit.



(a) First vector (1, 1)



(b) Second vector (1, 0)



(c) Third vector (0, 1)

Figure 2.9: An example of differential fault simulation

Figure 2.9 shows the simulation of two faults (1) and (2) [1]. The vector set contains three vectors, the first of which is simulated in Figure 2.9(a). The initial state of the flip-flop is assumed to be unknown and is denoted as X. After simulating the second vector (1, 0) in Figure 2.9(b), both faults are activated. The effects of fault (1) are denoted as D(1) and that of fault (2) as $\overline{D}(2)$. Only D(1) reaches the flip-flop input and is added to the next fault list. Note that no fault has been detected so far. Figure 2.9(c) shows the simulation of the third vector (0, 1). The current fault list is updated with D(1), which propagates to the primary output. Thus, fault (1) is now detected and can be dropped. Subsequent vectors will only simulate fault (2) until that is detected.

2.3.5 PROOFS

The PROOFS fault simulator combines the features of word-parallel, concurrent, and differential fault simulation algorithms. For each test vector, the good circuit is first simulated, and then only the differences between the good and faulty circuits are simulated. Several faulty circuits are simulated together, with one bit of the computer word used for each faulty circuit, and faults are grouped dynamically with each test vector simulated, in order to fully utilize all bits in the computer word. To limit the memory usage, faulty circuit values are stored at the flip-flops only. Faults are dropped from the fault list once they are detected, and faults that are identified as inactive in a given time frame are not simulated.

The overall algorithm of PROOFS [2] is shown in Figure 2.10. It consists of a main loop which reads in the next input vector, evaluates a logic circuit, and then simulates the faulty circuit for each fault group. To simulate a fault group, the group-id is first

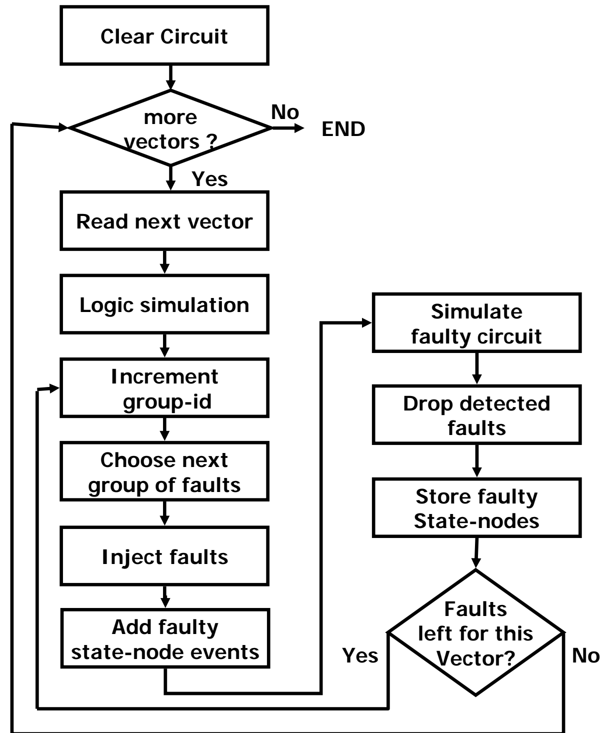
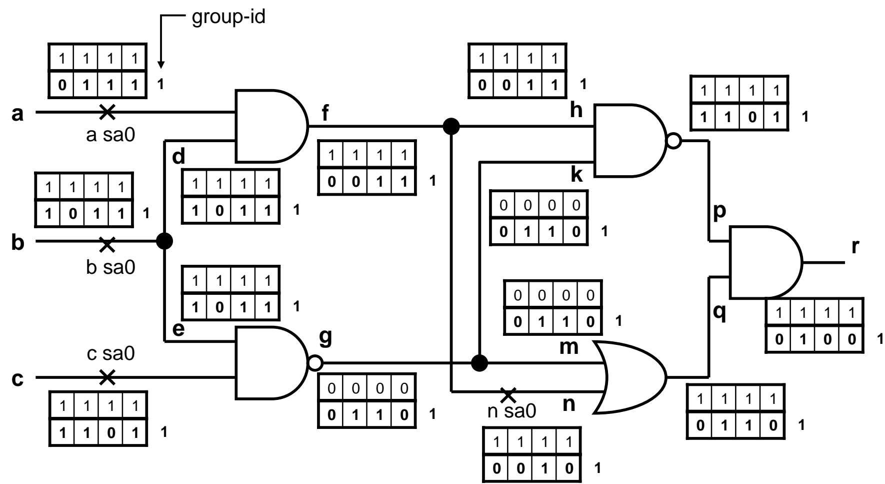
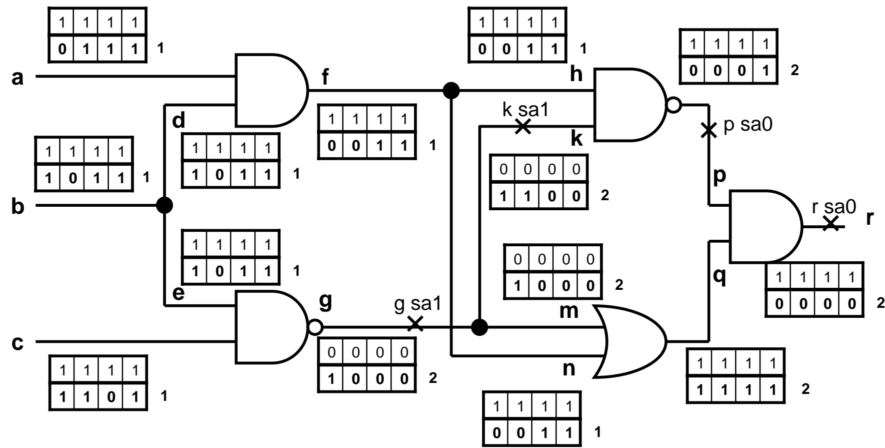


Figure 2.10: PROOFS algorithm.

incremented to identify each fault group. Next, the 32 faulty circuits to be included in the fault group are selected. The faults are then injected into the circuit and the node values for the state-nodes from the previous input vector are inserted into the faulty line values. The faulty circuits in this fault group are simulated, and the state-node values are stored for the next vector.



(a)



(b)

Figure 2.11: An example of PROOFS fault simulation.

Figures 2.11(a) and 2.11(b) show a simple example in which the PROOFS fault simulator is performed for each fault group. An input vector (1, 1, 1) is applied and the word length of four bits is assumed. The fault list groups a s-a-0, b s-a-0, c s-a-0, and n s-a-0 are simulated as shown in Figure 2.11(a). All the propagation lines of these faults have group-id of 1. After the first group is done, the following group g s-a-1, k s-a-1, p s-a-0, and r s-a-0 are simulated as shown in Figure 2.11(b). The lines of circuit affected by the second fault group are updated to a group-id of 2. As shown in Figure 2.11(b), three faults are detected in group 1 and four faults in group 2. These detected faults are dropped, and not simulated by the next test vector to avoid redundant simulation.

2.3.6 HOPE

HOPE is a PROOFS-based fault simulation [11]. It screens out faults with short propagation paths through the single fault propagation. A systematic method of identifying faults with short propagation paths reduces the number of faults simulated.

A significant speedup was achieved by finding representative stem faults for faults in fan-out-free regions [11]; only the representative stem faults are placed into the fault groups and simulated in word-parallel for faults whose effects do not propagate to flip-flops in the previous time frame. Single fault propagation is used to determine whether a fault in a fan-out-free region is active at the stem. Additional improvements in performance were obtained by modifying the fault injection procedure, statically ordering faults by fan-out-free region, and dynamically ordering faults to place potentially detected faults in separate fault groups [12]. The resulting fault simulator, HOPE, is about twice as fast as PROOFS, which is partially due to the improvements in implementation.

2.3.7 PARIS and PSF

The parallel-pattern single-fault propagation algorithm [13] for combinational circuits has been extended to synchronous sequential circuits in the fault simulators PARIS (PARallel Iterative Simulator) [14] and PSF (Parallel Sequence Fault simulation) [15]. For each group of 32 test vectors, the good circuit is simulated, followed by simulation of a single fault for all 32 vectors. Several iterations may be required before circuit values stabilize, due to the sequential nature of the circuits. Heuristics are used to minimize the number of iterations performed. Minor differences between PARIS [14] and PSF [15] exist. PARIS packs 32 consecutive vectors in each 32-bit computer word. PSF divides the test sequence into 32 equal subsequences and packs the nth vector of each subsequence in a single 32-bit word, where n ranges from one to the number of vectors in a subsequence. Both PARIS and PSF simulate one fault at a time, but the number of iterations required to stabilize the circuit state differs between them considerably.

The various existing approaches to sequential fault simulation were reviewed. Further improvements to these sequential algorithms have been made in the parallel algorithms using a cluster of workstations.

2.4 Parallel Methods

A number of approaches have been proposed for parallelization of fault simulation for both combinational and sequential circuits [18].

2.4.1 Circuit Parallelism

The main alternative to fault partitioning is circuit partitioning, in which the circuit being simulated is partitioned among available processors. Circuit-partitioned fault simulation is effectively a variant of parallel logic simulation [19].

Fault simulation based on circuit partitioning has been reported by Mueller-Thuns et al. [20] and Nelson [21] for vector-synchronous implementations on message passing machines. Ghosh [22] presents an implementation based on asynchronous logic simulation techniques that, while novel, falls short of achieving high efficiency. In [23], Patil et al. present a circuit-partitioned approach applicable to shared memory machines, that incorporates techniques from parallel logic simulation. A circuit is partitioned among the processors. Since the circuit is evaluated level-by-level with barrier synchronization at each level, the gates at each level are evenly distributed among the processors to balance the workloads. On the encore Multimax shared-memory multiprocessor system, an average speedup of 2.16 was obtained for 8 processors, and the speedup for the ISCAS89 circuit s5378 was 3.29.

2.4.2 Algorithmic Parallelism

Algorithm partitioning was proposed for concurrent fault simulation in [24][25]. A pipelined algorithm was developed, and specific functions were assigned to each processor. On a Sun Sparc 2 workstation with a MIPS (Million Instruction Per Second) rating of 28.5 million, an estimated speedup of 4 to 5 was reported for 14 processors, based on software emulation of a message-passing multi-computer [25]. The limitation of this approach is that it cannot take advantage of a larger number of processors.

2.4.3 Fault Parallelism

Fault partitioning is a simpler method for parallelizing fault simulation than other methods. In static fault partitioning, we have T test vectors that need to be simulated against F faults. A fault list is divided among available processors. Each processor simulated all faults in its partition independently. A fault parallel implementation has a good potential to achieve high speedup. However, the fault activity of each partition for a particular input vector may not be uniform across all partitions because the fault activity depends on the partitioning of the faults. Thus, this static partitioning has generally been considered ineffective in achieving high speedup especially for a large number of processors.

Several implementations based on dynamic fault partitioning have been attempted to even out the workloads of the processors at the expense of extra interprocess communication [17][26]. In [26], ProperPROOFS fault simulator use both static and dynamic partitioning model in which static partitioning is performed by dividing the fault list by the number of available processors at the start of processing and asynchronous fault redistribution follows. When a processor completes simulation of its existing fault list, it sends a request to another processor selected at random for more faults to be simulated. The processor receiving a request splits its fault list to share with the requesting processor, or forwards the request to another processor at random if it has an empty fault list. When all faults are simulated, each processor terminates simulation independently. In [17][26], speedup in the range 2.4 - 3.8 was obtained for static fault partitioning over 8 processors for the larger ISCAS89 circuits having reasonably high fault coverage (s5378

and s35932) on an INTEL iPSC/860 hypercube. Duba et al. [27] report a parallel implementation of CHIEFS for which speedup between five to six on a network of 8 Sun 3/280 file server workstations connected by a 10 Mb/s ethernet was achieved. Markas et al. [28] report a distributed algorithm for which speedups ranged from two to six on eight workstations in a heterogeneous cluster consisting of a Sun-3/160, Sun-3/60, a cluster of VAX 2000, and a cluster of VAX II/GPX. The performance of dynamic fault partitioning was not much better than the static fault partitioning due to the overheads of dynamic load balancing.

There is another drawback to both static and dynamic fault partitioning approaches in which the shortest execution time will be bounded by the time to perform the good circuit logic simulation (or simply logic simulation) on a single processor. Each processor must simulate the good circuit and the faulty circuit in its partition. Logic simulation on more than one processor is obviously redundant. Alternatively, if a shard-memory multiprocessor is used, the good circuit may be simulated by one processor only, but the remaining processors will be idle during the simulation, at least for the first time frame.

One observation that can be made about the fault partitioning experiments is that a higher speedup is obtained for circuits having lower fault coverage [17][26]. The potential speedup drops as the number of faults simulated drops. This is a reason for higher speedup unless the hard-to-detect faults are mostly assigned to a single or few processors. The logic simulation is not parallelized in the fault partitioning approach, and therefore, speedup is limited. Parallelizing logic simulation based on partitioning the circuit has been suggested, but has not been successful due to the high level of communication required between processors.

2.4.4 Pattern Parallelism

For parallelizing logic simulation, some test vector partitioning approaches were performed. The test vector partitioning provides a more scalable implementation, since the logic simulation is also distributed over processors. In SPITFIRE-0 [29], the test vectors are partitioned across the processors. This algorithm is presented as a base of reference for the various test vector partitioning approaches to be described later. As shown in Figure 2.12, the entire fault list is allocated on each processor. Thus, each processor targets the entire list of faults using a subset of the test vectors. Each processor proceeds independently and drops the faults that it can detect. This algorithm is somewhat inefficient in that many faults are very testable and are detected by most, if not all, of the subset of test vectors. Simulating these faults on all processors is a waste of time.

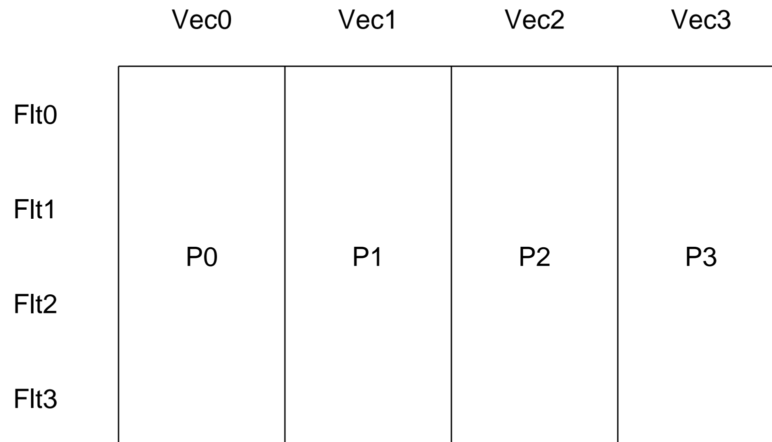


Figure 2.12: Test sequence partitioning in SPITFIRE-0

2.4.5 Fault and Pattern Parallelism

Some methods to combine fault and pattern parallelism were developed. SPITFIRE-1 [29], the synchronous two step algorithm, can filter out the easy-to-detect faults in an initial step in which both the fault set and the test set are partitioned among the processors. In the first step, each processor targets a subset of the faults using a subset of the test vectors, as illustrated in Figure 2.13 [29]. A large fraction of the faults is detected in this initial step, the undetected fault lists from the first step are combined and only the remaining undetected faults have to be simulated by all processors using test vectors in its partition in the second step.

	Vec0	Vec1	Vec2	Vec3
Flt0	P0			
Flt1		P1		
Flt2			P2	
Flt3				P3

1st Step for Fault Simulation

	Vec0	Vec1	Vec2	Vec3
Udt0		P1	P2	P3
Udt1	P0		P2	P3
Udt2	P0	P1		P3
Udt3	P0	P1	P2	

2nd Step for Fault Simulation

Figure 2.13: Partitioning in SPITFIRE-1

Other synchronous algorithm, SPITFIRE-2 and SPITFIRE-3, which are extensions of the SPITFIRE-1 algorithm, were presented in [31]. SPITFIRE-2, a hybrid approach,

attempted to reduce the partition size used in SPITFIRE-1. The fault and pattern partitioning for the two steps of fault simulation in SPITFIRE-2 is illustrated in Figure 2.14 [31]. As can be seen from the figure, processor i (P_i) uses Vec_0 and Flt_i in the first step of fault simulation. Since all faults are targeted in the first step using the input vectors in Vec_0, there is no need to re-simulate these vectors in the second step. In the second step, P_i uses the set of test vector Vec_{i+1} and any undetected faults left at the end of the first step. The advantage of this algorithm is that the number of vectors simulated in each step is now reduced by a factor $\frac{1}{P+1} \times 100 percent$ as compared to SPITFIRE-1. A small additional advantage is that the faulty circuit states available for the undetected faults in the set Udt_i can be used for simulation with the test set Vec_j in the second step of fault simulation on P_{j-1}. However, it is possible that the second step of this simulator may not drop as many faults as those by SPITFIRE-1 since less test vectors are used in the first step..

SPITFIRE-3 is a multistep pipelined synchronous algorithm which helps in overcoming any drawback in a single or two-step approach. The first step of fault simulation is identical to that in SPITFIRE-1. Synchronization points are introduced in the second step, in which processors exchange the information on the detected faults. This may reduce the amount of work that a processor has to do subsequently, since each processor does not need to target the faults that have been detected by other processors. However, the synchronization points introduce barriers which may slow down parallel execution, when the load is unbalanced among processors.

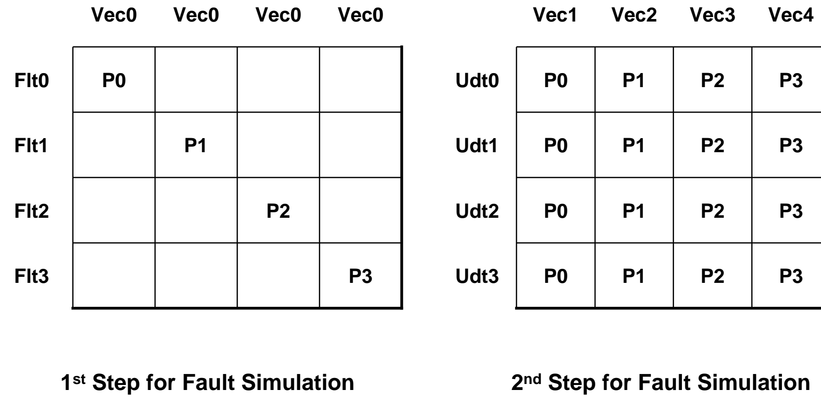


Figure 2.14: Partitioning in SPITFIRE-2

2.5 AUSIM

AUSIM is a gate-level, sequential circuit fault simulation program developed at Auburn University. AUSIM runs on the UNIX platform and targets single stuck-at faults in synchronous sequential circuits represented in the ISCAS89 benchmark format.

2.5.1 Program Configuration

The AUSIM consists of six sub-commands which are three pre-processing and three main commands. The command *default* checks the input file if the proper naming convention is used. The command *proc* indicates that all file names have been specified and processing is to begin. The processing of the files begins with syntax checks of the library file and the *ASL* file as well as a check for *subskt* that makes statements to initiate

flattening of the hierarchy. After hierarchical flattening is complete, the data structures are loaded and a number of audits and circuit checks are performed for items such as nets with multiple driving sources, nets with no driving sources, etc. The command *audit* records the audits results. After pre-processing, AUSIM can begin simulation with the three commands: *simul8* for logic simulation, *fltgen* for fault generation, and *fltsim* for fault simulation.

The command *simul8* is needed to initiate the application of the vector file (*circuit_name.vec*) to the circuit loaded into the data structures, producing the simulation output result file (*curcuit_name.out*). The command *fltgen* command generates gate-level stuck-at fault lists and writes the list to the file *circuit_name.flt.* Normally, the *fltgen* command produces a collapsed gate-level stuck-at fault list. The command *fltsim* takes input files (*circuit_name.out* and *circuit_name.flt*), performs simulation and produces the detected fault list (*circuit_name.det*), potentially detected fault list (*circuit_name.pdt*) and undetected fault list (*circuit_name.udt*).

2.5.2 Algorithm Analysis

Figure 2.15 shows the algorithm structure of sequential AUSIM. All the bits in a computer word are utilized to simulate 32 faulty circuits at once. Logic simulation is performed before simulating faulty circuits. Fault simulation consists of a main loop which reads in the next fault group. The faults are injected into the circuit and each test vector is inserted into the faulty circuit. The faulty circuits are evaluated, and state-node values are stored for the next vector. When each test vector detects all faults of the same fault group, AUSIM moves onto the next fault group.

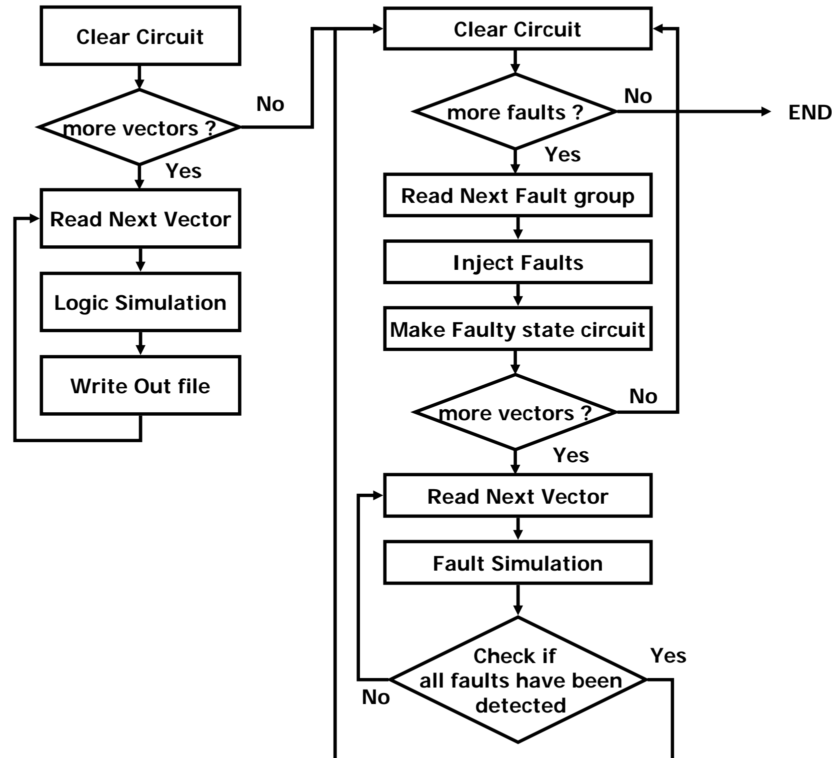


Figure 2.15: Sequential structure of AUSIM

The data structure used in AUSIM is shown in Figure 2.16, where the shaded memory blocks represent a linked list. There are three kinds of memory blocks which are *gate*, *input* and *net* structures of a circuit. Each *gate* memory block consists of a type, name, and input and output pointers. Each *input* memory block is used to control the state of the nets, which show interconnection among gates, corresponding to the state changed due to each single fault propagation. The fields in the *input* structure, *logval* and *umask*, are used to store the state of every net in the circuit. The fields in the *net* structure, *flt* and *saf*, store the fault information of each net in the faulty circuit.

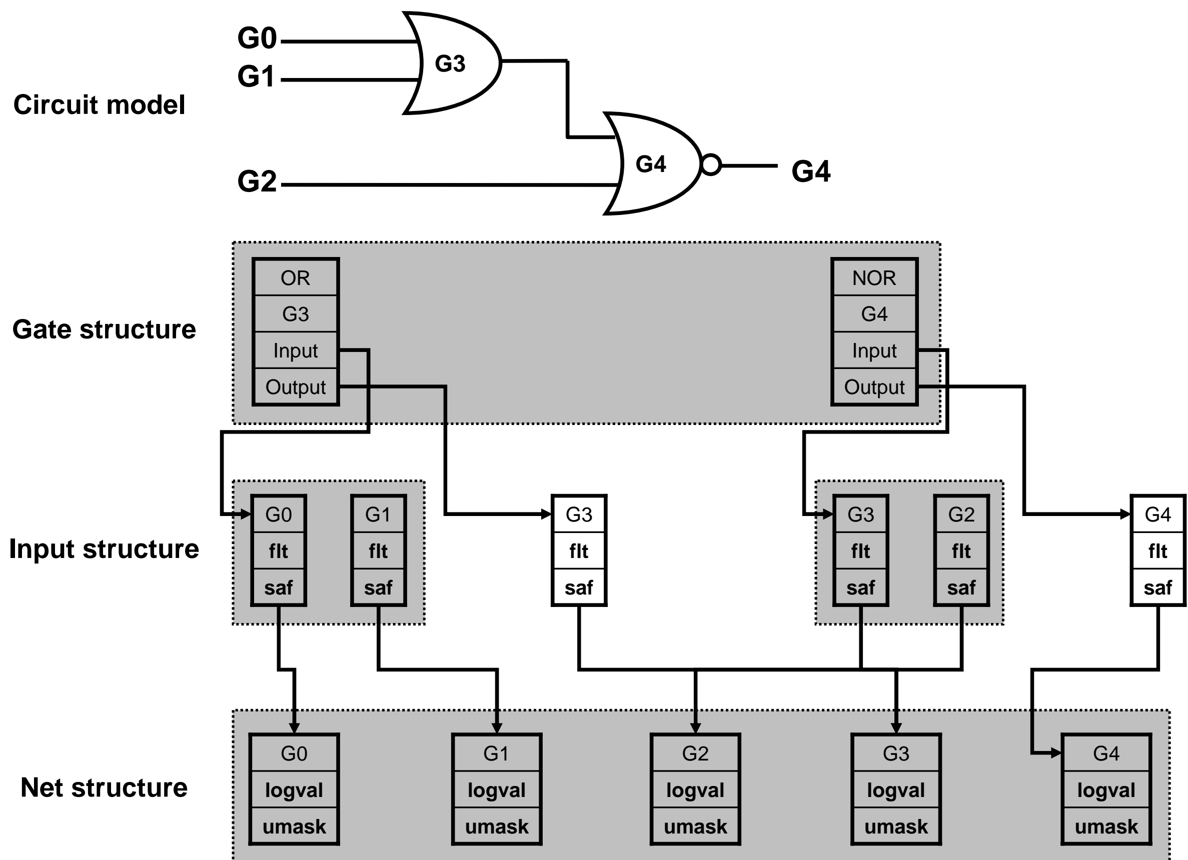
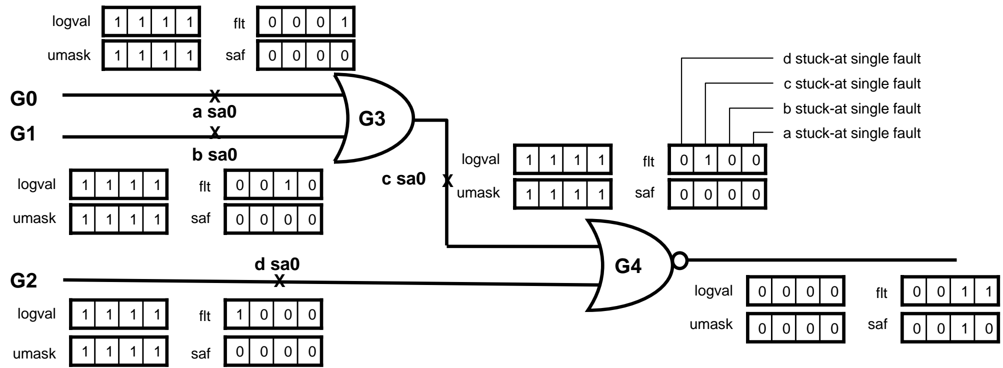
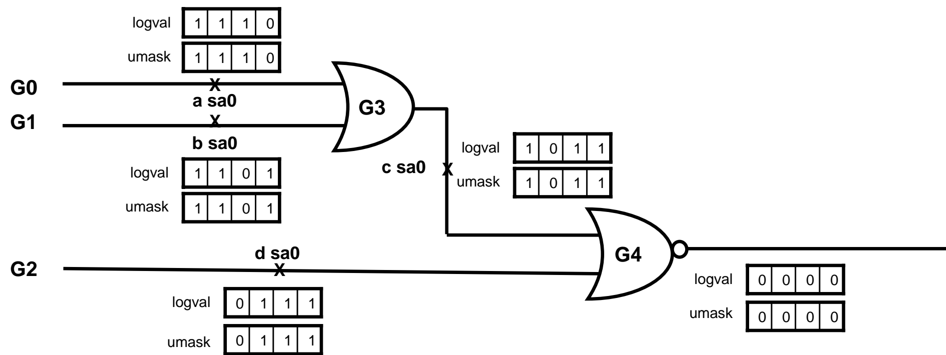


Figure 2.16: Data Structure of AUSIM

Each of *logval*, *umask*, *flt* and *saf* consists of two 32-bit words, where a pair of bits is used to store a different faulty machine's value. A three-valued logic (0, 1 and X) is used. Two bits are used to code the three values, one in *logval* and the other in *umask*. 0 is coded as (0, 0), 1 as (1, 1) and X as (0, 1).



(a)



(b)

Figure 2.17: An example of gate evaluation of AUSIM.

Figure 2.17 shows an example of gate evaluation for the test vector (1, 1, 1) on the fault list groups (a, b, c, d). A word length of four bits is assumed. Coded logics 0(0, 0), X(0, 1) and 1(1, 1) are used in a sequential circuit, but this example does not use the X value because it is a combinational circuit. Figure 2.17(a) illustrates the logic simulation for good circuit. The steady state value of each net in the circuit is kept in a single array in *logval* and *umask*. The information of faulty machine value is stored in another array *flt* and *saf*. Figure 2.17(b) represents the net state of circuit modified due to fault injection. After every fault in the same fault group is injected and the net state is updated with the faulty state, the gate evaluation is performed.

Chapter 3

Parallelization

In this chapter, a parallel fault simulator PAUSIM (Parallel AUSIM) which has developed based on AUSIM [32] is described. Three versions of PAUSIM have been implemented, i.e., PAUSIM-BL (BLock partitioning), PAUSIM-CY0 (CYclic partitioning) and PAUSIM-CY1.

3.1 PAUSIM-BL

3.1.1 Task Decomposition

PAUSIM-BL (Parallel AUSIM-BLock Partitioning) adopts the test vector and fault set partitioning algorithm for parallel processing. It performs a logic and fault simulations separately. Compared to SPITFIRE-1 consisting of the two steps of fault simulation [31], PAUSIM-BL's design focuses on eliminating a redundant work in logic simulation and fault simulations.

In SPITFIRE-1, logic simulation of the fault-free circuit is carried out in both steps of fault simulation, resulting in redundant computation. In order to avoid such redundancy, in PAUSIM-BL, the logic simulation results are saved so that they can be referred to during the fault simulation step.

In the second step of fault simulation of SPITFIRE-1, a fault may be detected by more than one processor since the fault space is not partitioned. That is, it is possible that a processor may simulate the faults which have been already detected by other

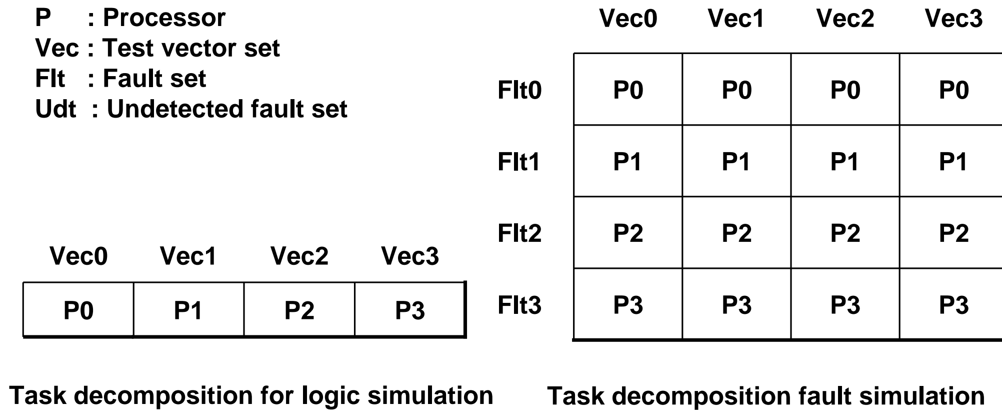


Figure 3.1: Task decomposition in PAUSIM-BL

processors. PAUSIM-BL avoids such possibility by assigning a disjoint set of faults to each processor in the second step as shown in Figure 3.1.

This also makes it unnecessary to filter out the multiply-detected faults when the faults detected by processors are combined following the (second step of) fault simulation. Fault simulation time on each processor is shorter on average for PAUSIM-BL than for SPITFIRE-1. Let's define a *unit simulation* as testing a circuit for a fault using a test vector. The number of possible unit simulations in the fault simulations would be the same regardless of partitioning faults or test vectors. However, the number of unit simulations actually carried out is smaller for PAUSIM-BL than for SPITFIRE-1. First, there is no redundant simulation in PAUSIM-BL. Second, since each processor in PAUSIM-BL is assigned less faults with more test vectors than in SPITFIRE-1, it is more likely in PAUSIM-BL than in SPITFIRE-1 that all of the assigned faults are detected

even before all possible unit simulations are tried. A processor stops fault simulation when all the faults in its assigned fault group are detected.

3.1.2 Fault Partitioning

In PAUSIM-BL, faults in a fault list are divided into n equal-size groups when there are n processors. In a fault list of a circuit, faults are arranged in the alphanumeric order of propagation gate and net names. Therefore, faults in a group tend to be from the contiguous parts of the circuit. All faults related to a gate are assigned to the corresponding processor.

Figure 3.2 shows the distribution of faults in an area of the s27 benchmark circuit. Faults from inputs of a gate are propagated on the same path to the primary outputs. Hence, the workloads for simulating the faults at different inputs of a gate are similar. If a fault group contains more hard-to-detect faults than other groups, there can be a significant load imbalance among processors.

3.1.3 Test Vector Partitioning

For combinational circuits, a test vector set may be partitioned into mutually exclusive subsets, each of which is assigned to a processor. In a sequential circuit, the current state depends on the previous state in general. Each processor initiates its fault simulation, starting from an unknown state at some outputs. This may cause detection of some faults to be missed. In order to eliminate or reduce unknown outputs, the test vector set may be partitioned in an overlapped manner as shown in Figure 3.3 [29]. Those test vectors in an overlapped portion are mainly used for updating the unknown states of outputs to the known states, rather than fault detection, i.e., they act as initializing

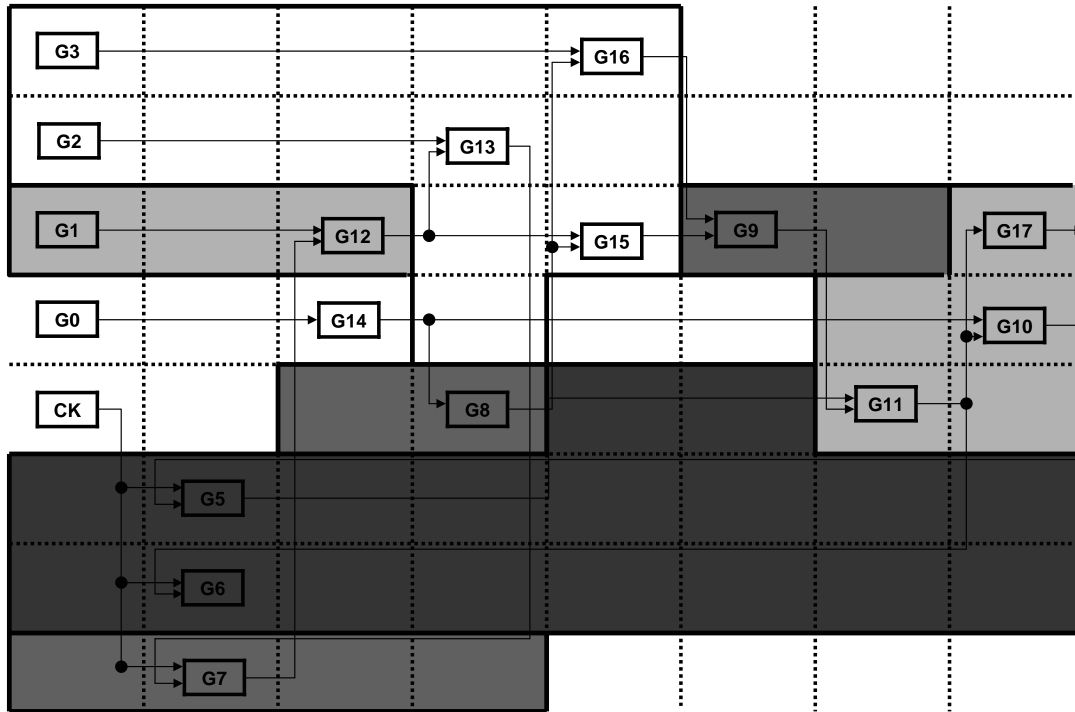


Figure 3.2: Fault partitioning in PAUSIM-BL for 4 processors where each fault group is distinguished by a different grey-scale

vectors. The optimal number of initializing vectors depends on the circuit. Too many initializing vectors would waste computation while few of them may lead to a low fault coverage.

3.1.4 Procedure

The procedure of PAUSIM-BL, which consists of logic simulation and fault simulation, is described in the flowchart in Figures 3.4 - 3.5. A master processor broadcasts

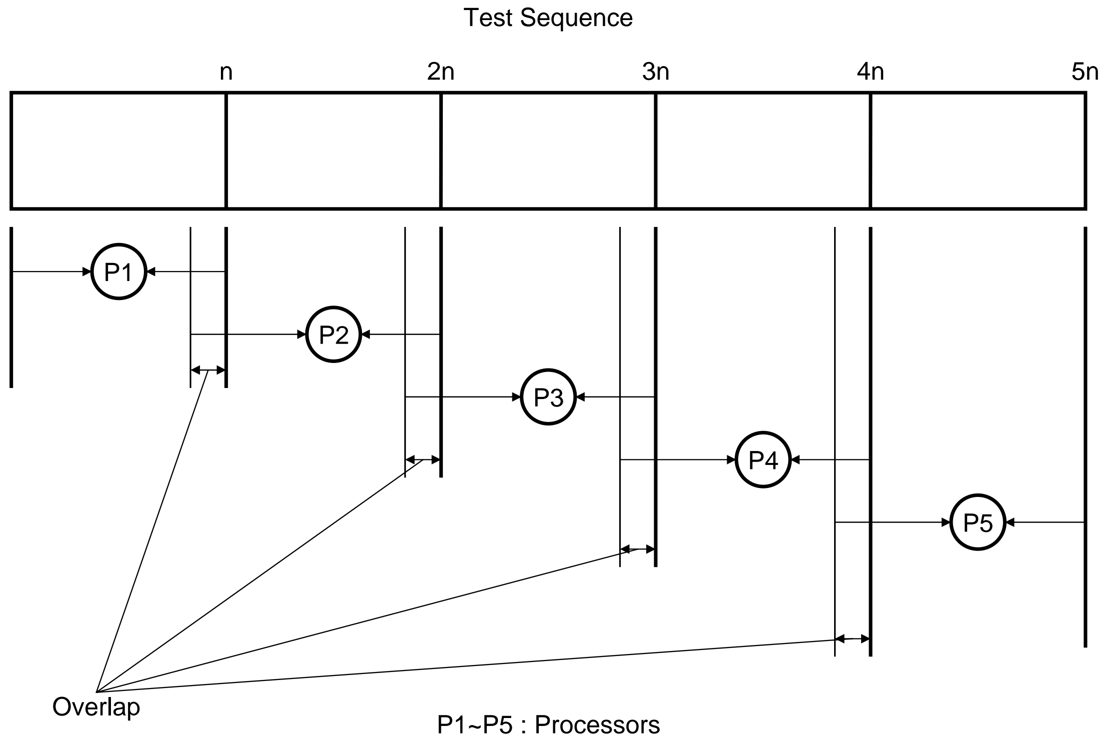


Figure 3.3: Test vector set partitioning

the circuit information and the entire test vectors to all slave processors. All processors including the master carry out the logic simulation with their respective partitions of test vectors. Local logic simulation results are collected to the master which then broadcasts the combined result to all slave processors. Then, the fault simulation starts. The master processor partitions a list of faults into subsets and distributes them among processors. The fault simulation is performed on all processors with faults disjointly distributed. At

the end of the fault simulation, slave processors report their detected and undetected faults to the master processor.

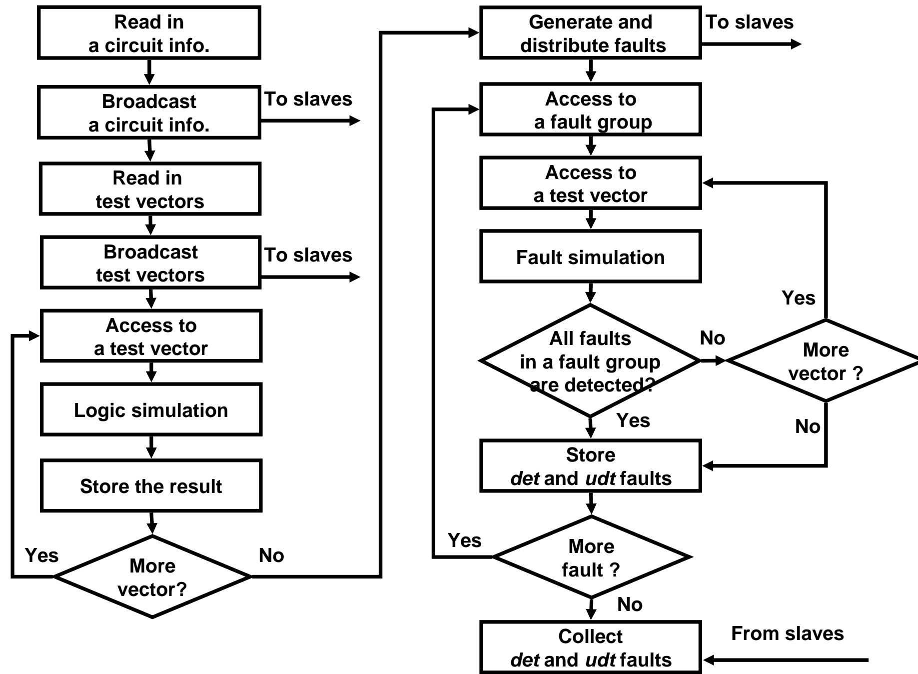


Figure 3.4: Flowchart of the master processor in PAUSIM-BL

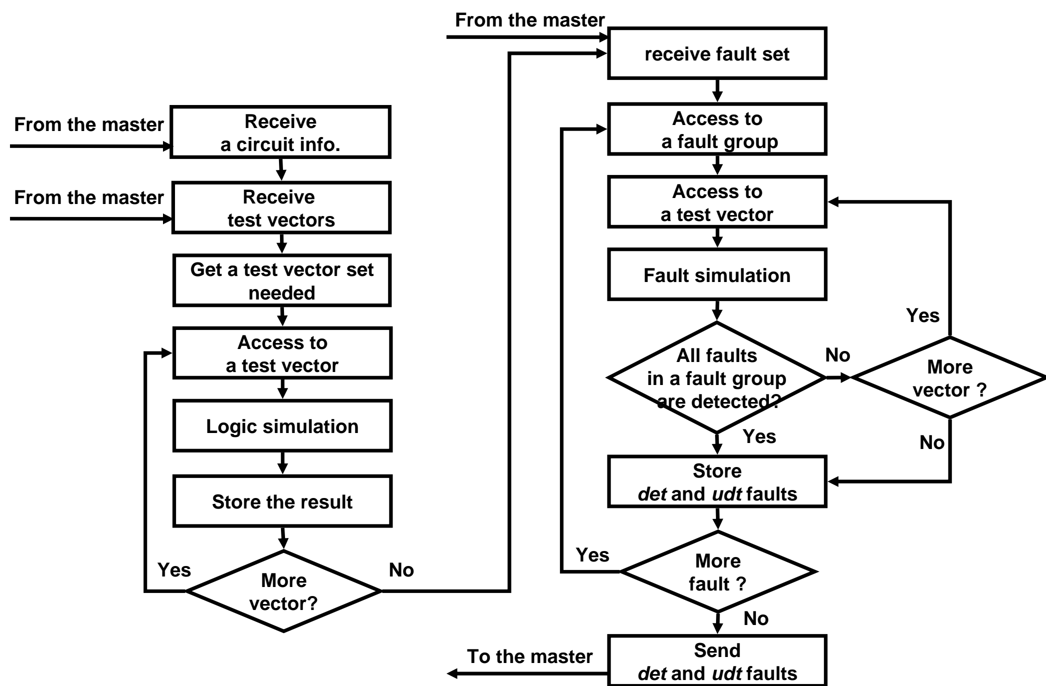


Figure 3.5: Flowchart of slave processors in PAUSIM-BL

3.2 PAUSIM-CY

3.2.1 Load Balancing

One problem in PAUSIM-BL is the potential load imbalance among processors. It is due to the fact that all the faults on a gate are assigned to the same processor and the faults assigned to a processor tend to be clustered in space. Computational requirements for the faults close to each other are similar, especially those at the same gate. Therefore, the load distribution among processors can be unbalanced significantly in PAUSIM-BL.

In order to achieve a more uniform load distribution, PAUSIM-CY0 (Parallel AUSIM-CYclic) and PAUSIM-CY1 adopt the LOG (Level Output Gate) partitioning of faults [33]. In the LOG partitioning scheme, faults on each gate and each circuit level are assigned to processors in a cyclic fashion. In this way, for example, the hard-to-detect faults on a gate would be distributed to multiple processors rather than a processor.

Figure 3.6 shows the difference between block partitioning and LOG partitioning used in PAUSIM-BL and PAUSIM-CY0 and PAUSIM-CY1, respectively. The fault list consists of four attributes which are gate name, net name connected to gate, input or output, and a type of single stuck at fault. Faults are arranged in the alphanumeric order of gate names. It is seen that faults in a PAUSIM-BL partition are mostly from a contiguous part of circuit while those in a PAUSIM-CY0 and PAUSIM-CY1 partition are scattered widely.

3.2.2 Procedures

PAUSIM-CY0 is identical with PAUSIM-BL except for the fault partitioning. Faults are distributed among processors in a cyclic manner in PAUSIM-CY in an effort to spread

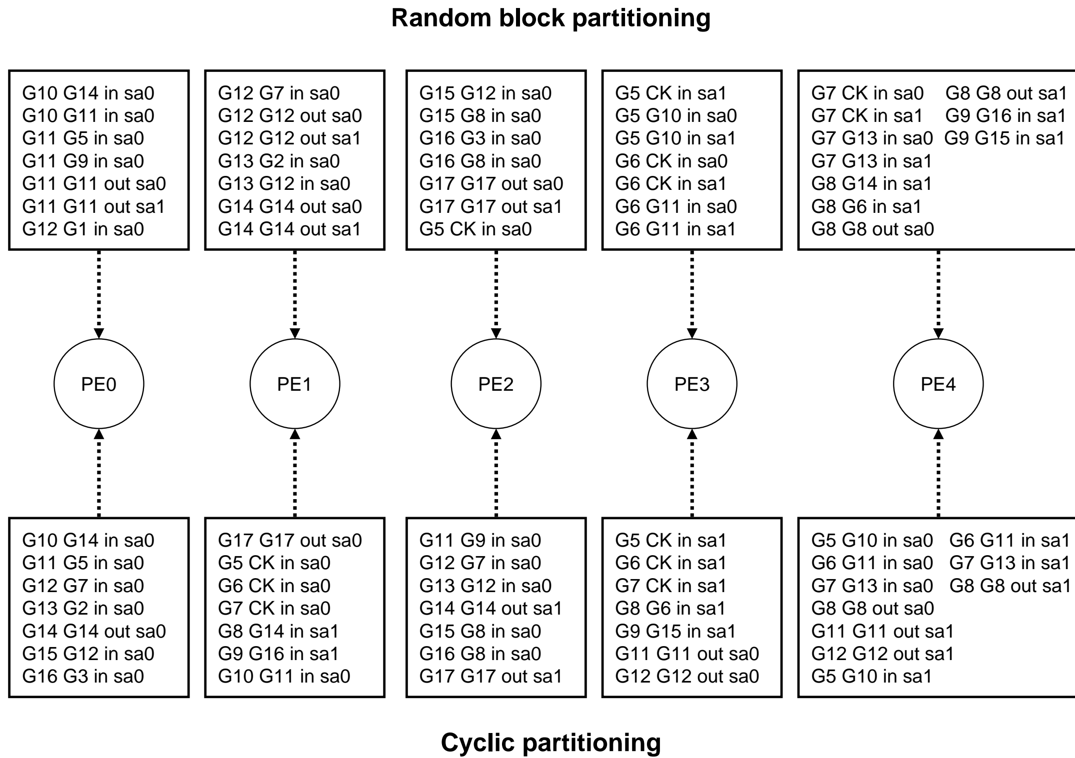


Figure 3.6: Example of fault partitioning for s27 benchmark circuit for 5 processors

distribution of hard-to-detect faults over as many different processors as possible. This cyclic partitioning of faults may not balance the load completely. Hence, PAUSIM-CY1 employs a two-step fault simulation for more effective load balancing as shown in Figure 3.7. In PAUSIM-CY1, after the first step of simulation, each slave processor reports its undetected faults to the master which then redistribute the undetected faults such that the load is well balanced over slave processors.

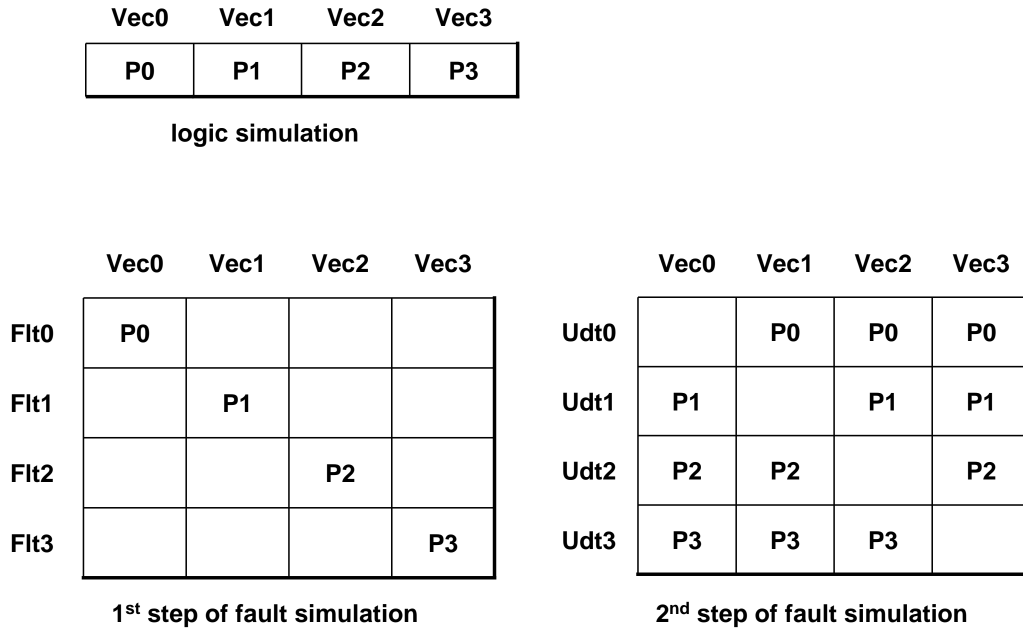


Figure 3.7: Task decomposition in PAUSIM-CY1

3.3 Implementation

The parallel simulation programs are written in C, using MPI library functions for communication. Figures 3.8-3.10 illustrate the logic simulation and two steps of fault simulation for PAUSIM-CY1, where the cluster consists of one master processor, *P0*, and three slave processors, *P1*, *P2*, and *P3*.

Communications among processors are required when (i) the master processor broadcasts test vectors (T_i) to the slave processors during logic simulation, (ii) the master processor distributes faults (F_i) to the slave processors in the first step of fault

simulation, (*iii*) the slave processors report the detected (DA_i), undetected (UA_i) faults and the results (O_i) of logic simulation to the master processor at the end of first step, (*iv*) the master processor redistributes the undetected faults (UB_i) and broadcasts the the results collected (O) to the slave processors in the beginning of the second step, and (*v*) the slave processors report the detected (DB_i) and undetected (UC_i) faults to the master processor at the end of fault simulation. These communications are implemented by *MPI_SEND* *MPI_RECV*, and *MPI_BCAST*.

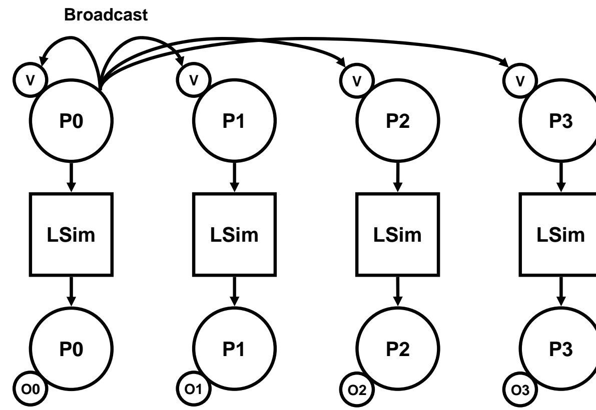


Figure 3.8: Communication among 4 processors in logic simulation in PAUSIM-CY1. V: Test vector, O: Logic simulation result

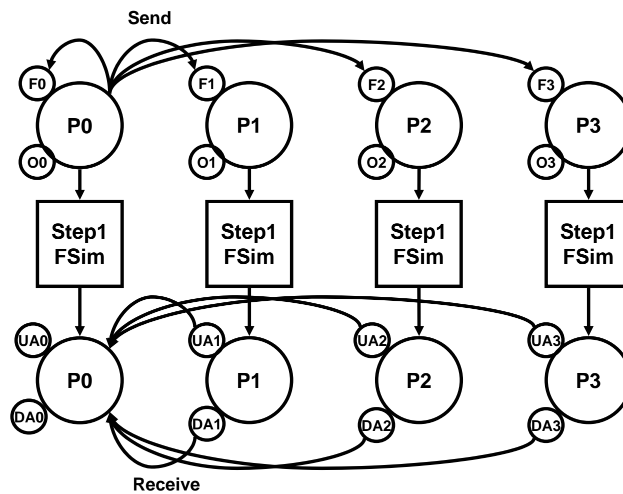


Figure 3.9: Communication among 4 processors in the first step of fault simulation in PAUSIM-CY1. F: Fault, U: Undetected fault, D: Undetected fault

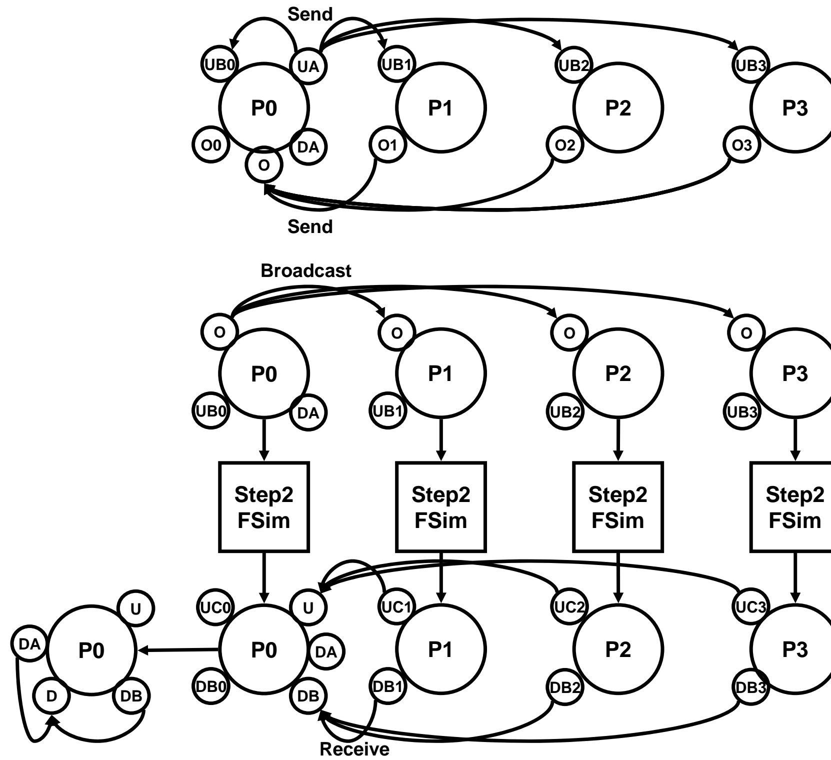


Figure 3.10: Communication among 4 processors in the second step of fault simulation in PAUSIM-CY1

Chapter 4

Performance

4.1 Experimental Environment

The four parallel schemes, PAUSIM-SF, PAUSIM-BL, PAUSIM-CY0 and PAUSIM-CY1, have been implemented, which were described in the previous section. The PAUSIM-SF refers to the parallel implementation which uses the task partitioning scheme of SPITFIRE-1 [31]. The PAUSIM-BL and PAUSIM-CY are the proposed parallel testing schemes described in chapter 3.

Table 4.1: Fault coverage statistics using 1600 random vectors on a single processor

Circuit	#Fault	#Gate	#FF	#PI	#PO	#Det.fault	Coverage
s1196	1250	388	18	14	14	1042	83.4%
s1423	1663	490	74	17	5	517	31.1%
s1512	1411	413	57	29	21	46	3.3%
s3271	3438	1035	116	26	14	3103	90.3%
s5378	4961	1004	179	35	49	2945	59.4%

Table 4.1 shows the characteristics of the benchmark circuits and the fault coverage on a single processor. Note that s1196 and s3271 have a high fault coverage while s1423 and s1512 contain lots of hard-to-detect faults. The four schemes were implemented on a cluster consisting of sixteen utra 5 Sun workstations. The workstations are interconnected by a 100-Mbps Ethernet. Results are provided for the five circuits, s1196, s1512, s1423, s3271, and s5378, taken from the ISCAS89 benchmark suite. Logic and fault

Table 4.2: Execution time (seconds) and speedups using 1600 random vectors on multiprocessor

Circ.	Sequ. time	Alg.	Execution time				Speedups			
			4	8	12	16	4	8	12	16
s1196	191.6	SF	131.4	58.6	52.9	50.2	1.5	3.3	3.6	3.8
		BL	119.1	52.9	44.8	36.0	1.6	3.6	4.3	5.3
		CY0	63.5	42.8	33.1	27.6	3.0	4.5	5.8	6.9
		CY1	56.9	36.3	28.0	27.4	3.4	5.3	6.8	7.0
s1423	489.7	SF	180.5	92.3	92.0	78.1	2.7	5.3	5.3	3.8
		BL	179.1	89.7	79.2	77.2	2.7	5.5	6.2	6.3
		CY0	147.0	78.3	65.6	52.3	3.3	6.3	7.5	9.4
		CY1	140.9	77.5	56.7	45.3	3.5	6.3	8.6	10.8
s1512	436.9	SF	135.3	87.6	71.7	61.7	3.2	5.0	6.1	7.1
		BL	134.4	85.3	67.0	47.0	3.3	5.1	6.5	9.3
		CY0	133.2	80.1	53.9	46.4	3.3	5.5	8.1	9.4
		CY1	127.3	73.9	53.4	47.0	3.4	5.9	8.2	9.3
s3271	1077.1	SF	1056.1	696.1	383.3	293.3	1.1	1.5	2.8	3.7
		BL	972.2	502.8	345.6	271.8	1.1	2.1	3.1	4.0
		CY0	516.6	294.4	274.1	204.4	2.1	3.7	3.9	5.3
		CY1	378.3	171.3	142.4	137.8	2.8	6.3	7.6	7.8
s5378	9898.4	SF	4701.3	2220.0	2089.3	1511.9	2.1	4.5	4.7	6.5
		BL	4602.3	1951.2	1724.6	1333.2	2.1	5.1	5.7	7.4
		CY0	3041.1	1471.6	1206.4	952.3	3.3	6.7	8.2	10.4
		CY1	2486.9	1266.8	849.7	675.6	4.0	7.8	11.6	14.7

simulations were done with 1600 random test vectors. The overlap in the test vector partitioning was 25 (test vectors).

4.2 Results

In Table 4.2, the execution time and speedup achieved by PAUSIM-SF, PAUSIM-BL, PAUSIM-CY0 and PAUSIM-CY1 on the cluster are provided for the five circuits in Table 4.1. The same fault coverage as in the sequential (uniprocessor) simulation was

obtained except for s1423 where one less fault was detected compared to the sequential result.

This minor difference between the sequential and parallel simulations is most probably due to the loss of state information in the beginning of parallel simulation (caused by test vector partitioning). When the test vector overlap was increased, there was no difference in the fault coverage between the sequential and parallel simulations.

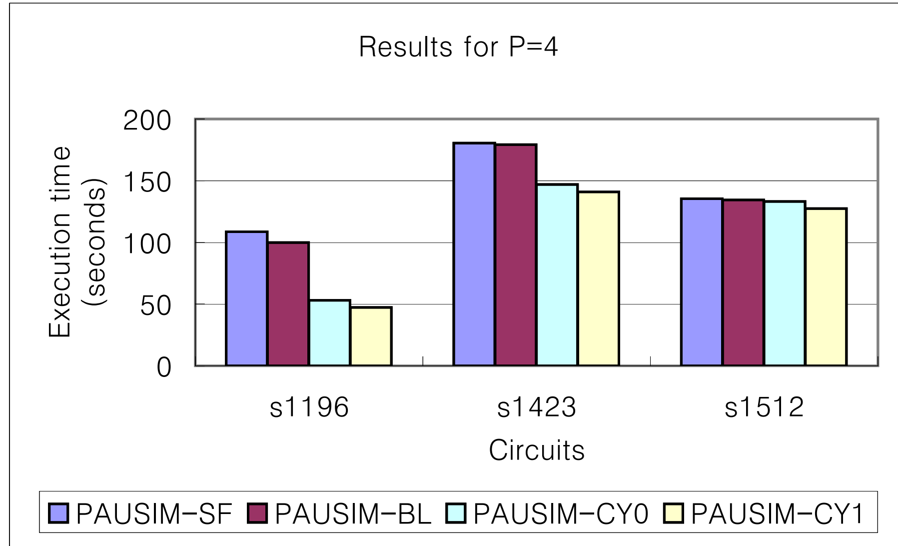


Figure 4.1: Execution time for 4 processors - small circuit

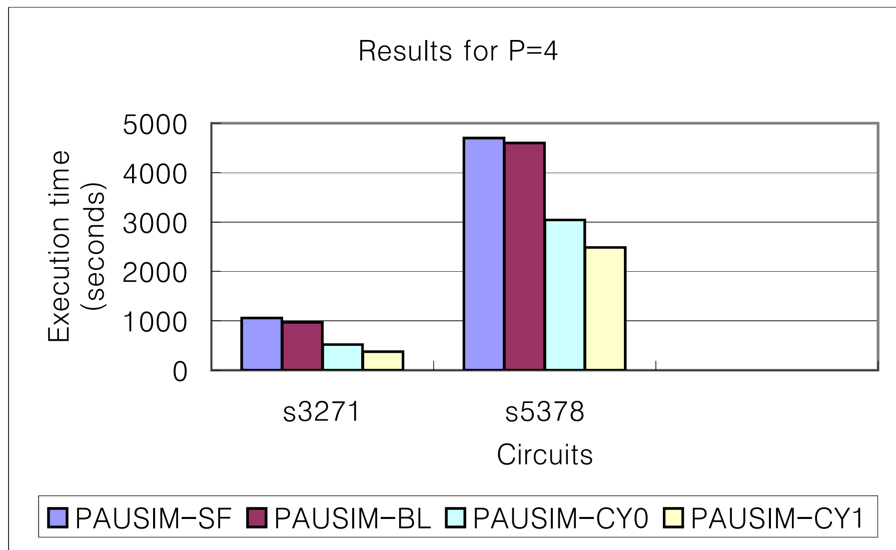


Figure 4.2: Execution time for 4 processors - large circuit

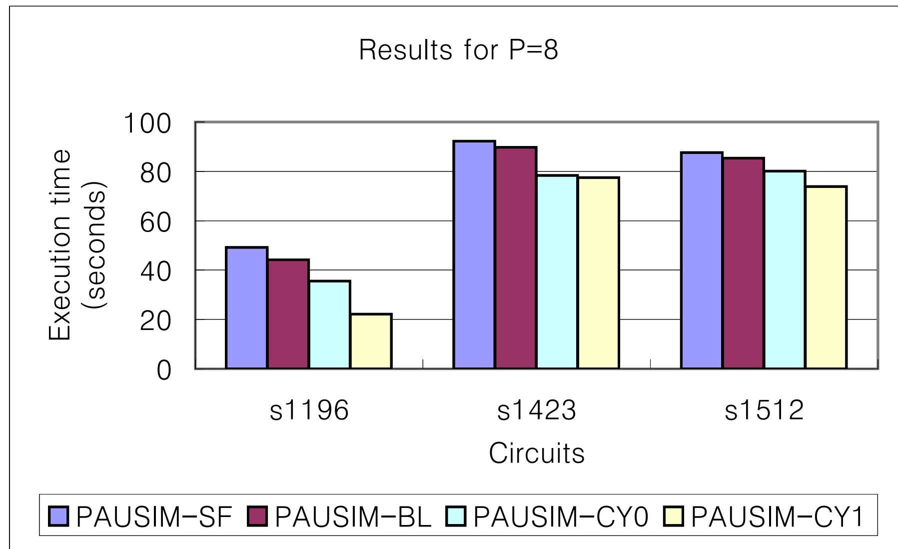


Figure 4.3: Execution time for 8 processors - small circuit

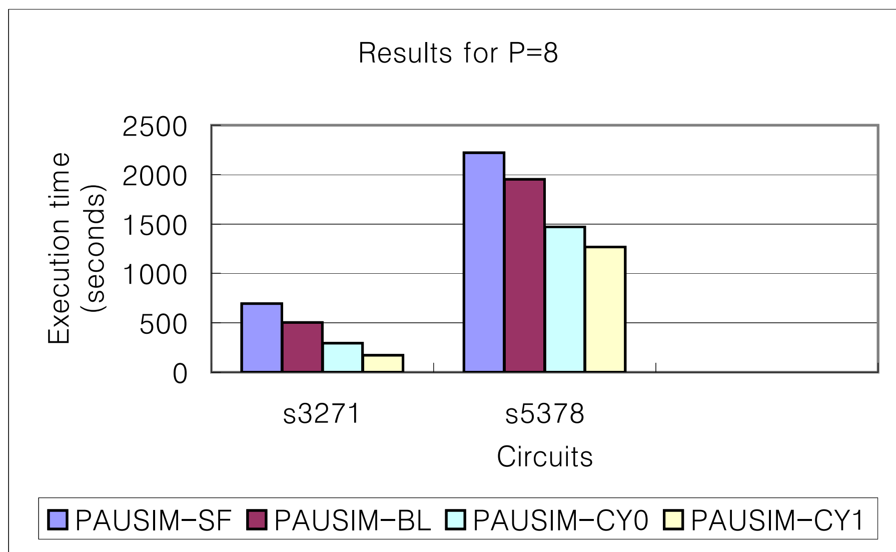


Figure 4.4: Execution time for 8 processors - large circuit

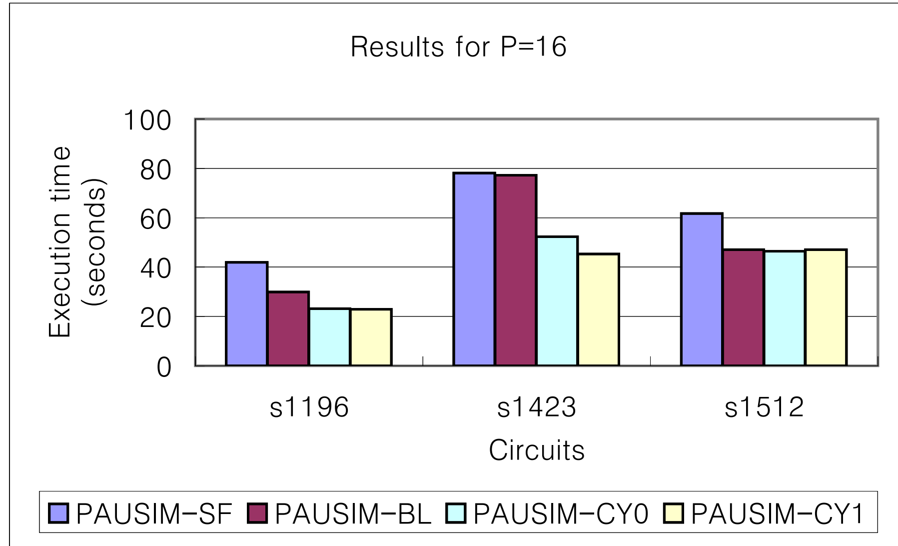


Figure 4.5: Execution time for 16 processors - small circuit

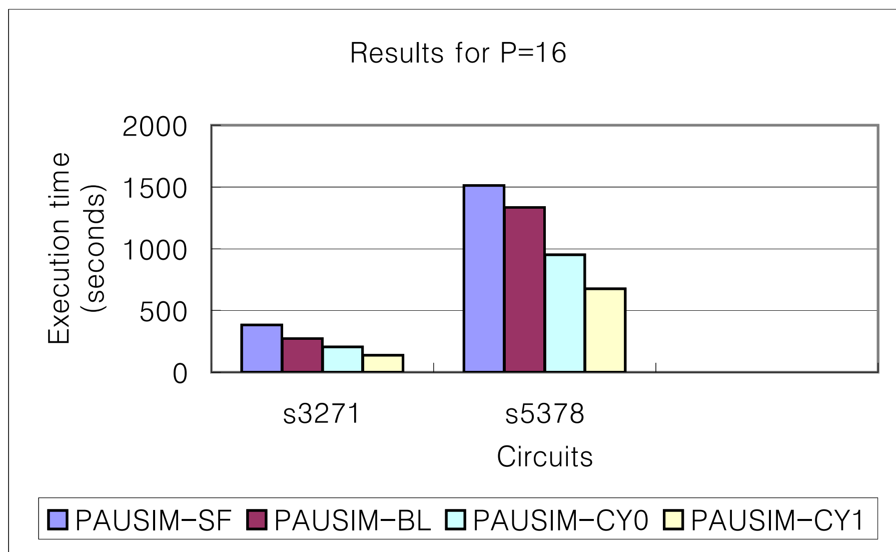


Figure 4.6: Execution time for 16 processors - large circuit

It can be seen that the proposed approach to task decomposition achieves shorter execution time (or higher speedup) than the existing approaches. As shown also in Figures 4.1 - 4.6, PAUSIM-BL performs substantially better than PAUSIM-SF, and PAUSIM-CY0 improves over PAUSIM-BL by the cyclic partitioning of faults instead of the block partitioning. As expected, PAUSIM-CY1 outperforms all other algorithms significantly mainly due to the two-step load balancing. It is also observed that the reduction in execution time is larger for a lager circuit. Note that a large circuit must have more room for improvement by a better load balancing scheme.

In Figures 4.7 - 4.18. execution time, speedup and efficiency are plotted as functions of the number of processors employed in parallel simulation. As number of processors increases, execution time monotonically decreases in all cases. One thing to note is that the improvement by using more processors tends to saturate in some cases of PAUSIM-SF as clearly seen in Figures. This is mainly due to the ineffective task partitioning. In the proposed algorithms, such saturation is either not observed or much less than in PAUSIM-SF.

It is to be mentioned that efficiency achieved by the proposed algorithms, especially PAUSIM-CY1, is very high in almost all cases. In addition to the effective parallelization in PAUSIM-CY1, the long simulation time (compared to the communication overhead) is another factor contributing to such high efficiency.

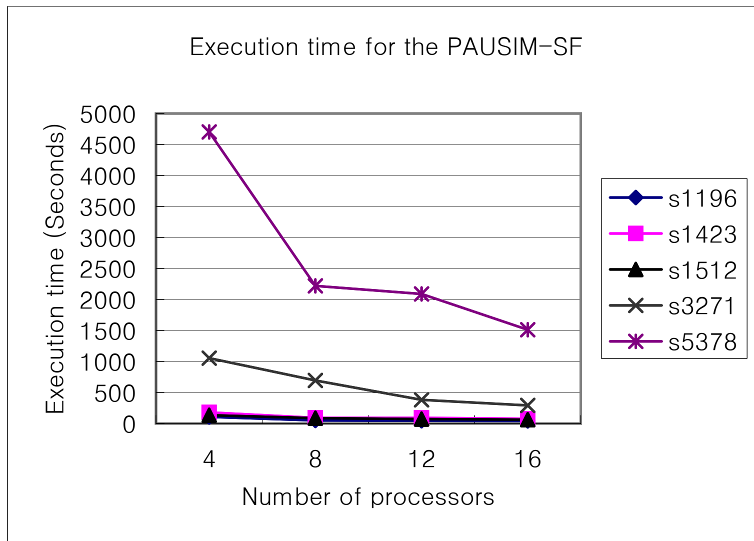


Figure 4.7: Execution times for PAUSIM-SF

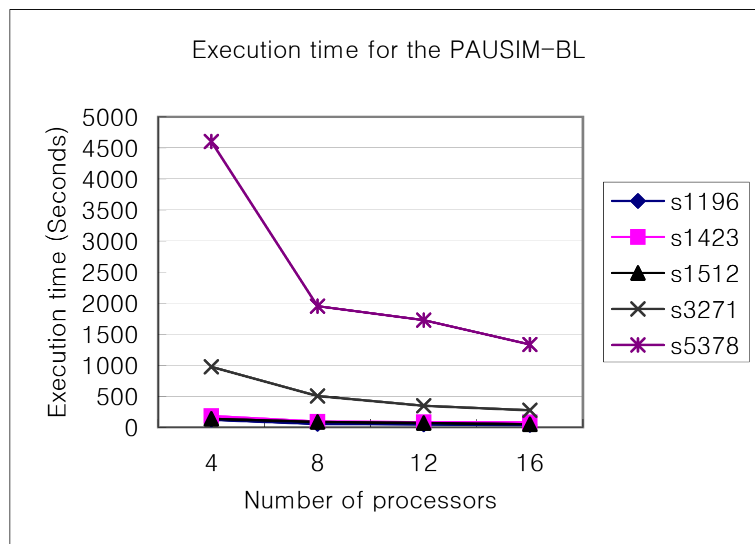


Figure 4.8: Execution times for PAUSIM-BL

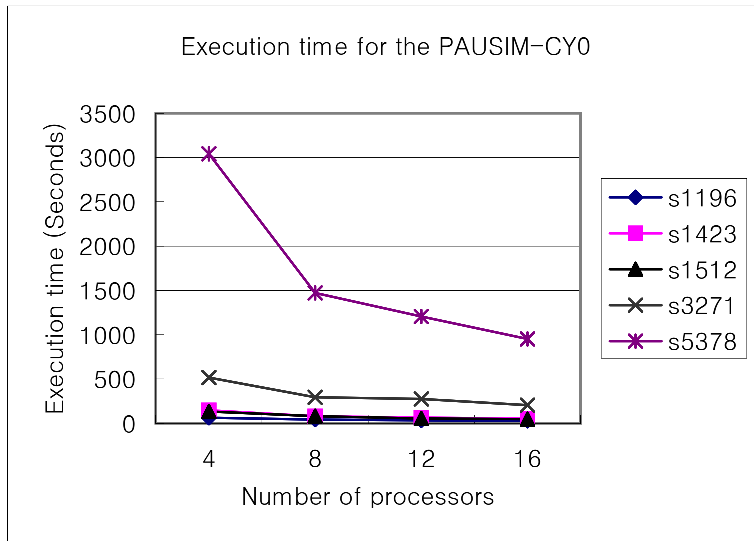


Figure 4.9: Execution times for PAUSIM-CY0

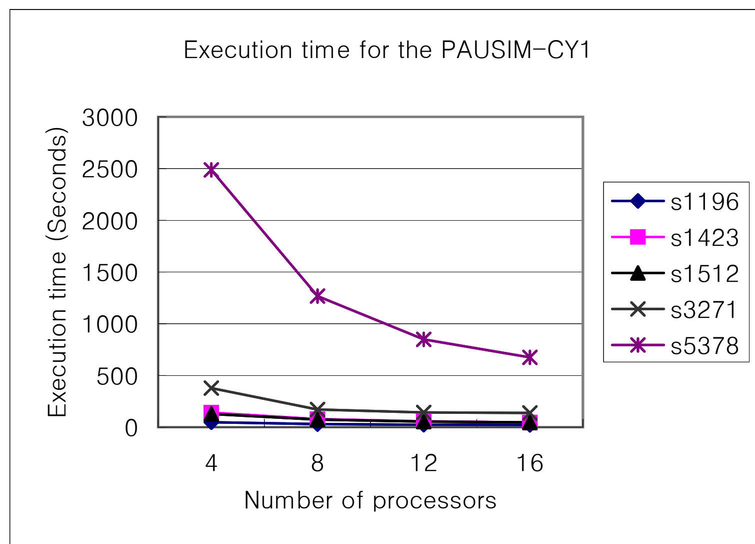


Figure 4.10: Execution times for PAUSIM-CY1

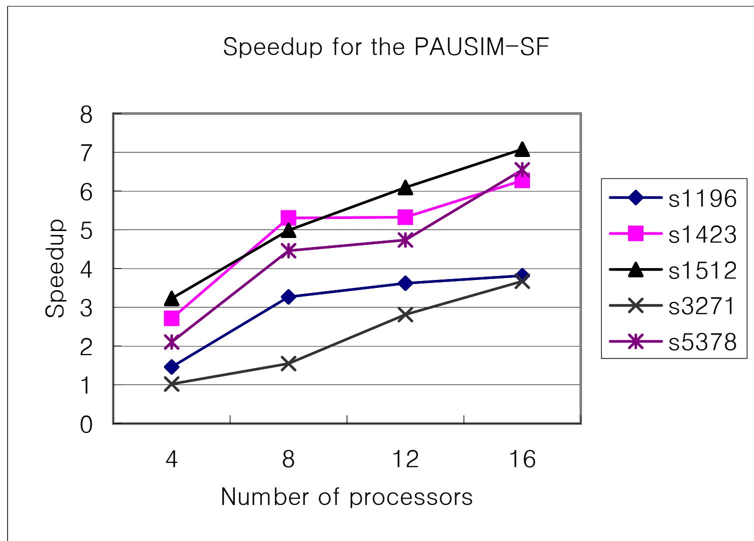


Figure 4.11: Speedup for PAUSIM-SF

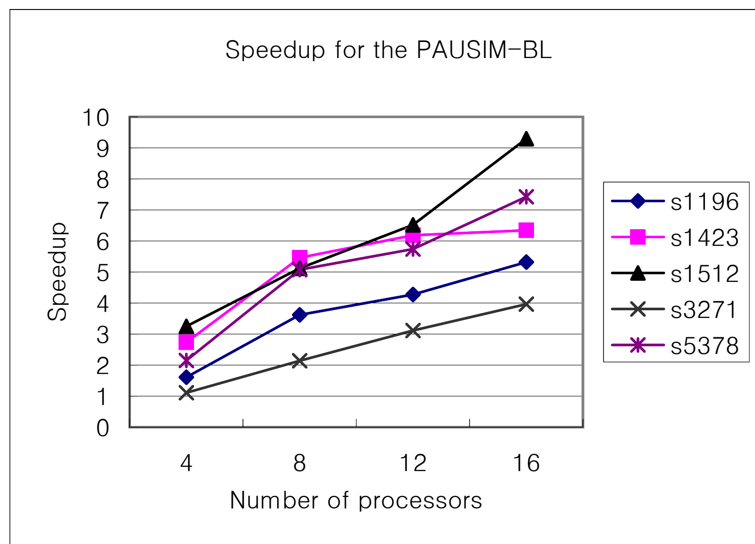


Figure 4.12: Speedup for PAUSIM-BL

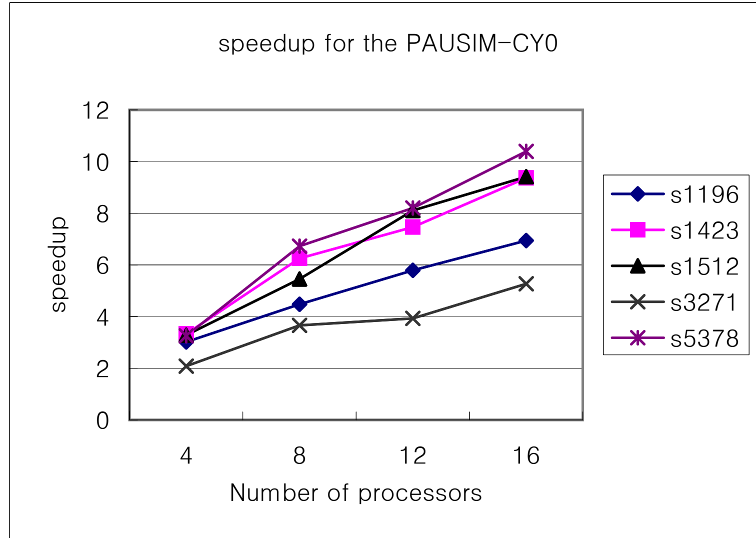


Figure 4.13: Speedup for PAUSIM-CY0

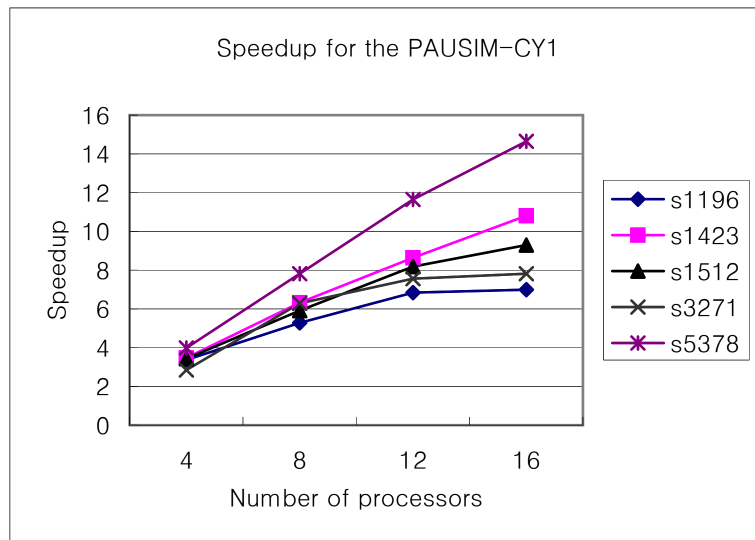


Figure 4.14: Speedup for PAUSIM-CY1

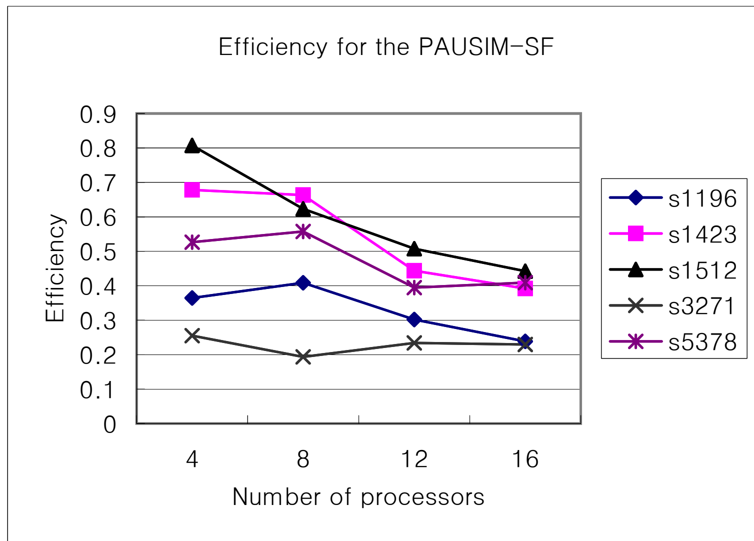


Figure 4.15: Efficiency for PAUSIM-SF

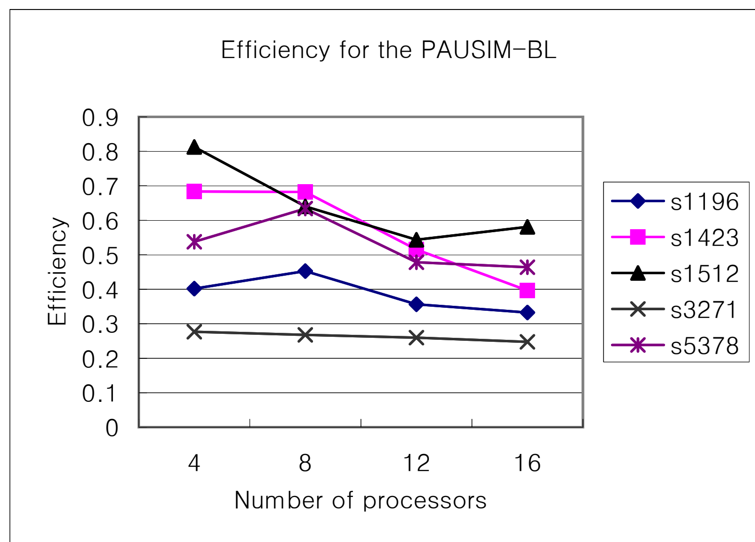


Figure 4.16: Efficiency for PAUSIM-BL

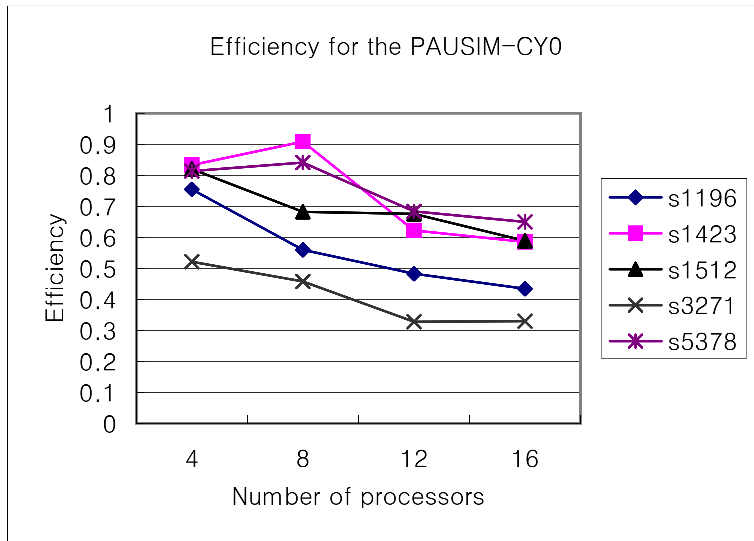


Figure 4.17: Efficiency for PAUSIM-CY0

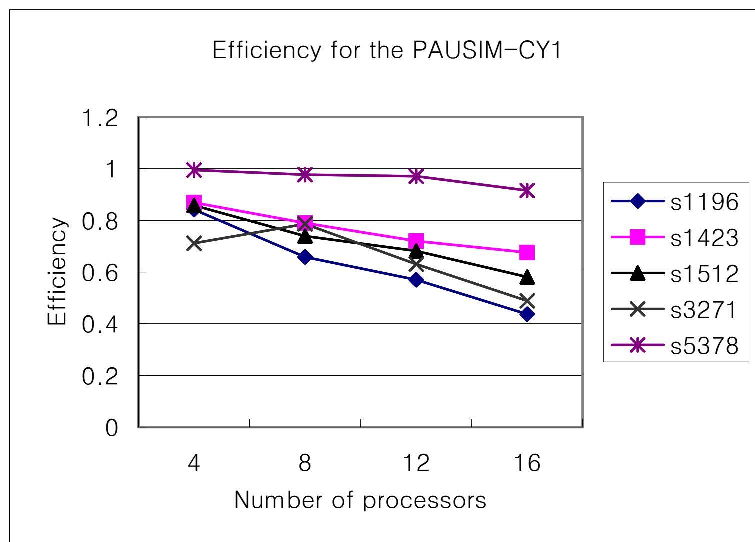


Figure 4.18: Efficiency for PAUSIM-CY1

In Tables 4.3 - 4.4, the minimum, maximum, mean, standard deviation, and normalized standard deviation of execution time over eight processors are provided in order to compare the four algorithms in terms of load balancing. A larger standard deviation of execution time among processors indicates a larger load imbalance among them. The logic simulation is a small fraction of the entire simulation and is well balanced over processors. What is to be noticed in the table is that the execution time significantly varies with processor in the case of PAUSIM-SF and PAUSIM-BL, leading to the relatively large standard deviation (also, the difference between the maximum and minimum execution time) in most cases. That is, the load is not well balanced in PAUSIM-SF and PAUSIM-BL. However, PAUSIM-CY1 greatly reduces the load imbalance and thereby achieves a significant performance improvement.

In Figure 4.19 - 4.26, execution times including communication times on individual processors are plotted for more detailed examination of load distribution among processors. It is confirmed that PAUSIM-CY1 balances the load over processors well at the expense of extra communication such that the overall parallel execution time is significantly reduced.

Table 4.3: Mean and standard deviation execution times on eight processors for PAUSIM-SF, PAUSIM-BL, PAUSIM-CY0 and PAUSIM-CY1. The unit for time is second. s1196, s1423 and s1512

Circuit	Algorithm	Step	Min.T.	Max.T.	Mean.T	Stdv	Norm.Stdv
s1196	PAUSIM	LSIM	0.8	1.1	0.9	0.10	0.12
	-SF	FSIM	18.2	43.3	36.9	5.13	0.14
	PAUSIM	LSIM	0.8	1.1	0.9	0.10	0.12
	-BL	FSIM	15.1	36.1	32.5	7.07	0.22
	PAUSIM	LSIM	0.8	0.9	0.8	0.05	0.06
	-CY0	FSIM	18.8	27.1	22.6	3.74	0.17
	PAUSIM	LSIM	0.8	0.9	0.83	0.05	0.06
	-CY1	FSIM	16.1	17.4	16.6	0.51	0.03
s1423	PAUSIM	LSIM	1.2	1.3	1.3	0.05	0.04
	-SF	FSIM	71.1	86.3	77.0	5.17	0.07
	PAUSIM	LSIM	1.2	1.3	1.3	0.05	0.04
	-BL	FSIM	73.5	84.0	77.3	4.36	0.06
	PAUSIM	LSIM	1.2	1.3	1.3	0.05	0.04
	-CY0	FSIM	60.9	72.4	66.1	3.97	0.06
	PAUSIM	LSIM	1.2	1.3	1.3	0.05	0.04
	-CY1	FSIM	60.5	69.9	66.0	3.06	0.05
s1512	PAUSIM	LSIM	1.3	1.3	1.3	0.00	0.00
	-BL	FSIM	59.9	82.1	68.5	4.82	0.07
	PAUSIM	LSIM	1.3	1.3	1.3	0.00	0.00
	-BL	FSIM	63.0	77.9	66.5	4.89	0.07
	PAUSIM	LSIM	1.3	1.3	1.3	0.00	0.00
	-CY0	FSIM	64.0	76.7	71.0	3.65	0.05
	PAUSIM	LSIM	1.3	1.4	1.4	0.1	0.10
	-CY1	FSIM	65.7	68.0	66.3	1.25	0.02

Table 4.4: Mean and standard deviation execution times on eight processors for PAUSIM-SF, PAUSIM-BL, PAUSIM-CY0 and PAUSIM-CY1. The unit for time is second. s3271 and s5378

Circuit	Algorithm	Step	Min.T.	Max.T.	Mean.T	Stdv	Norm.Stdv
s3271	PAUSIM	LSIM	4.1	4.4	4.3	0.10	0.02
	-SF	FSIM	345.3	684.8	477.7	110.67	0.23
	PAUSIM	LSIM	4.1	4.4	4.3	0.10	0.02
	-BL	FSIM	205.4	490.8	377.2	102.51	0.27
	PAUSIM	LSIM	4.1	4.4	4.3	0.10	0.02
	-CY0	FSIM	80.4	278.3	155.8	63.59	0.41
	PAUSIM	LSIM	4.1	4.4	4.3	0.09	0.02
	-CY1	FSIM	141.4	158.7	152.2	5.17	0.03
s5378	PAUSIM	LSIM	11.3	11.9	11.7	0.19	0.02
	-BL	FSIM	1711.4	2185.9.6	1923.3	132.92	0.07
	PAUSIM	LSIM	11.4	11.9	11.7	0.16	0.01
	-SF	FSIM	1586.9	1912.6	1827.0	122.00	0.07
	PAUSIM	LSIM	11.5	11.9	11.7	0.15	0.01
	-CY0	FSIM	1103.0	1438.6	1246.0	110.49	0.09
	PAUSIM	LSIM	11.3	11.9	11.7	0.19	0.02
	-CY1	FSIM	1197.9	1229.0	1219.8	13.36	0.01

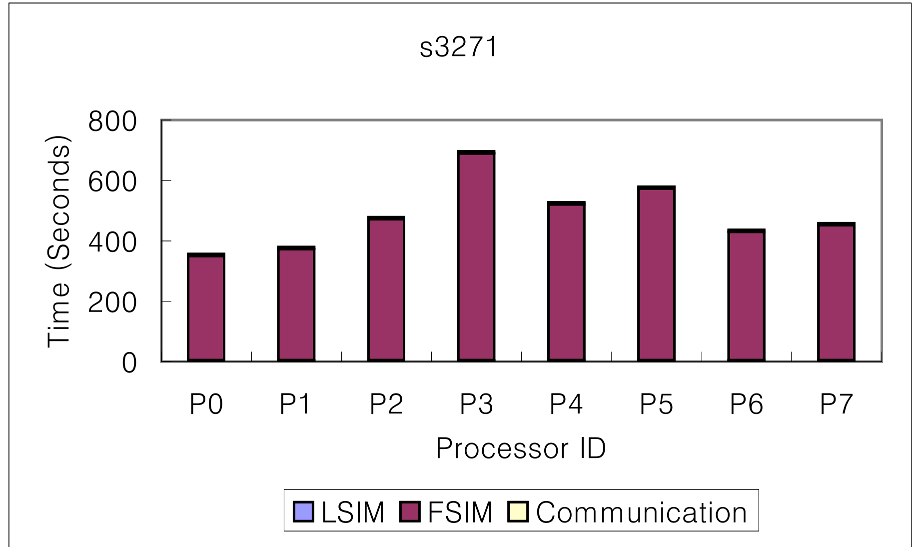


Figure 4.19: Workload distribution: s3271 benchmark circuit, PAUSIM-SF

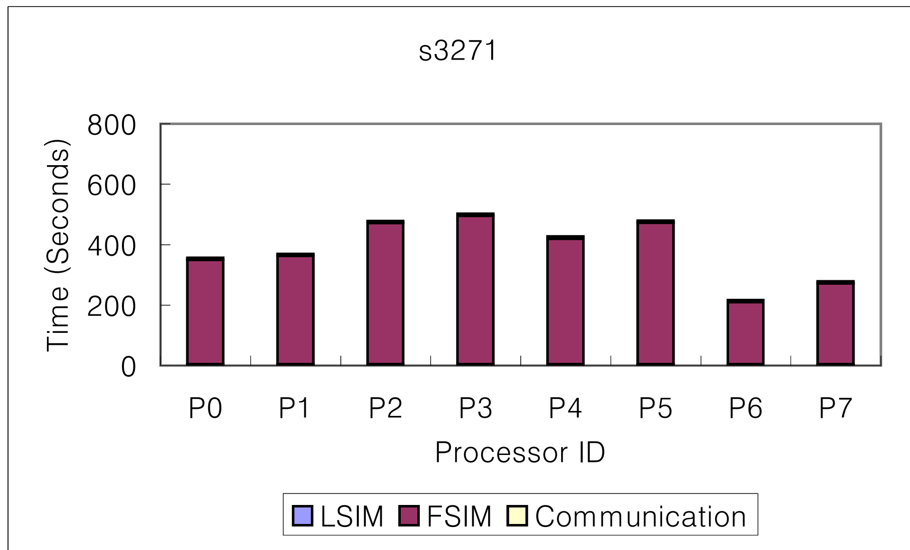


Figure 4.20: Workload distribution: s3271 benchmark circuit, PAUSIM-BL

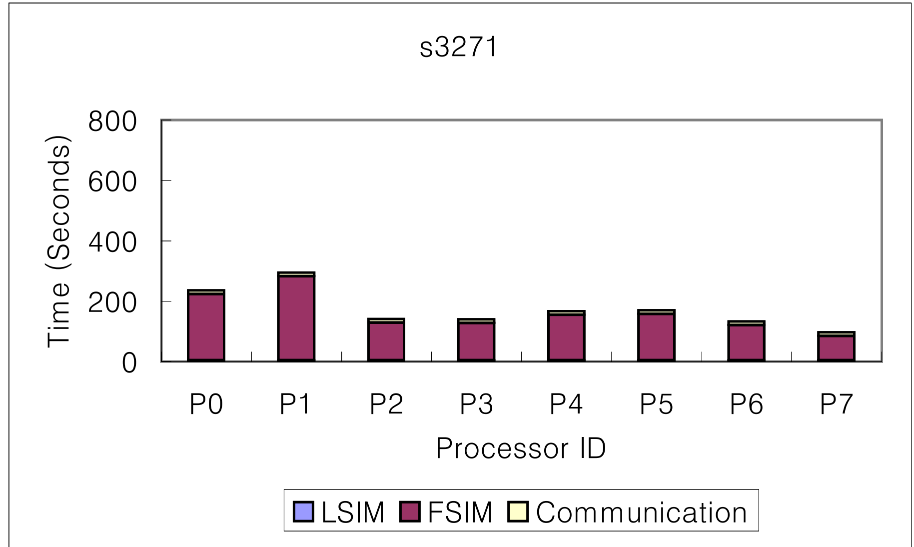


Figure 4.21: Workload distribution: s3271 benchmark circuit, PAUSIM-CY0

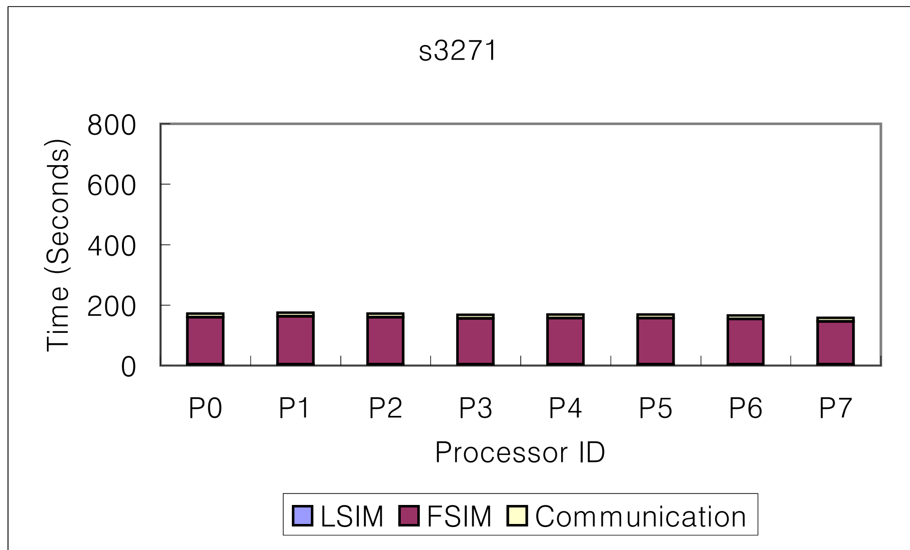


Figure 4.22: Workload distribution: s3271 benchmark circuit, PAUSIM-CY1

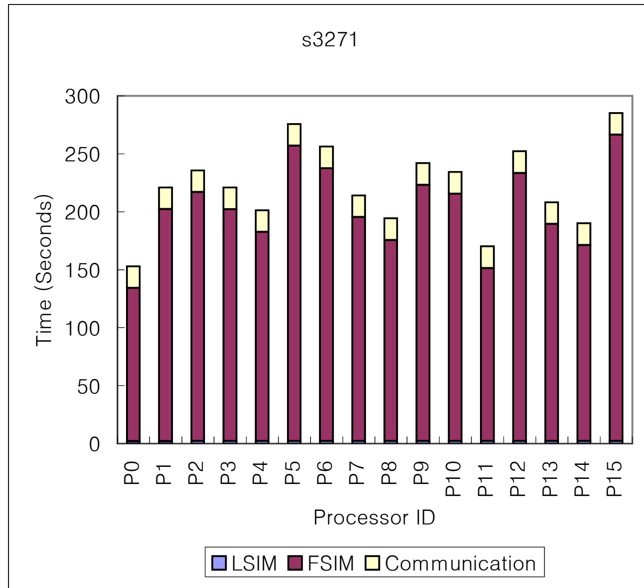


Figure 4.23: Workload distribution: s3271 benchmark circuit, PAUSIM-SF

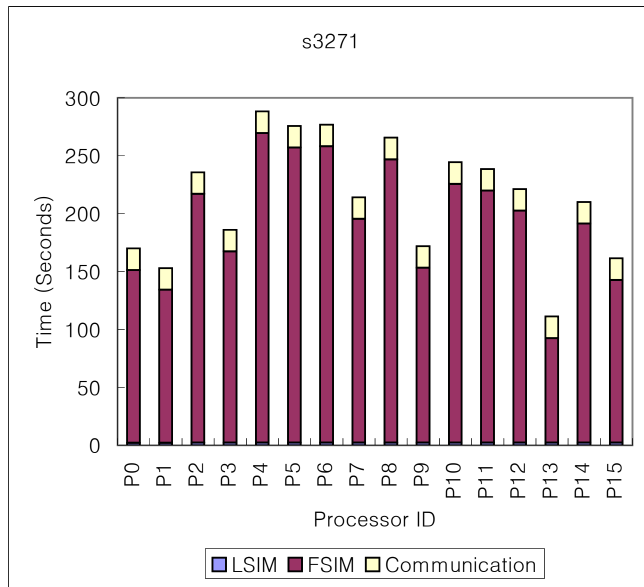


Figure 4.24: Workload distribution: s3271 benchmark circuit, PAUSIM-BL

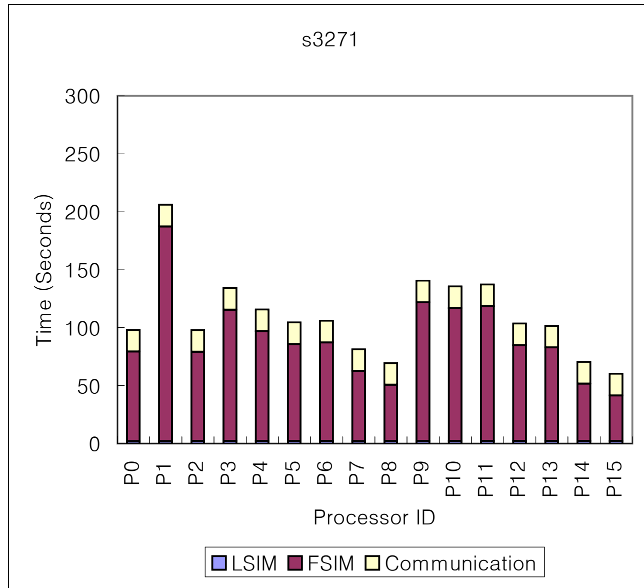


Figure 4.25: Workload distribution: s3271 benchmark circuit, PAUSIM-CY0

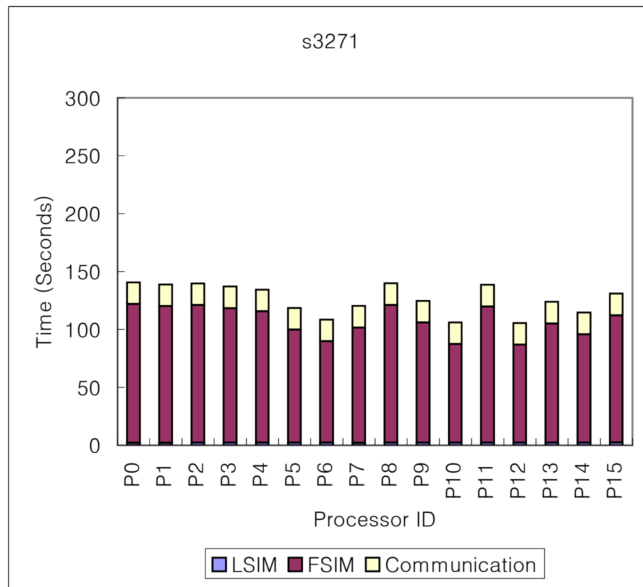


Figure 4.26: Workload distribution: s3271 benchmark circuit, PAUSIM-CY1

Chapter 5

Conclusion

Digital circuit testing including fault simulation is computationally intensive and therefore is a a good target application for parallel computing. In this thesis, efficient parallel fault simulation algorithms have been designed and implemented on a cluster of workstations. The proposed algorithms eliminates redundant simulation and reduces the actual number of unit simulation by partitioning faults among processors while assigned the entire test vector set to all processors. They, PAUSIM-CY0 and PAUSIM-CY1, achieve a better load distribution by adapting the cyclic partitioning of faults. In particular, PAUSIM-CY1 further improves the load distribution by allowing a load redistribution during fault simulation. The experimental results obtained on the 16-node cluster have demonstrated that the proposed parallel fault simulation algorithms can achieve significantly a shorter execution time than the existing algorithms.

Bibliography

[1] Michael L. Bushnell, Vishwani D. Agrawal, Essentials of Electronic Testing., Kluwer Academic Publishers.

[2] T. M. Niermann, W. -T. Cheng, and J. H. Patel, "PROOFS: A fast, memory-efficient sequential circuit fault simulator," *IEEE Trans. Computer-Aided Design*, pp. 198-207, February 1992.

[3] D. Harel and B. Krishnamurthy, "Is there hope for linear time fault simulation," *Proc. Fault Tolerant Computing Symp.*, pp. 28-33, June 1987.

[4] S. Seshu, "On an Improved Diagnosis Program," *IEEE Trans. Electronic Computing*, vol. EC-14, pp. 76-79, February 1965.

[5] D. B. Armstrong, "A deductive method for simulating faults in logic circuits," IEEE Transactions on Computers, vol. 21, pp. 462-471, May 1972.

[6] E. G. Ulrich and T. Baker, "The concurrent simulation of nearly identical digital networks," *Proc. Tenth Design Automation Workshop, vol. 6*, pp. 145-150. 1973.

[7] P. Goel, H. Lichaa, T. E. Rosser, T. J. Stroh, and E. B. Eichelberger, "LSSD fault simulation using conjunctive combinational and sequential methods," *Proc. Int. Test Conf.*, pp. 371-376, 1980.

[8] K. Kim and K. K. Saluja, "On fault deletion problem in concurrent fault simulation for synchronous sequential circuits," *Proc. VLSI Test Symp.*, pp. 125-130, 1992.

[9] D. G. Saab, "Parallel-concurrent fault simulation," *Trans. VLSI Systems*, vol. 1, no. 3, pp. 356-364, September 1993.

[10] W. -T. Cheng and M. -L. Yu, "Differential fault simulation - A fast method using minimal memory," *Proc. Design Automation Conf.*, pp. 424-428, 1989.

[11] H. K. Lee and D. S. Ha, "HOPE: An efficient parallel fault simulator for synchronous sequential circuits," *Proc. Design Automation Conf.*, pp. 336-340, 1992.

[12] H. K. Lee and D. S. Ha, "New methods of improving parallel fault simulation in synchronous sequential circuits," *Proc. Int. Conf. Computer-Aided Design*, pp. 10-17, 1993.

[13] J. A. Waicukauski, E. B. Eichelberger, D. O. Forlenza, E. Lindbloom and T. McCarthy, "Fault simulation for structured VLSI," *VLSI System Design*, pp. 20-32, December 1985.

[14] N. Gouders and R. Kaibel, "PARIS: A parallel pattern fault simulator for synchronous sequential circuits," *Proc. Int. Conf. Computer-Aided Design*, pp. 542-545, 1991.

[15] C-P Kung and C-S. Lin, "Parallel sequence fault simulation for synchronous sequential circuits," *Proc. European Conference on Desing Automation (EDAC-92)*, pp. 434-438, March 1992.

[16] R. Nair and D.S. Han, "Vision: An efficient parallel pattern fault simulator for synchronous sequential circuits," *Proc. 13th IEEE Test Symposium (VTS-95)*, p. 0221, 1995.

[17] M. B. Amin and B. Vinnakota, "ZAMBEZI: A parallel pattern parallel fault sequential circuit fault simulator," *Proc. VLSI Test Symp.*, pp. 438-443, 1996.

[18] P. Banerjee, parallel Algorithms for VLSI computer-Aided Design. Englewood Cliffs, NJ: PTR Prentice Hall, 1994.

[19] L. Soule and T. Blank, "Parallel logic simulation on general purpose machines," in *Proceedings of the 25th ACM/IEEE Design Automation Conference*, pp. 166-171, June 1988.

[20] R. B. Mueller-Thuns, D. G. Saab, R. F. Damiano,and J. A. Abraham, "Portable parallel logic and fault simulation," in *Digest of Papers, International Conference on Computer-Aided Design*, pp. 506-509, Nov. 1989.

[21] J. F. Nelson, "Deductive fault simulation on hypercube multiprocessors," in *Proceeding of the 9th ATT Conference on Electronic Testing*, Oct. 1987.

[22] S. Ghosh, "NODIFS: A noval, distributed circuit partitioning based algorithm for fault simulation of combinational and sequential digital designs on loosely coupled parallel processors," tech. rep., LEMS, Division of Engineering, Brown University, Providence, RI, 1991.

[23] S. Patil, P. Banerjee, and J. Patel, "Parallel test generation for sequential circuits on general purpose multiprocessors," in *Proceedings of the 28th ACM/IEEE Design Automation Conference*, (San Fransisco, CA), June 1991.

[24] P. Agrawal, V. D. Agrawal, K. T. Cheng, and R. Tutundjian, "Fault simulation in a pipelined multiprocessor system," *Proc. Int. Test Conf.*, pp. 727-734, 1989.

[25] S. Bose and P. Agrawal, "Concurrent fault simulation of logic gates and memory blocks on message passing multicomputers," *Proc. Design Automation Conf.*, pp. 332-335, 1992.

[26] S. Parkes, P. Banerjee, and J. Patel, "A parallel algorithm for fault simulation based on PROOFS, " *Proc. Int. Conf. Computer Design*, pp. 616-621, 1995.

[27] P. A. Duba, R. K. Roy, J. A. Rogers, "Fault simulation in a distributed environment, " in Proceedings of the 25th ACM/IEEE Design Automation Conference, pp. 686-691, June 1988.

[28] T. Markas, M. Royals, and N. Kanopoulos, "On distributed fault simulation," *IEEE Computer*, vol. 7, pp. 40-52, Jan. 1990.

[29] E. M. Rudnick and J. H. Patel, "Overcoming the serial logic simulation bottleneck in parallel fault simulation," *Proc 10th Intl. Conf. VLSI Design*, pp. 495-501, 1997.

[30] M. B. Amin and B. Vinnakota, "Zamlog: A parallel algorithm for fault simulation based on Zambezi," *Proc. Intl. Conf. on Computer-Aided Design.* pp. 509-512, 1996.

[31] D. Krishnaswamy, E. M. Rudnick, J. H. Patel and Prithviraj Banerjee, "SPITFIRE: Scalable Parallel Algorithms for Test Set Partitioned Fault Simulation." *Proc. 15th IEEE VLSI Test Symposium*, 1997.

[32] C. E. Stroud, "Using the Workstation version of AUSIM - Version 2.1."

[33] M. B. Amin and B. Vinnakota, "Data Parallel-Fault Simulation," *IEEE Trans. VLSI Systems*, vol. 7, NO.2 June 1999.