VLSI DESIGN
2001, Vol. 12, No. 1, pp. 81–99
Reprints available directly from the publisher
Photocopying permitted by license only

Distributed Fault Simulation Algorithms on Parallel Virtual Machine

C. P. RAVIKUMAR*, VIKAS JAIN and ANURAG DOD

Department of Electrical Engineering, Indian Institute of Technology, New Delhi 110016, India

(Received 1 August 1995; In final form 6 February 1997)

In this paper, we describe distributed algorithms for combinational fault simulation assuming the classical stuck-at fault model. Our algorithms have been implemented on a network of Sun workstations under the Parallel Virtual Machine (PVM) environment. Two techniques are used for subdividing work among processors – test set partition and fault set partition. The sequential algorithm for fault simulation, used on individual nodes of the network, is based on a novel *path compression* technique proposed in this paper . We describe experimental results on a number of ISCAS'85 benchmark circuits.

Keywords: Distributed algorithms; Fault simulation; Stuck-at fault model; Parallel processing; Fault coverage

1. INTRODUCTION

1.1. Fault Simulation

Fault simulation is a frequent activity during the process of VLSI test synthesis. Fault simulation is used to estimate the fault coverage of a circuit for a given test set, and hence evaluate (a) the effectiveness of the test set and (b) the testability of the circuit. If the fault coverage is found to be inadequate, then the either a different test set must be synthesized or changes must be made to the circuit to improve its testability. Unfortunately, fault simulation algorithms are compute-intensive. For large circuits, a single run of fault simulation can consume several hours of time. This motivates us to look for faster fault simulation alternatives. The existing techniques for speeding up fault simulation can be classified as

(1) algorithmic enhancements to achieve higher speed in fault simulation [10, 11]
(2) development of special-purpose hardware for fault simulation [3]
(3) development of parallel algorithms on general-purpose parallel computers for fault simulation [2, 9, 12, 13, 18].

*Corresponding author.

1.1.1. Algorithmic Enhancements

Jain and Agrawal [10] proposed Statistical Fault Analysis (STAFAN) to estimate the fault coverage of a circuit without actually performing fault simulation; their technique estimates the detection probability for each stuck-at fault using statistical techniques. However, such a technique cannot provide information regarding the faults that are detected (or undetected) by the given test set. The list of detected and undetected faults is useful in circuit design activity. For instance, the faults that are not detected by a large number of random tests are called *hard to detect* faults, and one can use design-for-testability techniques which are geared to address such hard-to-detect faults. Kumar *et al.* [11] presented an improved deductive fault simulation algorithm which has better time performance than conventional deductive fault simulation. But the reduction in execution time using these techniques is still not sufficient for larger circuits.

1.1.2. Hardware Accelerators

Fault simulation can be further speeded up by using special purpose hardware accelerators, *e.g.*, IBM's Yorktown Simulation Engine (YSE) and NEC's Hardware Accelerator (HAL) [3]. While hardware accelerators provide fast simulation capability, they have some severe drawbacks. Their initial cost is very high and they need specially trained staff for maintenance. Accelerators are inflexible as they support a specific algorithm. Moreover, they have fixed fault libraries, which are limited to one or a small number of technologies, and any major change in the prevailing technology may lead to their obsolescence.

1.1.3. Parallel Algorithms

An alternative to hardware accelerators is the use of distributed and/or parallel processing for fault simulation. A number of authors have reported parallel algorithms for various VLSI/CAD and testing applications on general-purpose parallel computers [2, 9, 12–14, 18]. While general-purpose parallel processors (such as parallel supercomputers) overcome the problem of inflexibility, distributed computing is more attractive than parallel supercomputing for several reasons. The recent upsurge in network-based computing is the chief reason for the popularity of distributed algorithms. VLSI design houses use a large number of engineering workstations which are networked together. It therefore makes sense to use the idle CPU cycles of the workstations by distributing a single job across a number of machines. This serves as a less expensive form of parallel processing in comparison to a general-purpose parallel super-computer or a special-purpose CAD accelerator. Programming environments for distributed computing have become available in the past few years [8]. They permit efficient implementation of large-granularity parallel processing. They provide automatic job scheduling, job migration and communication primitives. Heterogeneous computing is also supported by these programming environments. Sienicki *et al.* [16] have reported asynchronous algorithms for test generation for sequential circuits; their algorithms were programmed on a network of engineering workstations.

1.2. Our Approach

To achieve the above-mentioned objectives of speeding up the fault simulation process, we propose two different approaches which address these issues at different levels. The first approach tries to improve the speed of a sequential fault simulation algorithm and tries to achieve speedup by reducing the number of computations. This is achieved by applying a new concept of *path compression*. In the second approach, our objective is to employ *distributed processing* to the fault simulation problem. To partition the problem into several subtasks that can be computed in parallel, we have proposed and implemented two techniques, namely, *fault set partitioning* (FSP) and *test set partitioning* (TSP). We have also proposed and

compared several algorithms for each of the above partitioning techniques.

1.3. Organization of the Paper

We briefly review some of the earlier work in the area of distributed fault simulation in Section 2. In Section 3, we discuss a sequential fault simulator based on path compression. In Section 4, we describe our distributed fault simulation algorithms. In Section 5, we present experimental results of our simulation algorithms on a number of ISCAS'85 benchmark circuits [4]. Conclusions based on our experimentation are reported in Section 6.

2. PREVIOUS WORK

Duba *et al.* [6] proposed a distributed fault simulator, which uses a hierarchical partitioning algorithm. A straightforward data parallel approach was taken by the authors, where the fault list is partitioned into p subsets, where p is the number of processors. Each processor performs fault simulation on its private subset of faults on its private copy of the entire circuit. The partitioning algorithm creates a series of circuit representations (partitions), each of which is a complete representation of the circuit. Each partition considers a unique set of modules for fault injection, thereby partitioning the fault set. The algorithm uses a specified number of faults as a guide to create a partition with some amount of tolerance. Each partition considers only that part of a circuit in which faults are to be considered; the rest of the circuit is modeled at functional level of abstraction. A configuration is created by marking the nodes of a tree expansion of the circuit hierarchy. This tree data structure is also used to record the faults that have been simulated, the faults that are in current partition, the faults detected, and the undetected faults. Each higher level module contains the sum of the number of faults in all of its lower level constituent modules. This number is used in partitioning algorithm to decide if the module is an acceptable object to be considered for partitioning, or if the module contains too many faults and therefore must be broken into its constituent lower level modules.

The partitioning algorithm starts at the highest level in the circuit hierarchy and searches the tree data structure from left to right to find the modules which have not been incorporated into the partition. The first available module is checked against the recommended partition size. If it is within the tolerance limit specified, the module, all its descendants and all its direct ancestors are marked as fault enabled in current partitioning. All siblings of the module and the siblings of direct ancestors may be simulated by the functional modules if they exist. These modules are checked for the functional modules and if present, the module is marked for functional evaluation. This procedure continues until a complete circuit representation consisting of fault enabled and the functional modules is created. The partitions thus created are used for distributing the tasks as suggested by Duba *et al.* [6]. See Figure 1.

Patil and Banerjee [13] describe a parallel algorithm for combinational fault simulation on the iPSC/2 hypercube multiprocessor. Like Duba *et al.* [6], their approach is also to partition the fault list across processors. Four static fault partitioning algorithms were considered by the authors. The simplest fault partitioning algorithm is to randomly assign an equal number of faults to each processor. The remaining three approaches attempt to assign as many *compatible faults* to a processor as possible. A set of faults are said to be compatible to one another if they are all detected by the same test. By assigning compatible faults to the same processor, the authors reason that the overall execution time of the fault simulation algorithm will be reduced owing to increased rate of fault dropping. In the *Input Cones* fault set partitioning algorithm, all faults which lie on the same fanout cone of a primary input are assumed to be compatible. All faults on the fanout cone rooted at a randomly selected primary input are

```
procedure Distributed-Hierarchical-Fault-Simulation()
begin
        Client process determines the number of server processes available.
        Client process initializes the server processes.
        while there are simulations to perform {
                A server process sends results to the client process and requests
                a new partition number.
                The client process saves the results and provides a new partition
                number to the server process.
                The server process repartitions the circuit for the partition number
                it received.
                The server process simulates the faults contained in the partition.
        end
end
```

FIGURE 1 Distributed algorithm for hierarchical fault simulation.

assigned to the first processor. A second primary input is then selected and a fanout cone rooted at this input is found using depth-first-search; if a node is found to be visited previously, then the search procedure does not explore deeper into the circuit. Repeated application of the above algorithm is used to partition the fault list. In the *Output Cones* procedure, the faults that lie on a fanout cone rooted at a primary output are assumed to be compatible. Thus the *Output Cones* procedure is similar to the *Input Cones* procedure, except that fan out cones rooted at primary outputs are found. The above three algorithms are fast, requiring $O(n)$ time, where n is the number of gates in the network. The *Mandatory Constraint Propagation* (MACP) algorithm requires $O(n^2)$ time. The MACP procedure identifies, for any given fault f, a set of faults that are pseudo-compatible to f *i.e.*, faults which are likely to be compatible to f. The concept of global flow dominators is used to further enhance the partition into pseudo-compatible faults. The MACP procedure was found to be more effective. When considering a set of deterministically obtained test set, the number of faults is not an indication of the amount of time it may take to simulate all the faults. The authors of [13] verified this fact experimentally. Owing to the discrepancy of fault simulation times among processors which may originate due to the above observation, a load-imbalance may result and deteriorate the performance of the parallel algorithm. The authors tried dynamic reallocation of fault lists among processors to improve the performance of parallel fault simulation.

Mueller-Thuns, Saab, Damiano and Abraham [12] presented parallel algorithms for sequential logic simulation and sequential fault simulation on two general-purpose parallel computers, namely, the Intel iPSC/2, the Sequent Balance 21000 and Encore Multimax 320. The authors employed a circuit partitioning approach to speed up fault simulation. The objective of the circuit partition algorithm is to ensure that *highly connected elements* appear in the same partition. A simulation event generated during the simulation of subcircuit X must lead, as far as possible, to other simulation events which can be processed locally by the processor which deals with subcircuit A. This is required to minimize the number of synchronization events among processors. The partitioning algorithm used in [12] is to start with a randomly selected latch L in the circuit and use depth-first-search to identify a cone of predetermined size rooted at the input of L is found. This procedure is repeated until the entire circuit is covered by the subcircuits. Although this partition procedure is fast, there is no guarantee that it is optimal in terms of the number of synchronizations. A concurrent fault simulation algorithm was employed at each processor. Since a concurrent fault simulator is memory-intensive, there may be insufficient

memory at each processor node to hold the entire fault list. Therefore, the authors resort to multiple runs of the fault simulation algorithm to cover all the faults. Simple heuristics are used to decide how the fault list must be partitioned and considered sequentially at each node. Although general-purpose parallel processors can provide good speed-ups, they are expensive and are not easy to program. Debugging and program development facilities available for these machines are inadequate. In contrast, when we develop distributed algorithms for a network of workstations, the debugging and program development tools available for sequential code development can also be used for distributed programming. Furthermore, programming on a network of workstations is easier in comparison to programming a general-purpose multiprocessor.

Ghosh has reported a tool called NODIFS for distributed fault simulation of combinational and asynchronous sequential circuits [9]. The approach followed in [9] to parallelize fault simulation is to partition the circuit. Since each component in the partition is assigned to a different processor, and fault simulation can proceed concurrently on these components, a higher throughput can be realized. The author describes an asynchronous algorithm for distributed fault simulation which has been implemented on a loosely coupled parallel processor with up to 48 processors. The author has reported experimental results on small examples. It is unclear if the circuit partition-based asynchronous algorithm will be suitable for large circuits, since no experimental evidence to that effect is available in the paper.

We now review a recently reported sequential algorithm for fault simulation which has been shown to be highly effective. Kumar *et al.* [11] have suggested some new techniques for making deductive fault simulation [1] more efficient. Sensitivity analysis is used during true value simulation to classify each gate of the circuit into one of the two classes, namely, Definitely Sensitive (DS) and Potentially Sensitive (PS). A logic gate with all inputs at a non-controlling value is classified as *definitely sensitive*. Non-controlling values are 0 for OR/NOR gates and 1 for AND/NAND gates. Inverters and buffers are always DS gates. A logic gate with at least one input at a controlling value is said to be *potentially sensitive*. Controlling values are 0 for AND/NAND gates and 1 for OR/NOR gates. The propagating fault set (PFS) of a gate is defined as a set of faults in the circuit that would be detected at the output of the gate if its output were an observable primary output of the circuit. The PFS dynamically changes with input vectors. For a given test vector t, let v be the value at the output of a gate g when there are no faults. Let CI represent the set of all controlling inputs to g, and NCI represent the set of noncontrolling inputs to g. We denote the output of the gate also by g. If the gate g is a DS gate, the PFS of g is given by

$$\mathrm{PFS}(g) = \bigcup_{l \in NCI} \mathrm{PFS}(l) \bigcup \{g_{\bar{v}}\} \tag{1}$$

If the gate g is potentially sensitive, then PFS(g) consists of the difference between the intersection of PFS of all the controlling inputs to g and the union of PFS for all non-controlling inputs to g; added to this difference is any activated fault at the output of g. Mathematically,

$$\mathrm{PFS}(g) = \left(\bigcap_{l \in CI} \mathrm{PFS}(l) - \bigcup_{l \in NCI} \mathrm{PFS}(l) \right) \bigcup \{g_{\bar{v}}\} \tag{2}$$

Kumar *et al.* [11] made the above mentioned basic operations of set union, set intersection, and set difference used in fault list computation highly efficient by using pointer referencing on an implicitly ordered fault list.

3. FAULT SIMULATION BASED ON PATH COMPRESSION

In this paper, we propose a new fault simulation algorithm, which shows better time performance in comparison to an event-driven deductive fault simulator [1]. Our algorithm uses a technique called

"path compression" which gives it an edge over event-driven deductive fault simulation. Path compression takes advantage of the fact that fault simulation is performed in a levelized circuit, one level at a time. An additional feature of our technique is that fault dropping is carried out during fault simulation.

The circuit under test (CUT) is first levelized [7]. Path compression relies on the notion of *pipes* which we now define. For a given test vector t, a gate g at level l is known as a "pipe" for fault f if g is the only gate at level l through which fault f can propagate and no gates at level $l' > l$ have been scheduled for fault simulation when simulation for level l is to be started. The physical significance of marking a gate g at level l as a pipe is that thus far g is the only gate at level l which can propagate a given fault to a primary output. Note that the status of a gate can dynamically change from being a pipe to not being a pipe during the course of simulation even for the same test vector t. We shall associate a 0/1 variable $\phi(g,f,t)$ which is set to 1 if and only if g is a pipe for test t and fault f. The following lemmas are obvious.

LEMMA 1 *If the output of a gate g has a fanout of* 1 *and a fault f at the output of g is activated by a test t, then* $\phi(g,f,t)=1$.

LEMMA 2 *If the output of a gate g is a fanout stem and a fault f at the output of g is activated by a test t, then* $\phi(g,f,t)=1$.

We also introduce an attribute $\gamma(g,f,t)$ for a gate g which is a pipe for a fault f and test t. The attribute $\gamma(g,f,t)$ can be *visible* or *invisible*; if it is known that the fault f propagates to one of the primary outputs, we set $\gamma(g,f,t)$ to *visible*. Otherwise, the attribute is set to *invisible*. When the gate g is not a pipe for a fault f and a test t, the attribute $\gamma(g,f,t)$ is set to *unknown*.

The essential ideas behind path compression are the following theorems.

THEOREM 1 *During the fault simulation of a circuit for some test t, if it has been established that* $\phi(g,f,t)=1$, *then, for some other fault* f', $\phi(g,f',t)=1 \Rightarrow \gamma(g,f',t)=\gamma(g,f,t)$.

Proof See Appendix A.

THEOREM 2 *During the fault simulation of a circuit for some test t, if it has been established that* $\phi(g,f,t)=1$ and $\phi(g',f,t)=1$ then, $\gamma(g,f,t)=\gamma(g',f,t)$.

Proof See Appendix A.

Our sequential fault simulation procedure is illustrated in Figure 2 and is essentially an event-driven deductive fault simulator with the following important differences.

(1) Each fault is propagated individually, rather than sprouting all the faults at once and then propagating them, as done in deductive fault simulation.
(2) Path compression is used to quickly decide whether or not a fault is detected by a test. See the *SmartFaultSim* procedure in Figure 3.

We illustrate our fault simulation technique using the circuit shown in Figure 4(a). Suppose test vector under consideration is 1011 and the fault list includes b_1, (b-sa-1) and c_0 (c-sa-0). We use the notation l-sa-v to denote a stuck-at-v fault on line l. The algorithm first considers fault b_1. This fault is activated by the test 1011 and is visible at gate output G1. Deductive fault simulation establishes that the fault b_1, becomes visible at G3, G4, G5 and the primary output line j. The path of visibility for fault b_1 is shown using a dotted line in Figure 4(a). Gates G1, G3 and G6 are pipes for the test 1011 and fault b_1; however, the attributes of the pipes are unknown at this stage and the gates are put in the queue *PipeQueue*. After the simulation of fault b_1, these attributes are set to *visible*. Next, when we consider fault c_0, conventional deductive fault simulation establishes that c_0 is visible at G2; G2 is a *pipe*, but its attribute is still *unknown*; so G2 is placed in *PipeQueue*. Continuing the event-driven fault simulation, we discover that c_0 is *visible* at gate G2 and G3 is a pipe for c_0 as well. Since the attribute of G3 has been set to *visible* by the simulation of b_1 by Theorem 1, c_0 is

```
procedure FSIM (F,T,C)
{ F is the fault list, T is the test set, and C is the netlist for the combinational circuit }

begin
 Levelize the circuit C;
 for each fault f do f.detected := false ;
 for each test t do begin
    GoodCircuitSimulation (C,t);                    {Event driven logic simulation}
    for each gate g do
          γ(g,t) := unknown;        { g is not a pipe }
    for each fault f do
          if not (f.detected) then
                    if Activates(t,f) then begin
                              Let f = n-sa-v;
                              if n is a fanout stem or f has a fanout of 1 then
                                        enqueue(n,PipeQueue);
                              for each gate h which is fed by n do begin
                                        Let l be the level of gate h;
                                        enqueue (h,Queue,l);
                              end
                              SmartFaultSim (C,f,t,Queue,PipeQueue);
                                        {Use Path Compression}
                    end {if}
          end {if}
 end {for}
end {procedure}
```

FIGURE 2 Our fault simulation technique.

also visible at a primary output. Hence the event-driven fault simulation for c_0 can be terminated at gate G3 itself, and the conclusion that c_0 is detected by test 1011 can be drawn. The attribute of gate G2 can be set to *visible* (Theorem 2). In Figure 4(b), we show in dotted lines the (compressed) path along which gate evaluations are made during the simulation of c_0. The shaded region in Figure 4(b) is the part of the circuit for which no gate evaluations are made, indicating the reduction in computations.

4. DISTRIBUTED ALGORITHMS FOR FAULT SIMULATION

There are three ways to parallelize a fault simulation algorithm, namely, test partition, fault partition, and circuit partition. Test Set Partition (TSP) breaks the given test set T into smaller test sets $T_1, T_2, \ldots, T_p$ and assigns an individual task i to carry out a fault simulation for test set T_i. Fault Set Partition (FSP) decomposes the set of faults F into $F_1, F_2, \ldots, F_p$. An individual task i is assigned to simulate the circuit with fault list F_i, for all the tests. In both the above methods, each task must have a copy of the entire circuit. In the circuit partition technique, the circuit is broken into several subcircuits and these are assigned to independent tasks, thereby reducing the memory requirement of each task; however, it is a nontrivial task to partition the circuit to achieve load balancing among the tasks. The tasks must communicate with one another during the process of concurrent fault simulation, leading to overheads. Furthermore, the resulting parallel algorithm will be difficult to code and debug. In comparison, TSP and FSP are simpler to comprehend and easier to implement. Through experimental results, we found that TSP leads to more efficient parallelization.

```
procedure SmartFaultSim (C, f, t, Queue, PipeQueue)
{C is the netlist of a combinational circuit, f is a fault l-sa-v, and t is a test vector;
Queue is a queue of the pending events and PipeQueue is the queue of all the pipes that
have been encountered so far for test t and fault f.}
begin
 for each level l in {l1, l2, ···, lk}) do begin
          if some gate at level y > l is enqueued then flag := true ; else flag := false ;
          count := 0;
          while (Queue at level l not empty) do begin
                    g := dequeue(Queue, l);
                    if visible(f, g) then begin
                              for each gate h being fed by g do begin
                                        Let x be the level of gate h;
                                        enqueue (h, Queue, x);
                              end ; {for}
                              count++; PotentialPipe := g;
                    end {if};
          end {while} ;
          if (count = 1 and flag = false ) then
          case
                    γ(g,t) = unknown : enqueue(PotentialPipe, PipeQueue);
                    γ(g,t) = visible : f.detected := true ;              {Theorem 1 }
                              while (PipeQueue not empty) do begin {Theorem 2}
                                        p := dequeue(PipeQueue); γ(p,t) := visible;
                               end ;
                              return ; {return from procedure}
                    γ(g,t) = invisible : f.detected := false ;          {Theorem 1}
                              while (PipeQueue not empty) do begin {Theorem 2}
                                        p := dequeue(PipeQueue); γ(p,t) := invisible;
                              end ;
                              return ; {return from procedure}
          endcase ;
 end {for};
 if f is visible at a primary output then begin
          f.detected := true ;
          while (PipeQueue not empty) do     {Theorem 2}
                    p := dequeue(PipeQueue); γ(p,t) := visible;
           end ;
          return ; {return from procedure}
 end else begin
          f.detected := false ;
          while (PipeQueue not empty) do     {Theorem 2}
                    p := dequeue(PipeQueue); γ(p,t) := invisible;
          end ;
          return ; {return from procedure}
 end {for }
end {of procedure}
```

FIGURE 3 Fault simulation by path compression for a given fault f.

4.1. Test Set Partition

Given a combinational circuit C, a fault list F, and a test set T, let $D_{F_{T_i}}$ be the set of faults that are detected through a fault simulation on task i. The TSP-based fault simulation algorithm can be described as in Figure 5. Each task i further subdivides its own test set T_i into $(|T_i|/m)$ smaller subsets T_{ij}, $1 \leq j \leq (|T_i|/m)$. Let the size of T_{ij} be indicated by m, also known as Test Partition Parameter (TPP). At the end of fault simulation for subset T_{ij}, the resulting set of detected faults DF is broadcast to all the other tasks. Each task gathers the faults detected by all the tasks at the end of j-th fault simulation step into the set ND_j;

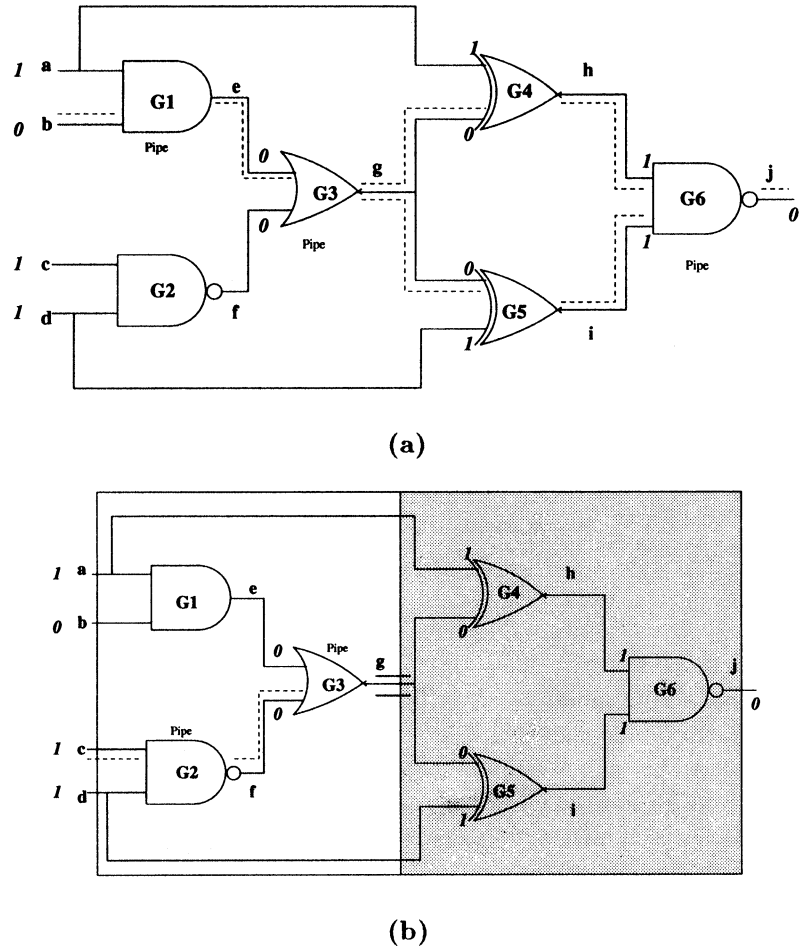

FIGURE 4 Illustration of our fault simulation technique. (a) Fault b_1; (b) Fault c_0.

these faults are then dropped from the local copy of the fault list F_i. The performance of the algorithm depends critically on the choice of the TPP. A large value of m indicates poor load balancing and a small value of m would mean a larger interprocessor communication.

In a distributed environment, the algorithm can be implemented by spawning a separate task for each of the p fault simulations. The overheads in the algorithm are of partitioning the test set, forming the union of the detected fault sets, and broadcast communication. In our implementation, we use the pvm-spawn function to create the tasks. We make copies of the netlist file and pass the file names as arguments to the tasks. We used random partitions of the test set; we also experimented with a constructive test partition procedure described later in this section.

4.2. Constructive Test Set Partition Procedures

The performance of the distributed algorithm will be affected by the procedure for TSP. To illustrate this, consider there are two tasks. Suppose that the first task receives a subset of test vectors which detect a large number of faults, and the second task receives vectors which do not detect many faults. Since the two tasks exchange their detected

```
procedure TestPartition(F, C, T)
begin
  TestSetPartition(T, p);
  for i = 1 to p do in parallel begin
        F_i = F; {Master fault list}
        DF_{T_i} = φ; {Faults detected by task i}
        for j := 1 to |T_i|/m do begin
              DF = FaultSimulate(C, F_i, T_{i,j}); {Simulate for j-th test subset}
              Broadcast(DF);
              for k := 1 to p-1 do
                    NonBlockingReceive ND_{j,k}
                    ND_j = ( ⋃_{k=1}^{p-1} ND_{j,k} ⋃ DF        {Collate results}
                    F_i = F_i − ND_j {Drop faults}
                    DF_{T_i} = DF_{T_i} ⋃ ND_j {Update list of detected faults}
        end {for}
  end {for}
  DF = ⋃_{i=1}^{p} DF_{T_i}
  FC = |DF|/|F|
end { procedure}
```

FIGURE 5 Test set partition.

fault lists at the end of every m test vectors, in the above case, the second task is essentially idling and the fault simulation is essentially serial. To avoid this situation, we must partition the tests such that the number of faults detected by each test subset is more or less equal. There is no easy way to count the number of faults detected by a test vector t. On the other hand, it is possible to count the number of faults which are activated by a test t. Our test set partition procedure is based on the presumption that a test vector which activates a large (small) number of faults is also likely to detect a large (small) number of faults. Finding the count of activated faults requires a good-circuit simulation for test t. We sort the test vectors on the *activated faults count*. The test vectors $t_{p\cdot j+i}$ are assigned to partition T_i, $0 \le i \le p$, $0 \le j < (|T|/p)$.

Since logic simulation is expensive, an alternate approach is described for test set partition. Here we partition the test set randomly into $(|T|/m)$ subsets of m vectors each; this partition is carried out randomly. For each subset i of size m, we find $A_i = \sum_{f \in F} a_f$, where a_f is the probability of activating a fault $f \in F$. See [15] or an explanation of activation probabilities. We sort the subsets on the values of A_i. The test subsets $p \cdot j + i$ are assigned to partition T_i, $0 \le i < p$, $0 \le j < (|T|/p \cdot m)$. A third partition heuristic is to treat each test vector as a k-input word, where k is the number of primary inputs, and sort the test vectors in ascending order. Then the test vectors $t_{j+p\cdot i}$ are assigned to partition T_i, $0 \le i < p$, $0 \le j < (T/p)$.

4.3. Fault Set Partition

In this technique, we partition the master fault list F across p tasks. Each task performs a fault simulation on the entire test set T. The complete algorithm is shown in Figure 6. In comparison to the TSP technique, the overhead in the FSP scheme is much less. The overheads include the partition of the fault set and the final collation of the detected faults.

We note that the FSP scheme duplicates a certain amount of work among the p tasks; each task must perform logic simulation on every test vector. The division of the fault list affects the load balancing factor in this scheme. To illustrate this point, consider the case when there are two tasks and the first task gets a list of easy-to-detect faults whereas the second task receives the remaining hard-to-detect faults. The first task may complete is work when 100% fault coverage is obtained after a small number of test vectors have been simulated. In contrast, the second task will have to wade through

```
procedure FaultPartition(F, C, T)
{ F is the fault list, T is the test set, and C is the netlist for the combinational circuit }
begin
        FaultSetPartition(F,p);
        for i = 1 to p do in parallel
                DF_i = FaultSimulate(C, F_i, T);
        DF = ⋃_{i=1}^{p} DF_i;
        FC = |DF| / |F|
end
```

FIGURE 6 Fault simulation based on fault set partition.

the entire list of test vectors. This load imbalance situation can be improved in two ways. Firstly, we can implement the distributed algorithm using the master-slave paradigm, where the master task maintains the master fault list and hands out subsets of the fault list to the slave tasks for simulation; each slave reports to the master as soon as its work is completed, and the master keeps the slaves busy by redistributing the faults when a slave task completes. In the above example of two tasks, the master task will take away some of the hard-to-detect faults from the second task and send them to the first task when the latter completes its work.

The second load balancing technique is to perform a more careful partitioning of the fault list. We experimented with a constructive FSP scheme which is described in the next section. The results of random FSP and constructive FSP are compared in Section 5.

4.4. A Fault Partition Procedure

Let $f = l$-sa-v be a stuck-at fault at line l. In order to activate the fault, the line l must be driven to value $\bar{v}$. The signal probability of a line l, denoted by p_l, is defined as the probability that a random test vector will set l to 1. Thus the probability that a random test vector will activate f is equal to a_f, where

$$a_f = v \cdot (l - p_l) + (l - v) \cdot p_l \tag{3}$$

The procedure for fault set partition, shown in Figure 7, tries to equalize the quantity $\sum_{f \in F_i} a_f$, for all fault subsets F_i. In doing so, we reason that

```
procedure PartitionFaults(C, F, T)
{ F is the fault list, T is the test set, and C is the netlist for the combinational circuit }
begin
        for each line l ∈ C do
                compute p_l; {See text for explanation}

        for each fault f ∈ F do { Let f = l-sa-v}
                a_f = v(1 − p_l) + (1 − v)p_l;          { Activation Probability for f}
        Sort fault list on activation probabilities;
        i = 0;
    j = 1;
        while (F ≠ φ) do begin
                F_i = F_i ⋃ {f_j};
                F = F − {f_j};
                j = j + 1;
                i = (i + 1) mod p;
        end
end {procedure}
```

FIGURE 7 Fault set partition procedure.

the faults in each subset are equally hard-to-activate. The computation of signal probabilities is known to be a difficult problem [15]. We used the following heuristic to compute the signal probabilities. Let t_l be the value of the l-th bit in test vector t. The signal probability for a primary input line l was computed as $p_l = (\sum_{t \in T} t_l / |T|)$. The signal probabilities of an internal line l which is an output of some gate g can be computed with a knowledge of the functionality of the gate g. Simple algebraic formulae relate the signal probabilities of the output with those of the inputs for each of the basic logic gates [17] as shown in Table I. We scan the signal lines of the circuit in a topologically sorted order to compute their signal probabilities.

4.5. Test Vector Ordering

The ordering of test vectors during fault simulation affects the performance of an event-driven deductive fault simulation. This is because the number of list events that are generated when a new test vector t is applied will depend on the test vector which preceded t. Savings in computation time can be obtained by ordering the test vectors so as to minimize the total expected number of fault list events that are generated. It may be mentioned in passing that a similar ordering of test vectors will also minimize the amount of switching activity in the circuit and hence reduce the power dissipation in the circuit. Therefore, finding such an ordering will also be useful in test application targeted towards low power. We suggest a heuristic technique to order the test vectors so as to minimize the number of fault events that are generated at the primary inputs. It is a reasonable assumption that the number of fault list events at the primary input is correlated to the number of fault list events in the entire circuit. The Hamming distance between two successive test vectors t_i and t_{i+1} denoted by $H(t_i, t_{i+1})$ measures the number of fault list events caused due to application of the two vectors in succession. Thus our heuristic procedure is to find an ordering which minimizes $\sum_i H(t_i, t_{i+1})$. This can be done by computing the grey code number $G(t_i)$ for each test vector t_i, and then sorting the test vectors on the G values. The computation of $G(t)$ requires $O(k)$ time, where k is the number of primary inputs. Thus the total time expended in ordering the test vectors is $O(|T|k + |T| \log |T|)$.

TABLE I Expressions for signal probabilities for gate outputs

Gate type	Signal probability of input A	Signal probability of input B	Signal probability of output C
Inverter $C = \bar{A}$	p_A		$1 - p_A$
2-input AND $C = A \cdot B$	p_A	p_B	$p_A \cdot p_B$
2-input OR $C = A + B$	p_A	p_B	$p_A + p_B - p_A \cdot p_B$
2-input XOR $C = A \oplus B$	p_A	p_B	$p_A + p_B - 2 \cdot p_A \cdot p_B$

5. IMPLEMENTATION DETAILS AND RESULTS

The fault simulator was implemented in C programming language on a network of SUN SPARCstations. In the experimental environment, there are 6 SUN SPARCstation 10 computers and 2 SUN SPARCstation 20 machines. We included only the SPARCstation 10 machines when conducting the experiments. The sequential as well as the distributed algorithms have been benchmarked on ISCAS'85 circuits [4]. Our distributed fault simulator was implemented using PVM [8].

In Tables II, III, and IV, we compare our fault simulation technique and the event driven deductive fault simulation technique for several ISCAS'85 benchmark circuits. Table II compares the performance of the event-driven deductive fault simulator with the fault simulation algorithm proposed in this paper. We considered two variants of our algorithm, namely, with and without path compression. We considered 2000

TABLE II Comparison of three fault simulation techniques. 2000 test vectors

Benchmark circuit	Event driven deductive fault simulator (s)	Our fault simulator (with path compression) (s)	Our fault simulator (without path compression) (s)	Fault coverage (%)
c432	39.52	8.80	7.40	90.75
c499	51.30	9.06	8.51	96.67
c1355	615.00	211.40	616.73	0.00
c1908	250.90	95.55	123.91	47.17
c2670	619.15	120.51	195.11	69.19
c3540	1035.05	101.17	97.45	67.51
c5315	2000.39	171.77	255.88	86.90
c6288	3513.91	95.64	121.11	89.10
c7552	4100.35	305.00	397.25	81.00

TABLE III Comparison of three fault simulation techniques. 6000 test vectors

Benchmark circuit	Event driven deductive fault simulator (s)	Our fault simulator (with path compression) (s)	Our fault simulator (without path compression) (s)	Fault coverage (%)
c432	96.25	17.98	16.11	93.57
c499	147.30	23.40	23.36	98.65
c1355	1912.15	836.34	1563.73	50.00
c1908	901.29	275.24	313.18	52.71
c2670	1519.47	352.76	379.47	61.42
c3540	2893.89	293.31	282.33	74.61
c5315	6799.22	468.79	610.28	86.93
c6288	10132.55	252.61	293.30	99.11
c7552	12591.93	686.53	768.42	86.83

TABLE IV Comparison of path-compression based fault simulation with event-driven fault simulation. 2000 test vectors

Benchmark circuit (ISCAS'85)	Event driven deductive simulator (s)	Our fault simulator fault (with path compression) (s)	Fault coverage (%)
c432	24.70	15.14	53.86
c499	34.48	12.94	57.91
c1355	–	296.23	50.00
c1908	192.61	108.40	46.77
c2670	30.10	194.48	8.71
c3540	944.28	169.12	46.51
c5315	70.93	403.34	9.93
c6288	1614.30	580.16	28.10
c7552	3045.26	479.81	29.07

random test vectors in Table II. Table III shows similar results for 6000 random test vectors. As can be seen from these tables, our algorithms perform better than event-driven fault simulation for almost all the circuits considered. As the fault coverage increases, our fault simulator outperforms the event-driven deductive fault simulation by a significant margin due to the fault dropping feature in our algorithm. For example, in the circuit c6288, when 6000 test vectors are applied, achieving a fault coverage of about 90%, our technique gives a speedup of about 40 times over the other technique. But our technique suffers from the disadvantage that if fault coverage is very

low, then it performs poorly as compared to event-driven deductive fault simulation. But this problem is mainly due to the style of implementation of our algorithm. We allocate and deallocate memory for queues while scheduling the gates for simulation and this is done a large number of times, about $O(tfg)$ times, where t is the size of the test set, f is the size of the fault set, and g is the number of gates in the circuit. If implemented using static allocation techniques (say arrays), the above overhead can be considerably reduced.

In Tables II and III, we have also compared the performance of the path compression technique with that of a fault simulator which does not use path compression. We see that our technique performs better for all the circuits sparing a few where the difference in execution time is not very large. When the fault coverage is very large the pipes with attribute invisible spring into action and give good results. As can be seen from the table, for the fault simulation of circuit c1355 using 6000 random test vectors, our technique takes almost half the time as compared to the other technique which doesn't uses path compression. Similarly for time high fault coverage, the pipes with attribute visible come into action and speedups the fault simulation.

5.1. Distributed Fault Simulation

5.1.1. Test Set Partitioning

Table V shows the results of the distributed fault simulation algorithm when random test set partition technique (RTSP) is used. We show the run time as well as speedup for varying number of partitions. The results are shown for three ISCAS'85 benchmark circuits, namely, c432, c3540, and c7552. Table VI shows the results when constructive test set partitioning (CTSP) is employed. In this scheme, we have sorted the test vectors in the increasing order of their equivalent decimal values and then divided the test set into chunks. In Tables V and VI, the TPP is set to 1. As can be observed from Tables V and VI, both constructive and random test partition exhibit sub-linear speedup. In both cases, the rate at which the speedup increases with the number of processors, *i.e.* (dS_p/dp) drops off for larger values of p. For example, in the case of circuit c432, the speedup

TABLE V Results for random test set partition scheme. 2000 test vectors

No. of	c432		c3540		c7552	
processors	Run time(s)	Speedup	Run time(s)	Speedup	Run time(s)	Speedup
1	7	1	92	1	263	1
2	4	1.75	53	1.74	166	1.58
3	3	2.33	37	2.49	103	2.55
4	2	3.50	28	3.29	85	3.09
5	2	3.50	26	3.54	76	3.46
6	2	3.50	24	3.83	61	4.31

TABLE VI Results for constructive test set partition scheme. 2000 test vectors

No. of	c432		c3540		c7552	
processors	Run time(s)	Speedup	Run time(s)	Speedup	Run time(s)	Speedup
1	8	1.0	97	1.00	263	1.00
2	5	1.6	61	1.59	143	1.84
3	3	2.67	41	2.37	98	2.68
4	2	4.00	35	2.77	79	3.33
5	2	4.00	30	3.23	76	3.46
6	2	4.00	28	3.46	73	3.60

reaches the maximum value of 3.5 for $p=4$ in Table V and does not increase thereafter. Similarly, the speedup in case of c432 reaches a value of 4.0 in Table VI for $p=4$ and does not increase thereafter. In the case of c3540, when the RTSP scheme is employed, the speedup increases from 1 to 3.29 when p is increased from 1 to 4; when two more processors are added, the increase in speedup is not proportional, reaching a value of 3.83. In all the cases, constructive test set partition shows better results than random test set partition. The speedup when p partitions are used with p slave processors is given by

$$S_P = \text{Time for Sequential Fault Simulation on Host Processor}/(\text{Time for Test Set Partition} + \text{Max}_{i=1}^{p}(\text{Fault Simulation Time on Processor } i)) \quad (4)$$

The time for partitioning the test set is clearly an overhead, and is the one of the causes for sublinear speedups. This overhead is significant for smaller circuits such as c432. The I/O time spent by each slave task in reading the circuit netlist and setting up the data structures affects the speedup.

We also studied the effect of the size of the test set on the performance of our algorithms. A marginal improvement in speedup was noticed for larger test sets. For example, when the test set consisted of 6000 random test vectors, the speedup was 4.33 with 6 processors for the circuit c7552, as opposed to a speedup of 4.31 when 2000 test vectors were used

5.1.2. Fault Set Partitioning

Table VII shows the results of the distributed fault simulation algorithm when random fault set partition scheme is employed. Table VIII shows the results when constructive fault set partitioning is employed. Our constructive fault set partitioning uses the signal probability method (see Section 4) for balancing the load among the slave tasks. From Table VII, we see that the random fault set partitioning technique shows a decrease in (dS_p/dp) when the number of processors is increased. For instance, the speedup reaches a value

TABLE VII Results for random fault set partition scheme. 2000 test vectors

No. of	c432		c3540		c7552	
processors	Run time(s)	Speedup	Run time(s)	Speedup	Run time(s)	Speedup
1	9	1.00	94	1.00	273	1.00
2	5	1.80	64	1.47	191	1.43
3	4	2.25	52	1.81	152	1.80
4	3	3.00	47	2.00	133	2.05
5	3	3.00	43	2.19	122	2.24
6	3	3.00	42	2.24	109	2.50

TABLE VIII Results for constructive fault set partition scheme. 2000 test vectors

No. of	c432		c3540		c7552	
processors	Run time(s)	Speedup	Run time(s)	Speedup	Run time(s)	Speedup
1	8	1.00	94	1.00	271	1.00
2	5	1.60	61	1.54	179	1.51
3	3	2.67	53	1.77	139	1.95
4	3	2.67	48	1.96	127	2.13
5	3	2.67	42	2.24	121	2.24
6	3	2.67	40	2.35	113	2.40

of 3.00 for $p=4$ and does not increase for larger values of p in case of c432. Similarly, in case of c7552, the speedup reaches the value of 2.05 for $p=4$, but shows a marginal increase of 2.50 when two more processors are added. This is due to insufficient load balancing among the processors, where a particular slave task works harder than others. We compare the variation of speedup with p for the constructive and random FSP schemes for the circuit c3540 in Figure 8. The plot marked with $*$ corresponds to constructive FSP whereas the plot marked with o corresponds to random FSP. It is evident from the plot that the constructive FSP scheme performs relatively better than the random FSP scheme in terms of the growth of speedup with p. This is because of reduced load imbalance in the constructive fault set partition scheme.

5.1.3. Fault Set vs. Test Set Partitioning

From our experimentation, it is apparent that TSP gives better speedups than the FSP scheme. The random TSP scheme shows a better value of (dS_p/dp) for increasing values of p than random FSP technique. For example, consider the case of circuit c7552. The speedups for random TSP and random FSP are plotted as functions of p in Figure 9. The ideal speedup curve is shown marked with x; the random TSP curve is marked with $*$, and the random FSP curve is marked with o. One of the reasons for the relatively better performance of the random TSP scheme is the dynamic load balancing which takes place in TSP due to fault dropping. This is because as soon as a slave task detects a fault, the information is conveyed to all the other subtasks and thus the

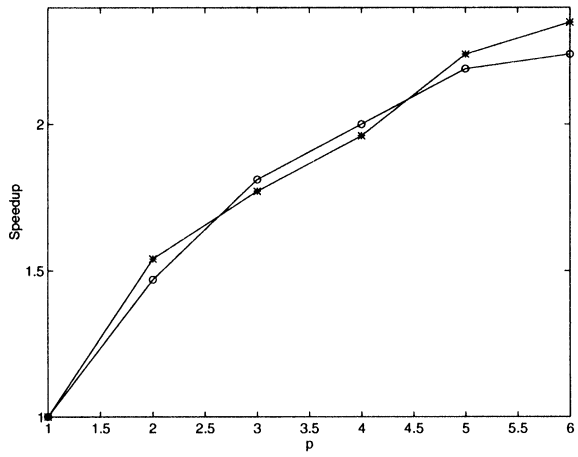

FIGURE 8 Speedup *vs.* p curves for constructive and random FSP schemes for circuit c3540.

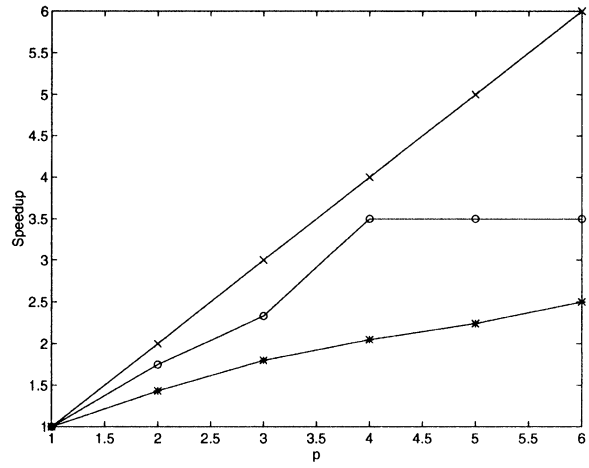

FIGURE 9 Speedup *vs.* p plots for random TSP and random FSP schemes for circuit c7552.

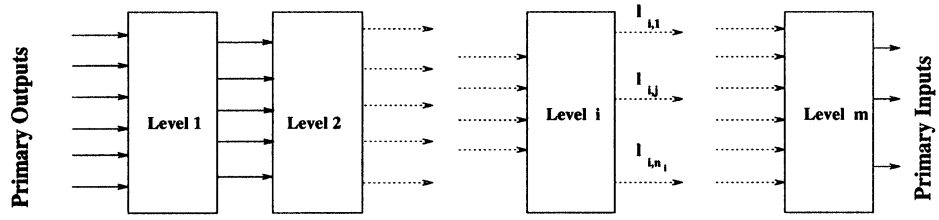

FIGURE 10 Model of a levelized combinational circuit.

load on each slave task decreases. Although both techniques showed sublinear speedups, the speed-up obtained using TSP is quite close to linear. Even RTSP proved to be better than CFSP.

6. CONCLUSIONS AND FUTURE WORK

In this paper, we have presented a novel algorithm for fault simulation of combinational circuits. We have described the implementation of our algorithm and presented experimental results on a number of benchmark circuits to establish the superiority of our algorithm over the event-driven deductive fault simulation algorithm. Our fault simulation technique based on path compression performs better than event driven deductive fault simulation, barring the case when the fault coverage is very low. The added advantage of our new technique is that it uses considerably less memory as compared to event driven deductive fault simulation (due to non-storage of large fault lists).

We have also presented distributed fault simulation algorithms based on test set and fault set partition. These algorithms were implemented on a network of workstations and were used on a number of fairly large combinational benchmark circuits. We found that the test set partition algorithms were superior in performance. We further found that the constructive test set partition procedure yielded better results than a random test set partition algorithm.

There are several extensions to the work presented here. Currently, the test partition parameter m in the test set partition algorithm is chosen to be 1. We believe that more experimentation is necessary in selecting an appropriate value of TPP, as pointed out in Section 4. We also believe that ordering the test vectors within each partition can improve the performance of the fault simulation algorithm. Ordering of test vectors is also important in reducing the power dissipation due to lower switching activity [5]. We have explained this in Section 5, but we have not reported experimental results relating to this issue. We also believe that more work is required in constructive test set partition based on circuit activity.

References

[1] Abromovici, M., Breuer, M. A. and Friedman, A. D. (1990). *Digital Systems Testing and Testable Design*, W. H. Freeman and Company.

[2] Balboni, G. P., Cabodi, G. P. and Gai, S. (1991). A parallel system for test pattern generation, In: *Proceedings of IEEE Symposium on Parallel and Distributed Processing*, pp. 708–715.

[3] Blank, T., A Survey of Hardware Accelerators Used in Computer Aided Design, *IEEE Design and Test*, pp. 21–39, August, 1984.

[4] Brglez, F. and Fujiwara, H., A neutral netlist of 10 combinational benchmark circuits and a target translator in Fortran, In: *ISGAS-85: IEEE International Symposium on Circuits and Systems*, **3**(3), June, 1985, Kyoto (Japan).

[5] Chou, R. M., Saluja, K. and Agrawal, V. D., Power constraint scheduling of tests, In: *Proceedings of 7th International Conference on VLSI Design*, Calcutta, pp. 271–274, January, 1994.

[6] Duba, P. A., Roy, R. K., Abraham, J. A. and Rogers, W. A. (1988). Fault Simulation in a Distributed Environment, In: *Proceedings of 25th Design Automation Conference*, pp. 686–691.

[7] Gai, S., Motessoro, P. L. and Somenzi, F. (1988). The Performance of Concurrent Fault Simulation Algorithms in MOZART, In: *Proceedings of 25th ACM/IEEE Design Automation Conference*, pp. 692–697.

[8] Geist, A. *et al.* (1994). *PVM: parallel virtual machine: a users' guide and tutorial for networked parallel computing*, MIT Press, Cambridge, Mass, ISBN 0262571080.

[9] Ghosh, S. (1995). A distributed algorithm for fault simulation of combinatorial and asynchronous sequential digital designs, utilizing circuit partitioning, on loosely-coupled parallel processors, *Microelectronic Reliability*, **35**(6), 947–967.

[10] Jain, S. K. and Agrawal, V. D., Statistical Fault Analysis, *IEEE Design and Test of Computers*, **2**(1), 38–44, February, 1985.

[11] Kumar, P. R. S., Jacob, J., Srinivas, M. K. and Agrawal, V. D., An Improved Deductive Fault Simulator, In: *Proceedings of 7th International Conference on VLSI Design*, Calcutta, pp. 307–310, January, 1994.

[12] Mueller-Thuns, R. B., Saab, D. G., Damiano, R. F. and Abraham, J. A., VLSI Logic and Fault Simulation on General-Purpose Parallel Computers, *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, **12**(3), 446–460, March, 1993.

[13] Patil, S. and Banerjee, P., Performance Trade-Offs in a Parallel Test Generation/Fault Simulation Environment, *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, **10**(12), 1542–1558, December, 1991.

[14] Ravikumar, C. P. (1991). *Parallel Algorithms for Automating Physical Design of VLSI Circuits*, *Ph.D. Thesis*, University of Southern California.

[15] Seth, S., Pan, L. and Agrawal, V. D. (1985). PREDICT – Probabilistic Estimation of Digital Circuit Testability, In: *Proceedings of Fault-Tolerant Computing Symposium*, pp. 220–225.
[16] Sienicki, J., Bushnell, M., Agrawal, P. and Agrawal, V. D. (1995). An Asynchronous Algorithm for Sequential Test Generation on a Network of Workstations, In: *Proceedings of International Conference on VLSI Design*, New Delhi.
[17] Tiwari, V. and Malik, S. (1993). Technology Mapping for Low Power, In: *Proceedings of the IEEE/ACM Design Automation Conference.*
[18] Xiang, D. and Wei, D.-Z. (1994). An Optimal Design for Parallel Test Generation Based on Circuit Partitioning, In: *7th International Conference on VLSI Design*, pp. 297–300.

A. PROOFS OF THEOREMS

A.1. Terminology

In this appendix, we prove the two theorems stated in Section 3. Figure A.1 shows a levelized combinational circuit. Let l_{ij} be the output line of gate j at level i. Note that l_{ij} can be written as a function of only lines of the form l_{i-1}, k, if we also permit the unary function 1 which leaves the input unchanged. After a good circuit simulation of the circuit for some test t, let v_{ij} be the value on line l_{ij}. Let v'_{ij} be the value on l_{ij} after a stuck-at fault f is introduced. We say that the fault f is visible at l_{ij} if $v_{ij} \oplus v'_{ij} = 1$. It is easy to see that a stuck-at fault at l_{ij} cannot affect the value at l_{pq} for $p < i$. A line l_{ij} is said to be pipe for a fault f and test t if $v_{ij} \oplus v'_{ij} = 1$ and $v_{ij'} \oplus v'_{ij'} = 0$ $\forall j' \neq j$. If l_{ij} is a pipe for some fault f and test t, we assign the attribute $\gamma(l_{ij}, f, t)$ to *visible* (*invisible*) if there exists some (no) j' such that $v_{m,j'} \oplus v'_{m,j'} = 1$ and m is the level number for primary outputs.

A.2. Proof of Theorem 1

We restate the theorem using the notation. For some test t and two different faults f and f', if a line l_{ij} is a pipe, then the attributes $\gamma(l_{ij}, f, t) = \gamma(l_{ij}, f', t)$. The proof is obvious if $i = m$, the level of primary outputs. Let us therefore consider $i < m$. In the absence of any fault, the gate outputs at level i can be written as a vector $(v_{i,1}, v_{i,2}, \ldots, v_{i,j}, \ldots, v_{i,n_i})$ where n_i is the number of outputs at level i. Since l_{ij} is a pipe for fault f, it is clear from the discussion of Section A.1 that the outputs at level i under fault f must be of the form $(v_{i,1}, v_{i,2}, \ldots, \overline{v_{i,j}}, \ldots, v_{i,n_i})$. By the statement of the theorem, the line l_{ij} is also a pipe for fault f for the same test t. Thus the output at level i will once again take the form $(v_{i,1}, v_{i,2}, \ldots, \overline{v_{i,j}}, \ldots, v_{i,n_i})$. Therefore, the values at the primary outputs are identical under the faults f and f'. Thus if f is visible at the primary outputs, so will be f', and *vice versa*. ■

A.3. Proof of Theorem 2

Let l_{ij} and l_{pq} be two different lines in the circuit, and let both of them be pipes for some fault f and test t. Suppose that $\gamma(l_{ij}, f, t) = visible$; this means there exists at least one primary output where the fault f is visible for test t. Since l_{pq} is a pipe, $\gamma(l_{pq}, f, t)$ must, by definition, also be *visible*. The proof for the second case, where $\gamma(l_{ij}, f, t) =$ *invisible* is identical. ■

Authors' Biographies

C. P. Ravikumar obtained a Ph.D. in Computer Engineering from the University of Southern California in 1991, an M.E. degree in Computer Science from the Indian Institute of Science in 1987 and a B.E. degree in Electronics from the Bangalore University, India, in 1983. He joined the faculty of the Department of Electrical Engineering, IIT Delhi, in 1991 as an Assistant Professor. Since 1995, he is an Associate Professor in the same department. Ravikumar serves on the editorial board of the International Journal of VLSI Design published by Gordon & Breach and the journal of Computers and Informatics published by the Computer Society of India. He is the author of the book Parallel Methods for VLSI Layout published by Ablex, New Jersey. He has served as a member of the program committee for several international conferences including the

International Parallel Processing Symposium, the International Conference on High Performance Computing, the International Conference on Parallel and Real Time Computing Systems, and the International Conference on Tools with Artificial Intelligence. His research interests are in the areas of VLSI Design Automation and Testing, Parallel Processing, and Combinatorial Optimization.

Vikas Jain obtained a B.Tech degree in Electrical Engineering from the Indian Institute of Technology, Delhi in 1995. He is currently pursuing an M.S. program in Computer Science at IIT Delhi. His research interests are in the areas of Distributed Computing, Computer Networks, and Multimedia.

Anurag Dod obtained a B.Tech degree in Electrical Engineering from the Indian Institute of Technology, Delhi in 1995. He worked with Tata Consultancy Services, Bombay during 1995–1996. Since 1996, he is with Duet Technologies, Noida, U.P., India. His research interests are in the areas of Distributed Computing, VLSI/CAD and testing.