Cellular Automata for Efficient Parallel Logic and Fault Simulation

Yih-Lang Li and Cheng-Wen Wu, *Member, IEEE*

***Abstract*—We present a unilateral 2-D cellular automata (CA) model and pipelining technique to parallelize logic and fault simulation. We show that given an acyclic digraph describing the Boolean function of a combinational circuit at the gate level, whose nodes are the logic gates of the circuit and whose directed edges stand for the propagating directions of signals, we can map this digraph onto a 2-D CA to simulate the signal propagation of the circuit on the CA. This mapping preserves not only the electrical connectivity of the circuit but also the massive parallelism inherited from the CA. Experimental results on ISCAS-85 benchmark circuits are obtained. Compared with previous fault simulation results, the time required for simulating one test pattern on an average is shorter by three to four orders of magnitude. As to pure logic simulation, our CA performs up to 9.24 billion gate evaluations per second using a 20 MHz clock and 8-b words. Scalability and extension to sequential circuits are discussed.**

I. Introduction

FAULT simulator is an important tool for logic and circuit designers. It helps the designer analyze the faults detected by a given set of test patterns and estimate the fault coverage during the process of automatic test pattern generation (ATPG). It usually includes two major operations: *fault injection* and *logic simulation*, where the latter consumes the majority of processing time.

As the complexity of VLSI circuits increases, their design costs soar, and the increased simulation cost leads to the urgent need of parallelism for logic simulation. The time complexity of sequential logic simulation is $O(n)$, where n is the number of gates in a circuit. Simulation of $O(n)$ faults sequentially, therefore, requires $O(n^2)$ time. The study of parallelism in logic simulation and fault simulation on general purpose multiprocessors has been reported by many researchers recently [1]–[4]. The experimental result of [1] shows that the theoretical maximum speedup can not be reached in a multiprocessor system under the conventional architectural constraint. In fact, the increase in the cost of interprocessor communication will follow the high concurrency. A statistical model to estimate the performance of logic simulation in a multiprocessor system was then presented in [2]. As predicted in [1], it seems that the speedup is dependent on some characteristics of the circuits, which lessens when the number of processors grows. Instead of circuit partition, partitioning the simulation process into several stages, i.e., pipelining, also succeeds in getting high concurrency [5]: a fixed-storage multipass concurrent fault simulation algorithm is used for the pipelined architecture of MARS, and their analysis shows that it may get better result for software as well as hardware implementation by using a multipass concurrent simulator.

Alternatively, massively parallel processing (MPP) has been used to speed up logic and fault simulation [6]–[10]. Using array processor to speed up logic and fault simulation has been under investigation [6], [7]. They implement a fault simulation method called Fault Information Tracing (FIT) algorithm on AAP2 which is an SIMD machine. Before simulation, logic gate placement and routing problem which determines the direction and order of signal flow among PEs must be solved. The performance estimation of logic simulation on AAP1 is about 770 times faster than software simulation for a circuit with 2105 gates. Fault simulation time on AAP2 is about 80 times faster than that on the RT-PC workstation for the ISCAS-85 circuit C7552 for one test pattern. In [10], three synchronous and asynchronous simulation techniques for event-driven logic simulation are implemented on the Connection Machine, which are shown to be several hundred times faster than a VHDL simulator on a SUN3 workstation. MPP is not suitable for event-driven logic simulation owing to the need of maintaining an event queue, especially for a l arge circuit [9]. In [9], a massively parallel compiled model simulation environment for multi-level (i.e., from behavior level to switch level) is proposed. The compiler handles circuit partitioning and scheduling. The performance of simulation relies heavily on the quality of the compiler; the compilation process can be time-consuming. For a 20 MHz clock, it was reported that a performance of 5 billion gate evaluations per second (GEPS) is achieved in a 256-processor architecture. A parallel algorithm using the methodologies of *parallel fault simulation* and *parallel pattern single fault propagation* (PPSFP) on the Connection Machine also was implemented, which achieves 2.7 billion GEPS [8]. In [11] and [12], logic simulation algorithms are vectorized and run on vector processors. A maximum performance of 7.7 billion GEPS is reported in [11]. In [13], several strategies are discussed with a goal to accelerate fault simulation. They use an important technique to make the algorithm practical, i.e., to parallelize fault simulation by utilizing the machine word-length. All of the above works boost the performance for fault simulation, but its complexity remains at $O(n^2)$, i.e.,

Manuscript received September 30, 1993; revised November 18, 1994. This work was supported in part by the National Science Council, R.O.C., under Contract NSC83-0404-E007-063. This paper was recommended by Associate Editor K.-T. Cheng.

Y.-L. Li is with the Department of Computer Science, National Tsing Hua University, Hsinchu, Taiwan.

C.-W. Wu is with the Department of Electrical Engineering, National Tsing Hua University, Hsinchu, Taiwan.

IEEE Log Number 9409643.

the improvement in performance is due largely to architecture and hardware innovation. In fact, there is little hope to do fault simulation in linear time so far as conventional computer architectures are assumed [14]. In [15], a content addressable memory (CAM) is used to present a "linear" time algorithm for fault simulation. However, the linear time complexity is achieved by duplicating n hardware resources, where n is the number of faults. Also, CAM access time is assumed to be $O(1)$. There are two important issues associated with the performance of multiprocessor system based on circuit partition, i.e., intercommunication and circuit parallelism. Efficient partition algorithm can reduce the intercommunication cost, however, circuit parallelism is unpredictable. Some research result shows that the relationship between parallelism and circuit size is much more complex than linearity, besides, the circuits in the same family has this same phenomenon [16]. Therefore, using the measured parallelism of small circuits to extrapolate parallelism of larger circuits is inappropriate, i.e., a larger circuit may have a lower parallelism.

In this paper, we present a unilateral 2-D *cellular automata* (CA) model and pipelining technique to parallelize logic and fault simulation, and show by analysis and simulation that a larger circuit has a higher simulation rate in our model. Instead of using array processors [6], [7], we present a systematic approach to map a circuit onto a 2-D CA architecture, which makes pipelining technique available to logic and fault simulation. This leads to more gate evaluations executed in parallel. Furthermore, the flow direction of any signal in CA is controllable and independent at any time, as opposed to the fixed and identical direction of all signals in AAP1 and AAP2 [6], [7].

CA consists of identical cells which are connected in a mesh structure, and each cell can only receive data from its direct *neighbors*. Each cell is in one of several states at each time instant (clock period, if the cells are synchronized), and its next state depends on its current state and those of its neighbors. The cell changes its states according to a set of predefined state transition *rules* which simulate the dynamic electrical signal flow through the components of the circuit. CA cells are simple and small, and their implementations exhibit temporal and spatial locality, homogeneity, and topological regularity, which make them ideal for VLSI implementation. Hundreds or even thousands of cells can be packed into one single chip based on today's VLSI technology, and the packing density is growing very fast. CA were originally introduced by von Neumann as a possible idealization of biological systems, with the particular purpose of modeling biological self-reproduction [17]. In statistical mechanics, CA have been used as simple mathematical models to investigate self-organization [17]. They also have been reintroduced for a wide variety of purposes, and referred to by a variety of names, including *homogeneous structures, cellular structures, cellular arrays*, and *iterative arrays*. It is possible for CA to be considered as parallel processing computers. Particularly in two dimensions, CA have been used extensively for image processing [17].

In our CA model, cells are connected to form a 2-D array and each cell has five neighbors: left, right, top, bottom, and itself. There are eight cell states in our CA, and the machine word-length is 8 b. We show that given an acyclic digraph describing the Boolean function of a circuit at the gate level, whose nodes are the logic gates of the circuit and whose directed edges stand for the propagating directions of signals, we can map this digraph onto a 2-D CA so advisably as to be able to simulate the propagation of signals throughout the circuit on the CA. This mapping preserves not only the electrical connectivity of the circuit to produce correct outputs but also the massive parallelism inherited from the CA. Experimental results on ISCAS-85 benchmark circuits are obtained. Compared with that given in [8], the time required for simulating one test pattern (on an average) is shorter by three to four orders of magnitude. As to pure logic simulation, when compared with [9], our CA performs up to 9.24 billion GEPS using a 20 MHz clock and 8-b words as opposed to 5 billion GEPS. Scalability and extension to sequential circuits are discussed.

II. CA Model

A. CA Definition

A *cellular automaton* can be characterized by the interconnection among its cells and the state transition rules dapplied to each cell. The interconnection defines the *neighborhood* of a cell. Cells are concerned only about the local information, i.e., the information (state) from their neighboring cells. At any time, each cell has a *current state*, and its next state is determined by a state transition rule which describes the next state as a function of current local information.

A formal definition of CA is given in [18]. Let Z be the set of integers. A *k-D cellular automaton* is an infinite array, indexed by Z^k, of cells. Each cell is identified by its *location* $I \in Z^k$. A cellular automaton is a quadruple $A = (k, S, N, f)$, where $k \geq 1$ is the dimension, S is the finite set of states, N is the *neighborhood*, and f is the *local function* of A. The (relative) neighborhood N is a sequence $(I_1, I_2, \ldots, I_h)$ of (relative) locations, where $I_j \in Z^k$, $1 \leq j \leq h$. The local function is a total function:

$$f\colon S^h \longmapsto S, \tag{1}$$

where S^h is the finite set of states of one cell's neighborhood. The local function f describes the state transition rules, i.e., each cell follows the local function f to alter its state at next time instance according to the states of its neighborhood. A *configuration* c of CA is a function:

$$c\colon Z^k \longmapsto S, \tag{2}$$

which assigns a state in S to each cell of the CA. The set of configurations is denoted as S^{Z^k}. The local function f is extended to the *global function* of the set of configurations into itself:

$$G_f\colon S^{Z^k} \longmapsto S^{Z^k}. \tag{3}$$

By definition, for all $c_1, c_2 \in S^{Z^k}$,

$$G_f(c_1) = c_2 \Longleftrightarrow \forall\, I \in Z^k,$$
$$c_2(I) = f(c_1(I + I_1), c_1(I + I_2), \ldots, c_1(I + I_h)). \tag{4}$$

```
{
  E={all the edges}; T = N ={the primary input nodes};
  for (i ∈ N) l(i)=0;
  while (T is not empty)
    N = T, T = φ,
    for (i ∈ N)
      if ((i,j)∈ E)
        l(j) = l(i) + 1, T = T∪{j};
}
```

Fig. 1. Algorithm for leveling a digraph.

The global function opens a global view on the effect of the local function (i.e., the state transition rule) when applied on the overall cells.

B. Our CA Model

In our CA model, cells are connected to form a 2-D array and each cell has five neighbors: left, right, top, bottom, and itself. There are eight cell states. We will discuss these eight states and the state transition rule in detail in Section 4. Formally, our CA can be written as a quadruple $A = (k = 2, |S| = 8, N = ((-1,0),(1,0),(0,1),(0,-1),(0,0)), f)$.

In some CA models, a group of cells observed globally may disclose meaningful information, so it is necessary to define the global function in order to observe the CA's dynamic behavior. Our goal however is to simulate circuits using CA, which dose not require the observation of the CA's dynamic behavior. Therefore, we do not define the global function.

C. Network

A *network* [19]. N is a quintuple $N = (V, E, s, t, c)$, where $(V. E)$ is a digraph with two distinguished vertices, s and t, called the *source* and the *terminal*, respectively. The source node has only outward edges, and the terminal node has only inward edges. Each edge e of N is assigned a nonnegative real number $c(e)$ which is the *capacity* of the edge, i.e., c is a function which assigns nonnegative real numbers to the edges of N.

In our approach, we will transform the combinational network under simulation into a *layered network* (LN), which is a 3-tuple $L = (V, E, l)$, where (V, E) is a digraph, and each node v of L is assigned a nonnegative integer number $l(v)$ called the *level* of the node. If $(u, v) \in E$, then $l(v) = l(u)+1$. The primary differences between a network and a layered network are that the layered network does not have the source and terminal nodes and that there is an edge (u, v) only if $l(v) = l(u) + 1$.

III. Graph Embedding

A. Problem Definition

In order to simulate the propagation of electrical signals throughout the circuit by using CA, we must have the ability to describe the internal structure and dynamic behavior of the circuit, including the interconnection and the direction of signal flow among the components of the circuits, from the CA's architectural point of view explicitly and systematically.

We now state the problem of *graph embedding*: Given an acyclic digraph describing the Boolean function of a circuit at the gate level, whose nodes are the logic gates of the circuit and whose directed edges stand for the propagating directions of signals, map this digraph onto a two-dimensional CA so advisably as to be able to simulate the propagation of signals throughout the circuit on the CA.

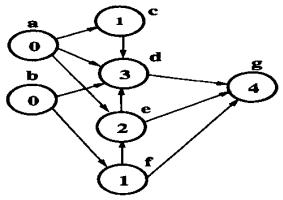


Fig. 2. A leveled digraph.

Altogether, this mapping must preserve not only the electrical connectivity to produce correct outputs but also the ability of massively parallel processing inherited from the CA.

From CA's architectural point of view, LN is an ideal candidate for mapping the circuit. In an LN, the signal of a node at level t can be propagated only to those nodes whose levels are $t + 1$. Accordingly, we place in the same column all nodes with the same level. In fact, a pair of columns of CA cells are used to represent the nodes of each level, where the left column corresponds to the fan-in edges and the other to the fan-out edges. The nodes with zero level are placed on the leftmost column and the level-one nodes occupy its neighboring column, and so on.

B. The Proposed Approach

We complete the graph embedding by three steps: 1) leveling the acyclic digraph; 2) modifying the leveled digraph into an LN; and 3) mapping the LN onto the CA after ordering the sequence of nodes with the same level.

A node is said to be *ready* if it has no inward edge or all its inward edges are *marked*. Initially, all the edges are *unmarked* and the level number l_{no} equals zero. Each primary input signal of the circuit is represented by one node. The primary input nodes obviously are *ready* nodes. During each iteration, the levels of the ready nodes are set to l_{no} and all their outward edges are marked. We then remove the ready nodes from the set of nodes, and the level number l_{no} is incremented by one. The iterative process continues until there is no more ready nodes. Fig. 1 shows an efficient algorithm for leveling an acyclic digraph. Fig. 2 shows a leveled digraph.

After leveling the digraph, there may exist edges (i, j) such that $l(j) > l(i) + 1$. These edges are illegal on the CA architecture due to its locality property. For such an edge, we add $l(j) - l(i) - 1$ new nodes between node i and node j.

Consider the following two cases for node j, where $j \in V$. We define the set of N_j to be $\{i \mid (i,j) \in E$ and $l(j)-l(i) > 1\}$.

Case 1: $|N_j| > 1$. In this case, we delete the edge (n, j), where $l(j) - l(n) = \max_{i \in N_j}(l(j) - l(i))$, and create, between node n and node j, $(l(j) - l(n) - 1)$ new nodes with labels from $l(n) + 1$ to $l(j) - 1$ and one path passing through every new nodes from node n to node j. The other edges (i, j), $i \in N_j$, must also be deleted. Each node i in N_j except node n has a new outward edge $(i, h_{l(i)+1})$,

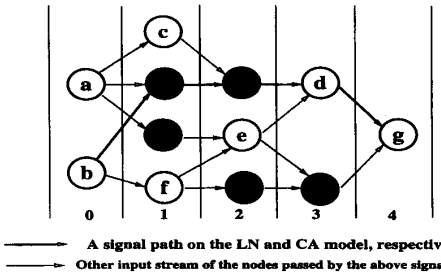


Fig. 3. The LN obtained from the previous figure, where a shaded node stands for a newly created node.

where $h_{l(i)+1}$ is the newly created node on the path from n to j whose level is $l(i)+1$. Node d in Fig. 3 is an example.

Case 2: $|N_j| = 1$. Assuming that $N_j = \{i\}$, we only need to delete edge (i,j) and create, between node i and node j, $(l(j) - l(i) - 1)$ nodes and a path passing through every new nodes from node i to node j.

The algorithm for converting a leveled digraph into an LN is shown in Fig. 4.

The mapping from an LN onto our CA architecture is straightforward. One cell is required for every edge of the LN. The outward edges of the nodes with level zero (i.e., the primary input nodes) are placed on the cells of the leftmost column; the second and third columns are for the nodes with level one; and so on. Fig. 5 shows a mapping example.

The ordering of nodes at the same level will affect the system performance. The problem of node ordering is similar to the module placement problem in VLSI layout, which is well known to be NP-complete. We use an efficient heuristic algorithm to determine the order of nodes, which reduces the start-up time by 14% to 77% when compared with random ordering. We first determine the order of primary output nodes randomly, because in most circuits the number of primary output nodes is less than that of the primary input nodes, and random ordering is cost-effective. Based on this known order and the interconnection between the primary output level and its previous level, we can easily calculate the average order of each node whose level is one less than that of the primary output nodes. The order of other nodes with level t can be obtained in the same way after the completion of the nodes with level $t+1$.

IV. Logic and Fault Simulation

We divide logic simulation into three steps: 1) input accumulation; 2) output distribution; and 3) output propagation to the next level. The input accumulation and output propagation are accomplished by the fanin cells of a node, and the output distribution by the fanout cells. We illustrate these three steps by tracing a signal path from node b to node g in Fig. 3, which is highlighted. Its corresponding path on the CA is shown in Fig. 5.

The first segment, i.e., the path from upper b^{out} to lower d_1^{in}, stands for the process of output signal propagation to the next level. During this step, upward and downward moves are possible and a node may be passed through by signals moving

```
{
  S=all the nodes; E=all the edges;
  Compute N_j for all j ∈ S, where N_j = {i | (i,j) ∈ E and l(j) − l(i) > 1};
  for (each node j in S)
    if (| N_j | > 1) {
      Create max_{i∈N_j}(l(j) − l(i) − 1) new nodes with labels from l(i) + 1 to l(j) − 1;
      Delete the edges (i,j), where i ∈ N_j;
      Create new edges (m,n) among the new nodes and (N_j ∪ {j}) if l(n) = l(m) + 1;
    }
    else if (| N_j |= 1) { /* Assume that N_j={i} */
      Delete the edge (i,j);
      Create (l(j) − l(i) − 1) new nodes with labels from l(i) + 1 to l(j) − 1;
      Create new edges (m,n) among the new nodes and {i,j} if l(n) = l(m) + 1;
    }
}
```

Fig. 4. Algorithm for converting a leveled digraph into an LN.

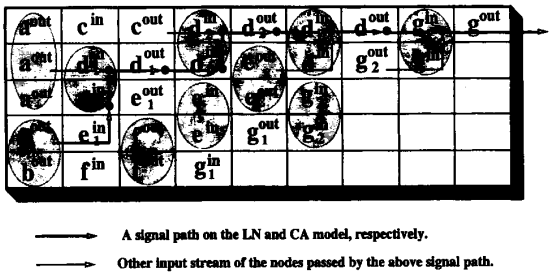


Fig. 5. A mapping from an LN onto our CA architecture, where the cells covered by a shaded ellipse are mapped to the inward or outward edges of the same node.

in different directions. The input accumulation of the next level follows the output propagation. A cell begins to evaluate its output according to its gate type when it receives its input signal from the previous level. The arrow from the lower d_1^{in} to the upper d_1^{in} and the shaded arrow from a^{out} to the upper d_1^{in} show the process of input accumulation. The next step after input accumulation is to distribute the output signal to the fanout cells. This operation is depicted by the arrow from d_1^{in} to d_1^{out}. The remaining segments of this signal path repeat these three steps until CA produces the primary output, g^{out}.

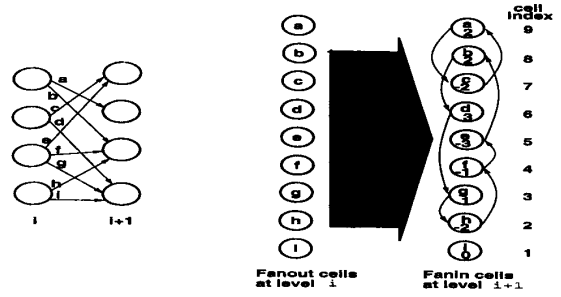
As mentioned before, our CA has eight states. Among them, six are designed to implement logic simulation: `Fanin`, `BotFanin`, `Fanout`, `FanoutRecv`, `NewPipe`, and `NewPipeRecv`. The first two states are available for the fanin cells and the other for the fanout cells. The other two states, `Detecting` and `Detected`, are designed for implementing fault simulation.

A. Input Accumulation and Output Propagation

Because a gate with n fanin edges is distributed among n cells (indexed by 1 to n from bottom to top), the logic operation of each cell is different from that of the original node. We use the **NAND** operation for illustration:

$$f^i_{\text{nand}}(x_1,\ldots,x_n) = \begin{cases} x_i & \\ i = 1 & \\ f_{\text{and}}(x_i, f_{\text{and}}(x_1, x_2, \ldots, x_{i-1})) & \\ 1 < i < n & \\ f_{\text{nand}}(x_i, f_{\text{and}}(x_1, x_2, \ldots, x_{i-1})) & \\ i = n. & \end{cases}$$

For example, a 3-input **NAND** gate has three fanin cells. The bottom cell, say Cell 1, does nothing but to keep its signal ready for Cell 2. Cell 2 then performs an **AND** operation on



(a) (b)

Fig. 6. An example of output signal propagation.

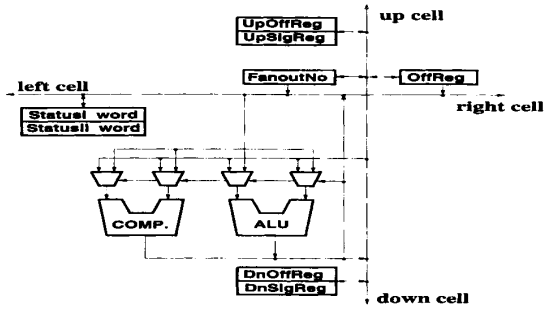


Fig. 7. The data path of one cell.

the signals from itself and Cell 1. Finally, Cell 3 performs a **NAND** operation on the signal that it receives and the one that comes from the result of the **AND** operation performed by Cell 2. A 3-input **NAND** operation is thus distributed among these three fanin cells. The fanin cells can be in `Fanin` or `BotFanin` state: the bottom cell of a node is in `BotFanin` state and the other in `Fanin` state. The difference between `Fanin` and `BotFanin` is that a cell in state `BotFanin` needs not do logic operation during the input accumulation process. The fanin cells keep their next state unchanged regardless of the state of their neighbors.

When a cell is in state `Fanin` or `BotFanin`, it continuously checks if the current state of its left cell is in state `NewPipe` or `NewPipeRecv` which indicates that a signal generated by a new input vector has arrived. If a new signal has arrived, the fanin cell receives it and then begins to propagate this signal to its target cell according to its offset which is loaded into a register called OffReg during the system initialization. The offset of a signal is defined as the difference of the index of the cell which receives it and that of its target cell. Each cell is indexed by its position that is counted upward, where the position of the bottom cell of each column is 1. Fig. 6 shows the offset of each signal. In Fig. 6(b), the bold arrow from Cell 9 to Cell 7 indicates that the destination cell of signal a which Cell 9 receives from left is Cell 7, therefore, the offset of signal a is 2. We introduce four registers inside each cell to solve the propagation problem, namely UpSigReg, UpOffReg, DnSigReg, and DnOffReg. These registers form two data paths across the cells at fanin columns, one for the signals moving upward and the other downward. Hence, the signals going in opposite directions can move simultaneously. When the topmost fanin cell of a gate finishes output signal propagation and input signal accumulation, it sets a one-bit register to 1 to inform its right neighbor of the completion of gate evaluation. Note that when a fanin cell receives a new signal, this signal can not be fed into the data path unless the data path is free.

B. Output Distribution

All fanout cells except the top cell of each node are in state `Fanout` during output distribution. The top cell of each node at fanout columns is in `FanoutRecv`, whose duties are not only to distribute the signal to its target cells but also to incessantly check whether the gate evaluation is completed. Similar to preloading for output propagation at fanin columns, the cells in state `FanoutRecv` are preloaded into the OffReg register with an offset which is the result of subtracting the largest index of the fanout cells from that of the current cell. In addition to the OffReg register, another register called FanoutNo is used to store the number of outward edges of a node. The top cells of the nodes at fanin columns keep this value, which is a preloaded value, too. Whenever the fanout cells in state `FanoutRecv` begin to distribute the output signal, they receive the FanoutNo of the signal from their left cells. A cell is the *target cell* of a signal if either the offset of the downward data path is one and the corresponding FanoutNo (which is decremented by one whenever the signal passes one target cell) is still larger than zero, or the offset of the upward data path is negative and the sum of the offset and FanoutNo is no less than zero. Note that during output distribution the signals passing through a node are in the same direction, although moves in opposite directions may coexist at fanout columns. When a signal is passed to its target cell, this cell changes its state to `NewPipe` or `NewPipeRecv` to announce the arrival of a new signal. A cell in state `NewPipe` (`NewPipeRecv`) changes to `Fanout` (`FanoutRecv`) in its next state. We show the data path of one cell in Fig. 7.

C. Fault Simulation

We modify the parallel pattern fault simulation method for our CA. The words of CA is divided into two parts of the same size. One is for the good circuit, and the other for the faulty one. After a new fault is injected into CA, $W/2$ patterns are simulated in parallel for both the good and the faulty circuits, where W is the word-length of CA. The primary output cells compare the results of good and faulty circuits to determine whether to continue logic simulation or to inject a new fault. Fault injection is completed by marking the *faulty* register of the corresponding cell for the faulty line and loading the faulty value into the *fault_signal* register. Only the fanin cells can be faulty because we drop the faults located at fanout edges during fault collapsing. Both the *faulty* register and the *fault_signal* register are one bit registers. The primary output cells can be in either the `Detecting` state or the `Detected` state. When a new fault is injected into CA, these cells are in state `Detecting`, and continuously receive and compare the outputs. They change their state into `Detected` when the fault is detected.

Although the above approach repeats good circuit simulation for each fault, it minimizes the communication overhead between the host computer and CA during the simulation process. There is an alternative approach when the communication between the host and CA is fast, i.e., we can complete the good circuit simulation first and store the results on the host computer, and then simulate W patterns in parallel for the faulty circuits. The host computer is responsible for fetching the results from CA and comparing them with the previously stored results to determine whether this fault has been detected. Whether other popular fault simulation techniques (such as starting simulation at the fault site and stopping as fault effect disappears) can be incorporated into our CA is the future subject of this research. It can be difficult if we want to maintain a simple cell structure.

V. Performance Analysis

We first define the following variables:

- $d_{f,l}$ The difference of arrival times between the first arriving signal and the last.
- T The set of cells in state `FanoutRecv`.
- B The set of cells in state `BotFanin`.
- $P_{\text{out}}(i)$ The number of cells in T with a positive (negative) offset above (below) cell i if the OffReg register of cell i is positive (negative).
- $P_{\text{in}}(i)$ The number of cells with a positive (negative) offset above (below) cell i if the OffReg register of cell i is positive (negative).
- $N_{\text{in}}(i)$ The fanin of the node containing cell i.

Let us consider the fanout columns. A fanout column performs output signal distribution, whose time complexity is $O(n)$ for a single signal, where n is the number of cells at this column. All signals can be distributed simultaneously except for the case when the downward data path is occupied by another downward signal coming from above, or similarly for an upward signal. The upper bound of distribution time for a fanout column therefore is

$$t_{\text{fanout}} \le d_{f,l} + \max_{i \in T}\{\text{OffReg}(i) + P_{\text{out}}(i)\}, \tag{5}$$

where $0 \le \text{OffReg}(i) \le n$ and $0 \le P_{\text{out}} \le |T|$. The best case occurs when all signals arrive at the same time, i.e.,

$$t_{\text{fanout}} = \max_{i \in T}\{\text{OffReg}(i)\}. \tag{6}$$

The complexity analysis for the fanin column is similar to that of the fanout columns. The upper bound of signal distribution time for a fanin column is

$$t_{\text{fanin}} \le d_{f,l} + \max_{1 \le i \le n}\{\text{OffReg}(i) + P_{\text{in}}(i)\} + \max_{i \in B}\{N_{\text{in}}(i)\}, \tag{7}$$

where $0 \le \text{OffReg}(i), P_{\text{in}}(i) \le \lceil n/2 \rceil - 1$. The sum of OffReg and P_{in} is less than n. The best case is when, in addition to the same arrival time for all signals, all other gate evaluations have been completed before the signal with the maximum offset stablizes and its target cell represents either an inverter or a buffer, i.e.,

$$t_{\text{fanin}} = \max_{1 \le i \le n}\{\text{OffReg}(i)\} + 1. \tag{8}$$

The *propagation delay* of a CA column, t_d, is constrained to

$$t_d + f(i) \ge c(i+1), \quad 0 \le i \le N, \tag{9}$$

where $f(i)$ is the first signal arrival time at column i, $c(i)$ is the operation completion time at column i, and N is the number of columns. This constraint avoids two successive input patterns to appear in a column at the same time. We therefore obtain

$$t_d = \max_{0 \le i < N}\{c(i+1) - f(i)\}. \tag{10}$$

Except fault injection, the complexity of fault simulation is the same as that of logic simulation. The upper bound of fault injection time is $O(n)$, where n is the number of columns in CA.

We now describe two ways to improve the performance of CA. By our observation, $d_{f,l}$ is a dominant factor in t_d due to the fact that the existence of buffers and inverters in a circuit often enlarges $d_{f,l}$. Usually all the columns receive their first signal in a short period of time, which increases t_d. We can eliminate the term $d_{f,l}$ from (5) and (7) by adding a global information for each column, which records the total number of cells having completed their operations. The state transition rules are also modified to enable the cells in a same column to receive signals from their left column simultaneously. When a fanin cell finishes the operations of output signal propagation and input signal accumulation, it increases the content of the global variable for this column by one. If it is the topmost cell of a gate, it begins to check whether all the cells of this column have completed their operations periodically. It will set a one bit register inside itself to one to inform its right neighboring cell of the completion of gate evaluation. For fanout columns, when a signal reaches its target cell, this cell will increment the content of the global memory belonging to this column, and then periodically check if all the cells at this column have received their own signal. It will change its state to `NewPipe` or `NewPipeRecv` to announce the arrival of a new signal if all the cells at this column have received their signal. Hence, (5) is reduced to

$$t_{\text{fanout}} \le \max_{i \in T}\{\text{OffReg}(i) + P_{\text{out}}(i)\}, \tag{11}$$

and (7) is also reduced to

$$t_{\text{fanin}} \le \max_{1 \le i \le n}\{\text{OffReg}(i) + P_{\text{in}}(i)\} + \max_{i \in B}\{N_{\text{in}}(i)\}. \tag{12}$$

Therefore,

$$t_d = \max_{0 \le i \le N}\{c(i) - f(i)\} + 3. \tag{13}$$

We can see that the performance of CA depends on the completion time of each column from (13). $\text{OffReg}(i)$ is a common term between (11) and (12), which appears to be the dominant factor affecting the completion time of each column as will be shown later by the experimental results.As stated before, a different gate ordering may result in a different maximum offset

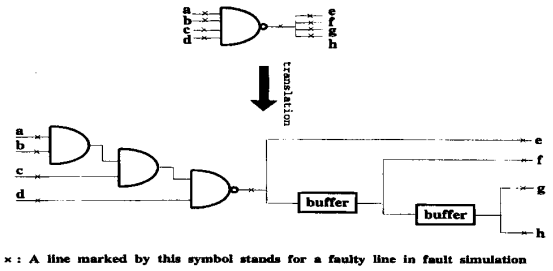


Fig. 8. An example of gate translation.

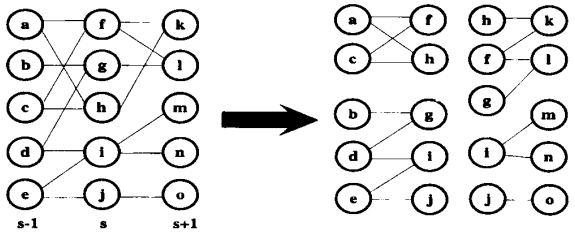


Fig. 9. Union of two adjacent connection graphs.

at the fanin column, however, this effect does not occur at the fanout column. Besides, the maximum offset among all fanin columns is much larger than that for fanout columns. The time for logic simulation of this parallel logic simulation scheme is then $O(n)$, where n is the maximum number of cells among all fanin columns. For fault simulation, $O(n^2)$ is required. The hardware complexity (the number of CA cells) for logic and fault simulation is $O(n)$, where n is the number of signal lines, and each cell contains eight 8-b registers. It may seem that no improvement is reached by using our parallel architecture according to the asymptotic bounds, however, constants are reduced drastically, and pipelining can be performed. Also, the maximum number of cells among all fanin columns is relatively small as compared with the gate count of a circuit, and there is no evidence that the maximum number of cells among all fanin columns grows linearly with the total gate count. The other advantage is that we can make the maximum offset grow slightly or even shrink when the maximum number of cells and the gate count increase by applying gate translation and a new gate ordering algorithm (to be discussed next). This indicates that the rate of increase in number of active gate evaluations may be much lower than linearity.

We now propose another method to reduce the maximum offset, which, however, doubles the hardware cost (in number of CA cells). The fanin and fanout of each gate are limited to two. We first translate a circuit into this format before graph embedding. Such a translation can be done as shown in Fig. 8. The resulting circuit has an important advantage: the fanin and fanout will not grow as the circuit size increases, so the maximum offset will increase only slightly even for a large circuit if we provide a good gate ordering algorithm. However, this will increase the fault injection time because the new circuit is deeper.

Consider the interconnection between two adjacent levels. We define G_c^i as the *connection graph* of level i whose vertices are the gates of level i and level $i+1$, and edges are the interconnection wires between these two levels. We can see that each component of G_c^i is either a path or a cycle since every node in the component has a degree of no more than 2. Hence, the graph can be represented as

$$G_c^i = \bigcup_{1 \le t \le n_c} C_i^t, \tag{14}$$

where n_c is the number of components in G_c^i, and C_i^t is a component in G_c^i. If we define $U_{i,i+1}^t$ as

$$U_{i,i+1}^t = \{v \mid l(v) = i+1, v \in V(C_i^t)\}, \tag{15}$$

then we have

$$V(G_c^i) = \left\{ \bigcup_{1 \le t \le n_c} U_{i,i}^t \right\} \cup \left\{ \bigcup_{1 \le t \le n_c} U_{i,i+1}^t \right\}. \tag{16}$$

Now consider the union of the connection graphs of adjacent levels, such as $G_c^{s-1} \cup G_c^s$ which is shown in Fig. 9(a). To determine the order of gates for level s, we have to take G_c^{s-1} and G_c^s into account. Fig. 9(b) shows the difference between the ordering by G_c^{s-1} and that by G_c^s.

We propose a bidirectional scanning algorithm for ordering the combinational network. A queue is assigned to each level, and in the beginning a gate is selected randomly from level 0 and inserted into the corresponding queue. We say a gate is *free* if it has not been inserted into the queue. Our algorithm scans the network from left to right (the first pass), and then from right to left (the second pass). When level i is scanned forward (left to right), we find the first gate u from the corresponding queue such that there exists an edge $(u, v) \in E(G_c^i)$, and that gate v is *free*. We then insert gate v into the queue of level $i+1$. When level i is scanned backward (right to left), we repeat the process, except that G_c^i is now replaced by G_c^{i-1}. This bidirectional scan process is repeated until there is no *free* gates. This algorithm requires at least $\lceil \frac{n-1}{N-1} \rceil$, and at most $2 * \lceil \frac{n}{N} \rceil - 1$ passes, where n is the total number of gates and N is the number of levels.

VI. Experimental Results

We have implemented a CA simulator on a SUN Sparc2 workstation using the C language. The preprocess, including leveling the digraph, translating the digraph into an LN, and calculating the necessary information needed by CA, takes less than 0.27 s for each of the ISCAS-85 benchmark circuits. In Table I we list the CPU time (s) required for the preprocess and the clock cycles for CA to complete the simulation of a single input pattern. It shows that our heuristic ordering algorithm boosts the performance of CA significantly.

We compare the experimental results of two CA versions in Table II. We use CA_∞ and CA_2 to represent the CA's without and with the process of gate translation, i.e., for gates with unlimited fanin and with fanin of two, respectively. We list the maximum number of cells in a column ($\text{Cell}_{\text{col}}^{\max}$), the total number of cells ($\text{Cell}_{\text{total}}$), t_d (Latency), CPU time (s) for ordering (T_{order}), and sequential run time of the logic simulation in Table II. In the table, it is clear that the disadvantage of gate translation on the circuit is to roughly

TABLE I
CPU TIMES FOR PREPROCESSING WITHOUT ORDERING AND WITH ORDERING

Benchmark Circuits	Runtime for Preproc. without Ordering (sec)	Runtime for Ordering (sec)	Clock Cycles for CA	
			Without Ordering	With Ordering
c17	0.0000	0.0000	24	19
c432	0.0000	0.0667	303	352
c499	0.0000	0.0833	614	378
c880	0.0000	0.1167	607	536
c1355	0.0000	0.1167	776	655
c1908	0.0333	0.4167	1634	955
c2670	0.0500	0.5333	2599	1066
c3540	0.0333	0.5000	3416	1669
c5315	0.1333	1.1000	8405	2959
c6288	0.2667	1.8500	4951	2254
c7552	0.1500	1.0333	8046	3007

TABLE II
EXPERIMENTAL RESULTS ON CA

Benchmark Circuits	$Cell_{col}^{max}$		$Cell_{total}$		Latency		T_{order}		Sequential Runtime
	CA_∞	CA_2	CA_∞	CA_2	CA_∞	CA_2	CA_∞	CA_2	
c17	6	6	32	32	8	6	0	0	0.00
c432	90	90	1691	4877	141	22	0.067	0	0.01
c499	128	91	1872	3726	207	29	0.083	0	0.01
c880	222	160	4426	8844	252	30	0.117	0	0.01
c1355	208	175	5904	8470	201	38	0.117	0	0.03
c1908	233	207	11622	16085	247	53	0.417	0	0.05
c2670	475	396	15613	32511	592	49	0.533	0.067	0.10
c3540	538	435	14591	27487	628	69	0.5.1	0.05	0.17
c5315	984	822	32321	66439	1674	103	1.1	0.283	0.48
c6288	541	517	58848	84898	451	49	1.85	0.267	0.34
c7552	1044	841	33616	62106	1663	70	1.003	0.283	0.65

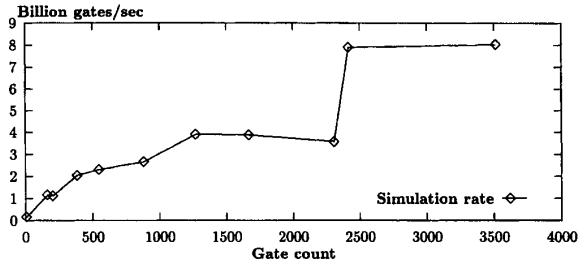


Fig. 10. CA has a higher simulation rate for larger circuits.

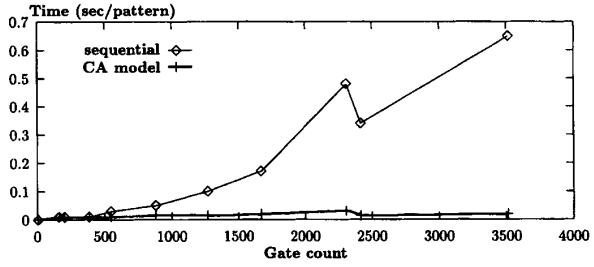


Fig. 11. Performance comparison between CA and sequential algorithm.

double the number of cells required, which may also increase t_d. The reason why the bidirectional scan algorithm takes less CPU time than that of the original ordering algorithm is that the original algorithm needs m more additions and one more division to determine the position of a gate (where m is the fanin of this gate), although it will complete the ordering operation in only one scan pass. The CA with gate translation is good for large circuits, since the gate interconnection complexity does not grow with the circuit size. The increase in t_d therefore is slow as the circuit size increases. This results in a higher performance measured by billion GEPS for larger circuits. We delineate this in Fig. 10. In Fig. 11, we compare the result with that obtained by running a sequential algorithm on a SUN Sparc2. For example, the completion time of c432 is 0.01 s under the assumption that the CA operates at 35.2 KHz. Let y_s and y_{ca} be the curves of computing time (per pattern) versus gate count for sequential algorithm and CA, respectively. From Fig. 11, we obtain the following approximated relationship,

$$y_{\text{ca}} = 0.0246 * y_s^{0.2070}, \tag{17}$$

which shows that using CA with pipelining is much more efficient than using a sequential processor.

In [9], the compiler needs to handle the circuit partitioning and scheduling. The min-cut algorithm which they use to partition the circuits spends much more time than our preprocessing algorithm as the circuit size is large. In [11], logic simulation techniques are implemented on a vector processor FACOM VP-200 to reach $7.7 * 10^9$ GEPS. We compare our results with these two in Table III. Notice that our simulation rate does not count those extra gates which are added during gate translation and transformation of a circuit into LN.

TABLE III
COMPARISON OF PEAK LOGIC SIMULATION RATES

Simulator	Peak Sim. Rate billion GEPS	Clock Cycle MHz	CPU Peak Speed MFLOPS	Pipeline Cycles ns
CA	9.24	20	-	-
Processor-array	5	20	-	-
FACOM VP-200	7.7	-	533	7.5

TABLE IV
FAULT SIMULATION RESULT FOR ISCAS BENCHMARK CIRCUITS ON CA

Benchmark	Fault Injection Time clock cycles/fault	Fault Sim. Rate billion GEPS
c17	4.5	0.28
c432	31.2	0.74
c499	18.4	0.65
c880	25.5	1.23
c1355	33.1	1.71
c1908	36.5	3.91
c2670	38.2	4.96
c3540	49.5	4.65
c5315	34.8	3.24
c6288	126.7	0.43
c7552	45.8	7.03

The fault simulation result is shown in Table IV under the assumption that the CA operates on a 20 MHz clock. In [15], the simulator takes 3.13 s for simulating 512 patterns on c6288, assuming that one instruction can be executed in 100 ns. Another fault simulator [20] implemented on the vector supercomputer FACOM VP-400 takes 0.938 s for the same simulation. Our CA uses only 0.29 s to complete the same simulation, in which 0.267 s are for the preprocessing. The speedup obviously will be more significant if more patterns are simulated.

VII. CONCLUSION AND DISCUSSION

We have shown a CA with eight states to implement the logic and fault simulation of combinational circuits. Simulation results on ISCAS-85 benchmark circuits show that our method is superior to the previously reported ones. Compared with the result given in [9], our CA achieves an up to 9.24 billion GEPS performance, which is 1.6 times higher than their peak performance. It is worth noting that the performance reported in [9] is on a circuit of 21446 gates, and the largest ISCAS-85 benchmark circuit has only 3512 gates. Our CA will run in a much higher simulation rate for such a big circuit. Although the number of cells required for larger circuits may exceed ten thousand, it is doable based on today's submicron VLSI technology and the fact that the cells are small and regularly connected so that they can be packed very densely. Encouraged by the simulation results, we are currently in the process of constructing an experimental CA machine. There however are a few problems to be solved next.

The first problem is the handling of timing errors, which may be required by some serious users. Event-driven simulation is a general approach for logic and fault simulation, especially when timing errors need to be reported, since it does not violate causality and correctly handles element delays in combinational as well as sequential circuits. However, MPP is not suitable for event-driven simulation owing to the need of maintaining an event queue, especially for a large circuit [9]. This dilemma needs to be further investigated. Preserving causality ensures the functional correctness of parallel logic and fault simulation, which is guaranteed in our method by leveling the circuit. Besides, we have designed the rules to make sure that a gate evaluation will not be finished until all its input signals arrive. We assume the zero-delay model at present. Incorporation of a more realistic delay model is our future work.

A fault simulator unable to handle sequential circuits is of little practical interest. Our work is progressive, and we intend to experiment on the next version of CA for sequential circuits since the results for the first version is encouraging. We now briefly describe how CA can deal with sequential circuits. We first map the combinational part of the circuit into CA as described before. Since a flip-flop (FF) has only one fanin and at least one fanout, the fanin edge needs no corresponding CA cell, and the gate providing output signal to a FF with n fanouts has n corresponding fanout cells. A special row of cells, called the *feedback path*, is inserted at an appropriate position for conveying all feedback signals. Some fanin cells thus will receive input signals from both the left neighbor and the feedback path. At the fanout column, some output signals flow toward the next level (primary outputs) and others flow upward or downward to the feedback path (state variables). When a feedback signal reaches the feedback path, it turns the flow direction to the left. The sequential CA can be implemented by either a 2-D or a 3-D cellular array architecture. We show a 3-D sequential CA example in Fig. 12. The upper layer is called the *feedback layer*. The fanout cells on the feedback layer replace D FFs. In the figure, e.g., DFF1 whose input is from node l has two fanouts, so there are two fanout cells on the feedback layer for DFF1.

For the 2-D architecture, the mapping process is the same as that for combinational circuits except for the fanin edges from FF's and the fanout edges towards them. A fanin edge from an FF has its corresponding fanin cell, moreover, it has a corresponding dummy fanout cell as its left neighbor (since the feedback path goes from right to left) which is inserted in the previous fanout column. A fanout edge towards an FF has a corresponding fanout cell and a dummy fanin cell as its right neighbor at the next fanin column. A regular and systematic sequential circuit mapping therefore can be completed. Note that we discuss only FF's. In the scope of fault grading, embedded memories (such as caches and register files) usually are handled separately using functional fault models.

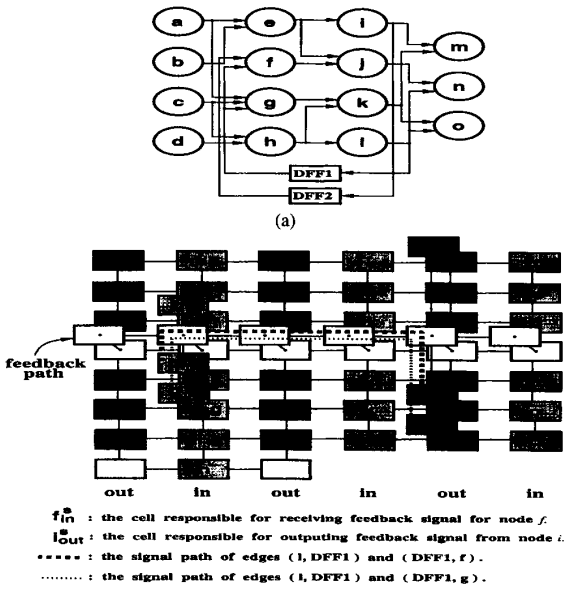


Fig. 12. A 3-D sequential CA example. (a) A sequential circuit. (b) Mapping on CA.

Another important issue is scalability, since the circuit size may exceed the hardware resource we currently have. Although we can partition a circuit (according to levels) into several stages so that the stages can be loaded into and simulated by the CA board one by one, the performance will be degraded due to host communication and software handling. To maintain the benefit of hardware acceleration, software intervention (which becomes the performance bottleneck) should be minimized. Our CA simulator is a 2-D cellular array architecture, so we can design a CA board such that it can be extended in both dimensions. The CA boards can communicate with each other through a high-speed bus. In this way we can use the CA simulator for a large circuit.

REFERENCES

[1] P. Agrawal, "Concurrency and communication in hardware simulators," *IEEE Trans. Computer-Aided Design*, vol. CAD-5, no. 4, pp. 617–623, Oct. 1986.

[2] V. D. Agrawal and S. T. Chakradhar, "Performance analysis of synchronized iterative algorithms on multiprocessor systems," *IEEE Trans. Parallel Distributed Syst.*, vol. 3, pp. 739–746, Nov. 1992.

[3] L. M. Huisman, I. Nair, and R. Daoud, "Fault simulation of logic designs on parallel processors with distributed memory," in *Proc. Int. Test Conf. (ITC)*, 1990, pp. 690–696.

[4] S. Bose and P. Agrawal, "Concurrent fault simulation of logic gates and memory blocks on message passing multicomputers," in *Proc. Design Automation Conf. (DAC)*, June 1992, pp. 332–335.

[5] P. Agrawal, V. D. Agrawal, K.-T. Cheng, and R. Tutundjiar, "Fault simulation in a pipelined multiprocessor system," in *Proc. Int. Test Conf. (ITC)*, Aug. 1989, pp. 727–734.

[6] Y. Kitamura, "Hardware engines for logic simulation," in *Advances in CAD for VLSI, Vol. 2, Logic Design and Simulation*, E. Hörbst, Ed. North-Holland: Elsevier, 1986, pp. 165–192.

[7] Y. Kitamura, "Exact critical path tracing fault simulation on massively parallel processor AAP2," in *Proc. Int. Conf. Computer-Aided Design (ICCAD)*, Nov. 1989, pp. 474–477.

[8] V. Narayanan and V. Pitchumani, "Fault simulation on massively parallel SIMD machines: Algorithms, implementations and results," *J. Electronic Testing: Theory Applicat.*, vol. 3, pp. 79–92, Sept. 1992.

[9] A. D. Gloria and P. Faraboschi, "Massive parallelism in multi-level simulation of VLSI circuits," *Integration*, vol. 14, no. 2, pp. 145–171, 1992.

[10] M.-J. Chung and Y. Chung, "Efficient parallel logic simulation techniques for the connection machine," in *Proc. Design Automation Conf. (DAC)*, 1990, pp. 606–614.

[11] N. Ishiura, H. Yasuura, and S. Yajima, "High-speed logic simulation on vector processors," *IEEE Trans. Computer-Aided Design*, vol. CAD-6, no. 3, pp. 305–321, May 1987.

[12] R. Raghavan, J. P. Hayes, and W. R. Martin, "Logic simulation on vector processors," in *Proc. European Design Automation Conf. (EDAC)*, 198, pp. 268–2718.

[13] M. Abramovici, B. Krishnamurthy, R. Mathews, B. Rogers, M. Schulz, S. Seth, and J. Waicukauski, "What is the path to fast fault simulation? (a panel discussion)," in *Proc. Int. Test Conf. (ITC)*, 1988, pp. 183–192.

[14] D. Harel and B. Krishnamurthy, "Is there hope for linear time fault simulation?" in *Proc. Int. Symp. Fault Tolerant Computing (FTCS)*, July 1987, pp. 28–33.

[15] N. Ishiura and S. Yajima, "Linear time fault simulation algorithm using a content addressable memory," in *Proc. European Design Automation Conf. (EDAC)*, 1992, pp. 442–445.

[16] M. L. Bailey, "How circuit size affects parallelism," *IEEE Trans. Computer-Aided Design*, vol. 11, pp. 208–215, Feb. 1992.

[17] S. Wolfram, "Statistical mechanics of cellular automata," *Rev. Modern Phys.*, vol. 55, pp. 601–644, 1983.

[18] K. Culik, J. Pachl, and S. Yu, "On the limit sets of cellular automata," *SIAM J. Computing*, vol. 18, no. 4, pp. 831–842, Aug. 1989.

[19] R. Gould, *Graph Theory*. Menlo Park, CA: Benjamin/Cummings, 1988, ch. 4.

[20] N. Ishiura, M. Ito, and S. Yajima, "Dynamic two-dimensional parallel simulation technique for high-speed fault simulation on a vector processor," *IEEE Trans. Computer-Aided Design*, vol. 9, no. 8, pp. 868–875, Aug. 1990.

Yih-Lang Li received the B.S. degree in nuclear engineering from National Tsing Hua University, Hsinchu, Taiwan, in 1987, and the M.S. degree in computer science from National Tsing Hua University, Hsinchu, Taiwan, in 1990.

He is currently working towards the Ph.D. degree at National Tsing Hua. His research interests include VLSI testing, parallel processing, and computer architecture.

Cheng-Wen Wu (S'86–M'87) received the B.S.E.E. degree from National Taiwan University, Taipei, Taiwan, in 1981, and the M.S. and Ph.D. degrees, both in electrical and computer engineering, in 1985 and 1987, respectively, from the University of California, Santa Barbara.

From 1985 to 1987 he was a Postgraduate Researcher at the Center for Computational Sciences and Engineering at UCSB. Since 1988 he has been with the Department of Electrical Engineering, National Tsing Hua University, Hsinchu, Taiwan, where he is currently a Professor. His interests include VLSI testing and design for testability, and design of high-performance application-specific VLSI circuits and systems.