Logic and Fault Simulation by Cellular Automata

Yih-Lang Li

Department of Computer Science
National Tsing Hua University
Hsinchu, Taiwan 30043, R.O.C.

Cheng-Wen Wu *

Department of Electrical Engineering
National Tsing Hua University
Hsinchu, Taiwan 30043, R.O.C.

Abstract

We propose a massively parallel architecture to speed up the logic and fault simulation. We use a 2-D cellular automata *(CA) to implement the logic and fault simulation of combinational circuits. Our CA has six cell states, and operates in a pipelined fashion. Experimental results on ISCAS85 benchmark circuits show that our CA outperforms the previously reported parallel simulators. As to pure logic simulation, our CA performs up to 9.24 billion GEPS using a 20 MHz clock and 8-bit words as opposed to 5 billion GEPS.*

1 Introduction

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 complexity of sequential logic simulation is $O(n)$, where n is the number of gates in a circuit. Simulation of 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 [1, 2]. The experimental result of [1] showed 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.

Massively parallel processing (MPP) has been used to speed up logic and fault simulation [3, 4, 5, 6]. In [5], a massively parallel compiled model simulation environment for multi-level (i.e., from behavior level to switch level) was proposed. The compiler handles circuit partitioning and scheduling. The performance of simulation relies heavily on the quality of compiler, and the process of compilation may be time-consuming. For a 20MHz 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 [3, 4]. In [7, 8], Ishiura and Reghavan vectorized the logic simulation algorithm on vector processors, and a maximum performance of 7.7 billion GEPS is obtained in [7]. All of the above works boost the performance for fault simulation, but its complexity remains at $O(n^2)$. In fact, there is little hope to do fault simulation in linear time so far as conventional computer architectures are assumed [9]. In [10], they used a content addressable memory 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.

In this paper, we present a unilateral *cellular automata* (CA) model to implement logic and fault simulation for combinational circuits. The 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. 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 wordlength is 8 bits. 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 to produce correct outputs but also the ability of massively parallel processing inherited from the CA. Experimental results on ISCAS85 benchmark circuits are obtained. Compared with that in [4], the time required for simulating one test pattern (in average) is shorter by three to four orders of magnitude.

2 CA Model

A. Formal Definition

A formal definition of *cellular Automaton* is given in [11]. Let Z be the set of integers. A *k-D cellular automaton* is an infinite array, indexed by Z^k, of cells.

*This work was supported in part by the National Science Council, R.O.C., under Contract NSC82-0404-E007-186.

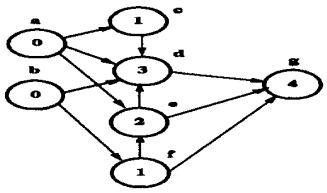

Figure 1: A leveled digraph.

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 $(I_1, I_2, ..., I_h)$ of (relative) locations, where $I_j \in Z^k$, $1 \leq j \leq h$. The local function is a total function:

$$f : S^h \longmapsto S. \tag{1}$$

A more detail definition of CA can be found in [11].

B. Our CA Model

In our CA model, cells are connected to form a 2D 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 Sec. 4. Formally, our CA can be written as a quadruple $A = (k = 2, |S| = 8, N = ((-1,0),(1,0),(0,1),(0,-1),(0,0)), f)$.

C. Other Terminology

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 [12] 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 $l(v) = l(u) + 1$.

3 Graph Embedding

A. Problem Definition

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 gate of the circuit and whose directed edges stand for the propagating directions of signals, map this digraph onto a two-dimensional tessellation so advisably as to be able to simulate the propagation of signals throughout the circuit on the tessellation.

B. The Proposed Approach

We complete the graph embedding by three steps: (1) level the acyclic digraph, (2) modify the leveled digraph into an LN, and (3) map the LN onto the tessellation after ordering the sequence of nodes with the same level.

We first level the digraph by the sequence of gate evaluations of nodes. Fig. 1 shows a leveled digraph.

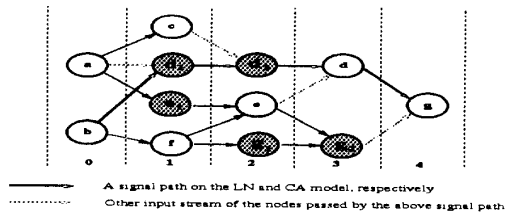

Figure 2: The LN obtained from Fig. 1, where a shadowed node stands for a newly created node.

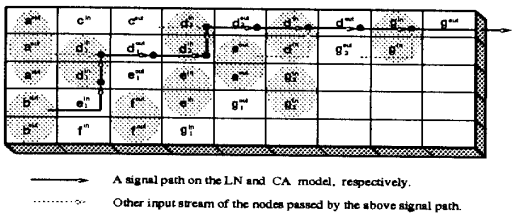

Figure 3: A mapping from an LN onto our CA architecture, where the cells covered by a shadowed ellipse are mapped to the inward or outward edges of the same node.

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 to guarantee that every edge connects two nodes in adjacent levels. We define the set of nodes N_j to be $\{i \mid (i, j) \in E \text{ and } l(j) - l(i) > 1, i, j \mid V\}$.

- $\mid N_j \mid > 1$. 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, and each node i in N_j except node n has a new outward edge $(i, h_{l(i)+1})$, 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. 2 is an example.

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. 3 shows a mapping example.

Different ordering of nodes at the same level will result in different 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 to random ordering.

4 Logic and Fault Simulation

We divide logic simulation into three steps: (1) input accumulation, (2) output distribution, and (3) output propagation to 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 in Fig. 2, where the signal path from node b to node g is highlighted, and its corresponding path on the CA is shown in Fig. 3.

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 next level. During this step, upward and downward moves are possible and a node may be passed by the signals moving in different directions. The input accumulation of 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 lower d_1^{in} to upper d_1^{in} and the slender arrow from a^{out} to 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, TopFanout, NewPipe, and TopNewPipe. The first two states are available for the fanin cells and the other for the fanout cells.

A. Input Accumulation and Output Propagation

Because a node with n fanin edges is distributed on 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:

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

The fanin cells can be in one of Fanin and BotFanin states: 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 need not do logic operation when the input accumulation proceeds. The fanin cells keep their next state unchanged regardless of the information 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 TopNewPipe 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. 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, where 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 the propagation of output signal propagation and the accumulation of input signal, 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 before 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 TopFanout, 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 the preloading for output propagation at fanin columns, the cells in state TopFanout 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 TopFanout begin to distribute the output signal, they get 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 the distribution the signals passing through a node are in the same direction, although the moves in opposite directions may coexist at fanout columns. When a signal passes its target cell, this cell changes its state to NewPipe or TopNewPipe to announce the arrival of a new signal. A cell in state NewPipe (TopNewPipe) changes its next state to Fanout (TopFanout).

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 wordlength 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. The 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. We assume that only the fanin cells can be faulty (after fault collapsing). Both *faulty* register and *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.

5 Performance Analysis

We first define the following variables.

$d_{f,l}$: the difference of the arrival times between the first arriving signal and the last.

T: the set of the cells in state TopFanout.

B: the set of the cells in state BotFanin.

$P_{out}(i)$: the number of the cells in T with a positive (negative) offset above (below) cell i if the *OffReg* register of cell i is positive (negative).

$P_{in}(i)$: the number of the cells with a positive (negative) offset above (below) cell i if the *OffReg* register of cell i is positive (negative).

$N_{in}(i)$: the fanin of the node containing cell i.

Let us consider the fanout columns. Output signal distribution of all signals can be done simultaneously except for the case when the downward data path is occupied by another downward signal beyond themselves, or similarly for an upward signal. The upper bound of distribution time for a fanout column therefore is

$$t_{fanout} \leq d_{f,l} + max_{i \in T}\{OffReg(i) + P_{out}(i)\}, \quad (2)$$

where $0 \leq OffReg(i) \leq n$ and $0 \leq P_{out} \leq \mid T \mid$.

Similarly, the upper bound of signal distribution time for a fanin column is

$$t_{fanin} \leq d_{f,l} + max_{1 \leq i \leq n}\{OffReg(i) + P_{in}(i)\} + max_{i \in B}\{N_{in}(i)\}, \quad (3)$$

where $0 \leq OffReg(i), P_{in}(i) \leq \lceil n/2 \rceil - 1$. The sum of *OffReg* and P_{in} is less than n.

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

$$t_d + f(i) \geq c(i+1),\ 0 \leq i \leq N, \quad (4)$$

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

$$t_d = max_{0 \leq i < N}\{c(i+1) - f(i)\}. \quad (5)$$

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 (2) and (3) by adding a global information for each column, which records the total number of cells having completed their operations. This can make the signals of one column pass to next column simultaneously. Hence, the $d_{f,l}$ term in (2) and (3) can be eliminated. Therefore,

$$t_d = max_{0 \leq i \leq N}\{c(i) - f(i)\} + 3. \quad (6)$$

Now the maximum offset for output propagation and output distribution becomes a dominant factor. We propose another method to reduce the maximum offset, which, however, doubles the hardware cost (in number of CA cells). We first translate a circuit into a new format before graph embedding, where the fanin and fanout of each gate are at most two, and then apply a new bidirectional scanning algorithm on this new format circuit to optimize the gate ordering.

6 Experimental Result

We have implemented a CA simulator on a SUN Sparc2 station in 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 0.15 second for all ISCAS85 benchmark circuits except circuits c6288 and c7552.

Table 1: Experimental results on CA.

Bench Ckts	$Cell_{total}$		Latency		T_{order}		Seq time
	CA_∞	CA_2	CA_∞	CA_2	CA_∞	CA_2	
c17	32	32	8	6	0	0	0.00
c432	1691	4877	141	22	0.067	0	0.01
c499	1872	3726	207	29	0.083	0	0.01
c880	4426	8844	252	30	0.117	0	0.01
c1355	5904	8470	201	38	0.117	0	0.03
c1908	11622	16085	247	53	0.417	0	0.05
c2670	15613	32511	592	49	0.533	0.067	0.10
c3540	14591	27487	628	69	0.5.1	0.05	0.17
c5315	32321	66439	1674	103	1.1	0.283	0.48
c6288	58848	84898	451	49	1.85	0.267	0.34
c7552	33616	62106	1663	70	1.003	0.283	0.65

We compare the experimental results of two CA versions in Table 1. We use CA_∞ and CA_2 to represent the CA without and with the process of gate translation, respectively. We list the total number of cells, t_d, CPU time (sec) for ordering, and the sequential runtime of logic simulation ona SUN Sparc2 in Table 1. In the table, it is clear that the disadvantage of gate translation on the circuit is to roughly double the number of cells required. The CA with gate translation is potentially good for large circuits, since the gate interconnection complexity does not increase. The increase in t_d is only slightly as the circuit size increases. This results in a higher performance measured by billion GEPS for larger circuits. We show this in Fig. 5. In Fig. 4, we compare the result with that obtained by running a sequential algorithm on

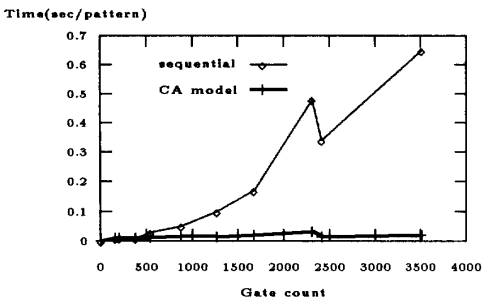

Figure 4: Performance comparison between CA and sequential algorithm.

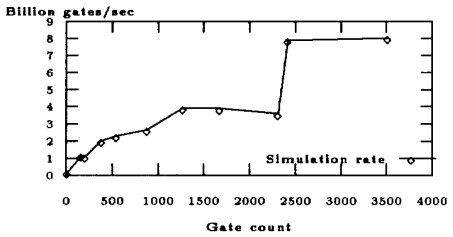

Figure 5: CA has a higher simulation rate for larger circuits.

a SUN Sparc2. For example, the completion time of c432 is 0.01 sec under the assumption that the CA operates at 14.1 kHz.

In [5], 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 [7], 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 2.

For fault simulation, in [10], the simulator takes 3.13 seconds for simulating 512 patterns on c6288 circuit assuming that one instruction execution time is $100ns$. Another fault simulator in [13] implemented on the vector supercomputer FACOM VP-400 takes 0.938 seconds for the same simulation. Our CA only takes 0.29 seconds for the same simulation, where 0.267 seconds for the preprocessing time and 0.023 seconds for 512 patterns simulation.

Table 2: Peak Logic Simulation Rate Table.

Simulator	Peak rate GEPS $*10^9$	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

7 Conclusion

We show a CA with eight states to implement the logic and fault simulation of combinational circuits. The experimental results on ISCAS85 benchmark circuits show that our method is superior to the previously reported ones. Compared with the result in [5], our CA achieves an up to 9.24 billion GEPS performance, which is 1.6 times higher than the peak performance in [5]. It is worth noting that the performance reported in [5] is on a circuit of 21446 gates, and the largest ISCAS85 benchmark circuit has only 3512 gates. Our CA will perform in a much higher simulation rate for a circuit of 21446 gates. 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.

References

[1] P. Agrawal, "Concurrency and communication in hardware simulators," *IEEE Trans. on Computer-Aided Degin*, vol. CAD-5, pp. 617–623, Octbor 1986.

[2] L. M. Huisman, I. Nair, and R. Daoud, "Fault simulation of logic designs on parallel processors with distributed memory," *IEEE International Test Conference*, pp. 690–696, 1990.

[3] V. Narayanan and V. Pitchumani, "A parallel algorithm for fault simulation on the connection machine," *IEEE International Test Conference*, pp. 89–93, Sept. 1988.

[4] V. Narayanan and V. Pitchumani, "A massively parallel algorithm for fault simulation on the connection machine," *26th ACM/IEEE Design Automation Conference*, pp. 734–737, 1989.

[5] A. D. Gloria and P. Faraboschi, "Massive parallelism in multi-level simulation of vlsi circuits," *INTEGRATION, the VLSI journal*, pp. 145–171, 1992.

[6] M. J. Chung and Y. Chung, "Efficient parallel logic simulation techniques for the connection machine," *EDAC*, pp. 606–614, 1990.

[7] N. Ishiura, H. Yasuura, and S. Yajima, "High-speed logic simulation on vector processors," *IEEE Transactions on Computer-Aided Design*, vol. CAD-6, pp. 305–321, May 1987.

[8] R. Raghavan, J. P. Hayes, and W. R. Martin, "Logic simulation on vector processors," *EDAC*, pp. 268–271, 1988.

[9] D. Harel and B. Krishnamurthy, "Is there hope for linear time fault simulation," *Proc. 17th FTCS*, pp. 28–33, July 1987.

[10] N. Ishiura and S. Yajima, "Linear time fault simulation algorithm using a content addressable memory," *EDAC*, pp. 442–445, 1992.

[11] K. Culik, J. Pachl, and S. Yu, "On the limit sets of cellular automata," *SIAM J. Comput.*, vol. 18, pp. 831–842, August 1989.

[12] R. Gould, *Graph Theory*. Benjamin/Cummings Publishing Company, Inc., 1988.

[13] N.Ishiura, M. Ito, and S. Yajima, "Dynamic two-dimensional parallel simulation technique for high-speed fault simulation on a vector processor," *IEEE Trans. on Computer-Aided Degin*, vol. CAD-9, pp. 868–875, August 1990.