CONCURRENT FAULT SIMULATION OF MOS DIGITAL CIRCUITS

Michael D. Schuster and Randal E. Bryant

California Institute of Technology
Pasadena, California 91125

5101:TM:83

To be presented at the Conference on Advanced Research in VLSI, to be held at the Massachusetts Institute of Technology, January 1984. Proceedings published by Artech House, Inc., Dedham, MA 02026.

ABSTRACT

The concurrent fault simulation technique is widely used to analyze the behavior of digital circuits in the presence of faults. We show how this technique can be applied to metal-oxide-semiconductor (MOS) digital circuits when modeled at the switch-level as a set of charge storage nodes connected by bidirectional transistor switches. The algorithm we present is capable of analyzing the behavior of a wide variety of MOS circuit failures, such as stuck-at-zero or stuck-at-one nodes, stuck-open or stuck-closed transistors, or resistive opens or shorts. We have implemented a fault simulator FMOSSIM based on this algorithm. The capabilities and the performance of this program demonstrate the advantages of combining switch-level and concurrent simulation techniques.

This research was supported in part by the IBM Corporation and by the Defense Advanced Research Contracts Agency, ARPA Order 3771. Michael Schuster was supported in part by a Bell Laboratories Ph.D. Scholarship.

CONCURRENT FAULT SIMULATION OF MOS DIGITAL CIRCUITS

Michael D. Schuster and Randal E. Bryant

Department of Computer Science
California Institute of Technology
Pasadena, California 91125

ABSTRACT

The concurrent fault simulation technique is widely used to analyze the behavior of digital circuits in the presence of faults. We show how this technique can be applied to metal-oxide-semiconductor (MOS) digital circuits when modeled at the switch-level as a set of charge storage nodes connected by bidirectional transistor switches. The algorithm we present is capable of analyzing the behavior of a wide variety of MOS circuit failures, such as stuck-at-zero or stuck-at-one nodes, stuck-open or stuck-closed transistors, or resistive opens or shorts. We have implemented a fault simulator FMOSSIM based on this algorithm. The capabilities and the performance of this program demonstrate the advantages of combining switch-level and concurrent simulation techniques.

INTRODUCTION

Test engineers use fault simulators to determine how well a sequence of test patterns, when applied to the inputs of an integrated circuit, can distinguish a good chip from a defective one. The fault simulator is given a description of the good circuit, a set of hypothetical faults in the circuit, a specification of the observation points of the test (e.g. the output pins of the chip), and a sequence of test patterns. It then simulates how the good circuit and all of the faulty circuits would behave when the test patterns are applied to the inputs. A fault is considered detected if at any time the simulation of that particular faulty circuit produces, at some observation point, a logic value different than that produced by the good circuit. By keeping track of which faults have been detected and which have not, the fault simulator can determine the *fault coverage* of the test sequence, which is defined as the ratio of the number of faults detected to the total number simulated. The simulator can also provide the user with information about which faults have not been detected, either because the test sequence failed to exercise the defective part of the circuit, or because the sequence failed to make the effect of such an exercise visible at some observation point. This information guides the engineer in extending or modifying the test sequence to improve its fault coverage. Such a tool is invaluable for developing test patterns for today's complex digital systems.

For a large integrated circuit such as a microprocessor chip, thousands of faults must be simulated to adequately characterize the fault coverage of a test sequence. Furthermore, test sequences can involve thousands of patterns. Hence a simple *serial* simulation, in which the good circuit and each faulty circuit are simulated separately, would require far too much computation. Fortunately, clever algorithms can reduce the amount of computation considerably. A technique known as *concurrent simulation*^{1} exploits the fact that a faulty circuit typically differs only slightly from the good circuit. Rather than simulating each circuit separately, only the good circuit is simulated in its entirety. The simulator keeps track of how the network state of each faulty circuit differs from the network state of the good circuit by selectively simulating portions of the faulty network. To the user, it appears as if the program is simulating many circuits concurrently, but the amount of CPU time required is a small factor (e.g. often less than 10 times) greater than the time required to simulate the good circuit alone. Furthermore, the simulator can easily determine when a faulty circuit produces a value different than the good circuit at some observation point without stor-

This research was supported in part by the IBM Corporation and by the Defense Advanced Research Contracts Agency, ARPA Order 3771. Michael Schuster was supported in part by a Bell Laboratories Ph.D. Scholarship.

ing the entire output history of the good circuit simulation. Once a fault has been detected, the simulation of this particular faulty circuit can be dropped, thereby reducing the amount of computation required for the remainder of the simulation. Typically, the faults that cause great differences from the behavior of the good circuit, and hence require the most computational effort, are detected quickly. Consequently, fault dropping greatly improves the overall performance of the simulator.

Most existing logic simulators model a digital circuit as a network of logic gates, in which each gate produces values on its outputs based on the values applied to its inputs, and possibly on the value of its internal state. Some of these simulators extend the simple Boolean gate model, in which only the value 0 or 1 is permitted on each input and output, with additional logic values and special types of gates to model circuit structures such as busses and pass transistors. These simulators are not suitable for modeling faults in MOS digital circuits for two reasons: First, many MOS circuit structures cannot be adequately modeled as a set of logic gates. Creating gate-level descriptions of pass transistor networks, dynamic memory elements, and precharged logic is at best tedious and inaccurate, and at worst impossible, even with extended gate models. The user must translate the logic design by hand into a form compatible with the simulator, and the resulting simulation is inherently biased toward the user's understanding of the functionality of the circuit. Second, logic gate simulators are especially poor at predicting the behavior of a MOS circuit in the presence of faults. Even simple logic gates can become seemingly complex sequential circuits when a fault such as an open-circuited transistor occurs.[2] As a result, fault simulators based on logic gates can model only a limited class of faults, such as the gate outputs and inputs stuck-at-zero or stuck-at-one. Faults such as short circuits across transistors and between wires, or open circuits in transistors or wires, are beyond their capability. Furthermore, even the modeling of stuck-at faults is limited in accuracy when the logic gate description is an artificial translation of the actual circuit structure.

To remedy these problems with logic gate simulators, we propose that fault simulations of MOS circuits be performed at the *switch level* with the transistor structure of the circuit represented explicitly, but with each transistor modeled in a highly idealized way. This approach has proved successful for logic simulation in programs such as MOSSIM[3] and MOSSIM II,[4] because properties such as the bidirectional nature of field-effect transistors and the charge storage capabilities of the nodes in a MOS circuit are modeled directly, rather than by some artificial translation into logic gates. Unlike the precise, but time-consuming algorithms used by circuit simulators, switch-level simulators model the circuit in a sufficiently simplified way that they operate at speeds comparable with conventional logic gate simulators. Furthermore, our switch-level logic model is well suited for modeling a variety of failures in MOS circuits in a reasonably realistic way, because many faults can be viewed as creating new switch-level networks which differ from the switch-level representation of the good circuit. Hence, while the switch-level model has proved successful for logic simulation, it seems especially attractive for fault simulation. Hayes[5] has proposed the Connector-Switch-Attenuator representation of logic circuits for modeling faults, and our switch-level model has essentially the same capabilities.

We have adapted the technique of concurrent simulation to implement a fault simulator for MOS circuits, where the problem is viewed as one of simulating a large number of nearly identical switch-level networks. This program FMOSSIM can simulate a large variety of MOS circuits, under a variety of fault conditions, at much higher speeds than would be possible with serial simulation. Other concurrent fault simulators for MOS have been implemented,[6] but these could only model a very limited class of networks. In this paper, we will present an overview of the switch-level model and how different faults can be represented in it. We also discuss our concurrent, switch-level simulation algorithm and present performance results from FMOSSIM.

NETWORK MODEL

The following network model is implemented in the simulators MOSSIM II and FMOSSIM. It provides a more general transistor model than provided by other switch-level simulators, giving better capabilities for fault injection. A switch-level network consists of a set of *nodes* connected by a set of *transistors*. Each node has a state 0, 1, or X, where 0 and 1 represent low and high voltages, respectively. The X state represents an indeterminate voltage arising from an uninitialized node, from a short circuit, or from improper charge sharing. No restrictions are placed on how transistors are interconnected.

Each node is classified as either an *input* node or a *storage* node. An input node provides a strong signal to the network, as does a voltage source in an electrical circuit. Its state is not affected by the actions of the network. Examples include the power and ground nodes *Vdd* and *Gnd*, which act as constant sources of the states 1 and 0, respectively, as well as any clock or data inputs.

The state of a storage node is determined by the operation of the network. Much like a capacitor in an electrical circuit, a storage node holds its state in the absence of connections to input nodes. To provide a simple model of charge sharing, each storage node is assigned a discrete *size* from the set $\{\kappa_1, \kappa_2, \ldots, \kappa_q\}$, where the sizes are ordered $\kappa_1 < \kappa_2 < \cdots < \kappa_q$. A larger storage node is assumed to have much greater capacitance than a smaller one. When a set of storage nodes charge share, the states of the largest nodes in the set override the states of the smaller nodes. The number of different sizes required (q) depends on the circuit to be simulated. Most circuits can be represented with just two node sizes. In this representation, high capacitance nodes such as busses assigned size κ_2, and all other nodes are assigned size κ_1.

A transistor is a device with terminals labeled *gate*, *source*, and *drain*. No distinction is made between the source and drain connections — each transistor is symmetric and bidirectional. Because transistors can be either *n-type*, *p-type*, or *d-type*, both nMOS and CMOS circuits can be modeled. A d-type transistor corresponds to a negative threshold depletion mode device. A transistor acts as a resistive switch connecting or disconnecting its source and drain nodes according to its type and the state of its gate node, as shown in Figure 1. Transistor states 0 and 1 represent open (nonconducting) and closed (fully conducting) conditions, respectively. The X state represents an in-

gate state	n-type	p-type	d-type
0	0	1	1
1	1	0	1
X	X	X	1

Figure 1. Transistor state function

determinate condition between open and closed, inclusive.

To model the behavior of ratioed circuits, each transistor is assigned a discrete *strength* from the set $\{\gamma_1, \gamma_2, \ldots, \gamma_p\}$, where strengths are ordered $\gamma_1 < \gamma_2 < \cdots < \gamma_p$. A stronger transistor is assumed to have much greater conductance than a weaker one. When a storage node is connected to a set of input nodes by paths of conducting transistors, its resulting state depends only on the states of the input nodes connected by paths of greatest strength. The strength of a path is defined to equal the strength of the weakest transistor in the path. The total number of strengths required (p) depends on the circuit to be modeled. Most CMOS circuits do not utilize ratioed logic and hence can be modeled with just one transistor strength. Most nMOS circuits require only two strengths, with pull-up loads assigned strength γ_1 and all other transistors assigned strength γ_2.

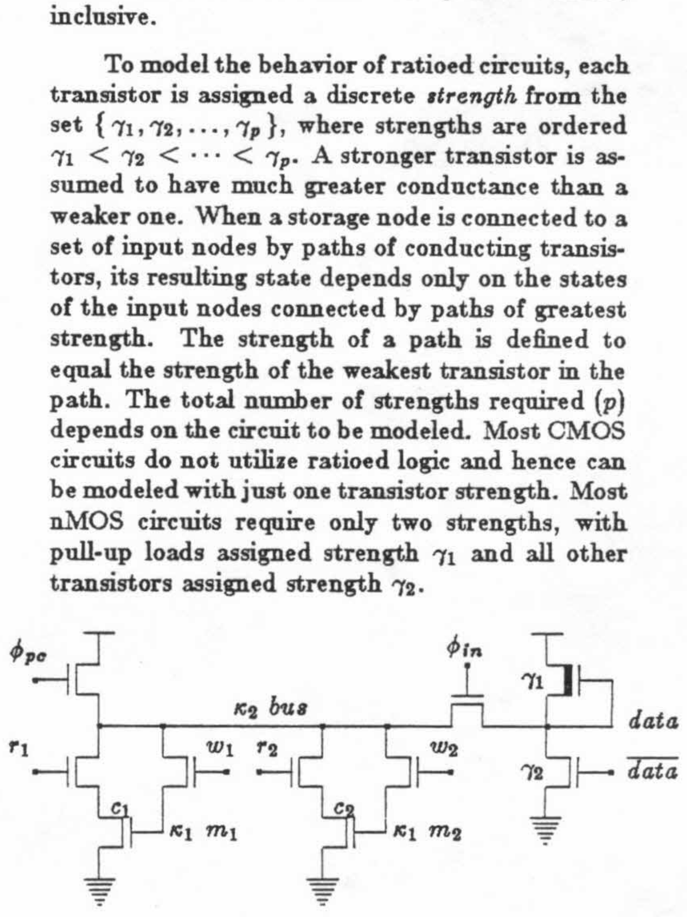

Figure 2. Three transistor dynamic RAM

As an example of a switch-level network, consider the three transistor dynamic RAM circuit shown in Figure 2. The bus node has size κ_2 to indicate that it can supply its state to the size κ_1 storage node (m_1 or m_2) of the selected memory element during a write operation (when w_1 or w_2 is 1) and to the size κ_1 drain node (c_1 or c_2) of the selected storage transistor during a read operation (when r_1 or r_2 is 1). The d-type pull-up transistor in the input inverter has strength γ_1, to indicate that it can drive the bus high only when the strength γ_2 pull-down transistor is not conducting. The strengths of all other transistors in the circuit are arbitrary, since they are not involved in ratioed path formation (except possibly when faults are present).

The switch-level network model strikes a reasonable balance between a detailed electrical model and an abstract logical model. As a result of this abstraction, the model may not predict the true behavior of circuits such as sense amplifiers and arbiters which rely on detailed analog properties. Moreover, the network model does not contain enough detail to accurately model timing behavior, because even in circuits with straightforward logical behavior, timing can be subtle. However, experience has shown that switch-level simulation works quite well for verifying logic designs.

FAULT INJECTION

Faults are represented in FMOSSIM as though extra *fault transistors* were added to the network, much like that proposed by Lightner and Hachtel[7] In the implementation, however, many of these faults are injected without actually adding fault transistors; nevertheless, the behavior is equivalent to what is described below. The gate nodes of the fault transistors are considered to be extra *fault inputs* to the network that control the presence or absence of the failures. A variety of MOS failures can be modeled with this method. For example, a short circuit between two nodes is modeled by connecting the nodes with a fault transistor that is open in the good circuit and closed in the faulty circuit. Similarly, an open circuit is modeled by splitting a node into two parts and connecting the resulting nodes with a fault transistor that is closed in the good circuit and open in the faulty circuit. By adjusting the strength of the fault transistor, the resistance of the short or open may be modeled in an approximate way. For example, if the strength of the fault transistor is set to γ_{p+1} (i.e. a strength greater than that of any normal transistor), then setting this transistor state to 1 shorts the source and drain nodes together such that they act as a single node. Moreover, because the state of each fault transistor can be controlled independently, both single and multiple faults can be injected.

Figure 3 illustrates the use of fault transistors

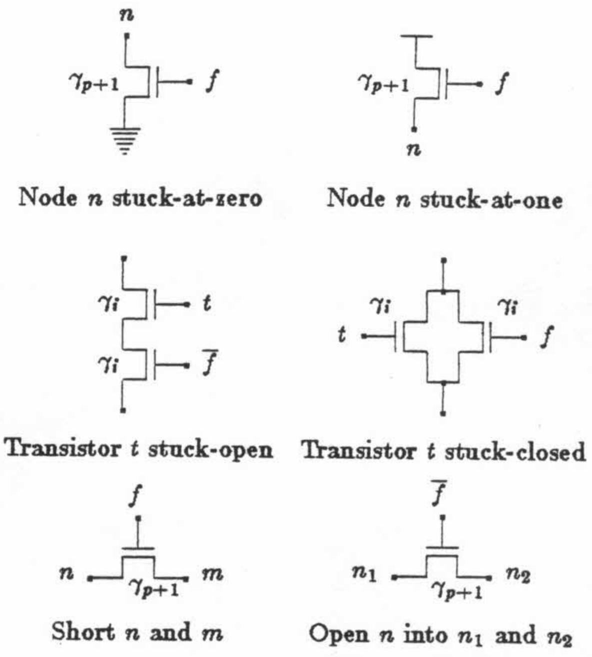

Figure 3. Modeling MOS failures

to create a variety of circuit faults. Those transistors with gate nodes labeled f are normally 0, but are set to 1 to create the fault; the transistors with gate nodes labeled $\bar{f}$ are normally 1, but are set to 0 to create the fault. A stuck-at-zero or stuck-at-one node fault can be modeled by inserting a strength γ_{p+1} fault transistor to short the node to *Gnd* or to *Vdd*, respectively. A stuck-closed transistor fault is injected by shorting the transistor's source and drain together with a fault transistor whose strength equals that of the failing transistor. Similarly, a stuck-open transistor fault is modeled by putting a fault transistor in series with it. In FMOSSIM, both stuck-at node states and stuck-at transistor states are implemented without extra fault transistors, while other faults require additional transistors to be inserted into the network.

Although FMOSSIM can model a larger class of faults than can be modeled by logic gate fault simulators, it still provides only a simplified representation of the faulty circuit. For example, the effects of manufacturing defects such as incorrect transistor thresholds, pinholes in the gate oxides, and variations in the circuit delays, cannot be described accurately. The effects of resistive shorts and opens can only be approximated. In fact, even existing circuit simulators cannot model defects that change the basic nature of the devices, such as pinholes in the gate oxides. However, even if the fault models supported by our simulator do not exactly match the failure modes in actual chips, the program can still help the designer in developing a set of test patterns. For circuits implemented in bipolar technologies such as TTL, experience has shown that a test sequence that yields a high level of coverage for single stuck-at-zero and stuck-at-one faults in the logic gate network generally provides a good test of the circuit. It seems reasonable to expect that the test coverage measured by a switch-level fault simulator for an idealized set of faults should reliably predict how well the test sequence will work on a MOS circuit. Such a conjecture, however, can only be confirmed by actual experience in a manufacturing environment.

Many faults in our model have the effect of creating an **X** state on a node when the good circuit has a 0 or 1. For example, if the control signal w_1 in the circuit shown in Figure 2 is stuck-at-zero, bit m_1 of the memory will never be initialized and will remain at **X**. On the other hand, if the precharge clock ϕ_{pc} is stuck-at-one, any time we try to read a 1 value out of a memory cell, a short circuit will develop between Vdd and Gnd giving an **X** on the bus. Whether or not such **X**'s would be detected in an actual test depends on detailed characteristics of the circuit that cannot be predicted at the switch-level, such as the initial voltages of dynamic nodes, how the voltage would divide across a shorting path, and the thresholds of the devices sensing these **X** values. On one hand, a pessimist might argue that an **X** in a faulty circuit should be considered undetectable, because there is no guarantee that the **X** will produce an effect different than the state of the node in the good circuit. On the other hand, a fault that prevents the circuit from being initialized, such as a stuck-at-zero clock line, would clearly be quickly detected. As a compromise FMOSSIM allows the user to specify a *soft detect* limit l such that if in the good circuit some output changes both to 1 and to 0 at least l times each, while the output in a faulty circuit remains at **X**, then this fault is considered detected. This approach seems to work reasonably well in practice.

BEHAVIORAL MODEL

The operation of a MOS circuit is characterized in the switch-level model in terms of its *steady state response function*,[8,9] which can best be explained in terms of an analogy to electrical networks. A MOS transistor behaves as a voltage-controlled, nonlinear resistor where the voltages of its gate, source and drain nodes control the resistance between its source and drain. Suppose in a transistor circuit we could control the transistor resistances independently of the node voltages. For a given setting of the transistor resistances, such a circuit acts as a network of passive elements which, for a given set of initial node voltages, has a unique set of steady state node voltages. Thus a function that maps transistor resistances and initial node voltages to steady state node voltages gives a partial characterization of the behavior of a transistor circuit. The steady state response function provides just this sort of characterization, but in terms of node and transistor states 0, 1, and **X**. That is, for a given set of initial node and transistor states, the steady state response function yields the set of states which the storage nodes would eventually reach if all transistors were held fixed in their initial states. This function only approximates network behavior, since it does not describe the rate at which nodes approach their steady states nor the effects of the changing transistor states as their gate nodes change state.

In general, a switch-level network may contain nodes and transistors in the **X** state. Such states arise from improper charge sharing or (transient) short circuits even in properly designed networks. The behavior of a network in the presence of **X** states must be described in a way that is neither overly optimistic (i.e. ignoring possible error conditions), nor overly pessimistic (i.e. spreading **X**'s beyond the region of indeterminate behavior). This can be accomplished by defining the steady state response of a node to be 0 or 1 if and only if the node would have this unique state regardless of whether each node and transistor in the **X** state had state 0 or 1; otherwise, the steady state of the node is defined to be **X**. Rather than computing the steady state for all possible combinations of the nodes and transistors in the **X** state set to 0 or 1 (a task of exponential complexity), an equivalent two-pass linear time algorithm is used[9] Each pass involves solving a set of equations expressed in a simple, discrete algebra using a relaxation algorithm.

Given a technique for computing the steady state response function, a switch-level logic simulator can be implemented that simulates the operation of the network by repeatedly performing *unit steps* until a stable state is reached. Each unit step involves computing the steady state response of the network, setting the storage nodes to these values, and setting the transistors according to the states of their gate nodes. This simulation technique implements a timing model in which transistors switch one time unit (i.e. one evaluation of the steady state response function) after their gate nodes change state. Such a timing model tells little about the speed of a circuit but usually suffices to describe the circuit's logical behavior. As with other unit delay simulations, this computation may not reach a stable condition due to oscillations in the circuit, and hence an upper bound must be placed on the number of steps simulated.

On a given unit step, often only a small portion of the network changes state, while the rest of the network remains inactive. Most logic simulators exploit this property by recomputing the output of a logic gate only if at least one of the gate's inputs has changed state. A similar effect is achieved in switch-level networks by viewing network activity as creating small *perturbations* of the network state, and only computing the effects of these perturbations incrementally rather than recomputing the state of the entire network. We say that a storage node is *perturbed* if it is the source or drain of a transistor that has changed state, or if it is connected by a transistor in the **1** or **X** state to an input node that has changed state. Such a perturbation can only affect storage nodes in the *vicinity* of the perturbed node, where two nodes are in the same vicinity if and only if there exists some path of transistors in the **1** or **X** state between the nodes which does not pass through any input nodes. This definition exploits the *dynamic* locality in the network where the source and drain of a transistor in the **0** state are considered to be electrically isolated. Typically, a vicinity contains only a few nodes, and hence activity remains highly localized.

```
unit-step($P$);
    $U := \emptyset$;
    for each $n \in P$ do
        $U := U \cup$ update-vicinity($n$);
    $P := \emptyset$;
    for each $n \in U$ do
        begin
            $P := P \cup$ perturb-transistors($n$);
            update-node($n$);
        end;
    return($P$);
end unit-step;
```

Figure 4. Implementation of unit step

Figure 4 shows a simplified implementation of the unit step operation that uses this incremental perturbation technique to recompute only selected parts of the network state. The argument P is a set of perturbed storage nodes derived from either new data and clock inputs to the circuit or from the last unit step. For each of these nodes, *update-vicinity* finds all the nodes in the same vicinity, computes their steady state response, and returns a set of nodes that changed state. These updated nodes are accumulated in the set U. Vicinities are found by a *depth first search*[10] originating at the perturbed node and tracing outward through transistors in the **1** or **X** state from source to drain until an input node is encountered. As each node is added to the vicinity, it is flagged to avoid duplication and endless cycles. For each updated node n in U, *perturb-transistors* finds all transistors whose gate node is n and checks to see if they have changed state. Nodes perturbed by these changing transistor states are accumulated in a new set P in preparation for the next unit step. Finally, *update-node* sets each updated node to its new state.

CONCURRENT SIMULATION

We have seen that the presence or absence of a fault in a switch-level network is controlled by the state of a fault input node. Suppose the test patterns that specify data and clock input values are extended to include values for the network's fault input nodes. Then the behavior of a set of faulty circuits can be determined by repeatedly simulating patterns that differ only in selected fault input values. Hence, concurrent fault simulation can be viewed as the problem of efficiently applying a large number of nearly identical test sequences to a single network. This viewpoint separates issues of fault modeling from concurrent simulation. For example, since values for fault input nodes are specified on an individual pattern by pattern basis, multiple and intermittent faults are easily modeled without changing the basic simulation algorithm. Furthermore, there are no inherent restrictions requiring that the data inputs of all test sequences be identical. Thus, concurrent simulation is useful not only for simulating faults, but for simulating sets of similar test patterns on a fault-free circuit.

The concurrent simulation algorithm is given a description of the network and a set of *test sequences* $T = \{t_0, \ldots, t_n\}$. A test sequence $t_i \in T$ consists of a sequence of test patterns, each specifying values for the data, clock and fault inputs of the network. The algorithm simulates the network to determine how each node behaves for each test sequence t_i. That is, at any point during the simulation, each node's state s_i in test sequence t_i is found. Since we assume that the behavior of the network differs only slightly from test sequence to test sequence, $s_i = s_0$ for most nodes in the network. This observation is exploited by representing node states compactly as a set of pairs $S = \{\langle t_i, s_i\rangle\}$, called a *state set*, where $\langle t_i, s_i\rangle \in S$ if and only if $i = 0$ or $s_i \neq s_0$. The behavior of the network for test sequence t_0 serves as a reference point, since states are explicitly stored only for test sequence t_0 and those sequences t_i whose states differ from t_0. For this reason, test sequence t_0 is called the *reference sequence*. For fault simulation, the reference sequence corresponds to the good circuit, while test sequences $t_i, i \neq 0$ differ only in selected fault input values, and hence correspond to faulty circuits. A node is said to be *diverged for* t_i if $s_i \neq s_0$. A node is said to be *diverged* if it is diverged for any t_i. If the gate node of a transistor is diverged, then the transistor itself is said to be diverged.

If a node is perturbed due to an input node or transistor changing state for the reference sequence, it is likely that the node is also perturbed for most other test sequences t_i. We exploit this observation by maintaining a set of perturbations of the form $P = \{\langle n_j, t_i\rangle\}$, called the *perturbation set*, where $\langle n_j, t_0\rangle \in P$ if and only if node n_j is perturbed for the reference sequence t_0 and $\langle n_j, t_i\rangle \in P, i \neq 0$, if and only if n_j is perturbed for t_i but not for the reference sequence. The perturbation $\langle n_j, t_i\rangle \in P$, where $i \neq 0$, indicates that the network has behaved differently in the area near node n_j for test sequence t_i when compared to its behavior for the reference sequence.

As described above, each unit step of the conventional switch-level simulation algorithm computes a steady state response for each node in the vicinity of a perturbed node, updates those nodes that have new steady states, and returns a set of perturbations for the next unit step. To generalize this operation for concurrent simulation, observe that the perturbation $\langle n_j, t_0\rangle \in P$ represents a perturbation not only for the reference sequence, but likely for most other test sequences. In general, the steady state response of nodes in a vicinity is a function of both their initial states as well as the states of the transistors whose source or drain node is in the vicinity. Thus, when the steady state response is computed for the nodes in some vicinity as a result of a perturbation for the reference sequence, we must check to see if any of the nodes or transistors are diverged. We expect that most of the time, for most test sequences t_i, nodes within the vicinity will not be diverged for t_i. In this case, the steady state response computation performed for the reference sequence will be valid for t_i, and hence there is no need to duplicate this computation for t_i. However, if some node n_j within the vicinity is diverged for t_i, then the steady state response computation using the states of the nodes and transistors for the reference sequence may not be valid for t_i. To guarantee that an accurate computation be performed for t_i, the perturbation $\langle n_j, t_i\rangle$ is added to P. In effect, we are simply scheduling a steady state response computation that will be performed sometime later. Diverged transistors are handled in a similar manner, for if some transistor with source n_s or drain n_d in the vicinity is found to be diverged for t_i, then the perturbations $\langle n_s, t_i\rangle$ and $\langle n_d, t_i\rangle$ are added to P.

To determine the steady state response for nodes in the vicinity of a perturbation $\langle n_j, t_i\rangle$, where $i \neq 0$, states of the nodes and transistors for test sequence t_i must be found. This involves searching node state sets S for elements of the form $\langle s_i, t_i\rangle$. If such an element is not found, then

$$\begin{aligned}
Vdd = m_1 = m_2 = \overline{data} &= \{\langle t_0, 1\rangle\} \\
Gnd = \phi_{pc} = \phi_{in} = data &= \{\langle t_0, 0\rangle\} \\
r_2 = w_1 = w_2 = c_1 = c_2 &= \{\langle t_0, 0\rangle\} \\
bus &= \{\langle t_0, 1\rangle, \langle t_1, 0\rangle\} \\
r_1 = f_1 &= \{\langle t_0, 0\rangle, \langle t_1, 1\rangle\} \\
f_2 &= \{\langle t_0, 0\rangle, \langle t_2, 1\rangle\}
\end{aligned}$$

Figure 5. Initial Node States

the state for the reference sequence is used. To reduce search time, elements in both the state sets S and perturbation set P are kept sorted by test sequence.

As an example of this simulation technique, consider the circuit shown in Figure 2. An operation that sets node m_2 to 0 will be described. Suppose initially that nodes Vdd, m_1, m_2, bus, and $\overline{data}$ have state 1 and all other nodes have state 0. Two fault transistors are added to the network, one connecting node m_2 to Vdd whose gate is fault input f_1, the other connecting node r_1 to Vdd whose gate is f_2. For the reference sequence, both of these fault input nodes have state 0 so that the faults are absent. For test sequence t_1, f_1 has state 1 to inject fault r_1 stuck-at-one. For test sequence t_2, f_2 has state 1 to inject fault m_2 stuck-at-one. Due to the fault injected by t_1, bus and Gnd are connected by conducting transistors, hence bus is initially 0 for t_1. The representation of these initial states is shown in Figure 5.

To set m_2 to 0, nodes ϕ_{in} and w_2 must be set to 1. These changes perturb bus, $data$, and m_2, since they are connected to the source or drain of transistors that have changed state. The vicinity for each of these perturbed nodes contains bus, $data$, m_2, Vdd, and Gnd, and so their steady state responses are determined. All three storage nodes have steady states 0 due to the connection to Gnd through transistors whose gates are ϕ_{in} and $\overline{data}$. The states of Vdd and Gnd are unchanged since they are input nodes. Notice that the pull-up connection between $data$ and Vdd has no effect on the steady state of $data$ since the strength of this connection, which is γ_1, is less than the strength γ_2 pull-down connection between $data$ and Gnd.

The steady state computation just described was performed relative to the reference sequence, since node states for the reference sequence were

$$\begin{aligned}
Vdd = m_1 = \phi_{in} = w_2 = \overline{data} &= \{\langle t_0, 1\rangle\} \\
Gnd = \phi_{pc} = data &= \{\langle t_0, 0\rangle\} \\
r_2 = w_1 = c_1 = c_2 &= \{\langle t_0, 0\rangle\} \\
bus &= \{\langle t_0, 0\rangle, \langle t_2, \mathbf{X}\rangle\} \\
r_1 = f_1 &= \{\langle t_0, 0\rangle, \langle t_1, 1\rangle\} \\
m_2 = f_2 &= \{\langle t_0, 0\rangle, \langle t_2, 1\rangle\}
\end{aligned}$$

Figure 6. Final Node States

used to determine which nodes were within the vicinity as well as their steady state responses. This computation may be invalid for sequences t_1 or t_2 since bus has state 0 for t_2 and m_2 is connected to a conducting fault transistor for t_1. So that the appropriate steady state response computations will be performed for both t_1 and t_2, the perturbations $\langle bus, t_1\rangle$ and $\langle m_2, t_2\rangle$ are generated as the vicinity is found.

Consider the effects of perturbation $\langle m_2, t_2\rangle$. A vicinity containing bus, $data$, m_2, Vdd, and Gnd is found, as in the simulation for the reference sequence. The steady state response of bus depends on the strengths of the transistors whose gates are ϕ_{in} and w_2. If both of these transistors have strength γ_1, $data$ stays 0 but bus becomes **X** due to the short between Gnd and Vdd through the fault transistor connected to m_2.

Now consider the effects of the perturbation $\langle bus, t_1\rangle$. In this case, the vicinity contains node c_1, in addition to those found above. The short between bus and Gnd has no effect, and c_1, m_2, bus, and $data$ all have steady state responses equal to those in the good circuit. The representation of the final node states is shown in Figure 6.

In this example, we have seen that faults may affect the steady state response of nodes as well as which nodes are contained within a vicinity. By explicitly generating perturbations for diverged nodes and transistors when a vicinity in the good circuit is simulated, we exploit the locality of activity in each faulty circuit independent of activity in other circuits. Furthermore, this technique selectively simulates only differing portions of a faulty circuit, and hence simulation proceeds quickly.

PERFORMANCE RESULTS

As a test case for evaluating the performance of FMOSSIM, we simulated a 64 bit dynamic RAM

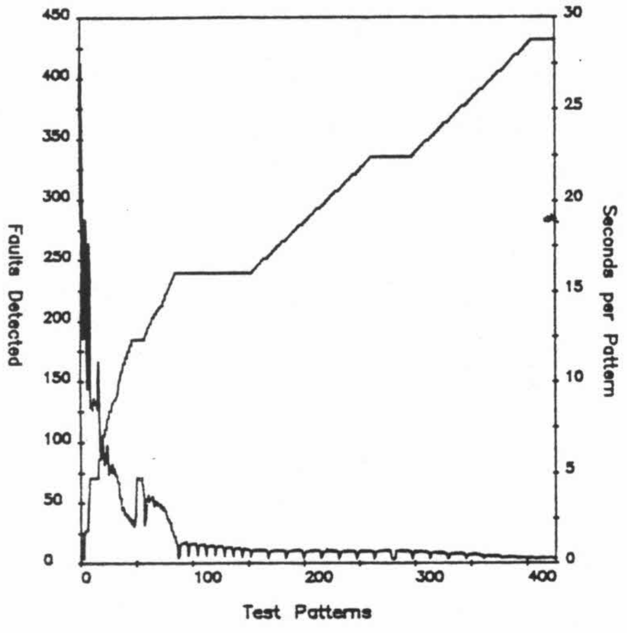

Figure 7. Performance on Memory Circuit

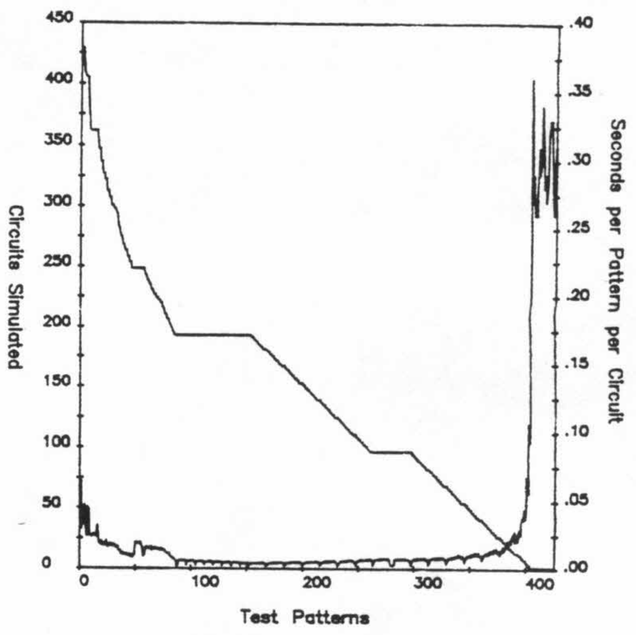

Figure 8. Effective Concurrency

circuit containing 374 transistors. This circuit incorporates a variety of MOS structures such as logic gates, bidirectional pass transistors, dynamic latches, precharged busses, and three-transistor dynamic memory elements. The circuit was simulated with 428 faults — each storage node stuck-at-zero, each storage node stuck-at-one, and pairs of adjacent busses shorted together. To validate the program, we also simulated other faults, including stuck-open and stuck-closed transistors. The simulator was implemented in the Mainsail programming language,[11] and run on a DEC-20/60.

Figure 7 illustrates the performance of FMOSSIM when simulating a test sequence consisting of a marching test[12] of the memory, together with special tests for the control logic. The curve climbing diagonally upward indicates the total number of faults detected as the test progresses. All faults were detected after 407 patterns. The falling curve indicates the CPU time required to simulate each pattern. This time starts at 27 seconds when the circuits are initialized. After 100 patterns, it drops to around 1 second as faults were detected and the simulations of these circuits were dropped. This time finally reaching 0.3 seconds at the end of the simulation, when only the good circuit is being simulated.

Figure 8 illustrates the performance advantage of concurrent simulation over simulating each faulty circuit separately. The curve falling diagonally to the right indicates the number of circuits being simulated as the test proceeds. The other curve indicates the CPU time required to simulate each pattern divided by the number of circuits being simulated for that pattern. This curve starts at about 0.05 seconds per pattern, drops to a low of 0.005 seconds once those faults causing major differences from the good circuit are dropped, and finally climbs back to 0.3 seconds when only the good circuit is being simulated. Considering that simulating a single circuit requires about *0.3* seconds per pattern, the effective benefit of simulating all of the circuits concurrently starts at 6 times serial simulation, rises to 60 times, and drops back down to 1.

Over the entire test sequence, simulating the good machine alone requires 2.5 CPU minutes. Our fault simulation requires 11 CPU minutes, whereas simulating each faulty circuit serially until it produces a different result than the good circuit would take almost 6 hours. Thus, in this case, concurrent simulation has a thirty-fold net advantage over serial simulation. Such a performance gain is clearly worth the effort.

CONCLUSION

Our experience with FMOSSIM has shown that it is a very useful tool for developing test sequences. Even when developing a test for a small section of an integrated circuit (such as an ALU or a register array), the fault simulator provides information that is hard to obtain by any other means. It quickly directs the designer to those areas of the circuit that require further tests. For example, in developing test sequences for the memory design described previously, we discovered that a simple marching test provided high coverage in the memory array itself, but that testing the control logic and peripheral circuits such as the input and output latches was more difficult.

It remains to be seen how the performance characteristics of FMOSSIM will vary as the size of the circuit and the number of faults to be simulated grows large. Even if it becomes impractical to run full-chip fault simulations with large numbers of faults, the program could still produce useful results by simulating portions of the chip, by eliminating faults that produce effects identical to other faults, or by simulating only a subset of the possible faults selected at random.

REFERENCES

[1] E. Ulrich, T. Baker, "The Concurrent Simulation of Nearly Identical Digital Networks," *Design Automation Workshop Proc.*, June 1973, pp. 145–160, and *IEEE Computer*, April 1974, pp. 39–44.

[2] R. Wadsack, "Fault Modeling and Logic Simulation of CMOS and MOS Integrated Circuits," *Bell System Technical Journal*, Vol. 57, May-June 1978, pp. 1449–1473.

[3] R. Bryant, "MOSSIM: A Switch-Level Simulator for MOS LSI," *18th Design Automation Conference Proceedings*, July 1981, pp. 786–790.

[4] R. Bryant, M. Schuster, D. Whiting, *MOSSIM II: A Switch-Level Simulator for MOS LSI, User's Manual*, Technical Report 5033, Department of Computer Science, California Institute of Technology, March 1982.

[5] J. Hayes, "A Fault Simulation Methodology for VLSI," *19th Design Automation Conference Proceedings*, July 1982, pp. 393–399.

[6] A. Bose, *et al*, "A Fault Simulator for MOS LSI Circuits," *19th Design Automation Conference Proceedings*, July 1982, pp. 400–409.

[7] M. Lightner, G. Hachtel, "Implication Algorithms for MOS Switch Level Functional Macromodeling, Implication and Testing," *19th Design Automation Conference Proceedings*, July 1982, pp. 691–698.

[8] R. Bryant, "A Switch-Level Model of MOS Logic Circuits," in J. Gray, ed., *VLSI 81*, Academic Press, August 1982, pp. 329–340.

[9] R. Bryant, *A Switch-Level Model and Simulator for MOS Digital Systems*, Technical Report 5065, Department of Computer Science, California Institute of Technology, January 1983.

[10] A. Aho, J. Hopcroft, and J. Ullman, *The Design and Analysis of Computer Algorithms*, Addison Wesley, 1974.

[11] Xidak, Inc. *Mainsail Language Manual*, Menlo Park, CA., 1982.

[12] Winegarden, S., and D. Pannell, "Paragons for Memory Test," *1981 IEEE Test Conference*, pp. 44–48.