Although this work has already appeared in IEEE Transactions on CAD, it is re-released as a Baylor technical report so that all information regarding our simulation work can appear in one place. Also, this is the first time that the source code of the Inversion Algorithm has been released in any form.

THE INVERSION ALGORITHM FOR DIGITAL SIMULATION*

Peter M. Maurer
Department of Computer Science
Baylor University
Waco, TX 76798

Abstract - **The Inversion Algorithm is an *event-driven* algorithm, whose performance rivals or exceeds that of Levelized Compiled code simulation, even at activity rates of 50% or more. The Inversion Algorithm has several unique features, the most remarkable of which is the size of the run-time code. The basic Algorithm can be implemented using no more than a page of run-time code, although in practice it is more efficient to provide several different variations of the basic algorithm. The run-time code is independent of the circuit under test, so the algorithm can be implemented either as a compiled code or an interpreted simulator with little variation in performance. Because of the small size of the run-time code, the run-time portions of the Inversion Algorithm can be implemented in assembly language for peak efficiency, and still be retargeted for new platforms with little effort.**

I. INTRODUCTION.

Of all the tools available to the modern VLSI designer, simulation is probably most important. The cost of fabricating a VLSI design is so high, that it is necessary to verify and debug the product before committing it to silicon. Despite steady improvements in simulator performance, it is not unusual for a VLSI designer to spend more time on simulation than on any other activity. There are many different styles of simulation, from high-level simulation at the algorithmic level, to electrical simulation using systems of differential equations. As a general rule, the more detailed the simulation, the more time consuming it becomes. Logic simulation represents a compromise between the extremes of algorithmic simulation and electrical simulation. Although more time consuming than algorithmic simulation, it is efficient enough to be used as a primary debugging tool. While it is not as detailed as electrical simulation, the logic gates that comprise the logic model can be mapped one-to-one into the electrical components of the final product.

Over the past several years, there has been a steady flow of papers describing new more efficient methods of logic simulation[1-14]. This research makes it obvious that

* This work was supported in part by the National Science Foundation under grant number MIP-9403414.

there are two methods for improving the performance of logic simulation: speed up the simulation of individual gates, or simulate fewer gates. Until now, these two methods have worked at cross-purposes to each other. Some simulators have used relatively complex algorithms for reducing the number of gates simulated, thereby increasing the simulation time for each gate. Other simulators have improved the speed of individual gate simulations by reducing or eliminating scheduling code, thereby increasing the number of gate simulations that must be performed for each input vector. When all scheduling code is eliminated the simulation time for each input vector becomes constant, and is no longer dependent on changes in the inputs. Such simulators are termed *Oblivious*. In contrast, simulators whose performance varies from one input vector to another are termed *Event-Driven*[2]. (This term is used for simplicity, and does not necessarily imply that the simulator processes events.)

The simplest form of oblivious simulation is Levelized Compiled Code (LCC) simulation, in which each gate in the circuit is simulated once per input vector. Many of these gate simulations produce no useful output, because they do not cause a change in any output net. (Any net visible to the user can be considered an output net, regardless of whether it is an actual output of the circuit.) Event-driven simulation is he most common method for eliminating useless simulations. In a typical event-driven simulation, an event is generated whenever a net changes value. A gate is simulated if and only if an event occurs on one of its inputs. Although this technique eliminates many useless simulations, it does not eliminate all of them. Even if the inputs of a gate change, this change may not propagate to an output net. This can occur in two ways. First, the change in the gate-input may not produce a change in the gate output. This situation is illustrated in Figure 1. Second, the change may propagate through the gate, but be "absorbed" by some other gate before reaching an output net, as illustrated in Figure 2.

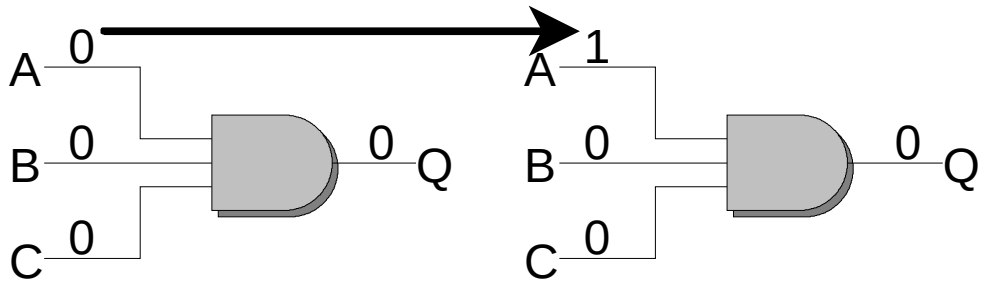


Figure 1. A Useless Simulation

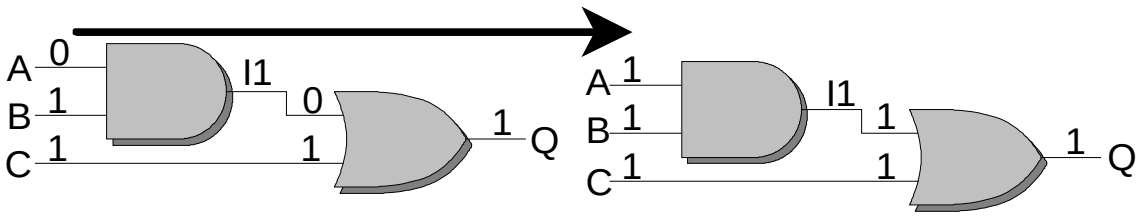


Figure 2. An Unpropagated Change.

The *Inversion Algorithm* is able to eliminate all useless simulations of the first kind, and some useless simulations of the second kind. The problem of eliminating *all* useless simulations is currently under study.

II. An Overview of The Inversion Algorithm.

Because the aim of the Inversion Algorithm is to eliminate gate simulations that produce no change in the output of a gate, it was designed around the underlying principle that no gate should be simulated unless its output is guaranteed to change value. Thus, when an event occurs on the input of a gate, it is necessary to determine whether the change will propagate through the gate, and suppress the gate-simulation if no propagation will occur. It is not immediately clear that making such a determination is any more efficient than simply simulating the gate, but it turns out that this is indeed the case.

When an event occurs on a gate input, the Inversion Algorithm performs a set of tests to determine if the event will propagate through the gate. The tests must be individualized for different gate-types, but the number of different kinds of tests that must be performed is surprisingly small. In its most basic form, the Inversion Algorithm supports the eight gate types AND, NAND, OR, NOR, XOR, XNOR, NOT, and BUFFER. These gate-types provide all essential functions and can be used as building blocks to construct more complex gate-types. However, it is not necessary to provide specific tests for each of these eight types. One set of tests is provided for NOT and BUFFER gates, a second set of tests is provided for XOR and XNOR gates, and a third set of tests is provided for AND, NAND, OR, and NOR gates.

The tests for the XOR, XNOR, NOT, and BUFFER gates are trivial, because any change in an input implies a change in the output. An identical set of tests could be used for all four gate-types, but because XOR and XNOR gates have more than one input, it is possible to propagate two or more simultaneous events through the gate. Because two consecutive simultaneous events cancel one another, the tests for XOR and XNOR have been optimized to eliminate consecutive events on the gate output. When an event propagates through the gate, the algorithm tests the queue to determine whether there is already an event queued for the net. If so, the existing event is removed from the queue, and no new event is queued. Since NOT and BUFFER gates have a single input, no test for collapsed events is necessary. The event-handlers for NOT/BUFFER gates and XOR/XNOR gates are illustrated in Figure 3.

```
NOT/BUFFER:
  Schedule OutputEvent;

XOR/XNOR:
  if OutputEvent Not Scheduled Then
      Schedule OutputEvent;
  else
      Deschedule OutputEvent;
  Endif
```

Figure 3. Event Handlers for NOT/XOR.

The tests for AND, OR, NAND and NOR are more complex and are based on the counting algorithm originally described by Schuler[16,17,18]. The counting algorithm, which is illustrated in Figure 4, was originally intended for use in a conventional event-driven simulation. In Figure 4 it is assumed that there has been a change in an input X to the gate G. The dominant value, 1 for OR/NOR, and 0 for AND/NAND, is a parameter to the algorithm.

```
If Value.of.X = Dominant.Value.of.G Then
    Count.of.G := Count.of.G + 1;
    If Count.of.G = 1 Then
        Output.of.G := Dominant.Value.of.G;
    EndIf;
Else
    Count.of.G := Count.of.G - 1;
    If Count.of.G = 0 Then
        Output.of.G := NOT Dominant.Value.of.G;
    EndIf;
EndIf;
```

Figure 4. The Original Counting Algorithm.

The counting algorithm of Figure 4 assigns a value to the output of G if and only if the output changes value. This algorithm is extremely efficient because it uses the value of a single input and an internal state to compute the value of the output, rather than computing a function using all input values. The counting algorithm used by the Inversion Algorithm is used to predict changes rather than compute output values, and is much simpler than the algorithm shown in Figure 4. One reason for the simplification is the underlying scheduling technique, which is based on the shadow algorithm[12]. In the shadow algorithm, each event is represented by the structure pictured in Figure 5.

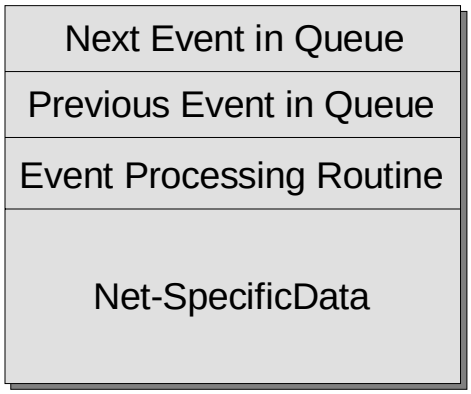


Figure 5. An Event Structure.

The Inversion Algorithm generates a dedicated event structure for each input net, each of which contains an indirect pointer to the event-processing routine for the net. The use of indirect pointers allows event handlers to be changed at run time. In the counting algorithm, the test for dominant values can be eliminated by observing that successive events on an input net will cause it to alternate between its dominant and non-dominant value. The Inversion Algorithm uses two event-handlers, one for dominant values and one for non-dominant values. When the dominant-value event-handler executes, it replaces the Event-Processing-Routine address in the event structure with the address of the non-dominant event handler. The non-dominant-value event handler performs similarly. The two event handlers execute in strict alternating fashion for each input net, with no test for dominant value required. The event-processing for AND, NAND, OR and NOR gates uses the same event-collapsing procedure as XOR and XNOR gates. The event-handlers for dominant and non-dominant values are illustrated in Figure 6.

```
Dominant:
Decrement DominantCount;
If DominantCount=0 Then
    If OutputEvent Not Scheduled Then
        Schedule OutputEvent
    Else
        Deschedule OutputEvent
    EndIf
EndIf
EventProcessingRoutine := AddressOf NonDominant;

NonDominant:
Increment DominantCount;
If DominantCount=1 Then
    If OutputEvent Not Scheduled Then
        Schedule OutputEvent
    Else
        Deschedule OutputEvent
    EndIf
EndIf
EventProcessingRoutine := AddressOf Dominant;
```

Figure 6. The AND/OR Event Processors.

The event handlers of Figure 3 and Figure 6 do not contain separate code for scheduling gate simulations. Because no gate is scheduled for simulation unless its output is guaranteed to change value, gate simulations are reduced to simple inversion operations. (This assumes that a two-valued logic model is being used. The Inversion Algorithm supports more complex logic models, but this is beyond the scope of this paper[19].) Surprisingly, because the correct operation of the Inversion Algorithm does not require net-values, it is possible to eliminate most gate simulations entirely. The event handlers pictured in Figure 3 and Figure 6 do not need to test net values to schedule new events. There are only two cases where net-values are required by the Inversion Algorithm. Net values are required for all primary inputs, because it is necessary to compare new and old net values when a new input vector is read. It is also necessary to maintain net values for any net visible to the user so that correct output values can be printed after an input vector has been simulated. Since the processing of events does not depend on net values, gate simulations can be performed during event processing without affecting the correctness of the algorithm. Thus, when the output of a gate is visible to the user, the event handlers will schedule a special event to invert the value of the output.

The elimination of net values has some surprising consequences which are worth noting. Inverted outputs and non-inverted outputs are identical. Therefore, for simulation purposes AND is identical to NAND, OR is identical to NOR, XOR is identical to XNOR, and NOT is identical to BUFFER. Because the increment and decrement operations are performed with respect to dominance rather than specific values, AND and OR gates are also identical for simulation purposes.

In a sense, the Inversion Algorithm performs *no* traditional gate evaluations, but simply processes a series of events. These events differ in one important way from the events that occur in traditional event-driven algorithms. In traditional event-driven simulation, each event corresponds to a change in a single net, while in the inversion

algorithm each event corresponds to a change in a single fanout branch of a net. Thus a single event in a traditional event-driven algorithm may correspond to several events in the Inversion Algorithm. Figure 7 illustrates why this is necessary.

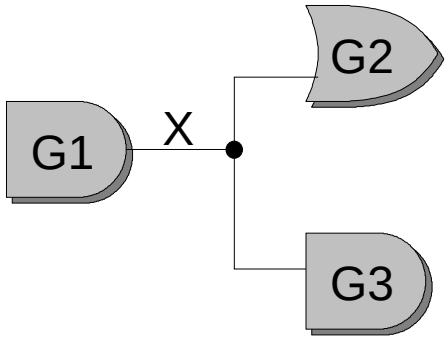


Figure 7. A Circuit Fragment.

In Figure 7, **X** is the output of gate **G1**, and the input for **G2** and **G3**. However, the dominant value for **G2** is the non-dominant value for **G3**, and vice-versa, so when an increment operation is performed for **G2**, a decrement operation must be performed for **G3**. Although both of these operations could be performed during the processing of a single event, it is more straightforward to treat them as separate events. This implementation style also facilitates the incorporation of inversion events for computing required net values. In Figure 7, if net **X** were visible to the user, the simulator would add a third event to compute the value of **X**.

Initialization for the Inversion Algorithm is somewhat more complex than for more conventional simulation algorithms. Although the dominant, non-dominant sequence is predictable for the inputs of AND, NAND, OR and NOR gates, it is necessary to commence the simulation with the correct event handler. As Figure 8 illustrates, a simple default is not sufficient to guarantee correct operation of the algorithm.

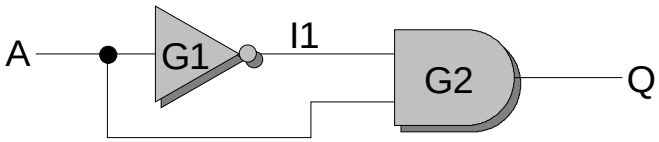


Figure 8. Complex Initialization Requirements.

In Figure 8, any time net **A** changes to the dominant value, net **I1** changes to the non-dominant value, and vice-versa. It is necessary to initialize the simulation in such a way as to guarantee that any time the non-dominant event handler is used for net A, the dominant event-handler will be used for net **I1**. The determination of this requirement cannot be made simply by examining gate **G2**. It is necessary to examine the entire circuit to determine the correct initialization state for the inputs of **G2**. To determine the correct initialization for all gate inputs, a single simulation is performed at compile time to determine consistent values for all nets in the circuit. These net values are then used to determine the initial event handler to be used for each fanout branch, and the initial

dominant count for each gate. (The three-valued Inversion Algorithm eliminates the preliminary simulation step, but this is beyond the scope of this paper.)

III. IMPLEMENTATION DETAILS.

The Inversion Algorithm consists of two major phases, the Translation Phase which prepares the circuit for simulation, and the Simulation Phase which performs the simulation. The primary function of the Translation Phase is to prepare the data structures used by the Simulation Phase. Most current implementations of the Inversion Algorithm also generate run-time code, but this code could just as easily be loaded from a library of precompiled routines.

The first step in the translation phase is to parse the circuit description, and translate it into internal data structures. Once this has been done, the circuit is levelized, the gates of the circuit are sorted into levelized order, and each gate is simulated once to generate a set of consistent values. Next, a SIMULATION fanout branch is added to each net visible to the user. Finally, a data structure known as a shadow is generated for each fanout branch of each net in the circuit.

Figure 9 illustrates the structure of a shadow. The **next** and **previous shadow** fields are used to link the shadow into the event list. The event list is doubly linked to facilitate dequeueing of events. The **subroutine** field points to the event processing routine for this fanout branch. The **first** and **last fanout branch** fields contain pointers to the first and last shadow that will be scheduled when an event propagates through the gate. All fanout branches of a net are scheduled and descheduled simultaneously. To make this process more efficient, all shadows for a net are statically linked during the translation phase. This allows all shadows for a net to be inserted into the queue or deleted from the queue as a unit.

The **lock address** field contains the address of the dominant count for the gate associated with the fanout branch. For NOT, BUFFER, XOR and XNOR gates, this field is unused. Finally, the **queue address** identifies the queue into which the shadow is to be inserted.

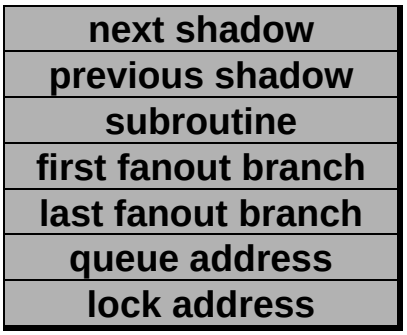


Figure 9. The Structure of a Shadow.

Eight different event processors are used during the simulation phase of the Inversion Algorithm. These occur in pairs and are called INCREMENT, INCREMENTX, DECREMENT, DECREMENTX, NOT, NOTX, XOR and XORX. The second routine of

each pair is used for shadows that are at the end of a subchain, while the second is used for the other shadows. The two subroutines of each pair are identical, except that the second routine removes the subchain from the queue. The routines were created in pairs to allow dequeueing to be performed without a conditional test. These routines are more detailed versions of the algorithms presented in Figure 3 and Figure 6.

Most existing implementations of the Inversion Algorithm use the zero-delay timing model, and are based on the LECSIM simulator developed by Wang[13]. (Unit-delay implementations of the Inversion Algorithm exist, but are beyond the scope of this paper.) LECSIM is a zero-delay event-driven levelized compiled code simulator. In LECSIM, gates are levelized and a queue is created for each level in the circuit, including the zero level. When a gate is queued for simulation, it is placed in the queue that corresponds to its level. Queues are processed in order by level. For asynchronous cyclic circuits, queues may be processed more than once.

Like LECSIM, the zero-delay Inversion Algorithm levelizes the circuit and creates one event queue per level. Each queue consists of a doubly linked list of shadows terminated by a special shadow known as the queue-trailer. The queue-trailer is responsible for advancing the simulation from one queue to the next and for terminating the simulation when appropriate. Figure 10 illustrates the structure of the queue. As Figure 10 illustrates, the queue headers are organized as an array of pointers, each of which points to a doubly linked list of shadows.

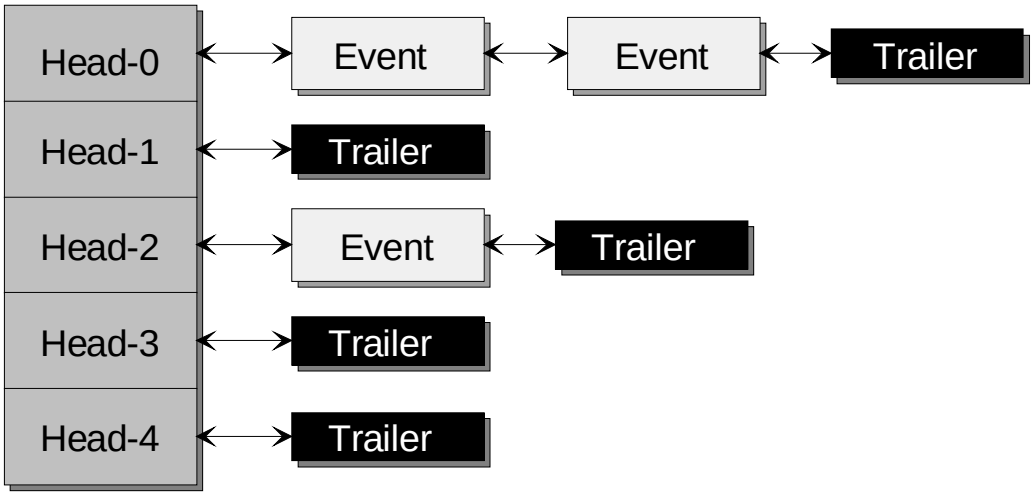


Figure 10. The Structure of the Simulation Queue.

The simulation of an input vector begins with the primary input tests. The value of each primary input is compared with the value from the previous vector (or with zero for the first vector) and if there is a change, the fanout branches of the primary input are inserted into queue zero. Once all primary input tests are complete, the simulator loads the address of the first shadow in queue zero into the **current-shadow** register and branches to the subroutine address contained in the shadow.

When an event is processed, additional shadows may be inserted into other queues. Once the last queue has been processed, simulation of the current vector terminates and a

new vector is read. Prior to reading the new vector, the value of each net visible to the user is printed. Figure 11 gives the code for the INCREMENTX routine.

```
INCREMENTX:
    Current_Shadow->subroutine = &DECREMENTX;
    (*Current_Shadow->Lock)++;
    if ((*Current_Shadow->Lock) == 1)
    {
        if (Current_Shadow->first_fanout->next == NULL)
        {
            Current_Shadow->last_fanout->next =
                Current_Shadow->Queue->next;
            Current_Shadow->Queue->next->previous =
                Current_Shadow->last_fanout;
            Current_Shadow->Queue->next =
        `       Current_Shadow->first_fanout;
            Current_Shadow->first_fanout->previous =
                Current_Shadow->Queue;
        }
        else
        {
            Current_Shadow->last_fanout->next->previous =
                Current_Shadow->first_fanout->previous;
            Current_Shadow->first_fanout->previous->next=
                Current_Shadow->last_fanout->next;
            Current_Shadow->last_fanout->next = NULL;
        }
    }
    Temp = Current_Shadow->next;
    Current_Shadow->next = NULL;
    Current_Shadow = Temp;
    Goto *Current_Shadow->subroutine;
```

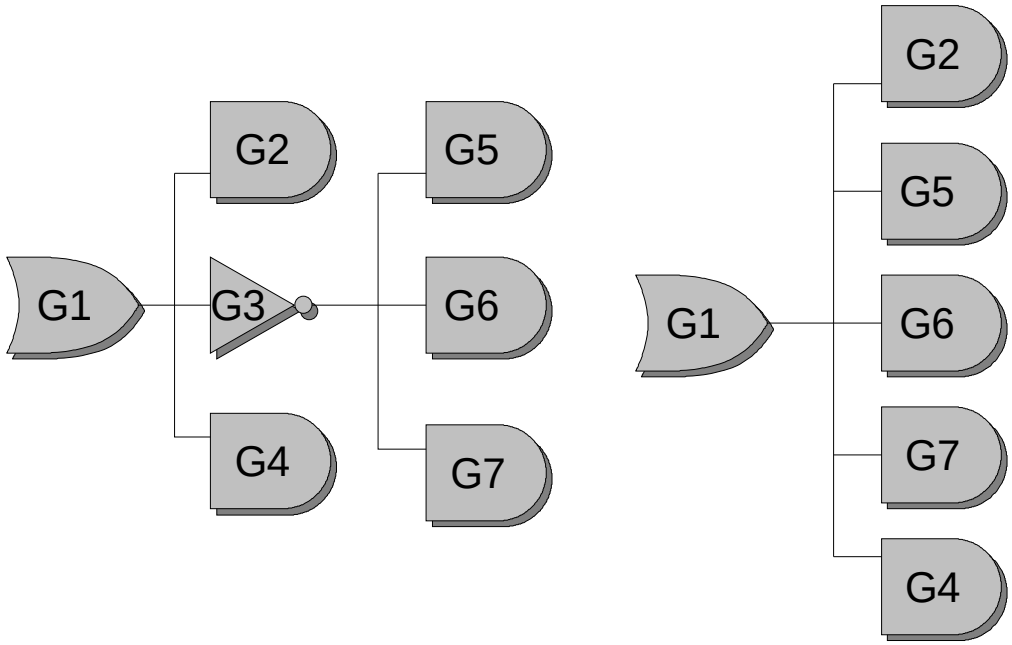
Figure 11. The INCREMENTX event handler.

IV. Optimizations of the Inversion Algorithm.

There are several simple optimizations that can significantly increase the performance of the Inversion Algorithm. The most important of these are the elimination of NOT and BUFFER gates, the elimination of XOR and XNOR gates, and the collapsing of homogeneous and heterogeneous connections.

A. The Elimination of NOT and Buffer Gates.

As the event-handler of Figure 3 illustrates, the processing of NOT and BUFFER gates is a no-op operation in the Inversion Algorithm. When the input of a NOT or a BUFFER gate is processed, the only action that is taken is scheduling the fanout branches of the output of the gate. The same simulation result can be achieved by eliminating the scheduling of NOT and BUFFER inputs and scheduling their fanout branches instead. Figure 12 illustrates this procedure. For the Unit-Delay and Multi-Delay timing models, special procedures are required to preserve the delay of the gate.



a. Before Elimination b. After Elimination

Figure 12. The elimination of a NOT gate.

B. The Elimination of XOR and XNOR gates.

It is also possible to eliminate all XOR and XNOR gates. As with NOTs and BUFFERs, the only action taken when processing the input of an XOR or XNOR is scheduling or descheduling the fanout branches of the gate. One can eliminate the processing of the input branch by scheduling or descheduling the output branches of the gate instead. This can interfere with block scheduling of fanout branches as Figure 13 illustrates.

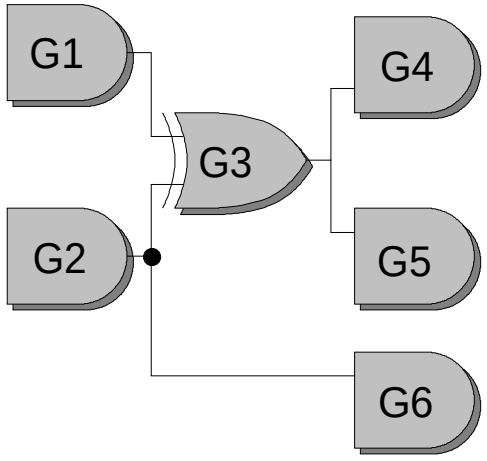


Figure 13. The Elimination of XOR Gates.

Suppose that gate **G3** of Figure 13 has been eliminated, and that events propagate through both **G1** and **G2**. Assume that the event for **G1** is processed first. When the

event for **G1** is processed, it is necessary to schedule events for the inputs of **G4** and **G5**. When the event for **G2** is processed, it is necessary to *deschedule* the events for the inputs of **G4** and **G5**, and schedule an event for the input of **G6**. If events are to be collapsed properly, the fanout branches of **G3** must maintain their identity, and cannot be grouped with the fanout branches of **G1** and **G2**. This problem arises only if one or more XOR input nets fan out to other gates. One could simply ignore event collapsing in these situations, but because of the relative rarity of XOR and XNOR gates, XOR/XNOR elimination has not been implemented in any current realization of the Inversion Algorithm.

C. Homogeneous and Heterogeneous Connections.

Once all NOT, BUFFER, XOR and XNOR gates have been eliminated from the circuit, all fanout branches other than primary inputs and outputs must be connections between AND, OR, NAND and NOR gates. These types of connections can be further categorized as *homogeneous* and *heterogeneous* connections. To distinguish between the two, suppose that net **A** is the output of **G1** and the input of **G2**, and suppose that an event on the input of **G1** propagates to **A**. This will cause the dominant counts of both gates to change. If both counts are incremented or both are decremented, then the connection is homogeneous, otherwise it is heterogeneous. Because a net may fan out to different types of gates, the heterogeneous and homogeneous properties apply to fanout branches rather than to entire nets. It is possible to categorize connections at compile time using the tables illustrated in Figure 14 and Figure 15. When using these tables, it is necessary to do the categorization *before* NOT gates are eliminated. An intervening NOT gate changes a heterogeneous connection to a homogeneous connection, and vice-versa. Two consecutive NOT gates cancel one another. A connection that passes through an XOR/XNOR gate cannot be categorized as either heterogeneous or homogeneous, because the dominant-counts of the two gates will sometimes move in the same direction and sometimes move in opposite directions.

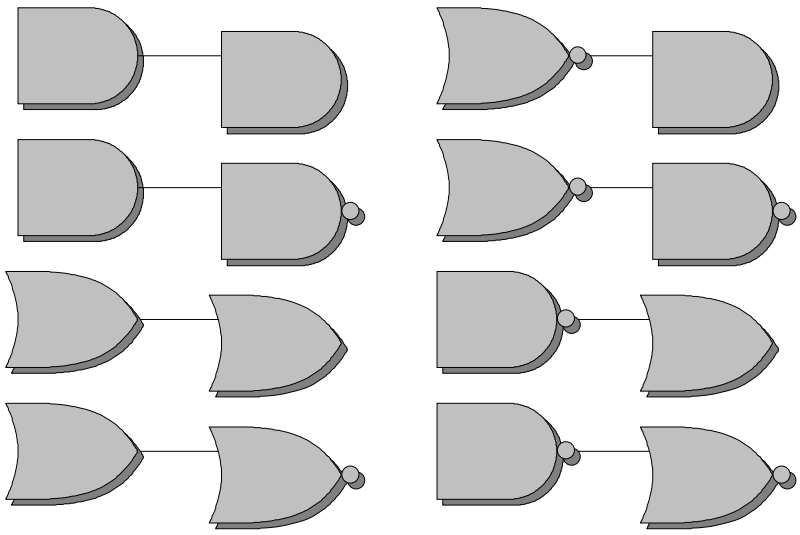


Figure 14. Homogeneous Connections.

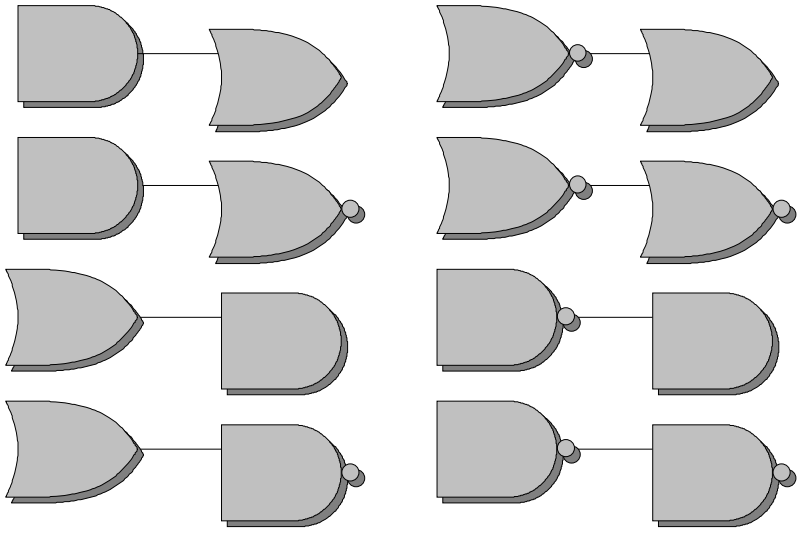


Figure 15. Heterogeneous Connections.

D. The Elimination of Homogeneous Connections.

It is possible to eliminate all homogeneous connections from a circuit using the procedure illustrated in Figure 16. To eliminate the connection **I1**, one simply removes gate **G1**, and treats the inputs **A**, **B**, and **C** as if they were inputs of **G2**. The initial value of the dominant-count for **G2** is recomputed by adding the initial dominant-counts of **G1** and **G2** and subtracting one.

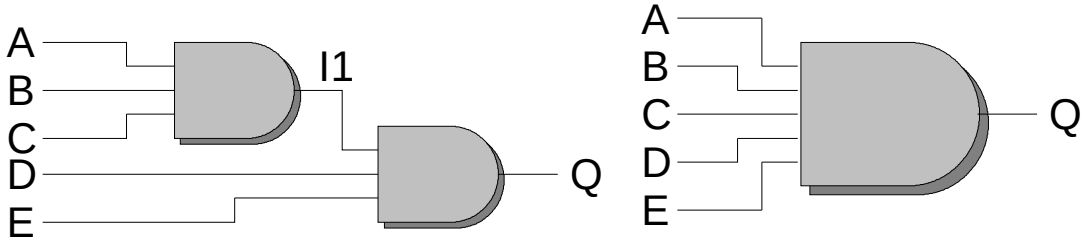


Figure 16. Eliminating a Homogeneous Connection.

The operation illustrated in Figure 16 could be performed conventional simulator, but the benefits are not as clear. After the collapse, any event on inputs **A**, **B**, and **C** would result in the simulation of a 5-input gate, regardless of whether the event would have propagated to **I1** in the uncollapsed circuit. In the Inversion Algorithm, the processing of events on the inputs **A**, **B**, and **C** is identical in both circuits, and all processing for net **I1** is eliminated in the collapsed circuit. Even if collapsing of heterogeneous connections proved to be beneficial in a conventional simulation, the ability to collapse connections is limited to those types illustrated in column 1 of Figure 14. The connections in column 2 and connections with intervening NOT gates would pose a problem.

E. Eliminating Heterogeneous Connections.

The procedures for eliminating heterogeneous connections are more complex than those for homogeneous connections, and do not always eliminate all operations for the connection. There are two procedures for eliminating heterogeneous connections, the *linear method*, and the *layered method*. The layered method allows more connections to be eliminated, but retains more operations for eliminated connections.

As Figure 17 illustrates, the linear method can eliminate *only one* input from any gate. It is possible to collapse either **G2** or **G3** into **G4**. However, once **G2** has been collapsed into **G4**, it is no longer possible to collapse **G3** into **G4**. It is, however, possible to collapse gates in linear fashion by first collapsing **G1** into **G2**, and then collapsing the **G1**/**G2** combination into **G4**.

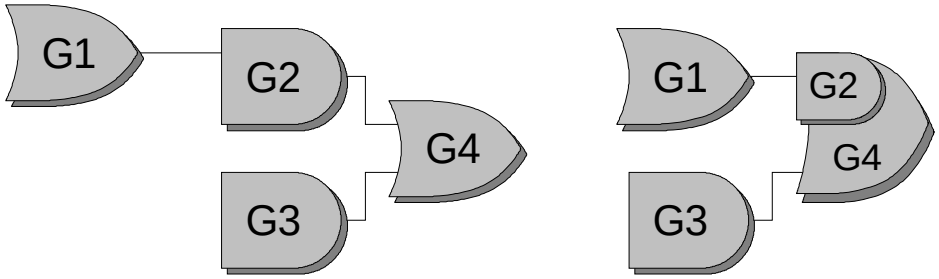


Figure 17. Eliminating a Heterogeneous Connection.

The linear method operates by changing the increment and decrement values used to update the dominant count of a gate. Instead of using a uniform value of one, the linear method uses different values for different inputs. Consider the collapsed gate illustrated in Figure 18.

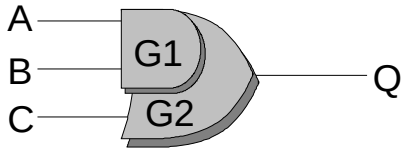


Figure 18. A Collapsed AND-OR Connection.

In Figure 18, a single dominant-count will be maintained for the combined gate **G1**/**G2**. Suppose that all three inputs, **A**, **B**, and **C**, have the value 0. The dominant-count corresponding to this input state is zero. For the output of **G1**/**G2** to change, it is necessary for **C** to change or for *both* **A** and **B** to change. The total effect of both changes in **A** and **B** must equal the effect that **G1** would have had in the original circuit, and must also equal the effect that **C** has on the collapsed gate. Neither **A** nor **B** by itself can have enough effect on the dominant-count to cause a change in the output of **G1**/**G2**. Because of the symmetry of the circuit, the effect of **A** and **B** must be the same.

There are many ways to assign increment/decrement values to the input nets **A**, **B** and **C** that will achieve these requirements. One acceptable procedure is to assign the value 1 to the input **C**, and 0.5 to the inputs **A** and **B**. The output Q changes when the dominant

count changes from a value less than 1 to a value greater than or equal to one, or when it changes from a value greater than or equal to one to a value less than one.

Collapsed connections with more than two levels follow the same principles as two-level collapsed connections. Values are assigned to inputs depending on their relative power to change the output of the collapsed gate, and there are many different assignments that will achieve the desired results. Figure 19 illustrates a three- and a four-level collapsed gate.

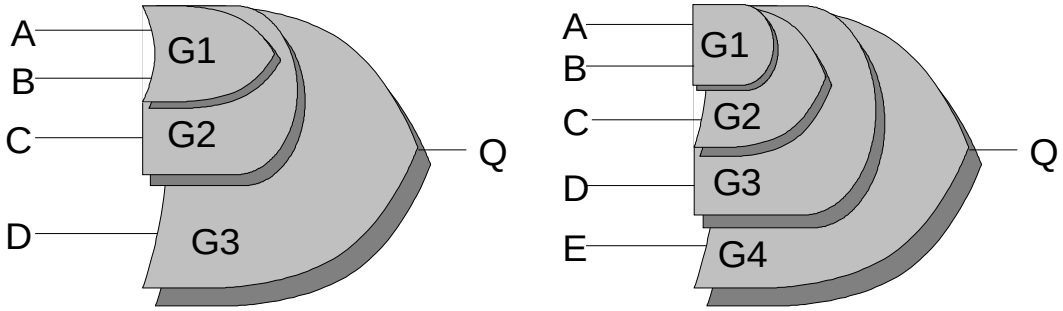


Figure 19. Multi-Level Collapsed Connections.

The values for the inputs of gate **G1**/**G2**/**G3** can be calculated in the following manner. Assume that **A**, **B**, **C**, and **D** have been initialized with the logic value of 0. The increment of **D** is set to 1. The total effect of the **G1**/**G2** combination must be equal to the effect of **D**. Since **A** and **B** are symmetric inputs, the increments assigned to **A** and **B** should be equal. Since the output of **G2** changes from 0 to 1 (thereby changing the output of **G1**/**G2**/**G3**) when **C** changes to 1 and either **A** or **B** changes to a 1, the sum of the increments assigned to **A** and **C** must equal 1. Since a change in both **A** and **B**, without an accompanying change in **C**, will not cause the output of **G1**/**G2**/**G3** to change, it is necessary that the sum of the increments assigned to **A** and **B** be less than 1. This implies that the increment assigned to **C** must be greater than the increments assigned to **A** and **B**. To achieve these requirements, an increment of .25 is assigned to both **A** and **B**, while an increment of .75 is assigned to **C**. Using similar principles, the increments assigned to the inputs of **G1**/**G2**/**G3**/**G4** are **A**->.125, **B**->.125, **C**->.25, **D**->.75, **E**->1. As with the two level collapsed gate, the output changes value only when the gate count changes from a value less than one to a value greater than or equal to one, or from a value greater than or equal to one to a value less than one.

The layered method of collapsing heterogeneous connections allows arbitrary collapsing of connections, as illustrated in Figure 20, but requires more computation in the simulation phase than the linear method.

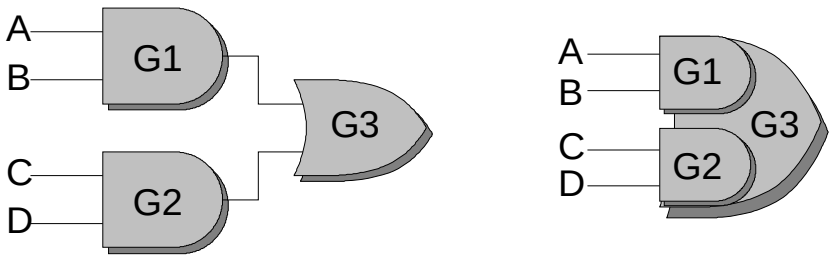


Figure 20. Layered Collapsing of Connections.

Unlike the linear method, which uses a single dominant-count for each gate, the layered method preserves the dominant-counts of the original gates. Because the original dominant counts are preserved, the number of run-time increment/decrement operations is not reduced, nor is the number of tests reduced. However the operations are performed hierarchically without any intermediate scheduling, which improves run-time performance. To illustrate assume that in Figure 20, input **A** of the collapsed gate of changes from 0 to 1. The dominant-count of **G1** is decremented, and if the new value is not zero, no further processing is done. However if the new value *is* zero, then the dominant-count of **G3** is incremented. If the new value is 1, then the fanout branches of **G3** are scheduled. No scheduling is done for the fanout branches of **G1** or **G2**.

The shadow of a layered connection differs from that illustrated in Figure 9 in that the **Lock** component of the shadow is an array of pointers rather than a single pointer. Figure 21 illustrates the code for a two-level layered connection. This code corresponds to the increment processor of a simple connection. As is the case for simple connections, both increment and decrement processors are used, the increment and decrement processors alternate with one another.

```
INCREMENT_LAYERED_2:
Current_Shadow->subroutine = &DECREMENT_LAYERED_2;
(*Current_Shadow->Lock[1])++;
if ((*Current_Shadow->Lock[1]) == 1)
{
    (*Current_Shadow->Lock[0])--;
    if ((*Current_Shadow->Lock[0]) == 0)
    {
        if (fanouts not on queue)
        {
            insert fanouts into queue;
        }
        else
        {
            remove fanouts from queue;
        }
    }
}
Current_Shadow = Current_Shadow->next;
Goto *Current_Shadow->subroutine;
```

Figure 21. The INCREMENT Routine for Two-Level Connections.

F. When to Collapse Connections.

Although it is possible to collapse all homogeneous and heterogeneous connections in a circuit, it is not always advantageous to do so. Consider the two circuits illustrated in Figure 22.

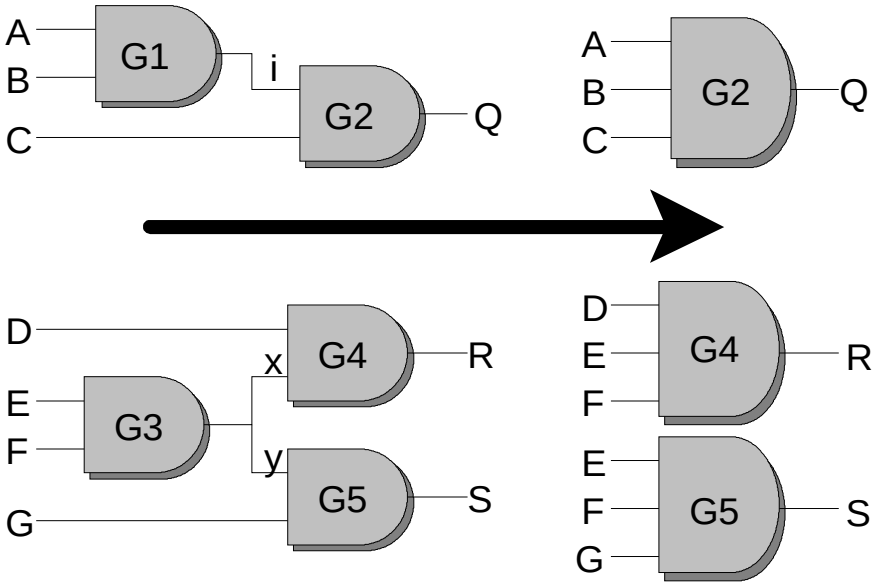


Figure 22. Collapsing Gates with Fanout.

In the first circuit of Figure 22, the net **i** does not fan out, so the connection can be collapsed by moving the inputs **A** and **B** of **G1** to **G2**. In the second circuit, the output of **G3** fans out into two branches **x** and **y**. To collapse this net it is necessary to move the two inputs **E** and **F** to both **G4** and **G5**. Because the Inversion Algorithm requires one event per fanout-branch, this doubles the number of events that must be processed when either **E** or **F** changes value. In the original circuit, a change in net **E** will cause one event to be processed, while in the collapsed circuit two events will be processed. For the first circuit, a change in net **A** will cause one event to be processed in both the original and the collapsed circuit.

It is not immediately clear that the elimination of the events on **x** and **y** won't offset the effect of doubling the inputs **E** and **F**. To further characterize the situation, the average number of events on the inputs and outputs of **G3** were calculated, assuming all input vectors to be equally likely. For the uncollapsed connection, the average number of events on **E**, **F**, **x**, and **y** is 1.75 events per input vector. After collapsing the connection, the average number of events on **E** and **F** is 2.00 events per input vector. On the average, the uncollapsed connection will be more efficient. Under the same assumptions, the average number of events on **A**, **B**, and **i** is 1.375 for the uncollapsed connection and 1.000 for the collapsed connection. In this case the collapsed connection is more efficient.

Extending this analysis to gates with larger numbers of inputs and larger fanouts, an uncollapsed 2-input AND with a fanout of 3 averages 2.125 events per vector, while the collapsed gate averages 3.00 events per vector. For a 3-input AND with a fanout of 2, the uncollapsed gate averages about 1.94 events per vector, while the collapsed gate averages 3.00 events per vector. A 3-input AND with a fanout of 3, gives even more striking results with an average of 2.16 events per vector for the uncollapsed gate and 4.50 events per vector for the collapsed connection. This analysis shows that it is advantageous to eliminate a connections only if it does not fan out to more than one gate.

V. PERFORMANCE EVALUATION.

Four prototype simulators were constructed to test the performance of the Inversion Algorithm and the various optimizations discussed in the previous section. The first prototype is unoptimized; the second eliminates NOT and BUFFER gates; the third eliminates homogeneous connections, NOT gates, and BUFFER gates; and the fourth heterogeneous connections, homogeneous connections, NOT gates, and BUFFER gates. Heterogeneous connections were eliminated using the layered method. The linear method of eliminating heterogeneous connections was not tested. All prototypes are levelized event-driven zero-delay simulators based on the LECSIM model.

The ISCAS-85 benchmarks[15] were used to certify the correctness of the prototypes. Each circuit was simulated with 5000 randomly generated input vectors, and the outputs were compared to those of the FHDL simulator[20], a levelized compiled code simulator that has been in usefor several years. These same circuits and input vectors were used to evaluate the relative performance of the four prototypes and the FHDL simulator. Each simulation was run on a SUN-4 IPC running SunOS with 12 megabytes of memory and a dedicated disk drive. This system was isolated from outside influences as much as possible during the execution of the tests. To isolate the effects of each algorithm, each simulation was done three different ways. First, a complete simulation was done with full input and output. Next the output functions of the simulators were disabled leaving all input and simulation functions intact, and a second set of simulations was run. Finally, both the simulation functions and the output functions were disabled, leaving only the input functions intact, and a third set of simulations was run. Each of these simulations was performed five times and the results were averaged to minimize errors in the UNIX /bin/time command, which was used to report the timings. The "user" field from the output of the /bin/time command was used to determine the execution time. The results of the read-only simulations were subtracted from the results of the no-print simulations to obtain the results reported in Figure 23. All simulations were run as compiled code simulations. The C language was used as the target language for all of the simulators. No optimization flags were used when compiling the simulators.

Circuit	Unopt.	NOT Elim.	Hom. Elim.	Hom/Het Elim.	LCC	Activity
c432	1.7	1.6	1.4	1.2	0.5	59.4
c499	2.0	1.9	1.9	1.9	0.6	63.2
c880	3.8	3.5	3.2	2.7	1.2	57.1
c1355	6.5	5.4	5.4	4.2	1.9	56.5
c1908	8.1	5.8	5.6	4.5	4.4	56.8
c2670	17.7	13.2	12.2	11.7	5.3	55.7
c3540	16.5	11.6	10.0	9.3	8.4	52.4
c5315	36.9	28.8	28.1	22.8	21.7	63.8
c6288	40.4	40.0	39.7	33.8	30.1	61.5
c7552	52.6	40.6	39.4	33.5	40.7	60.7

Figure 23. Experimental Results.

As Figure 23 indicates, the activity rates of the circuits tested ranged from just over 50% to over 60%. At this level of activity, Levelized Complied Code simulation (the LCC column) typically outperforms event driven simulation by a significant margin. However, for the Inversion Algorithm with deletion of homogeneous and heterogeneous connections, the timings are essentially the same for the circuits c1908, c3540, c5315, and c6288. For circuit c7552, the Inversion Algorithm actually outperforms Levelized Compiled Code simulation.

VI. CONCLUSION.

As the results of the previous section indicate, the Inversion Algorithm is competitive with Levelized Compiled Code simulation, even at very high activity rates. When the activity rate is reduced to a more reasonable level, the performance of the Inversion Algorithm will increase correspondingly while the performance of LCC simulation will remain the same. Since the activity rates reported here are significantly higher than those that are likely to be encountered in practice, the Inversion Algorithm can be expected to outperform Levelized Compiled Code simulation in most practical situations.

In addition to its performance, the Inversion Algorithm has several other desirable properties. The Inversion Algorithm can be run interpretively with only a minimal impact on performance. Therefore, it can be used for fast debugging of circuits during the initial phases of the design cycle, and for more massive testing later in the design cycle with only minor differences in performance.

Because of the small size of the run-time code, the Inversion Algorithm will be beneficial to tool developers who must support many different types of development platforms. The run-time code of the inversion algorithm can be written and debugged in a few hours, making it feasible to have several different assembly-language implementations of the same code.

The results shown here suggest that the Inversion Algorithm will be an effective tool for high-performance simulation of large circuits. In spite of this, there is a massive amount of research that must be done to realize the full potential of the techniques described in this paper. Work is currently under way to extend the Inversion Algorithm to more complex timing models, and multiple-valued logic models. There is also work in progress to identify further optimizations, and to further characterize the underlying theoretical issues. The work described here should provide the foundation for much new research in high-performance simulation of VLSI circuits.

VII. REFERENCES.

1. Bryant, R. E., D. Beatty, K. Brace, K. Cho and T. Sheffler, “COSMOS: A Compiled Simulator for MOS Circuits,” *Proceedings of the 24th Design Automation Conference*, 1987, pp. 9-16.

2. D. M. Lewis, “A Hierarchical Compiled Code Event-Driven Logic Simulator,” *IEEE Transactions on Computer Aided Design*, Vol 10, No. 6, pp.726-737, June 1991.

3. Chiang, M., and R. Palkovic, "LCC Simulators Speed Development of Synchronous Hardware," *Computer Design*, Mar. 1, 1986, pp. 87-91.

4. W. Y. Au, D. Weise, S. Seligman, "Automatic Generation of Compiled Simulations through Program Specialization," *Proceedings of the 28th Design Automation Conference*, 1991, pp. 205-210.

5. A. W. Appel, "Simulating Digital Circuits with One Bit Per Wire," *IEEE Transactions on Computer Aided Design*, Vol. CAD-7, pp. 987-993, Sept., 1988.

6. C. Hansen, "Hardware Logic Simulation by Compilation," *Proceedings of the 25th Design Automation Conference*, 1988, pp. 712-715

7. L. Wang, N. Hoover, E. Porter and J. Zasio, "SSIM: A Software Levelized Compiled-Code Simulator," *Proceedings of the 24th Design Automation Conference*, 1984, pp. 473-478.

8. Z. Barzilai, J. L. Carter, B. K. Rosen and J. D. Rutledge, "HSS -- A High Speed Simulator," *IEEE Transactions on Computer-Aided Design*, Vol. CAD-6, No. 4. July 1987, pp. 601-617.

9. P. Maurer "Two new techniques for unit-delay compiled simulation," *IEEE Transactions on Computer-Aided Design*, Vol. 11, No. 9, pp. 1120-1130, Sept. 1992.

10. Y. Lee, P. Maurer, "Two New Techniques for Compiled Multi-Delay Simulation," *Proceedings of Southeastcon 92*, Apr, 1992.

11. Y. S. Lee, P. Maurer, "Two New Techniques for Compiled Multi-Delay Logic Simulation," *Proceedings of the 29th Design Automation Conference*, 1992, pp. 420-423.

12. P. M. Maurer, "The Shadow Algorithm: A Scheduling Technique for Both Compiled and Interpreted Simulation," *IEEE Transactions on Computer Aided Design*, in press.

13. Z. Wang and P. M. Maurer, "LECSIM : A Levelized Event Driven Compiled Logic Simulator," *Proceedings of the 27th Design Automation Conference*, 1990, pp. 491-496.

14. S. P. Smith, M. R. Mercer, B. Brock, "Demand Driven Simulation: BACKSIM," *Proceedings of the 24th Design Automation Conference*, 1987, pp.181-87.

15. F. Brglez, P. Pownall, R. Hum, "Accelerated ATPG and Fault Grading via Testability Analysis," *Proceedings of the International Conference on Circuits and Systems*, 1985, pp. 695-698.

16. D. Schuler "Simulation of NAND Logic," *Proceedings of COMPCON 72*, Sept. 1972, pp. 243-245.

17. M. Breuer, A. Friedman *Diagnosis and Reliable Design of Digital Systems*, Computer Science Press, Rockville, MD, 1976.

18. M Abramovici, M. Breuer, A. Friedman, *Digital Systems Testing, and Testable Design*, Computer Science Press, New York, 1990.

19. Maurer, W. Schilp, “Three-Valued Simulation with the Inversion Algorithm,” USF Department of Computer Science & Engineering Technical Report DA-27, 1995.

20. P. Maurer, Z. Wang, C. Morency, A. Tokuta and N. Bhate, “The Florida Hardware Design Language,” *Proceedings Southeastcon-90,* pp. 430-434.