Unit Delay Simulation with the Inversion Algorithm

Peter M. Maurer, William J. Schilp
ENG 118
Department of Computer Science & Engineering
University of South Florida
Tampa, FL 33620

Abstract

The Inversion Algorithm is an event driven algorithm whose performance meets or exceeds that of Levelized Compiled Code simulation, even when the activity rate is unrealistically high. All existing implementations of the Inversion Algorithm are based on the Zero Delay model. This paper is the first step in extending the algorithm to more complex timing models. The main problems discussed in this paper are avoiding scheduling conflicts, and minimizing the amount of storage space. These problems are made considerably more difficult by the deletion of NOT gates and the collapsing of various connections. These optimizations transform the simulation into a multi-delay simulation under the transport delay model. A complete solution to the scheduling problem is presented under these conditions.
Unit Delay Simulation with the Inversion Algorithm\textsuperscript{12}

William J. Schilp Peter M. Maurer
ENG 118
Department of Computer Science & Engineering
University of South Florida
Tampa, FL 33620

1 Introduction

The Inversion Algorithm\textsuperscript{1} is an event-driven logic simulation algorithm that provides significant advantages over existing simulation techniques. Although it is event-driven, its performance is comparable to that of Levelized Compiled Code (LCC) simulation\textsuperscript{2}, even when the activity rate is as high as 60\%. At lower activity rates, the performance of the Inversion Algorithm improves, while the performance of LCC simulation remains constant. Furthermore, the amount of run-time code required by the Inversion Algorithm is only a tiny fraction of that required by other simulation algorithms, especially LCC simulation. During the course of our research, we have created several implementations of the Inversion Algorithm, all of which are based on the zero-delay timing model. However, the zero-delay timing model is not detailed enough to detect static or dynamic hazards or to do detailed timing analysis.

The zero-delay model can be enormously effective in diagnosing and fixing logical errors in the design, but during the later stages of the design process, it is necessary to do a more detailed timing analysis of the design. The simplest timing model is the unit-delay model, which assumes that each gate has a delay of one unspecified unit. Although the unit-delay model is crude, it is detailed enough to detect many types of hazards and race conditions, and is also fast enough to allow a significant amount of testing to be done in a short time.

The purpose of the current paper is to extend Inversion Algorithm to the unit-delay timing model. Supporting this model is an important first step in extending the Inversion Algorithm to various different timing models. As it turns out, certain optimizations of the Inversion Algorithm\textsuperscript{1} will require the simulator to handle delays greater than one, thus many of the problems that appear in more complex timing models must also be handled by the unit delay model.

2 Scheduling Problems

As in other event-driven algorithms, events represent changes in nets. Different methods of scheduling events lead to different timing models. There are (at least) two different scheduling techniques that are used for unit-delay simulation. \textit{Double-list} scheduling uses two queues, one for events and one for gates. The first step is to detect events on the primary inputs, and schedule them in the event queue. The event queue is then processed, and any gate that must be simulated is placed in the gate queue. When the event queue has been emptied, all gates in the gate queue are simulated, and any new events are queued in the event queue. Processing alternates between the two queues until there is nothing left to do (or until an oscillation is detected).

\textsuperscript{1} This work was supported in part by the National Science Foundation under grant MIP-9403414, and the University of South Florida Center for Microelectronics Research.

\textsuperscript{2} A preliminary version of this paper appeared in ICCAD-96.
**Single-List** scheduling uses a single event queue. When an event is processed for a net, all gates that use the net as an input are immediately simulated, and new events are added to the event queue. The simulation of the current vector stops when the event queue becomes empty. Different simulators use different methods to avoid scheduling duplicate events. There are two ways that that duplicate events could be scheduled. The first is when two simultaneous changes occur in the inputs of a gate. This problem is generally handled using a technique called *event collapsing*. The second way that duplicate events can be scheduled is when two or more events are scheduled for the same net, but for different times. Some simulators delete these events using a technique called *event cancellation*, while others process all such events. Unit-delay simulators generally process all such events.

Because the Inversion Algorithm performs no gate simulations, there is no gate evaluation step. Thus the Inversion Algorithm *must be* implemented as a single-list algorithm, which complicates the scheduling of events. The Inversion Algorithm uses dedicated event structures to achieve its high performance. This causes no difficulties in zero-delay simulation, because no event can be scheduled more than once. However, when it is possible to simultaneously schedule two or more events for the same net, the static storage management techniques of the zero-delay algorithm may fail, leading to errors in simulation. In the zero delay algorithm, a static structure is created for each fanout branch of each net, with static pointers that coordinate structures belonging to a single net. A single structure, or chain of structures, can appear in at most one location in the queue. Because dynamic storage techniques do not give the level of performance demanded by the Inversion Algorithm it is necessary to provide extra data structures to handle the double queuing of events.

**3 The Red/Green Method**

It is possible to avoid double queuing the same static structure by creating more than one set of structure per fanout branch. We will perform event collapsing to avoid queuing two simultaneous events for the same net, but we will allow multiple events to be scheduled for the same net, as long as they are scheduled to occur at different times. When an event is queued for a time slot $t$, the algorithm will search for an existing event at time $t$. If such an event exists, the two events will be combined into a single event. In two-valued simulation, this will cause both events to be dropped.

The number of structures required for each net is equal to the maximum number of events that could be simultaneously queued for any net. As Lemma 1 indicates, in unit delay scheduling, no more than two event structures will be required.

**Lemma 1.** *In Unit Delay Scheduling with event collapsing and events processed in ascending order by time, there can be no more than two events queued for any net at any time.*

**Proof.** Due to event collapsing, there can be no more than one event queued for any net in time-slot $t$. Any attempt to queue a second event for the net at time-slot $t$ will result in the two events being combined. Since the delay of every gate is at most 1, when event is processed during time-slot $t$, the latest time-slot for which events can be queued is time-slot $t+1$. Since events are processed in ascending order by time, there can be no events queued for any time-slot $s < t$. Thus, when an event $E$ is processed during time-slot $t$, there can be events queued for time slot $t$ and for time slot $t+1$, but there can be no events queued for any other time slot.

To simplify the process of scheduling events and to avoid duplicate queuing of the same static event structures, two event structures will be created for each fanout branch of each net. These two event structures are known as the red and green structures respectively. Each of these structures contains all the data required to process an event, and are chained together in alternating fashion. When a green structure is processed, any propagated events will be queued using the red data
structures, and vice versa. Input vector processing schedules only the red event structures. This alternating scheduling of event structures assigns colors to time-slots, as illustrated in Figure 1. The even-numbered time-slots, starting with zero, are red, while odd-numbered time-slots are green.

4 Eliminating Useless Event Structures

Although the Red/Green scheme can be implemented using static pointers, which require little or no run-time management, it also doubles the amount of storage required for static data structures. Fortunately, it is possible to eliminate many of these structures, because not all nets can undergo both a red and a green event. For example, consider the circuit pictured in Figure 2. When simulating this circuit, a red event will be processed for net A, a green event will be processed for net B, and so forth. Only one type of event will be processed for each net, and it is possible to eliminate the structures for the events that will never occur.

Since, in this case, the unused structures are never referenced by the simulation code; it may be possible to use a system of reference counts to eliminate them. However, there is a more effective method of doing this. The PC-set of a net contains the list of all times at which an event can occur on the net[3]. For unit-delay simulation, the PC-set will be a set of integers. If the PC-set contains an even integer, then the red event structure is required, and if it contains an odd integer, then the green event structure is required. PC sets are computed using a procedure similar to levelization. The primary inputs of a circuit are assigned a PC set of {0}. A gate becomes eligible for processing when all of its inputs have been assigned PC sets. When a gate is processed, the first step is to form the union of the PC-sets of the input nets. Each element of the union is incremented by the delay of the gate, which in this case will be one. The result is assigned to the gate as its PC-set. In most cases, the PC-set of a net is identical to the PC-set of its driving gate. For wired AND/OR connections, the PC-set of the net is the union of the PC-sets of the driving gates.

5 Elimination of Additional Structures

Since double queuing of events occurs only when events are queued in two consecutive time-slots, it is possible to eliminate one set of event structures for nets that have no consecutive PC-set elements. For example, in Figure 3, net Y has a PC-set of {1,4}. Even if two events occur on net Y, they cannot
be scheduled simultaneously. Even though the PC-set contains both odd and even elements, only a single data structure is required for net $Y$.

![Figure 3. Non-Consecutive PC-set Elements.](image)

However one must use caution when eliminating structures in this manner, as Figure 4 illustrates. In Figure 4, the circuit of Figure 3 has been inserted into a larger circuit. Even though in Figure 3 it was possible to eliminate one of the structures for net $Y$, it is not possible to do so in Figure 4. The problem arises from the use of static pointers. Because static pointers are used for scheduling, every event structure must specify the color of the propagated events. Although color is not important for net $Y$, it is important for net $Q$, which has a PC-set of $\{2,3,4,5\}$. When an event propagates from net $Y$ to net $Q$, it is necessary to schedule an event of the correct color. This cannot be done if a single, static structure is used for scheduling the events of net $Y$.

![Figure 4. Propagation Effects.](image)

To see how using a single event structure can lead to errors in simulation, assume that a single static structure is used to simulate all events on net $Y$ of Figure 4. Assume that the circuit is first simulated with input $(A=0, B=0, C=0, D=0, E=0, F=0, G=0)$, and then with $(A=1, B=1, C=1, D=0, E=0, F=0, G=0)$, output $Q$ should change from 0 to 1. The simulation is illustrated in Figure 5, assuming that the red structure for $Y$ is used for all events. Although Figure 5 shows several consecutive time-slots, these time-slots do not exist explicitly in the simulation. The alternate scheduling of red and green structures is the mechanism that creates the time slots. If these structures are queued incorrectly, the timing of the simulation will be destroyed. The sequence of events in Figure 5 should cause a dynamic hazard in the output $Q$. At time 2, $Q$ should change from 1 to 0. It should change back to 1 at time 3, and finally back to 0 at time 4. Because the green structure is used to queue events for both time slot 2 and time slot 3, these two events will be canceled and will never occur.
6 Color Analysis

To avoid the problems illustrated in Figure 4 and Figure 5, a procedure called color analysis is used to determine the number of event structures required for each net. During this process, PC-set elements are designated as red, green, or colorless. If a PC-set contains consecutive elements, the even element is set to red and the odd element is set to green. All other elements are set to colorless.

Each net will be assigned a color state, which is determined from the color of its PC-set elements. The possible color states are bicolored, monochrome, and colorless. Bicolored nets require two event structures per fanout branch, while monochrome and colorless nets require one. It is necessary to distinguish between monochrome and colorless nets to correctly propagate events. In addition to the three final color-states, there are two intermediate states, M+ and B+, which are monochrome and bicolored nets with some colorless PC-set elements. Before a final status can be assigned to M+ and B+ nets, it is necessary to assign colors to the colorless PC-set elements. This is done by identifying and properly coloring PC-set elements that could cause scheduling conflicts if left colorless. Six techniques are used: 1) Consecutive Element Coloring, 2) Color Forwarding, 3) Demand Coloring, 4) Sympathetic Coloring, 5) Minimal Coloring, and 6) Parity Coloring. The techniques are prioritized in the order given, so that the lowest number corresponds to the highest priority technique.

Consecutive Element coloring is performed once at the beginning of color analysis. The other coloring procedures are executed in priority order, until no more elements can be colored. A low-priority technique is used only if no higher priority techniques apply. For example Sympathetic Coloring is done only if no net can be colored by Demand Coloring or Color Forwarding. If a low-priority algorithm performs an action that permits the use of a higher-priority technique, the low-priority processing is immediately suspended, and the higher-priority technique is used.

Consecutive Element coloring identifies all PC-sets containing consecutive elements of the form k and k+1. Consecutive elements are colored consistently with their parity. That is to say, odd-numbered elements are colored green, and even-numbered elements are colored red. The objective of Consecutive Element Coloring is to permit consecutive events for the same net to be queued in consecutive time slots.

It is generally not possible to apply Color Forwarding until one of the other methods has been applied. Nevertheless, it is the highest priority technique, and must be used as soon as it is possible to do so. The objective of color forwarding is to propagate color information from a colored net to the primary outputs of the circuit. Color forwarding is applied when an input of a gate has a colored PC-set element k, and the corresponding element of the gate output, k+1 has not been colored. The element k+1 is colored with the opposite color of k. (If k is red, k+1 is colored green and vice-versa.) Color Forwarding propagates along all fanout branches of a net.

Demand Coloring is applied to all gates G with bicolored or B+ outputs. For each colored PC-set element, k, the corresponding elements k-1 are selected from all inputs of G. These elements are colored given the opposite color from k. Demand coloring may change a colorless net to a monochrome net or a monochrome net to a bicolored Net. When Demand Coloring colors a net with more than one fanout branch, Color Forwarding must be used to propagate the color through the other fanout branches of the net.
Sympathetic Coloring is applied to all gates with an M+ output. All PC-set elements are colored to match the existing color, making the net a monochrome net. Color Forwarding must then be used to propagate the coloring to the primary outputs. If the Color Forwarding causes elements of a B+ net to be colored, Sympathetic Coloring is immediately suspended, and Demand Coloring is invoked on the newly colored elements of the B+ net.

Minimal coloring is applied when a gate has a colorless input and a bicolor or B+ output. The lowest PC-set value of N is colored consistently with its parity, and the remainder of the PC-set elements are with the same color as the minimal element. Color Forwarding must be applied immediately after Minimal Coloring to propagate the color information to the primary outputs.

Parity coloring is used only when no other type of coloring applies. Parity coloring is applied to the PC-sets of B+ nets. The colorless elements are colored consistently with their parity, and the color is propagated to the primary outputs using Color Forwarding. Xxx illustrates some of the coloring techniques. In xxx, green PC-set elements are enclosed in a rectangle, while red PC-set elements are enclosed in an ellipse.
Color Forwarding

**Figure 6. Coloring Techniques.**

When none of the six coloring techniques can be applied, the coloring process stops. Data structures are then generated for all fanout branches of all nets. For bicolored nets, both red and green data structures are generated, while for other nets only a single data structure is generated. The structure for colorless nets always schedules the red data structure, while the structure for monochrome nets must schedule a structure of the opposite color. When the data structure for a monochrome net is generated, it is given two aliases, so it appears to be both a red and a green structure. This permits colorless nets to schedule the structure properly, even though they always schedule the red structure. It is important to note that because of Demand Coloring, a colorless net cannot schedule an event for a bicolored net.

7 Elimination of Gates and Connections

As in the zero-delay model, it is possible to eliminate certain gates and connections. However, the delays of eliminated gates must be retained to preserve timing. To see why this is so, consider the circuit of Figure 7. Under the unit-delay timing model, a 0-1 transition on input A will result in a static hazard in the output C. If the NOT gate is eliminated from this circuit without preserving the delay, the hazard will disappear.

![Figure 7. A Circuit with a Static Hazard.](image)

Preserving the delays of eliminated gates transforms the unit-delay simulation into a multi-delay simulation under the transport-delay model. This compounds the problems that led to the adoption of the red/green model. Figure 8 illustrates the effect of collapsing of gates and connections on the delays of a circuit. Once the NOT gate has been eliminated and the two remaining gates are combined, the resulting gate has three inputs A, B, and C. The delays of the original circuit must be preserved, which means that the delay between A and Q must be 3, the delay between B and Q must be 3, and the delay between C and Q must be one. Unlike a typical multi-delay simulation, delays are not associated with inputs. Instead, delays are associated with gate inputs and different inputs may have different delays. It is also possible for different fanout branches of a net to have different delays.

![Figure 8. Collapsed Gates and Connections.](image)
In the pure unit-delay model it was possible to queue all new events at the end of the event queue. In the collapsed model it is necessary to use a timing wheel similar to that used in ordinary multi-delay simulation. With such a mechanism, it becomes possible to queue many events for the same net, more than the two events that could be queued in the red/green model. Event cancellation is not possible, because delays larger than one represent the propagation of a signal through several gates of delay one. As in the red/green model, it is necessary to determine the maximum number of events that could be simultaneously queued for a single net. Lemma 2, which is the multi-delay version of Lemma 1, gives the answer to this question. The proof of Lemma 2 is essentially the same as that of Lemma 1, and is left to the reader.

**Lemma 2.** Let $G$ be a gate with output net $N$ and let $k$ be the maximum delay on any input of the gate. The maximum number of events that can be queued simultaneously for the net is $k+1$.

More important than the lemma itself is the following corollary.

**Corollary 2.1.** Let $k$ be the maximum delay value over all inputs of all gates. Then the maximum number of events that can be queued for any net is $k+1$.

Corollary 2.1 permits the extension of the red/green model to multiple colors. If $k$ is the maximum delay over all gates, then $k+1$ colors, designated 0, 1, 2, ..., $k$, are required. Scheduling conflicts can be avoided by generating $k+1$ event structures for each net and coloring them with the colors 0 through $k$.

## 8 Color Analysis for Collapsed Networks

Although generating $k$ structures for each fanout-branch will prevent scheduling conflicts, many of these structures will not be used. As in the Red/Green model, the first step in eliminating the unused structures computing the PC-sets of the collapsed network. However, because delays are associated with the inputs of a gate rather than with the gate itself, it is necessary to add the delay of the input to each PC-set element before forming the union of the PC-sets. (For uncollapsed networks, the delay is added after forming the union.)

In the Red/Green model, each PC-Set element was colored according to its parity. The multi-colored analog of this procedure is to categorize PC-set elements using the modulo $k+1$ function, where $k$ is the maximum color value. If the PC-set for a net $N$ has no elements of category $c$, then a structure of color $c$ is not required. In a more detailed analysis, the maximum number of colors needed by a net is equal to the maximum number of events that may be simultaneously queued for the net. The PC-set gives information about when events will be processed, but no information about when events will be queued. To determine the maximum number of events that may be queued at any one time, it is necessary to compute the Queue Density Function $D_N(i)$. For each net $N$, $D_N(i)$ gives the maximum number of events that can be queued at time $i$. The domain of this function is the set 0-$m$, where $m$ is the level number of the net in question. (Note that level number is equal to the maximum value of the PC-set.) The function can be extended to all integers by observing that after time $m$, there will be no events queued for any level-$m$ net. Thus, $D_N(i)=0$ whenever $i>m$.

The first step in computing $D_N(i)$ is to compute a modified form of the PC-set for each net in the circuit. Each element of the new PC-set is a pair of numbers giving the level at which the net will be queued and the delay of the net. Let $G$ be a gate with $n$ inputs. If the largest delay on any of the $n$ inputs is $k$, then a queue $Q$ with $k+1$ elements is created. The queue consists of a collection of Boolean values, which indicate whether the corresponding time slot is empty or full. The queue density function is computed by Algorithm 1.

Algorithm 1 can also be used to compute the Queue Population Function $P_N(i)$, which will be useful in assigning colors to slots. The function $P_N(i)$ is similar to $D_N(i)$, but $P_N(i)$ returns the set of filled queue positions along with pointers to the enhanced PC-set elements responsible for filling those queue positions.
Initialize Queue Positions 0-n to Empty

\[ QPos := 0; \]

/* Process elements of Modified-PC */

For each \((x,d)\) in Modified-PC, in ascending order by \(x\) do

While \(QPos < x\) do

\[ D_N(QPos) := \text{No. of full positions in } Q; \]

\[ \text{Pop } Q; \text{ Push empty onto tail of } Q; \]

\[ QPos := QPos + 1; \]

EndWhile

\[ Q[d] := \text{full}; \]

EndFor

While \(Q\) not empty do

\[ D_N(QPos) := \text{No. of full positions in } Q; \]

\[ \text{Pop } Q; \]

\[ QPos := QPos + 1; \]

EndWhile

Algorithm 1. Queue Density Function Computation.

Once all queue density functions have been computed, the initial computation of the maximum number of colors required by each net is done. Let \(N\) be a net with queue density function \(f\). Assume further that the level of net \(N\) is \(m\). Let \(c\) be the maximum of \(f(x): 0 \leq x \leq m\). The events of \(N\) can be scheduled without conflict using no more than \(c\) colors. This is a preliminary calculation because \(c\) colors may not be enough to prevent scheduling conflicts in successor gates. Even if \(C\) is the maximum value over all queue density functions, it may be necessary to use more than \(C\) colors to schedule the entire circuit without conflict. Two types of conflicts arise when combining gates into networks: parent-child conflicts and sibling conflicts. The circuit fragment pictured in Figure 9 illustrates a parent-child conflict. In this figure, the numbers in curly braces are the PC-sets of the corresponding nets, while the numbers indicated by “d=” are the delays associated with the inputs. (In a real circuit, the unrealistically wide NAND gate would be replaced by a much more complicated network.)

Figure 9. Scheduling Conflicts.

Figure 10 illustrates the computation of the queue density functions for the outputs of the gates of Figure 9. The maximum of these queue density functions is 2. However to avoid scheduling conflicts, it is necessary to use 3 colors to schedule the events of the first gate. For the first gate, events at times 10, 14 and 18 must be Red, while events at times 12 and 16 must be green. For the second gate, events
at time 19 must be Red, while events at times 11, 13, 15, and 17 must be green. When the gates are combined, events from the first gate at times 14 and 18 must have different colors, because they schedule events of different colors. To resolve this problem, three colors must be used for the events of the first gate.

Figure 10. Two Queue Density Functions.

To avoid parent-child conflicts, it is necessary to propagate color information from the output of a gate to its inputs, but this may give rise to sibling conflicts. Sibling conflicts arise between two or more fanout branches of a single net. The Inversion Algorithm schedules events using chains of data structures linked with static pointers. Each data structure represents one fanout branch of a net. When different collections of data structures must be scheduled by different events, there must be a static chain for each different situation. Figure 11 illustrates how sibling conflicts occur.

Figure 11. Sibling Conflicts.
In Figure 11, it is assumed that colors have been assigned to the outputs of G2 and G3, that color information has been propagated from the outputs if G2 and G3 to their inputs. The PC-set of the net is assumed to be \{A,B,C,D,E\}. The indicators R and G indicate the color of the propagated events for each fanout branch at each time slot A through E. At time A, a green event must be scheduled for the output of G2 and a red event must be scheduled for the output of G3. Each time slot gives rise to a different combination of colors for the propagated events. Writing the combinations as a sequence of ordered pairs, the list is (G,R), (G,G), (G,R), (G,G), and (R,R). There are three distinct pairs, (G,G), (G,R), and (R,R). Because static pointers are used, it is necessary to create three distinct chains of data structures. The data structures are illustrated in Figure 12.

![Figure 12. Sibling Conflict Resolution.](image)

We have developed a colorizing procedure which can color the time-slots of all nets to avoid all scheduling conflicts, as well as parent-child conflicts and sibling conflicts. Before applying this coloring function, it is necessary to break any cycles and levelize the resultant acyclic network. For synchronous sequential circuits, this can be done in straightforward way by breaking each synchronous flip-flop. For asynchronous sequential circuits, a method such as the convergence algorithm outlined in [4] can be used. Colorizing starts with all fanout-free primary outputs of the network. The queue-population function is used, as illustrated in Algorithm 2, to color the PC-set elements of these nets.

\[
\textbf{Algorithm 2. Assign Colors to Primary Outputs.}\\
\]

Once all primary outputs have been colored, the main coloring algorithm, Algorithm 3, is used to color the remainder of the network. This algorithm is based on two queues, a queue of gates, \(GQ\), whose input nets require coloring, and a queue of nets, \(NQ\), whose fanout branches must be combined to resolve sibling conflicts. In Algorithm 3, the modified PC-sets of the fanout branches of a net are considered to be distinct from the modified PC-set of the net itself. The \textbf{Propagate} function
propagates colors to fanout branches, the **Combine** function propagates colors from the fanout branches to the net itself, and the **Recolor** function refines the coloring generated by the **Combine** function. The details of these functions are presented below.

For Each gate $G$ whose output is a fanout-free Primary Output do
   Add $G$ to $GQ$;
EndFor

While $GQ$ is not empty do
   For Each gate $G$ with output $N$ in $GQ$ do
      Propagate the coloring of $N$ to each input $I$ of $G$;
      Delete $G$ from $GQ$;
      If all fanout branches of the input $I$ have been colored Then
         Add $I$ to $NQ$;
      EndIf
   EndFor;

   For Each Net $N$ in $NQ$ do
      Combine the color information of the fanout branches of $N$;
      Recolor slots based on the Queue Population Function of $N$;
      Delete $N$ from $NQ$;
      If $N$ is the output of a gate $G$ Then
         Add $G$ to $GQ$;
      EndIf
   EndFor
EndWhile

**Algorithm 3. The Main Coloring Algorithm.**

Algorithm 4 illustrates the **Propagate** function used by the main coloring algorithm. This algorithm is used to propagate color information from one fanout branch of a net to another. No conflicts can arise during this process.

**Propagate($N$:OutputNet,$G$:Gate)**
begin
   For Each input $I$ of $G$ do
      $d := \text{DelayOf}(I)$;
   EndFor
   For Each time-slot $t$ in the PC Set of $N$ do
      $c := \text{ColorOf}(t)$;
      If the PC Set of $I$ contains the element $t-d$ Then
         Color the element $t-d$ with color $c$, in the PC Set of the fanout branch of $I$ leading to $G$;
      EndIf
   EndFor
EndPropagate;

**Algorithm 4. The Propagate function.**

Algorithm 5 illustrates the **Combine** function used to prevent sibling conflicts. This algorithm also shows how forward scheduling information is obtained for each data structure.
The Combine function not only performs an initial color assignment, it also creates a vital piece of scheduling data, the $k$-tuple of colors associated with each time-slot. Effectively, this algorithm assigns a two-dimensional color to each time slot. The first component is the color created by the Assign statement, while the second component is the $k$-tuple of propagated colors. Both the color and the $k$-tuple will be used to create and link the final data structures.

Combine($N$:Net)
Var $S$:Set of Tuples, $k$:Integer;
begin
  $S :=$ The Empty Set;
  $k :=$ The number of Fanout branches of $N$;
  For Each element $t$ in the PC Set of $N$ do
    If the $k$-tuple of propagated colors for time slot $t$
    is not already contained in $S$ Then
      Add the $k$-tuple of propagated colors to $S$;
    Endif;
  EndFor;
  Assign the colors 0 through $|S|-1$ to the elements of $S$,
  and to the corresponding time-slots;
quirent negate
  Retain information for the Data-Structure Generator */
  Retain the set $S$, and links between the elements of $S$ and
  time slots;
End Combine;

Algorithm 5. The Combine Function.

Finally, the Recolor function is illustrated in. This function is used to assign the final colors to the time-slots of a net. The association between time-slots and $k$-tuples is maintained throughout the recoloring process. The mechanism for doing this does not appear explicitly in the Algorithm 6.
Recolor($N$:Net)
Var $K$:k-tuple, $c$:color;
begin
Repeat
For Each element $t$ in the PC Set of $N$ do
For Each element $s$ in $P_N(t)$ do
If $s$ has the same color as a previous element of $P_N(t)$ Then
$K :=$ the $k$-tuple associated with $t$;
If there is a time-slot $u$ of color $c$,
and $c$ does not appear in $P_N(t)$,
and $c$ is greater than the current color of $s$ Then
Assign $c$ to $s$;
Else
$c :=$ The smallest color not used to color any element of
the PC-Set of $N$;
Assign $c$ to $s$;
EndIf
EndIf
EndFor
EndFor
Until No Slots are Recolored;
End Recolor;

Algorithm 6. The Recolor Function.

Note that when a slot is recolored by the Recolor function, the new color will always be numerically larger than the existing color. This eliminates any circularity problem that may occur should it be necessary to recolor a time-slot more than once. Since it is possible to recolor events more than once, it is possible that multiple recolorings of several nets will leave the net in a state where scheduling conflicts are still possible. Hence it is necessary to repeat the recoloring process until no more nets can be recolored.

Once the main coloring algorithm has completed, it is possible to create scheduling data-structures for each of the nets. First, a set of data structures is created for each net. Suppose a net $N$ has fanout $k$ and that $c$ colors have been used to color the time-slots of the net. In this case $c$ chains of data structures will be created, each one of which has $k$ elements. The forward scheduling information for each data structure will be taken from the $k$-tuple associated with the color of the data structure. Although, strictly speaking, $k$-tuples are associated with slots rather than colors, the process of assigning colors to slots guarantees that if two slots have the same color, then they are associated with the same $k$-tuple.

The scheduling and simulation code is quite simple, and essentially identical to that used by the zero delay algorithm. For the sake of completeness, this code is replicated in Figure 13. In this code, the scheduling data structures are referred to as $Shadows[5]$. This code is written in C, the Inversion Algorithm implementation language. The computed goto at the end is actually implemented in assembly language.
INCREMENTX:
/* Alternate the INC & DEC processors */
*Current_Shadow->subroutine = &DECREMENTX;
(*Current_Shadow->Lock)++;
/* If a change in the output will occur */
if (*Current_Shadow->Lock) == 1)
{
    /* If the gate is not already queued */
    if (Current_Shadow->last_fanout->next ==
        Current_Shadow->last_fanout)
    {
        /* queue the gate for simulation */
        if (Queue_Tail != NULL)
        {
            Queue_Tail->next =
                Current_Shadow->first_fanout;
            Current_Shadow->first_fanout->previous =
                Queue_Tail;
        }
        else
        {
            Queue_Head =
                Current_Shadow->first_fanout;
            Current_Shadow->first_fanout->previous =
                Queue_Tail;
            Queue_Tail = Current_Shadow->last_fanout;
            Current_Shadow->last_fanout->next = NULL;
        }
    }
    else
    {
        /* dequeue the gate */
        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;
        Current_Shadow->first_fanout->previous =
            Current_Shadow->first_fanout;
    }
}
Temp = Current_Shadow->next;
Current_Shadow->next = Current_Shadow;
Current_Shadow = Temp;
if (Current_Shadow == NULL) return;
Goto **Current_Shadow->subroutine;

**Figure 13. Inversion Algorithm Code.**

As noted in reference [1] the run-time code for the Inversion Algorithm consists of less than ten
slightly different versions of the routine pictured in Figure 13.

9 Experimental Data

We have implemented the unit-delay algorithm. We have compared this algorithm to several
others [4,5] using the ISCAS 85 combinational benchmarks[6]. Experiments were all run on the same
dedicated machine, a SUN IPC workstation with 12 megabytes of RAM and a dedicated disk drive.
The results of these experiments are reported in Figure 14. Also included is the percentage of data
structures eliminated by color analysis.
<table>
<thead>
<tr>
<th>Ckt</th>
<th>Interp.</th>
<th>Gateway</th>
<th>C-Shad</th>
<th>Unit-Delay</th>
<th>Savings</th>
</tr>
</thead>
<tbody>
<tr>
<td>c432</td>
<td>23.4</td>
<td>3.6</td>
<td>3.9</td>
<td>3.1</td>
<td>31.5</td>
</tr>
<tr>
<td>c499</td>
<td>26.1</td>
<td>4.2</td>
<td>4.5</td>
<td>3.0</td>
<td>74.5</td>
</tr>
<tr>
<td>c880</td>
<td>46.3</td>
<td>14.0</td>
<td>8.3</td>
<td>7.6</td>
<td>49.0</td>
</tr>
<tr>
<td>c1355</td>
<td>93.8</td>
<td>30.9</td>
<td>18.0</td>
<td>13.1</td>
<td>26.3</td>
</tr>
<tr>
<td>c1908</td>
<td>172.9</td>
<td>61.1</td>
<td>32.9</td>
<td>26.4</td>
<td>35.1</td>
</tr>
<tr>
<td>c2670</td>
<td>192.1</td>
<td>81.1</td>
<td>43.8</td>
<td>39.1</td>
<td>38.3</td>
</tr>
<tr>
<td>c3540</td>
<td>277.1</td>
<td>112.7</td>
<td>63.9</td>
<td>48.6</td>
<td>29.6</td>
</tr>
<tr>
<td>c5315</td>
<td>519.1</td>
<td>228.3</td>
<td>126.9</td>
<td>97.6</td>
<td>38.1</td>
</tr>
<tr>
<td>c6288</td>
<td>5108.6</td>
<td>2602.5</td>
<td>1245.6</td>
<td>1348.8</td>
<td>22.4</td>
</tr>
<tr>
<td>c7552</td>
<td>795.1</td>
<td>372.7</td>
<td>201.1</td>
<td>155.0</td>
<td>25.1</td>
</tr>
</tbody>
</table>

Figure 14. Experimental Data

10 Conclusion

This paper shows that the Inversion Algorithm, which has been proven to be effective for zero-delay simulation, is a powerful technique that can be extended to more complex timing models. The two most significant problems solved in this paper, are preserving the ability to eliminate gates and connections as in the zero delay model, and reducing the number of data structures that are needed to prevent scheduling errors. This paper has also shown, in a preliminary way, how the more complex multi-delay model could be used with the Inversion Algorithm. Unlike the model presented here, a true multi-delay model must support several types of delay including both transport and inertial delay. The current paper discusses only the transport delay model. Support for the inertial delay model, with event canceling, should be quite different.

This paper also serves to demonstrate the versatility and adaptability of the Inversion Algorithm. It must be noted that the run-time code for the Unit-Delay model is virtually identical to that of the Zero-Delay model. Due to the extremely small size of the code, the Unit-Delay model should be just as cache-efficient, and just as adaptable to new hardware as the zero-delay model. This paper demonstrates again that the Inversion Algorithm is a widely applicable technique that will prove to be an effective design automation tool in the future.
References


