Keywords: Discrete Simulation, Logic Simulation, Multi-Delay Timing, Differential Simulation.

Abstract

Here we show that the Event Driven Condition Free (EVCF) simulation technique for gate-level circuits can be extended to the multi-delay timing model. In the multi-delay timing model, gates have different integer delays. Events are produced out of order and must be sorted for processing. The EVCF technique has shown spectacular gains in performance for the zero-delay and unit-delay models. The gains here are somewhat less spectacular, but are substantial, showing that the EVCF technique is an effective method for improving the performance of multi delay simulation.

1 INTRODUCTION

One of the most important tools in the VLSI designer's arsenal is simulation. Simulation is used in a number of ways, the most important of which is determining the correctness of a design before it is submitted to manufacturing. Event driven simulation algorithms [1] are popular because they avoid unnecessary gate simulations, thus reducing the overall cost of simulation. A number of different timing models have been used with event driven simulation, including the zero-delay model which treats gates as pure functions [2,3], and the unit-delay model which assigns a delay of one time unit to each gate [4-7]. More accurate timing models attempt to provide better timing accuracy by doing things like using different delays for different gates, providing separate delays for the 1/0 and 0/1 transitions of a gate output, and a number of other things [8].

Regardless of the timing model used, one constant is the need for higher and higher simulation speeds. Event driven simulation addresses this problem by eliminating unnecessary gate simulations. Other approaches address the problem by improving gate simulation performance. One successful technique is differential simulation [3,9] which treats both gates and nets as state machines. Instead of computing gate values, differential simulators determine whether events will propagate by performing state manipulations when events are processed. Differential simulation generally does not need to compute output values for gates. The only exception is when the output of a gate must be visible outside of the simulation.

An interesting variation on differential simulation is the metamorphic approach [3]. In this approach, gates and nets are objects whose identities change during the course of the simulation. We call this approach the Event Driven Condition Free (EVCF) approach, because the code, for the most part, contains no loops or conditional statements. (The zero-delay model [3] contains absolutely no conditionals, but more complex timing models [7] become too cumbersome without one or two conditional statements.)

In this paper we extend the EVCF technique to a new timing model, the multi-delay model. This model is similar to the unit-delay model, with gates having a variety of integer-valued delays. Up until this time, it was not clear whether the EVCF technique would be effective with this sort of a model, because as timing models become more complex, scheduling time begins to dominate gate simulation time. Because the EVCF technique is primarily aimed at reducing the cost of gate simulation, there was some concern that it might provide only negligible performance improvements. As we will show, the EVCF algorithm is extremely effective for the multi-delay model.

2 THE DELAY MODEL

In the multi-delay model different gates have different delays, but the delays are all integers. Specific time units can be used, but more often than not, the units of the delay are left unspecified, resulting in a relative timing model. It is possible to synchronize the reading of input vectors with the simulation-clock, however it is common to assume that the interval between input vectors is long enough for all gates to settle. (Violations of this assumption are easy to detect, regardless of which method is used.)

Our model is a relative delay model in which each gate has a small (<100) integer delay. Instead of synchronizing input vectors with the clock, we use a relative clock which starts with zero for each input vector. Our simulation model is event driven, where an event is a change in the value of a net.

Every multi-delay simulator must have an event sorting mechanism. Consider the circuit of Figure 1. If inputs A, B, and C change to 1 simultaneously, gates X, W, and V will be simulated at time 0. However, the change in net G must not arrive at gate V before time 4, even though the new value is available immediately. Events will be produced on nets D, E, and F, all of which must be processed before the event on net G. A sorting mechanism must be used to force the events on E and F to be processed before the event on G.

The two primary event sorting mechanisms are timing wheels and priority queues. A timing wheel is essentially an implementation of the bucket sort algorithm which requires O(n) time to sort n events. The priority queue mechanism is an implementation of the heap sort algorithm which requires O(n log n) time to sort n events. Obviously the timing wheel is preferable. Priority queues are used only when it is impossible to use a timing wheel. (An example is when delays are specified as floating point numbers.) We chose to use a timing wheel in our model.

Another issue that must be resolved is the handling of multiple events on the same net. In Figure 1 when all three inputs change to 1, events will be produced on net F at time 3, and on net G at time 4. If we process both events through gate V, net Q will change to 1 at time 5 and back to 0 at time 6, giving us a static hazard of length 1.

If these were physical gates, we would not observe such a pulse on the output of gate V. What we would see is an intermediate pulse that achieves a peak somewhere between high and low. There are two methods of handling
such pulses. The first is called “transport delay” in which we allow all pulses to propagate, regardless of length. The second is called “inertial delay” in which gates filter out all pulses larger than their delay.

We chose to use the transport delay model for two reasons. First, we our own conventional multi-delay simulator that we wanted to use for comparison purposes. This simulator uses the transport delay model. If we used the inertial delay model for the EVCF simulator the two simulators would not be directly comparable. The second reason is that we consider the transport delay model to be more challenging than the inertial delay model. The zero and unit delay models of the EVCF approach both use static data structures for scheduling. This is possible because no more than one event is ever scheduled for the same net. By the same token, the inertial delay model does not permit more than one event to be scheduled for the same net, permitting a direct adaptation of our earlier scheduling mechanisms. We felt that we needed to move beyond these models into a model that permits multiple events to be scheduled for the same net. The transport delay model serves this purpose.

Figure 1. Multiple Delays.

3 THE TIMING WHEEL

For the timing wheel implementation, we chose to imitate the model used in our conventional multi-delay simulator. The timing wheel consists of an array of queues, one for each unit of simulated time. When the simulator reaches time n, it processes all events in queue number n and queues any gates that use these nets as inputs. Then it empties queue number n, and simulates all queued gates. The gate simulations will (possibly) generate events in other queues, but during the gate simulations, all queues numbered n or less will be empty, and will remain empty.

This mechanism implies that the physical queues can be reused. Although we require one logical queue for each unit of simulated time, it is sufficient to provide only k queues, where k is the maximum delay of any gate in the circuit. The physical queues are used in a circular manner, giving rise to the term “Timing Wheel.” The events for simulated time n will be placed in physical queue n%k, where % represents the remainder operation.

Each queue in the timing wheel is implemented as an array. In our conventional simulator, the array elements are structures containing the new value of the net and a pointer to a data structure describing the net. Each queue has a current-index variable which points to the first unused slot in the array. The current-index variable is incremented each time an event is added to the queue.

We replicated this mechanism in the EVCF multi-delay implementation. Because the EVCF technique is a differential simulation scheme that does not use net values, we could eliminate the net values from the timing wheel queues. But since the EVCF technique is a metamorphic technique without loops or conditionals, it was necessary to add a queue-trailer event to the end of each queue. This queue-trailer event is used to perform the operations that are normally performed after processing all events in a queue.

Another complication was caused by the fact that in the EVCF technique a net is represented by a sequence of events, not just a single event. Since each event performs a differential computation to determine whether events will propagate through a gate, one event is needed for each fanout branch of a net. If a net is a primary output, an additional fanout branch is needed to compute the output value of the net. This means that each pointer in the queue must point to a linked list of events, not just a single event. Again, because of the the metamorphic nature of the EVCF technique, it is necessary to add an additional event to the end of each list to terminate the processing of the list and advance to the next list in the queue.

4 GATE AND NET STATES

The EVCF technique operates in differential mode and retains a state for each net and each gate in the system. (In some cases the state never changes.) We use a compiled simulation technique that transforms a circuit into a C module that must then be compiled and executed to simulate the circuit.

The data structure of Figure 2 is used to represent a net. The Next pointer is used to create a linked list of events. Each net is represented by a linked list of Nshadow structures. The Rtn pointer points to a segment of executable code that is used to process the event. This pointer is used to maintain the state of the net. We will discuss the potential values of the Rtn pointer in the next section. Each Nshadow structure is associated with a single gate. (The net serves as an input to this gate.) The Gate pointer points to the associated gate.

```
struct Nshadow
{
  struct Nshadow *Next;
  long *Rtn;
  struct Gshadow *Gate;
}
```

Figure 2. The Net Data Structure.

The structure of Figure 3 is used to represent a gate. The Next and Prev pointers are used to queue the gate. A doubly linked list is used to simplify the dequeuing of gates. The Rtn pointer, which is constant, is used to queue the event list associated with the gate, using the Delay element to locate the correct queue. The event list is given using the Begin and End pointers. Each gate is associated with a single event list. The Up, Down and Schedule pointers point to code segments. The Up and Down code segments are executed by the event processing routine of the Nshadow
structure. In most cases, the Schedule code segment is executed by the Up and Down code segments.

The state of the gate is maintained using the Up and Down pointers. These pointers determine the number of dominant inputs of a gate. When the number transitions either from or to zero, an event is scheduled on the output of the gate. Some gates have no internal state and propagate all input events. The code for all gates is given in the next section.

Each gate also has a second state which determines whether the gate is currently on the gate queue. This state is maintained by the Schedule pointer which points either to a scheduling routine or a descheduling routine. These routines alternate with one another to cancel complementary events that are processed during the same unit of simulated time.

It is necessary to initialize the entire circuit with consistent values before the simulation begins. This is done during the compilation process by setting all primary inputs and all memory elements to zero, and then performing a levelized zero-delay simulation of the remaining logic. Gates are leveled by performing a topological sort of the network. The level of a gate is the number of gates in the longest path between the primary inputs and the inputs of the gate. Gates are then sorted into levelized order, and each gate is simulated once. The values computed during this simulation are used to initialize the states of all gates and all nets. Values are retained only for primary inputs and outputs, because the differential nature of the simulation does not require net values for intermediate nets. The initialization data is translated into variable initializations in the generated C module. Thus during simulation, the circuit is automatically initialized when the object module is loaded.

```
struct Gshadow
{
    struct Nshadow *Next;
    struct Nshadow *Prev;
    long *Rtn;
    int Delay;
    struct Nshadow *Begin;
    struct Nshadow *End;
    long *Up;
    long *Down;
    long * Schedule;
}
```

Figure 3. The Gate Data Structure.

5 THE SIMULATION ALGORITHM

The simulation algorithm is given in Figure 4. Although this algorithm represents the general flow of the simulation, the code itself is broken into small segments, each one of which is pointed to by either an Nshadow or a Gshadow structure. These routines are, for the most part, devoid of conditional statements.

There are several potential values for the Rtn pointer of the Nshadow structure. The routines of Figure 5 are used for AND, OR, NAND, and NOR gates. These routines come in two flavors, EVUP and EVDN for most events, and EVUP1 and EVDN1 for events that are at the end of a chain. The two routines EVUP and EVDN alternate with one another, as do EVUP1 and EVDN1. The first thing that either routine does is overlay its own address in the Rtn pointer with a pointer to the opposite routine. The reason for this is that as an input to an AND, OR, NAND or NOR gate toggles between zero and one, it will alternately increment and decrement the number of dominant values of a gate. (Zero is the dominant value for AND and NAND, while one is the dominant value for OR and NOR.) If one event causes the dominant count to increment then the next event on the same net must cause it to decrement, and vice versa. (We use the gcc conventions of && to extract the address of a label into a void pointer, and goto * to go to a label whose address is contained in a void pointer.)

The EVUP and EVDN routines terminate by branching to either the Up routine or the Down routine of the associated gate structure. The Up and Down routines may cause the event to propagate to the output. Before going to the Up or Down routine, the current event pointer, shp, is advanced to the next event, and its routine pointer is copied into the NextR (next routine) variable. In all routines, the shp2 variable points to the current gate structure.

```
For Each input vector
    Time = 0
    For Each input that has changed value
        Queue an event in Queue[0]
    End For
    While there are events in the timing wheel
        // Each event is associated with a single gate
        For Each event in Queue[Time%MaxDelay]
            If Event Will Propagate
                If Gate is queued
                    Dequeue Gate
                Else
                    Queue Gate
                End If
            End If
            For Each Queued Gate
                NewTime = Time+Gate.Delay;
                Queue Event-List in Queue[NewTime%MaxDelay]
            End For
            Time++
        End While
    End For
```

Figure 4. The Simulation Algorithm.

The EVUP1 and EVDN1 also go to the Up and Down routines, but since there is no next event in the chain, they place the address of the NEXTQ routine into the NextR variable. This routine will advance the simulation to the next chain of events.

The code for NOT, BUFFER, XOR, and XNOR gates is given in Figure 6. Every change in the input(s) of these gates causes a change in the output, so input events always propagate to the output. Both the NOT and the NOT1
routines terminate by branching to the Schedule routine of the associated gate structure. The differences between the two routines are the same as the differences between EVUP/EVDN and EVUP1/EVDN1.

Figure 6 also contains the routine used to compute values for the primary outputs of the circuit. There is only one version of this routine, because MONITOR events are always at the end of a chain. It toggles the current value of the primary output from one to zero or from zero to one, and then proceeds with the next event in the timing wheel.

At the end of each queue in the timing wheel is a special trailer event whose routine is shown in Figure 7. This routine first checks to determine whether there are any events left in the timing wheel or any gates left in the gate queue. If not, then the simulation of the current vector is finished. For sequential circuits, this routine is also used to check for oscillations. (The oscillation test is not shown.)

If there is work left to do, the trailer routine prints an output vector, resets the current queue to the empty state, and then begins processing the gate structures. The Count variable, which contains a count of the number of events in the timing wheel, is decremented by the number of events in the current queue. The Count variable is used to determine when all work is finished. The variable Q is a two dimensional array containing the timing wheel. The first dimension is indexed by simulation time (modulo the largest delay) and is used to select a particular queue. The second dimension is used to select the various events in a particular queue.

EVUP:

```
shp->Rtn = &&EVDN;
shp2 = shp->Gate;
shp = shp->Next;
NextR = shp->Rtn;
goto * shp2->Up;
```

EVDN:

```
shp->Rtn = &&EVUP;
shp2 = shp->Gate;
shp = shp->Next;
NextR = shp->Rtn;
goto * shp2->Down;
```

EVUP1:

```
shp->Rtn = &&EVDN1;
shp2 = shp->Gate;
NextR = &&NEXTQ;
goto * shp2->Up;
```

EVDN1:

```
shp->Rtn = &&EVUP1;
shp2 = shp->Gate;
NextR = &&NEXTQ;
goto * shp2->Down;
```

NOT:

```
shp2 = shp->Gate;
shp = shp->Next;
NextR = shp->Rtn;
goto * shp2->Schedule;
```

NOT1:

```
shp2 = shp->Gate;
NextR = &&NEXTQ;
goto * shp2->Schedule;
```

MONITOR:

```
shp->Gate = (struct Gshadow*)((long)(shp->Gate) ^ 1);
goto NEXTQ;
```

Figure 6. The NOT, BUFFER, XOR, XNOR, and MONITOR Code.

The QPos array contains pointers to the current positions in each of the queues in the timing wheel. When an event is queued, an address to a chain of pointers is placed into the current queue position determined by QPos. The QCount array is used to count the events in an individual queue. When a queue is emptied, the count is set to zero, and QPos is set to the first position in the queue.

Conceptually each event queue in the timing wheel looks as shown in Figure 8. Each entry in the timing wheel points to a chain of events. Once a chain of events has been processed it is necessary to advance to the next chain in the queue. This is done by the NEXTQ routine shown in Figure 9. The variable QPtr points to the chain of events that is currently being processed. The NEXTQ routine increments this pointer and begins processing the first event in the next chain.

```
if (QCount[Time] == 0 && Count == 0)
{
    return;
}
Else
{
    Count = QCount[Time];
    QCount[Time] = 0;
    QPos[Time] = Q[Time];
    shp2 = GHead.Next;
    goto * shp2->Rtn;
}
```

Figure 7. The Trailer Routine.

Each Gshadow structure represents one gate. This structure contains four routine pointers, Rtn, Up, Down, and Schedule. The Rtn pointer points to the code shown in Figure 10. This code schedules events in the proper queue. The new queue index is computed in the variable NewTime. The current insertion point for each queue is contained in the QPos array. Each gate has an associated chain of events.
which is pointed to by the Begin element of the Gshadow structure. This pointer is inserted into the queue and the current insertion point is incremented. The counts of events in the queue and in the timing wheel are also incremented. The queued/not queued status of the gate is updated by assigning a pointer to the QUEUE routine to the Schedule pointer. Then processing advances to the next Gshadow structure.

![Queue Diagram](image)

**Figure 8. An Event Queue.**

Which is pointed to by the Begin element of the Gshadow structure. This pointer is inserted into the queue and the current insertion point is incremented. The counts of events in the queue and in the timing wheel are also incremented. The queued/not queued status of the gate is updated by assigning a pointer to the QUEUE routine to the Schedule pointer. Then processing advances to the next Gshadow structure.

```plaintext
NEXTQ:
  QPtr++;
  shp = *QPtr;
  goto * shp->Rtn;
```

**Figure 9. The Event Chain Trailer.**

When gates are queued for processing, they are inserted between the head and trailer Gshadow structures. This is done to simplify the dequeueing of complementary events, since each queued structure will then have a non-null forward and back pointer. The processing routine of the header Gshadow is a dummy, but the routine of the trailer Gshadow performs important operations. This routine is shown in Figure 11. First the Gate queue is reset by making the header and trailer structures point to one another. If the timing wheel is empty simulation terminates at this point. Otherwise, the current time is updated, and the QPtr variable is set to point to the head of the next queue in the timing wheel. Processing then advances to the first event in the new queue.

```plaintext
FORWARD:
  NewTime = (Time+shp2->Delay)%MaxDelay;
  *(QPos[NewTime])= shp2->Begin;
  QPos[NewTime]++;
  QCount[NewTime]++;
  Count++;
  shp2->Schedule = &&QUEUE;
  shp2 = shp2->Next;
  goto * shp2->Rtn;
```

**Figure 10. The Gate Processing Routine.**

For NOT, BUFFER, XOR, and XNOR gates, the Up and Down pointers are null. For AND, OR, NAND, and NOR gates, the Up and Down routines are used to keep track of the number of dominant inputs of the gate. (Once simulation code has been generated, AND, OR, NAND, and NOR gates become indistinguishable.) A pair of routines is used for each possible value of the dominant count. Figure 12 shows a few of these pairs. The number of required pairs can be determined by scanning the circuit for the AND/OR/NAND/NOR gate with the most inputs.

In Figure 12, the only two routines that schedule events are the UP0 and DN1 routines. This is done by going to the Schedule routine of the gate. The other routines simply continue with the next event or next chain of events.

```plaintext
GTAI:
  GHead.Next = &GTail;
  GTail.Prev = &GHead;
  if (Count == 0)
  {
    return;
  }
  else
  {
    FullTime++;
    Time = FullTime%2;
    QPtr = Q[Time];
    *(QPos[Time]) = &Trailer;
    shp = *QPtr;
    goto * shp->Rtn;
  }
```

**Figure 11. The Gate Queue Trailer.**

```plaintext
UP0:
  shp2->Up = &&UP1;
  shp2->Down = &&DN1;
  goto * shp2->Schedule;

UP1:
  shp2->Up = &&UP2;
  shp2->Down = &&DN2;
  goto * NextR;
  shp2->Up = &&UP3;
  shp2->Down = &&DN3;
  goto * NextR;
  shp2->Up = &&UP4;
  shp2->Down = &&DN4;
  goto * NextR;
```

**Figure 12. The Up and Down Routines.**

Figure 13 shows the QUEUE and DEQUEUE routines, which are pointed to by the Schedule pointer of the Gshadow structure. When simulation begins, each gate structure is initialized with a pointer to the QUEUE routine. During gate queue processing, these two routines alternate with one another to handle complementary events that are...
processed during the same unit of simulated time. The QUEUE routine performs a standard doubly linked list insertion, placing the gate just after the header structure in the gate queue. The state of the gate is updated by assigning the address of DEQUEUE to the Schedule pointer, and processing advances to the next event. Similarly, the DEQUEUE routine does a standard delink operation, updates the gate status by assigning the address of QUEUE to the Schedule pointer, and proceeds to the next event or chain of events.

QUEUE:

```c
shp2->Next = GHead.Next;
GHead.Next->Prev = shp2;
shp2->Prev = &GHead;
GHead.Next = shp2;
shp2->Schedule = &&DEQUEUE;
goto * NextR;
```

DEQUEUE:

```c
shp2->Next->Prev = shp2->Prev;
shp2->Prev->Next = shp2->Next;
shp2->Schedule = &&QUEUE;
goto * NextR;
```

Figure 13. The Queueing Routines.

The simulation kernel is quite compact. In fact, we have included virtually all of the code in this section. When compiled, the simulation kernel is less than 10,000 bytes.

6 AN EXAMPLE

To illustrate the operation of the simulation algorithm, we will use the following circuit.

The circuit has been initialized at load time with the values computed from an initial input vector of all zeros. It is important to remember, that net values do not exist for Nets X1 through X6. Net values exist only for primary inputs (Nets A through F) and for primary outputs (Nets Q, R, and S.)

We assume that a new input vector is read containing the following values: (A=0,B=1,C=1,D=0,E=1,F=1). The following is the event queue after reading the vector and queuing all input events.

<table>
<thead>
<tr>
<th>B/DN(1)</th>
<th>NEXTQ</th>
<th>C/DN(1)</th>
<th>NEXTQ</th>
<th>E/DN(1)</th>
<th>NEXTQ</th>
<th>F/DN(1)</th>
<th>NEXTQ</th>
<th>TR</th>
</tr>
</thead>
<tbody>
<tr>
<td>TR</td>
<td>TR</td>
<td>TR</td>
<td>TR</td>
<td>TR</td>
<td>TR</td>
<td>TR</td>
<td>TR</td>
<td>TR</td>
</tr>
</tbody>
</table>

At this time the Time-zero position of the queue contains four linked lists of events. Solid lines are used to separate the lists, while dotted lines are used to separate the items in a linked list. The notation B/DN(1) indicates that the event processing routine is EVDN (See Figure xxx), and that the EVDN routine will eventually load the address of DN1 (See Figure 12) from the gate structure and execute it. The notation TR indicates the trailer event which is executed when all events in the queue have been processed. Each TR event points to the TRAILER routine. (See Figure 7.) Once these events have been processed, the gate queue will look as follows.

| G3/(UP0,DNO)/DEQ |
| G4/(UP0,DNO)/DEQ |
| G5/(UP0,DNO)/DEQ |
| G6/(UP0,DNO)/DEQ |
| Trailer |

Each entry in the above diagram shows the gate, and its two states: routine pointer, and queue state. These states do not affect the processing of the gates, because all gates point to the FORWARD routine (See Figure 10.) However the FORWARD routine will reset the queuing state from DEQUEUE to QUEUE. The FORWARD routine will also cause a number of new events to be queued. The delay of each gate determines into which queue the new events will be placed. Once all gates have been processed, the event queue will look as follows.

| TR |
| TR |
| TR |
| X3/UP(0) | NEXTQ | TR |
| X4/UP | NEXTQ | TR |
| X6/UP | NEXTQ | TR |
| X5a/UP | X5b/UP | NEXTQ | TR |
| TR |

Note that two events are queued for net X5. This is because this net fans out to two gates, and one event is queued for each fanout branch.

No events are queued for time 1, so the net and gate trailer routines will advance the time to time 2. The event for
X3 will be processed at time 2, giving the following gate queue.

<table>
<thead>
<tr>
<th>G7/(UP1,DN1)/DEQ</th>
</tr>
</thead>
<tbody>
<tr>
<td>Trailer</td>
</tr>
</tbody>
</table>

When G7 is processed, it will add an event to the event queue, resulting in the following queue.

<table>
<thead>
<tr>
<th>TR</th>
<th>TR</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ra/NOT Rb/MON NEXTQ TR</td>
<td></td>
</tr>
<tr>
<td>X4/UP(1) NEXTQ TR</td>
<td></td>
</tr>
<tr>
<td>X6/UP NEXTQ TR</td>
<td></td>
</tr>
<tr>
<td>X5a/UP X5b/UP NEXTQ TR</td>
<td></td>
</tr>
<tr>
<td>TR</td>
<td></td>
</tr>
</tbody>
</table>

Again, two events are queued for Net R, one for each fanout branch. The first goes to a NOT gate. The NOT processing routine propagates all events unconditionally. (See Figure 6.) MON indicates the monitor routine given in Figure 6. Note that these events are queued at Time 9, not Time 2. Once the Time 2 events have been processed and removed from the queue, the queue is available for events at later times.

The event for X4 causes the routine UP1 to be executed. This does not cause any gates to be queued, because only the routines DN1 and UP0 queue gates. Thus we go on to the following event queue.

<table>
<thead>
<tr>
<th>TR</th>
<th>TR</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ra/NOT Rb/MON NEXTQ TR</td>
<td></td>
</tr>
<tr>
<td>X6/UP(0) NEXTQ TR</td>
<td></td>
</tr>
<tr>
<td>X5a/UP X5b/UP NEXTQ TR</td>
<td></td>
</tr>
<tr>
<td>TR</td>
<td></td>
</tr>
</tbody>
</table>

Executing the event for X6 will cause the gate G8 to be queued, giving the following gate queue.

<table>
<thead>
<tr>
<th>G8/(UP1,DN1)/DEQ</th>
</tr>
</thead>
<tbody>
<tr>
<td>Trailer</td>
</tr>
</tbody>
</table>

Processing the gate G8 will add new events to the queue as follows.

<table>
<thead>
<tr>
<th>TR</th>
<th>S/MON NEXTQ TR</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ra/NOT Rb/MON NEXTQ TR</td>
<td></td>
</tr>
<tr>
<td>TR</td>
<td>TR</td>
</tr>
<tr>
<td>X5a/UP(2) X5b/UP(1) NEXTQ TR</td>
<td></td>
</tr>
<tr>
<td>TR</td>
<td></td>
</tr>
</tbody>
</table>

Processing the events for X5 will cause the UP2 and UP1 routines to execute, neither of which will queue any gates. The simulation will then advance to time 8 and process the event for Net S. This will cause the value of Net S to toggle from 0 to 1, but no gates will be queued. The simulation will advance to time 9 and the events for R will be processed. This will cause the value of Net R to toggle from 0 to 1, and will cause an event to be queued for net Q, as follows.

<table>
<thead>
<tr>
<th>TR</th>
<th>TR</th>
<th>TR</th>
</tr>
</thead>
<tbody>
<tr>
<td>O/MON NEXTQ</td>
<td></td>
<td></td>
</tr>
<tr>
<td>TR</td>
<td>TR</td>
<td>TR</td>
</tr>
<tr>
<td>TR</td>
<td>TR</td>
<td>TR</td>
</tr>
</tbody>
</table>

Finally, at time 10, the event for Net Q will be processed, causing its value to toggle from 1 to 0. No gates are queued, so the event queue becomes empty and simulation terminates.

This simulation illustrates event propagation, but not event cancellation. In the multi-delay EVCF simulator, placing a gate in the gate queue eventually causes events to be queued in the event queue. Thus event cancellation is performed while queuing gates rather than while queuing events. Consider the following circuit

![Circuit Diagram]

Assume that we read a new input vector for this circuit with the following values: (A=1, B=0). This will give us two events at time zero as follows:

<table>
<thead>
<tr>
<th>A/DN(1)</th>
<th>B/UP(0)</th>
<th>NEXTQ</th>
<th>TR</th>
</tr>
</thead>
<tbody>
<tr>
<td>TR</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Before executing the first event, the gate G1 is in the following state: G1/(UP1,DN1)/QUEUE. Processing the first event causes the state of G1 to change to G1/(UP0,DN0)/DEQUEUE, and will cause the gate G1 to be placed in the gate queue. Because of the DEQUEUE state of G1, processing the event for B will remove G1 from the gate queue. Thus the resulting state will be G1/(UP1,DN1)/QUEUE, and no gates will remain in the gate queue. This will also empty the event queue and cause the simulation to terminate.

7 EXPERIMENTAL RESULTS
To determine the effectiveness of the EVCF simulator as compared to a conventional simulation, we used the ISCAS-85 benchmarks [10], which have become a de-facto standard for determining simulation performance. We
simulated each circuit with 500,000 randomly generated input vectors, and measured the amount of CPU time required to complete the simulation. The processor was a 3.06Ghz Xeon processor with 2GB of 233Mhz memory. This system was dedicated during the simulations, but could not be complexly isolated. The results are reported in terms of CPU seconds in Figure 14.

Although the performance gains are not as spectacular as the 3-4x gains seen in the zero-delay and unit-delay models, they are substantial, with half of the circuits seeing more than a 2x improvement. These results show that the EVCF technique can provide significant performance improvements over conventional multi-delay simulation and that this approach is definitely worth pursuing in the future.

<table>
<thead>
<tr>
<th>Circuit</th>
<th>Conventional</th>
<th>EVCF</th>
<th>Improvement</th>
</tr>
</thead>
<tbody>
<tr>
<td>c432</td>
<td>15.236</td>
<td>10.166</td>
<td>1.498</td>
</tr>
<tr>
<td>c499</td>
<td>16.762</td>
<td>11.954</td>
<td>1.402</td>
</tr>
<tr>
<td>c880</td>
<td>27.870</td>
<td>14.818</td>
<td>1.880</td>
</tr>
<tr>
<td>c1355</td>
<td>51.394</td>
<td>33.768</td>
<td>1.521</td>
</tr>
<tr>
<td>c1908</td>
<td>101.390</td>
<td>45.344</td>
<td>2.236</td>
</tr>
<tr>
<td>c2670</td>
<td>138.606</td>
<td>83.510</td>
<td>1.659</td>
</tr>
<tr>
<td>c3540</td>
<td>209.890</td>
<td>86.236</td>
<td>2.433</td>
</tr>
<tr>
<td>c5315</td>
<td>388.892</td>
<td>185.515</td>
<td>2.096</td>
</tr>
<tr>
<td>c6288</td>
<td>3273.420</td>
<td>1523.798</td>
<td>2.148</td>
</tr>
<tr>
<td>c7552</td>
<td>714.560</td>
<td>283.516</td>
<td>2.520</td>
</tr>
</tbody>
</table>

Figure 14. Experimental Results.

8 CONCLUSION
The EVCF technique has proven adaptable to new simulation models as has been shown in [7] and in this paper. Unlike early differential simulators, the adaptation to new timing models has proven to be relatively painless. Substantial performance gains over conventional simulation have been realized for each of the timing models that have currently been implemented. Additional work is needed to study models such as the inertial delay model and the nominal delay model, which uses floating point delays. As it stands, the EVCF technique is a powerful simulation tool which can be used effectively in existing simulators, and should prove to be adaptable to a number of new areas in the future.

9 REFERENCES