Unit-Delay Simulation with the EVCF Algorithm

Peter M Maurer
Dept. of Computer Science, Baylor University, Waco, TX, 76798-7356, USA

Abstract - The EVCF (Event-Driven Condition Free) algorithm is a differential simulation algorithm that is capable of simulating virtually any type of circuit. In this paper we extend the EVCF algorithm to the Unit-Delay model, which allows for the detection of both static and dynamic hazards in a circuit. The Unit-Delay EVCF algorithm is fast, more than four times faster than conventional unit-delay simulation, and has an extremely small simulation core of around 2,000 bytes. It is suitable for use in debugging commercial-grade circuits.

Keywords: Digital Simulation, Logic Simulation, Unit-Delay Timing

1 Introduction

Event driven logic simulation has been the subject of much research[1]. A variety of logic models and delay models have been studied[2]. There have been several improvements in performance based on two different approaches. Internally, a logic simulator consists of scheduling algorithms and gate simulation algorithms. One approach to improving performance is to improve the scheduling algorithms [3,4,5]. The other approach is to improve gate-simulation speed [6,7]. Differential simulators do both. They focus on improving scheduling speed by improving the efficiency of event propagation and they eliminate much if not all of the gate-simulation code. [7]. The Event-Driven-Condition-Free (EVCF) logic-simulation algorithm [8] is intriguing in two respects. First, it operates in differential mode. Instead of computing the values of gate outputs, it computes differences between successive inputs to determine whether events propagate from gate inputs to gate outputs. In such a simulation, no net values are required, except at the primary inputs and primary outputs. This enables some peculiar optimizations. Because NOT and BUFFER gates always propagate events, they can be eliminated entirely from the simulation. AND, OR, NAND, and NOR gates all appear identical during the simulation, permitting gates to be combined in unusual ways.

The second intriguing aspect of the EVCF algorithm is that it relies entirely on metamorphic code for implementing internal states. Except for processing the primary inputs and testing for termination, the code contains no conditional branches and no loops. The code is peculiar looking, and extremely compact. The EVCF algorithm is capable of simulating virtually any circuit using a simulation core of around 1,500 bytes. In addition, the EVCF algorithm is many times faster than conventional simulation.

The major drawback of the EVCF algorithm is that it is a zero-delay algorithm. Run-time queues are complicated, because the circuit must be levelized during simulation to avoid simulating a gate before its inputs have been computed. Hazards can neither be detected nor reported because each gate is simulated only once. This means that potential instabilities in the circuit can go undetected. Cyclic circuits cannot be handled properly for the same reason.

In the following we show that the EVCF algorithm can be adapted to handle the unit-delay model in which each gate has an implicit delay of one. This is sufficient to detect hazards and oscillations in a circuit, and it provides a basis upon which an even more accurate timing simulation can be built.

2 Event Handling

The unit-delay EVCF algorithm has two types of structures, event structures which represent one fanout branch of a net, and gate structures which represent gates. A net is represented as a linked-list of events, one for each fanout branch. Primary outputs have an additional fanout branch that is used to compute the value of the output. Figure 1 shows the details of the event structure. The Next and Prev fields are used to create a doubly-linked list of events. (Double links are used to facilitate event cancellation.) The Routine field points to the processing routine for the event. The Gate field points to the gate-structure of the fanout branch or the value of the net for primary output branches.

Figure 2 shows the representation of a net. This linked-list of structures is added and removed from the event queue as a unit. When an event occurs, the entire list is added to the event queue, and when an event is canceled, the entire list is removed from the queue.

The Routine field of the event structure points to one of four routines. These routines are accessed using a computed go-to. The four routines are shown in Figure 3. The code in Figure 3 uses the GCC extensions for handling pointers to labels and computed go-tos. EVUP and EVDN are used for AND, OR, NAND and NOR gates, while NOT is used for NOT, BUFFER, XOR, and XNOR gates. The MONITOR routine is used to compute the values of primary outputs.

In the routines of Figure 3, the variable shp points to the current event structure. The EVUP and EVDN routines alternate with one another by replacing the Routine pointer with each other’s addresses. After doing this, these routines load the Gate pointer and then branch to a routine contained
in the gate structure for further processing. The NOT routine skips the additional processing and goes immediately to the scheduling routine.

Figure 1. An Event Structure.

Figure 2. A Net Structure.

The MONITOR routine computes the new value of the gate by inverting the existing value. The MONITOR event structure will never be scheduled unless the value of the output changes. No additional events are scheduled by a MONITOR event, so the routine simply goes to the next event structure and begins executing it.

EVUP:
\[
shp->\text{Routine} = \&\&\text{EVDN};
shp2 = shp->\text{Gate};
goto * shp2->\text{Up};
\]

NOT:
\[
shp2 = shp->\text{Gate};
goto * shp2->\text{Schedule};
\]

EVDN:
\[
shp->\text{Routine} = \&\&\text{EVUP};
shp2 = shp->\text{Gate};
goto * shp2->\text{Down};
\]

MONITOR:
\[
shp->\text{Gate} =
(\text{struct } Gshadow *)((\text{long})(shp->\text{Gate}) \land 1);
shp = shp->\text{Next};
goto * shp->\text{Routine};
\]

Figure 3. Event-Handling Routines.

3 Gate Handling

The gate structure is shown in Figure 4. This structure is used by EVUP, EVDN, and NOT events. This structure will be scheduled as if it were an event, so the three important scheduling fields, \text{Next}, \text{Prev}, and \text{Routine}, are present in this structure. The \text{Begin} and \text{End} fields point to the linked lists of events that represent the output of the gate. This list of events will be scheduled if the output of the gate changes. The \text{Up} and \text{Down} fields contain pointers to processing routines. These routines are used by EVUP and EVDN events. The processing routines are used to recompute the state of the gate and to determine whether the output of the gate changes. If the output of the gate changes, these routines will schedule the gate using the \text{Next} and \text{Prev} fields of the gate structure.

The \text{Schedule} field points to the current scheduler of the gate. This alternates between the QUEUE and the DEQUEUE routines. The QUEUE routine adds the gate structure to the queue, while the DEQUEUE routine removes the gate structure from the queue, thus cancelling event propagation.

Figure 4. The Gate Structure.

The state of an AND, OR, NAND or NOR gate is maintained by using several different UP and DOWN routines. Figure 5 gives a sample of these routines. As a practical matter, only a few routines are needed, because the number of inputs of an AND, OR, NAND, or NOR gate is limited by the underlying technology.

Note that each of the UP and DOWN routines schedules new UP and DOWN routines. (Except DN0 which should never be executed) Except for UP0 and DN1, the routines continue with the next event. Instead of going to the next event, the UP0 and DN1 routines execute the routine pointed to by the \text{Schedule} field of the gate. The pairs of routines, UPx/DNx, represent the number of inputs of an AND, OR, NAND, or NOR gate that have the dominant value (x). The output of the gate changes when the dominant count goes from 0 to 1 (UP0 executes), or when the count goes from 1 to 0 (DN1 executes). In all other cases, the output of the gate does not change.

UP0:
\[
shp2->\text{Up} = \&\&\text{UP1};
shp2->\text{Down} = \&\&\text{DN1};
goto * shp2->\text{Schedule};
\]

UP1:
\[
shp2->\text{Up} = \&\&\text{UP2};
shp2->\text{Down} = \&\&\text{DN2};
shp = shp->\text{Next};
goto * shp->\text{Routine};
\]

DN0:
\[
shp = shp->\text{Next};
goto * shp->\text{Routine};
\]

DN1:
\[
shp2->\text{Up} = \&\&\text{UP0};
shp2->\text{Down} = \&\&\text{DN0};
goto * shp2->\text{Schedule};
\]

UP2:
\[
shp2->\text{Up} = \&\&\text{UP3};
shp2->\text{Down} = \&\&\text{DN3};
shp = shp->\text{Next};
goto * shp->\text{Routine};
\]

UP3:
\[
shp2->\text{Up} = \&\&\text{UP4};
shp2->\text{Down} = \&\&\text{DN4};
shp = shp->\text{Next};
goto * shp->\text{Routine};
\]

DN2:
\[
shp2->\text{Up} = \&\&\text{UP1};
shp2->\text{Down} = \&\&\text{DN1};
shp = shp->\text{Next};
goto * shp->\text{Routine};
\]

DN3:
\[
shp2->\text{Up} = \&\&\text{UP2};
shp2->\text{Down} = \&\&\text{DN2};
shp = shp->\text{Next};
goto * shp->\text{Routine};
\]

Figure 5. Up and Down routines.
Figure 6. The QUEUE and DEQUEUE routines.

Figure 6 shows the QUEUE and DEQUEUE routines. The QUEUE routine takes the current gate structure, which is pointed to by the variable \textit{shp2}, and adds it to the end of the queue. The variable \textit{Trailer} points to the element that is currently at the end of the queue. This structure is a marker that must remain at the end of the queue, so gate structures must be inserted ahead of this element. The queue is initialized with a dummy element at the head of the queue, so the Trailer structure will always point back to a valid queue element. The first four statements of the QUEUE routine perform a standard double-link operation. Next, the QUEUE routine replaces its own address with the address of the DEQUEUE routine, and finally advances to the next event.

The DEQUEUE routine is simpler than the QUEUE routine, because only the fields of the events remaining in the queue must be changed. The routine delinks the gate structure, replaces its own address with the address of the DEQUEUE routine, and finally advances to the next event.

4 Event Scheduling

Event scheduling is done by the gate data structures once they have been placed in the event queue. These structures are not placed in the queue unless the output of the gate changes. Trailer events are used to mark time intervals and to terminate the queue. Figure 7 shows the general structure of the queue at different times. Queue state A occurs right after a new input vector has been read, and new events have been scheduled. This state will recur many times throughout the course of the simulation. The \textit{Trailer} event marks the end of the current simulation time unit and the \textit{Tail} event marks the end of the queue. State B occurs after some events have been processed. These events will generally cause Gate structures to be scheduled. Since the gate processing belongs to the current time interval, the gates are inserted ahead of the \textit{Trailer} event.

Queue state C shows the queue after all events have been processed and after some gates have been processed. Processing a gate will cause events to be scheduled, but because these events belong to the next time interval, not to the current time interval, they are inserted after the \textit{Trailer} event and ahead of the \textit{Tail} event. Figure 8 shows the processing routine for the gate structures. All gate structures use the same processing routine.

4.1 Event Scheduling

The FORWARD routine of Figure 8 first adds events unconditionally to the queue. If the gate structure is in the queue, then the output of the gate has changed, and it is necessary to schedule events. The list of events is pointed to by the \textit{Begin} and \textit{End} fields of the gate structure. The routine then resets the corresponding gate structure so that it will queue the next event.

The \textit{Trailer} event is used to mark the boundaries between two consecutive units of simulated time. The first action of the \textit{Trailer} event handler is to print an output vector containing the value of each primary output. Next, the event handler tests its \textit{Next} pointer. If the next event is the \textit{Tail} event, indicating that there are no events queued for the next time period, the trailer event processor simply continues with the next event, terminating the simulation of the current input.
vector. However, if there are events queued for the next time period, the trailer event handler inserts itself at the end of these events. Figure 9 shows the code for the Trailer and Tail event handlers.

Apart from the code used to read input vectors and print output vectors, all code for the simulator appears in this paper. Needless to say, the simulation core is tiny compared to that of conventional simulators: around 2,000 bytes.

5 Experimental Data

Figure 10 shows the experimental results for the unit delay EVCF algorithm. These results use the ISCAS-85 benchmarks [9] which have become a de-facto standard for measuring the performance of simulation algorithms. The simulations were performed on a dedicated 3.06 Ghz Xeon processor with 2GB of 233 Mhz memory. The results are in seconds of execution time for 5,000 input vectors. The unit-delay EVCF algorithm outperforms conventional unit-delay simulation by a factor of between 4 and 5 for most circuits.

<table>
<thead>
<tr>
<th>Circuit</th>
<th>Conventional</th>
<th>EVCF</th>
<th>Improvement</th>
</tr>
</thead>
<tbody>
<tr>
<td>C432</td>
<td>0.528</td>
<td>0.138</td>
<td>3.826</td>
</tr>
<tr>
<td>C499</td>
<td>1.672</td>
<td>0.344</td>
<td>4.860</td>
</tr>
<tr>
<td>C880</td>
<td>1.752</td>
<td>0.464</td>
<td>3.776</td>
</tr>
<tr>
<td>C1355</td>
<td>3.988</td>
<td>0.760</td>
<td>5.247</td>
</tr>
<tr>
<td>C1908</td>
<td>4.092</td>
<td>0.944</td>
<td>4.335</td>
</tr>
<tr>
<td>C2670</td>
<td>13.226</td>
<td>3.180</td>
<td>4.159</td>
</tr>
<tr>
<td>C3540</td>
<td>5.448</td>
<td>1.172</td>
<td>4.683</td>
</tr>
<tr>
<td>C5315</td>
<td>24.302</td>
<td>5.076</td>
<td>4.788</td>
</tr>
<tr>
<td>C6288</td>
<td>30.928</td>
<td>13.692</td>
<td>2.759</td>
</tr>
<tr>
<td>C7552</td>
<td>25.440</td>
<td>5.108</td>
<td>4.980</td>
</tr>
</tbody>
</table>

Figure 10. Experimental Results.

6 Conclusions

The unit-delay EVCF algorithm is an unconventional algorithm that is extremely efficient both in terms of execution time and computer memory. By operating in differential mode it is able to realize an improvement of 4-5x over conventional unit-delay simulation. Because it uses metamorphic coding which is virtually devoid of loops and conditions, the simulation core is tiny, around 2,000 bytes. Despite the size of the simulation core, the unit-delay EVCF algorithm is able to simulate virtually any circuit, including the standard ISCAS-85 benchmark circuits. Unlike the zero-delay EVCF algorithm, the unit-delay EVCF algorithm is able to detect both static and dynamic hazards, making it a useful tool for debugging commercial-grade designs.

7 References