

Event Driven Simulation Without Loops or Conditionals

Peter M. Maurer

Dept. of Comp. Sci. & Eng. Univ. of South Florida, Tampa, Florida 33620.

1. Introduction

The past several years have seen much research in event driven logic simulation[1]. Various logic and delay models have been explored[2]. Most simulation research has focused on improving simulation performance. New approaches to both compiled and event driven simulation have been explored[3,4,5].

The internal operations of event-driven simulators can be divided into two categories, scheduling, and gate simulation. Much effort has been focused on reducing the cost of scheduling[3,4,5]. There has also been effort to reduce the cost of gate simulation[6,7]. It has also been shown that explicit computation of gate outputs is unnecessary, as long as event-propagation is computed correctly[7].

Even though research has reduced the complexity of both scheduling and gate simulation, it is still necessary to test for event propagation and cancellation, and it is necessary to perform *some* computations during gate simulation.

This paper will show that *none* of these computations are necessary. Most computations are devoted testing internal states and computing new internal states. In our technique, subroutine addresses are used to maintain states. This permits the elimination of all state-testing and state-computation code. Our technique is significantly faster than conventional event-driven simulation[1]. Unlike earlier methods[7], our approach can easily be extended to any logic model or any delay model.

2. The EVCF Technique

The EVCF (Event-Driven Conditional-Free) event-driven simulation technique does not perform any conditional tests. In fact, it operates without performing computation of any kind. The simulation kernel has two types of statements, assignments and computed gotos. The assignment statements do nothing more than assign a value to a variable. These are simple assignments, no computation is performed by these statements. To a person reading the simulation kernel, it is not apparent that the code performs *any* useful function. Nevertheless, the EVCF works correctly and outperforms more conventional simulation algorithms, and uses far less run-time code than other algorithms.

The EVCF Technique is based on an idea borrowed from object-oriented programming. In a conventional simulation system, a gate could be represented by a data structure similar to that shown in Figure 1. From time to time during the simulation it will be necessary to create a queue of gates and process them in some manner. Since each type of gate requires a different type of processing, it is necessary to examine the type code of the data structure before deciding what actions to perform.

In object oriented programming, a different structure would be used for each different gate type. The type code would be eliminated, and replaced by a table of virtual functions. Each

different gate-type would have its own table of virtual functions. These functions perform the various different kinds of processing needed by the individual gates. Instead of using a type code to distinguish one type of gate from another we use the function addresses in the virtual function table. The function addresses are more useful than a plain type code, because we can branch to them directly to perform the required operations. No decoding of the type-code is necessary.

Next
Type
Inputs
Outputs
Other

Figure 1. A Sample Data Structure.

The concept of function addresses instead of type codes has been used in simulation by the shadow algorithm[8]. In the shadow algorithm, the data structures for nets and gates contain processing routine addresses. Because the shadow algorithm is a threaded-code algorithm, the processing routine addresses are branch targets for computed gotos rather than virtual function addresses.

The EVCF Technique is a state-machine based Technique similar to those described in [7], however the EVCF Technique uses state machines in several new ways. In addition to using state machines to represent nets and gates, they are also used to represent the queue state, the queueing status of gates and nets, and the value of each monitored net. The state of each machine is represented by a collection of processing routine addresses. Rather than decoding the current state, the EVCF Technique branches directly to the processing routine for a particular state. The correspondence between states and processing routines is not one-to-one. Some states require several processing routines.

The precise structure of the state machines in the EVCF Technique depends on the delay model. (The EVCF Technique can be used with any conventional delay model.) One of the simplest state machine structures is that used to implement zero delay model of the LECSIM algorithm. In this model, a state machine is required for each gate and each net in the circuit. The net state machines are used to model net values, while the gate state machines are used for event propagation. In addition, each gate requires a second state machine that is used to model the queueing state of the gate outputs. These machines are used both for queuing events and for event cancellation. A master state machine is used to keep track of the progress of the simulation. In the LECSIM model, there are several event queues, one for

each level in the circuit. The master state machine is used to determine which queue is currently being processed.

The structure of Figure 2 is used to represent an event. Because different fanout branches of a net have different effects on the gate to which they are input, one such structure is required for each fanout branch of a net. These structures are placed in the event queues, and control the progress of the simulation. As in reference[7] all structures for a particular net are pre-linked to one another before the simulation starts. Pre-linking permits a chain of structures to be queued using essentially the same operations required to queue a single structure.

```

struct Net
{
    struct Net *Next;
    struct Net *Prev;
    Address ProcessRoutine;
    struct Gate *Output;
    struct Gate *Driver;
}

```

Figure 2. The Net Structure.

In the *Net* structure, the process routine address points to different routines depending on the type of gate pointed to by the *Output* element. The *Next* and *Prev* elements are used for queueing, while the *Driver* element is used for queue management (see below). Figure 3 gives the routines for AND and OR gates. In this code, it is assumed that the variable *Shp* points to the *Net* structure. This code is written in pseudo-C, assuming that indirect gotos are legal, and that it is possible to obtain the address of a label by preceding it with an amersand. Since neither of these assumptions is true, the actual code for these routines must contain some assembly language.

```

EVUP:
    Shp->ProcessRoutine = &EVDN;
    Shp2 = Shp->Output;
    goto *Shp2->Up;
EVDN:
    Shp->ProcessRoutine = &EVUP;
    Shp2 = Shp->Output;
    goto *Shp2->Down;

```

Figure 3. Event Handling Code.

The “EVUP” and “EVDN” routines of Figure 3 are used to model the state of a net. Rather than explicitly modeling net values, these routines model the expected result a net will have on a gate once an event occurs. Although the state of a gate is modeled using a count of dominant inputs, no explicit count is maintained. Instead, there are a number of different handling routines for each possible value of the count. The structure used to represent AND and OR gates is illustrated in Figure 4. (Modeling other types of gates requires changes in both the “Net” and “Gate” structures, as well as changes in the handling routines.)

The “Up” and “Down” elements shown in Figure 4 are used to maintain the state of the gate, while the remainder of the elements are used for queueing events. The “Schedule” element determines whether an event is currently on the queue, the

“Begin” and “End” elements give the beginning and ending of the event chain for the output of the gate, and the “QueueHead” element determines the queue in which to schedule events for the gate output. (In the LECSIM algorithm, events must be queued in the queue that corresponds to the level of the net.)

```

struct Gate
{
    struct Net *Begin;
    struct Net *End;
    Address Up;
    Address Down;
    Address Schedule;
    struct Net *QueueHead
}

```

Figure 4. The Gate Structure.

Maintaining the state of a gate requires several handling routines, some of which are illustrated in Figure 5. It is necessary to provide one pair of routines for each possible value of the dominant count. Figure 5 illustrates the routines for values 0, 1, and 2. The routines for higher values are identical to those for the value 2. The “UPx” routines are used when it is necessary to increment the dominant count, while the “DNx” routines are used for decrementing the count. The routines themselves are used as the counter, so no explicit count is maintained. Events are scheduled by the routines UP0, and DN1. All other routines simply process the next event. Our current implementation of the EVCF Technique supports AND, OR, NAND, and NOR as well as XOR, XNOR, NOT, and BUFFER gates. AND, OR, NAND, and NOR gates are simulated using the routines given in Figure 5, while XOR, XNOR, NOT, and BUFFER gates are simulated using a “pass-through” routine that propagates all events.

UP0: shp2->Up = &UP1; shp2->Down = &DN1; goto *shp2->Schedule;	DN0: shp = shp->Next; goto *shp->ProcessRoutine;
UP1: shp2->Up = &UP2; shp2->Down = &DN2; shp = shp->Next; goto *shp->ProcessRoutine;	DN1: shp2->Up = &UP0; shp2->Down = &DN0; goto *shp2->Schedule;
UP2: shp2->Up = &UP3; shp2->Down = &DN3; shp = shp->Next; goto *shp->ProcessRoutine;	DN2: shp2->Up = &UP1; shp2->Down = &DN1; shp = shp->Next; goto *shp->ProcessRoutine;

Figure 5. AND/OR Simulation Routines.

As the *UP0* and *DN1* routines of Figure 5 illustrate, events are propagated by branching to the *Schedule* routine of the *Gate* structure. This routine must handle two situations. If no event is queued for the gate output, then a new event must be placed in the queue. If an event is already queued, the new event will be complementary and simultaneous with the currently queued event. This implies that the queued event should be cancelled. The *Schedule* routine itself is used to keep track of whether an event is queued for the gate output. Two scheduling routines are

used, one that queues a new event and one that cancels an existing event. These routines schedule one another as illustrated in Figure 6. The queue/dequeue operations are the standard operations that one must perform to insert or delete an item from a doubly linked list. To simplify queue processing, each queue has a head and a tail that are never removed from the queue. To schedule an event, one simply inserts a new item after the existing queue head. Removing items from the queue is simpler than queueing a new element, because the state of the *Prev* and *Next* items does not need to be maintained for items that are not queued.

There are two additional problems that must be solved with respect to event queueing. Processed events must be removed from the queue, and the state of the queueing routine must be reset to allow new events to be scheduled for the next input vector. Resetting the queueing routine is done by modifying the EVUP and EVDN routines of Figure 3. These are replaced with two new routines EVUP1 and EVDN1, which are shown in Figure 7. These new routines use the Driver element of the Net structure to reset the queueing routine for the driving gate of the net. It is not necessary to physically remove the event from the queue at this point, since this will be taken care of during the queue management phase of the simulation. If a net, due to fanout, has several event structures, the new routines are used only for the last event in the chain. If a net is a primary input and has no driving gate, the new routines are not used.

QUEUE:

```
shp2->Schedule = &DEQUEUE;
shp2->End->Next = shp2->Head->Next;
shp2->Head->Next->Prev = shp2->End;
shp2->Head->Next = shp2->Begin;
shp2->Begin->Prev = shp2->Head;
shp = shp->Next;
goto *shp->Rtn;
```

DEQUEUE:

```
shp2->Schedule = &QUEUE;
shp2->Begin->Prev->Next = shp2->End->Next;
shp2->End->Next->Prev = shp2->Begin->Prev;
shp = shp->Next;
goto *shp->Rtn;
```

Figure 6. Queueing Routines.

The queue management phase of the simulation has three parts, initiation of queue processing, termination of queue processing, and resetting processed queues. As mentioned above, the simulation requires an array of queues, one for each level in the circuit. Each queue has a header element and a trailer element that are never removed. The header element is a degenerate data structure consisting of a single *Next* pointer. The entire collection of header elements is maintained as an array of pointers to *Net* structures. The trailer element is a real element with its own processing routine. The trailer processing routines are responsible for advancing the state of the simulation from one queue to the next, resetting queues by deleting all queued events, and for termination of the simulation. Queue processing is initiated by initializing the *shp* pointer to the head of the first queue and branching to the first processing routine. It is assumed

that events for the primary inputs have already been queued at this point. Figure 8 shows the code used to initiate queue processing. In this code, the variable *Head* is the array of *Net* pointers, and the variable *Headwork* contains a pointer to the current queue.

EVUP1:

```
shp->Rtn = &EVDN1;
shp->Driver->Schedule = &QUEUE;
shp2 = shp->Gate;
goto *shp2->Up;
```

EVDN1:

```
shp->Rtn = &EVUP1;
shp->Driver->Schedule = &QUEUE;
shp2 = shp->Gate;
goto *shp2->Down;
```

Figure 7. Dequeing processed events.

```
shp = Head[0];
goto *shp->Rtn;
```

Figure 8. Initiating Queue Processing.

There are two trailer routines, one that advances the simulation to the next queue, and one that terminates the simulation. These two routines are shown in Figure 9. The first of these routines advances the simulation to the next queue, while the second terminates the simulation. The second routine is used only in the final queue.

TRAILER:

```
shp->Head->Next = shp;
shp->Prev = shp->Head;
shp = *(struct Net **)(shp->Gate);
goto *shp->Rtn;
```

TRAILER1:

```
shp->Head->Next = shp;
shp->Prev = shp->Head;
return;
```

Figure 9. Trailer Routines.

The routines shown in Figure 9 reset the queue by making the queue header point to the trailer structure. To save space, the *Gate* element is overloaded to hold a pointer to the head of the next queue. It is not necessary to explicitly remove elements from the queue, because the values *Prev* and *Next* elements are unimportant for elements not on the queue. It is the value of the *Schedule* pointer that determines the queueing status of an event, not the actual presence of the event in the queue.

The final state machines are those that are used to maintain the value of monitored nets. Since these net values do not affect the progress of the simulation, they can be updated at any convenient place. The EVCF Technique creates a special gate called a Net Monitor, and adds it to the fanout of each monitored net. The routines to handle the Net Monitor are shown in Figure 10. As in the trailer routines, the *Gate* element of the *Net* structure is overloaded.

```

MONITOR0:
  shp->ProcessRoutine = &MONITOR1;
  (long)(shp->Gate) = '0';
  shp = shp->Next;
  goto *shp->Rtn;

```

```

MONITOR0:
  shp->ProcessRoutine = &MONITOR0;
  (long)(shp->Gate) = '1';
  shp = shp->Next;
  goto *shp->Rtn;

```

Figure 10. Net Monitor Routines.

3. Input and output.

With a couple of minor exceptions, the entire simulation kernel for the EVCF Technique is given in the preceding figures. A complete printout of the code requires about two and a half pages. No loops or conditionals are required. One place where conditionals *are* required is in the processing of input vectors. Before event processing begins, it is necessary to read and parse an input vector, and schedule events for those primary inputs that have changed.

There are several ways that conditional testing could be eliminated from input processing, but they are either inefficient or impractical at this time. For example, one could pre-parse a set of input vectors into an array of routine addresses, and use the pre-parsed vectors for several different simulations. Two different routines could be used, one that queues an event, and one that does not. Trailer addresses could be used to start event processing and to terminate the simulation. Unfortunately, to be useful, the pre-parsed vectors would have to outlive the process that created them. Once the process that creates them terminates, the routine addresses will no longer be valid. A few years ago, this approach wouldn't even be worth mentioning. However, recent progress in object-oriented programming, remote object access, and the marshalling of complex data structures suggests that this approach *may* be practical at some time in the future.

The EVCF Technique permits the elimination of much of the traditional vector-output code. In traditional simulations, net values are represented as bits and must be translated into ASCII characters for output. (A bit-level representation is required for efficient gate simulation.) In the EVCF technique, the MONITOR routines of Figure 10 store the ASCII output values directly into the output vector. A simple print statement at the end of the event-processing routine is all that is necessary for vector output.

4. Other Logic and Timing Models.

Despite the peculiar-looking code used for the EVCF technique, the structure of the simulation is quite close to that of more traditional simulation techniques. Gate processing and event processing are clearly separate from one another, and there are explicit points at which queue management is performed. This allows easy extension to more complex delay models, such as unit-delay and multi-delay simulation. Both single-list and double-list simulations are possible without great difficulty. Extension to more complex logic models, particularly those that include the X and Z values should be no problem.

5. Performance.

Several experiments were run to demonstrate the performance of the EVCF Technique. The numbers reported in the table are expressed in CPU seconds. The hardware was a SUN 300MHz single processor Ultra SPARC-II with 128MB of RAM. The final column presents test results for the Inversion Algorithm[7]. The performance of the Inversion Algorithm and the EVCF Technique are virtually the same, but the EVCF is considerably more versatile than the Inversion Algorithm.

Circuit	Conventional Event-Driven	EVCF	Speedup	Inversion Algorithm
C432	10.8	1.4	7.71	1.1
C499	12.1	1.7	7.11	1.2
C880	20.2	4.0	5.05	3.2
C1355	43.2	5.6	7.71	5.3
C1908	82.5	8.1	10.19	6.4
C2670	89.3	13.6	6.57	11.5
C3540	128.5	15.3	8.40	12.0
C5315	252.9	27.5	9.20	24.2
C6288	2549.5	42.1	60.56	33.1
C7552	396.8	40.2	9.87	39.9

Figure 11. Experimental Data.

To eliminate the times for reading and writing vectors and other overhead, each test was run twice, once using fifty thousand random vectors, and again using fifty thousand vectors of all zeros. (Since the zero vectors have no activity, no simulation is performed.) The reported times are the difference of the two times.

6. References.

1. E. G. Ulrich, "Event Manipulation for Discrete Simulations Requiring Large Numbers of Events, " *JACM*, V.21, N.9, Sep. 1978, pp. 777-85.
2. Szygenda, S., D. Rouse, E. Thompson, "A Model and Implementation of a Universal Time-Delay Simulator for Large Digital Nets," *Spring Joint Computer Conference*, 1970, pp. 491-496.
3. 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.
4. D. M. Lewis, " Hierarchical Compiled Event-Driven Logic Simulation," *Proceedings of ICCAD-89*, pp.498-501.
5. 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.
6. M. Heydemann, D. Dure, "The Logic Automation Approach to Accurate Gate and Functional Level Simulation," *Proceedings of ICCAD-88*, pp. 250-253.
7. P. M. Maurer, "The Inversion Algorithm for Digital Simulation," *Proceedings of ICCAD-94*, pp. 259-61.
8. P. M. Maurer, "The Shadow Algorithm: A Scheduling Technique for Both Compiled and Interpreted Simulation," *IEEE Transactions on Computer Aided Design*, V11, No 12, Sept. 1993, pp. 1411-1413.
9. F. Brglez, Pownall, Hum, "Accelerated ATPG and Fault Grading via Testability Analysis," *ISCAS-85*, pp. 695-698.