Fanout Free Blocks and Multi-Delay Logic Simulation

Peter M. Maurer

Technical Report DA-29, 1995
VCAPP Laboratory
Dept. of Computer Sci. & Eng.
University of South Florida
Tampa, Florida 33620

D. Maclise, R.A.

FANOUT FREE BLOCKS AND MULTI-DELAY LOGIC SIMULATION

Abstract

The aim of the work described here to reduce the number of events processed during a simulation by clustering gates into groups, and scheduling the only the groups rather than the gates within them. Because a fanout free block represents a single complex function, these blocks were the units chosen for clustering. Two versions of a fanout-free block event driven multi-delay simulator have been developed, FFB and I-FFB. These have almost the same structure and internal representation, but the representation of the blocks slightly different. Detection of fanout-free blocks is identical for both simulators. The first algorithm, FFB, preserves the precise delay characteristics of the original circuit, but is advantageous only for small circuits. The second algorithm, I-FFB, alters the delay characteristics of the circuit somewhat, but is applicable to circuits of any size.

1. Introduction

Performance of event-driven simulation depends heavily on the number of events, which in turn also depends on the number of gates in the circuit. The number of events generated during simulation is based on the number of paths of differing lengths between primary inputs and input nets of the gate. The number of such paths in the multi-delay model[1,2] is much greater than that of unit or zero-delay simulation models[3]. The purpose of this research is to reduce the number of events by reducing the number of gates in the active queue and the number of potential events in the timing wheel.

The basic procedure is to group the gates in the circuit into the blocks, and reconstruct it using the blocks as basic elements while it preserving the circuit characteristics, such as path delay between primary inputs to the outputs of a gate. The detection of fanout-free blocks is based on the fanout-free net relation, which states that two gates are related if they are connected by a single two-terminal net. The fanout-free blocks of a circuit are the equivalence classes of the transitive closure of this relation. To illustrate the computation of the fanout-free blocks, consider the circuit illustrated in Figure 1. Initially, each gate is

assigned to its own block. Since gates *G1* and *G4* are related, the two blocks containing them are combined and treated as a single gate.

In Figure 1, even though *G1* and *G4* have been combined, the total delay between the inputs *A* and *C*, and the output *F* must be preserved. To accomplish this, the delays of a block are associated with the inputs of the block rather than with the block itself. Thus, the block consisting of *G1* and *G4* will have a delay of 3 associated with inputs *A* and *C*, and a delay of 2 associated with input *E*. Within the block boundary, gates can be simulated in zero-delay fashion.

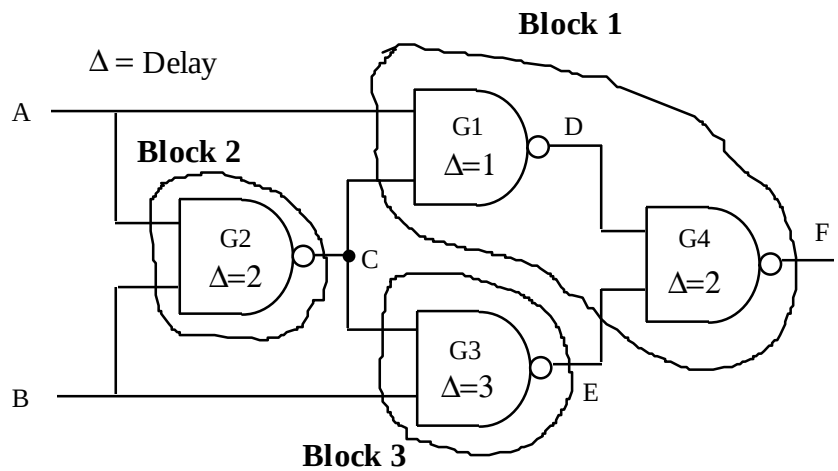

Figure 1. The First Stage of Block Detection.

At the next stage, additional blocks are examined and merged using the same relation. In Figure 2, since blocks 1 and 3 are connected by the two-terminal net *E*, they will be merged and form a new block with new input delays.

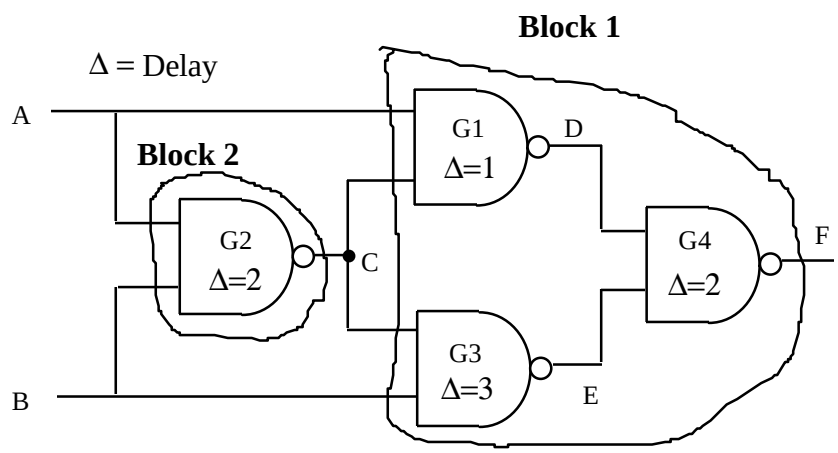

Figure 2. The Second Stage of Block Detection

After merging the blocks, the inputs, outputs and path delays of the new block are computed based on the characteristics of the component blocks. As shown in Figure 2, the delay of the net *B* is set to 5, and the net *C*, which constitutes two inputs to the block, is assigned two delays, 3 and 5. To accomplish this, the net *C* is split to two nets, C_1 with delay 3 and C_2 with delay 5. The merged block has 4 inputs and one output. Finally, the circuit now consists of two blocks instead of 4 gates. These blocks will be the basic units in the simulation procedure. In the original circuit, if an event occurs on net A, and the event is propagated to all gates, the gate-simulation sequence is *G1*, *G2*, *G4*, *G1*, *G4*, *G3*, *G4*. The final value of the primary output can be obtained by performing 7 gate simulations, while with fanout-free blocks, only by 3 block simulations are necessary: *B1*, *B2*, and *B1*. This is the major advantage of using fanout-free blocks. By representing the circuit as a collection of fanout-free blocks, the number of levels in the circuit is reduced[4].

2. Fanout Free Block Simulation

As with other event-driven simulators, the fanout free block simulator consists of two phases, the net evaluation phase and the block simulation phase. These two phases alternate during the simulation cycle. The input vector routine is executed at the beginning of each input vector followed by the net evaluation phase. A trailer routine is used to switch between the block simulation and the input vector routine. The input vector routine compares the primary inputs and inserts the input vector events into the timing wheel

at time zero slot. Then the net evaluation phase is activated. Net evaluation fetches the fanout blocks, and makes these active by placing them in the active queue. Once all events for a particular time slot have been processed, the active queue elements are fetched and simulated by their own individually generated block simulation routines. After processing all active block elements in the active queue, the trailer routine is activated. This routine checks the number of events in the timing wheel, and transfers to the net evaluation phase or to the input vector routine depending on the event count.

The timing wheel consists of *N+1* pointers, each of which points a stack of the events for one particular time slot. Each of the stacks in the timing wheel has a variable number of events in the form of a linked list. The size of *N* is determined by the maximum delay of the blocks in the circuit. The *N+1*st pointer does not point to an event list, but to the active block queue elements generated in the net evaluation phase. Figure 3 illustrates the structure of the simulator.

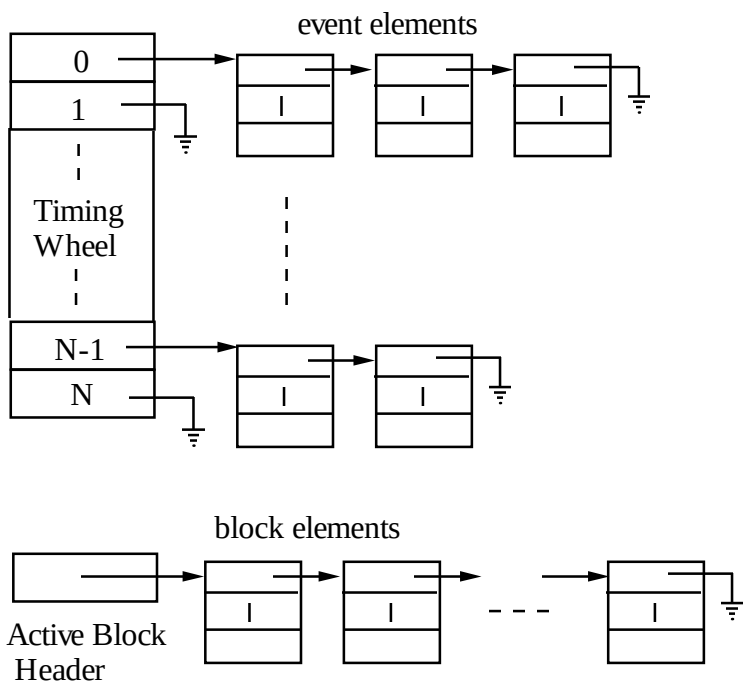

Figure 3. The Structure of the Timing Wheel

The block and event elements in Figure 3 are structures which carry the information for the simulator. The block elements consist of a link field, a pointer field pointing the corresponding block structure, and an

ID field for the block. The block structure is used to fetch block simulation routine address, and other required information. The ID field represents the input net that caused the block to activate, and is used to select delay path length within the block simulation routine. The event elements also contain a link field, a net structure pointer, and the net value computed by the block simulation routine.

The simulator generates the net and gate structures used to perform the circuit simulation. The net structure contains the current value of the net, the fanout count, the address of the fanout blocks, and the ID of each block. One generic net evaluation routine is generated which processes all net structures. All nets in the circuit, including those internal to blocks blocks, are mapped into individual net structures.

A gate structure contains the address of the corresponding simulation routine, a flag array, a count of input nets, and the addresses of the input nets. The size of array flag is determined by the number of input nets of the block. The flag indicates that the entry point of the block is in the active queue. These flags are mapped to the nets of the block, which in turn correspond to entry points of the block simulation routine. The block can be simulated more than once if the entry points are different for each simulation, because each entry point distinguishes the delay path length between the input net and the output net of the block. The input net entry points are fetched using the input net structure.

The structure of nets and gates in the fanout free block simulator are illustrated in Figure 4. These structures are maintained in a linked list, using the "next" field to perform insert and delete operations. In the net structure, the size of the IDs and Blocks fields are determined by the input count of the net. The input count specifies fanout in the net structure, and the number of input nets of the block in the block structure. For each input net, the IDs field contains the corresponding entry point of the block simulation routine, and the path delay through the block. The "Blocks" field contains pointers to each of the fanout block structures. In the block structure, the IDs pointer points the array of flags, each of which corresponds an entry point of the fanout-free block simulation code. These flags are used to determine whether the entry point is active. The "routine" field holds the starting address of the block simulation routine. Each block of the circuit has its own simulation routine, which is called when the block is active. A block simulation routine is illustrated in Figure 5.

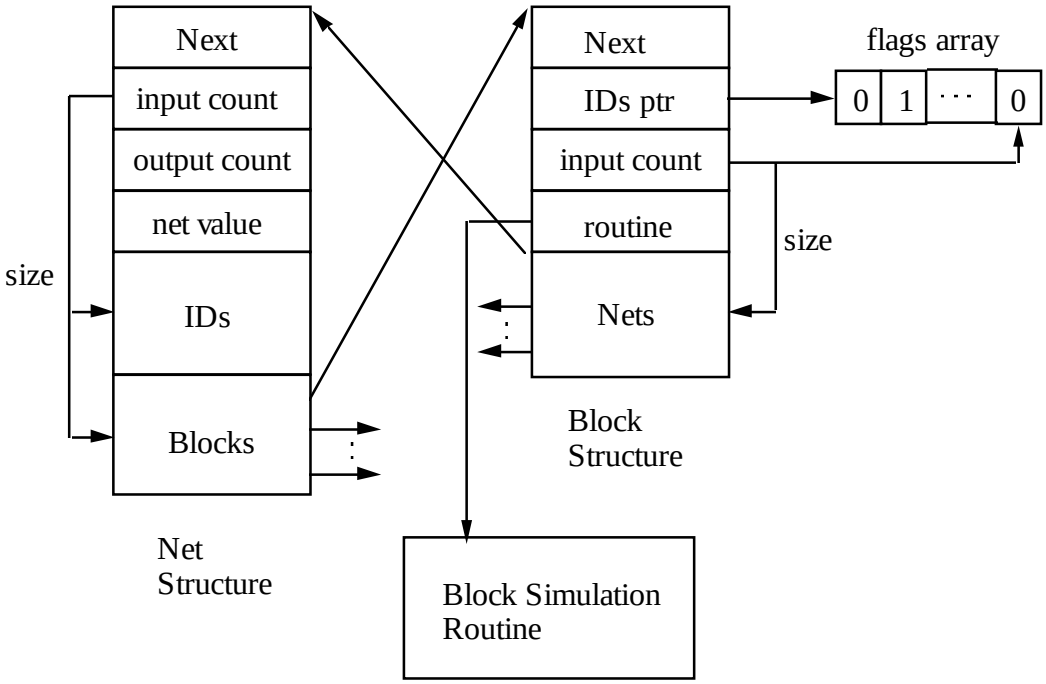

Figure 4. Net and Gate Structures

```
Block_X(id)
int id;

/* All gates function, within the block, are simulated. */
net[netid] := ~(net[netid].value & net[netid].value);
..

switch(id)
{
        case x:
           delay = X;  break;
        case y:
           delay = Y;  break;
        default:
           printf("No id found");   ERROR();
}
temp := net[netid].value & net[netid].value;
index := (current_time+delay) MOD sizeof(TimingWheel);
Put the temp value in the event queue element;
TimingWheel[index] := event;
increase event_count by one;
```

Figure 5. Block Simulation Routine

The block simulation routine simulates all gates included in the block, and then inserts the results into the timing wheel in the form of an event. Initially, a pool of free event elements is generated, and elements are fetched, one-by-one, from the pool. Each event element is inserted at the proper timing wheel index with result of a block simulation. The index of the timing wheel is computed by adding current time and delay of the path, and then reducing modulo the size of the timing wheel. The timing wheel size is computed by finding the maximum path length for any block in the circuit.

To reduce the size of the generated code and exploit any cache that might be available on the target system, only one generic net evaluation routine is generated. The net evaluation routine processes one event structure at a time. The new net value is compared with the previous value to determine whether an event occurred. If an event is detected, the fanout blocks of the net are fetched and placed in the active queue. The fetched blocks are examined by checking the id of the event net in the block. Events due to the same net will be discarded. If the block is activated due to different input net of the block, then the block becomes active twice (or more) by placing additional block elements in the active queue. The flag is then set to avoid making the same entry point active more than once during the same simulation cycle. The flag will be reset after executing the block simulation routine with the entry. One block element is allocated from a pool of free block queue elements, and it is inserted to the active queue. Like the event element, the block queue element consists of a link field, a pointer to the block structure, and the ID of the net which is the entry point of the block simulation routine. The generic net evaluation routine is illustrated in the Figure 6.

The net and block simulation routines are scheduled by the centralized main scheduler. This scheduler performs timing wheel management by updating the current simulation time and the corresponding time wheel slot, counting the events in the timing wheel in the future slots, managing the active queue elements, and jumping to block simulation routines.

```
Process_Net (net, value)
Net_Structure *net;
int        net_value;

if (net_value _ net->value)
{
        net->value := net_value;  /* update the new value */
        for(j=0; j < net->input_count; j++)
        {
                if (idarray[net->id] = 0)
                {
                        idarray[net->id] = 1;
                        Get block queue element;
                        block_element.id = net->id;
                        block_element.block = address of net->blocks[j];
                        Put the block_element in the Active queue.
                }
        }
}
```

Figure 6. Generic Net Handling Routine

The main scheduler consists of three major submodules: the general scheduler, the block scheduler, and the trailer routine. The general scheduler is the entry point of the main scheduler, it is invoked by the input vector routine. The input vector routine compares the primary input with the previous input, and if there are differences, it inserts the primary input events into the timing wheel at time slot 0. The input vector routine also increases the event count whenever it detects primary input vector events. After finishing the input vector comparison, the input vector routine invokes the general scheduler. The general scheduler checks the event count, and if it is not zero, it calls the generic net handling routine. As mentioned earlier, the net handling routine inserts blocks into the active queue if it detects events. When control returns from the net handling routine, the event elements for all processed events are released to the pool of free elements, and the event count is decremented. Simulation continues until the event count becomes zero.

The block scheduler is initiated when all events in the current time-slot have been processed. It checks the active queue, fetches the active queue elements, and calls block simulation routines. Each of these routines may generate new events, load the corresponding event structures into the appropriate timing wheel slots, and increase the event count. After control is returned from the block simulation routine, the

block scheduler resets the id of the id array in the block structure. The processed block queue element is deallocated, and the next active block is processed. When all active blocks are processed, trailer routine is activated. This routine increases the current time by one, and selects the next timing wheel slot. If the current time is larger than the size of the timing wheel, the modulo operation is applied to find the current timing wheel position. Finally, when the event count reaches zero, the trailer routine terminates the simulation of the current input vector. Control is returned to the input vector routine, which will initiate a new vector or terminate the entire simulation. The main scheduler of the fanout-free block simulator is illustrated in Figure 7.

```
Main_Scheduler:
        /* Net evaluation Phase */
        while (event_count > 0 )
        {
                Net_Handler(net,net_value);
                Deallocate current_ptr element;
                current_ptr := current_ptr->next;
                decrement event_count;
        }

Block_Scheduler:
        while (active_head _ NULL)
        {
                block = active_head->block;
                call (block->addr, block->id);
                Deallocate block from active_head;
        }

Trailer:
        current_time := (current_time+1) MOD size(TimingWheel);
        if (event_count = 0) then return;
        jump to Main_Scheduler;
```

Figure 7. The Main Scheduler Procedure

3. Experimental Results

Figure 8 summarizes the experimental results from the fanout free block multi-delay simulation. The circuits listed in Figure 8 are the ISCAS-85 combinational benchmarks which have become a standard for measuring the performance of logic simulators. Delays were added to each gate by randomly selecting a

delay value from 1 to 8 for each gate. Each circuit was simulated on 5,000 randomly generated vectors using the techniques described in the previous section, and by a conventional multi-delay interpretive simulator. The "gates" column indicates the total number of gates in the circuit, while the "blocks" shows the number of fanout free blocks for each stage of the compilation process. The number of fanout-free blocks of the second stage is the final number of the blocks. This number of block simulation routines is generated by the simulator.

ckt	gates	blocks		seconds	
		1st stage	2nd stage	FFB	interpreted
c432	160	96	60	26.1	35.9
c499	202	120	58	28.2	72.1
c880	383	217	105	28.3	55.9
c1355	546	392	258	159.8	196.0
c1908	880	507	377	210.5	219.5
c2670	1269	789	539	250.2	253.5
c3540	1669	962	555	548.5	467.6
c5315	2307	1397	806	705.8	644.6
c6288	2416	1936	1456	15214.0	12330.1
c7552	3513	2033	1331	1150.0	1019.0

Figure 8. The Experimental Result of FFB(Fanout Free Block)

The FFB simulator is an enhanced two valued logic simulator with the value X used only during the simulation of the first vector. The column represented in the "seconds" field is the seconds measured by the UNIX /bin/time command. This is the time required to complete the simulation, ignoring the time to read input vectors and write outputs, and is obtained by the arithmetic average of five separate runs.

The performance of the FFB(fanout free block) simulation is somewhat faster than the interpreted simulation when the circuit is relatively small, but is slower when the circuits become larger. For instance, in the c6288 circuit, the FFB simulator took 15,214 seconds, while the interpreted simulator took 12,330

seconds. This demonstrates that the FFB algorithm can be applied to relatively simple circuits, but needs more research to improve the performance for complex circuits.

4. Improved Fanout Free Block

The performance of the FFB algorithm can be improved by adding some restrictions. The block structure is described in the earlier section is retained, but each of the fanout-free blocks is assigned a single delay, instead of different delays for each input. The delay assigned to each block is the maximum delay between the inputs and the output of the block. Only a single flag is required to determine whether the block is in the active queue. If two or more input nets of a block have events simultaneously, only one of these will place the block in the active queue. This causes fanout-free blocks to be treated in much the same fashion as simple gates. In addition to collapsing the array of flags into a single flag, it is also possible to remove the multiple entry points to the block simulation routine. These changes also entail minor changes to the block structure.

Figure 9 illustrates the benchmark results of the improved fanout-free block (I-FFB) multi-delay simulator. The experiments were conducted in the same manner as those for the FFB algorithm. The number of generated blocks is the same for both algorithms.

ckt	gates	blocks		seconds	
		1st stage	2nd stage	I-FFB	interpreted
c432	160	96	60	10.9	35.9
c499	202	120	58	10.9	72.1
c880	383	217	105	16.4	55.9
c1355	546	392	258	54.1	196.0
c1908	880	507	377	85.9	219.5
c2670	1269	789	539	105.6	253.5
c3540	1669	962	555	189.3	467.6
c5315	2307	1397	806	245.7	644.6
c6288	2416	1936	1456	5403.3	12330.1
c7552	3513	2033	1331	420.8	1019.0

Figure 9. Experimental Result of I-FFB

The performance improvement can be computed by (time_interpreted - time_I-FFB)/ time_inpterpreted. The overall average performance improvement of the I-FFB is 65% over the interpreted simulator. This means that the I-FFB algorithms is roughly three times faster than the conventional logic simulator.

5. Conclusion.

Clustering circuits into fanout free blocks has been show to be an effective method for reducing the multi-delay simulation time of logic-level circuits, especially when each block can be assigned a uniform delay. When the original delays must be preserved, clustering is not advantageous for large circuits. However, the poor large-circuit performance obtained from this algorithm is primarily due to the necessity of associating a delay with each input net of a block. In simulations that already model input delays, this technique should prove to be advantageous. This research points out that gate-clustering is an option that should be provided by multi-delay simulators, to enable users to obtain enhanced performance for simulations that do not require the modeling of precise delays within a block. We believe that fanout free block clustering will be a good addition to the tools currently available to the designers of VLSI circuits.

6. References.

1. S. Szygenda, D. Rouse, E. Thomson, “A Model and Implementation of a Universal Time Delay Simulator for Large Digital Nets,” *Spring Joint Computer Conference, 1970*, pp.207-216.

2. P. Agrawal, W.J. Dally, W.C. Fisher, H.V. Jagadish, A.S. Krishnakumar, and R. Tutundjian, "MARS: A Multiprocessor-Based Programmable Accelerator," *IEEE Design and Test of Computers*, Oct. 1987, pp. 28-36.

3. M. Breuer, A. Friedman, “Diagnosis & Reliable Design of Digital Systems,” Computer Science Press, Woodland Hills, CA, 1976.

4. Z. Wang, P. Maurer, “LECSIM: A Levelized Event Driven Compiled Logic Simulator,” *Proceedings of DAC-27*, 1990, pp. 491-496.