Two New Techniques for Compiled Multi-Delay Logic Simulation*

Yun Sik Lee
Peter M. Maurer
Department of Computer Science and Engineering
University of South Florida
Tampa, FL 33620

Abstract

This paper describes two techniques for compiled event driven multi-delay logic simulation that provide significant performance improvements over interpreted multi-delay logic simulation. These two techniques are based on the concept of retargetable branch instructions that can be used to switch segments of code into and out of the instruction stream. Our second algorithm, called the shadow technique, has been designed especially for systems with instruction caches. Benchmark experiments shows these two techniques are up to 15 times faster than our interpreted multi-delay simulator, with an average improvement of about 5 times for our fastest method.

1. Introduction.

As the design of a circuit proceeds, more accurate simulation is required. In particular, more accurate models than zero or unit-delay timing are usually needed during the final phases of the design. The many well-known techniques for compiled simulation [1-6], are based on zero and unit-delay which do not provide an accurate picture of a circuit's timing behavior. For some circuit elements, delay is the essential nature of the function, and a accurate timing model is necessary to model them.

This paper focuses on multi-delay timing [7], in which the delay of each gate is an integral multiple of some basic unit. Delays may be the same for each instance of a gate type or delays can be assigned based on layout parameters. There are several types of delay which could be modeled, such as transport delay, which is the amount of time taken for changes in a gate's input to reach the output, ambiguous delay, which is a short interval in which a net is undefined, and rise-fall delay, which is the amount of time a signal takes to change from low to high and vice-versa[1]. For simplicity, our simulators deal only with transport delay, but could easily be extended to other delays.

The problem we encountered with compiled multi-delay simulation is that simple adaptations of unit-delay techniques have usually been unsuccessful in providing significant increases in performance. In Section 2 we describe work that we have done in oblivious multi-delay, simulation and show why it did not provide the expected performance improvements. In Section 3 we describe how we applied the concepts of threaded code[2] to multi-delay simulation, and why these were also unsuccessful. Sections 4 and 5 describe two new techniques that *do* provide significant performance improvements. The results for these two algorithms are presented in Section 6. During this research, we replaced our existing benchmark machine with a new, faster machine that also possesses an instruction cache. We discovered that locality of reference has a profound effect on simulators run on such machines, much more than we had anticipated. The algorithm presented in Section 5 was designed to provide good locality of reference at the expense of using somewhat slower operations.

2. Oblivious compiled simulation.

In both zero-delay and unit-delay simulation, compiled oblivious techniques have demonstrated performance improvements over interpreted event driven simulation, as long as the activity rate is above a certain threshold. The reason for this performance improvement is that the number of instructions executed per gate simulation is reduced to a minimum. Although oblivious simulatators usually simulate many more gates than event-driven simulators, the gate simulations are so efficient that a performance improvement can be realized.

However, the problem with oblivious techniques is that they must provide for every eventuality. In zero-delay simulation, this means simulating every gate for every input vector. However in unit-delay and multi-delay simulation this means the simulation of a gate at every time that an event-driven simulator could schedule the simulation. Thus if there is a path of delay d between the primary inputs and gate G, then G must be simulated at time d. The number of required simulations depends on the circuit structure, the the number of different delays, and the circuit size.

Two oblivious methods of unit-delay simulation that can be adapted to multi-delay are the PC-Set Method and the Parallel Technique[4,5]. We concentrated on the PC-Set Method, because it allowed us to pack several vectors

* This work was supported in part by the National Science Foundation under grant number MIP-906444 and the USF Center for Microelectronics Research (CMR).

into a single word, thus performing many simulations for the price of one. Our preliminary studies of the PC-Set method have suggested that oblivious simulation will not provide significant speedups over interpretive techniques for realistic circuits. In our studies, we used the ISCAS85 benchmarks, and assigned a random delay of 1-8 to each gate. Without vector packing, there was no significant difference between the performance of the PC-Set method and our interpreted simulator. Although vector packing gave a performance improvement of over 90%, similar improvements could probably be realized with the interpreted simulator. Furthermore, the size of the generated code was unacceptably large for many circuits, almost 50 megabytes for c6288. Experiments with random delays over a larger range of values have shown that the size of the generated code expands significantly with an increase in the number of delays.

Although the Parallel Technique has been shown to be substantially more efficient than the PC-Set Method, the it does not permit vector packing, and can probably not compete with an interpreted simulator using packed vectors. It seems that the number of gate simulations that must be performed by an oblivious multi-delay simulator is so large that little or no performance gains can be realized, even though the gate simulations can be done more efficiently than those of an interpreted simulator. Therefore, we have abandoned the oblivious approach and concentrated on event-driven techniques.

3. Threaded code techniques.

Lewis has demonstrated that using threaded code in a compiled unit-delay simulation can provide performance improvements over interpreted unit-delay simulation[2]. Apart from the distributed scheduler, the technique is much the same as that used by an interpreted simulator. Simulation is done in two phases, the event phase which processes events and produces a queue of gates, and the simulation phase which simulates gates and produces a queue of events. Because neither events nor gates are queued for more than one phase, event generation and storage is relatively simple, and it is possible to suppress event generation if a new value produced by a gate simulation identical to the existing value of the net.

Event-driven multi-delay simulation is also done in two phases, but because of different gates may have different delays, events must often remain queued for several iterations of the event phase. The event phase processes only those events for the current time. It is not possible to suppress event generation. If there is already an event queued for the net, it may change value before the value produced by the current simulation becomes effective.

Since only the events for the current time are processed during the event-phase, it is necessary to sort events by time. The simplest mechanism for doing this is the timing wheel, which is an array of event queues, each of which corresponds to a single time. If the number of queues is less than the total number of times at which events are processed, the queue may be reused by indexing the array modulo its size.

We have implemented a multi-delay simulator using threaded-code techniques similar to those used by Lewis[2]. This simulator provided an improvement of about 45% over our interpretive simulator. Although this is a significant improvement, it is less than we had expected. Subsequent investigations of the interpreted and compiled algorithms showed that the threaded-code technique was able to eliminate only a small part of the code used by the interpreted algorithm, hence the unexpectedly low performance.

Although multi-delay simulation demands the use of an event-driven algorithm, our experiments with threaded code demonstrate that a simple adaptation of an interpretive algorithm will not provide the significant performance improvements that we are seeking.

4. Gateways for multi-delay simulation.

We have recently investigated a concept called "gateways" that can be used to introduce event-driven behavior into oblivious simulations [6]. A gateway is a retargetable branch instruction that can be used to switch portions of the generated code into and out of the instruction stream. Although this concept is not directly applicable to multi-delay simulation because of the code size, we have discovered an adaptation of the gateway technique that can be used to substantially improve the performance of an event-driven multi-delay simulation. In the gateway technique, the branch target for a particular gateway is stored in a fixed location which are dynamically altered to construct an instruction stream. The generated code contains a simulation routine for each gate and a net handler for each net. Each routine is terminated by a gateway.

The primary difficulty in adapting the gateway technique to multi-delay simulation is that several events may be queued for a net at one time, making fixed branch targets infeasible. To get around this problem, our technique stores all branch targets in a collection of stacks. The gateway instructions pop the stack to obtain the address of the next block of code to be executed. (Lewis used a similar technique for unit-delay simulation in [2].) Our current implementation uses n+1 stacks where n is the number of slots in the timing wheel. The first n stacks implement the timing wheel, while the n+1st stack is used to schedule gate simulations.

The timing wheel structure is illustrated in Fig. 1, along with the structure for the gate-simulation routines. Pseudo-Code for the net and gate trailer routines is illustrated in Fig. 2. Gateway operations are represented by "jump *label" statement. Because this is not legal C, assembly language must be used to generate this operation, otherwise, C is generated.

Initially, the primary input tests, illustrated in Fig. 3, are used to detect changes in the input vector. In Fig. 3, "index" is the ID of a gate, which is assigned at compile

time. The flag is used to avoid switching the simulation routine into the instruction stream twice.

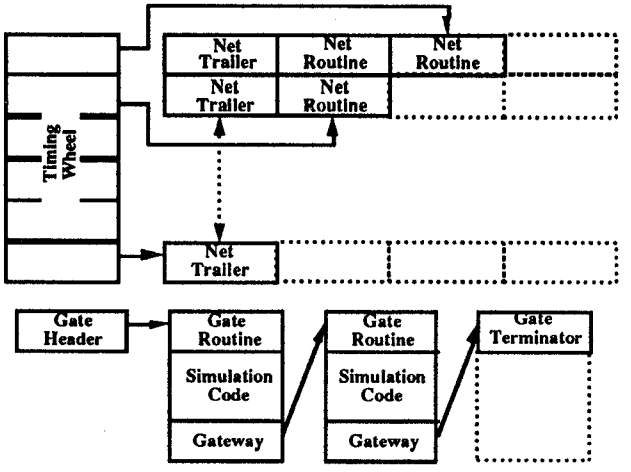

Fig. 1. The gateway stack structures.

```
Net_Trailer:
  jump *Gate_Header;  /* Gateway Operation */

Gate_Trailer:
  Restore Gate_Header to ADDRESS_of_Gate_Trailer;
  Clear TIMING_WHEEL[current_time];
  if event_count = 0 then
    return;
  Increment current time by 1;
  jump *TIMING_WHEEL[current_time].addr;
```

Fig. 2. Net and gate trailer routines.

Initially, the top of the simulation stack will cause an immediate exit. The branch address of the gateway instruction us used to form a singly linked list of simulation routines. The link address is conveniently located in the address field of a branch instruction..

After all input tests have been performed, the Net_Trailer branches to the first simulation routine in the stack. The last routine branchs to the gate trailer, which advances the current time, and jumps indirectly to the top of the new event stack.

```
/* index = fanout gate's ID at compiled time; */
  if (new_net_k != old_net_k) then
    if (flag[index] = 0) then
      old_net_k = new_net_k;
      Gateway[index] = Gate_Header;
      Gate_Header = ADDRESS_of_Net_k;
    endif
  endif
```

Fig. 3. An input test.

Four basic routines are used to simulate a single state at certain time *t*, the trailer routines illustrated in Fig. 2, and the gate-simulation and net-handling routines illustrated in Fig. 4. The trailer routines do the bridge operations between the net handlers and the gate simulations, simulation is performed by the gate-simulation routines, and scheduling is done by the net handlers.

```
/* Net handling routine */
Net_k_Handler:
  if (new_net_k != old_net_k) then
    old_net_k = new_net_k;
    if (flag[index] = 0) then
      Gateway[index] = Gate_Header;
      Gate_Header = Address_of_Gate_Routine;
    endif
  endif
  Decrement event count;
  Timing_Wheel[current_time] =
    Timing_Wheel[current_time] - Address_Size;
  goto *(Timing_Wheel[current_time]);
/* Gate simulation routine */
Gate_index_Simulator:
  Reset flag[index];
  new_net_k = old_net_x & old_net_y;
  Add_Time =( Current_Time + Gate_Delay) MOD
    Timing_Wheel_Size;
  Timing_Wheel[Add_Time] = Addr. of Net_k_Handler;
  Timing_Wheel[Add_Time] =
    Timing_Wheel[Add_Time] + Address_Size;
  Increment the event-count;
  goto *Gateway[index];          /* the next stream */
```

Fig. 4. Generated code structure.

5. The shadow technique.

The shadow technique improves performance on machines with instruction caches, because it generates a smaller amount of code than the gateway method. All information for a gate or a net is placed a descriptor called a shadow . All values are accessed indirectly using the shadow but, the reduced amount of code increases the performance of the instruction cache.

Only a minimal number of generic gate and net processing routines are generated, but a descriptor is generated for each gate and each net. A gate descriptor has the address of a simulation routine, a flag to prevent duplicate scheduling of gates, the gate delay, and pointers to input and output values. Each net descriptor has the address of a net handling routine, pointers to old and new values, and the addresses of fanout gate descriptors. Fig. 5 illustrates these descriptors.

Gate simulators are generated for each combination of gate-type and input-count in the circuit. Thus if the circuit contains only two and three-input NANDs, then two routines will be generated, one for two-input NANDs and one for three-input NANDs. Each of these routines assumes that a register variable called the "Shadow Pointer" contains the address of the current descriptor. One net-handler is generated for each different fan-out in the circuit.

Each of the routines terminates with load of the Shadow Pointer, and a branch to the routine of the new descriptor. The net handling routines use a timing-wheel that is identical to that used by the gateway algorithm, except the stacks contain descriptor addresses rather than routine

addresses. The gate-simulation stack is also similar to that used by the gateway algorithm, except that a linked list of descriptors is used instead of a list of routines.

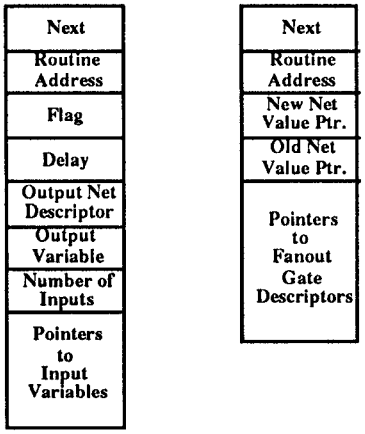

Fig. 5. Net and gate descriptors.

As usual, simulation is a two-phase process that alternates between gate-simulation and net-handling. At the end of each phase either a termination routine is executed, which is accessed through its own descriptor.

6. Experimental results.

A random delay from 1 to 8 was added to each gate of the ISCAS-85 benchmarks without regard to type. Because the same gates were simulated by all simulators, the range of delays has no effect on the conclusions drawn in this paper. Tests were run on a SUN 4-IPC with 12 megabytes of memory and a dedicated disk. The values of Fig. 6 are in seconds of execution time for 5,000 random vectors. The time required for reading inputs and printing outputs is excluded. Two interpreted simulators were designed for the purpose of comparing benchmark performance, one based on the enhanced two-valued logic model described below, and one based on a more conventional 3-valued logic model. The results for these simulators are reported in Fig. 6 under "2-Interp" and "3-Interp." The performance improvement percentages are computed using the results from the 2-valued simulator in the formula (interp_time - comp_time)/interp_time.

The logic model of our compiled simulators uses an additional bit to force every gate to be simulated at least once on the first input vector. The value of each net is initialized to 2 or 3, which represent values 0 and 1. The simulation routines set this second bit to zero, producing a value of either 0 or 1, allowing us to use a simpler 2-valued logic model, and still guarantee that nets are set to consistent values. It would be a simple matter to extend our simulators to 3-valued logic.

The Gateway Algorithm gives an average performance improvement of 67% running in about 1/3 the time of an interpreted simulation, while the Shadow Algorithm gives a performance improvement of 81%, running in about 1/5 the time of an interpreted simulation.

	Multi-Delay			
Ckt	2-Interp	Gateways	Shadows	3-Interp
c432	35.9	6.8	6.4	40.9
c499	72.1	12.4	10.3	83.9
c880	55.9	23.9	13.4	61.8
c1355	196.0	46.5	26.6	273.4
c1908	219.5	85.8	43.9	248.0
c2670	253.5	114.3	59.9	280.7
c3540	467.6	161.6	83.2	523.2
c5315	644.6	288.0	151.8	716.2
c6288	12330.1	1677.2	788.7	13850.5
c7552	1019.0	484.9	256.3	1122.4
MaxImprov.(%)		86	94	
MinImprov.(%)		52	75	
AvgImprov.(%)		67	81	

Fig. 6. Performance results.

The shadow algorithm executes essentially the same instructions as the gateway algorithm. On a machine with no cache, we would expect the gateway algorithm to perform slightly better than the shadow algorithm. The performance improvement in Fig. 6 is due entirely to the instruction cache. Because of the steady advance in the today's technology, we expect caches to become more common in the future. Many of today's compiled simulators do not perform well with caches, because the generated code is one long sequence of instructions without loops or branches.

7. Conclusion.

Our experimental results show that the average performance improvement over interpreted simulation is 67% for the gateway algorithm and 81% for the shadow algorithm. This research demonstrates the importance of designing compiled simulators that work well with instruction caches. The differences between the gateway and shadow algorithms demonstrate the effect that locality of reference can have on simulation performance.

References

1. M. Breuer and A. Friedman, *Diagnosis & Reliable Design of Digital Systems*, Woodland Hills, CA: Computer Science Press, 1976.
2. D. Lewis, "A Hierarchical Compiled Code Event-Driven Logic Simulator,", *TCAD*, vol 10, 1991, pp. 726-737.
3. R. Bryant, D. Beatty, K. Brace, K. Cho and T. Sheffler, "COSMOS: A Compiled Simulator for MOS Circuits", *DAC- 24* 1987, pp. 9-16.
4. P. Maurer and Z. Wang, "Techniques for unit-delay compiled simulation", *DAC-27*, 1990, pp. 480-484.
5. P. Maurer, "Optimization of the Parallel Technique for Compiled Unit-Delay Simulation," *ICCAD-90*, 1990, pp. 70-73.
6. P. Maurer, "Gateways: A Technique for Adding Event-Driven Behavior To Compiled Unit-Delay Simulations," Submitted for Publication.
7. S. Szygenda, et. al, "A Model and Implementation of a Universal Time Delay Simulator for Large Digital Nets," SJCC, 1970, pp. 207-216.