Two New Techniques for Compiled Multi-Delay Simulation

Yun Sik Lee

Technical Report DA-7, 1991, VCAPP Laboratory

Dept. of Computer Sci. & Eng., University of South Florida
Tampa, Florida 33620

TWO NEW TECHNIQUES FOR COMPILED MULTI-DELAY SIMULATION

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

TWO NEW TECHNIQUES FOR COMPILED MULTI-DELAY SIMULATION

ABSTRACT

Two new techniques for compiled multi-delay simulation are presented, one which is event-driven and another which is based on the concept of levelized compiled simulation. Experimental results are presented which show a significant performance improvement for compiled event-driven simulation over interpreted event-driven simulation, although this improvement is somewhat less than would normally be expected. An analysis of both the compiled and interpretive simulators is presented that supports the experimental data. The effects of caching and locality of reference are presented for the compiled event-driven simulator. The performance enhancements for the non-event-driven technique are substantial, but this technique has the disadvantage of generating an enormous amount of code for some circuits. Suggestions for future research are also presented.

TWO NEW TECHNIQUES FOR COMPILED MULTI-DELAY SIMULATION

1. Introduction.

The problem of compiled simulation has been the subject of much recent research [1-8]. The primary reason for this interest is that compiled simulation promises, in many cases, to provide significant performance improvements over interpreted event-driven simulation. Much work has focused on the zero-delay timing model [4-8], because this model is one of the simplest to implement, and also tends to provide the best performance improvements. There has also been much work in the area of compiled unit-delay simulation [1-3], because the unit-delay model is considered to be a more accurate representation of a real circuit than the zero-delay model, however, the unit-delay techniques tend to be more complex than those for the zero-delay model.

Existing work in compiled simulation falls into two broad categories which can be termed "event-driven"[1,2,8] and "oblivious."[3,6,7] In event-driven simulation, the number of gates simulated for a particular input vector depends on the content of the vector, while in oblivious simulation the number of gates simulated per vector is constant. Although we are using the term "event-driven" in a broader sense than usual, virtually all the techniques in question are literally driven by events, so no confusion should arise. (See, however, [8].)

In a sense, multi-delay simulation [9] represents both the next "level of accuracy" and the next "level of difficulty" in logic simulation. Because multi-delay simulation can be defined in different ways, it is necessary to clarify how the term is used here. In this paper it is assumed that every gate has an integer delay which is determined externally to the simulation. (That is, 2 and 5 are permissible delays, but 2.5 is not.) No restrictions are placed on the size of the delay (but see Section 6), nor is it necessary for two gates of the same type to exhibit the same delay. Event scheduling for multi-delay simulation is more complex than for unit-delay simulation, and is, in a sense, more unpredictable. In most cases, the number of times at which a net may change value is much larger than for unit-delay simulation, so oblivious techniques such as the PC-set method [3] must perform many more gate simulations.

Section 2 describes our compiled multi-delay event-driven simulator, while Section 3 compares the our compiled simulator with the compiled simulator. Section 4 describes the effects of running the compiled event-driven simulator with a cache. Section 5 describes

our oblivious compiled multi-delay simulator. Section 6 describes our experimental procedure while Section 7 draws conclusions.

2. Event Driven Compiled Simulation.

Because event-driven multi-delay simulation is quite similar to event-driven unit-delay simulation, similar techniques can be used to attack both problems. However, there are important differences between the two types of simulation. Event-driven unit-delay The event-phase processes all queued events and produces a queue of gates. The gate-phase simulates all queued gates and produces a queue of events. Because neither events nor gates must be queued for more than one phase, event generation and storage is relatively simple. Furthermore, it is possible to suppress the generation of an event in the gate-phase if the new value produced by the simulation of a gate is identical to the existing value of the net. This reduces the amount of work that must be done by the event-phase.

Event-driven multi-delay simulation is also a two phase process, but because different gates may have different delays, events must often remain queued for several iterations of the event-phase. The simulator keeps track of the current time, and the event-phase processes only those events that correspond to that time. It is not possible to suppress event generation in the gate-phase by simply comparing the new value of a net to its current value. If there is an event already queued for the net, the value of the net may change before the value produced by the current simulation becomes effective. Figure 1 illustrates this point. If the two inputs are changed from 0 to 1, simple event suppression will cause the output of the circuit to be erroneously set to 1. More complex event-suppression tests may be used, but if these tests are designed to be 100% effective in suppressing unnecessary events, they may end up consuming more time than simply allowing the event to be generated and suppressing its effect when it is processed.

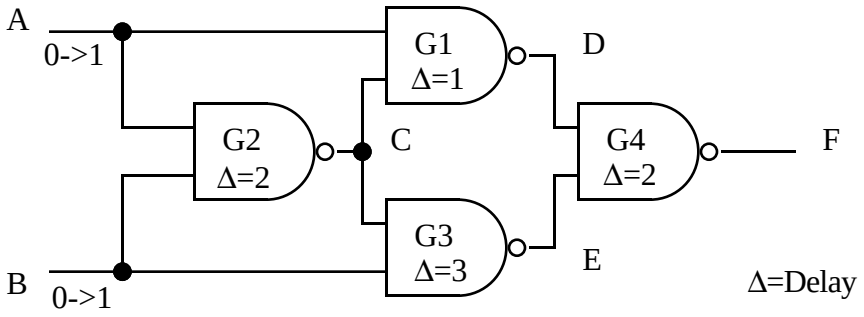

Figure 1. A Multi-Delay Example.

Since only those events that correspond to the current time are processed during the event-phase, it is necessary to sort events according to time. The simplest mechanism for performing such sorting is the timing wheel, which consists of an array of event queues, each of which corresponds to a single time. The current time is used as an array index to select the proper event queue for the current time. If the number of queues is less than the total number of distinct timing points at which events are processed (which is almost always the case), the queues may be reused by indexing the array modulo its size. If the number of queues is less than the maximum delay of any gate in the circuit, a linear overflow queue will be required.

Wang [8] used an approach similar to the timing wheel to schedule gates for a zero-delay event-driven simulation. (Events are not required in this algorithm.) The circuit was first levelized, and one gate-queue was allocated for each distinct level in the circuit. Each gate-queue was of fixed size, and each gate could be assigned to one and only one gate queue. The total number of slots in all queues was equal to the number of gates in the circuit. Although we considered using fixed sized queues for multi-delay simulation, this turned out not to be desirable. The maximum number of events that can be generated during the simulation depends on the circuit structure and the number of different delays that appear in the circuit, not just on circuit size. Another alternative would have been to use a timing wheel with fixed size queues. However, in this case each queue would need to contain one entry for each net in the circuit. (It may be possible to reduce this requirement, but probably not by a significant amount.) Even though this approach appears to be feasible, we have observed that most circuits generate far fewer than the theoretical maximum number of events even when tested with randomly generated vectors. (For the circuits we tested, only 5 to 10% of the maximum number of events were generated.)

Because fixed-length event queues would consume an enormous amount of space, most of which would be unused, we elected to implement each queue as a variable-sized linked list. Because dynamic allocation of storage can consume hundreds of instructions, it is necessary to allocate event structures at compile time. Initially, three event structures are allocated per net. (These are general-purpose structures, not dedicated to any particular net.) Should this prove to be insufficient, more structures can be allocated at run time. The event queues are organized as stacks, while the free event structures are saved in yet another stack.

The event-phase of the algorithm is performed by a collection of independent event-processing routines, while the gate-phase is performed by a collection of gate simulation routines. One gate simulation routine is generated for each gate, and one event processing

routine is generated for each net. These routines are executed in a threaded-code manner [10] similar to that used by Lewis [2]. When an event is added to an event queue, the address of the event handler is inserted into the event structure along with the new value of the net. The structure of these queues is illustrated in Figure 2.

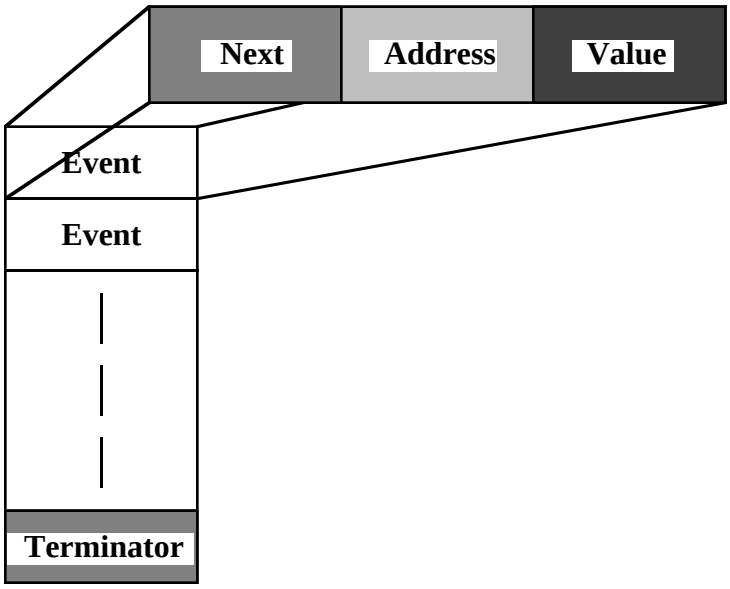

Figure 2. The Event Queue.

The event-phase of the algorithm consists of selecting the queue that corresponds to the current time, and executing all the routines found there. Instead of being called by a centralized scheduler, these routines branch to one another in chain-like fashion. When an event processing routine is executed, it first pops the event queue, then compares the current value of the net with the new value that was obtained from the queue entry. If the two values differ, it updates the current value of the net and inserts all gates in the fan-out set of the net into the gate queue for simulation. (A flag is used to avoid placing any gate in the queue more than once.) Finally the event handler branches to the next routine in the event queue. Figure 3 illustrates the operations performed by the event handler. For clarity, the testing and setting of the gate queue flags has been omitted from this figure.

```
new_value = TIMING_WHEEL[current_time]↑.value;
POP TIMING_WHEEL[current_time];
if new_value ≠ net_value then
        net_value := new_value;
        PUSH gate_1;
        PUSH gate_2;
        …
        go to TIMING_WHEEL[current_time]↑.routine;
endif
```

Figure 3. The Event Handler.

The final entry in each event queue is a queue termination routine which consists of a simple branch to the first gate simulation routine. The queue entry for the termination routine is never deleted.

When a gate simulation routine is executed, it first pops the gate queue. It then simulates the gate using the current values of the inputs and stores the value of each output net in a temporary variable. It then creates an event for each output net and inserts them into the proper event queue. The delay of the gate is added to the current time and reduced modulo the size of the timing wheel to determine the index of the queue into which the events will be inserted. Event suppression is not performed at this point. Finally, the gate simulation routine branches to the next routine in the gate queue. The code for the simulation of a 2-input AND gate is illustrated in Figure 4. For simplicity, a two-valued simulation of the gate is illustrated in Figure 4, although the simulator actually performs a three-valued simulation.

```
POP GATE_QUEUE;
temp_value := NET_X & NET_Y;
new_queue := ( current_time + gate_delay ) MOD number_of_queues;
PUSH NET_Y_HANDLER and temp_value onto
      TIMING_WHEEL[new_queue];
go to GATE_QUEUE↑.routine;
```

Figure 4. The Gate Simulator.

The final entry in the gate queue is also a queue termination routine, but this routine is somewhat more complex than that for the event queues. The gate-queue terminator first checks to see if there are any events queued in the timing wheel. (The gate-simulation and event-processing routines maintain a count of the number of events currently in the timing wheel. For simplicity this code has been omitted from Figures 3 and 4.) If there are more events to process, the gate-queue termination routine updates the current time by 1 (reduced

modulo the number of queues), selects the new event queue, and branches to the first event-handler in the new queue. If there are no events to process, the gate-queue termination routine returns control to the vector input routine, which will either read the next vector or terminate the entire simulation. The queue entry for the gate-queue termination routine is never removed.

To begin the simulation of a new input vector, the vector input routine sets the current time to zero, and then reads a new vector. It then compares the current values of the primary inputs to the values obtained from the current input vector. If the values differ for a particular input, an event is queued for the net at time zero. Finally, the vector input routine then selects the time-zero event queue and branches to the first routine it contains.

Figure 5 contains the results of our testing the compiled simulator on the first eight ISCAS-85 benchmarks using a SUN-3/280 Processor. (See Section 6 for a description of the experimental procedure.) These results show that, on this processor, there is a relatively constant performance increase of 44-45% over interpreted simulation for all benchmarks. Although this is a significant improvement, it is somewhat less than what we had expected. Section 3 contains our analysis of the reasons for the observed performance improvement.

Event-Driven (SUN 3 Data)			
	gates-simulated	interpreted	compiled
c432	1,276,897	163.0	89.9
c499	2,766,217	317.2	175.4
c880	2,009,913	241.0	133.0
c1355	7,659,897	855.7	464.8
c1908	8,451,813	954.1	521.8
c2670	9,389,592	1044.4	572.0
c3540	18,443,225	1964.3	1064.4
c5315	24,007,199	2629.9	1415.1

Figure 5. SUN-3 Performance Data.

Figure 6 contains data similar to that pictured in Figure 5, except that these results were obtained on a SUN-4/IPC. Surprisingly, much of the observed performance improvement disappeared on the SUN-4. This was due, at least in part, to the SUN-4 cache, and the poor locality of the code generated by the compiled simulator. Figure 6 shows the results of the compiled simulator with no changes, and with changes designed to improve locality. (These changes are detailed in Section 3.) Unfortunately, the changes to improve locality also slowed the simulator down. The overall performance improvement on the SUN-4 processor, after all changes, was about 23%.

Event-Driven (SUN 4 Data)				
			compiled	compiled
	gates	interpreted	no locality	with locality
c432	1,276,897	40.9	31.1	29.6
c499	2,766,217	83.9	71.5	61.3
c880	2,009,913	61.8	59.6	45.1
c1355	7,659,897	273.4	196.1	175.1
c1908	8,451,813	248.0	234.6	188.9
c2670	9,389,592	280.7	289.1	215.5
c3540	18,443,225	523.2	520.8	396.3
c5315	24,007,199	716.2	**	551.2
c6288	466,079,322	13850.5	**	10660.9
c7552	37,756,914	1122.4	**	1017.9

Figure 6. SUN-4 Experimental Data.

3. Performance Analysis.

To understand the reasons for the observed improvement in performance between the compiled simulator and the interpretive simulator, it is necessary to take a close look at the interpreted simulator and how it differs form the compiled simulator. The interpreted simulator was specially constructed for the purpose of comparing compiled and interpreted event-driven multi-delay simulation, so the two simulators are very similar. As in the compiled simulator, the timing wheel is pre-allocated dynamically to size, and each queue in the timing wheel is implemented as a stack. These stacks are implemented as singly linked lists of event structures, which are also pre-allocated and placed on a free queue for later use by the gate-simulation routines. The management of the event structures is identical in the two simulators. The simulation algorithm used by the interpreted simulator is pictured in Figure 7.

```
1   while there are events in the TIMING_WHEEL do
2           while TIMING_WHEEL[current_time] is not empty do
3                   temp_value := TIMING_WHEEL[current_time]↑.new_value;
4                   temp_net := TIMING_WHEEL[current_time]↑.net_structure;
5                   POP TIMING_WHEEL[current_time];
6                   if temp_net↑.value ≠ temp_value then
7                           for each fan-out pointer P in temp_net do
8                                   PUSH P onto GATE_QUEUE;
9                           endfor
10                  endif
11          endwhile
12          while GATE_QUEUE is not empty do
13                  temp_gate := top of GATE_QUEUE;
14                  POP GATE_QUEUE;
15                  CASE temp_gate↑.type;
16                  …
17                  if AND:
18                          result := VALUE OF FIRST INPUT;
19                          for each subsequent input I do
20                                  result := result & VALUE OF I;
21                          endfor
22                  …
23                  ENDCASE;
24                  compute new_queue, and push an event
25                          onto TIMING_WHEEL[new_queue]
26                          for each output of temp_gate;
27          endwhile
28          current_time := (current_time + 1) MOD size of TIMING_WHEEL;
29  endwhile
```

Figure 7. Pseudo-Code for interpretive multi-delay simulation.

As Figure 7 shows, the interpretive algorithm performs much the same operations as the compiled simulator, but using generic routines rather than routines that are dedicated to specific gates and nets. Gates and nets are represented as data structures rather than as executable code. These data structures are illustrated in Figure 8. A net structure contains both the current value of the net and its fan-out list, which is an array of pointers to gate structures. A gate structure contains the gate-type and the input and output lists of the gate, which are arrays of pointers to gate structures.

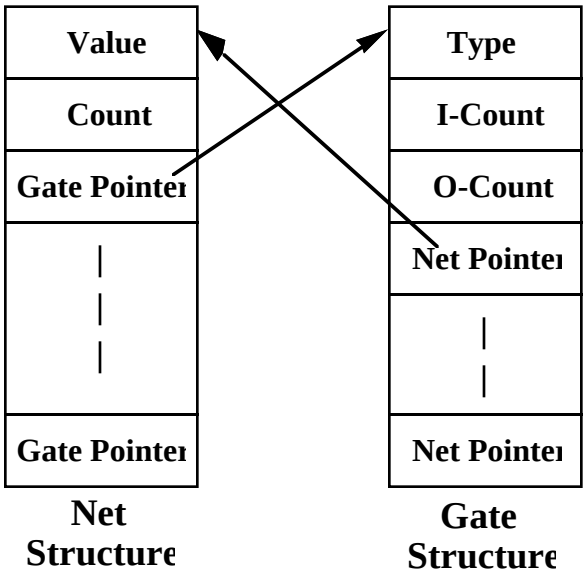

Figure 8. Structures for the interpretive simulator.

The major differences between the interpretive simulator of Figure 7 and the compiled simulator are as follows.

1. The loop on line 2 is partially unrolled. Only the test is eliminated, since the final branch of the loop is replaced by the final branch of each gate-simulation routine. Furthermore, the branches in the compiled code are *indirect* branches while those in the interpretive code are direct.
2. The loop on line 7 is completely unrolled. All tests and branches are eliminated. One level of indirection is eliminated in generating events, since in the compiled code net-routine addresses are constants, while in the interpretive code net-structures must be accessed through a gate structure.
3. The loop on line 12 is partially unrolled in a fashion identical to that of line 2.
4. The CASE statement on line 15 is eliminated. Since the C compiler implements this statement as a branch table, only about half a dozen instructions are saved.
5. The loop on line 19 is completely unrolled. All tests and branches are eliminated. Two levels of indirection are eliminated in accessing net values. Note that in both the compiled and interpreted simulator, the simulation three-valued, and hence significantly more complicated than the two-valued simulations illustrated in Figures 4 and 7.

6. The implied loop on lines 24-26 is completely unrolled. All tests and branches are eliminated. One level of indirection is eliminated in placing items on the gate stack. Note that since both simulators deal only with single-output gates, that no loop is actually required in the current version of the interpreted simulator.

This analysis makes it clear that the main advantages of the compiled simulator are the unrolling of loops and the elimination of indirect accesses to various data items. It is also clear that a significant portion of the code is retained by the compiled simulator. Although we have not counted each instruction in the two implementations, we feel that the improvement in performance observed on the SUN-3 processor is consistent with the amount of code eliminated by the compiled simulator.

4. Cache Effects.

As noted above, transfering the compiler from a SUN-3 to a SUN-4 processor virtually eliminated the performance improvement observed on the SUN-3. This was quite surprising, since we had expected roughly similar results on the two processors. The loss in relative performance turned out to be attributable, at least in part, to the SUN-4 cache and the lack of locality in the code generated by the compiled simulator. Furthermore, the code generated by the compiled simulator is not executed in straight-line fashion, so the compiled code could not benefit from automatic cache pre-paging, were it available. Because of the steady advance in microprocessor technology, we expect caches to become more common in the future rather than less so. This represents an important consideration for many compiled simulators, which typically exhibit very little locality of reference.

To deal with this problem in our compiled multi-delay event-driven simulator we redesigned the generated code to use generic gate-simulation routines rather than a separate routine for each gate. One generic routine was generated for each gate-type such as NAND and NOR. Generic event-processing routines were also generated for each different fanout-count encountered in the circuit. Thus if the maximum fan-out of any gate was three, at most four generic routines would be generated, including the zero-fanout routine.

Gate and event queues were modified to include pointer to a gate or net descriptor as appropriate. A gate descriptor contains a pointer to each input value as well as a pointer to the net descriptor for each output. A net descriptor contains a pointer to the descriptor of each of its fan-out gates. These descriptors, which are quite similar to those used by the interpretive simulator, are illustrated in Figure 9.

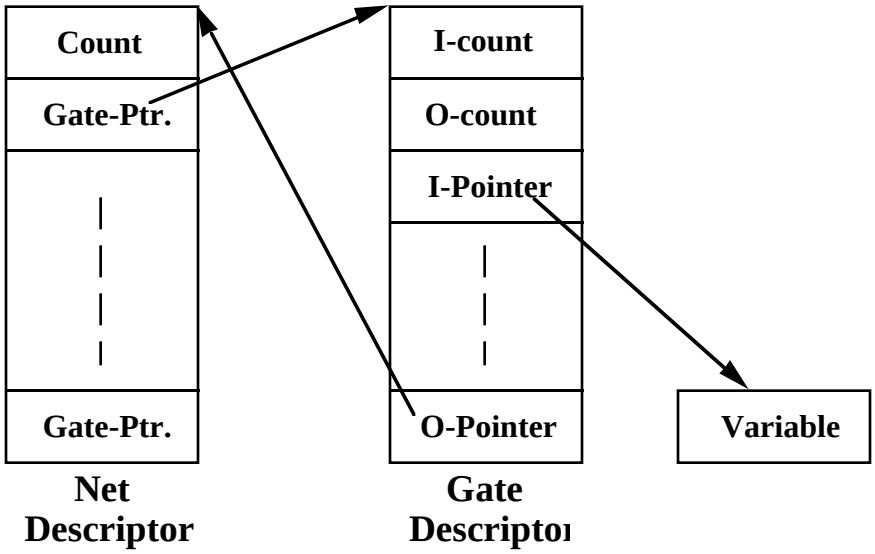

Figure 9. Gate and Net descriptors.

It is important to note that in the interest of locality of reference, we have sacrificed many of the gains detailed in the previous section. Indirect references have been reintroduced, and the gate-evaluation loops have been "re-rolled." (The unrolling could be preserved by generating more generic evaluation routines, but this sacrifices some locality of reference.) These changes enabled us to obtain a performance improvement of around 23% on the SUN-4 processor.

5. Oblivious Compiled Simulation.

Our results on the SUN-4 led us to explore the idea of oblivious compiled multi-delay simulation. We had avoided oblivious methods in favor of event-driven methods because in oblivious simulation one must "provide for every eventuality." For multi-delay simulation this means simulating a gate at every time point at which an event-driven could possibly schedule the simulation. Put more simply, if there is a path of delay *d* between the primary inputs of a circuit and a gate *g*, then *g* must be scheduled at time *d*. Obviously the number of simulations that would be required depends on the circuit structure and the number of different delays as well as on the size of the circuit. For some circuits, such as the ISCAS-85 c6288 benchmark, this implies that the generated code will be truly enormous. However, some forms of oblivious simulation allow multiple input vectors to be packed into a single word, thus allowing many simulations to be done for the price of one.

One such technique is the PC-set method of unit-delay compiled simulation [3], which is easily adaptable to multi-delay simulation. This technique has the disadvantage that it is suitable only for synchronous circuits, and that it uses a two-valued logic model, however it could easily be adapted to other logic models, and the restriction to synchronous circuits is not especially restrictive. The first step in this technique is to determine the PC-set of each gate, which is the list of times at which the gate could be scheduled. First, each primary input is assigned the set {0} as its PC-set. If it is desirable to model input delays, a PC-set containing all possible input delays for the primary input replaces the set {0}. Next, the circuit is levelized and gates are processed in levelized order. If the circuit is cyclic, every cycle must contain a flip-flop. The inputs of the flip-flop are treated as primary outputs and the outputs are treated as primary inputs. The PC-set of a gate is computed by forming the union of the PC-sets of its input nets, and incrementing each element of the union by the delay of the gate. This new set is then assigned to the gate and all of its outputs. If there are wired AND or OR connections in the network, the PC-set of the wired connection is the union of the PC-sets of the gates driving the connection. Once all PC-sets have been computed, code is generated in levelized order. One gate simulation is generated for each element of the PC-set of a gate. The method for determining the inputs of each gate simulation is essentially the same as that outlined in [3].

For zero-delay simulation, it is possible to pack vectors into words in more-or-less arbitrary fashion, as long as all bits corresponding to a particular vector end up in the same bit-position. However, in multi-delay simulation there are often dependencies between successive vectors. To be specific, the value of a particular net at a particular time may depend not only on the current input vector, but also on the previous vector. To preserve the sequentiality of a set of input vectors, it is necessary to pack them vertically rather than horizontally. For example, suppose a circuit has a single input, and that it is necessary to pack 16 single-bit input vectors into four 4-bit words. For clarity, we will designate the vectors with the letters A through P rather than with actual bit values. Horizontal packing will pack the first four vectors into the first word, the next four into the second word, and so forth, resulting in the following four words.

```
ABCD
EFGH
IJKL
MNOP
```

If these four words were used in a multi-delay simulation, Vector H would be simulated on the results of Vector D, not on the results of Vector G, as required. Vertical packing, on the other hand, gives the following four words vectors.

```
AEIM
BFJN
CGKO
DHLP
```

This set of vectors gives the proper ordering for G and H, but E is not simulated on the results of D as required. To force proper ordering between the vectors in successive columns, we add an additional initialization vector at the beginning of the vector which will initialize all nets to their proper values. The initialization vector is created by shifting each word of the final vector to the right one bit, resulting in the following.

```
xDHL
AEIM
BFJN
CGKO
DHLP
```

It must be emphasized that vector packing of this nature generally cannot be used with sequential circuits, because the state of the circuit for a particular input vector may depend on the entire sequence of vectors preceding it. Sequential circuits can, however, benefit from vertical vector packing, as long as the sequence of vectors in one bit position is independent of the sequence of vectors in any other bit position. Since the testing of any circuit requires many independent tests to be run, supplying such vectors will normally not be a problem. Since all of our benchmark tests were combinational circuits, we chose to pack our vectors in the above fashion, with 32 vectors per word.

As Figure 10 illustrates, oblivious simulation along with vector packing provides a significant speedup over the interpretive simulation. (See Section 6 for a description of the experimental procedure.) For all circuits we tested, the performance improvement was over 90%, sometimes as high as 98%. To gauge the importance of vector packing for oblivious simulation, we multiplied the CPU-Sections for the packed vectors by 32 to obtain a rough estimate of how much time would be required for unpacked vectors. As can be seen from Figure 10, these times are comparable to those obtained for event-driven compiled simulation. In one case, c6288, the size of the generated code was too large to be run on our test system, which points out the need for techniques that reduce the size of the generated code. It should also be noted that the generated code was executed in straight-line fashion with virtually no locality of reference. Thus there may be potential for additional improved performance if locality can be improved without sacrificing too much performance.

Oblivious Simulation (SUN-4)				
	gates/vector	CPU-seconds	Gates-simulated	Unpacked-Time (Projected)
c432	3,945	2.1	623,310	67.2
c499	3,383	1.6	534,514	51.2
c880	8,018	3.8	1,266,844	121.6
c1355	27,702	13.1	4,376,916	419.2
c1908	31,607	10.4	4,993,906	332.8
c2670	21,778	10.1	3,440,924	323.2
c3540	59,254	16.4	9,362,132	524.8
c5315	54,671	17.1	8,638,018	547.2
c6288		**	**	**
c7552	89,137	26.4	14,083,646	844.8

Figure 10. The Results for Oblivious Simulation.

6. Experimental Procedure.

All experiments reported in Figures 5, 6, and 10 were run using the ISCAS-85 combinational benchmarks, which have been used by several researches to measure the quality of both logic simulation and automatic test generation. These benchmarks were converted to our local net-list language [11], and a random delay from 1 to 8 was added to each gate, without regard to gate type. Because our net-list language is incapable of handling feed-throughs (a primary-input/primary-output pair with no intervening gate), non-inverting buffers were added to two of the circuits to model the feed-throughs. These added buffers were also assigned random delays.

The SUN-3 experiments were run on a SUN-3/280 file server. Although this machine could not be isolated while the experiments were performed, each experiment was performed five times and the results were averaged to minimize the error in the reported results.

The SUN-4 experiments were run on a SUN-4/IPC with 12 megabytes of memory and a dedicated disk drive. This system was dedicated during these experiments, but could not be completely isolated. As with the SUN-4 experiments, each experiment was run five times and the results were averaged to obtain the numbers reported here.

All numbers were obtained using the UNIX /bin/time command. The numbers reported here do not include the time required to read input vectors or print output. This was accomplished by running three versions of each simulator (each five times), the first of which performed an ordinary simulation. The second version was identical to the first, but with all calls to output routines suppressed. The third version was identical to the second, but with all calls to the simulation routines suppressed. (Reading of vectors proceeded

normally.) The numbers reported here were obtained by subtracting the averaged CPU time for the read-only version from the averaged CPU time for the non-printing version.

7. Conclusion

Although the performance improvements reported here are substantial, there is still considerable room for improvement. There are a number of minor enhancements that can be made to our compiled event-driven simulator to improve its performance. Although many of these enhancements can also be applied to our interpretive event-driven simulator, the should improve the *relative* performance of the compiled simulator (because a constant amount of code has been subtracted from the interpretive simulator.

The performance improvements given by the PC-set method are impressive, but this technique also has some serious drawbacks. It is vitally important to find methods of reducing the amount of generated code if this technique is to be of value for a wide class of circuits. It is also important to note that much of this performance improvement is due to vector packing, so techniques that reduce the amount of generated code must preserve this property.

This research also points out the problem of dealing with cache effects in compiled simulation, a problem that has not, as yet, received wide attention. It is clear that this issue cannot be ignored in future work.

Although the results presented in this paper are in no way preliminary, our research in compiled multi-delay simulation is still in its earliest stages. We are currently considering a number of different approaches to solving the problems of reducing the size of the generated code in both event-driven and oblivious techniques, reducing the number of gate simulations required by oblivious techniques, and increasing the locality of both approaches without sacrificing too much performance. It is clear that there is much work yet to be done. The results presented here represent a fundamental first step, and suggest the directions in which future work in this area ought to proceed.

REFERENCES

1. Bryant, R. E., D. Beatty, K. Brace, K. Cho and T. Sheffler, "COSMOS: A Compiled Simulator for MOS Circuits," Proceedings of the 24th Design Automation Conference, 1987, pp. 9-16.

2. D. M. Lewis, " Hierarchical Compiled Event-Driven Logic Simulation," Proceedings of ICCAD-89, pp.498-501.

3. Peter M. Maurer and Z. Wang, " Techniques for unit-delay compiled simulation", 27th Design Automation Conference, 1990, pp. 480-484.

4. Wang, L., N. Hoover, E. Porter and J. Zasio, "SSIM: A Software Levelized Compiled-Code Simulator," Proceedings of the 24th Design Automation Conference, 1984, pp. 473-478.

5. Hansen, C., "Hardware Logic Simulation by Compilation," Proceedings of the 25th Design Automation Conference, 1988, pp. 712-715.

6. Barzilai, Z., J. L. Carter, B. K. Rosen and J. D. Rutledge, "HSS -- A High Speed Simulator," IEEE Transactions on Computer-Aided Design, Vol. CAD-6, No. 4. July 1987, pp. 601-617.

7. Chiang, M., and R. Palkovic, "LCC Simulators Speed Development of Synchronous Hardware," Computer Design, Mar. 1, 1986, pp. 87-91.

8. Z. Wang and Peter M. Maurer, " LECSIM : A Levelized Event Driven Compiled Logic Simulator", 27th Design Automation Conference, 1990, pp. 491-496.

9. Stephen A. Szygenda, David M. Rouse and Edward W. Thomson, " A model and implementation of a universal time delay simulator for large digital nets", in Spring Joint Computer Conference, 1970, pp. 207-216.

10. Bell, J. R., "Threaded Code", Communications of the ACM, Vol. 16, No. 6, June, 1973, pp. 370-372.

11. P. Maurer, Z. Wang, C. Morency, A. Tokuta and N. Bhate, "The Florida Hardware Design Language," *Proceedings Southeastcon-90*, pp. 430-434.