Author’s Note: Although this paper was published many years ago in journal form (TCAD), I am reissuing this as a Baylor Computer Science Technical Report in the interest of keeping all simulation papers in the same place.

THE SHADOW ALGORITHM: A SCHEDULING TECHNIQUE FOR BOTH COMPILED AND INTERPRETED SIMULATION

Peter M. Maurer
Department of Computer Science
Baylor University
Waco, TX 76798

Abstract.

The shadow algorithm is an event-driven unit-delay simulation technique that has been designed to take advantage of the instruction caches present in many of the latest workstations. The algorithm is based on the threaded-code technique, but uses a dynamically created linked list of environments called shadows. Compiled shadow algorithm simulations run in about 1/5th the time required for a conventional interpreted event-driven simulation. The interpreted shadow algorithm runs in about 1/4th the time of a conventional interpretive simulation.

1. Introduction.

The shadow algorithm, which is based on the threaded code algorithm [1], has been designed to take advantage of the instruction caches that are present in many of the latest workstations. It is also important to note that the algorithm is designed to handle asynchronous cyclic circuits. Combinational and synchronous cyclic circuits are better handled by other techniques [2,3,4]. This paper is *not* intended to be a treatise on simulation performance with various types of caches. However, in many cases the performance improvements realized by compiled simulations are achieved through a process which is similar to loop unrolling. On a machine with a large instruction cache, there is a point at which unrolling loops ceases to be advantageous, and can actually become detrimental to the performance of the program. Unfortunately, many compiled simulations are large enough that the “loop unrolling” process has actually become detrimental, primarily because of the lack of locality of reference in the generated code. The related, and far more complex problem of locality of reference to data-structures, is not addressed here.

The underlying mechanism of the Shadow Algorithm is a dynamically created linked-list of sub-environments called Shadows These environments work in a fashion similar to the threaded code scheduling queues. Each sub-environment invokes a short code-segment to perform a portion of the simulation. Shadow-Algorithm simulations run in about 1/5th the time required for a our interpreted event-driven simulator. The shadow algorithm may also be run interpretively without generating a separate simulation program. The interpretive algorithm runs in about 1/4th the time of our interpretive simulator.

Two approaches have been taken in the development of compiled unit-delay simulators, which are called "oblivious" and "event-driven," respectively. In an oblivious simulator the number of gates simulated per vector is constant, even if all input vectors are identical. In event-driven simulation, the state of the simulation is used to eliminated unneeded computations. Although oblivious simulators usually simulate many more gates than event-driven simulators, much overhead is eliminated, so the individual simulations are much faster. This allows oblivious simulators to outperform event-driven simulators when the activity rate is above some threshold which depends on the simulator and the timing model (usually from 2-20%).

Activity rate is usually computed by counting the number of gates simulated, and dividing by the number of gates in the circuit. However, because in a unit-delay simulation a gate can be simulated many times during the simulation of a single vector (even if the circuit is acyclic), this method of computing activity rates can give values of more than 100%. Since the maximum value of the activity rate is not known, it is difficult or impossible to compare the relative performance of oblivious and event-driven simulation simply by using the number of gates simulated. We have developed an alternative approach to obtaining the activity rate which corrects this problem. We compute the potential number of simulations that could occur during the simulation of a vector. This can be done by several different methods, the simplest of which is to modify the comparison between the old and new values of a gate-output, so that new events and gate simulations are scheduled regardless of the outcome of the comparison. We use the total number of potential simulations, rather than the gate count as the denominator when computing the activity rate. For cyclic circuits, our method of computing activity rates must be modified slightly. First, we run a conventional unit-delay simulation, and record the maximum number of time units required to simulate any vector. When computing the potential number of simulations per vector, we assume that each vector will be simulated this number of time units. For circuits that exhibit oscillations for some input combinations, we use the maximum number of gates simulated until forcible termination. (The precise method of calculation is more complicated than is suggested here, but lack of space precludes a more accurate description.)

This method of computing activity rates gives the actual simulations that must be performed by an oblivious simulation. We can then use the activity rate as a guide to determine whether an oblivious simulation will outperform an event-driven simulation for a particular circuit and a particular set of inputs. Unfortunately, our results have shown that for asynchronous circuits the activity rate seldom exceeds 1-2%, well below the thresholds of most oblivious compiled simulators. As the simulation of a vector progresses, the number of potential simulations per time unit tends to grow larger, while in a well designed sequential circuit, one would expect the number of actual gate simulations per time unit to become smaller as time progresses. This accounts for the observed low activity rates.

Our analysis has been supported by experimental data. We created several oblivious unit-delay simulators for asynchronous cyclic circuits, and these simulators were seldom able to match the performance of our interpreted event-driven simulator, much less surpass it. Because of this, we elected to abandon the oblivious approach to simulating asynchronous circuits, and concentrate on developing an efficient method for performing event-driven simulation. One such approach is the Gateway algorithm, which is an adaptation of Lewis's threaded code techniques[1]. The gateway algorithm does a somewhat better job of predicting when duplicate gate simulations can occur, and thus is able to eliminate some of the code used to detect these simulations. Furthermore, since the gateway algorithm generates C code rather than machine code, it does not perform as well as Lewis’s technique. However, for the purposes of this discussion, the two techniques can be considered equivalent.

We experimented with many different methods, and as these experiments progressed, one important fact became evident. When all else is equal, those methods with good locality of reference significantly outperformed those with poor locality. Our experiments were run on a SUN4-IPC, which possesses a large instruction cache. In similar experiments on a SUN3 with no cache, locality of reference appeared to have little or no effect. Because the effect of locality of reference was significantly larger than we had anticipated, we began looking at algorithms that would enhance the locality, possibly at the expense of executing more instructions or slower instructions. The shadow algorithm is the result of this investigation. The shadow algorithm has good locality of reference, but uses somewhat slower instructions (or more instructions, depending on the architecture) than those used in the Gateway algorithm. The shadow algorithm tends to outperform the Gateway algorithm, particularly on large circuits.

One surprising development was that of the interpretive shadow algorithm. The amount of executable code generated by the shadow algorithm is so small that many of the most commonly generated routines can be incorporated into a small “simulation kernel.” By also including a few generic routines to handle the less common cases, it is possible to create an interpreted simulator whose performance rivals that of a compiled simulation. Unlike conventional interpreted simulators, however, the interpretive shadow algorithm has no centralized scheduler.

2. The Shadow Algorithm.

In our current implementation, each gate and each net in the circuit is represented by a data structure called a shadow, containing information unique to the gate or net, such as input count and fanout. Each shadow also contains the address of a processing routine for objects of that type. In the compiled version of the shadow algorithm, a separate routine is generated for each type of of object, whereas in the interpretive version, generic routines are used to handle less commonly occurring objects, such as gates with large numbers of inputs. (Nets can differ in their fanout count and number of driving gates. The differences between gate-types are obvious.)

The simulation of an input vector is represented as a linked list of shadows which is created dynamically as the simulation progresses. The method for constructing this linked list is similar to the threaded code techniques of Lewis [1], and the Gateway algorithms described in [5]. To be specific, the list of shadows is managed as a linked-list queue. For efficiency the head of the queue is kept in a register, which also acts as a secondary environment pointer. This register is used in much the same way as local variable access through the system stack pointer.

Simulation begins with testing primary inputs for changes, and then proceeds in a two-phase manner alternating between net processing and gate simulation. Net processing determines if any changes have taken place in a net, while gate simulation adds new net structures to the processing queue after each simulation. Net trailer and gate trailer routines are used to switch from one phase to the other and perform bookkeeping operations. Each gate simulation routine or net processor removes the head element from the queue, loads the next element into the shadow pointer register, and branches to the processing routine for the new shadow.

3. The Interpretive Algorithm.

Perhaps the most striking feature of the shadow algorithm is small amount of generated code. Furthermore, the content of the gate-simulation routines and the net-handling routines does not depend on the structure of the circuit. This implies that the gate simulation routines could be pre-compiled and loaded from a library, rather than being generated and compiled. Since the number of different routines, even in the worst case, is quite small, this suggests the possibility of using a single standard simulation routine for all circuits, and generating only the data structures. This procedure permits the shadow algorithm to be used interpretively, eliminating the need for compilation. The standard simulation routine has the drawback that not all of the simulation routines will be used during the simulation of a particular circuit, which implies that unexecuted code could be loaded into the cache. This will affect the performance of the algorithm, but apart from this, one would expect the performance of the interpretive algorithm to be quite close to the performance of the compiled algorithm.

In our implementation of the interpretive shadow algorithm, we provided explicit routines for AND gates with from 2 to 10 inputs, and a generic AND routine for gates with more than 10 inputs. We provided similar routines for other gates such as NAND, OR, and NOR. Similarly, explicit net-handling routines were generated for nets with fanouts from 0 through 10, with a generic routine for larger fanouts. Other, reasonably obvious changes were made to convert the algorithm from a compiled algorithm to an interpreted algorithm.

4. Performance Results.

We performed several experiments to compare the shadow algorithm with a typical interpretive event-driven simulator. The experiments were based on the ISCAS85 combinational benchmarks, and the ISCAS89 sequential benchmarks. All experiments were run on a SUN-4/IPC with 12 megabytes of memory and a dedicated disk drive. This system was dedicated during the experiments, but could not be completely isolated. These benchmarks were run using 5000 randomly generated vectors. For the sequential benchmarks, two consecutive copies of each vector were generated, one with clock inactive, and the following with clock active, resulting in a total of 10000 vectors per circuit.

The results of the experiments are given in Figs. 1 and 2. All reported times are in CPU seconds of execution time. The times were obtained using the UNIX /bin/time command. To minimize errors in the /bin/time command, each experiment was run five times and the results were averaged. The results reported here do not include the time required to read and write vectors. It must be noted that these results depend on the quality of the implementations. We make no claim that any of our implementations are optimal, but we believe that all of *our* implementations are comparable in quality. In all sequential circuits, flip-flops are simulated directly rather than being expanded into combinational gates.

Sequential Circuits				
Ckt	Interp	Gateways	Comp-Shad	Interp-Shad

s27	2.0	0.4	0.4	0.6
s208	6.9	1.0	1.1	1.8
s298	10.2	1.5	1.8	3.3
s344	16.6	2.6	2.7	4.8
s349	16.3	2.5	2.8	4.7
s382	13.1	2.0	2.2	3.7
s386	15.4	2.1	2.3	4.0
s420	12.8	1.8	2.0	3.2
s444	14.7	2.0	2.4	3.4
s510	9.5	3.7	2.6	3.9
s526	15.3	3.7	2.4	3.6
s526n	15.2	3.6	2.4	3.6
s641	31.9	6.7	4.5	7.9
s713	34.4	8.4	5.3	7.8
s820	29.3	7.3	5.9	7.1
s832	29.6	7.1	5.8	7.2
s838	24.8	5.8	3.8	5.8
s953	36.8	7.0	5.5	6.5
s27	2.0	0.4	0.4	0.6
Perform. Impr.		5.0×	5.9×	3.8×

Fig. 1. Results for Certain ISCAS89 Benchmarks.

Fig. 1 presents a comparison between the two versions of the shadow algorithm, the gateway algorithm, and conventional interpreted simulation. The effect of locality in the shadow algorithm is apparent when one compares the individual results for the gateway algorithm with those for the shadow algorithm. When the circuits are small, the gateway algorithm outperforms the shadow algorithm. Both algorithms simulate precisely the same set of gates, but the shadow algorithm uses slower addressing modes than the gateway algorithm, (or more instructions, depending on the architecture). As circuits become larger, the locality of the shadow algorithm allows it to outperform the gateway algorithm, even though slower instructions are used. The results presented in Fig. 2 also support these observations. The Gateway algorithm is faster for the smallest two circuits, but slower for all others.

Combinational Circuits				
Ckt	Interp.	Gtway	C-Shad	I-Shad
c432	23.4	3.6	3.9	6.7
c499	26.1	4.2	4.5	6.5
c880	46.3	14.0	8.3	9.4
c1355	93.8	30.9	18.0	20.3
c1908	172.9	61.1	32.9	42.3
c2670	192.1	81.1	43.8	55.2
c3540	277.1	112.7	63.9	70.4
c5315	519.1	228.3	126.9	147.8
c6288	5108.6	2602.5	1245.6	1268.4
c7552	795.1	372.7	201.1	236.6
Perform.Impr.		2.9×	4.7×	3.9×

Fig. 2. Results for the ISCAS85 Benchmarks.

5. Conclusion.

The shadow algorithm has proven to be an effective method for accelerating unit-delay simulation of asynchronous circuits on computers with large instruction caches. Although the vectors used for this study provide a relatively high activity rate (at least for the combinational circuits), the simulations are "event driven" so the run time is proportional to the activity of the circuit. This implies that the shadow algorithm will outperform conventional interpreted event driven simulation, regardless of the activity rate. Note that all circuits in the ISCAS89 benchmark set were simulated as if they were asynchronous circuits.

Because of its success with unit-delay simulation, we are currently attempting to extend the scope of the shadow algorithm to other timing models, in particular, the multi-delay and zero-delay models [6]. It is clear that these two models pose problems that do not exist in the unit-delay model. In the multi-delay model, a timing-wheel or other sorting mechanism must be used to delay modification of the net-variables. A similar mechanism must be used in the zero delay model to delay the evaluation of a gate until all predecessor gates have been evaluated.

In any case, the existing version of the shadow algorithm can be used to speed up unit-delay simulation by a significant amount. Since the shadow algorithm does not impose any restrictions that are not already present in many existing simulators, we believe that it could be adapted very quickly for use in commercial products.

REFERENCES

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

2. P. M. Maurer and Z. Wang, " Techniques for unit-delay compiled simulation", *Proceedings of the 27th Design Automation Conference*, 1990, pp. 480-484.

3 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.

4. Z. Barzilai, 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.

5. P. Maurer, "Gateways: A Technique for Adding Event-Driven Behavior To Compiled Unit-Delay Simulations," Submitted for Publication.

6. Y. Lee, P. Maurer, “Two New Techniques for Compiled Multi-Delay Simulation,” DAC-29, to Appear.