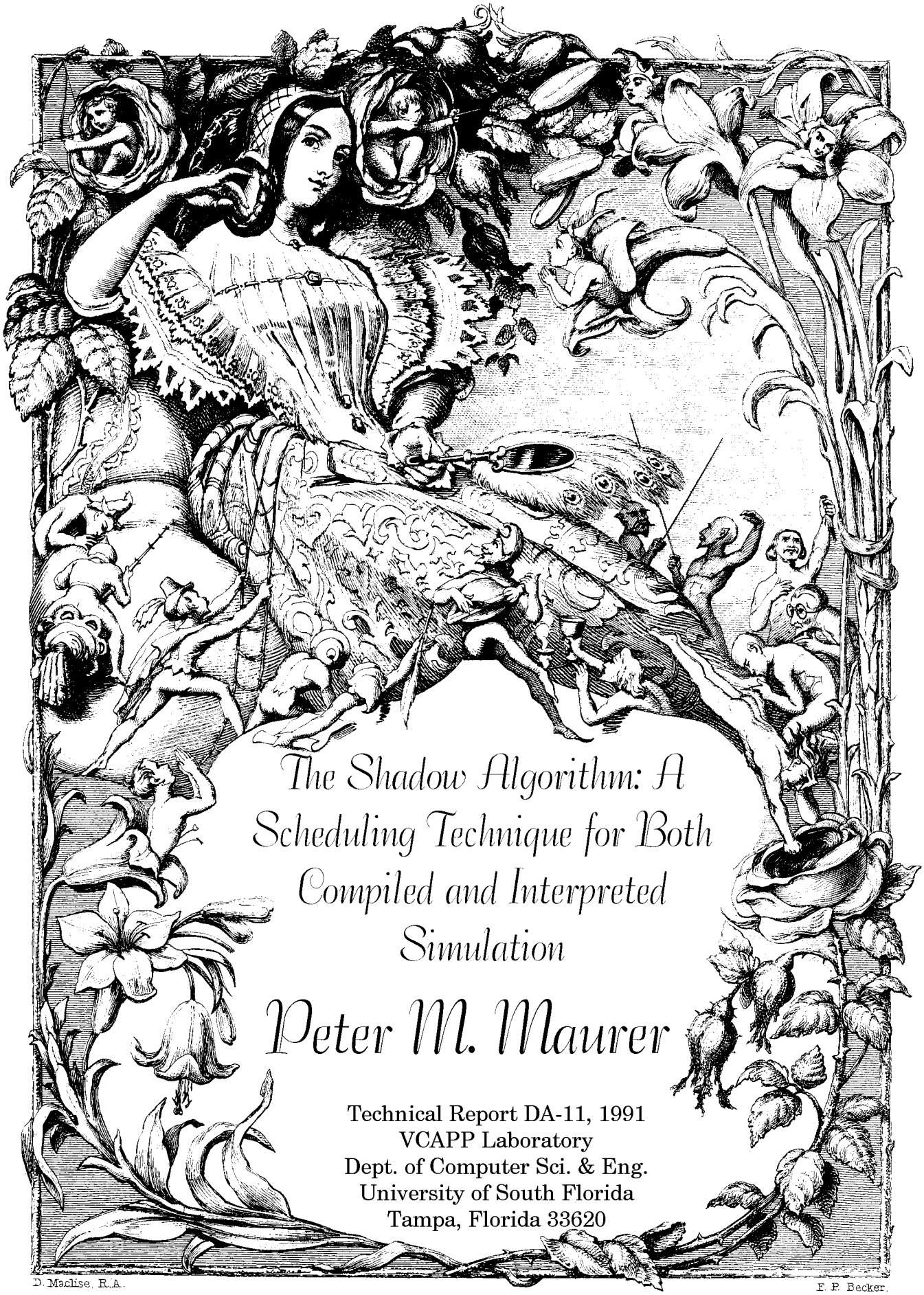

The Shadow Algorithm: A Scheduling Technique for Both Compiled and Interpreted Simulation

Peter M. Maurer

Technical Report DA-11, 1991
VCAPP Laboratory
Dept. of Computer Sci. & Eng.
University of South Florida
Tampa, Florida 33620

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

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

ABSTRACT

The shadow algorithm is designed to perform compiled event-driven unit-delay simulations of asynchronous sequential circuits. It has been specifically designed to take advantage of the instruction caches that are present in many of the latest workstations. The underlying mechanism is a dynamically created linked-list of environments called shadows. Each 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 conventional interpreted event-driven simulation. The shadow algorithm may also be run interpretively without generating a separate simulation model. This version of the algorithm runs in about 1/4th the time of a conventional interpretive simulation.

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

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

1. Introduction.

Over the past several years, several new techniques for compiled simulation of VLSI circuits have been developed [1-10]. The well known technique of levelized compiled code (LCC) [11] simulation has received much attention from both researchers and developers [10]. Although LCC simulation is based on the zero-delay timing model, it generally provides sufficient accuracy for debugging high-level logic models of a synchronous design. Compiled simulators have also been developed for unit-delay logic simulation [2,3], and for switch-level simulation [1].

In the development of compiled unit-delay simulators, two approaches have been taken which are called "oblivious" and "event-driven," respectively. An oblivious simulator takes no notice of the internal state of the circuit during the simulation of a vector. 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. (This definition includes some simulators that, technically, are not driven by events [4,12].)

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 (usually from 2-20%, depending on the timing model and the algorithm). On the other hand, the execution time of an oblivious simulator is not proportional to the activity rate, so in cases of very low activity, event-driven simulators tend to outperform oblivious simulators.

In most cases, the activity rate of a circuit is 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%. Because of this, we have developed an alternative approach to obtaining the activity rate. 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

---

* Manuscript Received _____. This work was supported in part by the National Science Foundation under grant number MIP-906444 and the USF Center for Microelectronics Research (CMR).
The author is with the Department of Computer Science and Engineering, University of South Florida, Tampa, FL 33620

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.

We use this method of computing activity rates because this number of potential simulations is the number of *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 cyclic circuits the activity rate seldom exceeds 1-2%, even when the input activity (i.e. the average number of changes between two input vectors), is very high. This activity rate is well below the thresholds of most oblivious compiled simulators. The reason the activity rate is so low is that as the simulation of a vector progresses, the number of potential simulations per time unit tends to grow larger. In most of our test circuits, this number converged to a number that was only slightly smaller than the number of gates in the circuit. However, 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 analysis has been supported by our experimental data. Our oblivious 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. This lead to the gateway algorithm [13], which uses retargetable "goto" instructions to switch portions of code into and out of the instruction stream. Each segment of code terminates with a retargetable branch instruction, the addresses of which are modified during the course of the simulation. Effectively, this creates a linked list of code segments as pictured in Fig. 1.

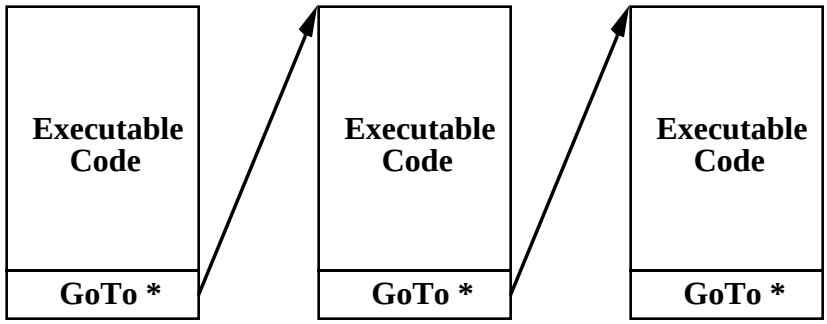

Fig. 1. The Gateway Algorithm Code Structure.

In our experiments with various different versions of the gateway algorithm, we noticed that when all else was equal, those versions 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. Because the effect of locality was significantly larger than we had anticipated, we began looking at algorithms that would enhance the locality of reference, 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 slower instructions than those used in the Gateway algorithm. Nevertheless, the shadow algorithm tends to outperform the Gateway algorithm, particularly on large circuits.

One surprising development was that of the interpretive shadow algorithm. It turns out that the amount of executable code generated by the shadow algorithm is extremely small. So much so that this code can be incorporated into a subroutine of the circuit compiler. Once this is done, the data-structures can be stored internally instead of being written to the

output, and the shadow algorithm can be performed by the circuit compiler. Effectively, this provides the capability of performing an interpreted simulation whose performance rivals that of a compiled simulation.

Section 2 of this paper describes the compiled shadow algorithm in detail, while Section 3 describes the interpretive shadow algorithm. Section 4 presents experimental results, while Section 5 draws conclusions.

2. The Shadow Algorithm.

To understand the concepts behind the shadow algorithm, it is necessary to review some concepts from programming language design. In a high-level language such as C or PASCAL, each activation of a procedure creates a new environment in which the activation runs. In particular, parameters and local variables are allocated space on the stack, which are accessed using offsets from the current stack address. Creation of a new environment for each procedure activation permits nested procedure calls, and in particular, recursive procedure calls to function correctly. For a recursive procedure, there may be several environments for the same procedure stored in the stack simultaneously. Furthermore by placing pointers to global variables in the environment, the same procedure can be made to act upon several different sets of variables.

Unfortunately, there are severe drawbacks to using a large number of procedures in compiled logic simulation. Procedure calls are relatively slow compared to the logic operations used for simulation. Procedure arguments (the source of the global variable pointers) must be pushed on the stack one at a time, the procedure call itself is normally somewhat slower than a simple jump instruction, and the old environment must be restored when the procedure terminates. The shadow algorithm addresses these problems by using preallocated environments called "shadows," and by accessing procedures through direct branches rather than procedure calls.

The code generated by the shadow algorithm consists of a small collection of routines which perform simulation and scheduling, and a large number of preallocated environments. Physically all of these procedures are sections of a larger simulation procedure. The shadows are actually small secondary environments which augment the environment of the procedure in which they are contained. To avoid large movements of data into the stack, a second address register is reserved for the pointer to the current shadow. At a minimum, each shadow contains the address of the procedure to be executed and the address of the next shadow to be activated. A shadow is activated by storing its address in the shadow pointer register and by branching to its procedure. This process mimics the restoration of environments that normally takes place when an interrupt handler terminates, but involves only the shadow pointer and the program counter. Fig. 2 illustrates the structure of a shadow and its associated procedure. Strictly speaking, the pointer to the next shadow is a local variable which is used by our scheduling algorithm. It is possible for other scheduling algorithms will omit this pointer. Furthermore, even with our scheduling algorithm, it is possible to drop out of shadow mode by performing a direct branch to code that does not make use of the shadow register, and re-enter shadow mode by loading the shadow pointer with an explicit address. At present, our algorithm does this only when initiating and terminating simulation.

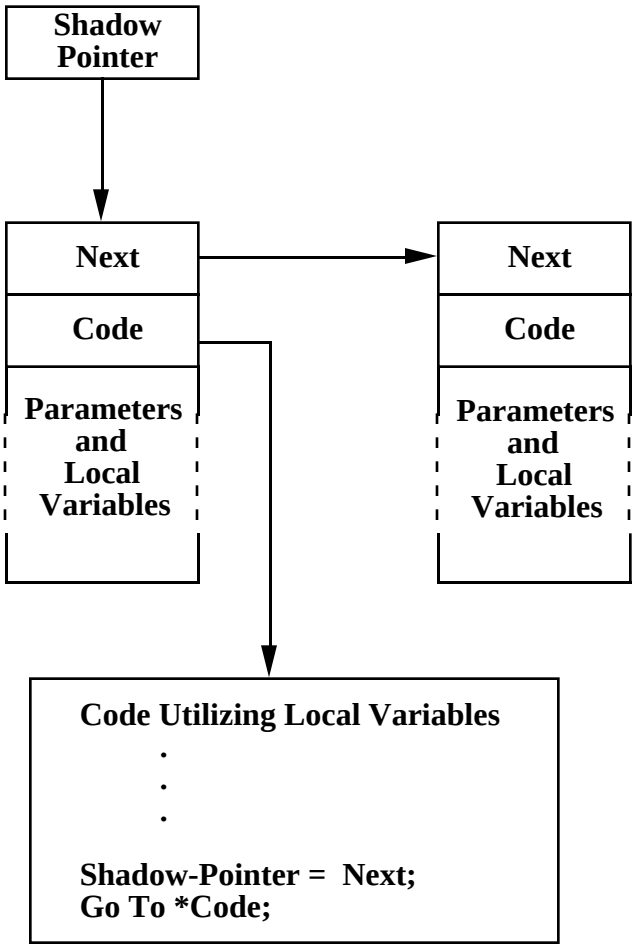

Fig. 2. The Structure of a Shadow.

In our current implementation, each gate and each net in the circuit is represented by a shadow. The value of each net is stored in a global variable. The shadow for a gate contains the address of the variables containing the values of the input nets, and the address of a temporary variable to hold the value of the output net. The gate shadow also contains the address of the shadow corresponding to the output net. (The current version of our simulator supports only single-output gates, but could be extended quite trivially to support gates with multiple outputs.) Shadows that correspond to nets contain the address of the variable containing the value of the net, the address of the temporary variable used by the simulation routines to store new net values, and pointers to the shadows of the gates in the fanout list of the net. (Again, our current simulator does not support wired connections, but could be trivially extended to do so.) There are three additional shadows called the net, gate, and simulation terminators respectively. Fig. 3 illustrates the shadows used for a three input gate and for a net with a fanout of three.

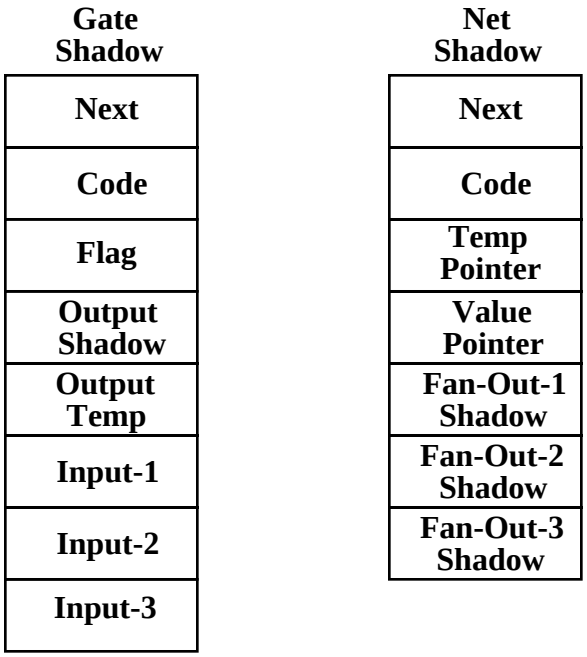

Fig. 3. The Gate and Net Shadows.

The code pointer of each gate shadow points to a routine that will simulate the correct type of gate. One routine is generated for each combination of gate type and input count found in the circuit, thus if a circuit consists entirely of 3 and 4 input NAND gates, two simulation routines will be generated, one for 3-input NANDS, and one for 4-input NANDS. Each gate simulation routine schedules the net processing routine for the output net of the gate.

Similarly, one net processing routine is generated for each different fan-out count detected in the circuit. The net processing routine will compare the new value of the net against its old value, and schedule additional gate simulations if a change is detected. Fig. 4 illustrates the gate-simulation and net-processing routines. In the code of Fig. 4, the symbol • precedes those variables that are contained in the shadow. These variables are accessed using offsets from the shadow pointer register.

```
AND3:
    *•Temp = *•Input_1 & *•Input_2 & *•Input_3 & 1;
    Flag = 0;
    Current->Next = •Output_Shadow;
    Current = •Output_Shadow;
    Shadow_Pointer := •Next;
    go to *•Code;

FAN_OUT1:
    if (*•Temp_Pointer ≠ *•Value_Pointer)
    {
        *•Value_Pointer = *•Temp_Pointer;
        if (•Fan_Out_1_Shadow->Flag = 0)
        {
            •Fan_Out_1_Shadow->Flag = 1;
            Current->Next = •Fan_Out_1_Shadow;
            Current = •Fan_Out_1_Shadow;
        }
    }
```

Fig. 4. Gate Simulation and Net Processing Routines.

The simulation of an input vector is represented as a linked list of shadows which is created dynamically as the simulation progresses. A register variable called "current" contains a pointer to the last shadow in the chain, and of course, the shadow pointer register points to the current element of the chain. The chain is initialized with two elements, the net trailer routine followed by the simulation trailer routine. The "current" pointer is initialized to point to the net trailer routine. The first step in simulation is to test all primary inputs for changes, and schedule any gates that use the changed nets as inputs. The "Flag" component of the gate shadow is used to avoid scheduling any gate twice. (Note that scheduling a gate twice will usually cause some gates to be erroneously dropped from the list of gates to be simulated.)

Once all primary inputs have been examined, the net trailer routine is entered via a direct branch, and the simulator enters shadow mode. The first action of the net trailer routine is to test whether any gate simulations are scheduled. If there are no gate simulations, the simulation terminates by activating the simulation trailer routine. Recall that all the routines referenced by shadows are actually segments of the simulation routine. The simulation termination routine simply executes a return, thereby terminating the simulation of the vector. If, on the other hand, one or more gate simulations have been scheduled, the net trailer then schedules the gate trailer routine. The net terminator finishes by activating the next shadow in the chain.

As gate simulations are performed, each gate simulation routine schedules its output net processor, and activates the next shadow in the chain. When the gate terminator is executed, it increments a counter and compares this counter against a user-specified limit to detect oscillations in the circuit. Next, it schedules the net termination routine and activates the next shadow. Note that there will always be net handling routines following the gate trailer. Each net handling routine will test the output of a gate for changes and will possibly schedule new gate simulations. After all net handling routines have been executed, the net termination routine is reactivated. The process continues until no more changes are detected or until the iteration limit is reached.

In our earlier work on the Gateway algorithm[13], we discovered several conditions under which it was possible to eliminate the setting and resetting of the gate flag. Some of these conditions are also applicable to the Shadow algorithm. First, if a gate has only one input, then the flag test is not needed, because the gate can never be scheduled more than

once. Second, if a gate has two inputs, one of which is a primary input, and the other of which is an internal node, the flag test is not required. (Recall that the primary input processing is performed separately from internal-node processing.) Unfortunately, use of these two conditions complicates the process of generating executable code. For example, when generating code for a two-input NAND, it may be necessary to generate two routines, one that resets the flag and one that does not. Generating the net-handling routines is even more complicated. Because each gate in the fanout list is handled by a separate segment of code, handling a fanout of n may now require as many as 2^n separate routines to be generated. Although this worst-case situation hardly ever occurs in practice, it is still necessary to generate additional executable code when using test-elimination. This tends to reduce the locality of reference in the generated code. We have implemented test elimination in the shadow algorithm, and it has resulted in some slight performance improvements, but not enough to justify the additional complexity in the compiler.

3. The Interpretive Algorithm.

Perhaps the most striking feature of the shadow algorithm is small amount of generated executable 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 for each circuit to be simulated. 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 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. It was necessary to add an "input_count" field to the gate shadow for the generic routines. Similarly, explicit net-handling routines were generated for fanouts from 0 through 10, and a generic routine was added for fanouts greater than 10. A fanout-count was added to the net-shadow for use by the generic routine.

It was also necessary to modify the vector-input routine. In the compiled algorithm, the primary input tests are performed explicitly as part of the simulation routine. Since this cannot be done in a standard simulation routine, it was necessary to create shadows for the primary inputs, and perform the primary input tests using the normal net-handling routines. The vector-input routine adds the shadows for the primary inputs to the chain of shadows to be processed. The simulation routine begins by activating the first shadow in the chain. In all other respects, the interpretive algorithm and the compiled algorithm are identical. (Test elimination is infeasible in the interpretive 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 [14] combinational benchmarks, and the ISCAS89[15] 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. Two different versions of the sequential benchmarks were run, one that expanded the D-flip-flops using the model pictured in Fig. 5 (which was taken from [15]), and one that simulated the flip-flops directly. As recommended in [15], each of the D-flip-flops was supplied with a clock input which is also a new primary input of the circuit. The same clock is used for all flip-flops. The standard interpreted simulator simulated all flip-flops directly.

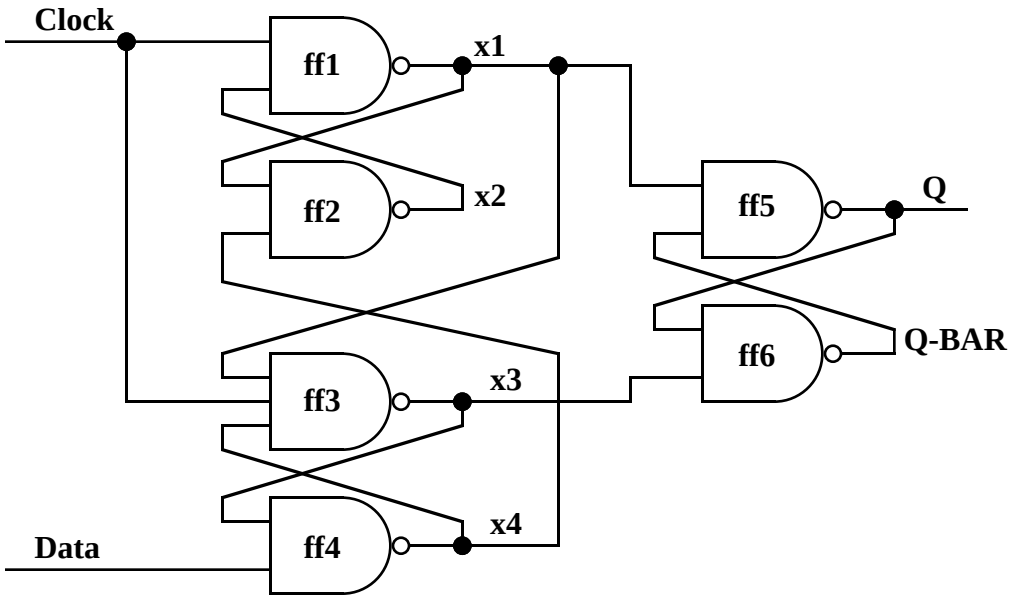

Fig. 5. The expansion model for D flip-flops.

The results of the experiments are given in Figs. 6, 7, 8, and 9. 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. These results do not include the time required to read and write vectors. This was accomplished by running three versions of each simulation (each five times), the first of which was an ordinary simulation. The second simulation was identical to the first, but with all calls to output routines suppressed. The third simulation was identical to the second, but with all calls to the simulation routines suppressed, but with the reading of vectors proceeding 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. The interpreted simulations were run on our own simulator which was developed for the purpose of performing these and other comparisons. 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 they are comparable in quality.

Sequential Circuits							
		Expanded Flip-Flops			Unexpanded Flip-Flops		
		Gateways	Shadows		Gateways	Shadows	
Ckt	Interp.	Tst-Elim	Unoptim	Tst-Elim	Tst-Elim	Unoptim	Tst-Elim
s27	2.0	0.8	1.0	1.0	0.4	0.5	0.4
s208	6.9	2.0	2.2	2.2	1.0	1.3	1.1
s298	10.2	3.4	4.0	3.9	1.5	2.0	1.8
s344	16.6	6.3	5.7	5.3	2.6	3.1	2.7
s349	16.3	6.2	5.8	5.3	2.5	3.1	2.8
s382	13.1	8.6	5.5	5.2	2.0	2.5	2.2
s386	15.4	2.9	3.4	3.2	2.1	2.5	2.3
s420	12.8	5.0	4.2	4.0	1.8	2.2	2.0
s444	14.7	8.4	5.7	5.3	2.0	2.6	2.4
s510	9.5	4.9	4.1	4.0	3.7	2.8	2.6
s526	15.3	9.0	5.7	5.3	3.7	2.7	2.4
s526n	15.2	9.0	5.6	5.4	3.6	2.7	2.4
s641	31.9	12.7	9.9	9.1	6.7	5.6	4.5
s713	34.4	13.4	10.8	10.0	8.4	6.1	5.3
s820	29.3	8.5	6.9	6.6	7.3	5.9	5.9
s832	29.6	8.8	6.1	6.6	7.1	5.1	5.8
s838	24.8	13.4	10.8	8.9	5.8	4.4	3.8
s953	36.8	15.8	12.1	11.5	7.0	6.2	5.5
Max Improv. (%)		81	79	79	86	84	85
Min Improv. (%)		34	50	50	61	71	72
Avg Improv. (%)		58	65	67	80	81	83

Fig. 6. Results for Certain ISCAS89 Benchmarks.

Fig. 6 presents a comparison between the two versions of the shadow algorithm, the gateway algorithm, and conventional interpreted simulation. As mentioned above, test-elimination provides little benefit over the unoptimized shadow algorithm. The effect of locality in the shadow algorithm is apparent when one compares the individual results for the gateway algorithm with those for the unoptimized shadow algorithm. When the circuits are small, the gateway algorithm outperforms the shadow algorithm. The effect is more pronounced for unexpanded flip-flops, primarily because less code is generated by the gateway algorithm for these circuits. Both algorithms simulate precisely the same set of gates, but the shadow algorithm uses slower addressing modes than the gateway algorithm. As circuits become larger, the locality of the shadow algorithm allows it to outperform the doorway algorithm, even though slower instructions are used. The results presented in Fig. 7 also support these observations. The Gateway algorithm is faster for the smallest two circuits, but slower for all others.

Combinational Circuits				
		Gtway	Shadows	
Ckt	Interp.	Tst-Elim	Unoptim	Tst-Elim
c432	23.4	3.6	4.4	3.9
c499	26.1	4.2	4.8	4.5
c880	46.3	14.0	9.2	8.3
c1355	93.8	30.9	20.2	18.0
c1908	172.9	61.1	36.6	32.9
c2670	192.1	81.1	47.6	43.8
c3540	277.1	112.7	68.1	63.9
c5315	519.1	228.3	134.0	126.9
c6288	5108.6	2602.5	1250.0	1245.6
c7552	795.1	372.7	211.4	201.1
Max Improv. (%)		85	82	83
Min Improv. (%)		49	73	75
Avg Improv. (%)		65	77	79

Fig. 7. Results for the ISCAS85 Benchmarks.

Figs. 8 and 9 contain the results for the interpreted shadow algorithm. For comparison purposes, the results for the standard interpreted algorithm and the gateway algorithm are repeated here.

Sequential Circuits					
		Expanded Flip-Flops		Unexpanded Flip-Flops	
Ckt	Std. Interp.	Gateways Tst-Elim	Shadows Interpreted	Gateways Tst-Elim	Shadows Interpreted
s27	2.0	0.8	1.6	0.4	0.6
s208	6.9	2.0	3.4	1.0	1.8
s298	10.2	3.4	5.2	1.5	3.3
s344	16.6	6.3	6.8	2.6	4.8
s349	16.3	6.2	6.7	2.5	4.7
s382	13.1	8.6	6.3	2.0	3.7
s386	15.4	2.9	4.2	2.1	4.0
s420	12.8	5.0	5.7	1.8	3.2
s444	14.7	8.4	7.1	2.0	3.4
s510	9.5	4.9	5.3	3.7	3.9
s526	15.3	9.0	7.7	3.7	3.6
s526n	15.2	9.0	7.4	3.6	3.6
s641	31.9	12.7	10.8	6.7	7.9
s713	34.4	13.4	11.7	8.4	7.8
s820	29.3	8.5	8.4	7.3	7.1
s832	29.6	8.8	8.5	7.1	7.2
s838	24.8	13.4	10.5	5.8	5.8
s953	36.8	15.8	11.6	7.0	6.5
Max Improv. (%)		81	73	86	82
Min Improv. (%)		34	20	61	59
Avg Improv. (%)		58	56	80	74

Fig. 8. The Interpreted Shadow Algorithm (Sequential).

As Fig. 8 shows, the interpreted shadow algorithm is slower than both the Gateway algorithm and the compiled shadow algorithm. However, the interpreted shadow algorithm gives an average performance improvement of almost 75% (i.e. running in 1/4th the time) over conventional interpreted simulation. It is possible that rearranging the code so that the most frequently used routines are clustered together would improve this performance somewhat, but we have not attempted to do this. As with the compiled algorithm, the results for the combinational circuits pictured in Fig. 9 reinforce the results for the sequential circuits.

Combinational Circuits			
		Gateway	Shadows
Ckt	Interp.	Tst-Elim	Interpreted
c432	23.4	3.6	6.7
c499	26.1	4.2	6.5
c880	46.3	14.0	9.4
c1355	93.8	30.9	20.3
c1908	172.9	61.1	42.3
c2670	192.1	81.1	55.2
c3540	277.1	112.7	70.4
c5315	519.1	228.3	147.8
c6288	5108.6	2602.5	1268.4
c7552	795.1	372.7	236.6
Max Improv. (%)		85	80
Min Improv. (%)		49	70
Avg Improv. (%)		65	74

Fig. 9. The Interpreted Shadow Algorithm (Combinational).

5. Conclusion.

The shadow algorithm has proven to be an effective method for accelerating unit-delay simulation 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. It should also be emphasized that the shadow algorithm is not limited to synchronous circuits, as are some compiled simulation techniques [3]. In fact, 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. 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-like mechanism must be used to delay modification of the net-variables until the appropriate time. 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. R. E. Bryant, 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. P. M. Maurer and Z. Wang, " Techniques for unit-delay compiled simulation", *Proceedings of the 27th Design Automation Conference*, 1990, pp. 480-484.
4. 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.
5. 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.
6. L. Wang, 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.
7. C. Hansen, "Hardware Logic Simulation by Compilation," *Proceedings of the 25th Design Automation Conference*, 1988, pp. 712-715
8. A. W. Appel, "Simulating Digital Circuits with One Bit Per Wire," *IEEE Transactions on Computer Aided Design*, Vol. CAD-7, pp. 987-993, Sept., 1988.
9. W. Y. Au, D. Weise, S. Seligman, "Automatic Generation of Compiled Simulations through Program Specialization," *Proceedings of the 28th Design Automation Conference*, 1991, pp. 205-210.
10. M. Chiang, and R. Palkovic, "LCC Simulators Speed Development of Synchronous Hardware," *Computer Design*, Mar. 1, 1986, pp. 87-91.
11. M. Breuer and A. Friedman, *Diagnosis & Reliable Design of Digital Systems*, Woodland Hills, CA: Computer Science Press, 1976.
12. S. P. Smith, M. R. Mercer, B. Brock, "Demand Driven Simulation: BACKSIM", *Proceedings of the 24th Design Automation Conference*, 1987, pp.181-87.
13. P. Maurer, "Gateways: A Technique for Adding Event-Driven Behavior To Compiled Unit-Delay Simulations," Submitted for Publication.
14. F. Brglez, P. Pownall, R. Hum, "Accelerated ATPG and Fault Grading via Testability Analysis," *Proceedings of the International Conference on Circuits and Systems*, 1985, pp. 695-698.
15. F. Brglez, D. Bryan, K. Kozminski, "Combinational Profiles of Sequential Benchmark Circuits," *Proceedings of ISCAS-89*, pp. 1929-1934.