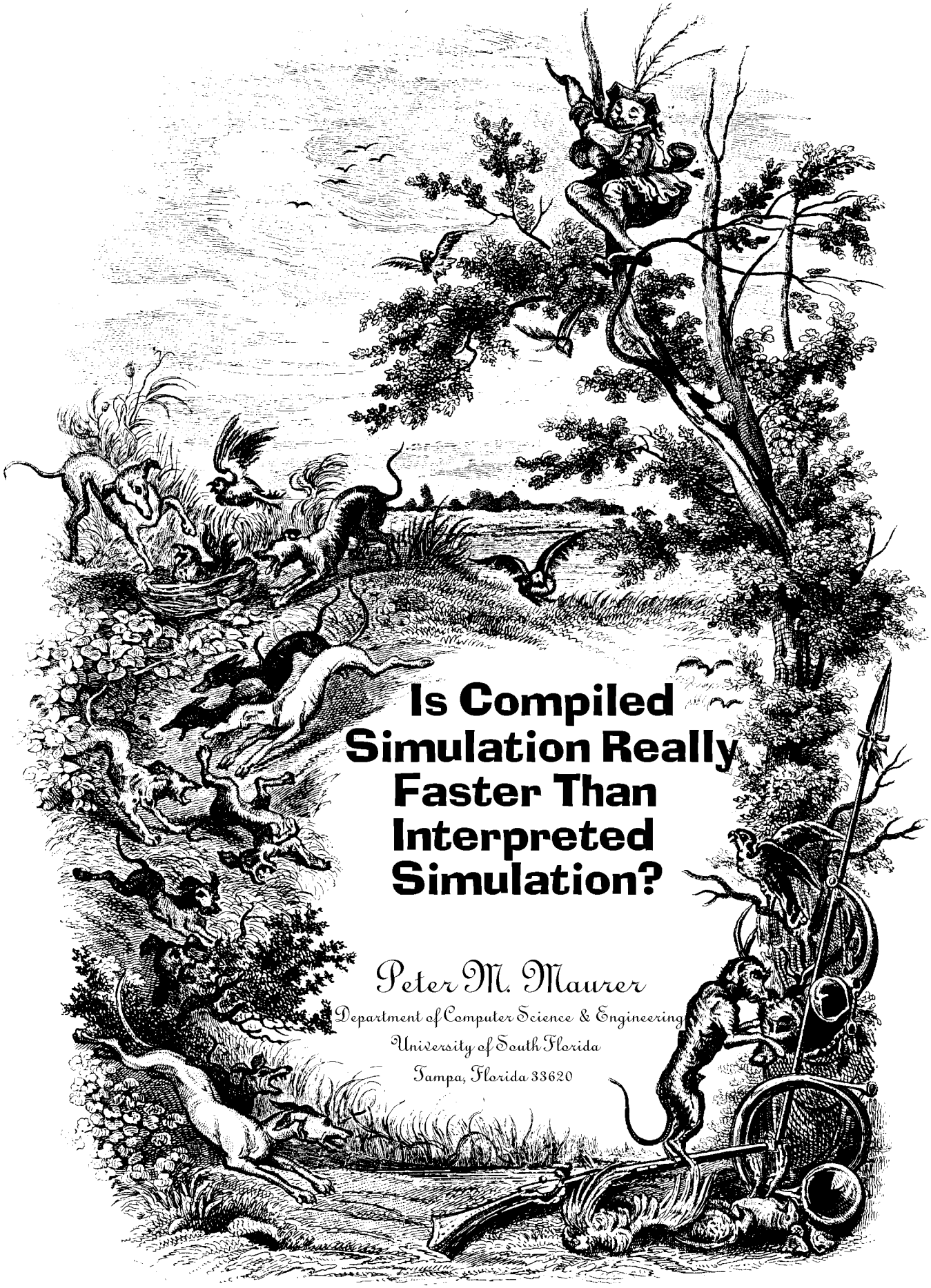

Is Compiled Simulation Really Faster Than Interpreted Simulation?

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

IS COMPILED SIMULATION REALLY FASTER THAN INTERPRETED SIMULATION?

Peter M. Maurer
ENG 118
Department of Computer Science & Engineering
University of South Florida
Tampa, FL 33620

ABSTRACT

It is commonly assumed that compiled simulation will provide significant performance improvements over interpreted event-driven simulation,. This paper demonstrates that for a new algorithm called the Inversion Algorithm, that this assumption is not true. The Inversion Algorithm can be run in either compiled or interpreted mode with only a slight difference in performance. Experimental data confirms this, and also demonstrates that first simulation results can be obtained much faster using the interpreted form of the algorithm.

IS COMPILED SIMULATION REALLY FASTER THAN INTERPRETED SIMULATION?

Peter M. Maurer
ENG 118
Department of Computer Science & Engineering
University of South Florida
Tampa, FL 33620

1. Introduction.

Simulation is an integral part of virtually every integrated circuit design methodology. Few of today's complex integrated circuits could have been designed without some form of simulation, and for many integrated circuits, simulation consumes more time than any other design activity. Because of this, there have been many recent improvements in the speed of simulation algorithms[1-15], and there have been several implementations of simulation algorithms in hardware[16-17]. Logic simulation is popular because it is reasonably efficient, and provides a reasonably accurate model of the circuit being designed. The components of the logic model can be compiled directly into silicon. For full custom designs, netlist comparison tools can be used to verify the layout against the logic model. Because simulation is a critical part of VLSI design, and because integrated circuits are growing ever larger and more complex, there has been a constant search for new and more efficient simulation algorithms.

Oddly enough, the search for more efficient algorithms has lead back to an algorithm that was widely used before the advent of event driven simulation, namely Levelized Compiled Code (LCC) simulation[4]. The LCC algorithm enjoys the reputation of being the fastest available method for performing logic simulation. This reputation is somewhat undeserved, but under certain conditions it is accurate. LCC simulation belongs to a class of simulation algorithms known as "Oblivious," because the number of gates simulated is not dependent on circuit activity. Such algorithms tend to minimize the amount of time required to simulate a single gate while maximizing the number of

gates simulated per input vector. As expected, oblivious algorithms out-perform event-driven techniques when circuit activity is high, but are less advantageous when circuit activity is low. LCC simulation is attractive because the break-even activity rate between LCC simulation and traditional event-driven simulation is quite low, typically from 1 to 3 percent. This low break-even point is crucial to the success of the LCC algorithm. If the break-even point were 40 or 50 percent, the LCC algorithm would be considerably less attractive.

One disadvantage of LCC simulation is the time required to build the simulator. Because code must be generated and compiled, a significant amount of time must elapse between making a change to a circuit and running the first simulation. This can slow the debugging process and increase development time. Nevertheless, many VLSI designers are willing to put up with long compile times to obtain increased simulation performance.

Recently, a new algorithm, the Inversion Algorithm, has challenged LCC simulation for its position as "the fastest known type of simulation." The Inversion Algorithm is event driven, and has been shown to be capable of exceeding the speed of LCC simulation in some cases, even when activity rates exceed 50%[1]. When activity rates decrease, the speed of the Inversion Algorithm increases, while the speed of LCC simulation remains constant. In addition to its speed, the Inversion Algorithm has a number of other intriguing advantages. It can be implemented either as an interpreted or as a compiled algorithm using virtually the same run-time code, and it can be implemented in less than ten pages of code. This suggests that an interpreted implementation of the Inversion Algorithm ought to run at about the same speed as a compiled version. The purpose of this paper is to describe the differences in the compiled and interpreted implementations of the Inversion Algorithm, and to present experimental data to compare the two approaches.

2. An Overview of the Inversion Algorithm.

Although the Inversion Algorithm is described in detail in reference [1], a short description is necessary here to make this paper self-contained. The Inversion Algorithm is an event-driven simulation algorithm which differs from other event-driven algorithms

in one important respect. In ordinary event-driven simulation, no gate is scheduled for simulation unless one of its *inputs* changes value. In the Inversion Algorithm no gate is scheduled for simulation unless its *output* is guaranteed to change value. When an event occurs on a gate input, the event processor performs tests to determine whether the event will propagate through the gate. If the gate simulation will not cause event propagation, no simulation is performed. These tests are relatively simple compared to the amount of processing required to simulate a gate. Since no gate is simulated unless its output changes value, a gate can be simulated by inverting its output value. The inversion can be done using a single unary operation.

For simplicity, the current implementation of the Inversion Algorithm is limited to eight gate types: AND, OR, NAND, NOR, XOR, XNOR, BUFFER, and NOT. (This *is not* a limitation of the algorithm, only of the current implementation.) Different event processors must be used for the inputs of different gates. Since any change in an input of an XOR, XNOR, BUFFER, or NOT gate implies a change in the output, no special tests are required for these gates. All events propagate unconditionally through the gate. Because an XOR or an XNOR gate has multiple inputs, it is possible to schedule one of these gates several times during the processing of a single input vector. However, successive changes in an output imply successive inversions of the output value. Since two successive inversions cancel one another, the inversion algorithm combines successive events and eliminates them.

Processing for AND, OR, NAND, and NOR gates is more complex, because not all events propagate through the gate. To handle gates of this type, the Inversion Algorithm maintains a count of dominant values for the gate. For OR gates, this is a count of the number of inputs with the value one, while for AND gates this is the count of the number of inputs with value zero. When the count changes from one to zero or from zero to one, a change is generated in the output. (As noted in references [20] and [21], a similar counting technique can be used in conventional event-driven simulation.)

Every change in an input causes the count of dominant values to be incremented or decremented. Two event handlers are used for each net, one that increments the count and one that decrements it. These routines are forced to alternate with one another in

strict fashion. As long as the identity of the next event processor is known, there is no need to maintain the value of the input net. Furthermore, since every change in the inputs of XOR, XNOR, BUFFER, and NOT gates generates a change in the output, it is not necessary to maintain values for the input nets of these gates. This implies that, except for primary inputs and monitored nets, no net values are ever required.

The key element which enables the Inversion Algorithm to be implemented in either compiled or interpreted form is the underlying implementation using the shadow algorithm[13]. The general structure used to represent an event is pictured in Figure 1.

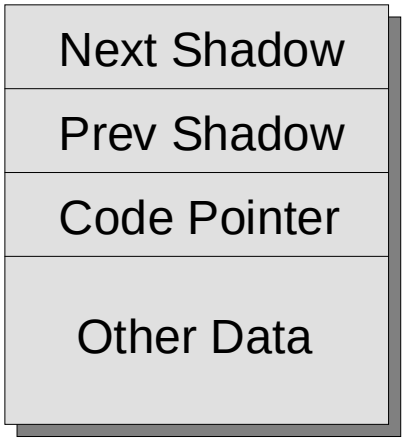

Figure 1. The Structure of a Shadow.

In Figure 1, the Previous and Next-Shadow pointers are used to queue and dequeue the event, while the Code Pointer contains a pointer to the event-processing routine. This structure differs from that of the "pure" Shadow Algorithm in that it contains both forward and backward pointers, and the value of the Code Pointer changes during simulation. In the Inversion Algorithm, one event structure is generated per fanout branch, and the number of different event-processing routines is fixed. In the "pure" Shadow Algorithm, one structure is generated per net and the number of different event and gate processing routines varies from circuit to circuit. The Inversion Algorithm has no gate processing routines.

3. Changing to an Interpreted Implementation.

The changes required to adapt the Inversion Algorithm from a compiled implementation to an interpreted implementation are surprisingly simple. To illustrate, the steps performed by the compiled implementation are listed in Figure 2.

1. **Create the global data structures and the queues.**
2. **Write a data-structure definition to the output file for each fanout branch in the circuit.**
3. **Write a copy of the event-handling routines to the output file.**
4. **Generate the input-vector processing routine, and write it to the output file.**
5. **Generate the output-vector print routine and write it to the output.**
6. **Generate the main simulation loop and write it to the output file.**
7. **Compile the output file.**
8. **Run the compiled program.**

Figure 2. Outline of the Compiled Implementation.

The primary change between the compiled and interpreted implementation is dynamically allocating data structures, and writing data to memory rather than to an output file. The circuit compiler must contain the event processors, and must be supplied with generic input and output routines as well as a generic main loop. The generic input and output routines are somewhat slower than the customized versions created for the compiled implementation, because in the customized routines all loops are completely unrolled. Otherwise the compiled and interpreted routines are identical. Because the Inversion Algorithm uses a fixed set of event handlers, no customization of the code is required in the compiled implementation, and no generalization of the code is required for the interpreted implementation. Figure 3 illustrates the steps performed by the interpreted algorithm.

1. **Allocate the global data structures and the queues.**
2. **Allocate space for the event-descriptors.**
3. **Write a data-structure definition into memory for each fanout branch in the circuit.**
4. **Enter the main simulation routine.**

Figure 3. The Interpreted Implementation.

The next section describes several performance comparisons that were made between the two implementations.

4. Experimental Results.

Because the interpreted and compiled simulations result in almost precisely the same sequence of instructions being executed during simulation, intuition would suggest that the simulation times for the two implementation techniques should be nearly identical. This intuition is very nearly correct, however the interpreted implementation runs in a larger partition, and places the event-descriptors in different portions of the virtual address space. As the results of Figure 4 show, these factors have some effect on the simulation time. The times reported here are in seconds of CPU execution time, and exclude both the time required to read and print vectors, and the time required to compile the circuit. The times were obtained using 5000 randomly generated test vectors. The tests were run on a SUN IPC with a dedicated disk drive and twelve megabytes of memory.

	LCC	Unoptimized			NOT Elimination		
Circuit	Sim.	Comp.	Interp.	%Incr.	Comp.	Interp.	%Incr.
c432	0.5	1.7	2.7	240.00	1.6	2.3	- 99.33
c499	0.6	2.0	2.6	233.33	1.9	2.5	- 99.19
c880	1.2	3.8	4.7	216.67	3.5	4.3	- 98.38
c1355	1.9	6.5	6.5	242.11	5.4	5.7	- 97.77
c1908	4.4	8.1	8.5	84.09	5.8	5.6	- 93.10
c2670	5.3	17.7	19.7	233.96	13.2	14.8	- 94.36
c3540	8.4	16.5	18.2	96.43	11.6	14.0	- 87.97
c5315	21.7	36.9	39.5	70.05	28.8	30.2	- 58.89
c6288	30.1	40.4	43.5	34.22	40.0	43.8	16.89
c7552	40.7	52.6	56.4	29.24	40.6	42.5	38.85

Figure 4. Interpreted Implementation Performance.

The results of Figure 4 show that the performance penalty for using interpreted simulation ranges from a high of about 60% for the smallest circuit, to a low of less than 5%. During a debugging session where relatively few test vectors are used this difference in speed would be barely noticeable. The results of Figure 4 show two different versions of the Inversion Algorithm, one which performs no optimizations, and one which eliminates NOT gates before performing simulation. (As explained in reference [1], it is possible to eliminate all NOT and BUFFER gates from the Inversion Algorithm

simulation without altering the results of the simulation.) The difference in compilation time between the two implementations is given in Figure 5. The circuits listed are the ISCAS-85 combinational benchmarks that have been used to evaluate the performance of many new simulation techniques, as well as algorithms for automatically generating test vectors[18]. The times reported here are the amount of real time that must elapse before the first vector can be simulated. Real times are reported here instead of CPU times, because this represents the amount of time that the user must wait before the first meaningful result can be obtained. These times were obtained using a dedicated system, and depending on other activity in the system, may vary by a few seconds from run to run. The system configuration for these tests was the same as that for the data reported in Figure 4.

Compile Time			
Circuit	**Gates**	**Compiled**	**Interpreted**
c432	160	12.1	0.6
c499	202	14.2	0.7
c880	383	21.9	1.3
c1355	546	30.8	2.1
c1908	880	37.0	2.6
c2670	1269	64.7	4.2
c3540	1669	73.5	5.7
c5315	2307	128.1	8.7
c6288	2416	132.9	9.9
c7552	3513	193.8	14.4

Figure 5. Compilation Time Differences.

As Figure 5 illustrates, the difference in compilation time for the two techniques is more than a factor of 10. However, for the circuits studied, the absolute amount of time required for the compiled implementation is small enough to be tolerable: slightly more than 3 minutes for the largest circuit. The real benefit of using the interpreted implementation comes when debugging larger circuits. To determine the benefit of the interpreted implementation for a large circuit, let us assume that the relationship between compile time and circuit size is linear, and that circuit c7552 is representative of this relationship. (As Figure 6 and Figure 7 show, these assumptions are not unreasonable.)

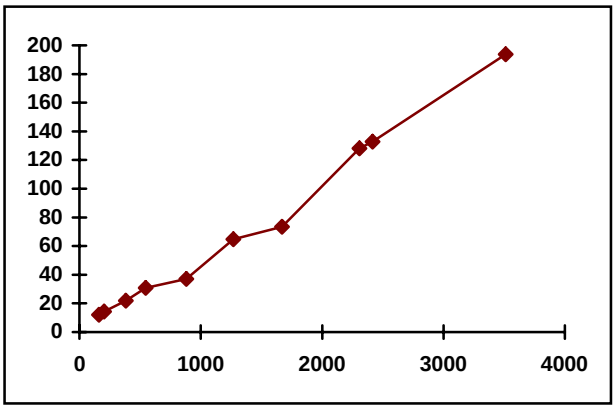

Figure 6. Compiled Compilation Time vs. Circuit Size.

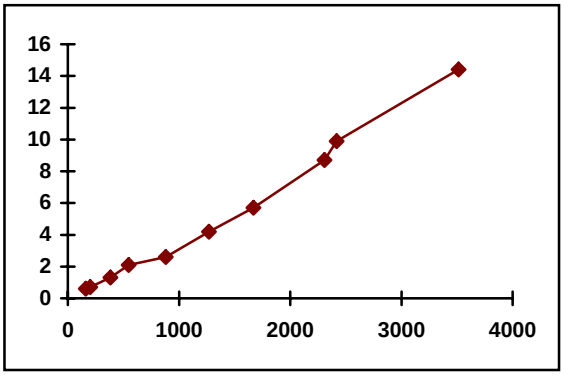

Figure 7. Interpreted Compilation Time vs. Circuit Size.

For circuit c7552, the compiled simulator compiles an average of 18.13 gates per second, while the interpreted simulator compiles an average of 243.96 gates per second. Extrapolating these numbers to a quarter-million gate circuit, the respective compilation times are 3 hours and 48 minutes for the compiled implementation and 17 minutes for the interpreted implementation.

5. Conclusion

This paper has discussed the differences between compiled and interpreted versions of the Inversion Algorithm. The Interpreted implementation has been shown to be almost as fast, in terms of simulation time, than the compiled technique, especially for larger

circuits. At the same time it has also been shown that the interpreted compilation time is significantly smaller than that of the compiled implementation.

The numbers reported here suggest directions for future research. Although the interpreted implementation provides for much faster compilation times, 17 minutes for a complete recompilation is nearly intolerable for intense debugging sessions. While it is true that a circuit of this size would normally be partitioned into smaller pieces for debugging, there might be advantages to doing at least a portion of the debugging on the entire circuit. For such a technique to be feasible for extremely large circuits, some form of incremental compilation will be necessary, regardless of what type of implementation is used. Because the executable portions of the interpreted Inversion Algorithm are fixed, it is an ideal vehicle for performing such compilations. Precompiled portions of the circuit will consist only of data structures which can be loaded or replaced as needed. The internal links used by these data structures, and the multi-level queue structure of the simulation will complicate matters, but these problems have well-known straightforward solutions.

Regardless of future research, the Inversion Algorithm has been shown to be effective technique for debugging medium to large circuits, and will undoubtedly become an indispensable tool in the logic designer's repertoire.

6. References

1. P. Maurer, "The Inversion Algorithm for Digital Simulation," Submitted for publication.

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

3. D. M. Lewis, "A Hierarchical Compiled Code Event-Driven Logic Simulator," *IEEE Transactions on Computer Aided Design*, Vol 10, No. 6, pp.726-737, June 1991.

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

5. 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.

6. 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.

7. C. Hansen, “Hardware Logic Simulation by Compilation,” *Proceedings of the 25th Design Automation Conference*, 1988, pp. 712-715

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

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

10. P. Maurer, “Two new techniques for unit-delay compiled simulation,” *IEEE Transactions on Computer-Aided Design,* Vol. 11, No. 9, pp. 1120-1130, Sept. 1992.

11. Y. Lee, P. Maurer, “Two New Techniques for Compiled Multi-Delay Simulation,” *Proceedings of Southeastcon 92*, Apr, 1992.

12. Y. Lee, P. Maurer, “Two New Techniques for Compiled Multi-Delay Logic Simulation,” *Proceedings of the 29th Design Automation Conference,* 1992, pp. 420-423.

13. P. Maurer, “The Shadow Algorithm: A Scheduling Technique for Both Compiled and Interpreted Simulation,” *IEEE Transactions on Computer Aided Design,* in press.

14. Z. Wang, P. Maurer, “LECSIM : A Levelized Event Driven Compiled Logic Simulator,” *Proceedings of the 27th Design Automation Conference*, 1990, pp. 491-496.

15. S. P. Smith, M. Mercer, B. Brock, “Demand Driven Simulation: BACKSIM,” *Proceedings of the 24th Design Automation Conference,* 1987, pp.181-87.

16. T. Blank, “A survey of hardware accelerators used in computer-aided design,” *IEEE Design and Test of Computers*, vol. 1, Aug. 1984, pp. 21-39.

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

18. 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.

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

20. M. Breuer, A. Friedman *Diagnosis and Reliable Design of Digital Systems*, Computer Science Press, Rockville, MD, 1976.

21. M Abramovici, M. Breuer, A. Friedman, *Digital Systems Testing, and Testable Design*, Computer Science Press, New York, 1990.