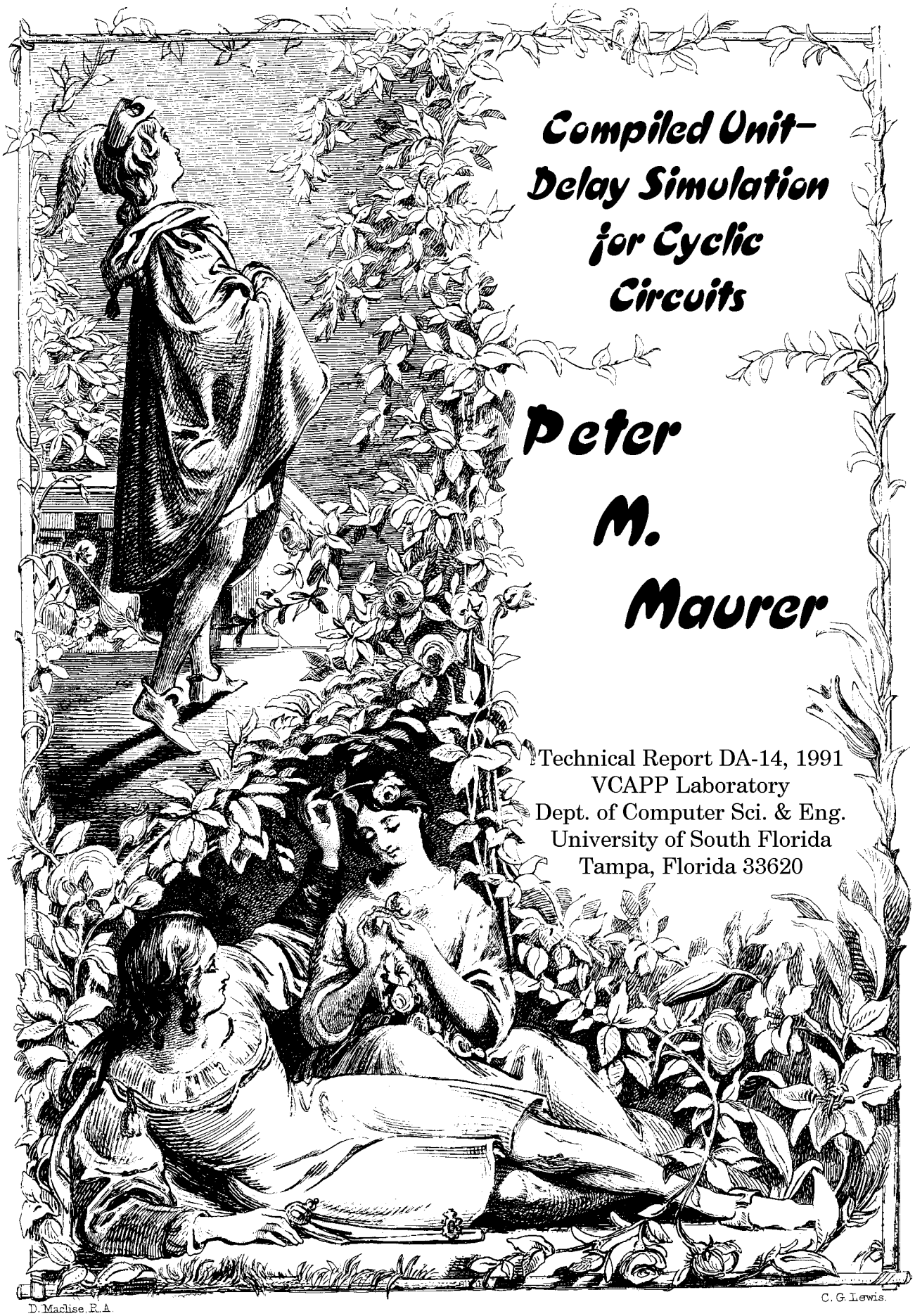


Compiled Unit-Delay Simulation for Cyclic Circuits

Peter M. Maurer

Technical Report DA-14, 1991
VCAPP Laboratory
Dept. of Computer Sci. & Eng.
University of South Florida
Tampa, Florida 33620

COMPILED UNIT-DELAY SIMULATION FOR CYCLIC CIRCUITS

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

COMPILED UNIT-DELAY SIMULATION FOR CYCLIC CIRCUITS

ABSTRACT

Three techniques are presented for handling cyclic circuits in a compiled unit-delay simulation. These techniques are based on the PC-set Method and the Parallel Technique of compiled unit-delay simulation. The first technique, called the Synchronous Parallel Technique, is applicable only to synchronous circuits, but provides significant performance improvements over interpreted unit-delay simulation. The second and third techniques, called the Convergence Algorithm and the Asynchronous Parallel Technique, are applicable to all circuits, both synchronous and asynchronous. The Convergence Algorithm, which is based on the PC-set Method, provides significant performance increases for some circuits, but performs poorly on others. The Asynchronous Parallel Technique performs rather poorly and is covered only briefly.

COMPILED UNIT-DELAY SIMULATION FOR CYCLIC CIRCUITS

1. Introduction.

Recent research in compiled simulation has provided a wide variety of techniques that can be used at different levels of the design hierarchy [1-9]. Traditionally, compiled simulation has focused on the zero-delay model [5-7], but there are also several techniques that are based on the unit-delay model [1-4]. Existing simulation techniques fall into two broad categories which can be termed "dynamic" and "oblivious." With dynamic techniques, the number of gates simulated varies from input vector to input vector based on both the content of the vectors and the values obtained from simulating various gates, while with oblivious techniques, the number of gates simulated per vector is constant. By far the most common dynamic technique is event-driven simulation.

Compiled unit-delay simulation has been implemented using both dynamic techniques [1,2] and oblivious techniques [3,4]. Two oblivious techniques are the PC-set method and the parallel technique [3]. Because both of these techniques are based on the well-known concept of levelization, [10] they are restricted to acyclic circuits. The purpose of this paper is to extend these techniques to cyclic circuits.

Cyclic circuits fall into two distinct categories, synchronous and asynchronous. Although synchronous circuits tend to be more important for modern-day VLSI design, the ability to handle asynchronous circuits is still important. In this paper we present three techniques, one of which is applicable only to synchronous circuits, and the other two of which are applicable to both synchronous and asynchronous circuits. The first technique extends the Parallel Technique to synchronous sequential circuits, while the second and third techniques extend the PC-set method and the Parallel Technique to asynchronous cyclic circuits. It should be noted that it is impossible to handle asynchronous circuits in a strictly oblivious manner, and that neither of the asynchronous techniques presented here are truly oblivious. As Section 3 shows, some sensitivity to changes in the network is necessary to obtain reasonable performance on asynchronous circuits.

2. The Synchronous Parallel Technique.

The parallel technique of compiled unit-delay simulation is fully described in [3], but for the sake of completeness, a short description of the technique is included here. The first step in the parallel technique is to levelize the circuit and determine its depth, which is the total number of levels in the circuit. A bit-field, whose width is equal to the depth of

the circuit, is allocated for every net in the circuit. The bit-fields are mapped into 32-bit words, and appropriate variables are generated, initialization code is generated for each net in the circuit, and finally code is generated for each gate in levelized order.

Each bit in a bit-field corresponds to one instant of simulated time. The low-order bit represents time zero, the time at which primary inputs are assumed to change. (Because of the unit-delay assumption, no other net is permitted to change at time zero.) Times increase sequentially as one moves to the left. Figure 1 illustrates a typical bit-field for a circuit of depth *d*.

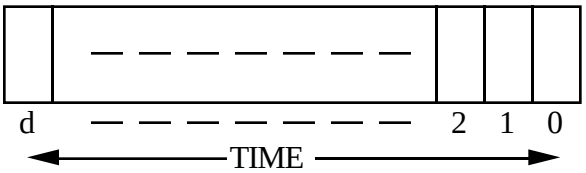


Figure 1. A Typical Bit-Field.

After a vector is simulated, the bit-field will contain a complete history of the value of a net during the simulation of the vector. For nets that are not primary inputs, the initialization code copies the final value of the net (which is contained in the high-order bit) into the low-order bit, which represents the initialization value of the net. For primary inputs, the net-value obtained from the input vector is replicated throughout every bit of the bit-field. Simulation code for a gate consists of bit-parallel logical operations which are replicated for each 32-bit word in the bit-field. Because of the delay of the gate, the result of these operations is a bit-field whose low-order bit corresponds to time one rather than time zero. To align the intermediate result with the bit-field of the output net, it is necessary to shift the intermediate result to the left one bit. The low-order word of the intermediate result must be ORed with the low-order word of the net to preserve the time-zero bit. It is possible to improve the speed of the parallel technique by using different alignments for different nets, thereby eliminating some of the shift operations For a more complete explanation of this technique, see [3] and [4].

2.1. The Changes in the Algorithm.

The first step in compiling a synchronous cyclic circuit is to convert it into an acyclic circuit so it can be levelized. Since the circuit is assumed to be synchronous, every cycle must contain at least one clocked flip-flop. Furthermore, these flip-flops must either be edge-triggered with respect to the clock, or the circuit must be designed in such a way that the data inputs of the flip-flop do not change while the clock is active. This allows the

circuit to be broken at the flip-flops, thus converting the circuit to an acyclic circuit that can be levelized.

Although the procedure for converting a cyclic synchronous circuit to an acyclic circuit is straightforward, it is not obvious exactly how this should be done. First, does one simulate the flip-flop before or after the rest of the circuit? As a practical matter, it is necessary to break the nets that are attached to a flip flop rather than the flip-flop itself. However, does one break the inputs of the flip-flop or the outputs? Once the circuit has been made acyclic, how does one handle the cyclic dependency that exists within the flip-flop itself? Finally, how does one deal with hazards in the clock or data inputs?

For clarity, we will describe our solution to these problems in terms of a clocked D flip-flop, but these comments apply to other types of flip-flops as well. Naturally, the choices we have made are based on the assumption that the underlying circuit is synchronous. This implies that if a D flip-flop is contained in a cycle, only the D input is actually part of the cycle, and that the clock input will not depend on the output of the flip-flop. Furthermore, most synchronous circuits are designed so that the D input is stable for some time before the clock becomes active, and remains stable while the clock is active. This is generally true even when the flip-flop is edge-triggered. In other words, the input of the flip-flop is generally designed to behave more or less like a primary input. On the other hand, the output of a flip-flop is normally expected to change one gate-delay after the clock becomes active, and this change may be visible at the primary outputs of the circuit. Therefore, we have elected to break cycles by breaking the D-input of each flip-flop. The clock input is not broken, so if there is clock-generation logic in the circuit, the flip-flop will be simulated after this logic, but before any gate that depends on the output of the flip-flop.

Note that breaking the D-input forces its value to remain stable for the purposes of simulation. The final value computed from the simulation of the previous input vector is used as the value of the D-input, and is propagated through all bits of the field that represents the input. Of course, simulation may show that the input does not, in fact, remain stable. If this is the case, and the flip-flop is not edge triggered, then the circuit is behaving in an asynchronous fashion, and an error message must be issued. A similar warning must be issued if the clock is not hazard-free.

The combination of a hazard-free clock and a stable D-input greatly simplify the generation of code for the flip-flop. There are three possibilities, each of which must be handled differently. The first possibility is that the clock may remain active throughout the simulation of the vector or may change from active to inactive. In both cases, the D-input is copied to the output in its entirety. The second possibility is that the clock remains inactive throughout the simulation, in which case no change is made in the output. The

third possibility is that the clock goes from inactive to active during the simulation. In this case, only those bit-positions of the D-input that correspond to active clock positions are copied to the output. The rest of the output bits retain their previous value.

2.2. Experimental Data.

Figure 2 shows experimental data for 18 of the ISCAS-89 sequential benchmarks [11]. The numbers in the "Interpreted" and "Compiled" columns are CPU-seconds of execution time. These benchmarks were run using 5000 randomly generated vectors. 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 CPU times in Figure 2 were obtained 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. 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. 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 either implementation is optimal, but we believe that they are comparable in quality.

	Gates Simulated	Interpreted CPU	Compiled CPU	Percent Improvement
s27	70,520	2.0	0.2	90.00
s208	232,227	6.9	1.4	79.71
s298	384,878	10.2	2.3	77.45
s344	632,928	16.6	2.4	85.54
s349	636,790	16.3	2.3	85.89
s382	475,540	13.1	3.1	76.34
s386	512,094	15.4	2.8	81.82
s420	409,042	12.8	2.8	78.13
s444	504,389	14.7	3.4	76.87
s510	234,745	9.5	3.2	66.32
s526	519,314	15.3	4.1	73.20
s526n	517,940	15.2	4.1	73.03
s641	1,109,530	31.9	13.8	56.74
s713	1,226,258	34.4	15.0	56.40
s820	972,013	29.3	6.2	78.84
s832	974,787	29.6	6.3	78.72
s838	750,483	24.8	13.7	44.76
s953	1,278,161	36.8	8.0	78.26

Figure 2. Synchronous-Parallel Experimental Data.

As Figure 2 shows, the percent improvement is highly variable for the circuits in question, ranging from 44% to 90%, with an average of 74%. These performance improvements are less than those observed for acyclic circuits, which averaged around 90%. This is due, in part, to the lower activity in the input vectors. Since every vector is repeated, the activity in the event-driven simulator for the second vector will normally be considerably less than for the first vector. Since the parallel technique is oblivious, it will simulate the same number of gates for each vector.

It should be noted, that random vectors are not always suitable input for sequential circuits, but due to the obscure documentation for the ISCAS-89 circuits, no attempt was made to insure that the randomly generated vectors were suitable for the circuits in question. This may have affected activity rate in the interpreted event-driven simulation.

3. The Convergence Method.

The Convergence Method of generating unit-delay compiled code is applicable both to asynchronous circuits and synchronous circuits, and is based on the PC-set method of compiled simulation [3]. The PC-set method first analyzes a circuit to determine every possible time at which a net may change value, and then generates a gate simulation for every such time. Although the PC-set method is applicable only to combinational circuits, the concept of generating a gate simulation at every possible simulation point can be extended to asynchronous cyclic circuits.

3.1 The Convergence Algorithm.

Conceptually, a unit-delay simulation can be viewed as a sequence of state transformations, where each state contains the value of all nets at a particular simulated time. The simulated times are consecutive integers, and the state transformations are logic equations that simulate the logical behavior of one or more gates. Figure 3 illustrates this concept.

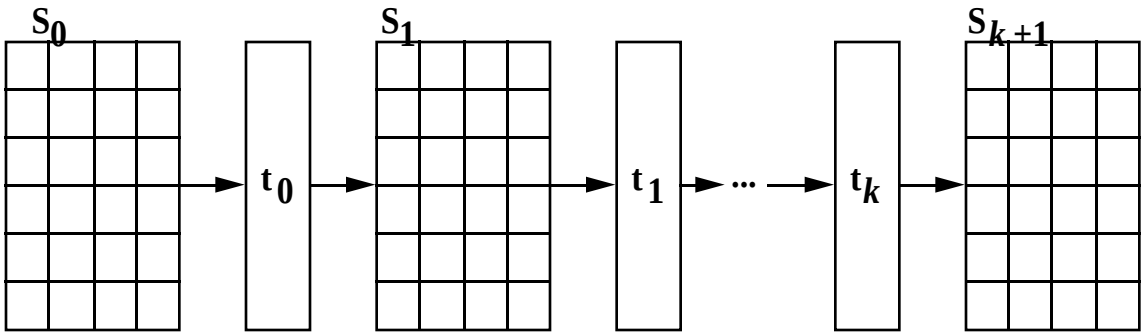


Figure 3. The State-Transition Model of Unit-Delay Simulation.

For each state transition, t_k illustrated in Figure 3, there is a minimal set of gates that must be simulated to correctly compute the new state S_{k+1}. Although there is a performance penalty for simulating too many gates in a transition, the correctness of the simulation will not be affected by doing so. In fact, it is possible to simulate *every* gate in the circuit for each transition, without affecting the correctness of the simulation. The event-driven method seeks to reduce the number of simulations to a minimum by simulating only those gates whose inputs have changed value since the last time the gate was simulated. However, because the amount of code required to detect changes in nets and schedule gate simulations is significant, it is sometimes more efficient to perform some extra simulations, if they can be done quickly.

The convergence algorithm is a method for identifying all gates that could contribute to a particular state transition t_k . This done by generating a sequence of subsets of gates, T_0, T_1, ...,T_k, ... Each subset T_k contains all gates whose inputs could potentially change value at time *k*. The following is a formal description of the convergence algorithm.

1. Place any gate that is directly connected to a primary input into T_0.
2. Set *n* equal to 1.
3. Place any gate which is a direct successor of a gate in T_{n-1} into T_n.
4. If T_n is empty or equal to T_k for any $k<n$, then stop,
   otherwise, increment *n* by 1 and return to step 3.

If the circuit in question is acyclic, then T_n will be empty for some $n>0$, otherwise the algorithm will terminate when $T_n=T_k$ for some $0\le k<n$. Termination is guaranteed for cyclic circuits, because the number of gates in the circuit is finite which implies that the number of different sets, T_k, must also be finite. Note that if $T_n=T_k$ for some $0\le k<n$, then $T_{n+i}=T_{k+i}$ for all $i>0$. More precisely, the sequence of sets from T_k through T_{n-1} will repeat indefinitely. The sequence of sets T_0 through T_{n-1} is called the *convergence sequence* of the circuit.

The code generated for the circuit will consist of a prefix portion followed by an iterative portion, as illustrated in Figure 4. The prefix portion will be generated from sets T_0 through T_{k-1}, while the iterative portion will be generated from sets T_k through T_{n-1}. At execution time, the iterative portion will be repeated until there is no change in the outputs of the gates contained in T_{n-1}, or until a predetermined limit is reached. When T_{n-1} contains a small number of gates, m, then the predetermined limit will be equal to 2^m+1 (assuming that each gate has only one output), otherwise an arbitrary value of 20 is used.

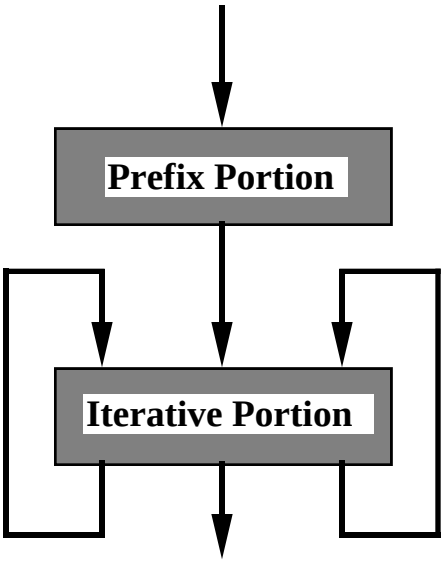
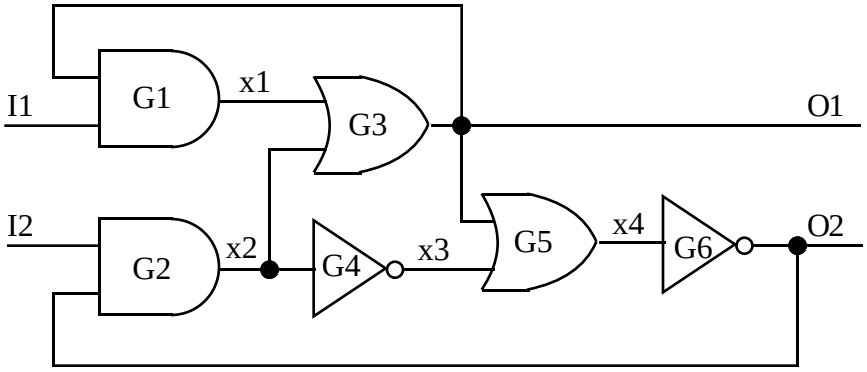


Figure 4. The Structure of the Convergence Algorithm's Generated Code.

Although it is possible to generate code in such a way as to maintain a complete history of each net for each input vector, the convergence algorithm allows a more space-efficient approach to be used. Since there is a clear division between gates simulated at different times, it is possible to print an output vector after each state transition is executed. If necessary, a history can be maintained for the purpose of hazard detection. When generating the code for a particular transition, it is necessary to avoid using net values that are computed within the transition. In other words, if T_j contains two gates G1 and G2, such that the signal x1 is an output of G1 and an input of G2, it is necessary to guarantee that the value computed for G1 is *not* used by G2 *in this transition*. This is necessary because the output of G1 must be delayed by one time unit and will not be available until the *next* transition. In some cases it is necessary to store the output of a gate in a temporary

variable until the transition has been completely computed. Figure 5 illustrates a cyclic circuit and its associated convergence sequence.



Convergence sequence:

T_0 = {G1,G2}
T_1 = {G3,G4}
T_2 = {G1,G5}
T_3 = {G3,G6}
T_4 = {G1,G5,G2}
T_5 = {G3,G6,G4}
(T_6 = T_4)

Figure 5. A Circuit and Its Convergence Sequence.

The convergence sequence illustrated in Figure 5 can be used to generate code for the circuit. The generated code for the circuit of Figure 5 is illustrated in Figure 6. The code of Figure 6 is written in the C language. For those not acquainted with this language, the operators &, | and ~ are used to perform bit-level AND, OR, and NOT operations, while the operators && and || are the AND and OR connectives for conditions. The operator != is read "not equal to."

```
I1 = vector input;
I2 = vector input;

/* time 0 */
x1 = I1 & O1;
x2 = I2 & O2;

/* time 1 */
O1 = x1 | x2;
x3 = ~ x2;

/* time 2 */
x1 = I1 & O1;
x4 = O1 | x3;

/* time 3 */
O1 = x1 | x2;
O2 = ~ x4;

t1 = O1; t2= O2 ; t3 = x3; i = 0;
while ((t1 != O1 || t2 != O2 || t3 != x3) && i<9)
{
    t1 = O1; t2= O2 ; t3 = x3; i = i+1;

    /* time 3+i */
    x1 = I1 & O1;
    x2 = I2 & O2;
    x4 = O1 | x3;

    /* time 3+i+1 */
    O1 = x1 | x2;
    x3 = ~ x2;
    O2 = ~ x4;
}
```

Figure 6. Code Generated by the Convergence Algorithm.

Although the individual gate simulations are very efficient, our preliminary experiments with the convergence algorithm showed that an enormous number of gates had to be simulated for each vector, as many as 72 times more than event-driven simulation in one case. For most of the circuits we tried the convergence algorithm performed around 30 times the number of simulations done by the interpretive event-driven simulator. In most cases this was enough to completely outweigh the benefit of eliminating the scheduling code of the interpretive simulator. In an effort to reduce the number of gate simulations, each state transition was broken into subsets of gates and tests were performed to determine if the simulation of the subset were necessary. If two gates share an input, the gates are placed in the same subset. Effectively, the subsets are the equivalence classes of the transitive closure of the following relation.

R={(G1,G2) | G1 shares an input with G2}

The inputs of a set are tested for changes to determine if the set needs to be simulated. If no simulations are done for a particular state transition, the simulation of the current input vector is terminated. The reason for partitioning the gates in this fashion is to avoid having to test any net more than once per state transition. It should be noted that this method of reducing gate simulations is less effective than event-driven simulation, since an output of a gate may be tested for changes even though the gate itself was never simulated. Although the addition of the tests improved the performance of the convergence algorithm for most circuits, the algorithm still does not exhibit performance improvements over interpreted simulation for all circuits, as the results in the following section show.

3.2. Experimental Results.

The experimental results for the convergence algorithm appear in Figure 7. These results were obtained in the manner described in Section 2.2 above. The careful reader will note that there are differences in the interpreted simulation times between Figures 7 and 2. These differences are due to inaccuracies in the UNIX /bin/time command. All experiments were run using the same input vectors, and all interpretive simulations were run using the same simulator.

	Interpreted	Compiled	%-Change
s27	1.9	0.4	78.95
s208	6.9	5.0	27.54
s298	10.2	4.0	60.78
s344	16.3	21.6	-32.52
s349	16.2	21.3	-31.48
s382	13.0	10.1	22.31
s386	14.9	13.8	7.38
s420	12.9	40.9	-217.05
s444	14.2	15.1	-6.34
s510	9.4	18.8	-100.00
s526	15.2	9.5	37.50
s526n	15.2	9.7	36.18

Figure 7. Convergence Algorithm Experimental Data.

While the results shown in Figure 7, do show significant performance increases for some circuits, they also show a significant performance degradation for others. This suggests that the convergence algorithm can be beneficial as long as one is selective about the circuits with which it is used.

4. The Asynchronous Parallel Technique.

We have also been able to extend the Parallel Technique to asynchronous cyclic circuits, but the performance of this algorithm in comparison with interpreted event-driven simulation is quite poor for most of the circuits we tested. Therefore we will provide only sketchy details of this algorithm. We believe there is room for improvement in this algorithm, but it is not yet clear whether these improvements would make the technique competitive with interpreted event driven simulation.

4.1. The algorithm.

The fundamental idea is to break the circuit into strongly connected components, and simulate the strongly connected components in levelized order. (A strongly connected component is a maximal set of gates such that the outputs of any two gates in the set depend on each other.) Within each strongly connected component, feedback arcs are identified and broken, and the gates in the component are levelized. Code is generated for each gate in the component in levelized order, as described in Section 2 above. This code is then executed iteratively to generate a predetermined number of output bits.

To understand why this approach works, suppose that one is simulating a simple cycle containing *n* gates, as illustrated in Figure 8. The values of all inputs to the cycle will have been completely computed before this component is simulated. Because of the standard initialization code, the correct value of the feedback arc, F1, is known at time zero. After simulating G1, the correct value of N1 will be known at time zero and time one. Similarly, after simulating gate G2, the value of N2 will be known at times zero, one, and two. After the component is completely simulated once, the value of the feedback arc, F1, will be known at times zero through *n*. After the first iteration, each subsequent iteration will produce *n* new correct bits for each net. The successive *n*-bit values produced on the feedback arc can be tested for changes to determine when to terminate the iteration, but the current simulator simply iterates until a fixed number of bits have been generated. This number, which is supplied manually to the compiler, is determined by running the interpretive simulator and measuring the maximum number of simulation cycles done for a single vector. We have experimented with a version of the compiler that generates dynamic tests instead of a fixed number of iterations, but the performance of this technique turned out to be worse than the results shown in the next section.

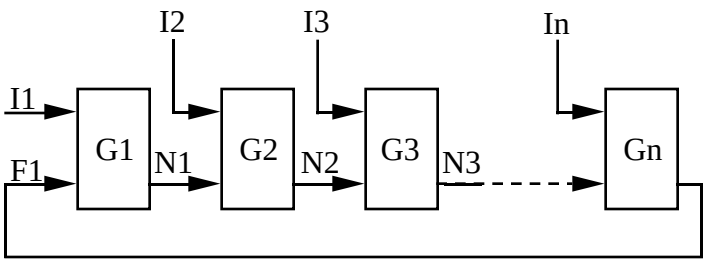


Figure 8. A Simple Loop.

There is some room for improvement in this algorithm. The number of bits generated per iteration depends on the length of the shortest loop in the strongly connected component. Obviously, the more bits per iteration, the better the algorithm will perform. The performance also depends on the number of feedback arcs detected in the strongly connected component. The fewer arcs that are detected, the better the performance will be. (The number of feedback arcs is *not* constant, and depends heavily on where the algorithm begins its search, and in what order gates are visited.) Although the relationship between the number of feedback arcs and the length of the shortest cycle is not known, minimizing the number of feedback arcs appears to produce longer cycles than choosing some non-optimal number. Unfortunately, the problem of minimizing feedbacks is known to be NP-Complete, which will make future improvements to this algorithm difficult.

4.2. Experimental Results.

Figure 9 contains experimental data for the asynchronous parallel technique. This data was obtained in the manner described in Section 2.2 above. As in Figures 2 and 7, the numbers represent CPU-seconds of execution times. The times for interpreted simulation may differ from those presented in Figures 2 and 7 due to inaccuracies in the UNIX /bin/time command, even though the same simulator and the same input vectors were used for all simulations.

	Interpreted	Compiled	%-Change
s27	1.9	3.1	-63.16
s208	7.0	12.9	-84.29
s298	10.3	21.2	-105.83
s344	16.5	41.3	-150.30
s349	16.6	40.2	-142.17
s382	13.0	30.1	-131.54
s386	15.0	13.9	7.33
s420	12.8	40.0	-212.50
s444	14.1	32.7	-131.91
s510	9.5	20.2	-112.63
s526	15.3	27.9	-82.35
s526n	15.2	28.4	-86.84

Figure 9. Asynchronous Parallel Technique Data.

5. Conclusion.

For the circuits tested, the synchronous parallel technique provides significant performance improvements over interpreted event-driven unit-delay simulation. Although the results reported here are not as good as those reported for purely combinational circuits, the performance improvements are around 75% on the average, which is significant. The advantage of compiled unit-delay simulation for asynchronous circuits is not as clear. Although the convergence algorithm provides significant performance improvements for several of the circuits tested, it also provides performance degradations for other circuits. This implies that some care must be used in selecting the circuits to which this algorithm is applied.

Although the Asynchronous Parallel Technique does not appear to be useful as it stands, it may be possible to make use of the algorithm in special-purpose simulation hardware. The primary problem with the Asynchronous Parallel Technique is that it is almost completely oblivious. Unfortunately, it is extremely difficult to add any dynamic features to the algorithm, because this requires testing segments of bit-fields for changes, which is quite time consuming on a general purpose computer. On the other hand, such a test could be implemented much more easily in special purpose hardware.

In certain environments, the poor performance of the Asynchronous Parallel Technique may not matter. If the simulator is used in an environment where there are many large combinational and synchronous sequential circuits to be simulated, and where the only asynchronous circuits are relatively small, it may be advantageous to have a single simulator that is capable of handling all circuits, even though its performance on the asynchronous circuits is not ideal.

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", *Proceedings of the 27th Design Automation Conference*, 1990, pp. 480-484.

4. P. Maurer, "Optimization of the Parallel Technique for Unit-Delay Compiled Simulation," *Proceedings of ICCAD-90*, pp. 70-73.

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

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. 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.

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

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

11. F. Brglez, D. Bryan, K. Kozminski, "Combinational Profiles of Sequential Benchmark Circuits," *Proceedings of ISCAS-89*, pp. 1929-1934.