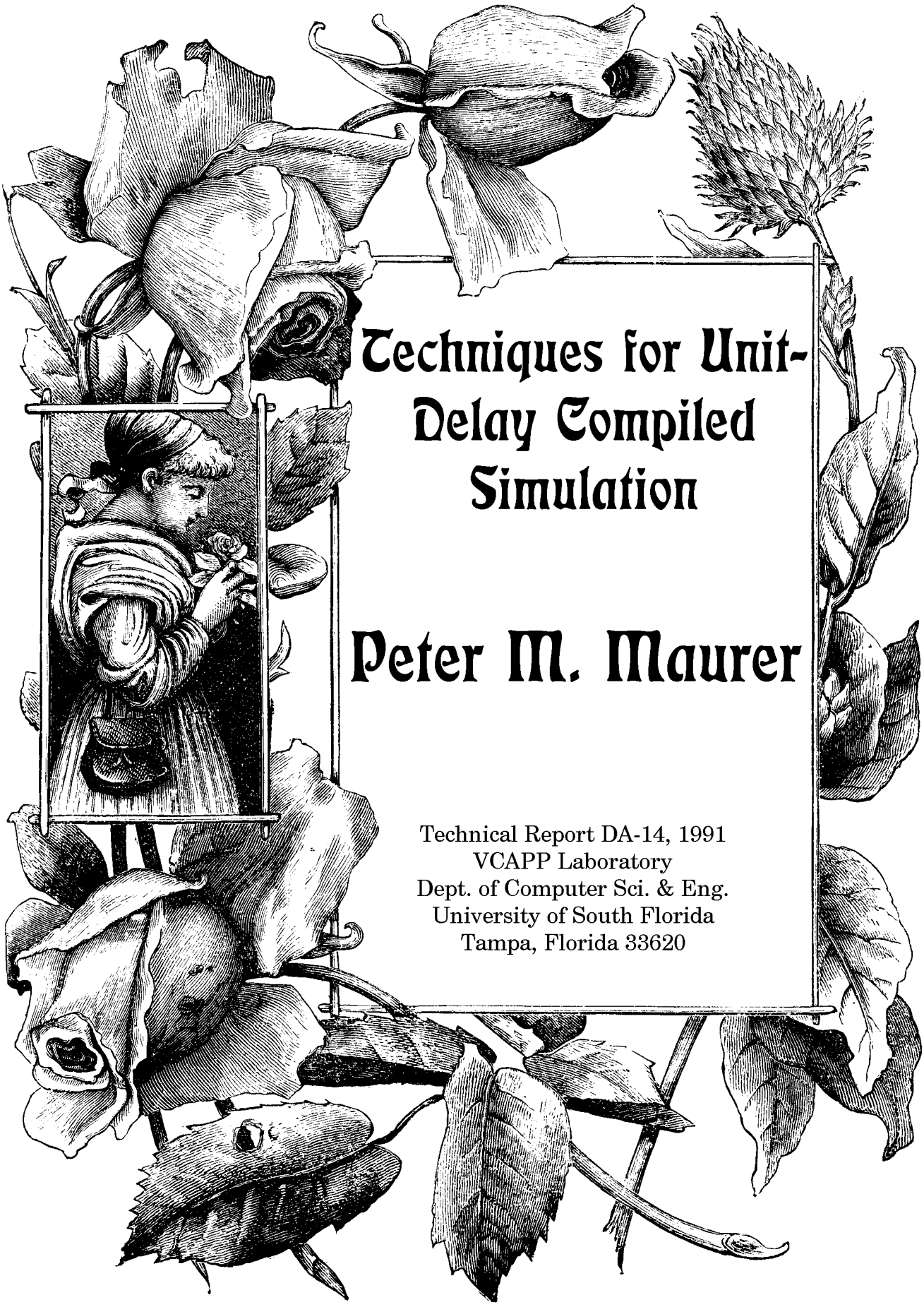


Techniques for Unit-Delay Compiled Simulation

Peter M. Maurer

Technical Report DA-14, 1991

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

TECHNIQUES FOR UNIT-DELAY COMPILED SIMULATION

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

TECHNIQUES FOR UNIT-DELAY COMPILED SIMULATION

ABSTRACT

The PC-set method and the parallel technique are two methods for performing compiled unit-delay simulation. The PC-set method analyzes the network, determines the set of potential change times for each net, and generates gate simulations for each potential change. The parallel technique, which is based on the concept of parallel fault simulation, is faster and generates less code than the PC-method, but is is less flexible. Benchmark comparisons with interpreted event-driven simulation show a factor of 4 improvement for the PC-set method and a factor of ten improvement for the parallel technique.

Category 1.2, Discrete Simulation.

TECHNIQUES FOR UNIT-DELAY COMPILED SIMULATION

1. Introduction

Recently there has been much renewed interest in compiled simulation, primarily because it provides significant increases in performance over interpreted methods [1-6]. As was pointed out in [5], some of the reasons that complied simulation has received less attention than interpreted simulation is that the traditional methods for producing Levelized Compiled Code (LCC) simulators are oriented toward zero-delay simulation and synchronous circuits, while interpreted event-driven simulation is oriented toward asynchronous circuits and more realistic timing models. While the natural bias of Levelized Compiled Code simulation is indeed toward zero-delay simulation, more sophisticated code generation algorithms can be used to generate code for other timing models. For example, the COSMOS compiled simulator [2] incorporates an event queue to perform unit-delay simulation. The techniques presented in this paper are also oriented towards unit-delay simulation, but forgo the use of an event queue in favor of a more extensive compile-time analysis of the network to be simulated.

The algorithms presented here are extensions of the LCC method, and give rise to straight-line code. Because of this, the generated code is extremely fast. On the average, our fastest technique runs in about 1/10 the time required for an interpreted event-driven simulation.

2. Fundamentals

The algorithms presented in this paper assume each gate has a delay of 1 time-unit. The length of these time units is not specified, and it is tacitly assumed that they are small enough that any number of such time units could elapse between two successive clock phases. All circuits are assumed to be synchronous and acyclic, however the algorithms can be applied to a wide variety of cyclic networks by requiring that any cycle in the network contain at least one flip-flop. The circuit can then be broken at the flip-flops by treating flip-flop inputs as primary outputs and their outputs as a primary inputs. We are currently working to extend these techniques to asynchronous circuits. Both of the techniques described in this paper are based on the well-known technique of Levelized-Compiled-Code (LCC) simulation. In LCC simulation every net and every gate in the circuit is assigned a level number, with primary inputs, constant signals and flip-flop outputs being assigned level zero. Once all inputs of a gate have been assigned levels, the maximum of these levels is found, incremented by one, and assigned to the gate and all of its outputs. If there are wired connections in the circuit, the net is assigned the maximum of the levels of its source gates. Code is generated for gates in ascending order by levels. Levelization requires a circuit to be acyclic.

The level of a net is the number of gates on the longest path between the primary inputs and the net. In unit-delay simulation, it also represents the *time* at which all changes in the primary inputs will have propagated to the net. It is also useful to know the time at which changes in the primary inputs first reach the net which we call the *minlevel* of the net. Minlevel can be calculated using an algorithm identical to the levelization algorithm, except the maximum operation is replaced by a minimum operation.

3. Calculation of the PC Set

Both level and minlevel represent potential times at which the value of a net may change. A net may have additional potential-change times between its minlevel and its level, but in general not all times between these extremes represent potential changes. The

set of potential change times, which we call the PC-set, can be calculated precisely as follows. Every primary input, constant signal, and flip-flop output is assigned the set {0}. Once all PC-sets have been calculated for the inputs of a gate, the union of these sets is formed and each element is incremented by one. This new PC-set is assigned to the gate and each of its outputs. For wired connections, the PC-set of the net is the union of the PC-sets of its source-gates.

The PC-set represents the lengths of all paths between a net and the primary inputs. For example, if net A has the PC-set {3,7,15} then there are paths of length 3, 7, and 15 between the net and the primary inputs. The size of the PC-set of any net is restricted to *level-minlevel*+1. Note that the PC-set contains both the level and the minlevel of a net.

4. The PC-Set Method of Simulation.

Because the PC-set of a gate determines when the gate is permitted to change its output, the PC-set can be used to generate Levelized-Compiled-Code for unit-delay simulation. However, it is first necessary to determine which nets must retain their values from the previous input vector. To see why this is necessary, consider the gate pictured in Figure 1.

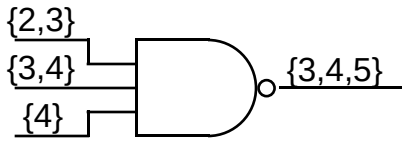
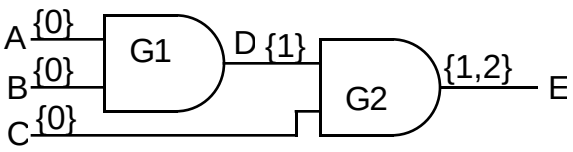


Figure 1. A Gate with PC-Sets.

Three output values must be calculated for the gate pictured in Figure 1, one at time 3, one at time 4 and one at time 5. When the time-3 value is calculated, the time-2 values of all input nets must be used. However, the two remaining nets cannot change value until times 3 and 4 respectively. This implies that the calculation of the time-3 output must use the values that have been retained from the previous input-vector. To determine which nets must retain their values from the previous iteration, the minlevels of the inputs of each gate are compared. For any gate whose inputs do not have identical minlevels, any net whose minlevel is larger than the minlevel of at least one other input must retain its value from the previous input vector. Every net that must retain its value from the previous vector has the element zero added to its PC-set.

The next step is to generate the variables that will be used to hold net values. For each net, one variable is generated for each element of the PC-set. Code is generated to read input vectors and to initialize those nets whose PC-sets contains zero and at least one other element. (The value that corresponds to the level of the net is copied into the variable that corresponds to the zero element.)

Finally gate simulation code is generated in levelized order. One gate simulation is generated for each element of the gate's PC-set. The operands of each gate simulation are obtained by searching the PC-sets of the input nets for the largest element that is strictly smaller than the PC-element for which code is being generated. The insertion of zero elements guarantees that such an element always exists. Figure 2 shows a network and part of the code that is generated from it.




```
char A_0,B_0,C_0,D_0,D_1,E_1,E_2;
…
D_0 = D_1;
D_1 = A_0 & B_0;
E_1 = D_0 & C_0;
E_2 = D_1 & C_0;
…
```

Figure 2. Code Generated by the PC-Set Method.

The code illustrated in Figure 2, is expressed in the C, the target language of our compiler. The "char" statement declares a collection of 8-bit binary variables, while the "&" operator is performs a bit-wise logical AND. (It is assumed that the low-order bit of each variable is used to hold net values.)

This code is executed once per input vector and creates a complete history for the vector. Assuming that output will consist of a trace of the primary outputs, the output code is treated as if it were a gate whose inputs correspond to the primary outputs of the circuit. Zero insertion is required, and the output of a single vector corresponds to the simulation of a single gate. A procedure similar to that for handling gate inputs is used to determine which values to place in each vector.

Because PC-sets represent potential changes rather than actual changes, the PC-set method causes some unnecessary computation to be done. Nevertheless, our studies on standard benchmark circuits show that the PC-set method is significantly faster than interpreted event-driven simulation. (See section 7.)

5. The Parallel Technique.

Although the PC-set method is fast, it tends to generate an enormous amount of code (over 100,000 lines for one of our benchmark circuits). We first began exploring the parallel technique as a means of reducing the size of the generated code. The parallel technique, which is based on the concept of parallel fault simulation[7], not only generates significantly less code than the PC-set method, it also runs much faster. (On the other hand, the PC-set method is amenable to bit-parallel simulation of multiple input vectors[8], while the parallel technique is not.)

The first step in the parallel technique is to allocate a bit-field for each net in the circuit and map the bit-fields into variables. If there are *n* levels in the circuit, an *n*-bit field is allocated for each net. Our current implementation maps the fields into 32-bit words with the low-order bit of the field aligned with the low-order bit of the first word. If the number of levels is not a multiple of 32, the high-order bits of the last word are unused. Words are never split between nets. Each bit in the field corresponds to one time unit. The low-order bit represents time zero, and contains either the initialization value of the net or the final value calculated from the last input vector. The values of all other bits are calculated by the simulation code.

The generated code first initializes all bit-fields by shifting the high-order bit into position zero, and clearing the remainder of the field to zeros. Next, a vector is read and the value of each primary input is propagated throughout every bit in the field.

Gate simulation code is generated in levelized order. If bit-fields are small enough to be contained in a single word, the gate-simulation code is essentially identical to that produced by a LCC simulator. However, if bit-fields must split into *k* words, then *k* simulations must be generated for each gate. The resulting bit field is shifted to the left one bit and ORed into the bit-fields representing the gate's outputs. The left-shift represents the time-delay of the gate, while the OR operation allows the low-order bit of the output-field to be preserved.

A trace of the primary outputs can be printed by using a sliding mask to determine the value of each primary output at each time unit. Hazard analysis can be done quickly by using a binary search technique and comparison fields of the form 0...01...1 and 1…10…0.

Figure 3 contains a portion of the code generated by the parallel technique for the network pictured in Figure 2.

```
char A,B,C,D,E;
…
D = (D>>2)&1;
E = (E>>2)&1;
D = D | ((A & B)<<1);
E = E | ((D & C)<<1);
…
```

Figure 3. Code Generated by the Parallel Technique.

The operator ">>" shifts its first operand right a number of bits equal to its second operand. The operator "<<" is similar and performs a left shift. The operator "|" performs a bit-wise logical OR. Figure 4 illustrates the effect of executing this code.

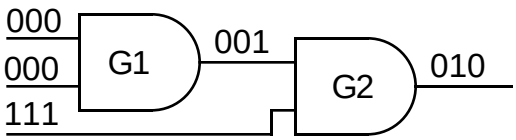


Figure 4. Bit-Fields Computed by the Parallel Technique.

In Figure 4, the nets of the diagram are labeled with the bit-fields computed by the code of Figure 3. The high-order bit of each field represents the final value of the net. The current input vector is 001, while the previous vector was 110. The low-order bits of the outputs of G1 and G2 have been computed from the previous vector. Note the glitch in the output of G2. When bit-fields must be split among words the shift operations are considerably more complicated, since bits must be transferred from one word to another. Our experimental results show a jump in execution time when bit-fields become large enough to be split. (See section 7.)

6. Optimizing of the Parallel Technique.

Most of our efforts have gone towards eliminating the shift operations that follow each gate simulation. Much of the simulation time for deep circuits is consumed by these operations, so eliminating them can be quite profitable. To see why these operations can be eliminated, consider why they are necessary in the first place. Each bit-field is aligned so that the low-order bit represents time zero. Since a gate simulation requires one time unit, the unshifted bit-field is aligned so that the low-order bit represents time 1. Thus it is necessary to shift the resultant field so its alignment matches that of the output nets. By

allowing different bit-fields to have different alignments shift operations can be eliminated. For example, consider the network pictured in Figure 2. Because the smallest element of the PC-set of net-E is 1, the alignment of net-E's bit-field must be 1 or smaller. If we set the alignment of this bit-field to 1, we can then set the alignment of net-C and net-D to zero. Finally, we set the alignment of nets A and B to minus one. All shifts have now been eliminated, and the width of all bit-fields can now be reduced from three to two . Although negative alignments seem peculiar, they can be handled by making a small change to the procedure for reading input vectors. When a primary input is assigned a value from an input vector, the previous value of the primary input is copied into all bits whose index is negative. The new value is propagated through those bits whose indexes are zero or positive. Figure 5 illustrates the code generated by this technique, and the bit-fields that are computed by the generated code.

```
char A,B,C,D,E;
…
D = A & B;
E = D & C;
…
```

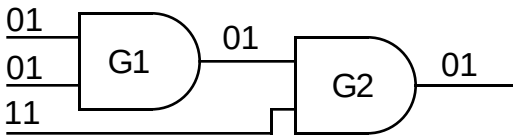


Figure 5. Optimization of the Parallel Technique.

If all shifts are to be eliminated, alignments must be adjusted so the following four conditions hold.

1. The alignment of every net is less than or equal to its minlevel.
2. All input nets of a gate must have have the same alignment.
3. All output nets of a gate must have the same alignment.
4. The alignment of the output nets of a gate must be is one larger than the alignment of the input nets.

Unfortunately, for most networks it is impossible to force all four conditions to be true simultaneously. Figure 6 gives an example of such a network.

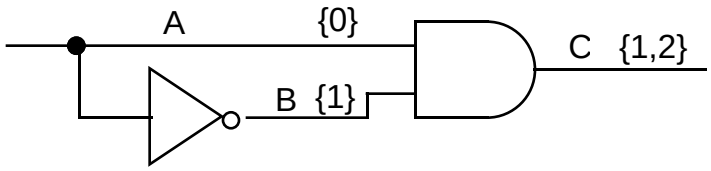


Figure 6. A Network That Requires One Shift.

The reason that conditions 1-4 cannot be enforced for the network of Figure 6, is that condition 2 requires that the alignments of A and B be equal, while condition 4 requires that the alignment of B be one larger than the alignment of A. Because conditions 1-4 cannot be enforced, it is impossible to eliminate all shifts from the simulation code for this network. The shift after the AND simulation can be eliminated, but the shift after the NOT simulation

must be retained. In general, if a network contains reconvergent fanout along differing length paths, then it will be necessary to retain some of the shift operations. However, there are networks without reconvergent fanout that also must retain some shift operations. Figure 7 gives an example of such a circuit.

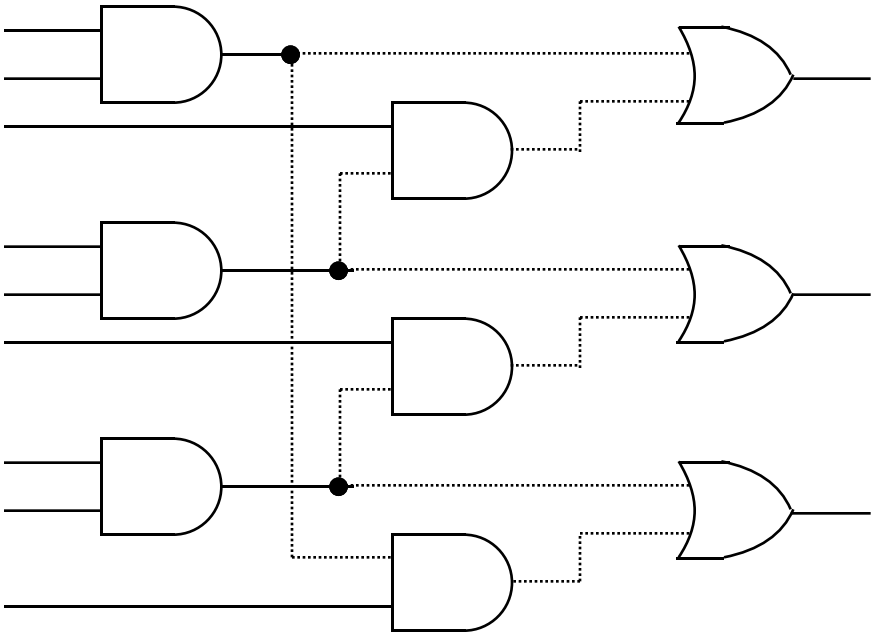


Figure 7. A Network Without Reconvergent Fanout That Requires One Shift.

The dotted lines in Figure 7 illustrate the portion of the network that prevents conditions 1-4 from being enforced. The concept of an undirected network graph is used to characterize such networks. The undirected network graph contains one vertex for each gate and each net in the circuit. All edges in the undirected network graph connect a gate vertex to a net vertex. There is an undirected edge between a gate vertex and a net vertex if and only if the gate uses the net either as an input or an output. Figure 8 illustrates the undirected network graph for the network of Figure 6.

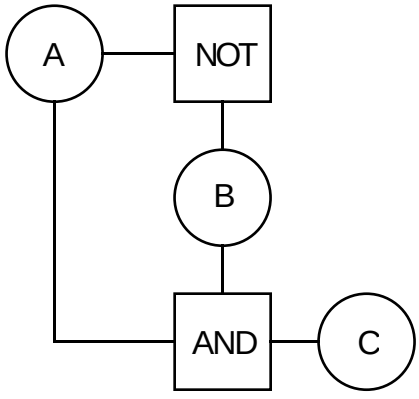


Figure 8. An Undirected Network Graph.

Note that the graph of Figure 8 is cyclic. Although the absence of cycles in the undirected network graph is sufficient for conditions 1-4 to be enforceable, the presence of cycles is not sufficient to guarantee that they are *unenforceable*. To determine whether a cycle will prevent conditions 1-4 from being enforced, a further analysis of the cycle is necessary. This is done by traversing the cycle and assigning weights to the vertices. (The choice of direction is arbitrary.) For simplicity, the traversal is assumed to begin on a net-vertex. Each net-vertex is given a weight of zero, while gate vertices can be given given weights of zero, one, or minus one. Assume that when visiting a gate-vertex *G*, the path *NGM* is traversed. (Only simple cycles are considered, so *G* can be visited only once.) The vertices *N* and *M* must be net vertices that correspond to inputs or outputs of *G*. If *N* and *M* are both inputs or both outputs, then *G* is assigned a weight of zero. If *N* is an input and *M* is an output then *G* is assigned a weight of one. If *N* is an output and *M* is an input then *G* is assigned a weight of minus one. The weight of the cycle is the total of the weights of the vertices. A necessary and sufficient condition for a cycle to prevent the enforcement of conditions 1-4 is that its weight be non-zero. Assuming that the cycle of Figure 8 is traversed in a clockwise direction, vertices A, B, and AND will be given the weight zero, and the vertex NOT will be given the weight 1. The cycle will have the weight 1. The choice of the initial direction affects only the sign of the weight, not its magnitude. The cycle represented by the dotted lines in Figure 7 has a weight of 3 or -3 depending on direction.

Each cycle in the undirected network graph requires one shift in the simulation code. The absolute value of the weight of the cycle determines the magnitude of the shift. (Shifts are no longer restricted to one bit. Both right and left shifts may be used.) The minimum number of shifts is equal to the minimum number of cycles that must be broken to convert the undirected network graph into an acyclic graph. In general, this problem is known to be NP-complete, so an approximation algorithm must be used. We have experimented with various techniques for determining net alignments, and have found that the following procedure gives the best results for realistic networks.

The initial alignment of each net is set equal to the minlevel of the net. The remainder of the procedure is carried out using two mutually recursive routines called the gate processor and the net processor. Initially the net processor is called on every primary output in the network. When the net processor is invoked on a net *N* with alignment *a*, it examines every gate that uses *N* as an output, and if the alignment of the gate is greater than *a* it sets the alignment of the gate equal to the alignment of *N* and calls the gate processor. When the gate processor is called on a gate *G* with alignment *a*, the gate processor examines all input nets of *G* and if the alignment of the net is greater than *a*-1, it sets the alignment of the net equal to *a*-1 and calls the net processor. Note that alignments are never allowed to become larger. This guarantees that condition 1 will be true for all nets. Furthermore, for each gate, the alignment of all inputs will be less than or equal to the minimum of the minlevels of all inputs. The alignment of a gate is guaranteed to be equal to the alignment of at least one of its outputs.

We have also experimented with general cycle-breaking algorithms that operate on the undirected network graph. Although these algorithms give better results for pathological circuits such as that pictured in Figure 7 (one three-bit shift as opposed to three one-bit shifts), they tend to perform much worse on more realistic circuits. We conjecture that this is because most cycles are caused by reconvergent fanout rather than configurations such as that pictured in Figure 7. The algorithm described above appears minimize the number of breaks by pushing them closer to the point of fanout. Furthermore, general cycle-breaking algorithms tend to greatly expand the size of the bit-fields by using very small alignments for many nets. The procedure described above does not expand the size of the bit-field, and may reduce it.

Once alignments have been determined for all nets, the width of each bit-field is calculated using the formula *level-alignment*+1. Since it is more convenient for all bit-fields to be the same width, the maximum bit-field width is used for all fields. When code

is generated for a gate, it is assumed that all operands have been pre-aligned to the alignment of the gate minus 1. After code is generated to simulate the gate, the alignments of those gates that use the current gate's outputs are examined, and any necessary shifts are generated.

Although the results we have obtained so far are encouraging, we believe there is more work to be done on the optimization phase. In particular if very small negative alignments are used for the primary inputs, there will be much redundant computation performed. It may be advantageous to reduce the size of the bit-field in a portion of the generated code at the expense of performing a few more shifts. This will be advantageous only if the number of words per bit-field can be reduced. We are also actively investigating other methods of breaking cycles and performing alignments.

7. Experimental Results.

The following table summarizes some of the results we have obtained from a comparative study of various simulation techniques. The circuits listed are the ISCAS 85 benchmarks which have become a standard for measuring the performance of logic simulators[9]. Each circuit was simulated on 5,000 randomly generated vectors using four simulators of our own design. The first two are conventional unit-delay event-driven simulators, which use a three-valued and a two-valued logic model respectively. Since three-valued logic is the more natural model for event-driven simulation, the first column contains the most realistic figures. The two-valued simulation results were provided to show that the increased performance of the PC-set method and the parallel technique are *not* due to the difference between the logic models used. (Both the PC-set method and the parallel technique simulators use a two-valued logic model.) The figures listed do not include the time required for reading vectors, printing output, or compiling circuit descriptions. For the parallel technique, circuits c432-c1355 required one word per bit-field, while all others except c6288 required two words per bit field. The circuit c6288 required 4 words per bit field.

The simulations for the parallel technique were run *without* optimization.

Simulation Time in Seconds				
	Interp. 3 value	Interp. 2 value	PC-Set Method	Parallel Technique
c432	46.4	41.2	9.9	3.4
c499	51.1	44.3	5.2	4.4
c880	87.1	78.1	22.4	8.1
c1355	177.2	157.7	84.9	9.8
c1908	330.2	295.9	162.7	54.3
c2670	368.2	346.1	89.9	90.7
c3540	531.1	479.1	211.6	122.2
c5315	1024.0	894.7	245.2	176.0
c6288	9555.9	8918.3	1757.3	369.3
c7552	1483.2	1348.5	395.2	269.7

These figures clearly show the advantage of the PC-set method over interpreted event-driven simulation, and also demonstrate the superiority of the parallel technique. The anomaly between the PC-set method and the parallel technique for circuit c2670 is due to the unusually small size of the PC-sets for this circuit. On the average, the PC-set method runs in one fourth the time required for an interpreted event simulation, while the parallel technique runs in about one tenth the time. To put these numbers in perspective, we performed a similar study for zero delay simulation. Our results for zero-delay simulation show that on the average a compiled simulation runs in 1/23 the time of an interpreted simulation. Compared to the figures reported in [5] this number is extremely conservative.

8. Future Work.

One strong appeal of PC-set method is that it is not limited to unit delay simulation. The PC-set generation algorithm will work equally well with nominal delays. For certain types of circuits, this PC-set method could be used virtually unchanged to generate code for compiled nominal-delay simulations. We have conducted a few experiments with nominal delay simulation, and have obtained acceptable results when the number of different gate delays is restricted to a small set. Such a simulation could be of value when simulating a circuit consisting of instances of library cells, where all instances of the same type are assumed to have the same delay. For simulations in which all gate delays are different, we have found that this procedure leads to prohibitively large PC sets. For example, an 8x8 array multiplier with random gate delays had some PC-sets with over a million elements. More research is needed in this area.

Unlike the PC-set method, the parallel technique is firmly tied to unit-delay simulation. However, it could be used for more general timing simulation if all gate delays were small integers. In theory, the technique could also be used for more general timing simulation if a small enough time-scale were used. Unfortunately this could result in extremely wide bit-fields, which would have to be split into many variables. This would probably negate the beneficial effects of parallel simulation. We plan to conduct experiments in the near future to confirm these suppositions. We are also investigating the possibility of using a non-uniform time-scale in the bit-fields, but as yet this has not proven to be practical.

An intriguing avenue for future research is the possibility of implementing the parallel technique directly in special-purpose hardware. Such hardware could be constructed with a very wide data-path, because none of the operations used by the simulator require carry propagation. In such hardware, the left-shift and the OR operation could be implemented quite simply as part of the data path, eliminating the need for explicit operations for these functions. Special-purpose hardware could eliminate the need for the optimizations described in Section 6.

Both the PC-set method and the parallel technique are oriented towards synchronous circuits. We are currently investigating the problem of handling asynchronous circuits and hope to have results in the near future.

9. Conclusion.

Two techniques for compiled unit-delay simulation have been presented. Experimental data shows that these techniques provide significant performance improvements over event-driven interpreted simulation. The PC-set method of generating code is quite general, but is slower than the parallel method, and in some cases generates much larger programs than the parallel technique. Although the parallel technique provides the fastest simulations, there are significant opportunities for optimizing the code produced by this technique. Work is currently under way to extend these techniques to nominal-delay simulation and to adapt them to asynchronous circuits. Nevertheless, these techniques can be used without change for a wide variety of circuits. Even if more powerful techniques are discovered in the future, this work demonstrates that compiled simulation is a powerful concept that can be used to improve the performance of most, if not all types of simulation.

REFERENCES

1. 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.

2. 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.

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

4. 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.

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

6. Wang, Z. and Maurer, P. M., "Scheduling High-Level Blocks for Functional Simulation," Proceedings of the 26th Design Automation Conference, 1989, pp. 87-90.

7. Chappell, S. G., H. Y. Chang, C. H. Elmendorf and L. D. Schmidt, "Comparison of Parallel and Deductive Simulation Techniques," IEEE Transactions on Computers, Vol C-23, pp. 1132-1139.

8. R. E. Bryant, "Data Parallel Switch-Level Simulation," IEEE International Conference on Computer Aided Design, 1988, pp. 354-357.

9. F. Brglez, P. Pownall, R. Hum, "Accelerated ATPG and Fault Grading via Testability Analysis," Proc ISCAS-85, pp. 695-698.