Levelized Compiled Code Multi-Delay Logic Simulation

Peter M. Maurer
Dept. of Computer Science
Baylor University
Waco, TX, 76798
Peter_Maurer@Baylor.edu

ABSTRACT
Levelized Compiled Code (LCC) multi-delay simulation is an idea whose time has come. Although it is not a new idea, when it was first proposed the hardware and software technology of the time was not capable of meeting the demands of such a simulation technique[1]. The basic technique is to predict the points in time when each gate can change value, and generate simulation code to compute the output of the gate at those times. For multi-delay simulation in which each gate has an integer delay greater than or equal to one, many circuits require several megabytes of straight-line code. Running such code was technologically infeasible when the technique was first proposed. However, technology has caught up with the algorithm, and it is now possible to determine whether oblivious multi-delay simulation is a viable method of logic simulation. Our experimental data shows that oblivious multi-delay simulation is many times faster than event-driven simulation for typical circuits, and significantly faster even with extremely demanding simulation parameters.

Author Keywords
Logic Simulation; Oblivious Simulation; Multi-Delay Simulation; Levelized Compiled Code Simulation

ACM Classification Keywords
I.6.8 TYPES OF SIMULATION

INTRODUCTION
Many people feel that levelized compiled code (LCC) simulation, an oblivious simulation technique, is the fastest form of logic simulation [2]. Logic simulation treats a digital circuit as a collection of gates, and simulates each gate as a function of its inputs. Many different logic simulation algorithms have been discovered [3-13]. Basic LCC simulation uses the zero-delay model in which gates are treated as pure functions with no internal delay. In [1] Levelized Compiled Code simulators were developed for the unit-delay model, in which all gates are assumed to have identical delays. When the unit-delay LCC simulators were first developed in the early '90s there was also some interest in developing an LCC multi-delay simulator, which would permit gates to have differing integer delays. However, there were several factors that made multi-delay LCC simulation virtually impossible. In any form of delay simulation, the output of a gate may change several times during the simulation of a single input and simulation code must be generated for each such potential change. For multi-delay simulation, this yields a huge amount of generated code. For one test circuit, the generated code was 50 megabytes and required several days to compile. At the time, computer memory sizes were something on the order of 4-8 megabytes, making it impossible to execute the compiled code. The generated code was essentially all straight-line code. Few if any, caches of the early '90s were pre-paging. Such caches strongly favor tight loops that fit completely in the cache. Even if it had been possible to execute 50 megabytes of straight-line code, it would have been too slow to compete with event-driven methods.

But things have changed since the early 90's. Today 50 megabytes of code is nothing. Memories are now many gigabytes in size. Compiling 50 megabytes of code now takes about two minutes. And best of all, virtually all caches are now pre-paging. The advantage of having small tight loops is much lower that it was in 1990. It is time to take a second look at multi-delay LCC simulation.

THEORETICAL FOUNDATION
Logic simulation treats digital circuits as networks of gates. A gate is a small circuit that computes a Boolean function such as AND or OR. The wires that connect gates to one another, and to the outside world are called *nets*. Nets that receive input from the outside world are called *primary inputs*, and those that produce results for the outside world are called *primary outputs*. The entire circuit description is called a *network*.

Our oblivious multi-delay simulator is based on a concept called Potential Change Sets, or *PC-sets*. The PC-set of a net gives a global picture of how changes on a net can take place. Consider the circuit of Figure 1. A, B, and C are primary inputs. We assume that these can change only at time zero. (Our simulator is not restricted to this assumption.) Because gate G1 has a delay of 2, D cannot change until time 2. In fact, because the inputs of G1 can change only at time 0, D can change only at time 2. G2, however, has differing length input paths. Two through G1, and one through the primary input C. E can change at time 3, because C can change at time 0. But because D can change at time 2, E can also change at time 5. E cannot change at any other times other than 3 or 5. The PC-set is used to capture this information for each net in the circuit. It is, essentially, a list of the differing length paths (in terms of delay) between the primary inputs of a circuit and a net.

SpringSim-TMS/DEVS, 2015, April 12 - 15, Alexandria, VA, USA
© 2015 Society for Modeling & Simulation International (SCS)

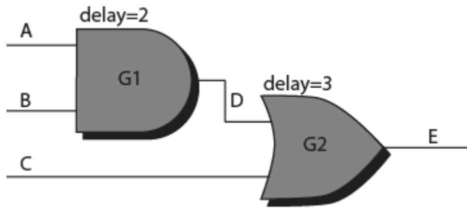
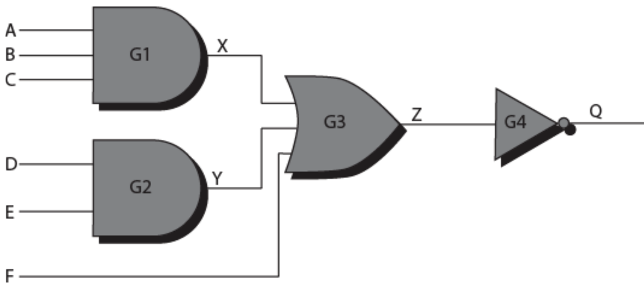


Figure 1. A Sample Network.

GENERATING PC-SETS

The first step in computing PC-sets is creating the companion graph of a gate network. A vertex is created for each gate and each net in the circuit, as illustrated in Figure 2. It is not possible to use nets as edges, because net fanout would yield edges with more than two ends. Also, proper handling of wired-or and wired-and connections makes it necessary to treat nets as vertices.



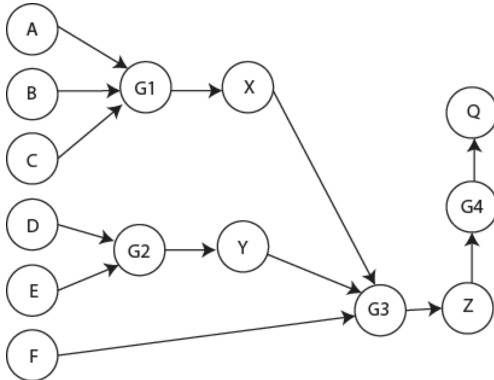


Figure 2. The Companion Graph of a Network.

Note that the resulting graph is bipartite with net-vertices alternating with gate vertices. PC-sets are created by performing a topological search of the companion graph, and a PC-set is assigned to each vertex. Although only the PC-sets of the net vertices will be used to generate code, it is necessary to compute PC-sets for gate vertices to guarantee that the PC-sets of the net vertices are computed correctly. The topological search starts with the source vertices, which represent the primary inputs of the circuit.

The primary inputs of the circuit are assumed to change at time zero, and remain constant throughout the simulation. (This assumption is not essential to the algorithm. If necessary, a few simple modifications will permit more complex changes in the primary inputs.) Each primary input vertex is assigned a PC-Set of {0}, the set containing the single element, zero. If the circuit contains any constant 1 or constant 0 nets, these nets are treated as source vertices with empty PC-sets. However our benchmark circuits contain no constant signals, and in general, these are uncommon.

When a vertex is visited, all preceding circuit elements will have been assigned PC-sets, including the immediate predecessors of the vertex. The first step is to obtain the PC-sets of all immediate predecessors, and form the union of these sets, as shown in Figure 3.

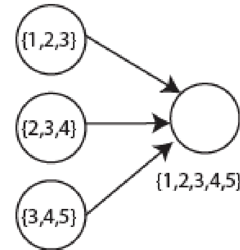


Figure 3. Initial PC-Set computation.

If the current vertex is a net vertex, the union is assigned to the vertex as its PC-set. If the current vertex is a gate vertex, the delay of the gate is added to each element of the PC-set and the result is assigned to the vertex as its PC-set, as shown in Figure 4.

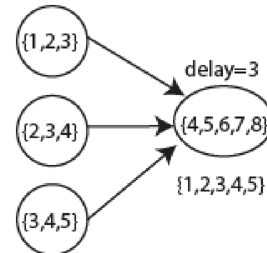


Figure 4. Assigning a PC-Set to a Gate.

Most net vertices will have a single predecessor, which causes the PC-set to simply be copied from the predecessor. The exception is those nets that are wired together to form wired-or or wired-and connections. Although our benchmark tests do not contain connections of this type, their PC-sets can be easily computed as illustrated in Figure 5. In this case, the connection acts as a zero-delay gate, and is treated accordingly.

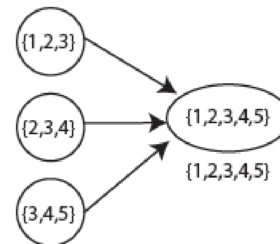


Figure 5. Assigning a PC-Set to a Wired AND/OR.

Once PC-sets have been computed for each vertex of the companion graph, code is generated for each net vertex.

CODE GENERATION

Code is generated using the net vertices of the companion graph in topological order. This guarantees that new signal values are available when they are needed, regardless of the simulated time at which they occur. Thus for the network of Figure 2, all changes in nets X and Y will be computed first, followed by all changes in net Z. The changes in net Q will be computed last. The type of code generated is based on the logic model used in simulating the circuit. In our simulations, we use a two-valued (0, 1) model, however it is also possible to use three valued logic (0, 1, Unknown) or four-valued logic (0, 1, Unknown, High-Impedance). It is also possible to mix logic models, using different logic models for different nets.

When generating simulation code for a net, one simulation statement (or set of statements) must be generated for each element of the PC-set. To simplify the presentation, we will assume a two-valued transport delay model.

The first step is to generate a variable for each member of each PC-set. To do this correctly, it is necessary to compare the PC-sets of the nets attached to a single gate and add a zero element to some PC-sets. Figure 6 illustrates the problem. To compute the value of net C at time 4, we need the values of nets A and B at time 1. However, as it stands, net B has no PC-set value for times earlier than time 3. To correct this problem, we add a zero to the PC-set of B, which represents the value of net B from the previous input vector. It is necessary to generate a variable to hold the time-zero value, but the value of the variable is computed differently from other net values. Zero insertion is performed for the inputs of every gate in the network before any variables are generated. A gate-input, *A*, requires zero insertion if there is another input, *B*, of the same gate, and the smallest PC-set value of *B* is smaller than the smallest PC-set value of *A*. The variables generated for the nets pictured in Figure 6 would be A1, A2, A3, B0, B3, B4, B5, C4, C5, C6, C7, and C8.

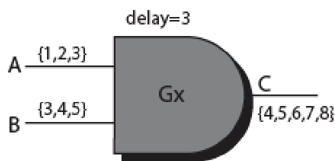


Figure 6. Zero Insertion.

After generating the variable for each PC-set value, the next step is to generate simulation code. Code generation begins with nets that have been subject to zero insertion. It is necessary to copy the value of the variable corresponding to the largest PC-set element into the time-zero variable. The variable with the highest PC-set value will automatically contain the final value of the net from the previous input vector. The first line of Figure 7 shows how this would be done for the gate of Figure 6.

Next, simulation code is generated for each PC-set element of the gate output (except zero if it is present). This code consists of a set of assignment statements that compute the function of the gate. The operands of the assignments are determined using the delay of the gate. Suppose the value at time x is being computed for a gate with delay d. Each input of the gate must appear in the simulation statement. If the value for time d-x is available, it is used. If no value for time d-x is explicitly available, the next earliest time is used. The last five lines of Figure 7 show how this would be done for the gate of Figure 6.

During code generation, nets are processed in topological order. All values for all times are computed for a net before the net will be used as an input to another gate. This insures that all net values are available when needed. For a single net, simulation code is generated in ascending order by time.

```
B0 = B5;
C4 = A1 & B0;
C5 = A2 & B0;
C6 = A3 & B3;
C7 = A3 & B4;
C8 = A3 & B5;
```

Figure 7. Code Generation.

OTHER TIMING MODELS

In Figure 7, we have shown code for two-valued transport delay simulation. The oblivious multi-delay technique is not restricted to this timing model. For example, we could have generated code for the three valued model, using the truth tables of Figure 8. Figure 9 shows the code for the value C4. There are more efficient ways to execute the three-valued model, but the example of Figure 9 shows explicitly how the model could be handled.

Similarly, we could also generate code using a four-valued model containing the values 0, 1, U (unknown) and Z (high-impedance). Any optimization used in event-driven simulation can also be used in the oblivious multi-delay simulator.

TRANSPORT DELAY VS. INERTIAL DELAY

The transport delay model is, in some sense, the most pessimistic of delay models. It assumes that every change in a gate's inputs may propagate through the gate to the outputs. In many cases, this is not true. The delay of a gate is not simply the time it takes for a signal to propagate from the inputs to the outputs, it is the time it takes for the internal circuit elements to transition from one state to another. Because these transitions are not instantaneous, it is not true that all signals that appear on the inputs of a gate will be transmitted to the outputs. Some simulators adopt the policy that no gate can transmit a pulse shorter than its delay. However, the output of the gate does indeed change, and if the output fans out to several different gates, different down-stream gates may see the short pulse differently. We

feel that it is dangerous to ignore these short pulses completely, but can also be desirable to allow the gate to filter out short pulses, at least for certain gates.

AND	0	1	U
0	0	0	0
1	0	1	U
U	0	U	U

OR	0	1	U
0	0	1	U
1	1	1	1
U	U	1	U

Figure 8. Three-Valued Logic.

```
if (A1==0 || B0==0)
{
    C4 = 0;
}
else if (A1 == U || B0 == U)
{
     C4 = U;
}
else
{
     C4 = 1;
}
```

Figure 9. Three-Valued Logic.

To filter a short pulse, it is necessary to test several values of a gate. Suppose in Figure 6 we wish to detect a short pulse between times 6 or 7 and time 8 of output C. This would be done as shown in Figure 10. It is possible to optimize this code. We have written it in this form to make it explicitly clear what is going on. If a short pulse, starting at times 5 or 6, has occurred, we flatten the pulse by setting the intermediate values to the starting and ending values. Because simulation code is generated in topological order, no downstream gate will see the short pulse.

```
C8 = A3 & B5;

// check for width 2 pulse

if (C8 == C5 && C6 != C8 && C7 != C8)

{

     C7 = C8;

     C6 = C8;

}

// check for width 1 pulse. Earlier pulses are
already gone.

else if (C8 == C6 && C7 != C6)

{

     C7 = C8;

}
```

Figure 10. The Inertial Delay Model.

Although the testing shown in Figure 10 will increase simulation time, similar tests must be done in an event driven simulator. Before inserting an event into the queue, the simulator must test for the presence of an earlier event for the same net. If such an event is found, the new value of the net contained in the event must be tested against the new value being computed for the current time. Furthermore, if a short pulse is found, it is then necessary to cancel the existing event, which can be more time consuming that simply copying a value from one variable to another.

Oblivious multi-delay simulation has another advantage over event-driven simulation. Because event-driven simulation does not have the global perspective of the PC-set, it is necessary to check for short pulses every time a gate is simulated even if such a pulse could not possibly occur. Suppose a delay-3 gate has an output PC-set of {3, 12, 23, 47}. No short pulses can occur for this net, because changes cannot occur close enough together. For this net, it is safe to generate transport-delay type code, and not check for short pulses.

SEQUENTIAL CIRCUITS

Oblivious multi-delay simulation can be used with combinational circuits and with sequential circuits. The procedure for synchronous sequential circuits is to break the simulation into two phases. In a synchronous sequential circuit, every cycle in the circuit must contain at least one clocked flip-flop. Once the flip-flops are removed from the circuit, the circuit becomes combinational. Our approach does exactly this. Simulation is broken into two phases. During phase 1, the combinational logic of the circuit is simulated using the techniques described above. During phase 2, all flip-flops are simulated. Flip-flops are simulated directly, not as collections of combinational gates with feed-back arcs. This approach serves as an aid to debugging the circuit, because errors, such as a 1,1->0,0 transition in an RS flip-flop can be reported explicitly. Simulating the individual gates would lead to an oscillation which would then have to be traced and diagnosed.

Many synchronous circuits have a few embedded asynchronous elements. These are generally not a problem. However, if an asynchronous flip-flop appears in a cycle that does not contain a synchronous flip-flop, or if feed-back loops appear in collections of combinational gates with no synchronous flip-flop in the cycle, then performing a topological search of the combinational network will be impossible. Both of these conditions are rare. If either of these situations occurs, they can be handled using a technique known as the *convergence algorithm* [14]. The convergence algorithm is an extension of multi-delay oblivious simulation. It can handle any circuit, including asynchronous circuits, with some sacrifice in performance. The details of the convergence algorithm are beyond the scope of this paper.

EXPERIMENTAL DATA

To determine the efficacy of oblivious multi-delay simulation, we performed two sets of experiments using the two-valued transport delay model. The basis of these experiments was the ISCAS85 combinational benchmarks, which have been a standard for measuring simulation performance for many years [15]. The multi-delay simulator is an implementation of the algorithm described in [16]. (The source code of this simulator as well as that for our oblivious simulator is available upon request.) The results of the experiments are reported in Tables 1 and 2 in terms of seconds of execution time. The time to read vectors and print output was eliminated from these results. In the first experiment, each gate was assigned a typical delay with a base delay of 1 and an increment of 1 added for each gate input. For the second experiment, gates were assigned random delays from 1 through 8. Random delays are more stressful, because this increases the probability that two paths to a gate will have differing delays. This will cause more events to be executed, and significantly more code to be generated. To make it clear, the experiment with typical delays reflects the sort of delay patterns that one would expect to encounter in practice. The experiment with random delays uses artificial delay patterns specifically designed to create a stressful experiment. In both cases oblivious simulation significantly outperformed event driven simulation. As is to be expected, the highest gains were for the typical delays. Although there were also significant performance gains for random delays (except for circuit c1908), the gains were more modest. We initially assumed that the gains for this experiment would be non-existent or negative for all circuits, so the performance improvement was a pleasant surprise. Compilation times for the circuits were negligible.

Circuit	Oblivious	Event Driven	Improvement
c432	1.640	13.544	8.26
c499	0.664	12.650	19.05
c880	3.650	23.738	6.50
c1355	9.284	47.294	5.09
c1908	14.454	95.778	6.63
c2670	6.972	116.426	16.70
c3540	48.544	208.076	4.29
c5315	47.522	409.620	8.62
c6288	660.622	3487.456	5.28
c7552	121.662	876.104	7.20

Table 1. Typical Delays.

When evaluating these results, it is important to keep in mind that oblivious simulation times are static, regardless of any changes in the inputs. Our experiments used randomly generated inputs with a large number of changes. As the number of changes in the inputs approaches zero, event driven simulation will eventually begin to out-perform oblivious simulation. However, inputs with few changes tend to do a relatively poor job of verifying the correctness of a circuit. When simulating a thoroughly verified circuit as part of a larger circuit, event-driven simulation may well be preferable. But for the initial design verification of a circuit, we believe that randomly generated inputs will be preferable.

Circuit	Oblivious	Event Driven	Improvement
c432	4.574	16.240	3.55
c499	2.284	30.218	13.23
c880	9.824	27.038	2.75
c1355	55.972	79.276	1.42
c1908	143.770	106.198	.74
c2670	43.408	140.660	3.24
c3540	210.026	257.798	1.23
c5315	214.972	415.552	1.93
c6288	2128.690	6955.960	3.27
c7552	346.032	827.178	2.39

Table 2. Random Delays.

CONCLUSION

The LCC multi-delay simulation technique provides significant performance improvements over event-driven multi-delay simulation. These improvements are evident even when using the technique with highly stressful input. The PC-Set method is adaptable to many different logic models and timing models, making it useful in many different contexts. It also can be used with all but a handful of sequential circuits, and can be extended to handle even these cases. The algorithm is somewhat more complex than zero-delay LCC simulation, but compilation times are still negligible. The PC-Set method is only one of several LCC multi-delay simulation algorithms. It is possible that the other algorithms will show improvements similar to that of the PC-Set method. Regardless of this, the multi-delay PC-Set method should prove a valuable addition to the current repertoire of simulation algorithms.

REFERENCES

1. P. M. Maurer. Two new techniques for unit-delay compiled simulation. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions On 11(9),* pp. 1120-1130. 1992.
2. M. Chiang and R. Palkovic, "LCC simulators speed development of synchronous hardware," *Computer Design,* pp. 87-91, 1986.
3. L. Wang, N. Hoover, E. Porter and J. Zasio, "SS1M: A software levelized compiled-code simulator," in *Proc. 24th Design Automat. Conf.* 1984, pp. 473-478.
4. C. Hansen, "Hardware logic simulation by compilation," in *Proceedings of the 25th Design Automation Conference,* 1988, pp. 712-715.
5. P. M. Maurer, "Efficient simulation for hierarchical and partitioned circuits," in *Proceedings of the 12th International Conference on VLSI Design,* 1999, pp. 236-240.

6. Z. Barzilai, J. L. Carter, B. K. Rosen and J. D. Rutledge, "HSS-A high speed simulator," *IEEE Trans. Computer Aided Design,* vol. CAD-6, pp. 601-617, 1987.
7. P. Maurer, "Software bit-slicing: A technique for improving simulation performance," in *European Design Automation and TEst Conference,* 1999.
8. P. M. Maurer. The inversion algorithm for digital simulation. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions On 16(7),* pp. 762-769. 1997.
9. P. M. Maurer, "Event driven simulation without loops or conditionals," in *Proceedings of the 2000 IEEE/ACM International Conference on Computer-Aided Design,* 2000, pp. 23-26.
10. P. Maurer, "Unit-delay simulation with the EVCF algorithm," in *WorldComp 12/MSV 12,* http://elrond.informatik.tu-freiberg.de/papers/WorldComp2012/MSV4422.pdf, 2012.
11. P. M. Maurer, "A universal symmetry detection algorithm," in *Proceedings of Design Automation and Test in Europe,* 2014.
12. P. Maurer, "Anti-symmetry and logic simulation," in *Proceedings of the Conference on Modeling, Simulation and Verification,* 2013.
13. P. M. Maurer, "Metamorphic differential simulation using the multi-delay timing model," in *Proceedings of the Symposium on Theory of Modeling & Simulation - DEVS Integrative M&S Symposium,* 2013.
14. P. M. Maurer and Y. S. Lee, "Compiled unit-delay simulation for cyclic circuits," in *Proceedings of Southeastcon 92,* 1992, pp. 184-188.
15. F. Brglez, P. Pownall and R. Hum. Accelerated ATPG and fault grading via testability analysis. Presented at Proceedings of IEEE Int. Symposium on Circuits and Systems. 1985.
16. D. Szygenda, D. Rouse and D. Thompson. A Model and Implementation of a Universal Time-Delay Simulator for Large Digital Nets. Spring Joint Computer Conference. pp. 491-496. 1970.