The Inversion Algorithm for Digital Simulation

Peter M. Maurer

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

Abstract

The Inversion Algorithm is an event-driven *algorithm, whose performance rivals or exceeds that of Levelized Compiled code simulation, even at activity rates of 50% or more. The Inversion Algorithm has several unique features, the most remarkable of which is the size of the run-time code. The basic Algorithm can be implemented using no more than a page of run-time code, although in practice it is more efficient to provide several different variations of the basic algorithm. The run-time code is independent of the circuit under test, so the algorithm can be implemented either as a compiled code or an interpreted simulator with little variation in performance. Because of the small size of the run-time code, the run-time portions of the Inversion Algorithm can be implemented in assembly language for peak efficiency, and still be retargeted for new platforms with little effort.*

1. Introduction.

In the past several years there has been much research in improving simulation performance[1-3]. Two basic approaches to simulation have evolved, which are termed *Oblivious*, and *Event-Driven*[1]. Oblivious simulators eliminate scheduling code to improve the performance of gate simulations, but provide no performance improvements for circuits that require few simulations. Event-Driven simulators use scheduling algorithms to reduce the number of gates simulated, but perform poorly when the number of gate simulations is large.

Although event-driven simulation eliminates many gate simulations, it does not eliminate *all* of them. A gate simulation is useless if it does not produce a change in any monitored net, a net visible to the user. In event-driven simulation, a gate is simulated only if its inputs change value. Even if the inputs of a gate change, the input change may not result in an output change for the gate, or the change in the output may be absorbed before reaching a monitored net.

The Inversion Algorithm was designed to reduce useless simulations of the first kind, those gate simulations that do not result in a change in the output of the gate. Eliminating useless simulations of the second kind is significantly more difficult.

2. An Overview of The Inversion Algorithm.

The Inversion Algorithm will not schedule a gate for simulation unless its output is will change value. When the Inversion Algorithm processes an event, it performs tests to determine whether the output of the gate will change. While it is not immediately clear that this is more efficient than simulating the gate, the tests can be streamlined to an extent that would not be possible in a traditional event-driven simulation. Tests must be individualized for different gate-types, but one of the consequences of never simulating a gate unless its output changes is that different gate-types may appear to be identical during simulation.

The current implementation of the Inversion Algorithm supports the 8 gate types AND, NAND, OR, NOR, XOR, XNOR, NOT, and BUFFER. The tests for the XOR, XNOR, NOT, and BUFFER gates are trivial, because any change in an input of one of these gates implies a change in the output. For XOR and XNOR gates the simulation can be optimized further by testing to see whether the gate is already scheduled for simulation. When this is the case, the two simulations will cancel each other leaving the output unchanged. This allows both simulations to be eliminated.

The tests for AND, OR, NAND and NOR based on the counting algorithm originally discovered by Schuler[4]. In a traditional simulation, the counting algorithm operates as follows. This algorithm assumes that there has been a change in an input X to the gate G. The dominant value is 1 for OR and NOR gates, and 0 for AND and NAND gates.

```
if Value.of.X = Dominant.Value.of.G then
begin
      Count.of.G := Count.of.G + 1;
      if Count.of.G = 1 then
            Output.of.G := Dominant.Value.of.G;
      endif;
end
else
begin
      Count.of.G := Count.of.G - 1;
      if Count.of.G = 0 then
            Output.of.G := NOT Dominant.Value.of.G;
      endif;
endif;
```

The counting algorithm assigns a value to the output of G only if the output changes value. This algorithm is extremely efficient because it uses the value of a single input and a count rather than using all input values to compute the output. The Inversion Algorithm uses a highly optimized form of the counting algorithm to determine whether the output of an AND, NAND, OR, or NOR gate will change. As with XOR and XNOR gates, when two simultaneous output changes occur, they cancel one another and both simulations are eliminated.

Unlike traditional event-driven simulation, the Inversion Algorithm uses the counting procedure during the event processing phase, rather than the gate simulation phase. Modifying and testing counts increases the amount of work that must be done during event processing, but the amount of work required during the gate simulation phase is minimal. It is not necessary to test the output of a gate to determine whether an event has occurred. Since any time a gate is simulated, its output is changes either from one to zero or from zero to one, the only operation required to simulate a gate is the inversion operation.

Surprisingly, it is possible to eliminate most gate simulations entirely. Because the correct operation of the Inversion Algorithm *does not require net-values,* it is possible to eliminate all net values and simulations, except when the net is monitored. To schedule XOR, XNOR, NOT, and BUFFER gates, the only information needed to schedule a simulation is that an input has changed value. The input values for AND, NAND, OR and NOR gates are also unnecessary. The counting algorithm increments the gate count when a net value changes from the non-dominant value to the dominant value, and decrements the gate count when the reverse change occurs. Because a net is processed only when its value changes, and because the value of each input alternates between the dominant and the non-dominant value, the only information for correct processing of AND, NAND, OR, and NOR inputs, is whether the next counting operation is an increment or a decrement. Because this can be done in a way that does not require testing, the counting technique used by the Inversion Algorithm is much more efficient than the procedure illustrated above.

In the Inversion Algorithm, each event is represented by the structure illustrated in Figure 1.

next shadow
previous shadow
subroutine
first fanout branch
last fanout branch
queue address
lock address

Figure 1. The Structure of a Shadow.

The element "subroutine" illustrated in Figure 1 is a pointer to the code to be executed to process the event. For AND and OR gates, two event-processors are created, one that increments the count, and another which decrements the count. These routines toggle the routine pointer, so the two routines are executed alternately for any particular net.

Since the simulator does not need to keep track of the value of gate outputs, the presence of an inverted gate output can simply be ignored. Thus the processing for AND and NAND gates is identical, as is that of several other pairs of gate-types. Because the increment and decrement operations performed for AND and OR gates are with respect to the dominant value of the gate, the processing for these two gates is identical once net-values are eliminated. Those few inversion operations that are required can be performed in the event-processing phase of the Inversion Algorithm, eliminating the need for a separate gate-simulation phase.

In reality, the Inversion Algorithm performs *no* traditional gate evaluations, but simply processes a series of events. In traditional event-driven simulation, each event corresponds to a change in a single net. However, in the inversion algorithm each event corresponds to a change in a single fanout branch of a net. Thus a single event in a traditional event-driven algorithm may correspond to several events in the Inversion Algorithm. Figure 2 illustrates why this is necessary.

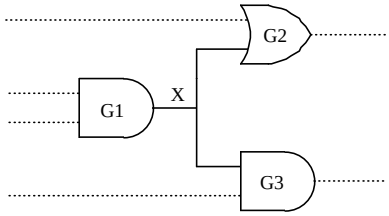

Figure 2. A Circuit Fragment.

In Figure 2, the output of gate G1 is the input for two additional gates G2 and G3. However, when an increment operation is performed for G2, a decrement operation must be performed for G3, and vice-versa. Although both of these actions could be performed during a single event, it is more convenient to incorporate them into separate events. Thus the units of scheduling in the Inversion Algorithm are fanout branches.

Because the Inversion Algorithm computes only changes in signals, it is necessary to assign each signal a value before simulation begins, and it is necessary that the assignment of values be consistent with the structure of the circuit. For example, the input and output of NOT gates must have opposite values. Determining the initial operation for AND, NAND, OR and NOR fanout branches, and computing initial gate-counts requires net values to be consistent. Computing initialization values

does not depend on the timing model, so an interpretive zero-delay simulation can be performed immediately after parsing. This operation is performed only once during the lifetime of the circuit.

3. The Translation Phase.

The Inversion Algorithm consists of two major phases, the Translation Phase which prepares the circuit for simulation, and the Simulation Phase which performs the simulation. Once the circuit has been parsed, it is levelized, the gates of the circuit are sorted into levelized order, and each gate is simulated once to create initialization values. A "simulation" fanout branch is added to each monitored net to generate net values. Finally a data structure known as a shadow is generated for each fanout branch of each net in the circuit.

Figure 1 illustrates the structure of a shadow. The event list is doubly linked to facilitate fast cancellation of events. The subroutine field points to the event processing routine for this fanout branch. The first and last fanout branch fields contain pointers to the first and last shadow that will be scheduled when the output of the gate associated with this shadow changes value. Because all fanout branches of a net must be scheduled or descheduled simultaneously, the corresponding shadows are statically linked during the translation phase. The subchain of shadows is inserted as a group into the event list. The lock address field contains the address of the counter for the gate associated with the fanout branch. For NOT, BUFFER, XOR and XNOR gates, this field is unused. Finally, the queue address is the address of the queue into which the shadow is to be inserted.

4. The Simulation Phase.

Eight different event processors are used during the simulation phase of the Inversion Algorithm. These occur in pairs and are called INCREMENT, INCREMENTX, DECREMENT, DECREMENTX, NOT, NOTX, XOR and XORX. For each pair, the first routine is used for nets that are not at the end of a subchain while the second is used at the end of the subchain to delete the subchain after processing. These eight routines, which consist of less than a page of code each, constitute the major portion of the run-time code of the Inversion Algorithm. (The other components are Vector I/O and Input testing.)

The INCREMENT and DECREMENT processors are used for AND, NAND, OR, and NOR gates. If the initialization value of an AND or a NAND input branch is zero, then the decrement processor is used, otherwise the increment processor is used. Similarly, if the initialization value of an OR or a NOR gate is one, then the increment processor is used, otherwise the decrement processor is used. The initial value of the gate count is computed by counting the dominant values of the inputs of the gate.

Existing implementations of the Inversion Algorithm are based on the zero-delay simulator, LECSIM, developed by Wang[3]. This implementation was chosen because it provides the most direct comparison with oblivious levelized compiled code simulators, which are viewed as the primary competition of the Inversion Algorithm. For details on this algorithm, see [3].

5. Optimizations.

There are several optimizations that can be used to significantly improve performance. The most important of these are eliminating of NOT and BUFFER gates, eliminating of XOR and XNOR gates, collapsing homogeneous connections, and collapsing heterogeneous connections.

It is possible to eliminate all NOT and BUFFER gates from an Inversion Algorithm simulation. When the input of a NOT or BUFFER gate is processed, the only action that is taken is scheduling the fanout branches of the output of the gate. The same effect can be achieved by simply not scheduling the input of the gate, but scheduling its fanout branches instead. It is necessary to retain the NOT and BUFFER gates during the generation of initialization values for all signals. Once this operation is complete, the translation phase may simply *ignore* all NOT and BUFFER gates.

Furthermore, the only action taken when processing an input to an XOR or an XNOR is to schedule or deschedule its output branches. One can eliminate the processing of the input branch by scheduling or descheduling the output branches of the gate instead. Because this can interfere with block scheduling of fanout branches none of the current implementations use this optimization.

It is possible to eliminate much of the processing for other types of nets in the circuit. Assume for the moment that all NOT and XOR gates have been eliminated from the circuit. This leaves only the AND/OR gates, AND, NAND, OR, and NOR. Any connection between these types of gates can be classified as heterogeneous or homogeneous, but the classification must be done before eliminating NOT gates from the circuit. Let A and B be two AND/OR-type gates, and suppose the output of A is one input to B. Suppose a change in the input of A is propagated to B. This will cause the counters for both gates to be incremented or decremented. If both counters move in the same direction, then the connection is homogeneous, otherwise it is heterogeneous. A connection between two AND gates is homogeneous, but a connection between two NAND gates is heterogeneous because of the inverted output. In general, an intervening NOT gate transforms a homogeneous connection into a

heterogeneous connection, and vice versa. Two consecutive NOT gates cancel one another.

A homogeneous connection between gates A and B can be eliminated by treating the inputs of A as if they were inputs of B, and adjusting the initial value of the counter accordingly. It is possible to eliminate all homogeneous connections from a circuit, however it is not always advantageous to do so. If a net fans out to more than one gate, then collapsing it may decrease performance rather than increasing it. It is always advantageous to eliminate homogeneous connections for nets that do not fan out.

It is possible to eliminate some homogeneous connections in a conventional simulator. AND-AND and OR-OR connections can be eliminated by treating the combination of gates as a single AND or a single OR gate. The Inversion Algorithm permits NAND-OR and NOR-AND connections to be eliminated without restructuring the circuit.

It is also possible to eliminate some or all of the heterogeneous connections in a circuit, however the procedure is more complex and does not always completely eliminate all the operations required to simulate these connections. (As with homogeneous connections, it is advantageous to eliminate a homogeneous connections only for those nets that do not fan out.) There are two procedures for eliminating heterogeneous connections, which are called the *linear method*, and the *layered method*. Due to lack of space, only the layered method will be discussed here.

Unlike collapsing homogeneous connections, which eliminates gate-counts for collapsed gates, the layered method preserves the gate-counts of each of the original gates. The same number of tests are required as for uncollapsed connections, but the tests are done without any intermediate scheduling. To illustrate, suppose that a heterogeneous connection gates A and B has been collapsed. Suppose further that there is a change in an input to A that would cause A's count to be decremented. This operation proceeeds as it would before the connection were collapsed, but if the new count is zero, then the count for gate B is incremented. If B's count is incremented from zero, then the output branches of B are scheduled. The shadow for a layered connection differs from that of a simple connection in that the "Lock" component of the shadow is an array of pointers rather than a single pointer.

6. Performance Evaluation.

Four prototype simulators were constructed, an unoptimized version, one which eliminates NOT and BUFFER gates, one which eliminates NOTs, BUFFERs, and homogeneous connections, and one which eliminates NOTs, BUFFERs, homogeneous connections and heterogeneous connections. All are event-driven zero-delay simulators based on the LECSIM model.

The prototypes were certified to produce correct results by running them on the ISCAS-85 benchmarks[5]. Each circuit was simulated with 5000 randomly generated input vectors using the four prototypes and a Levelized Compiled Code simulator. The results of these experiments are reported in Figure 3. The numbers are seconds of CPU time on a SUN IPC.

Circuit	Unopt.	NOT Elim.	Hom. Elim.	Hom/Het Elim.	LCC	Activity.
c432	1.7	1.6	1.4	1.2	0.5	59.4
c499	2.0	1.9	1.9	1.9	0.6	63.2
c880	3.8	3.5	3.2	2.7	1.2	57.1
c1355	6.5	5.4	5.4	4.2	1.9	56.5
c1908	8.1	5.8	5.6	4.5	4.4	56.8
c2670	17.7	13.2	12.2	11.7	5.3	55.7
c3540	16.5	11.6	10.0	9.3	8.4	52.4
c5315	36.9	28.8	28.1	22.8	21.7	63.8
c6288	40.4	40.0	39.7	33.8	30.1	61.5
c7552	52.6	40.6	39.4	33.5	40.7	60.7

Figure 3. Experimental Results.

7. Conclusion.

As Figure 3 indicates, the activity rates of the circuits tested ranged from just over 50% to over 60%. At this level of activity, Levelized Complied Code simulation (the LCC column) typically outperforms event driven simulation by a significant margin. However, for the Inversion Algorithm with deletion of Homogeneous and Heterogeneous connections, the timings are essentially the same for the circuits c1908, c3540, c5315, and c6288. For circuit c7552, the Inversion Algorithm actually outperforms Levelized Compiled Code simulation.

8. References.

1. 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.
2. P. M. Maurer, "The Shadow Algorithm: A Scheduling Technique for Both Compiled and Interpreted Simulation," IEEE Transactions on Computer Aided Design, in press.
3. Z. Wang and P. M. Maurer, "LECSIM : A Levelized Event Driven Compiled Logic Simulator," Proceedings of the 27th Design Automation Conference, 1990, pp. 491-496.
4. D. Schuler "Simulation of NAND Logic," *Proceedings of COMPCON 72*, Sept. 1972, pp. 243-245.
5. 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.