Incremental-in-Time Algorithm for Digital Simulation

Kiyoung Choi, Sun Young Hwang, and Tom Blank

Center for Integrated Systems
Stanford University

Abstract

Recently, an incremental algorithm (*incremental-in-space* algorithm) for digital simulation has been studied with good results in speeding up simulation. In this paper we present another algorithm (*incremental-in-time* algorithm) for incremental simulation of digital circuits. The *incremental-in-space* algorithm pessimistically resimulates the circuit components that could be affected by design changes throughout the simulation time frames. On the other hand, the *incremental-in-time* algorithm resimulates a circuit component only for the simulation time frames when its inputs or internal state variables make different state transitions from the previous simulation run. It maximally utilizes the past history thereby reducing the number of component evaluations to a minimum. Experimental results obtained for several practical circuits show speedups up to 30 times faster than conventional event-driven simulation.

1. Introduction

These days, due to hardware complexity, a designer usually repeats debug cycles until a design is completed. In each debug cycle, changes are made and then the design is reverified. During the design, considerable time is taken by this repeated verification. Motivated by the fact that, most of the time, the verification is done with minor changes, an incremental approach to design verification is proposed. Though the incremental approach can be applied to various kinds of verification tools, event-driven functional simulation [1] is a good candidate since it is still one of the most effective tools for verification of hardware design.

In incremental simulation, the information on all events or on selected events occurring during each simulation run is stored into a *state table*. A *state table* is a generic data structure that stores simulation results. It resides in memory or is dumped onto disk after each simulation run. During the next simulation run on the modified design, the information in the *state table* is used and updated with new information to generate the correct results. This technique is shown to be effective even though it has some overhead due to *state table* maintenance.

There are two approaches to incremental simulation of digital circuits. One is the *incremental-in-space* approach [2] where only circuit components that might be affected by design changes are simulated. In this approach, the network is traversed from the location of design changes to mark circuit components that could be affected by the changes. Then all the marked elements are simulated throughout the simulation time frames. The simulation mechanism is simple and is identical with that of a conventional event-driven simulator. In this paper, we present another, called the *incremental-in-time* approach. In this approach, the simulator resimulates only those components (*active* components) whose input values or internal states differ from those in the *state table*, suppressing evaluations of the *inactive* components. (Remember that the *state table* contains the information on the events occurred during the previous simulation run.) Thus, this approach does the absolute least amount of simulation work. However, the more complex simulation mechanism lessens the overall performance gain when compared to a theoretical limit.

In the following sections we will present the *incremental-in-time* concept with a detailed algorithm description and an evaluation of its performance.

2. *Incremental-in-Time* Concept

When the designer makes a change in his circuit, he may do it without significantly changing the logic, or even keep the same logic putting buffer stages or replacing a module with one having the same function. In these cases, during most of the simulation time frames, most of the circuit components (*inactive* components) have the same input values and internal states. Therefore, they generate the same outputs and do not need to be resimulated. The aim is to resimulate only those components (*active* components) whose input values or internal states are different from those of the previous simulation. Of course, *inactive* components could become

active or *active* components could become inactive, depending on the changes in their input values and internal states as simulation progresses.

The rules for a component to change between *active* and *inactive* are:

1. *Inactive* to *active*
   when any input values or internal states have deviated from those of the previous simulation run as a result of an event.
2. *Active* to *inactive*
   when all input values and internal states have merged to those of the previous simulation run as a result of an event.

To detect the change between active and inactive, we need to compare all the input and internal state values of each component with those of the previous simulation run. Therefore, we have to store all the net values (or events), including the internal state values, in the *state table*. (Note that in the *incremental-in-space* algorithm we do not need to store the internal state values.)

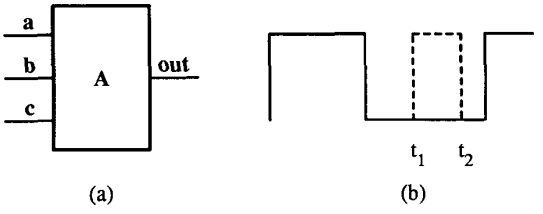

Figure 1: (a) A circuit element
(b) Signal transition at input **a**

Figure 1 describes how a circuit element changes between the *active* and *inactive* states. Figure 1(a) shows a functional element with three inputs (and possibly with internal state variables) and one output. Figure 1(b) shows the signal transitions at **a** in the previous (solid line) and current simulation (dotted line) runs. Assume that all the input values (**a**, **b**, and **c**) and internal states are the same as those of the previous simulation run until time t_1. Element **A** remains *inactive* until t_1. At time t_1 in the previous simulation run, the value of the input **a** does not change and retains value "0". Assume that, at time t_1 in the current simulation run, the input deviates to "1" because of some design changes in its fanin stage. So, element **A** becomes *active* at t_1. Then, at time t_2 in the current simulation run, input **a** merges to "0" and element **A** returns to the *inactive* state (assuming no internal state variables change). In this example, the simulator does not need to resimulate element **A** until time t_1. But it must be resimulated from t_1 to t_2 and the contents of the *state table* are updated. Then at time t_2 it returns to the *inactive* state, and it does not need to be resimulated for the rest of the simulation time frames.

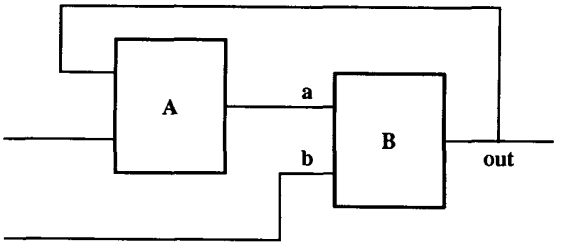

Figure 2: A circuit with a feedback loop

Figure 2 shows an example circuit with a feedback loop. Assume that all the input values and internal states of elements **A** and **B** are the same as those of the previous simulation run until time t_1. Elements **A** and **B** both remain *inactive* until time t_1. Assume that, at time t_1, input **b** of element **B** changes to a value different from the previous simulation run because of some design changes. Element **B** becomes *active* at t_1. However, unless the output **out** deviates, element **A** remains *inactive*. Therefore, element **A** does not need to be resimulated. This example shows the potential power of the *incremental-in-time* algorithm in the simulation of a circuit with feedback loops.

3. Algorithm Description

The *incremental-in-time* algorithm consists of two phases: *token processing* and *simulation*. In the *incremental-in-space* algorithm, after the simulator processes tokens and before it simulates the circuit, it traverses the network to mark components possibly affected by design changes (*net traversal phase*) [2]. However, in the *incremental-in-time* algorithm, there is no net traversal. Instead, circuit components affected by the design change are maintained dynamically in the simulation phase.

In the token processing phase, *net change tokens* are processed to determine how and where the circuit changes have been made. The *net change tokens* inform the simulator of the circuit changes since the previous simulation. They are generated by the network compiler when circuit changes are detected. The network compiler saves the previous circuit description to detect changes by comparing it with the new one. There are various kinds of *net change tokens*: tokens representing connection/disconnection of particular wires, tokens representing insertion/deletion of circuit elements,

tokens for model changes, and tokens for stimulus changes. Through token processing, all the components directly affected by the design changes are permanently (throughout all simulation time frames) set to *active*. The token processing phases in the *incremental-in-time* algorithm and the *incremental-in-space* algorithm are identical.

Simulation does not necessarily start from primary circuit inputs but from the components directly affected by the design changes. Input signals to the active components are then extracted from the *state table*. In some cases, the user may wants to simulate the design longer than the previous run. Assume that he simulated his design until time t_1 in the previous simulation run and now he wants to simulate it until time t_2, where $t_1 < t_2$. He can do it incrementally until time t_1. After t_1, he must simulate the entire circuit because there is no event history stored in the *state table* for the time from t_1 to t_2.

Events are propagated only to *active* components, thereby blocking event propagation to *inactive* components. This mechanism suppresses unnecessary evaluations of *inactive* components. For this mechanism to work correctly, the simulator needs to dynamically manage "in-time" the list of *active* components. This is the main algorithm complexity.

The key concepts in the *incremental-in-time* algorithm are:

1. Maintaining the *active/inactive* component status according to the rules described in Section 2.
2. Event scheduling and propagation for the *active* components.

In addition to the event scheduling and propagation in conventional event-driven simulation, a special mechanism is necessary to properly handle the components in the *active* state.

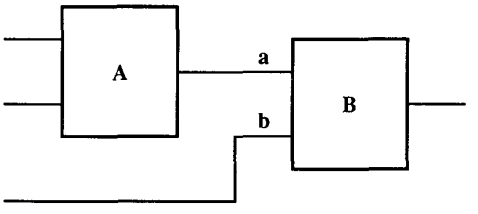

Figure 3: A simple circuit

Figure 3 is a simple circuit demonstrating the event scheduling and propagation mechanism. Assume that element **A** is *inactive* while the signal at the input **b** of element **B** deviates from that of the previous simulation at time t_1, and that the input **a** had an event at time t_2 in the previous simulation run, where $t_1 < t_2$. Element **B** becomes *active* at t_1 and remains *active* until after time t_2. Because element **B** is in the *active* state, it should be evaluated for any events at its inputs until it becomes *inactive*. At time t_2 in the current simulation run, input **a** must have the same event as the one in the previous simulation run because it is connected to an output of the *inactive* element **A**. (Remember that inactive components generate the same outputs as those in the previous simulation run.) In this algorithm, however, the *inactive* element **A** is not evaluated and so the event is not propagated explicitly. For element **B** to be evaluated for the event, there is a special mechanism so that it can be scheduled for evaluation at time t_2, even though there is no explicit event propagation from the previous circuit stage.

The following rules are used to schedule *active* components:

1. Schedule the component for evaluation if there is a new event at any of its inputs. (Scheduling is identical with that of conventional event-driven simulation.)
2. Schedule the component for evaluation at the time of the next potential event. The next potential event is the nearest future event among the events occurring at the inputs of the component during the previous simulation run. In this case, no events are explicitly propagated through its inputs. Therefore the simulator must schedule the component repeatedly until it returns to the *inactive state*.

All of the old events extracted from the *state table* and the new events occurring in the current simulation run are propagated only to the *active* components. Therefore, the event propagations are blocked at *inactive* components suppressing unnecessary component evaluations, which is the main objective of the algorithm.

The overall *incremental-in-time* algorithm is presented below:

Step 1: Process net change tokens setting all directly affected components to *active*.

Step 2: Schedule all *active* components.

Step 3: Deschedule the next component evaluation or net-update.

Step 4: Expand each net-update into component evaluations using the fanout list. (Remember that events propagate only to *active* fanout components.)

Step 5: Evaluate all the descheduled components and new fanout components. (Remember that *active* components are rescheduled by the next potential event.)

Step 6: Repeat Steps 3 to 5 until the event queue is empty.

Notice that Steps 3 to 5 in the *incremental-in-time* algorithm are identical with the simulation algorithm of conventional event-driven simulation except for the slight difference in the event scheduling and propagation mechanism.

4. Results and Performance Evaluation

The *incremental-in-time* algorithm has been implemented based on the THOR functional simulator [3]. Measurements of the simulation run-time have been performed on a VAXstation II/GPX running UNIX™ for several designs. The designs include a 16-bit parallel adder [4], a 16-bit parallel multiplier using a modified Booth algorithm [5], and SETI, a special purpose signal processor developed at Stanford University [6]. For the multiplier, descriptions at both behavioral and gate levels have been used.

Table 1: Comparison of simulation run-times

circuits (level)	num. of circuit elem's	num. of test vectors	conv. sim.	inc-in-space/inc-in-time		
				delta 1	delta 2	delta 3
16b adder (Gate)	140	15000	62.3	20.1/17.9	18.5/6.0	3.5/4.3
16b mult. (Beh)	378	3000	56.0	36.7/22.9	33.0/6.2	14.0/5.6
16b mult. (Gate)	1357	3000	499.9	62.8/74.5	29.5/16.9	18.7/19.6
SETI (Mixed)	566	300	136.3	117.9/11.3	73.6/17.2	14.8/10.8

- Run-time in seconds on VAXstation II/GPX.
- Initialization time excluded.

Table 1 compares run-times for conventional, *incremental-in-space*, and *incremental-in-time* simulations. In this data, the common initialization time, such as net flattening and model instance-specific initialization, has not been included. In incremental simulation, run-time is measured for different circuit changes. The data in the column "delta 1" are obtained when simulating the circuits incrementally for design changes made near primary inputs of the circuits. The data in the columns "delta 2" and "delta 3" are obtained for circuit changes in the middle and near primary outputs of the circuits, respectively. Table 2 shows the number of element evaluations for the same circuits and conditions.

For some cases, the *incremental-in-space* algorithm shows better run-time performance than *incremental-in-time* algorithm. In the multiplier case at the gate level, the performance of the *incremental-in-time* algorithm is worse for the "delta 1" and "delta 3" changes even though it does fewer evaluations. This is partly due to the overhead related to the complicated event scheduling and propagation mechanism. However, the efficiency of the *incremental-in-time* algorithm is exemplified in the simulation of the SETI chip, which has feedback loops spanning a large portion of the circuit. For the "delta 1" change, a speedup approximately 10 times faster than the *incremental-in-space* algorithm has been obtained. Notice that it is faster for the "delta 1" change than for the "delta 2" change. This shows that in contrast to the *incremental-in-space* algorithm where run-time heavily depends on the relative locations of the circuit changes, the *incremental-in-time* algorithm depends on the activity of the circuit components.

Table 2: Comparison of number of element evaluations

circuits (level)	conv. sim.	inc-in-space/inc-in-time		
		delta 1	delta 2	delta 3
16b adder (Gate)	175268	22465/10036	16095/565	2/2
16b mult. (Beh)	163325	49355/7189	43047/181	4856/114
16b mult. (Gate)	1634046	112905/51940	26512/30	1632/3100
SETI (Mixed)	117308	110079/695	81784/3757	6497/543

Another factor causing the performance degradation in the *incremental-in-time* algorithm is the maintenance of a bigger *state table*. In the *incremental-in-space* algorithm used in this performance comparison, we reduced the overhead in maintaining the *state table* by reducing its size. We can achieve good *state table* compaction by storing events selectively. The idea of *selective storage* can be implemented in several ways. Here we used an algorithm based on circuit levelization (see [7] for details). The state table sizes for our examples range from 40 kbytes to 220 kbytes using the compaction technique. In the *incremental-in-time* algorithm, to decide whether a component is *active* or not, the simulator compares its input values and internal state values with those from the previous simulation. Because this mechanism is inherent in the *incremental-in-time* algorithm, all the net values and internal state values must be stored in the *state table*. It is not easy to obtain a good *state table* compaction through the same selective storage techniques as those used in the *incremental-in-space* algorithm, because the net or internal state values that are not stored cannot be easily derived in the *incremental-in-time* algorithm. In current implementation, the state table size is roughly 10 times bigger than that of the *incremental-in-space* algorithm. We can alleviate this problem by compaction through coding of the data in the *state table*, but with some overhead due to the coding.

5. Summary and Conclusions

Incremental simulation is an attractive method for reducing the capture/validate cycle time in hardware design practice. By reducing the number of evaluations over the entire circuit utilizing the past history of simulation, run-time savings can be obtained.

An incremental simulator implementing the *incremental-in-space* algorithm [2, 7] has been successfully used for the functional simulation of practical circuits at Stanford University. This algorithm has its own advantages in that it is simple and requires few bookkeeping tasks, and the overhead inherent in incremental simulation can be reduced to a minimum by employing a *state table* compaction technique. In many cases, it can reduce the simulation run-time significantly. However, it pessimistically resimulates all circuit components possibly affected by design changes throughout the entire simulation time frames. Thus, with circuits that have feedback paths spanning most circuit components or circuits with bidirectional busses where many signal lines are connected, no speedups are achieved.

The *incremental-in-time* algorithm simulates a circuit component only for the simulation time frames when its inputs make different state transitions from the previous simulation run, maximally utilizing the past simulation history and thereby reducing the number of component evaluations to a minimum. Independent of the circuit topology, significant speedups can be obtained in the simulator employing the *incremental-in-time* algorithm. However, this algorithm requires more bookkeeping to maintain the state for each circuit component (*active* or *inactive*) and detecting changes in the state transitions from the previous simulation run. Further, a large storage space for the *state table* is required, because the complete network state throughout the entire simulation time frames should be maintained in the *state table*.

In an experiment with circuits without many internal state variables, speedups up to 10 times faster than the *incremental-in-space* algorithm have been obtained, even with larger overhead due to the complicated event scheduling and propagation mechanism and *state table* maintenance. Compared to conventional event-driven simulation, speedups up to 30 times have been obtained.

Comparing the *incremental-in-space* and *incremental-in-time* algorithms, the first is more efficient when simulating circuits that have circuit components with many internal state variables, as in a microprocessor with control memories. The second algorithm is more attractive when simulating circuits without many internal state variables, especially when the circuits have feedback loops or system-wide busses.

6. Future Work

The large size of the *state table* is one of the limitations of the *incremental-in-time* algorithm. We are looking for an efficient data coding method for compacting the *state table*.

We are planning to implement those two incremental simulation algorithms in the THOR simulator so that the most efficient algorithm can be chosen for the simulation of a particular circuit depending on its structure and topology. More experiments will be performed on circuits with different structures and topologies in the practical world to derive the heuristics for choosing a particular algorithm for a given circuit.

Finally, the incremental simulator will be combined with a schematic capture system and network compiler into an integrated environment so that the capture/validate cycle time for large designs should be less than one minute using the incremental technique.

References

[1] M. A. Breuer and A. D. Friedman, *Diagnosis & Reliable Design of Digital Systems,* Computer Science Press, Rockville, MD, 1976.

[2] S. Y. Hwang, T. Blank, and K. Choi, "Incremental Functional Simulation of Digital Circuits", *Proc. IEEE Int. Conf. Computer-Aided Design*, Nov. 1987, pp. 392-395.

[3] R. Alverson, T. Blank, K. Choi, S. Y. Hwang, A. Salz, L. Soule, and T. Rokicki, "THOR User's Manual: Tutorial and Commands", Technical Report CSL-TR-88-348, Stanford University, Jan. 1988.

[4] R. P. Brent and H. T. Kung, "A Regular Layout for Parallel Adders", *IEEE Trans. Computers*, Vol. C-31, No. 3, Mar., 1982, pp. 260-264.

[5] D. A. Henlin, M. T. Fertsch, M. Mazin, and E. T. Lewis, "A 16 X 16 bit Pipelined Multiplier Macro Cell", *IEEE Journal of Solid-State Circuits*, Vol. SC-20, No. 2, Apr., 1985, pp. 542-547.

[6] J. B. Burr, J. F. Duluk, W. Li, J. D. Twicken, K. Choi, T. Ogunfunmi, B. Ekroot, W. Park, I. Linscott, and A. M. Peterson, "A 20MHz Prime Factor DFT Processor", document in preparation.

[7] S. Y. Hwang, T. Blank, and K. Choi, "Fast Functional Simulation: An Incremental Approach", *IEEE Trans. Computer-Aided Design of Integrated Circ. Syst.*, 1988, to appear.