Incremental Circuit Simulation and Simulation-on-demand based on Backward-traversing Waveform Relaxation and Portion-simulating Algorithms

Chun-Jung Chen
Department of Computer Science Chinese Culture University Taipei, Taiwan
teacherchen62@yahoo.com.tw

Abstract

In this paper, the practical behavior of circuit simulator users are considered and utilized to speed up the circuit simulation. The used methods are incremental simulation and simulation-on-demand (SOD, just simulate the portion of circuit contributing to user-wanted outputs). A specialized relaxation-based algorithm, Backward-traversing Waveform Relaxation (BTWR), is adopted as the fundamental simulation algorithm. All proposed methods have been implemented and tested. Experimental results justify the values of proposed methods.

© 2012 Published by Elsevier B.V. Selection and/or peer review under responsibility of ICMPBE International Committee. Open access under CC BY-NC-ND license.

Index Terms – Circuit simulation; relaxation-based algorithms; incremental simulation; simulation on demand

1. Introduction

Circuit simulation is crucial in circuit design process. Accurate approaches to solve circuit simulation problem include the “standard” simulators (such as SPICE) and relaxation-based simulators (such as RELAX [1], and SPLICE [2]). There exist faster simulation approaches using simpler simulation models (such as piecewise-linear model and switch-level model). But they only provide course waveforms. The trade-off between the solution accuracy and simulation speed firmly exists. In this paper, we propose an alternative strategy to break this trade-off. We want to derive precise waveforms and also save the simulation time. Our strategy bases on the practical behavior of design engineers (or circuit design software), so they are called practical-considering methods. We find that circuit design is an iterative process, so the circuit simulator is always invoked repeatedly. Each simulation process usually simulates the circuit only a little bit different from the previous one. Incremental simulation [3, 4], which utilizes the waveforms of previous simulation to save computation efforts, is indeed valuable. Another observation is that only few outputs rather than all are required by simulator users. If we bypass
subcircuit calculations that have no contribution to the required outputs, we could not only provide satisfactory results but also save simulation time. To sum up, the alternative strategy includes two methods: incremental simulation and Simulation-on-demand (SOD).

We need to pick a fundamental algorithm to undertake the circuit simulation. Backward-traversing Waveform Relaxation (BTWR) [5] is chosen. This algorithm is a specialized algorithm that simulates subcircuits by traversing subcircuits from the rear end to the front end backwardly. So, it’s nature to implement SOD in BTWR. Besides, due to that the traversing is dynamic, the SOD can be implemented more precisely. For example, to consider dynamical situations (such as turning on/off of transistors) is possible.

There exist several strategies for circuit-level incremental simulation. The traditional methods are Incremental-in-space (IIS) and Incremental-in-time (IIT) methods based on ITA and WR respectively. However, these methods re-calculate numerous subcircuits whose waveforms are not changed (in compared to those in previous simulation), which induces considerable waste. We call the subcircuits whose waveforms are different from that of previous simulation are in incremental-changing status. Note that this status is time-varying, for circuits operate dynamically. Another incremental simulation method [3], Incremental-in-change (IIC), is more efficient. It exactly simulates the incremental-changing subcircuits only, but some overheads still retain and worse the simulation efficiency. Therefore, this paper presents a new technique called Portion-simulating Method (PS Method) to perform the incremental simulation. Using PS Method, we can eliminate redundant overheads retaining in IIC. To sum up, this paper researches utilizing PS Method and SOD in BTWR.

The rest of this paper is as follows. In section 2, we explain the fundamental numerical methods used. In section 3, we illustrate the implementation of proposed methods. Section 4 then demonstrates experimental results to show the effeteness of proposed methods. Finally, a conclusion section, Section 5, is given.

2. BTWR and Practical-considering Methods

A. The BTWR Algorithm

The fundamental algorithm is BTWR. The two famous algorithms of relaxation-based algorithms are WR (Waveform Relaxation) and ITA (Iterated Timing Analysis) [1, 2]. BTWR collects advantages of WR and ITA and undertakes circuit simulations efficiently and stably [5].

The basic idea of BTWR is to consider the “cause-and-consequence” concept. The left part of Fig. 1 is a signal flow graph for partitioned subcircuits, in which the transient solution of subcircuit a is obviously influenced by the transient solutions of b and c. Therefore, b and c need to be solved before a in order to raise the computation efficiency.

![Fig. 1 A traversal starting from subcircuit a. Subcircuit b and c are asked to be calculated at time point later then ta.](image)

To trace these cause-and-consequence relations, BTWR uses the backward graph traversal technique. For clarity, we describe variables inside each subcircuit: tc is the time point for which the subcircuit has converged so far, tnow is the current time point to be solved, and ta (time asked) is the time at which the subcircuit is asked to be solved. In Fig. 1, the traversal starts from subcircuit a (tries to solve for solution at a.tnow). The traversal visits a’s fan-in subcircuit b at first and ask it to be solved at time point later or
equal to b.ta (which is also a.tnow) in order to provide waveform references for subcircuit a. This traversal then continually visits c and asks it to be solved at time point later or equal to c.ta (which is also b.tnow) for the same sake. Subcircuit c will forward two time points to move its c.tnow to over c.ta. In Fig. 1, the actual subcircuit calculation sequence would be: c be solved at its two tnow time points, b be solved at its tnow time points, and then a be solved at its tnow time points. This process might repeat several times until a.tnow is converged. The main routine just picks subcircuit with smallest tnow, activates the initial backward traversal from it, and repeats the same process until no subcircuit left.

There are software schemes to handle the feedback subcircuits (including adjacent coupling and global feedback loops, GFL) to strengthen the robustness of BTWR. BTWR exhibits several advantages. First, the multi-rate behaviors of circuits can be exploited. Second, the windowing technique is automatically applied. Third, the function of Selective-tracing Scheme of ITA (to calculate fan-out subcircuits) is retained. Finally and most important for this paper, it’s possible to implement high quality SOD in BTWR.

B. The Portion-simulating Method

An ideal incremental simulation algorithm should only re-simulate incremental-changing subcircuits. But practical incremental simulation algorithms usually simulate some incremental-following (not incremental-changing) subcircuits. These are waste computations. For an ideal incremental simulation algorithm, its “waste ratio” (ratio of waste computations compared to total computations) should be zero. The smaller the waste ratio is, the better the incremental simulation algorithm is. The three major previous works are IIS, IIT, and IIC, in which the last one exhibits the smallest waste ratio. IIC intends to simulate the incremental-changing subcircuits only, and copies corresponding previous waveforms for incremental-following subcircuits. In using IIC, the original simulation algorithm just operates normally except that incremental-following subcircuits’ waveforms are obtained by copying and some additional managing operations are undertaken. So, it still keeps considerable waste computations (which are computation efforts for incremental-following subcircuits).

We propose the Portion-simulating Method, PS Method, to further improve IIC. The basic concept of PS Method is to view the incremental-changing portion of the simulated circuit as the ordinary circuit and view the other portion (incremental-following portion) of the simulated circuit as nothing. The incremental-changing portion of the circuit is called Incremental Portion Circuit (IPC). IPC of the simulated circuit ckt at tn+1 is defined as:

\[ IPC(t_{n+1}) = \{ s | s \in subckt(ckt), s.cstate(t_{n+1}) = "changing" \} \]

Where subckt is the set of all subcircuits, tn+1 is the current time point, and s.cstate is the “varying” situation of a subcircuit (“changing” means that s is incremental-changing). Another value for s.cstate is “following” that indicates s is incremental-following. IPC is composed of many subcircuits which might be not connected, and it is a time-varying circuit whose size and content (the enclosed subcircuits) varies from time to time. PS Method can only see the IPC and simulate it. Therefore, PS Method is more efficient than IIC Method.

PS Method is an abstract strategy. In practical, there exist some requirements for PS-method to be well adopted in real circuit simulation algorithm, which are listed as follows:

Definition 1 (PS Method’s Requirements):
1. IPC should be identified correctly.
2. Only IPC is simulated.
3. Simulation algorithm calculates correct waveforms of IPC.
4. IPC’s waveforms could be combined with those of incremental-following portion well.

To correctly identify IPC, we have to manage the incremental-changing/following statuses of subcircuits. These statuses are propagated from “source” (modified subcircuits or modified primary
inputs) initially. They propagate to succeeding subcircuits or vanish. Mentioning previously, we assign each subcircuit a variable called cstate to dynamically record these “varying” statuses. The following rules manage cstate:

**Definition 2 (Managing Rules for cstate):**

1. **Simulation Rule:** During simulation, only the subcircuit with incremental-changing status is re-simulated.
2. **Subcircuit Design Change Rule:** The subcircuit modified is always in incremental-changing situation.
3. **Input Design Change Rule:** In the time interval that a primary input differs from the same input of previous simulation, all the connected subcircuits are in incremental-changing situation.
4. **Vanishing Rule:** Once an incremental-changing subcircuit has been calculated, the obtained waveform should be compared to that of previous simulation to check whether the incremental-changing status could vanish.
5. **Propagating Rule:** An incremental-changing subcircuit passes the incremental-changing statuses to its fan-out subcircuits.

Managed by above rules, the estate of subcircuits can be processed well, which means that IPC can be recognized correctly. To simulate IPC only, rule 2 of Definition 1, the simulation algorithms then refer cstate of subcircuits to simulate. The rule 3 of Definition 1 can be fulfilled as long as the input waveforms of the incremental-changing subcircuits have been updated correctly and subcircuits are solved normally. To meet rule 4 of Definition 1, we just need a data structure that can process segments of waveforms, for a subcircuit might toggle between incremental-changing and incremental-following statuses repeatedly. This data structure also needs to combine newly-calculated waveforms with previous waveforms well.

**C. The Simulation-on-demand in BTWR**

In BTWR, it’s straightforward to implement the SOD function. BTWR simulates by backwardly traversing a sequence of subcircuits. We just ask the starting subcircuits of backward traversals to contain any wanted circuit variables, and then the SOD function is installed. Note that when the two practical-considering techniques work together, the really-simulated portions of circuits would be further reduced. Fig. 2 shows this phenomenon, where the intersection of “disturbed” and “contribution” zone needs to be simulated only.

**3. Implementation of Proposed Methods**

The proposed methods have been implemented in the experimental circuit simulator MOSTIME [3, 5]. To emphasize the simulation algorithm’s effects, we use the simple analytic model for MOSFET. We partition circuits into smaller subcircuits (basically the CMOS gates), which will induce more feedback relations among subcircuits (to test the robustness of the algorithm). Algorithm 1 lists pseudo codes for the modified-BTWR algorithm that encloses practical-considering techniques.

**Algorithm 1 (BTWR and PS-based Circuit Simulation):**

```
// s.cstate == true means that s is incremental-changing
```
BTWR(ckt, T_begin, T_end) { // Simulation duration is T_begin ~ T_end
Set \( t_c, t_{now} \) of all subcircuits to their initial values;
while (there is any subcircuit whose \( t_c \) is not equal to \( T_{end} \)) {
    Pick the subcircuit \( x \) with smallest \( t_{now} \);
    Sod1:
    if (!contribute[\( x \)]) continue; // the SOD function
    Rtr1: BTWRtrace(0, x, t_{now}, x); // begin to call the traversal
}
}

BTWRtrace(mode, t_{now}, sub) {
    // sub.in_stack flag records whether sub has been traversed
    t_{now} = 0;
    if (mode is 0) Clear all subcircuits' in_stack flag;
    else if (mode == 1) sub.in_stack = 1;
    if (mode is 0) Clear GFL; // the set containing subcircuits of GFLs
    ot_{now} = sub_{now}; // the old t_{now}
    do {
        for (all sub's fan-in subcircuit \( x \)) {
            // backwardly traverse all predecessors
            Rtr2: if (!sub.in_stack) BTWRtrace(1, sub_{now}, x);
        }
        Loop: else { // has encountered a back edge
            Add subcircuits in loops into GFL;
        }
    }
    S1: if (sub is not in GFL) { // simulate sub or all subcircuits in GFL
        Isim: if (sub.cstate) Solve sub at sub_{now};
        else Just modify subcircuit time variables;
        if (results have been converged) {
            sub_{t} = sub_{now};
            Estimate new sub_{now};
        }
        t_{now} = MAX(t_{now}, sub_{now}); // t_{now} is the max. time reached
    }
    Ivan: if (sub.cstate) Go checking whether cstate can be vanished;
    Ipro: if (mode is 1 && !sub.cstate) Go checking whether cstate can be propagated from the preceding subcircuit;
    }
    // simulate GFL
    S2: if (sub is the first subcircuit of GFL) {
        Simulate GFL by using RN algorithm;
        break;
    }
    Sod2: contribute[sub] = true; // the SOD function
    Stop1: if (mode is 0 && sub_{t} >= ot_{now}) break;
    Stop2: else if (mode is 1 && t_{now} >= i_{t}) break;
    } while (true);
}

The backward traversing is accomplished by recursively calling to the function BTWRtrace() (line Rtr1 and Rtr2). After the backward traversing is completed, the subcircuit sub is simulated (line S1). Global feedback loops are recognized dynamically and simulated together by using Relaxation Newton (or called Nonlinear Relaxation) algorithm (line Loop and S2). The SOD function is implemented at line Sod1 and Sod2. There are two simulation modes in BTWRtrace() (mode=0, 1), whose termination conditions are different (line Stop1 and Stop2). Mode-0 is the starting traversal, and Mode-1 are succeeding traversals. Finally, the rules for cstate have been added (line Isim, Ivan, and Ipro, which are for some rules of Definition 2 accordingly).

4. Experimental Results

In this section, we demonstrate experimental results to see the effects of proposed methods. Table 1 shows the specifications of tested circuits and their design changes. Table 2 shows the required outputs.
There are subcircuit-modifying and the input-modifying design changes. Some circuits’ schematics are shown in Fig. 3, which can be referred to understand the design changes.

Five versions of simulations have been performed, whose running results (subcircuit calculation counts and used CPU times) are listed in Table 3 and 4 (respectively). We use the result of BTWR as the comparison basis, where other algorithms’ results can be shown in ratios. Firstly, we check Table 3 to find the saving effects on subcircuit calculation counts, which are quite obvious and pleasing. Due to the

![Fig. 3 Schematics of circuits. (a) Gated inverter chain. (b) Parallel multiplier.](image)

**TABLE I Specifications of Simulated Circuits**

<table>
<thead>
<tr>
<th>Ckt</th>
<th>Name</th>
<th>Node #</th>
<th>MOSFE T#</th>
<th>Subckt #</th>
<th>Design Change</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Gated 100-stage Inv. Chain</td>
<td>102</td>
<td>204</td>
<td>101</td>
<td>Vgate is modified for 6% of duration</td>
</tr>
<tr>
<td>2</td>
<td>4-bit Parallel Multiplier</td>
<td>400</td>
<td>800</td>
<td>112</td>
<td>X0 is modified for 80% of duration</td>
</tr>
<tr>
<td>3</td>
<td>64-bit ALU</td>
<td>3200</td>
<td>6400</td>
<td>1792</td>
<td>One gate in 1st bit has been modified</td>
</tr>
<tr>
<td>4</td>
<td>4-bit Sync. Counter</td>
<td>88</td>
<td>176</td>
<td>44</td>
<td>One gate in 4th bit has been modified</td>
</tr>
</tbody>
</table>

**TABLE II The Wanted Outputs of Circuits**

<table>
<thead>
<tr>
<th>Ckt</th>
<th>Wanted Output Nodes</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>The 20th and 60th inverters’ output</td>
</tr>
<tr>
<td>2</td>
<td>P0 and P1</td>
</tr>
<tr>
<td>3</td>
<td>D0-D7 (least significant 8 bit sum) and most significant Cout</td>
</tr>
<tr>
<td>4</td>
<td>Q0-Q3, and most significant Eout</td>
</tr>
</tbody>
</table>

**TABLE III Subcircuit Calculation Counts**

<table>
<thead>
<tr>
<th>Ckt</th>
<th>Subcircuit Calculation# (K)/Ratio to BTWR's</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>BTWR</td>
</tr>
<tr>
<td>1</td>
<td>278.7</td>
</tr>
<tr>
<td>2</td>
<td>62.51</td>
</tr>
</tbody>
</table>
difference of algorithm characteristics, STWR[3] and BTWR have different saving ratios. But, BTWR+IIC and BTWR+PS have saved similar amounts of counts. SOD has contributed more saving effects. In the first two circuits, SOD has additionally saved much more counts. We should note that contributions of SOD depend on the number and positions of wanted outputs.

Secondly, we check Table 4 to see the used CPU time. We can find the ratios for STWR+IIC are much bigger than those in Table 3. The same phenomenon can be found for the ratios for BTWR+IIC. But in BTWR+PS, the ratios are closer to those in Table 3. The similar ratios of BTWR+PS tell that PS Method has successfully saved waste computations retained in IIC Method.

Next, we check the amount of waste computations of IIC and PS Methods. We define a quantitative index, called Waveform Differentiation Ratio (WDR), as the ratio of the number of “changing” (whose

<table>
<thead>
<tr>
<th>C</th>
<th>kt</th>
<th>Used CPU Times / Ratio to BTWR’s</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>BTWR</td>
</tr>
<tr>
<td>1</td>
<td>2.637</td>
<td>3.104/11%</td>
</tr>
<tr>
<td>2</td>
<td>3.369</td>
<td>0.936/27%</td>
</tr>
<tr>
<td>3</td>
<td>15.42</td>
<td>9.625/62%</td>
</tr>
<tr>
<td>4</td>
<td>1.747</td>
<td>0.924/52%</td>
</tr>
</tbody>
</table>

$^{5}$CPU time is in Pentium 2.5G seconds.
value is different from that of previous waveform) time points to total time points. We then compare the CPU time ratio with it. The more they are similar, the less waste computations have been undertaken. Table 5 shows these comparisons, in which PS Method’s ratios are quite close to WDR. So, PS Method is a good incremental simulation algorithm.

Finally, we demonstrate waveforms of the 3rd circuit in Fig. 4 to see the good accuracy of the proposed method. Since the WDR is small, we find that all waveforms match together.

5. Conclusions

We have proposed an alternative practical-considering strategy to speedup the circuit simulation. We add the incremental simulation and SOD into the flexible relaxation-based simulation algorithm BTWR to implement this strategy. Simulation results justify the success of the proposed methods.

References