Efficient Switching Window Computation For Cross-Talk Noise

Bhavana Thudi
University of Michigan, Ann Arbor, MI
(734) 717 2384

David Blaauw
University of Michigan, Ann Arbor, MI
(734) 763 4526

Abstract

In this paper, we present an efficient method for computing switching windows in the presence of delay noise. In static timing analysis, delay noise has traditionally been modeled using a simple switch-factor based noise model and the computation of switching windows is performed using an iterative algorithm where timing window propagation and switch factor updates are computed repeatedly until convergence. It was shown that the worst-case number of iterations required for convergence is $O(n)$, where n is the number of gates in the circuit, resulting in an overall run time of $O(n^2)$. It was also shown that the iterations converge to different solutions, depending on the initial assumptions, making it unclear which solution is correct. In this paper, we show that the iterative nature of the problem is due to the switching-factor noise model and the order in which events are evaluated. Based on superposition model, we propose a time-sort based algorithm to compute the impact of delay noise on timing windows. We prove that the proposed algorithm has a run time that is linear with the circuit size. Since the algorithm is non-iterative and does not require initial assumptions, it eliminates the multiple solution problem. We tested the algorithm on a number of designs and show that it achieves significant speedup over the iterative approach.

Categories and Subject Descriptions: B.7.2 [**Integrated Circuits**]: Design Aids - *Verification.*

General Terms: Algorithms,Theory, Verification.

1 INTRODUCTION

With increased clock frequencies and larger aspect ratios of the wires due to technology scaling, noise from cross-coupling capacitance between neighboring nets has become a dominant factor in static timing analysis[1][2]. In noise analysis, the net under consideration is referred to as the *victim* net, and the neighboring nets that inject noise on the victim net are referred to as *aggressor* nets. Noise is broadly classified into two types. *Functional noise* occurs when a victim net is intended to be at a stable state and results in an unwanted noise pulse on the net. *Delay noise*, which is the topic of this paper, occurs when noise is injected on a victim net when the victim transitions. Delay noise can cause the delay of the victim net to increase, if the aggressor and victim nets switch in opposite directions, or can cause it to decrease, if they switch in the same direction. For performance analysis of high-speed designs, it is therefore critical that delay noise is accurately accounted for. In order to reduce the pessimism of noise analysis, timing windows[3][4] and logic correlations[5][6] are often computed to determine which aggressor nets can switch simultaneously with the victim net. If an aggressor net cannot switch at the same time as a victim net, the noise from that aggressor net is excluded from the analysis, thereby increasing the accuracy of the analysis. However, the computation of timing windows in the presence of delay noise exhibits a well know "chicken and egg" problem [7]. The delay noise depends on the overlap of victim and aggressor timing windows and the timing windows depend on the delay of the circuit, which, in turn, is impacted by the delay noise. An iterative approach has been used to solve this problem. Initially, timing windows are computed without coupling noise using arrival time propagation in topological order. These arrival times are then updated with delay noise and re-propagated in multiple iterations until convergence.

To evaluate the impact of noise on delay, typically a so-called *switch factor* model is used. For both victim and aggressor nets, the coupling capacitance is replaced with a grounded capacitance, where the value of the grounded capacitance is equal to the coupling capacitance, multiplied by a constant *k*, referred to as the switch factor. If the aggressor and victim nets have the same signal transition time, the switch factor is 2 when the nets switch in opposite directions (for late arrival times) and 0 when they switch in the same direction (for early arrival times). Switching factors for nets with unequal transition times have also been proposed [8][9].

The iterative approach of the timing window computation problem is most clearly explained through illustration. We consider two coupled nets as shown in Figure 1(a) and their initial timing windows, computed without coupling noise and a switch factor of 1. If, after this initial window computation, aggressor and victim timing windows overlap, as shown in Figure 1(b), the switch factor for the computation of the leading (early) edge of window *m* is set to 0 and that for the computation of the trailing (late) edge of *l* to 2. This has the effect of increasing the size of the windows, which can cause other nets in the circuit, which initially did not have overlapping windows, to have overlapping windows. Multiple iterations are therefore needed to reach convergence. Also, an individual pair of nets may require multiple iterations, since with each iteration, the region of overlap between the two nets increases, allowing coupling between later transitions in the windows, as shown in Figure 1(c).

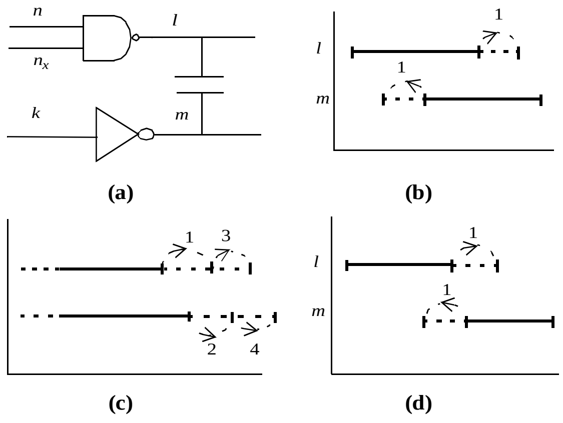

Figure 1. Coupled net and timing window computation. Arrows show change in window size in iteration as numbered.

It was shown in [10] that an iterative computation of timing windows is guaranteed to converge and that the maximum number of iterations is $O(n)$, where n is the number of circuit elements, leading to a worst-case run time complexity of $O(n^2)$. In [11], different scheduling methods are explored to reduce the run time. In practice, the number of iterations is typically less, but can still reach 5 - 10 for circuits with significant coupling, therefore still substantially increasing the run time of static timing analysis. Also, as coupling noise increases with technology scaling, the number of required iterations is expected to grow. It was later shown in [12] that the solution of the iterative computation depends on the assumption used to compute the initial windows. If the initial windows are computed in the absence of delay noise (assuming all windows do not overlap), verses with delay noise (assuming all windows overlap), the iterative approach will converge to different solutions, corresponding to different fixed-points in a lattice representation of the problem. For instance, in Figure 1(d) two windows are shown that do not overlap in the absence of delay noise, whereas they could overlap if the initial windows were computed assuming the presence of delay noise, therefore converging on different solutions. It is unclear what assumption results in an analysis that matches the delay of the actual circuit.

In this paper, we present a new approach for computing circuit performance in the presence of coupling noise, which differs from the existing approach in two fundamental ways.

- Instead of using a switch factor based model for delay noise computation, we use a superposition based model. Superposition has been used extensively in functional noise computation and has also been proposed for delay noise computation [11][13][14][16]. In superposition, the victim and aggressor drivers are represented by linear models and are simulated individually, summing their response to obtain the final waveforms at the victim node. In the switch factor based model, delay noise is a function of the timing windows at the victim and aggressor nodes themselves, creating a cyclical dependency. In the superposition based model, delay noise is a function of the timing windows at the driver *inputs* of the victim and aggressor nets, thereby removing this cyclical dependency and allowing for a more efficient solution to the problem.
- Instead of traversing the timing graph in topological order based on the structure of the circuit, we use a time sort based algorithm where, early and late window events are scheduled separately and processed in non-decreasing order of their event times. Each time an event is processed, new events are scheduled for the fanout nodes and coupled nets of these fanout nodes.

In general, time progresses in monotone increasing fashion in the proposed algorithm. Each event is therefore processed exactly once and is final when processed. However, in certain coupling situations, it is possible that new events require time to be rolled back requiring some events to be reprocessed. However, we will show that time roll-back can occur only once for an event at a particular node and requires only a fixed number of event reprocessing. Therefore, the worst-case run time is linear with the circuit size. Also, since the proposed algorithm is non-iterative and does not require initial timing window assumptions, it eliminates the problem with multiple solutions present in the current approach and removes the ambiguity of delay noise computation. To our knowledge, this is the first solution that is linear in run time and that eliminates the multiple solution problem. We implemented the proposed algorithm and show results on large circuit blocks. We show that, in practice, time roll-back is very rare and that the time sort algorithm achieves speedups of up to 5X over the iterative approach.

The remainder of this paper is organized as follows. Section 2 discusses the superposition approach for delay noise computation and the delay model assumptions and properties. Section 3 presents the time sort algorithm and shows its linear run time complexity. Section 4 contains the experimental results and Section 5 our conclusions.

2 DELAY MODEL AND PROPERTIES

Given a victim net and a set of capacitively coupled aggressor nets, referred to as a *noise cluster*, we construct linear Thevenin models for the victim and aggressor driver gates consisting of a Thevenin resistance in series with a linear voltage ramp [15][16]. Using superposition, each of the voltage sources is simulated in turn, while the other voltage sources are shorted. The voltage waveforms observed at the receiver gate input from all simulations are then added together using superposition to obtain the combined waveform. Figure 2(a)

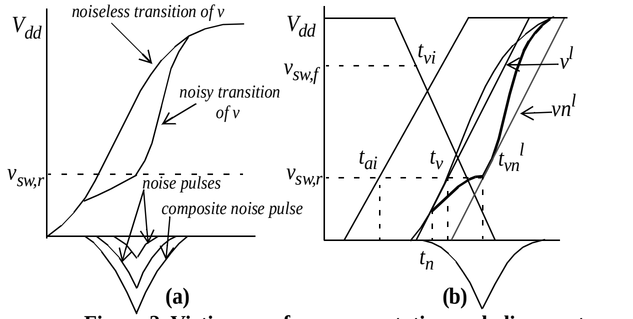

Figure 2. Victim waveform computation and alignment.

shows the *noiseless transition*, when only the victim driver voltage source is simulated, the noise pulses obtained by simulating each of the aggressor drivers, the *composite noise pulse* obtained by adding the individual noise pulses, and the *noisy transition*, obtained by adding the composite pulse to the noiseless transition. Linear driver models have the advantage that a reduced-order model of the linear network can be created only once with methods such as PRIMA [17], after which it can be reused in all different driver simulations. Also, the use of linear superposition allows the noise waveform induced by each aggressor to be shifted to search for the worst-case alignment with respect to the noiseless transition without requiring re-simulation of the network.

We first consider a noiseless transition at the victim driver node and define an early event $v^e = (t_v^e, s_v^e)$ and late event $v^l = (t_v^l, s_v^l)$, where t_v^e and t_v^l are early and late arrival times, and s_v^e and s_v^l are the early and late transition times. An arrival time is the point in time where the noiseless transition crosses the so-called *switching voltage* v_{sw} and the transition time is the time interval between the 20% and 80% V_{dd} crossing times of the transition. In order to ensure that all gate delays are positive, we use separate switching voltages for rising and falling transitions. We set the switching voltage for a rising transition $v_{sw,r} = V_{tn}$ and the switching voltage for a falling transition $v_{sw,f} = V_{dd} - V_{tp}$, where V_{tn} and V_{tp} are the NMOS and PMOS threshold voltages. Referencing the switching voltages to the threshold voltage ensures that the output transition of a gate starts after the input transition reaches v_{sw} and hence all gate delays are positive. Given the early event $v^e = (t_v^e, s_v^e)$ and late event $v^l = (t_v^l, s_v^l)$ at a node, the timing window at that node is given by the time interval $[t_v^e, t_v^l]$.

When one or more noise pulses are present on a net, we need to compute so-called noisy early and late events vn^e and vn^l, which are a function of the alignment of the noise pulses relative to the noiseless victim transition. In [13], it was shown that under reasonable noise pulse assumptions, the interconnect delay is maximized by aligning all aggressor noise pulses such that their peaks coincide. The peak of this composite noise pulse is then placed at the point where, for a late event computation, the noiseless victim transition reaches $v_{sw} + v_n$, where v_n is the height of the composite noise pulse. Similarly, for early events, the peak of the composite noise pulse is aligned at the point where the noiseless victim transition reaches $v_{sw} - v_n$. The optimal alignment for a late event is shown in Figure 2(a). If the alignment of the noise pulses is constrained by timing windows at the aggressor inputs, an optimal alignment may not be possible, thereby reducing the impact of noise on the arrival time. Also, if the noise pulse height exceeds V_{dd} - v_{sw} for a late event, the alignment is no longer well defined since $v_{sw} + v_n > V_{dd}$. In this case, we align the peak of the composite noise pulse at the end of the noiseless victim transition when it reaches V_{dd}, which is the alignment for a noise pulse with height $v_n = V_{dd} - v_{sw}$. Similarly, for an early event, we align a noise pulse that exceeds v_{sw} at the start of the noiseless victim transition.

Given a late event $v^l = (t_v^l, s_v^l)$ for a noiseless victim transition at node n, and the alignment of one or more noise pulses, we compute the noisy victim event $vn^l = (t_{vn}^l, s_{vn}^l)$ by setting t_{vn}^l equal to the time point where the noisy transition crosses v_{sw} and by setting $s_{vn}^l = s_v^l$ As illustrated in Figure 2(b), we have effectively shifted the ramp approximation of the noiseless victim transition to the point where the noisy victim transition crosses the switching voltage, while keeping its transition time constant. A noisy early event is computed similarly. Although a number of other models for abstracting the arrival time and transition time from the noisy transition are possible, the described model is a common model that is conservative for typical noise pulse shapes and has useful properties for timing window computation.

In general, the superposition approach for computing delay noise is more accurate than the switch-factor based approach, since it correctly models victim and aggressor transition times, their interconnect resistances, and the possibility of suboptimal noise pulse alignments. The noisy victim transition, and hence the impact of noise on delay, is a function of the alignment of the noise pulses relative to the noiseless victim transition. It is important to note that this alignment is constrained by the timing windows at the *input* of the aggressor and victim drivers. Unlike the switch factor noise model, delay noise is not a function of the noisy timing window $[t_{vn}^e, t_{vn}^l]$ at the aggressor and victim nodes themselves, thereby breaking the cyclical dependency between delay noise computation at a victim node on the timing window at that same node. Instead, the early and late events at a victim node are completely determined by the model of the interconnect and drivers of the noise clusters, and the timing windows at the inputs of the aggressor and victim drivers.

We present the following properties, based on the proposed noise alignment and noisy event computation:

Property 1. The arrival time t_{vi} at the victim driver input node is earlier than the noiseless arrival time t_v at a victim driver output node: $t_{vi} < t_v$

Property 2. The arrival time t_{ai} at an aggressor driver input is earlier than the time t_n, which is the start of the noise pulse induced by the aggressor: $t_{ai} < t_n$

Both Property 1 and Property 2 follow directly from the use of separate switching voltages $v_{sw,r}$ and $v_{sw,f}$ for rising and falling transitions and from setting these voltages such that the output transition of a gate starts after the input transition reaches the switching voltage. Figure 2(b) illustrates Property 1 and Property 2 for a late event. Both properties hold for early and late events.

Property 3. Given a victim transition with an optimally aligned noise pulse, the noisy arrival time t_{vn} at the victim node is later than the aggressor driver input arrival time t_{ai} that induced the noise pulse: $t_{ai} < t_{vn}$

This property follows from Property 2 which states that a noise pulse starts after the aggressor input arrival time and from the fact that the noisy victim arrival time is equal to the point in time where the noisy transition reaches the switching voltage, as shown in Figure 2(b). Again, Property 3 holds for both early and late events.

Next, we examine which of the victim input transition times that fall in its event window must be considered to obtain the worst-case noisy arrival time at the victim node. In Figure 3(a), we have shown a

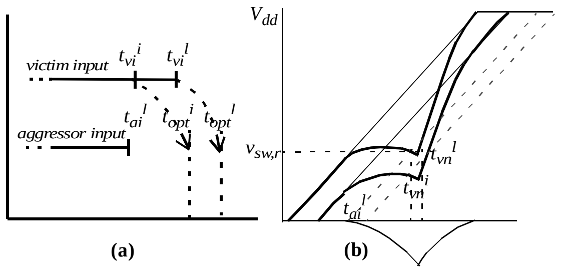

Figure 3. Delay noise for non-extreme points in the victim window.

victim and aggressor input time window. The time t_{opt}^l is the transition time at the aggressor input that results in optimal noise pulse alignment for the latest victim input transition t_{vi}^l. Since t_{opt}^l falls beyond the aggressor input window, an optimal noise alignment is not possible for the victim input transition at t_{vi}^l and the impact of noise on the delay at the victim node is reduced. We also consider an earlier victim input transition time t_{vi}^i, which has a more optimal noise alignment and therefore has a larger impact of the noise on its delay. This raises the question if, due to its larger delay noise, the earlier victim input transition at t_{vi}^i will result in a later *noisy* arrival time at the victim node than the victim input transition at t_{vi}^l. In Figure 3(b), we show the noiseless and noisy victim node transitions corresponding to the victim input transition time t_{vi}^l and t_{vi}^i. Note that the position of the noise pulse is fixed, since it is constrained by t_{ai}^l. From Figure 3(b) it is clear that t_{vi}^l will have a later noisy arrival time at the victim node than t_{vi}^i, as expressed in the following property:

Property 4. Consider two victim input events vi_i and vi_j, one or more noise pulses constrained by timing windows and the resulting noisy late events vn_i and vn_j at the victim node. If $t_{vi,i} > t_{vi,j}$ then $t_{vn,i} > t_{vn,j}$.

A similar property holds for early victim events. Based on this property, only the start and end of timing window need to be propagated, thereby substantially reducing the complexity of timing window computation.

3 TIME SORT ALGORITHM

In the proposed algorithm, events are processed ordered by non-decreasing arrival times. We refer to the time t for which events are being processed at a particular point in the algorithm as the *current time*. All events with arrival time t are retrieved and processed resulting in scheduling of new events. We will only schedule and process *noisy events*, i.e. events for which the impact of noise from aggressor nets has been considered. We will also compute noiseless events which are not scheduled and are only used to compute noisy events. Based on Property 4, we restrict our computation to the early and late events of a victim node, which correspond to the start and end points of its timing window.

Initially, we restrict our discussion to nets with a single aggressor net and then show how to extend the analysis to nets with multiple aggressors in Section 3.2. Figure 1(a) shows the victim net l with aggressor net m. The victim net has a victim driver gate with one or more input nets n and the aggressor net has an aggressor driver with one or more inputs k. However, for clarity, we will restrict our explanation to aggressor drivers with a single input since the extension to aggressor drivers with multiple inputs is straightforward. An event at the input of the victim driver is referred to as a victim input event vi and the resulting noiseless event at the victim driver node as the victim node event v. Similarly, an event at the aggressor driver input is

```
schedule early and late events for primary inputs
for (t = 0; scheduled events left; t++) {
    while (events left at time t) {
        select event e at node n from queue at time t;
        for (all fanout nodes l of n) {
            if (e is an early event) victim_early_event(e);
            else if (e is a late event &&
                    for all fanin nodes n of l a late event is defined)
                victim_late_event(e);
            for (all nets m coupled to l)
                if (e is early event) aggressor_early_event(e);
        }
        select check point event cp at time t;
        if (cp is an early cp event) cp_early_event(cp);
        else if (cp is a late cp event) cp_late_event(cp);
    }
}
```

Figure 4. Time sort algorithm for computing timing windows.

referred to as an aggressor input event *ai*.

The basic operation of the algorithm is to process early and late events at victim driver input node *n* and aggressor driver input node *k* and compute a new noisy event at the victim driver *output l*. A key property of the algorithm is that events of coupled nets *l* and *m* do not interact directly and are fully determined by the events of their driver inputs. An event at node *n* is first considered as a victim input event, resulting in a new event at its driver output *l*, and is then considered as an aggressor input event resulting in a possible change of the arrival time of the early and late events at net *m*.

The overall algorithm is shown in Figure 4 and consists of 5 event processing steps. We first illustrate the general approach of the algorithm and then discuss each procession step in detail in Section 3.1. First, all early and late events are processed as victim input events in functions *victim_early_event*() and *victim_late_event*(). For a victim input event *vi* at node *n*, the noiseless event *v* at the victim node *l* is first computed. If the early event *ai* at aggressor input *k* is defined (i.e. it has an early event time $t_{ai}^{e} < t = t_{vi}$), we compute the optimal alignment time t_{ai}^{opt} at input *k* for this aggressor *m*, as illustrated in Figure 5 for early and late events. Note that this is the arrival time for

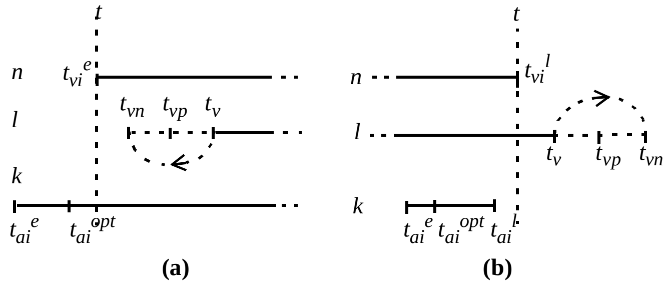

Figure 5. Early and late victim input event scheduling.

the *input* of the aggressor driver *k* such that the noise from the aggressor driver *output m* is aligned optimally. If the optimal alignment time $t_{ai}^{opt} \leq t$, we superimpose the noise from aggressor *m* on the noiseless event *v* at node *l* and a new noisy event *vn* at *l* is computed and scheduled. During the computation, the noise alignment is constrained by the timing window at node *k*. On the other hand, if the optimal aggressor alignment time $t_{ai}^{opt} > t$, there exists ambiguity as to whether the aggressor can transition at the optimal time, since the time window at *k* could end before t_{ai}^{opt}. In this case, we schedule a so-called *check point event cp* with check point time $t_{cp} > t$, when the ambiguity is resolved. The early and late check point events are scheduled much like regular events and are processed by functions *cp_early_event*() and *cp_late_event*().

We consider the impact of noise from aggressor *m* on node *l* only if the aggressor input *k* has a defined early event at time *t*. If the early event at aggressor input *k* is later than the victim input event *vi* at node *n* (i.e. $t_{ai}^{e} > t_{vi}$), we initially schedule event *vn* without considering the noise from aggressor *m* and then later update event *vn*, if necessary, when processing the early event ai^{e} at aggressor input node *k* in the function *aggressor_early_event*().

During the algorithm, early and late events at a node may update multiple times, due to processing of new aggressor early events when $t_{ai}^{e} > t_{vi}$. However, we will show that the worst-case number of event updates is limited by the number of aggressor nets that couple to a victim net and is independent of the circuit size. Also, the sorting and scheduling of new events can be performed in constant time by discretizing time, $t = i \cdot \Delta t$, where Δt is the time step and integer *i* is used as an array index, thereby avoiding the $O(n\log(n))$ complexity of sorting. If Δt is chosen sufficiently small, such as 1ps or 0.1ps, the error incurred due to discretization is negligible.

The objective of the algorithm is to process only one early and late event at a node. Hence, we introduce the following conditions:

- **Condition 1**: When scheduling a new event, the arrival time t_{vn} of this event *vn* falls after the current time *t*: $t_{vn} > t$.
- **Condition 2**: When scheduling a new event *vn*, an existing event of that type has not already been processed at that node.

Condition 1 ensures that newly scheduled events fall in the future and do not require a roll-back of the current time in the algorithm. Condition 2 ensures that a new event is not scheduled after a previous event of the same type has already been processed for a node. In other words, when an event is processed for a node, it is the final event of that type for that node. It is clear that satisfying Condition 1 and Condition 2 is a sufficient criteria for guaranteeing that only one event of each type is processed at each node.

In Section 3.1, we will show that in all but two cases, the two conditions are satisfied in the proposed algorithm. However, under certain timing window alignments, it is possible that in function *victim_early_event*() Condition 1 is not met, and in function *aggressor_early_event*() Condition 2 is not met. In these cases, the current time *t* of the algorithm is rolled back and the event in question, as well as other processed events that depended on it, must be reprocessed. We will show, however, that for a particular circuit node, time roll-back can occur only once for each of the two types of events. Also, in each case of time roll-back, the number of reprocessed events is fixed based on the topology of the circuit and is not a function of the circuit size. Hence, the worst-case run time of the algorithm is linear with the circuit size as opposed to $O(n^2)$ for the worst case run time using the existing iterative approach. Also, we found that time roll-back and event reprocessing are extremely rare in actual circuits while the number of iterations required in the existing approach is often quite significant. Finally, since the algorithm is non-iterative and does not operate using an initial overlapping or non-overlapping timing window assumption, it does not exhibit the multiple solution problem of the iterative approach. Our results show that, depending on the initial assumption, the timing results obtained with the iterative approach can either over- or under-estimate the results obtained with the proposed approach. The proposed approach therefore also improves the accuracy of the analysis.

3.1 Event Processing and Scheduling

In this Section, we present the 5 event processing steps and also discuss their adherence to Condition 1 and Condition 2.

3.1.1 Victim Early Event

In this scheduling step, we consider the response of the victim node *l* due to an early event at one of its driver inputs *n*, as shown in Figure 6. Node *l* may already have an early event scheduled, due to one of the other driver inputs of *l*. We first compute the noiseless vic-

victim_early_event(vi^e)

```
1  compute noiseless victim event v with time $t_v$ at fanout node l
2  find current early event vp at l with event time $t_{vp}$(set $t_{vp}$ to +inf if unde
3  compute noise from aggressor m with aggressor input k
4  find early and late event time $t_{ai}^e$ and $t_{ai}^l$ of k (set $t_{ai}^l$ to t if undef)
5  if (early event time $t_{ai}^e$ is defined) {
6      compute optimal switching time $t_{ai}^{opt}$ at k resulting in noisy event $t_v$
7      if ($t_{ai}^{opt} \leq t$) {
8          if ($t_{ai}^e \leq t_{ai}^{opt} \leq t_{ai}^l$)
9              compute event vn at l, with optimal noise alignment
10         else compute event vn at l, with nearest alignment in [$t_{ai}^e$, $t_{ai}^l$]
11         if ($t_{vn} < t_{vp}$) remove vp and schedule vn
12     } else {
13         if ($t_{ai}^l$ is defined) {
14             compute noisy event vn at l, with noise aligned at $t_{ai}^l$
15             if ($t_{vn} < t_{vp}$) remove vp and schedule vn
16         } else
17             if ($t_{vn} < t_{vp}$) remove vp and schedule check point event cp at t
18 else {
19     schedule event v
20 }
```

Figure 6. Victim early event processing.

tim event *v* at *l* due to event vi^e at *n*, as illustrated in Figure 5(a). We then consider the aggressor *m* and its driver input *k*. If the early event time t_{ai}^e is defined (i.e. $t_{ai}^e < t$), we determine the noise pulse injected from node *m* on node *l* and find the switching time t_{ai}^{opt} at node *k* that will result in the optimal alignment of this noise pulse with respect to the noiseless transition on node *l* (line 3 and 6). If $t_{ai}^{opt} < t$, as shown in Figure 5(a), we compute the earliest noisy event *vn*, constrained by the switching window [t_{ai}^e, t_{ai}^l] at node *k* (lines 8-10) and compare its arrival time t_{vn} with the current early event time t_{vp} (line 11). If $t_{vn} < t_{vp}$ we remove the existing event *vp* and schedule the new event *vn*. If $t_{ai}^{opt} > t$, we check if the event window at aggressor input *k* has ended. If it has, we align the noise based on the latest point in the aggressor input window t_{ai}^l. Otherwise, there is ambiguity whether the optimal alignment can occur (since it falls in the future) and we schedule a check point event for node *l* at time t_{vn} (line 17), which is processed in function *cp_early_event*(). Note that if the early event on *k* is not defined, we ignore this aggressor driver input and compute its effect on node *l* in function *aggressor_early_event*(). Note also that aggressor *m* may have multiple driver inputs, in which case, the procedure is repeated for each aggressor driver input.

We only schedule event *vn* if a previously scheduled event *vp* has an event time $t_{vp} > t_{vn}$ and has not been processed, which satisfies Condition 2. Also, if the noise height is less than switching voltage v_{sw}, it is clear that noisy victim node transition time t_{vn} must fall after the victim input early event time t_{vi} and hence, Condition 1 is met. However, if the noise pulse is larger than the switching voltage v_{sw}, it is possible that $t_{vn} < t$, if $t_{ai}^e < t_{vi}^e$ and Condition 1 is not met. This means that the victim node early event time t_{vn} occurs before the victim input early event time $t_{vi}^e = t$. In this case, events with arrival times in the window [t_{vn}, t_{vi}^e], that were already processed and that depend directly or indirectly on the transition at node *l* must be reprocessed. We examine all events at fanout nodes of net *l* or coupled to the fanout nodes of net *l* and reschedule any that fall in the time window [t_{vn}, t_{vi}^e]. We apply this procedure recursively as shown in Figure 7. We now show the following important properties of this reprocessing step:

reschedule_events(l, t_{vn}, t_{vi}^e)

```
1  for (all nets p, fanout of net l or coupled to fanout of net l) {
2      if (p was not already visited for rescheduling) {
3          find event e with event time $t_e$ at node p, initiated by node l
4          if ($t_{vn} < t_e < t_{vi}^e$) reschedule event e at node p
5          reschedule_events(p, $t_{vn}$, $t_{vi}^e$)
6      }
7  }
```

Figure 7. Event rescheduling procedure.

1. An early event *vn* at a victim node will be rescheduled only once. Since the victim input early event vi^e with arrival time t_{vi}^e lies in the rescheduling window [t_{vn}, t_{vi}^e], event vi^e will be rescheduled if node *n* depends on node *l*. When vi^e is processed the second time, the question arises whether the resulting victim node early event *vn*,2, will have an event time $t_{vn,2} < t_{vn}$ such that it must be rescheduled again, leading to the possibility of multiple iterations. Note that if $t_{vn,2} > t_{vn}$, *vn,2* will not be scheduled, due to line 11 in Figure 6, terminating the iteration. From delay model Property 3, if follows that in order for $t_{vn,2} < t_{vn}$, event *vn* must cause one or more aggressors inputs of *l* to have a new early event time $t_{ai,2}^e < t_{vn,2}$. However, delay model Property 1 and Property 3 also indicates that any impact of *vn* on an aggressor is such that $t_{ai,2}^e > t_{vn}$ and hence it follows that $t_{vn,2} > t_{vn}$, meaning that only one rescheduling iteration can occur. Intuitive, we can see that for *vn* to be further decreased in further iterations, *vn* would need to cause one of the aggressors of *l* to become earlier than itself which contradicts delay model Property 3.
2. Since our method of alignment dictates that the noise peak must occur at or after the start of the victim noiseless transition, the size of the rescheduling window [t_{vn}, t_{vi}^e] is bounded by the transition time of the leading edge of the noise. This is a quantity that is a function of technology and routing and is independent of circuit size. Therefore, from the point of view of run time complexity, we can consider the window size to be a constant. The number of events that need to be scheduled for a fixed window size depends on the number of fanouts of node *l* and the number of nets coupled to these fanouts which is independent of circuit size, ensuring the linear run time of the algorithm. Furthermore, the rescheduling window was found to be very small in practice and requiring no more than two events to be rescheduled.

3.1.2 Victim Early Check Point Event

In this scheduling step, we process early check points that were previously scheduled by early victim input events as shown in Figure 8. The current time *t* is equal to the noisy victim event time under optimal alignment time of noise from node *k* computed in *victim_early_event*(). We first check if the latest event time t_{ai}^l on

cp_early_event(*cp)*

```
1  find current early event vp at l with event time $t_{vp}$ (set $t_{vp}$ to +inf if undef)
2  compute optimal switching time $t_{ai}^{opt}$ at aggressor input k
3  find late event time $t_{ai}^l$ of aggressor input k
4  $t_{align} = min(t_{ai}^l, t_{ai}^{opt})$
5  compute noisy early event vn at l, with aggressor input k aligned at $t_{align}$
6  if ($t_{vn} < t_{vp}$) remove vp and schedule vn
```

Figure 8. Early check point event procedure.

aggressor input m has occurred and set the alignment time t_{align} to the minimum of t_{ai}^{opt} (for optimal alignment) and t_{ai}^{l} (for latest possible alignment). We then compute the new early victim node event vn and schedule it if there is no previous event vp with an earlier arrival time.

To satisfy Condition 1, we first consider the case where $t_{align} = t_{ai}^{opt}$. In this case, the noise is optimally aligned and $t_{vn} = t$. In the case where $t_{align} = t_{ai}^{l}$, a suboptimal alignment of the noise occurs, since $t_{ai}^{l} < t_{ai}^{opt}$ and the impact of the noise on the circuit delay will be reduced. Therefore, the victim event time t_{vn} in this case will be later than the event time in the previous case, and hence $t_{vn} > t$, which satisfies Condition 1. To satisfy Condition 2, we again note that event vn is only scheduled if $t_{vn} < t_{vp}$, which guarantees that vp was not already processed.

3.1.3 *Victim Late Event*

In this scheduling step, we consider the response of the victim node l due to a late event at one of its driver inputs n. The computation is analogous to *victim_early_event*(), and is shown in Figure 9.

```
victim_late_event($vi^l$)

1  compute noiseless victim event $v$ with event time $t_v$ at node $l$
2  compute noise from aggressor $m$ with aggressor input $k$
3  find early and late event time $t_{ai}^e$ and $t_{ai}^l$ of $k$ (set $t_{ai}^l$ to $t$ if undefined)
4  if (early event time $t_{ai}^e$ is defined) {
5      compute switching time $t_{ai}^{opt}$ at $k$ for optimal noise alignment
6      if ($t_{ai}^{opt} \leq t$) {
7          compute vn at $l$, with noise at nearest alignment in $[t_{ai}^e, t_{ai}^l]$
8          schedule vn
9      } else {
10         if ($t_{ai}^l$ is defined) {
11             compute noisy event vn at $l$, with noise aligned at $t_{ai}^l$
12             schedule vn
13         } else {
14             compute late event $vn'$ at $l$, with noise aligned at $t$
15             schedule check point event cp at time $t'_{vn}$ for node $l$
16             }
17     }
18 }else{
19     schedule event v
20 }
```

Figure 9. Victim late event processing.

Again, we compute the optimal alignment time for the noiseless victim event v, due to an aggressor input k with a defined early aggressor input event. If the optimal alignment $t_{ai}^{opt} \leq t$, as is illustrated in Figure 5(b), we compute the noise on node l at the optimal alignment, constrained by the switching window of $[t_{ai}^{e}, t_{ai}^{l}]$ of k. If $t_{ai}^{opt} > t$ and the switching window of k has not yet ended at time t, there is again ambiguity about whether the optimal alignment can occur and we schedule a check point for a future point in time to determine if the optimal alignment is possible. However, contrary to the early event time computation, the check point is scheduled at the noisy victim late event time t'_{vn}, computed with the aggressor aligned at time t at node k (line 14,15). This computed t'_{vn} is a lower bound on the final t_{vn} when the actual alignment occurs.

It is important to note that a late event at the victim input n is not processed, unless the late event at all of its driver fanin nodes has been processed, as shown in line 8 in Figure 4. This is necessary to ensure that Condition 2 is met. Also, since this situation means that at least one driver input of n has not completed its event window at time t, it ensures that the latest event on l will lie after the current time t, based on delay model Property 1.

Based on delay model Property 1, the noiseless victim event time t_v at node l falls after t. Since the impact of noise on node l will only increase the late arrival time, the noisy late event time $t_{vn} > t$, satisfying Condition 1. To satisfy condition 2, we must show that the event time t_{vp} of all removed events vp is greater than t. Such previously scheduled events vp would be scheduled by another victim driver input n_x of node l (see Figure 1(a)), such that $t_{vp} < t_{vi}$. However, this means that event vp at node l would be processed before event vi at node n and therefore will be skipped in line 8 in Figure 4. This ensures that Condition 2 is satisfied.

3.1.4 *Victim Late Check Point Event*

In this scheduling step, we consider a check point that was previously scheduled by the algorithm. We again check the optimal alignment time and if $t_{ai}^{opt} < t$, we compute the final noisy late event at node l and schedule it. If, however, the optimal alignment time $t_{ai}^{opt} > t$, there is ambiguity on whether the optimal alignment time can occur, and we schedule another check point. It is therefore possible that multiple check points are scheduled for a particular victim late event. However, it is easy to show that the transition time of the leading edge of the noise pulse is a lower bound on the amount of time by which the check point progresses forward in each iteration. Also, the first check point is initiated when $t = t_{vi}^{l}$ at node n and the latest check point must occur before the end of the victim transition at node l. Since the window of time that a late check point can span and the minimum distance between check points are comparable, typically very few, if any, iterations are necessary. Also, the number of required check points is a function of the gate delay and signal slopes and is independent of the circuit size and can be considered a constant for run time analysis. The event processing code for a late check point is identical to the code for victim late events, except that the *if* statement on line 4 of Figure 9 is not needed.

To show that the processing of a late check point event adheres to Condition 1, we need to show that $t_{vn} > t$. The current time $t = t'_{vn}$, where t'_{vn} is the noisy victim event time computed when the check point was scheduled in *victim_late_event*() or in a previous iteration of *cp_late_event*() (line 14 and 15, Figure 9). Since t'_{vn} was computed using a less optimal noise alignment than t_{vn}, if follows that $t_{vn} > t'_{vn}$, and hence, Condition 1 is satisfied. To satisfy Condition 2, we note that function *victim_late_event()* and *cp_late_event()* will schedule either a checkpoint or a victim late event and only one event is scheduled for a victim node at any point in time. Therefore, a victim late event cannot occur before a check point event, which satisfies Condition 2.

3.1.5 *Aggressor Early Event*

In this step we compute the impact of an aggressor early event on early and late victim events as shown in Figure 10. We only process an aggressor early event if the victim event has already been processed (condition $t_{vi}^{e} < t$ on line 4 and 13). If the victim input early event has not been processed, the impact of the aggressor is accounted for in the function *victim_early_event*() and *victim_late_event*(). When an aggressor early event is processed, the current time $t = t_{ai}^{e}$, as illustrated in Figure 10. We first compute the optimal alignment time t_{ai}^{opt} at the aggressor input node n. If $t_{ai}^{opt} < t$, we compute the new victim event vn and schedule it (lines 9,18 and 19). If the optimal alignment time falls after the current time, there is again ambiguity on whether the optimal alignment can occur and we again schedule a check point (lines 11,21 and 22), in similar fashion to function *victim_early_event*() and *victim_late_event*().

aggressor_early_event(ai^e)

```
1  for (all victim driver inputs k) {
2    find current early and late event vp and check point event cp at m
3    compute noise from aggressor l on victim m with aggressor input n
4    if (early victim input event $vi^e$ with event time $t_{vi}^e < t$) {
5      compute noiseless victim event v with event time $t_v$ at fanout node m
6      compute switching time $t_{ai}^{opt}$ at n with optimal noise alignment
7      if ($t_{ai}^{opt} \leq t$) {
8        compute early event vn at m, with aggressor input n aligned at t
9        if ($t_{vn} < t_{vp}$) remove vp and schedule vn
10     } else
11       if ($t_{ai}^{opt} < t_{vp}$) schedule check point event cp at time $t_{ai}^{opt}$
12   }
13   if (late victim input event $vi^l$ with event time $t_{vi}^l < t$) {
14     compute noiseless victim event v with event time $t_v$ at fanout node m
15     compute switching time $t_{ai}^{opt}$ at n with optimal noise alignment
16     compute late event vn at m, with aggressor input n aligned at t
17     if ($t_{ai}^{opt} \leq t$) {
18       if ($t_{vn} > t_{vp}$) remove vp and schedule vn
19       else if($t_{vn} > t_{cp}$) remove cp and schedule vn
20     }else {
21       if($t_{vn} > t_{vp}$) remove vp and schedule check point event cp at $t_{vn}$
22       if($t_{vn} > t_{cp}$) remove cp and schedule check point event cp at $t_{vn}$
23     }
24   }
25 }
```

Figure 10. Aggressor early event procedure

Condition 1 is again satisfied for both the noisy early and late event computation by delay model Property 3, since the aggressor input transition time $t_{ai}^e = t$ must fall before the victim noisy event time t_{vn}. For the computation of the noisy victim early event, Condition 2 is satisfied by the fact that we only schedule event *vn* if $t_{vn} < t_{vp}$. However, Condition 2 is not met for the computation of the victim late event in lines 13 - 24, meaning that a previously scheduled late event *vp* at the victim node *m* may have arrival time $t_{vp} < t$ and is already processed. We therefore need to reschedule the victim late event *vp* at node *m*, as well as any events dependent on it with event times in the window $[t_{vp}, t_{ai}^e]$. We again perform this task using the re-scheduling routine *reschedule_events*(), shown in Figure 7. Note that as events are reprocessed, erroneously scheduled events are automatically removed and replaced by correct events. We again show two important properties of the rescheduling process:

1. The victim late event at node *m* will be rescheduled only once. Since the instigating aggressor event ai^e falls in the window $[t_{vp}, t_{ai}^e]$, it is also recomputed, giving rise to a new event ai^{e}', if node *n* is in some way dependent on node *m*. When event ai^{e}' is processed, it will produce a new victim late event *vn'* at node *m*, raising the possibility of cyclical dependence and the need for multiple iterations. However, we show that $t_{vn'} < t_{vn}$, meaning *vn'* is not scheduled, terminating the loop after the first iteration.

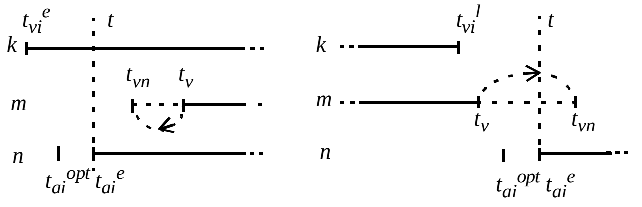

Figure 10. Early aggressor event processing for (a) early victim events and (b) late victim events.

First, we note that since ai^e is an early event a new event ai^{e}' can occur only if $t_{ai'}^e < t_{ai}^e$. Second, we note that since $t_{ai}^e > t_v$ it follows that the noise pulse induced by aggressor *l* starts after the victim noiseless event time $t_n > t_v$. From this it is clear that the noise pulse can affect the victim noiseless transition only if it has a slope steeper than the slope of the victim noiseless transition. Since the new event $ai^{e'}$ is earlier than ai^e, it follows that a noise alignment at time $t_{ai}^{e'}$ is suboptimal and produces a victim noisy event time t_{vn}' which is earlier than t_{vn}, meaning that it will not be scheduled.

2. The size of the reprocessing window $[t_{vp}, t_{ai}^e]$ is bounded by the size of the noiseless victim transition at node *m*. Similar to victim early events, we can therefore show that the number of dependent events that fall in the window $[t_{vp}, t_{ai}^e]$ is not a function of the circuit size and is small in practice, as demonstrated in our experimental results.

3.2 Multiple aggressors

In practice, each victim net may have multiple aggressors. We therefore present the following extension of the time sort algorithm for victim nodes with multiple aggressor nets. The main scheduling loop in Figure 4 remains unchanged. The event processing steps are changed in that the optimal alignment time computation t_{ai}^{opt} must consider multiple aggressors. We show the code for computing the optimal alignment time for multiple aggressors in Figure 11. Given a

```
1 compute noiseless victim event v with event time $t_v$ at node l
2 find early and late event time $t_{ai,i}^e$ and $t_{ai,i}^l$ of $k_i$ (set $t_{ai,i}^l$ to t if undef)
3 construct set $S_a$ of aggressors $m_i$ with defined early event time $t_{ai,i}^e$
4 computed noise height $n_h$ = sum of noise heights from aggressors in $S_a$
5 compute alignment of composite noise pulse with noise height $n_h$
6 compute optimal alignment $t_{ai,i}^{opt}$ for aggressor inputs $k_i$ of $m_i$ in $S_a$
7 compute maximum alignment $t_{ai}^{max,opt} = max(t_{ai,i}^{opt})$ for $m_i$ in $S_a$
```

Figure 11. Optimal alignment computation for multiple aggressors.

set of aggressor nets m_i each with input k_i, we first construct a set of aggressor nets S_a that have defined aggressor input early event times $t_{ai,i}^e$ at their aggressor input k_i (line 3). We then compute the noise pulse from each of the aggressors m_i in S_a as well as the sum of their noise pulse heights, n_h (line 4). We compute the optimal alignment of the composite pulse and, based on the delay of each of the aggressor drivers, the optimal alignment time at the aggressor drivers inputs $t_{ai,i}^{opt}$ (lines 5,6). From among these optimal aggressor input alignment times we select the latest, $t_{ai}^{max,opt}$. The processing of events is now identical to that in the single aggressor case, except that t_{ai}^{opt} is replaced with $t_{ai}^{max,opt}$.

Table 1. Benchmark Circuit Information

Circuit	#Nets	# Gates	Total # couplings	% Nets with coupling	Average # of couplings per net
c432	218	181	445	81.65	4
c499	582	540	1100	82.47	4
c880	431	370	245	32.01	3
c1355	626	584	1072	90.57	3
c1908	506	472	682	90.11	2
c2670	1118	884	1749	78.35	3
c3540	1069	1018	3529	94.76	6
c5315	1876	1697	3920	62.95	6
c6288	2450	2417	11210	93.06	10
c7552	2449	2241	10940	91.42	9
alu64	1935	1803	9085	93.17	10

We note that delay model Property 3 applies not only to a single aggressor but also to multiple aggressors. In other words, the noisy arrival time t_{vn} due to multiple optimally aligned noise pulses is later than *all* aggressor driver input arrival times $t_{ai,i}$ that induced the noise pulse. Hence, it follows that $t_{vn} > t_{ai}^{max,opt}$, from which it can be shown that the run time of the algorithm is also linear for nets with multiple aggressor couplings.

4 RESULTS

The proposed time-sort algorithm was implemented and tested for the ISCAS85 benchmark circuits and a 64-bit ALU, which were synthesized using Synopsys Design Compiler. The characteristics of these circuits are shown in Table 1. Coupling capacitance was generated randomly between nodes in the circuit. Table 1 shows the percentage of nets with one or more couplings and the average number of coupling capacitances per net. For the larger circuits, an average number of couplings per net of 10 was used, which corresponds to the number of expected couplings for nets in high performance designs [4]. Delay and slope computation was performed using pre-characterized library models for a TSMC $0.18\mu m$ library.

Table 2. Results for ISCAS85 combinational Circuits

Circuit	Iterative Approach		Proposed Time Sort Approach					speed up
	Run Time (sec)	# Iterat ions	Run Time (sec)	% Check Point Events	Avg # check points	# Roll Backs	# Events reprocessed	
c432	0.08	5	0.02	18.05	0.62	0	0	3.89
c499	0.33	5	0.12	19.54	0.57	0	0	2.71
c880	0.17	3	0.03	21.91	0.69	0	0	4.67
c1355	0.34	5	0.15	27.19	0.89	0	0	2.31
c1908	0.31	6	0.08	26.64	0.93	0	0	3.59
c2670	0.65	7	0.19	21.83	0.62	0	0	3.27
c3540	1.37	6	0.52	31.30	1.36	0	0	2.63
c5315	1.43	5	0.45	30.52	1.01	0	0	3.18
c6288	4.9	11	0.98	25.89	1.12	0	0	5.01
c7552	2.81	6	0.93	27.13	1.16	2	4	2.99
alu64	3.65	6	0.73	31.14	1.29	0	0	4.98

The traditional iterative approach was also implemented and Table 2 compares the results from the two methods. The reported run times are on a Pentium IV 1.8GHz PC running Linux. For the iterative method, the number of iterations needed for convergence ranged between 3 and 11. The improvement in run-time ranged from 2.31x to 5.01x and increased with circuit size and the number of couplings in a circuit. Roll-back occurred only for circuit c7552, causing recomputation of two events for both cases of roll-back. This demonstrates that time roll back is extremely rare and that in practice each net is processed only once. Table 2 also shows the average number of check points per net, which ranged between 0.57 and 1.36. Note that a check point does not represent multiple processing of an event, but only multiple updates to an event time before the event is processed.

Finally, we compared the timing analysis results for the two methods. The iterative method converged to different solutions depending on the initial overlap assumption in all of the 11 circuits, with 4 of them more significantly so. As expected, we found that the proposed time sort algorithm obtained a result that fell between these two solutions. The proposed approach therefore increases run time efficiency, as well as analysis accuracy.

5 CONCLUSIONS

We have presented a new timing window computation method for static timing analysis in the presence of cross-coupling noise. The proposed method is based on delay noise computation using linear superposition and early and late event processing ordered by arrival times. We have shown that the proposed algorithm has linear run time with circuit size as opposed to the $O(n^2)$ worst-case complexity of the traditional iterative approach. Since the algorithm is non-iterative and does not use an initial coupling assumption for timing window computation it eliminated the multiple solution problem present in the iterative approach. We demonstrated the proposed method on benchmark circuits with extensive capacitive coupling and show that it obtains significant speedup over an iterative method.

Acknowledgements

This research was supported by SRC contract 2001-HJ-959 and NSFgrant CCR-0205227.

References

[1] Shepard K.L., Narayanan V., “Noise in Deep Submicron Digital Design”, Proc. of ICCAD, pp. 524 -531, 1996

[2] D. Sylvester and K. Keutzer, “Getting to the bottom of deep submicron,” Proc. of ICCAD, pp 203-211, Nov. 1998.

[3] K. L. Shepard, etal., “Global Harmony: Coupled noise analysis for full-chip RC interconnect networks,” ICCAD, 1997.

[4] R. Levy, D. Blaauw, G. Braca, A. Dasgupta, A. Grinshpon, C. Oh, B. Orshav, S. Sirichotiyakul, V. Zolotov, “ClariNet: a noise analysis tool for deep submicron design”, Proc. of DAC, pp. 233-238, 2000.

[5] P.Chen, K.Keutzer. “Towards True Crosstalk Noise Analysis”, Proc. of ICCAD, pp.132-137, 1999.

[6] A. Glebov, S. Gavrilov, D. Blaauw, S. Sirichotiyakul, V. Zolotov, Chanhee Oh, “False-Noise Analysis Using Logic Implications”, Proc. of ICCAD 2001.

[7] S. Sapatnekar, “Capturing the effect of crosstalk on delay”, Proc. VLSI Design India, pp. 364-369, 2000.

[8] A. B. Kahng, S. Muddu, E. Sarto, “On Switching Factor Based Analysis of Coupled RC interconnects,” DAC, pp 79-84, 2000.

[9] P. Chen, D. A. Kirkpatrick, K. Keutzer, “Miller Factor for Gate-Level Coupling delay calculation”, Proc. ICCAD 2000.

[10] S. Sapatnekar, “A timing model incorporating the effect of crosstalk on delay and its application to optimal channel routing,” IEEE Trans. on CAD, Vol 19, No. 5, pp. 550 - 559, 2000.

[11] P. Chen, D. A. Kirkpatrick, K. Keutzer, “Switching Window Computation for Static Timing Analysis in the Presence of Crosstalk Noise,” ICCAD, pp. 331-337, 2000.

[12] H. Zhou, N. Shenoy, W. Nicholls, “Timing Analysis with crosstalk as fixed points on Complete Lattice”, DAC 2001.

[13] F. Dartu, L. T. Pileggi, “Calculating Worst-Case Gate Delays Due to Dominant Capacitance Coupling,” DAC, 1997

[14] S. Vrudhula, S. Sirichotiyakul, D. Blaauw, “Estimation of the likelihood of capacitive coupling noise”, Proc. of DAC 2002.

[15] F. Dartu, N. Menezes, and L. T. Pileggi, “Performance Computation for Precharacterized CMOS Gates with RC Loads,” IEEE Trans. on CAD, Vol.15, No. 5, pp. 544-553, May 1996

[16] P.D. Gross, etal, “Determination of worst-case aggressor alignment for delay calculation,” ICCAD, pp. 212-219, 1998.

[17] Odabasioglu, A.; et al., “PRIMA: passive reduced-order interconnect macromodeling algorithm,”Corrolary ICCAD, 1997.