

# Accurate Timing Analysis using SAT and Pattern-Dependent Delay Models

D. Tadesse\*, D. Sheffield\*, E. Lenge<sup>o</sup>, R. I. Bahar\*, J. Grodstein<sup>†</sup>

\*Brown University, Division of Engineering, Providence, RI 02912

<sup>†</sup>Intel Corporation, Hudson, MA 01749

<sup>o</sup>Rensselaer Polytechnic Institute, Troy, NY 12180

## Abstract

*Accurate delay modeling beyond static models is critical to garnering better correlation with post-silicon analysis. Furthermore, post-silicon timing validation requires a pattern-dependent timing model to generate patterns. To address these issues, we propose a timing analysis tool that integrates a data-dependent delay model into its analysis. Our approach solves for the delay by using the concept of circuit unrolling and formulation of timing questions as decision problems for input into a SAT solver. The effectiveness and validity of the proposed methodology is illustrated through experiments on benchmark circuits.*

## 1. Introduction

As the complexity of a chip increases due to advances in IC technology, so do the pre- and post-fabrication test and debug challenges. Although pre-silicon test tools attempt to ensure correct circuit functionality, verification through exhaustive simulation is unattainable. This puts the burden on the post-silicon process, which has been recognized as a growing and intractable problem.

Static timing analysis (STA) has been used for many years as a pre-silicon tool to help design high-performance integrated circuits. However, STA tools (being static) generally do not deal well with input patterns. The post-silicon world, on the other hand, requires specific test patterns. Specifically, post-silicon *timing validation* requires a pattern-dependent timing model to generate patterns for a number of reasons. Among them are:

- **Speed Testing** to determine if manufactured parts meet a speed specification based on their process corner.
- **Delay-Fault Testing** to determine if parts have defects that affect their behavior at speed.

These patterns also may be used for hypothesis generation for:

- **Fault Isolation** for determining which gate's delay is different from pre-silicon estimates due to a delay fault.
- **Speed Debug** for determining which of multiple sensitized paths converging on a failing latch is indeed the speed limiter.

In each case, the reliance of post-silicon debug on patterns makes it difficult to take advantage of the large pre-silicon static timing effort, resulting in redundant effort and schedule impact. In short, *static timing is not post-silicon friendly*.

In this paper, we will propose a pattern-dependent delay model. Our approach solves for the delay by using the concept of circuit unrolling and formulation of timing questions as decision problems for input into a SAT solver. The goal is to answer timing questions in the context of specific input patterns. For the purposes of this paper, we will focus on one application: determining

the longest critical delay-path through a cone of logic. We will not only generate a critical path, but the input vector that stimulates it. All delays will take into account pattern-dependent effects such as data-dependent gate delays and multiple-inputs switching.

Such a tool could easily generate test patterns for the paths it finds. Furthermore, in doing so, it could help shed light on the relative importance of the different sources of miscorrelation between STA and measured silicon speeds. Though there are many potential sources (see, for example, [9]), there is little real silicon data that identifies one as the most important.

Post-silicon timing is starting to come into its own as a research topic [10, 13]. In addition, there has been much prior work in related areas. For example, STA with dynamic sensitization was well studied in the early 1990s [8, 17, 19, 20]. Unfortunately, this work was too CPU intensive to be commercially practical. More recent work has explored modeling the effects of crosstalk on delay [7, 11], input slope on delay [5], crosstalk on noise [6], and on the effect of multiple-inputs switching on delay [1, 16, 22]. However, these approaches use either static techniques or min-max switching windows. Min-max approaches are limited, as the existence of a short path from one input vector and a long path from another vector does not guarantee existence of intermediate paths.

The idea of formulating circuit properties as decision problems is not new. The technique presented in [2] used a SAT-based search algorithm for leakage power reduction, while the work of [21] used a SAT solver to compute combinational circuit delay. However, their approach was limited to a floating mode delay, which assumes that the primary inputs and all other nodes start at unknown values and make one transition to a defined value. Our approach is based on a transition-mode, which calculates the delay for a pair of actual vectors. This approach is much better defined than for a floating mode approach and it also fits naturally for post-silicon work.

The unrolling technique was first described in [15], but our work is most similar to [6]. Like them, we convert a collection of timed combinational gates to a larger collection of zero-delay gates by unrolling it (similar to Bounded Model Checking [3, 4]). That is, we quantize continuous time into *a priori* fixed time points, make a separate copy of the network at each time point, and stitch the copies together using our delay model. Our immediate contribution is to extend this formulation past the use of the simple constant-delay gates used in [6] by including data-dependent delays and multiple inputs switching. We will show that this imposes significant challenges, and requires us to move from a simple transport delay model to a more complex inertial-delay model. Furthermore, while [6] attacked the problem of crosstalk analysis, we will attack general pattern-dependent timing problems. Our long-term contribution (beyond this paper) will be the use of this delay model for many of the post-silicon timing areas previously mentioned.

## 2. Circuit Unrolling

Our approach is based on unrolling a circuit using data dependent delay information for each gate. We first explain the unrolling procedure with a constant delay, for ease of illustration, followed by enhancements needed to handle data dependence information.

### 2.1 Unrolling Basics

The constant-delay model is a simple representation of the delay behavior of a gate. For any given gate, its logic function is modeled as a perfect, infinite-bandwidth zero-delay boolean gate. Its temporal function is modeled as a perfect delay line after the boolean gate. Thus, if its delay is  $d$ , then its output at any time  $t$  is simply a Boolean function of its inputs at time  $t - d$ . This is often called a *transport-delay* model.

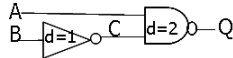


Figure 1. Simple Circuit (Constant delay model)

Consider the simple circuit shown in Figure 1, with delay values as shown on each gate. We will basically “unroll” the network, making a separate copy of the network at every time point. This approach is similar to the popular unrolling techniques used for Bounded Model Checking [3,4]. The small network of timed gates will thus become a larger network of zero-delay gates.

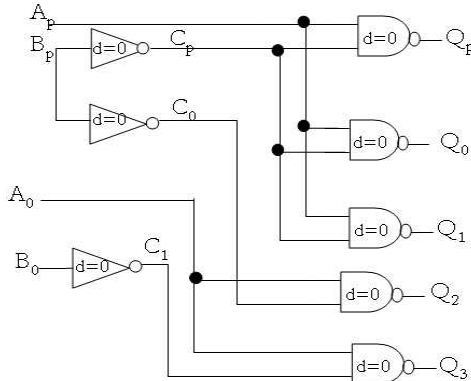


Figure 2. Unrolled network (constant delay model)

Figure 2 is a straightforward unrolling of Figure 1, beginning with output node  $Q$ , for  $t = 0, 1, 2$ , and 3. We are considering the primary inputs ( $A$  and  $B$ ) to be outputs of flip-flops, whose values can change once at  $t = 0$  from previous-cycle values  $A_p$  and  $B_p$  to the current values  $A_0$  and  $B_0$ . We assume that continuous time has been discretized into evenly-spaced time points  $t = 0, t = 1, t = 2$ , etc.

$Q_0$  corresponds to unrolling the node  $Q$  at  $t = 0$ . Naïvely, it would be a function of inputs  $A$  at  $t = -2$  and  $B$  at  $t = -3$ . Instead of this negative time, we use the previous-cycle input values. That is, when the path length is greater than the value of time we are unrolling for, the output node becomes based on the previous cycle’s values instead. Similar conditions are exhibited at  $t = p$ , and  $t = 1$  as well. At  $t = 2$ ,  $Q$  is now a function of input  $A$  at  $t = 0$ , but still depends on  $B$  from the previous cycle. In the next time point,  $t = 3$ ,  $Q_3$  is a function of  $B$  at  $t = 0$ , and  $A$  at  $t = 1$ . Note that  $Q_3$  is a function of  $A$  at  $t = 0$  instead of  $t = 1$ . This corresponds to our assumption that the primary inputs are released by flip-flops once per clock cycle, and do not change value until the next rising/falling clock edge.

We have now created an unrolled network that is completely Boolean, with no timing behavior. In our global flow, we would start at a primary output node and work backwards to the primary inputs unrolling as shown. We now consider unrolling a network using a more realistic data-dependent delay model.

### 2.2 Unrolling using Data-Dependent Delay

Before describing the unrolling using a data-dependent delay model, we first highlight the impact of input transition on delay through a simple NAND gate example. Using SPICE simulation, we obtained the delay values shown in Table 1 as a function of input values for a NAND gate with inputs  $A$  and  $B$  and output  $C$ . The logic values for the output are also included in the table.

Transitions	Delays	Logic Value at Output
$A \downarrow, B_{\downarrow 1}$	476ps	1
$A_{\downarrow 1}, B \downarrow$	850ps	1
$A \downarrow, B \downarrow$	285ps	1
$A \uparrow, B_{\downarrow 1}$	259ps	0
$A_{\downarrow 1}, B \uparrow$	384ps	0
$A \uparrow, B \uparrow$	460ps	0

Table 1. Delay Characterization for NAND gate (from SPICE)

There are three input transitions that lead to a rising node  $C$  and three input transitions that lead to a falling node  $C$ . We will have a rising node at  $C$  if  $A$  falls and  $B$  stays high ( $A \downarrow, B_{\downarrow 1}$ ),  $A$  is a constant high and  $B$  falls ( $A_{\downarrow 1}, B \downarrow$ ) or both  $A$  and  $B$  fall ( $A \downarrow, B \downarrow$ ). Similarly, a falling node  $C$  can occur if input  $A$  rises and  $B$  holds at high ( $A \uparrow, B_{\downarrow 1}$ ),  $A$  holds at high and  $B$  rises ( $A_{\downarrow 1}, B \uparrow$ ) or both inputs rise ( $A \uparrow, B \uparrow$ ). We see that there is a wide range of delay values in our gate-delay model given the different possible input transitions. Similar results would be found for other gate types. In our implementation approach, this gate-delay characterization is assumed to have been done as a preprocessing step for all the gates in our library. While gate characterization is not a trivial process, our approach does not address this step. Instead, our timing tool is independent of the characterization and requires as a starting point a library of gate delays that have been acquired in any appropriate characterization process.

A transport-delay model cannot completely model the data dependent delay behavior just described. Consider a simple buffer with rising delay  $d = 10$  and falling delay  $d = 1$ . Assume its input starts low, then rises at  $t = 20$  and falls at  $t = 21$ . According to a transport-delay model, the output should start low, rise at  $t = 30$ , and fall at  $t = 21$ . In other words, our transport-delay model breaks when the input moved faster than the gate delay. To better handle this situation, we now move to an inertial-delay model, which rejects any input pulses shorter than the gate delay.

Given the above assumption for inertial-delay and the gate delays acquired during characterization, an accurate data-dependent delay model can be constructed, to be used later when the network is unrolled. We explain the delay model using a 2-input NAND gate; however, the process can be generalized to all other gate types. Assume data-dependent delays for the NAND gate as shown in Table 2 which is a simplified version of Table 1.

The output will rise at time 10 if

$$C(10) \uparrow = A(4) \cdot \overline{A(5 : 9)} \cdot B(5) + A(1) \cdot B(0) \cdot \overline{B(1 : 9)} + A(6) \cdot \overline{A(7 : 9)} \cdot B(6) \cdot \overline{B(7 : 9)}. \quad (1)$$

Transitions	Delays	Logic Value at Output
$A \downarrow, B \uparrow$	5	1
$A \uparrow, B \downarrow$	9	1
$A \downarrow, B \downarrow$	3	1
$A \uparrow, B \uparrow$	3	0
$A \uparrow, B \downarrow$	4	0
$A \downarrow, B \uparrow$	5	0

**Table 2. NAND gate delays used for the unrolling example.**

Intuitively, each line of the equation corresponds to one of the three falling-input cases in the table. Using the delays given in Table 2, the first line shows  $C$  rising at  $t = 10$  as a result of  $A$  falling at  $t = 5$  (i.e.,  $A(4) \cdot \overline{A(5)}$ ) and  $B$  staying constant high (i.e.,  $B(5)$ ). We further require that  $A$  remain low without glitching, hence  $A(4) \cdot \overline{A(5 : 9)}$ . Note that we allow  $B$  to go low after  $t = 5$ , since this would only maintain the rising output transition. Similarly, the two other possible conditions that lead to a rising node at  $C$  at time  $t = 10$  are expressed in lines 2 and 3 of Eqn. 1.

The corresponding falling node can be described in a similar manner and combined with rising delay to describe the conditions for a transition at node  $C$ . That is,

$$C(10) \downarrow = \overline{A(4)} \cdot A(5 : 9) \cdot \overline{B(4)} \cdot B(5 : 9) + \overline{A(6)} \cdot A(7 : 9) \cdot B(7 : 9) + A(6 : 9) \cdot \overline{B(5)} \cdot B(6 : 9) \quad (2)$$

$$C(10) = C(10) \uparrow + C(9) \cdot \overline{C(10) \downarrow} \quad (3)$$

The value at node  $C$  at time  $t = 10$  will either be high (if it rises at  $t = 10$ , or if it was previously high and does not fall at  $t = 10$ ), and would be low otherwise.

We have now described the timed gate output as an untimed combinational function of the gate inputs. This model can be easily extended to include other elements that affect delay of a gate. For example, to include the effect of fan-out on delay, we can express the gate delay,  $D_g$  for a particular input transition,  $Y$ , using the following formula:

$$D_g(Y) = \alpha(Y) + \beta(Y) \cdot \gamma, \quad (4)$$

where  $\alpha(Y)$  is the inertial-delay for transition  $Y$ ,  $\beta(Y)$  is the fanout ratio (inversely proportional to the drive strength of the gate) for the same transition  $Y$ , and  $\gamma$  is the total capacitance seen at the output of the gate. This extension requires collecting additional delay information during the gate delay characterization process; however, such constant load analysis is typically part of standard delay characterization. With this extension, instead of accessing a single value to obtain  $D_g(Y)$  during the unrolling process, the delay is computed “on the fly” using Equation 4. Using this new delay value during the unrolling process will lead to more accurate delay information. However, it is important to note that including the fanout extension will not significantly impact the complexity of the unrolling process since we are using a constant load; the total number of clauses generated, whether or not fanout is considered, should be about the same. Algorithms that handle the unrolling will be explained in Section 4.

The resultant Boolean network can be analyzed with any Boolean solver. For our purposes, we generate network-satisfiability clauses in product-of-sums (CNF) form and then use a high-quality public-domain SAT solver. We can ask the SAT solver any timing analysis question by adding a small number of extra clauses; the exact

details of this are described in the following section.

### 3. Timing Analysis as a Boolean Satisfiability Problem

We are now ready to pose our timing analysis question as a decision problem. The first step is to convert the network itself into CNF form for a SAT solver. We do this in the standard manner, generating satisfiability clauses [12] for each gate in the network. We then add a small number of clauses to represent the particular query we would like to make. Any query about timing must be translated into product-of-sum clauses. To illustrate this process, let us focus on the specific question of finding the latest rising transition at the output.

If glitches were not an issue, a simple binary search would quickly narrow down the answer. Assume we know that no rising transition can occur after  $t = 100$  (e.g., from a simple topological search). We would first add the two clauses  $Q_0 = 0$  and  $Q_{100} = 1$  and call the SAT solver. It would return true if there were a transition anywhere in the range  $(0, 100)$ . If so, we might next remove these two clauses and instead add the two clauses  $Q_{50} = 0$  and  $Q_{100} = 1$  and try again. We would thus quickly narrow in on the latest transition, with only a logarithmic number of calls to the solver. Thus, by simply adding two trivial timing clauses to our satisfiability clauses, the SAT solver can find any input patterns that generate transitions within a given range.

However, if glitches do exist, and we need to detect them, then a binary search is not sufficient. Consider the circuit  $Q = OR(BUF(A), INV(A))$ , where the buffer delay is 10, the inverter delay is 9, and the OR delay is 1. Though the output is logically a constant 0, a rising input will cause a unit glitch at  $t=10$ . Our initial query would tell us that there is indeed no way for the output to be low at  $t = 0$  and high at  $t = 100$  (assuming again that inputs change once at  $t = 0$ ), causing the binary search to abort. We would thus miss the glitch.

We might consider, as an extreme, doing a linear search, checking for a rising transition at every single time point, from latest to earliest, and stopping when we find one. Our first call to the SAT solver would add the clauses  $Q_{99} = 0$  and  $Q_{100} = 1$ . If this failed, we would instead try  $Q_{98} = 0$  and  $Q_{99} = 1$ , until we found a transition. Though somewhat slow, this might seem foolproof.

Unfortunately, even the linear search is imperfect. Consider setting our buffer delay to  $d = 9.7$  and our inverter delay to  $d = 9.6$ . Remember that, though analog gate delays are continuous, we quantize time into digital intervals. If we quantize to single units, both the buffer and inverter will have  $d = 10$ , and the glitch will “vanish.” So, if we make our quantization interval too large, we may lose glitches; if we make it too small, compute time for our linear search will suffer.

Instead, we steer a middle course. We first do a linear search, using a fairly coarse time discretization, to narrow down what region the latest transition is in. Once we have found that region, we do another linear search of just that region, using a finer-grained discretization. Using a modulus of 10, for example, we would first divide our search space into 10 pieces and search for transitions in the ranges  $t = 90 - 100$ ,  $t = 80 - 90$ ,  $t = 70 - 80$ , etc. If we found a transition in  $t = 80 - 90$ , the next iteration would use a step-size of 1, searching for transitions in  $t = 89 - 90$ ,  $t = 88 - 89$ ,  $t = 87 - 88$ , etc.

Like the binary search, this takes  $O(\log(P))$  calls to the SAT

solver to split down an interval into  $P$  portions. It is indeed an  $N$ -ary search (with  $N = 10$  in this example) — by setting  $N$  significantly larger than 2, we resemble a linear search more closely, and hence miss fewer glitches. Of course, due to the initial coarse discretization, we may miss some glitches.

## 4. Implementation Algorithm

Our timing analysis tool takes a gate-level netlist, identifies the primary output node and its corresponding fan-in and stores the fan-in as a Directed-edge Acyclic Graph (DAG). At this point, we are ready to find the delay of this primary output by calling the function `FIND_DELAY()` with this DAG and initial delay estimates as parameters. The pseudo-code for this function is shown in Figure 3. The function includes the three major components of our approach: unrolling the network at various time points, generating time constraints, and iteratively calling the SAT solver until the Boolean constraints are satisfied. The logic and timing constraints are generated using the `unroll()` and `find_rising_transition()` functions respectively. The process to find the delay is an iterative part of the `FIND_DELAY()` function.

```

Procedure FIND_DELAY(DAG, max_delay, min_delay) {
1.  _stepsize = 1
2.  unroll(min_delay, max_delay, _stepsize)
3.  time = max_delay
4.  while(!find_rising_transition(time))
5.    time = time - _stepsize
6.  if(accuracy = 0 decimal point)
7.    return(time)
8.  _stepsize = _stepsize / 10
9.  unroll(time - 1, time, _stepsize)
10. while(!find_rising_transition(time))
11.   time = time - _stepsize
12. if accuracy = 1 decimal point
13.   return(time)
14. _stepsize = _stepsize / 10
15. unroll(time - 0.1, time, _stepsize)
16. while(!find_rising_transition(time))
17.   time = time - _stepsize
18. if accuracy = 2 decimal point
19.   return(time)
20. ...

```

**Figure 3. Procedure FIND\_DELAY.**

The `min_delay` and `max_delay` passed to the function are initial lower and upper bounds on the critical path delay of the output. Note that 0 may be used if a lower bound is not available. The first step in our implementation is to unroll the network for each time point between these minimum and maximum delay estimates in steps specified by the `_stepsize`. Next, the timing question is translated into timing constraints. The `find_rising_transition()` routine generates these timing constraints (for example, node  $X$  is low at the starting timepoint and high at the end), combines them with the logic constraints from `unroll()` and poses the timing/satisfiability question to the SAT solver.

There are two levels of iteration in `FIND_DELAY()`. The first one iterated over successively more accurate step sizes (1, 0.1, 0.01, ...) while the second one iterates over successively decreasing time points (given a certain step size). The second level of iterations are represented by the `while` loops in the pseudo-code. For a given unrolled network, each iteration of the while loop is done for time points starting at `max_delay` (which has to be greater than or equal to the true delay) and for time points decreasing by

the current step size until the actual delay value is reached (i.e., until the SAT solver finds a delay value that matches the timing question posed). To illustrate this, assume that an output node has a latest rising transition at  $20.67ps$  and the initial estimate from a topological analysis is  $25ps$ ; note that the initial estimate is a rough upper-bound. The network is first unrolled for time points 25, 24, 23, ... until the `min_delay` is reached. In the first `while` loop in Line 4, `find_rising_transition()` will generate the timing constraint for the conditions of a rising delay at  $t = 25ps$ . These constraints will be sent to the SAT solver as the timing question *Does the circuit rise at 25ps?* Since 25ps is not the rising delay, the SAT-solver will provide a not-satisfiable solution prompting the function `find_rising_transition()` to be called again for  $t = 24$ . This iterative process continues until we reach  $t = 21ps$ . At this time point, the response from the SAT solver will be positive (satisfiable solution) which denotes that the delay for the `_stepsize = 1` resolution is  $21ps$ . If the accuracy required is higher than this resolution, the `_stepsize` is adjusted and the iteration continues.

Next, a new unrolled network is generated for time points 21.0, 20.9, 20.8, ..., 20.0 and in the second `while` loop (Line 10) `find_rising_transition()` will be iteratively called until time point  $t = 20.7ps$  is reached. At this point, the `_stepsize = 0.1` iteration will stop due to a satisfiable solution from the SAT solver. This search then moves to the `_stepsize = 0.01` iteration to zoom into the actual delay value of  $20.67ps$  using the steps outlined above.

Note that the granularity of the time points directly impacts the size of our unrolled network. A smaller step size leads to a larger unrolled network which translates to a bigger decision problem that has to be solved by the SAT solver. Fine grained granularity is possible, but the computation will be expensive. Thus, trade-offs between accuracy of delay and computation time can be determined using the `_stepsize` parameter. When the time point that satisfies the specified accuracy is found, that time point is returned as the response to the timing analysis question that is posed. For instance, if an accuracy of only 1 decimal point is required for the example above, the procedure `FIND_DELAY` would have returned the value 20.7 from Line 12 in the pseudo-code. The high level iterative search to find the delay with the specified accuracy can be done as a binary search or by using any other efficient search algorithm. The above routine is one such approach.

## 5. Experiments and Results

Our timing analysis tool has three component; network unrolling, generating SAT clauses to represent the circuit and its timing constraints, and solving the resulting decision problem with a SAT solver. We chose to use the ZChaff SAT solver, from Princeton University, with our tool [14]. ZChaff has been shown to handle a large number of clauses (i.e., 1 million variables and 10 million clauses). We expect this capacity to meet our needs. The first and main goal of our experiments is to highlight the difference between the delay values that are reported by a static timing tool and those from our SAT-based timing analysis tool. Although several commercial STA tools are available, we chose to compare our results to SIS [18] because of controllability and ease of use. Specifically, the experiments tried to determine the longest critical delay-path through a network of gates. For our purposes, the network of gates was determined by the fan-in cone of a primary output of each circuit. Using this approach, one could find the critical delay-path in any circuit by identifying the primary output with

Circuits	Depth	Transition	SAT-Delay	SIS-Delay	% Difference	Run time(s)	Avg.# Clauses
<i>con1</i> [ <i>fl</i> ]	6	Rising	15	18	16.7	4.5	19713
		Falling	15	18	16.7	4.5	19772
<i>ecc3</i> [ <i>d2out</i> ]	8	Rising	23	29	20.7	20.5	79914
		Falling	24	29	24.1	22.0	37158
<i>t481</i> [ <i>v16.0</i> ]	9	Rising	22	27	18.5	41.3	44541
		Falling	23	28	17.9	36.5	42843
<i>vg2</i> [ <i>25.1</i> ]	10	Rising	25	29	13.8	69.8	72434
		Falling	26	29	10.3	132.8	186748
<i>5xp1</i> [ <i>v7.2</i> ]	11	Rising	34	41	14.6	57.3	139548
		Falling	35	41	17.1	46.1	147460
<i>clip</i> [ <i>v9.3</i> ]	13	Rising	35	47	25.5	57.1	206870
		Falling	41	49	16.3	73.6	245877
<i>rd</i> [ <i>73</i> ][ <i>v7.0</i> ]	14	Rising	43	53	18.9	191.8	523420
		Falling	39	54	27.8	163.8	469184
<i>apex</i> [ <i>v117.3</i> ]	15	Rising	40	58	31.0	100.5	197860
		Falling	37	57	35.1	113.4	173645
<i>sao2</i> [ <i>v10.1</i> ]	16	Rising	45	58	22.4	181.8	356620
		Falling	48	58	17.2	222.5	382923
<i>ex4</i> [ <i>v128.5</i> ]	17	Rising	46	65	29.21	267.8	355708
		Falling	53	66	19.7	225.6	440688
<i>duke</i> [ <i>v22.6</i> ]	21	Rising	51	83	38.6	182.1	369880
		Falling	50	81	38.3	193.2	375606

**Table 3. Critical path analysis results for selected MCNC benchmark circuits**

the largest delay. We selected circuits from the MCNC benchmark suite using range of depth and size as selection criteria.

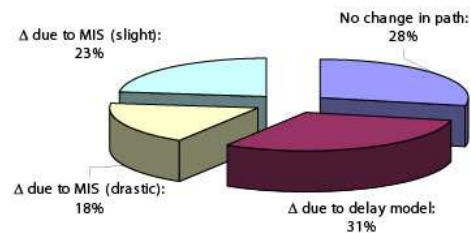
The benchmark circuits were mapped for delay using the `map -n 1 -AFG` command from SIS with a library of basic gates (2-input NOR/NAND and INV). The static timing analysis tool within SIS requires that each pin-to-pin delay of a gate in the above library be assigned two delay values (one for the rising transition and another for the falling transition). On the other hand, our data-dependent tool requires a series of delay values for each gate that corresponds to the various rising and falling conditions. This implies that every combination of possible transitions can have a unique delay. To make a fair comparison between the STA tool and the data-dependent tool, the delay values for SIS and our data-dependent tool have to be similar. However, the two delay components are not identical because SIS only allows for pin-to-pin delays based on a single input transition. As a good approximation, when creating the SIS delay library, a pin-to-pin delay was determined by the maximum of all corresponding delays. For example, if the data-dependent falling delays for a 2-input NAND gate are 2ps (first input rising), 3ps (second input rising) and 5ps (both inputs rising); the pin-to-pin falling delay for either input would be 5ps.

Table 3 shows results from our first experiment. All experiments were run on a Pentium D machine running at 3.2GHz with 4GB of RAM. We computed the longest critical-path delay for various networks of gates (fan-in cones) from the MCNC benchmark circuits using both our tool and SIS. The first column lists the benchmark circuits and the primary output names that correspond to the fan-in cones. The size for the fanout cones range from 20 gates to 160 gates. The rising and falling delays are shown under SAT-Delay and SIS-Delay. As the results show, the SIS-based delay is always pessimistic and overestimates the actual critical path. This is clearly shown in column 6 which lists the percent difference between the SIS values and those computed by our tool. The worst case difference is close to 39%, and the best case is about 13%. We can also see from this data that the correlation between the static and data-dependent tool deteriorates as the size of the circuit grows.

The run time for our SAT-based tool is shown in column 7, and

column 8 shows the average number of clauses that were generated for the SAT solver. The run time for SIS to synthesize the logic and compute the delay values was only a few seconds. As expected, the number of clauses generated is a function of the size of the fan-in cones as well as the granularity of the unrolling time — unrolling every time unit will generate more clauses than unrolling every 10 time units. For our granularity of 1/10th of a gate delay, the number of clauses is well within the limit of the 10 million clauses that the SAT solver can handle. The computation time also follows the same trend; the bigger the circuit and the smaller the granularity, the higher the computation time. Nevertheless, the run time to recurse through various timepoints and reach the correct delay was always less than 300 seconds.

To further understand the strengths and limitations of our tool, we extrapolated the data from the benchmark circuits and found that a *fan-in cone* of up to 500 gates at a granularity of 1/10th of a gate delay would still be within the limits of the SAT solver. Looking at the breakdown of the run time, we were able to see that the bottleneck is in the amount of time that the SAT solver takes to answer the decision problems, especially as the circuit size grows. Therefore, the efficiency of our tool will benefit from the extraordinary effort that is being invested in developing faster SAT solvers. What is clear is that our tool is able to give accurate delay values and path information at a reasonable run-time. Furthermore, our tool will provide not only the critical path but also the input vector that stimulates the path. This is very advantageous in post-silicon validation where patterns are used in speed and delay-fault testing as well as for hypothesis generation for fault isolation and speed debug, thus allowing the designer to minimize the vast amount of redundant effort during post-silicon debug.



**Figure 4. Path Analysis Breakdown**

One cause for the difference in delay between our tool and SIS is the amount of information that is included in the delay models. However, our goal is to identify all other causes that contribute to the miscorrelation. To get a better understanding, we computed the critical path (rising and falling) for 20 MCNC benchmark circuits using both approaches and extracted the predicted paths. By comparing each pair of paths, we were able to observe four different scenarios. As shown in Figure 4, about 18% of the paths were drastically different due to a multiple input switching (MIS) that could not be identified by the static timing tool. About 23% of the paths also had MIS that was close to the primary inputs and thus did not cause a major change in the computed paths. However, it was still a different path with different delay contributions and thus affected the delay values. These two scenarios clearly illustrate the importance of delay models that consider MIS; without a pattern-dependent delay model, the timing analysis tool could produce an incorrect critical path. About 31% of the paths differed due to the fact that the data-dependent model had more detailed delay information available during the analysis than the pin-to-pin delay information used by SIS; i.e., it wasn't specifically multiple input switching that accounted for the different critical paths. Finally, about 28% of the paths were identical. However, this does not mean that the delays were identical as well (again, due to the differing delay models). In the post-silicon world where extracting the correct path is a vital step in the debug and validation process, a timing analysis tool that does not provide the correct path will severely slow down the debug effort.

## 6. Conclusions and Future Work

We have presented a timing analysis tool that aims to integrate logic and delay information into the analysis. By better accounting for patterns and multiple inputs switching, we hope to obtain better correlation with post-silicon analysis. Using a data-dependent delay model, we have developed a timing analysis tool that is able to determine the longest critical delay-path through a logic cone by unrolling the cone and posing the timing analysis question as a decision problem. This allows an iterative search for the delay by using a SAT solver to answer the timing questions. The results from our benchmark experiments show that data-dependent timing analysis using our tool is viable in terms of run-time and delay accuracy.

The results have shown the importance of a data-dependent timing tool in pre-silicon analysis. Future work will look to model other factors such as fan-out and crosstalk and extend our tool to include these models. Other avenues of extension include using our pattern-dependent delay model for post-silicon timing to generate patterns for speed testing and delay-fault testing as well as for hypothesis generation for fault isolation and speed debug.

## 7. References

- [1] S. Pilarski A. Pierzynska. Pitfalls in delay fault testing. *IEEE Transactions on CAD*, 16(3):321–329, Mar. 2002.
- [2] F. Aloul, S. Hassoun, K. Sakallah, and D. Blaauw. Robust sat-based search algorithm for leakage power reduction. In *PATMOS '02*, pages 167–177, 2002.
- [3] A. Biere, A. Cimatti, E. Clarke, O. Strichman, and Y. Zhu. *Advances in Computer Science*, volume 58, chapter Bounded Model Checking. Academic Press, 2003.
- [4] A. Biere, A. Cimatti, E. M. Clarke, M. Fujita, and Y. Zhu. Symbolic model checking using SAT procedures instead of BDDs. In *ACM/IEEE Conference on Design automation*, pages 317–320, 1999.
- [5] D. Blaauw, V. Zolotov, S. Sundareswaran, C. Oh, and R. Panda. Slope propagation in static timing analysis. In *ICCAD*, Nov. 2000.
- [6] P. Chen and K. Keutzer. Towards true crosstalk noise analysis. In *ICCAD*, pages 132–138, 1999.
- [7] P. Chen, D. Kirkpatrick, and K. Keutzer. Switching window computation for static timing analysis in presence of crosstalk noise. In *ICCAD*, pages 331–337, Nov. 2000.
- [8] S. Devadas, K. Keutzer, S. Malik, and A. Wang. Computation of floating mode delay in combinational circuits: practice and implementation. *IEEE Transactions on CAD*, 12(12):1924–1936, December 1993.
- [9] A. Gattiker, S. Nassif, R. Dinakar, and C. Long. Timing yield estimation from static timing analysis. In *International Symposium on Quality Electronic Design*, page 437, 2001.
- [10] J. Grodstein, D. Bhavsar, V. Bettada, and R. Davies. Automatic generation of critical-path tests for a partial-scan microprocessor. In *ICCD '03*, page 180, 2003.
- [11] A. Krstic, J. Liou, Y. Jiang, and K. Cheng. Delay testing considering crosstalk-induced effects. In *IEEE International Test Conference*, page 558, 2001.
- [12] T. Larrabee. Test pattern generation using boolean satisfiability. *IEEE Transactions on CAD*, 11(1), January 1992.
- [13] L. Lee, L. Wang, T. Mak, and K. Cheng. A path-based methodology for post-silicon timing validation. In *ICCAD*, pages 713–720, Nov. 2004.
- [14] Y. Mahajan, Z. Fu, and S. Malik. Zchaff2004: An efficient sat solver. *Lecture Notes in Computer Science*, 3542:360–375, 2005.
- [15] P. Maurer. Two new techniques for unit-delay compiled simulation. *IEEE Transactions on CAD*, 11(9):1120–1130, Sept. 1992.
- [16] K. Nepal, H. Song, R. Bahar, and J. Grodstein. Resta: a robust and extendable symbolic timing analysis tool. In *GLSVLSI*, pages 407–412, 2004.
- [17] S. Devadas, K. Keutzer, S. Malik, and A. Wang. Certified timing verification and the transition delay of a logic circuit. *IEEE Transactions on VLSI Systems*, 2(3):1063–8210, September 1994.
- [18] E. M. Sentovich, K. J. Singh, C. Moon, H. Savoj, R. K. Brayton, and A. Sangiovanni-Vincentelli. Sequential circuit design using synthesis and optimization. In *ICCD*, Oct. 1992.
- [19] J. Silva and K. Sakallah. Dynamic search-space pruning techniques in path sensitization. In *DAC*, pages 705–711, 1994.
- [20] J. Silva and K. Sakallah. Efficient and robust test generation-based timing analysis. In *ISCAS*, pages 660–663, Aug. 2003.
- [21] L. Silva, J. Silva, L. Silveira, and K. Sakallah. Satisfiability models and algorithms for circuit delay computation. *ACM Trans. on Design Automation of Electronic Systems (TODAES)*, 7(1):137–158, 2002.
- [22] S. Sun, D. Du, and H. Chen. Efficient timing analysis for CMOS circuits considering data dependent delays. In *ICCS '94*, pages 156–159, 1994.