A Symbolic Simulation-Based Methodology for Generating Black-Box Timing Models of Custom Macrocells

Clayton B. McDonald (cbmcdonald@alumni.carnegiemellon.edu) [*]
Randal E. Bryant (randy.bryant@cs.cmu.edu)
Electrical and Computer Engineering
Carnegie Mellon University

Abstract

We present a methodology for generating black-box timing models for full-custom transistor-level CMOS circuits. Our approach utilizes transistor-level ternary symbolic timing simulation to explore the input arrival time space and determine the input arrival time windows that result in proper operation. This approach integrates symbolic timing simulation into existing static timing analysis flows and allows automated modelling of the timing behavior of aggressive full-custom circuit design styles.

1 Introduction

Over the last 20 years, static timing analysis[3] has become the dominant approach for the timing verification of integrated circuits, primarily due to its high efficiency and capacity. The majority of work on static timing analysis techniques has been focused on analyzing gate-level representations, and the path-tracing and delay-calculation engines for these tools are quite mature.

However, analysis of transistor-level representations is considerably more complex, although a number of tools have been developed for this purpose[9, 11]. While most of the techniques from gate-level analysis transfer nicely, there remain additional issues which have not yet been adequately resolved. The most difficult of these is the identification of the timing constraints implied by clocked logic structures. Since the transistor-level representation does not include information on functional requirements such as latch setup and hold times, the timing analyzer must derive them using pattern matching or other heuristic techniques. While these heuristics can be quite successful for a given design style, it is nearly impossible to keep up with innovative new circuit structures or single-use optimizations created during full-custom design. As a result, full-custom designers are forced to spend countless hours reviewing the analyses produced by transistor-level static timing analyzers to make sure their circuits were properly interpreted.

Symbolic timing simulation (STS) [6] is an alternative approach for the verification of full-custom transistor-level circuits. Like conventional timing simulation, it verifies timing by checking functionality under a given timing model, rather than explicitly verifying delay paths against implied timing constraints. Through the use of symbolic encodings, STS can verify timing over all possible input patterns, thus achieving nearly the same verification power as static timing analysis. Furthermore, STS can be applied to arbitrarily complex full-custom digital design styles without requiring sophisticated heuristics.

While STS has substantial advantages over static timing analysis in terms of the breadth of circuit styles that can be reliably analyzed, static timing analysis has vastly higher capacity. Ideally, we would like to use STS only where necessary, while static analysis would be used for more straightforward portions of the design. Fortunately, such a divide and conquer strategy is already well-developed in static timing analysis[13, 9, 2]. Typically, large macrocells are assigned to one or two design engineers, and are designed and analyzed in isolation. When a certain level of quality is reached, different types of timing models can be generated which capture the timing behavior of the input and output signals. The timing models from each macrocell on the chip can then be composed to determine if inter-block timing requirements are being met. In some cases, even multiple levels of composition are used, resulting in a hierarchy of timing models.

This paper discusses a method for performing selected low-level analyses with STS, and generating timing models of the same form as those generated by static analyzers. In this way, STS can be used on blocks containing the most difficult circuit styles, while more efficient static analysis is used elsewhere. The results from both types of block analyses can then be composed into full-chip-level runs using the existing hierarchical static analysis methodology.

A number of other researchers have addressed the generation of black box timing models through simulation. Generally, this work has concentrated on producing high accuracy models of library cells, for use in gate-level static timing analyzers. Cirit [1], Vo[12] and Patel[10] propose methods for ASIC standard-cell characterization based on iterative SPICE analysis. While our approach also relies on iterative simulation, it differs in that we use symbolic methods to analyze much larger blocks of logic. Like static timing analysis-based methods, ours also produces a more approximate and courser-grained model than direct SPICE analysis of library cells.

There are a number of important sub-problems in the development of timing models, such as determining output drive strengths, pin capacitances, pin-to-pin delays, and input setup and hold times. The most difficult of these by far is the determination of setup and hold times, especially in the presence of multiple interacting inputs. This paper focuses on an iterative symbolic simulation approach to

[*]This research was supported by the SRC (contract DC-068).

compute setup and hold times of inputs to large custom macrocells based solely on a model of functional correctness.

The following section presents two levels of block timing models in detail. Section 3 then describes a methodology for computing setup and hold times using symbolic timing simulation. Section 4 gives our results, and 5 will conclude.

2 Timing Models

Block timing models attempt to capture the timing behavior of significant blocks of logic. As one would expect, various levels of detail are possible, and the two most common are typically termed "black-box" and "gray-box". These terms refer to the level of internal detail that is made visible to the block's environment.

Black-box models abstract away all information regarding the internal nodes of the circuit block. The only information retained are setup and hold constraints on the input signals and the arrival times of the outputs relative to a particular clock edge. In some cases, purely combinational delay values are also included.

Gray-box models[9, 2] retain additional timing information to model latch transparency. They typically contain absolute timing arcs between latching points, from clocks and inputs to latches, and from latches to outputs. In this manner, the gray-box models can perform latch transparency checks, and are valid at all operating frequencies. At the cost of some efficiency and frequency independence, black-box models can be built which mimic the behavior of gray-box models using the aliasing technique described in [8].

For each of the entries in black-box or gray-box models, it is also often possible to specify separate data for rising and falling transitions. For normally-ratioed static CMOS, where the timing behaviors of the rising and falling transitions are symmetric, this separation has little value. However, for dynamic logic styles such as domino, this ability is crucial, and we typically even need to specify different clock-phases for each transition direction.

3 Methodology

Our methodology for generating timing models uses iteration to identify the range of input arrival times that lead to correct operation. As with our approach to timing analysis, correctness is defined solely by functionality, rather than explicit identification of the timing constraints implied by certain circuit topologies such as latches or domino gates.

3.1 Signal Model

In modelling the behavior of input signals, we wish to capture several important aspects. First, we must capture a range of possible arrival times, typically specified as a $\langle min, max\rangle$ window, and guarantee that they satisfy setup and hold checks. Second, to handle dynamic and other non-symmetric design styles, we want to be able to separate the timing behavior of rising and falling transitions.

One of the keys to our approach is the use of a ternary signal model, having values of 0,1, and 'X'. The additional 'X' value, first introduced by Jephson et.al.[4], can be used to capture unknown initial states and to model the values of inputs up until their stabilization times. The second key, which enables exhaustive simulation of substantially larger systems, is the use of symbolic encodings. With symbolic timing simulation, we can apply Boolean variables to the inputs of the circuit rather than constant 0's and 1's, and these variables are propagated through the circuit as Boolean functions. If the functions on the output nodes and stored states match the functional description of the block, then we know that the arrival times specified satisfy any setup and hold conditions.

To verify setup constraints, we must check if the latest possible arrival time of a signal satisfies the setup constraints of the block. This can be modelled under our ternary simulation model by placing an 'X' on the input initially, and then scheduling a transition to the valid Boolean value at the maximum arrival time.

Hold times are somewhat more subtle. A hold time check is a requirement that a signal be valid until a particular time, in order to allow proper sampling by the receiving circuit. Static analyzers verify that the earliest possible arrival time of an input signal satisfies this hold check. However, in reality, a hold check limits how early a signal can switch to a new value in the *next* state. This is one of the fundamental reasons that static timing analysis models are limited to synchronous systems. Thus, a $\langle min, max\rangle$ arrival time window for an input signal translates into input transitions in two different states.

Figure 1 shows the timing diagram used to model the behavior of an input taking the Boolean variable *a*. Note that the 'X' value does not mean "invalid", but rather "unknown". In other words, it captures the fact that the input might be high or low or even glitching uncontrollably between these two values. Thus we can model the signal as being 'X' initially, taking it's specified value at the max arrival time, and then returning to 'X' at the min arrival time in the subsequent clock cycle. If the circuit being verified computes the correct output function, then we can say that the input arrival time window satisfies the circuit's setup and hold requirements.

To separate rising and falling transitions on an input signal, we must introduce additional steps. Let us assume that the latest rising transition will occur before the latest falling transition, as shown in Figure 2. In this case, the input should transition at the latest rising time only for cases where the input's valid value *a* is *true*. For cases where the *a* is *false*, the signal should remain in the 'X' state. A similar analysis gives the value of the signal between the earliest rising and falling times in the next cycle. For clarity, Figure 2 also shows the signal behavior separately for each of the two possible values of *a*.

To represent ternary node values ($value \in 0, 1, X$), we utilize a dual-rail encoding scheme with two Boolean functions *value.h* and *value.l*:

$$
\begin{array}{lcl}
value.h = 1, value.l = 0 & : & \text{HIGH} \\
value.h = 0, value.l = 1 & : & \text{LOW} \\
value.h = 1, value.l = 1 & : & \text{UNKNOWN (X)} \\
value.h = 0, value.l = 0 & : & \text{not defined}
\end{array}
$$

As shown in Figure 2, we must create two transitions to capture edge-specific behavior. The first transition goes to an intermediate value that represents the fact that only one edge has arrived, and the second transition places the proper Boolean value on the input node. Using the two-bit encoding, we can compute a mask qualify-

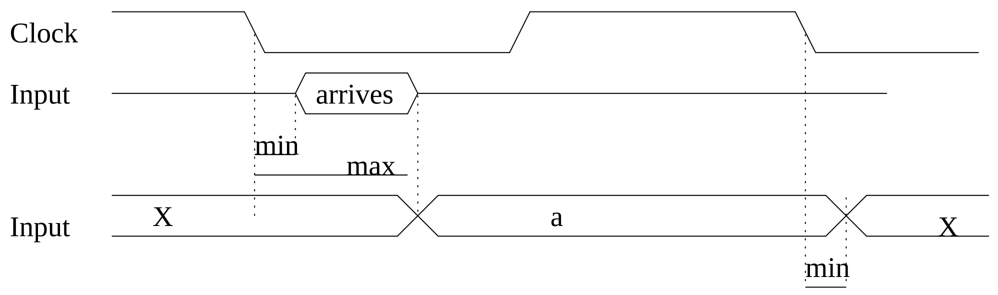

Figure 1: Arrival Time Window Modelling

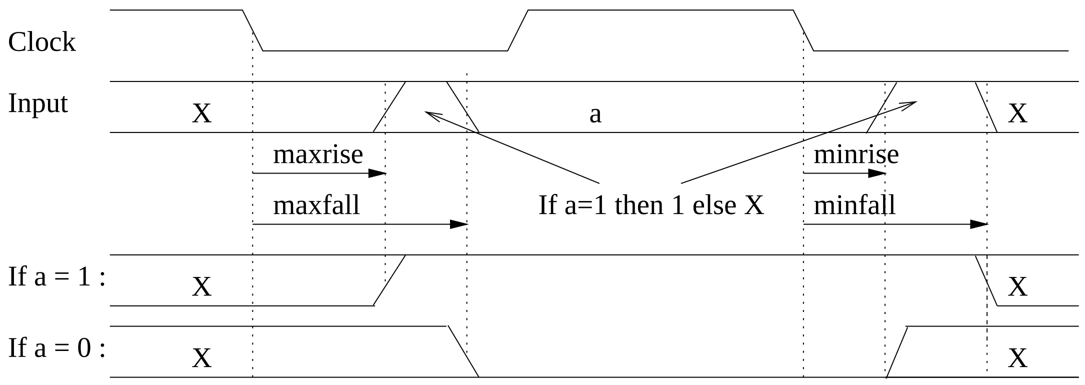

Figure 2: Separating Rise and Fall Transitions

ing any unidirectional transition from ternary value *old* to ternary value *new* as:

Rising only:

$$mask \leftarrow (new.h \wedge \overline{old.h}) \vee (\overline{new.l} \wedge old.l)$$

Falling only:

$$mask \leftarrow (new.l \wedge \overline{old.l}) \vee (\overline{new.h} \wedge old.h)$$

Given this mask, we can compute the intermediate ternary value *int* as:

$$\begin{aligned} int.h &\leftarrow (mask \wedge new.h) \vee (\overline{mask} \wedge old.h) \\ int.l &\leftarrow (mask \wedge new.l) \vee (\overline{mask} \wedge old.l) \end{aligned}$$

3.2 Correctness Criteria

Since our timing verification is based on checking functionality while considering realistic delay values, we must be very careful about how we define functional correctness. In particular, we require a specification of correct behavior that includes both functional and timing information.

For our methodology, we require a model *M*, which is a tuple containing all information needed to stimulate the circuit and determine correctness:

$$M \equiv \langle C, I, IV, IW, O, OF, OW, OI \rangle$$

C	:	Set of clock tuples $\langle \textit{startval}, \phi_1, \phi_2, \textit{offset} \rangle$
I	:	Set of input pins
IV	:	Boolean variables for input pins
IW	:	Set of $\langle min, max \rangle$ tuples for Input arrival time windows
O	:	Set of output pins
OF	:	Expected functions for output pins
OW	:	Set of $\langle min, max \rangle$ tuples for output required time windows
OI	:	Initial values for output pins.

Verification relative to this model is performed by the function *CheckModel* which is basically a wrapper around the symbolic timing simulation engine. It stimulates the input pins based on the input arrival time windows according to the model described in Section 3.1. It then runs the simulation and issues probes to verify that the output nodes matched their expected functions for the duration of the output required time window. Internal state nodes requiring initialization and validation are handled in the same manner as outputs, except that they are set to the initial values specified in *OI* at their latest arrival time in the cycle previous to their required time window, and then left to float as driven by the circuit.

CheckModel returns a flag indicating whether or not the circuit outputs were correct, and, as will be discussed below, the *equivalence window* for a specified input transition.

3.3 Model Generation

Now that we have a model for correct behavior, we need to develop a strategy to determine valid ranges for input arrival times. Our basic approach is to start with a nominal set of arrival times, for which it is known the circuit operates properly, and grow each arrival time into the largest possible arrival time range without altering circuit functionality.

However, the handling of multiple inputs requires special attention. Depending on the circuit being simulated, various combinations of arrival times could result in proper operation of the circuit. Figure 3 shows a hypothetical valid arrival time region for a two-input circuit. In other words, this circuit will compute the expected output function for any combination of arrival times whose coordinates fall in the shaded region. In general, these regions need not be convex, nor even continuous.

Based on our input signal model, min and max arrival times are specified relative to clock edges, which remain constant. Therefore, any combinations of arrival times expressible in our model represent rectangular regions (or rather hyper-rectangular regions). Our goal is to identify the largest possible hyper-rectangular region, since this will represent the least restrictive specification of allowable input signal behavior.

We have implemented a greedy strategy, whereby we iteratively select one input transition and maximally expand its arrival time window while keeping all other arrival windows the same. After each window is expanded, it remains maximized while all subsequent input transitions are processed. In our geometric interpretation, the nominal set of arrival times represents an initial point in the arrival time space. Expanding each arrival time window selected corresponds to growing the rectangular region along the associated axis.

Because we expand along each dimension separately, we can potentially obtain different answers by processing the input transitions in various orders, as depicted by the two potential sub-regions in Figure 3. A more concrete example is the dynamic logic stage in Figure 4(a) having one static and one dynamic input signal – a risky but not uncommon tactic in full-custom designs. For proper logical operation, we must guarantee that either both signals stabilize before the clock rises, or that the static signal *A* stabilizes before *B* rises. This combination of constraints is captured by the valid arrival time region in 4(b). The nominal arrival times are depicted in the timing diagrams and marked as an ‘X’ in the arrival time region. Based on our choice of which arrival time window to expand first, we could obtain either of the rectangular subregions shown.

Various search methodologies could be employed for finding the largest arrival time window for a given input. The first such methodology is simple binary search, which proves to be quite effective. However, in some cases, we can utilize the fact that we are using an event-driven simulator to accelerate the process.

3.3.1 Equivalence Windows

We can often accelerate the search by reporting *equivalence windows*, which are ranges of time over which inputs can arrive that are guaranteed to preserve the circuit’s functionality. Since our simulation environment is event-driven, we can examine the event lists to

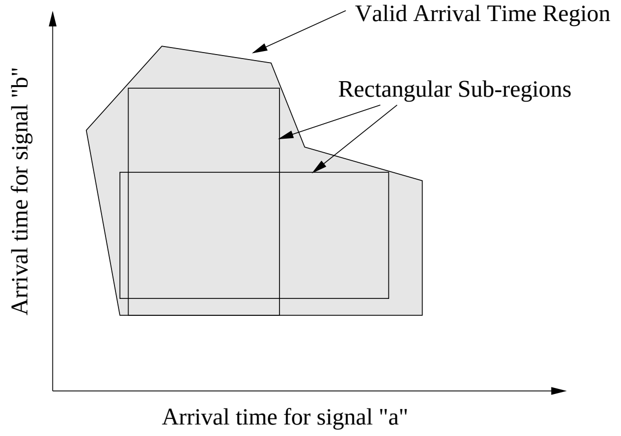
Valid Arrival Time Region
Rectangular Sub-regions
Arrival time for signal "b"
Arrival time for signal "a"

Figure 3: Arrival Time Regions

determine a lower bound on acceptable variations to a particular input’s arrival time. As long as changing an input’s arrival time does not cause any internal events to become re-ordered, we know that the same sequence of states will be visited by every node in the circuit. Consequently, we can safely vary input arrival times right up to the point where re-ordering occurs. By automatically detecting these equivalence windows for input arrival times, we effectively convert our search over the arrival time space from a continuous to a discrete problem.

The basic concept is to mark the input transition of interest, and all internal events which it spawns, as *active*. All other input transitions and internal events are then marked *inactive*. Delaying the *active* input transition will delay all other *active* internal events by exactly the same amount, so long as we can guarantee that all internal node-to-node delays will remain unchanged. Since internal delays are based solely on topology and node state, we need only guarantee that node state will remain unaffected. This in turn is satisfied so long as no event reordering occurs. Taken together, these conditions guarantee that we can delay the *active* input transition up until it causes any *active* event to become re-ordered with respect to any *inactive* event.

Computation of these equivalence windows is quite straightforward, and requires only that we determine the minimum separation time between *active* and *inactive* events. We determine the maximum amount we can retard the event (make it arrive later) as the minimum time separation between any active event and a subsequent inactive event. Correspondingly, the maximum amount we can advance the input transition will be the minimum time separation between any inactive event and a subsequent active event. The only complications arise from the symbolic nature of our events. The reader is referred to [5] for complete details.

3.3.2 Search Algorithm

Our algorithm for identifying these rectangular regions using binary search and equivalence windows is shown in Figure 5. It takes as input a nominal model *M*, and iteratively expands the input arrival time windows of *M* in a greedy fashion.

The function *Grow* is broken into two parts, one each to determine the min and max arrival times for a particular signal. To deter-

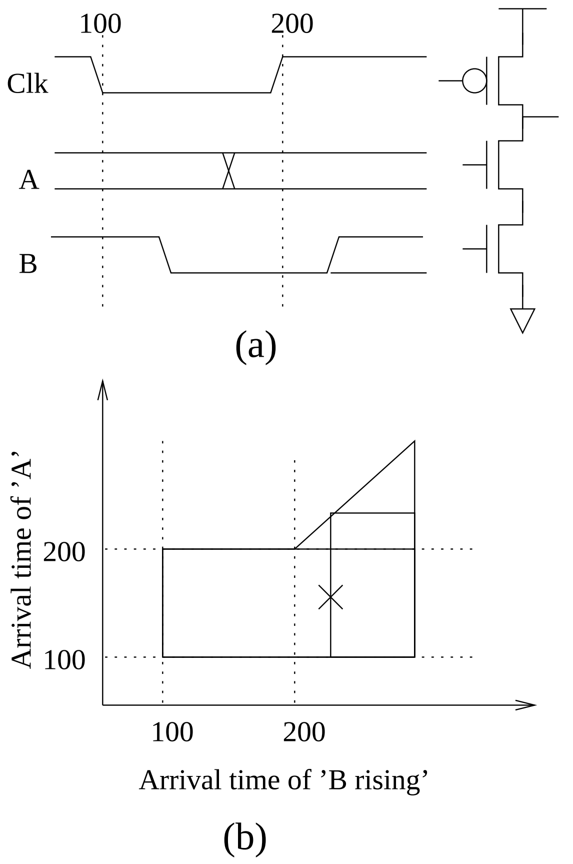

Figure 4: Dynamic Stage Example

```
Grow( $M, Input, Edge$ )
  $cycle \leftarrow M.C.\phi_1 + M.C.\phi_2$

  $B_l \leftarrow M.IW(Input).Edge.max$
  $B_u \leftarrow M.IW(Input).Edge.min + cycle$
  while( $B_u - B_l >$ Tolerance )
    $M.IW(Input).Edge.max \leftarrow \frac{(B_u+B_l)}{2}$
    $\langle advance, retard, pass \rangle \leftarrow CheckModel(M)$
    if( $pass =$ true )
      $B_l \leftarrow M.IW(Input).Edge.max + retard$
    else
      $B_u \leftarrow M.IW(Input).Edge.max - advance$
  $M.IW(Input).Edge.max \leftarrow B_l$

  $B_u \leftarrow M.IW(Input).Edge.min$
  $B_l \leftarrow M.IW(Input).Edge.max - cycle$
  while( $B_u - B_l >$ Tolerance )
    $M.IW(Input).Edge.min \leftarrow \frac{(B_u+B_l)}{2}$
    $\langle advance, retard, pass \rangle \leftarrow CheckModel(M)$
    if( $pass =$ true )
      $B_u \leftarrow M.IW(Input).Edge.min - advance$
    else
      $B_l \leftarrow M.IW(Input).Edge.min + retard$
  $M.IW(Input).Edge.max \leftarrow B_u$

ExpandModel( $M$ )
  $\forall i \in M.I$
    Grow( $M, i,$ rise)
    Grow( $M, i,$ fall)
```

Figure 5: Model Generation

mine the max time, we initialize the upper and lower bounds (B_u and B_l respectively) to search over the full clock cycle after the nominal max arrival time. We then iteratively shrink the bounds until they are within a prespecified tolerance value. We can further shrink the bounds after any simulation by using the equivalence window information. The min time calculation follows similarly.

By using equivalence windows, we can often substantially reduce the remaining bounds. Take, for example, an input which arrives nearly a full cycle before it is sampled. In this case, the equivalence window will reflect that the signal can be delayed nearly a full cycle without impacting functionality. This can potentially eliminate several iterations, each of which can be fairly expensive. Even if the gain from using this information were slight, or only occasionally useful, we might as well use it since it requires little extra effort to compute.

3.4 Limitations

This approach has some limitations that should be mentioned. First, before the iteration process can begin, the designer must supply a set of nominal input arrival times that produce correct functional behavior. In this way, we can start from a known-correct point in the input arrival space and search outwards until failure. While this is a constraint that is not present in static timing analysis, it should not be difficult to satisfy in practice. Since we envision designers using STS as a design and debug aid during the implementation process, a nominal set of input arrival times should be readily available at the time model-generation is begun.

The second limitation of our approach is that, while non-synchronous circuit design styles can exist internally, the block interface must be synchronous. This limitation is imposed primarily by the need to integrate with static timing analysis. The timing models that we generate produce setups and hold-times relative to reference clocks that exist only in synchronous systems. This limitation still allows for application to a much broader range of circuits than static timing analysis, since only the interface is constrained.

Lastly, since we are generating black-box models, we are unable to capture the timing behavior of transparency. Our approach will still safely model transparent latches, but it will do so in a conservative manner by forcing a setup check and setting the output arrival time relative to the reference clock. Unfortunately, since our current model does not incorporate combinational arcs, we can only adequately model state-holding circuitry. However, static timing analysis can easily handle blocks containing only static combinational circuits and would be preferred for its efficiency.

4 Results

Runtime results for our iterative model generation approach are shown in Table 1. All results were compiled on a 300 MHz Ul-

Table 1: Model Generation Runtimes

Name	FETs	Search Dimensions	Iterations Required	Runtime
simple	21	2	13	6 sec
s27	93	8	59	1 min
s208	582	22	166	32 min
s298	1092	6	44	33 min
s400	1397	6	52	1.5 hr
s820	2959	36	289	13.3 hr
adder4-l	190	4	12	23 sec
adder32-l	1478	4	14	1.1 hr

trasparc, using SirSim[6] and a Perl implementation of the model-generation layer. For these experiments, the Elmore delay calculator was used. High accuracy TETA-based delay calculation, as discussed in [7], incur runtime multiples of 2-50x depending on circuit topologies.

The first test-case, simple, contains a single input, for which the min and max arrival times were found. The number of iterations shown (13), are representative of the work required for each input on the larger simulations. The next 5 testcases are from the ISCAS benchmark suite. Transistor level networks for these cases were generated using nominally sized static logic gates, with the flip-flops implemented as pulsed transparent latches. The final two testcases are 4- and 32-bit dynamic Manchester carry-chain adders followed by transparent latches.

Since we use an iterative search technique, runtimes are affected solely by the number of iterations required and the time for each iteration. The iteration time is itself heavily dependent on the circuit topology, and the efficiency with which its internal Boolean functions can be represented. The iteration count is most dependent on the number of dimensions to the search, and to a lesser degree, the complexity of interactions between input signals. For the ISCAS benchmarks, the number of dimensions is always twice the number of inputs, one each for the min and max arrival times. The adders on the other hand, require only 4 search dimensions, regardless of their width, since the individual bits of each operand bus can be grouped together.

In general, we found the use of equivalence-window information often substantially reduced the number of iterations required. For the 4-bit adder, we required 32 iterations with simple binary search, and only 12 using equivalence windows. For the 32-bit adder, the count was similarly reduced from 34 to 14 runs. In each case, the overall runtime was cut by nearly 60%.

5 Conclusion

We have described a methodology by which black-box timing models can be generated using symbolic timing simulation. This capability integrates symbolic timing simulation into existing static analysis verification flows, and allows full-chip static timing analysis to include automatically generated models for aggressive full-custom circuit styles.

References

[1] M.A. Cirit. Characterizing a VLSI Standard-Cell Library. *Proceedings of the IEEE Custom Integrated Circuits Conference*, page 756, May 1991.

[2] K. Dioury, A. Greiner, and M.M. Louerat. Hierarchical Static Timing Analysis for CMOS ULSI Circuits. *TAU: ACM/IEEE International Workshop on Timing Issues in the Specification and Synthesis of Digital Systems*, pages 65–70, March 1999.

[3] R. B. Hitchcock, G. L. Smith, and D. D. Cheng. Timing Analysis of Computer Hardware. *IBM Journal of Research and Development*, 26(1):100–105, Jan 1982.

[4] J.S. Jephson, R.P. McQuarrie, and R.E. Vogelsberg. A Three-Value Computer Design Verification System. *IBM Systems Journal*, 3:178–188, 1969.

[5] C. B. McDonald. *Symbolic Functional and Timing Verification of Transistor-Level Circuits*. PhD thesis, Carnegie Mellon University, May 2001.

[6] C. B. McDonald and R. E. Bryant. CMOS Circuit Verification with Symbolic Switch-Level Timing Simulation . *IEEE Transactions on Computer Aided Design of Integrated Circuits and Systems*, March 2001.

[7] C. B. McDonald and R. E. Bryant. Computing Logic Stage Delays Using Circuit Simulation and Symbolic Elmore Analysis. *Proceedings of the Design Automation Conference*, June 2001.

[8] C. B. McDonald, T. Indermaur, and M. Buckley. The HP PA-8000 Timing Analysis Methodology. *Proceedings of Design SuperCon*, pages S232 1–22, January 1997.

[9] S. Napper. Static Timing Analysis - A Demanding Solution. *Electronic Engineering*, pages 35–42, January 1996.

[10] D. Patel. Characterization and Modeling System for Accurate Delay Prediction of ASIC Designs. *Proceedings of the IEEE Custom Integrated Circuits Conference*, May 1989.

[11] V. Rao, J. Soreff, T. Brodnax, and R. Mains. EinsTLT: Transistor Level Timing with EinsTimer. *TAU: ACM/IEEE International Workshop on Timing Issues in the Specification and Synthesis of Digital Systems*, pages 1–6, March 1999.

[12] D. Vo. Automated Spice Characterization of Gate Array and Standard Cell Libraries. *Proceedings of the IEEE Custom Integrated Circuits Conference*, pages 363–366, May 1987.

[13] Hakan Yalcin, Mohammad Mortazavif, Robert Palermo, Cyrus Bamji, and Karem Sakallah. Functional Timing Analysis for IP Characterization. *Proceedings of the Design Automation Conference*, pages 731–736, June 1999.