Efficient Computation of the Exact Worst-Delay Corner

Luis Guerra e Silva
Cadence Labs / INESC-ID
Dept. of ISCE - IST/TU Lisbon
Lisbon, Portugal
lgs@inesc-id.pt

Joel R. Phillips
Cadence Berkeley Labs
Cadence Design Systems, Inc
San Jose, CA 95134, U.S.A.
jrp@cadence.com

L. Miguel Silveira
Cadence Labs / INESC-ID
Dept. of ECE - IST/TU Lisbon
Lisbon, Portugal
lms@inesc-id.pt

ABSTRACT

Timing analysis and verification is a critical stage in digital IC design. As feature sizes decrease to nanometer scale, the impact of process parameter variations in circuit performance becomes extremely relevant. Notwithstanding the advances in statistical timing analysis as a form of incorporating variability effects in the timing verification flow, corner analysis is still the standard timing signoff methodology for any industrial design. Since it is impossible to analyze a design for all the process corners, due to the exponential size of the corner space, the design is usually analyzed for a set of carefully chosen corners, that are expected to cover all the worst-case scenarios. However, there is no established systematic methodology for picking the right worst-case corners. This paper addresses this problem by proposing an efficient automated methodology for computing the exact worst-delay process corners of a digital IC, given a linear parametric characterization of the gate and interconnect delays. The key aspect of our methodology is the use of branch-and-bound techniques that enable an effective pruning of the timing graph, significantly reducing the number of relevant paths that require detailed analysis.

1. INTRODUCTION

As integrated circuit feature sizes decrease, the impact of process and operational parameter variations in circuit performance and reliability becomes very significant [7]. Under these circunstances, proper timing of the circuit is now considerably harder to predict and ensure. The ability to accurately identify the parameter settings that correspond to critical timing conditions is therefore increasingly important. In a traditional timing verification flow a design is analyzed for a small set of carefully chosen parameter settings, designated as *corners*, usually corresponding to extreme conditions in the process space. Such corners are chosen based on the knowledge of designers and process engineers. Unfortunately, picking the right corners in a realistic manner is not trivial and most often than not either the actual worst settings are missed or gross over-design may happen. Compounding the problem, for nanometric feature sizes, the number of relevant parameters increases significantly, making it necessary to develop design and analysis models that account for those variations directly. In this context, the ability to determine the exact worst corner of the circuit becomes an issue of increasing importance, as it enables actual correction or optimization of a design.

Statistical static timing analysis (SSTA) has been introduced as a form of incorporating variability effects in traditional timing verification. Even though several promising SSTA modeling techniques have been proposed [4, 1, 9], their practical applicability is still quite limited, as their usage could ultimately entail an overhaul of the timing verification flow [5]. Industrial tools and design flows are not yet prepared to handle statistical information, and EDA companies are still evaluating the best ways to incorporate it in their products and design flows, as it does not represent a natural extension to the current flows. Further, SSTA requires complex parameter characterization, like multidimensional statistical distributions, and most foundries still do not provide that information on their fabrication technologies in a consistent manner, either due to confidentiality issues, or simply because they are still trying to figure out themselves the best way to represent and convey such information. In fact, process control steps can be better described by ranges than peaked distributions.

Even though SSTA is not yet a mature methodology, mainly due to poor parameter characterization and lack of tool support, the *parametric* delay and slew formulations that it prescribes, can be employed in automating the complex selection of the worst corners, to be used in traditional corner-based analysis. This paper proposes an efficient methodology for computing the *exact* worst-delay corners of a digital integrated circuit, given a linear parametric characterization of the cell and interconnect delays. The key aspect of our methodology is the utilization of branch-and-bound techniques that enable an effective *pruning* of the timing graph, thus significantly reducing the number of relevant paths requiring detailed analysis. Parameters only need to be characterized by their respective value ranges, as opposite to SSTA where they need to be characterized by statistical distributions. Additionally, this approach produces more insightful information for the designer, like worst-delay corners and specific paths where they induce critical timing conditions, making it much more useful in guiding manual or automated design optimization than other approaches.

Recently, [6] proposed a linear-time approach for timing analysis that computes a delay upper bound, covering all process corners. Experimental evidence shows that for the limited set of ISCAS combinational benchmarks the delay upper bound computed by this approach is close to the delay computed by exhaustive corner analysis. However, no matter how tight, this estimate is just an approximation. Futher, the worst-case corner for the delay upper bound may not be the true worst-delay corner of the circuit. In fact, the worst-delay corner may correspond to a completely different setting of parameters. The inability to compute the exact worst-delay corner, somehow reduces the usefulness of this technique as it becomes difficult to infer which paths or areas of the circuit are critical or need to be fixed. The goal of our work is quite different as we target the determination of the *exact* worst-delay corner and associated paths.

This paper is organized as follows. Section 2 introduces a few basic concepts and notation. Section 3 formulates the worst-delay corner problem and discusses possible exhaustive solutions. Section 4 proposes a solution using branch-and-bound techniques. Section 5 discusses application of the former techniques to the computation of worst-slack corners. Section 6 discusses the tightness of bounds and its effect on performance. Finally, Section 7 presents the experimental results and Section 8 presents some concluding remarks.

2. BACKGROUND

The timing information of a circuit is usually modeled by a *timing graph*, where vertices correspond to pins in the circuit, and edges to pin-to-pin delays in cells or interconnect. The timing graph is an acyclic graph, $G = (V, E)$, composed of *vertices*, $v \in V$, and directed *edges*, $e \in E$, connecting them. Each edge is annotated with the corresponding *delay*. The *primary inputs* are vertices with no incoming edges. All vertices with no outgoing edges are *primary outputs*. The sets of primary inputs and outputs of G are respectively $PI(G)$ and $PO(G)$. A *complete path* is a sequence of edges, connecting a primary input to a primary output, and will be referred to simply as a *path*. A *partial path* is a sequence of edges connecting any two vertices.

In this work, instead of assuming delays to be constant real-numbered values, we assume them to be described by affine functions [8] of process/operational parameter variations, corresponding to a first-order linearization of every delay, d, around a nominal point, λ_0, in the parameter space,

$$d(\lambda-\lambda_0) = d(\lambda_0)+\left.\frac{\partial d}{\partial \lambda}\right|_{\lambda_0}(\lambda-\lambda_0) = d(\lambda_0)+\left.\frac{\partial d}{\partial \lambda}\right|_{\lambda_0}\Delta\lambda \quad (1)$$

where $\Delta\lambda = \lambda - \lambda_0$, represents the incremental parameter variation vector. Considering the parameter space to have size p, then Eqn. (1) can be rewritten more compactly as

$$d(\Delta\lambda) = d_0 + \sum_{i=1}^{p} d_i \Delta\lambda_i = d_0 + d^T \Delta\lambda \quad (2)$$

where d_0 is the nominal value of d, computed at the nominal values of the parameters, λ_i, $i = 1, 2, \ldots, p$, and d_i is the sensitivity of d to parameter λ_i, computed at the nominal point λ_0. It is out of the scope of this paper to discuss the delay computation procedure [3]. Therefore, in the following, we will assume the timing information of any circuit to be available in the form of an annotated timing graph.

Two main approaches have been proposed for timing analysis: block-based and path-based. In the block-based approach, characterized by linear runtime, arrival times are pushed through the timing graph in a levelized fashion, performing sum operations with delays over the edges and min/max operations over the vertices with multiple incoming edges. The alternative path-based approach consists in individually computing the delay of each path in the circuit by adding the delay of each of its edges. Even though more accurate, this approach is computationally much more expensive than the former, since the number of paths can grow exponentially with the number of vertices (pins).

The min/max operations required by block-based timing analysis are trivial for constant real-valued delays, but more complex when delays are affine function of process parameters, as assumed here. However, the min/max of affine functions is a piecewise-affine function, and the min/max of piecewise-affine functions is also a piecewise-affine function. Similarly, the sum of affine functions is an affine function, and the sum of a piecewise-affine function and an affine function is a piecewise-affine function. Therefore, any arrival time can be exactly represented by a piecewise-affine function, since it is the result of a sequence of min/max and sum operations between piecewise-affine functions and affine function. If no simplification is performed, the piecewise-affine representation of arrival times should grow linearly with the number of paths, and therefore can be exponential in the number of vertices.

An important property of affine functions is their *convexity* [2]. Both the min/max and sum operator produce convex functions when operating on convex functions. The convexity implies that *the smallest/largest value for a given affine or piecewise-affine function is obtained by setting each variable to one of its extreme values.* In the context of timing analysis this corresponds to state that the smallest/largest delay or arrival time is obtained by setting each parameter to one of its extreme values. For the simple case of delays, that are represented by affine functions, this value is fairly easy to compute. Assuming that $\Delta\lambda_i \in \left[\Delta\lambda_i^{min}, \Delta\lambda_i^{max}\right]$, if in Eqn. (2) we set to $\Delta\lambda_i^{max}$ all the parameter variations with positive sensitivities, and to $\Delta\lambda_i^{min}$ the remaining ones, we are maximizing the value of the affine delay function over the parameter space, therefore obtaining the maximum value,

$$\max_{\Delta\lambda}[d(\Delta\lambda)] = d(\Delta\lambda^*) = d_0 + \sum_{i=1}^{p} d_i \Delta\lambda_i^* \quad (3)$$

where the maximizing parameter variation assignment is

$$\Delta\lambda_i^* = \begin{cases} \Delta\lambda_i^{max} & if \;\; d_i \leq 0 \\ \Delta\lambda_i^{min} & if \;\; d_i > 0 \end{cases}, \; i = 1, 2, ..., p \quad (4)$$

The min can be computed in a similar way. For affine functions this computation takes linear time in the number of parameters, however, for piecewise-affine functions this computation is much more expensive, since it requires an implicit or explicit enumeration of all the 2^p possible solutions (corners), which makes it exponential in the number of parameters. Throughout this paper, and without loss of generality, we will assume that all the parametric formulas have been normalized such that $\Delta\lambda^{max} = 1$ and $\Delta\lambda^{min} = 0$.

3. WORST-DELAY CORNER

3.1 Problem Formulation

In the following we will consider the timing graph of a combinational block with n inputs and m outputs. Assuming that delays are affine functions of the process parameters, in the form of Eqn. (2), then any delay, $d_{i,j}(\Delta\lambda)$, from an input i to an output j is a piecewise-affine function.

The *worst-delay corner* (WDC) problem, consists in computing an assignment, $\Delta\lambda^*$, to the parameter variation vector, $\Delta\lambda$, that produces the worst delay, $d_{i,j}(\Delta\lambda)$, from any input $i = 1, \ldots, n$ to any output $j = 1, \ldots, m$.

In *late* mode, the worst delay is the *largest* delay. Assuming that $d_{i,j}^{late}(\Delta\lambda)$ is the piecewise-affine function of the delay in late mode from input i to output j, then the WDC problem is formulated as

$$\max_{\Delta\lambda}\left\{\max_{j=1,\ldots,m}\left[\max_{i=1,\ldots,n} d_{i,j}^{late}(\Delta\lambda)\right]\right\} \quad (5)$$

Conversely, in *early* mode the worst delay is the *smallest* one, therefore the WDC problem is formulated as

$$\min_{\Delta\lambda}\left\{\min_{j=1,\ldots,m}\left[\min_{i=1,\ldots,n} d_{i,j}^{early}(\Delta\lambda)\right]\right\} \quad (6)$$

As we have seen, since arrival times are represented by piecewise-affine functions which are convex, their worst (min or max) value is obtained by setting each parameter variation to one of its extreme values. Therefore, in essence this problem can be cast as a combinatorial optimization problem where, by searching in a finite but typically large set of

elements, we want to optimize a given cost function. In this case the set of elements can be the set of all the 2^p possible parameter variation assignments, and the cost function is the arrival time at a given primary output. The major difficulty with this type of discrete problems, as opposed to continuous linear problems, is that we do not have any optimality conditions to check if a given (feasible) solution is optimal or not. Therefore, in order to conclude that a feasible solution is optimal, we must somehow compare its cost with the cost of all other feasible solutions. This amounts to always explore the entire solution space, either *explicitly* or *implicitly*, by a complete or partial *enumeration* of all the feasible solutions and their associated costs.

3.2 Exhaustive Methods

The simplest exhaustive algorithm that can be conceived for computing the WDC is to evaluate the delay of the circuit for all the 2^p possible parameter variation assignments, and verify which assignment produces the worst arrival time at a primary output. That assignment clearly corresponds to the WDC. Arrival times can be computed in linear time using a block-based timing analysis procedure. However, since such procedure must be executed for each of the 2^p parameter variation assignments, the overall algorithm will be exponential in the number of parameters.

The WDC can instead be computed by searching the path space, rather than the parameter space. Essentially, this corresponds to performing an exhaustive path-based timing analysis and, for each path, compute the corresponding affine delay function, by adding the delay functions of its edges. Given the affine delay function of a path, its WDC can easily be computed by applying Eqns. (3) and (4). The WDC of the circuit is the WDC of the critical path. Computing the delay function and the corresponding WDC of a path has linear complexity in the number of parameters. However, since such computation must be performed for every path, and the number of paths can grow exponentially with the number of vertices, the overall procedure can exhibit exponential complexity in the number of vertices.

Clearly, both exhaustive methods exhibit exponential run time complexity, either in the number of parameters or in the number of vertices. For very small circuits, or in situations where a small number of parameters is of interest, they may constitute viable options. However, even average size circuits will render both approaches unpractical.

4. DYNAMIC PRUNING

In this section we propose an approach for computing the *exact* WDC using branch-and-bound techniques, that enable dynamic pruning of parts of the search space, avoiding an explicit enumeration of all the possible solutions. We start by briefly explaining the basic foundations of branch-and-bound techniques and subsequently present path-space and parameter-space search algorithms based on them.

4.1 Branch-and-Bound

Most combinatorial problems, including the one at hand, can only be solved by explicitly or implicitly evaluating a specific, nonlinear, cost function over the entire solution space, in order to compute the solution that yields the optimal cost. Branch-and-bound techniques focus on pruning useless regions of the solution space, thus avoiding the explicit enumeration and evaluation of all the possible solutions that they may contain. During the execution of the algorithm, the best known value for the cost function is maintained, corresponding to the cost of the best solution already found. If by some simple and quick procedure we are able to determine that the cost of all the solutions contained in a certain subspace is worse than the best known cost, then it is useless to explore that subspace, since no improvement on the cost function will be obtained. Therefore, that portion of the solution space can be pruned, and an explicit enumeration of all the solutions it may contain is avoided. Even though in the worst case this approach can be as bad as the exhaustive enumeration if no prunning occurs, on average, for a wide range of applications, it has proven to perform significantly better.

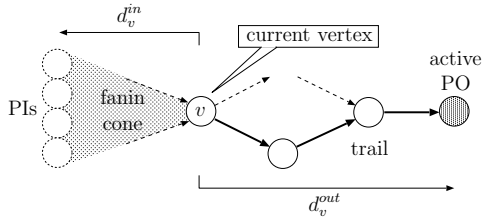


Figure 1: Illustration of delay estimates.

4.2 Path Space Search

In the following we detail a branch-and-bound based algorithm that computes the WDC by finding one path where it occurs. Without loss of generality, but for the sake of simplicity, the following discussion will assume late mode operation (e.g. worst delay = largest delay). Considering a primary output at a time, the algorithm performs an analysis of all complete paths that end at that primary output, that we will designate as the *active* primary output. For most paths, that analysis will be *implicit* as they will get pruned out therefore not requiring exhaustive analysis, and hopefully only a few will require an *explicit* analysis. To accomplish this, the timing graph is traversed in a backward fashion, starting at the active primary output, going through the internal vertices, and eventually ending at the primary inputs (if no pruning is performed). The vertex being visited in a given step is designated by *current vertex*. The path taken to reach that vertex from the active primary output is designated by *trail*. If reconvergent fanouts exist, the same vertex can be reached from the same primary output, through distinct trails. The worst delay, w^*, found among the complete paths already analyzed is continuosly updated, as well as the corresponding corner, $\Delta\lambda^*$. For each current vertex of the timing graph, v, the algorithm relies on three parametric delay estimates:

- d_v^{in} is an upper bound on the delay from any primary input to vertex v (e.g. in the fanin cone of v);
- d_v^{out} is the *exact* delay of the trail;
- $d_v^{path} = d_v^{in} + d_v^{out}$, which represents an upper bound on the delay of any path going through v, that contains the trail.

The affine expression of d_v^{in} is calculated beforehand, through a forward-traversal, block-based analysis of the timing graph, using a technique similar to the one proposed in [6], but with looser bounds, and therefore faster to compute. An illustration of these estimates is presented in Figure 1.

The rationale underlying the proposed algorithm is that, if for a given vertex v, and assuming late mode, the following condition is verified,

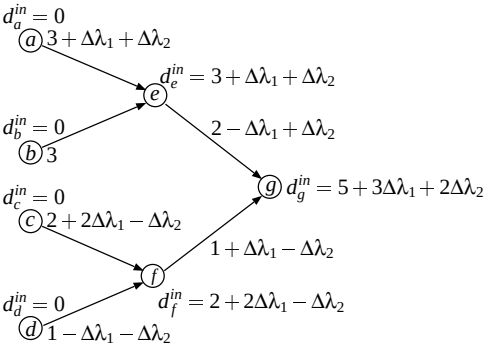
$$\max_{\Delta\lambda}[d_v^{path}] \leq w^* \tag{7}$$

then the fanin cone of v can be pruned. What Eqn (7) says is that in this case the delay of *any* path going through v

```
1: function PROCESS-VERTEX(G, v, w*, d_v^out)
2:     d_v^in ← IN-DELAY-ESTIMATE(v)
3:     d_v^path ← d_v^in + d_v^out
4:     ⟨w, Δλ⟩ ← max_Δλ[d_v^path]
5:     if w ≤ w* then                    ▷ Fanin cone gets pruned
6:         return ⟨w*, 0⟩
7:     else if v ∈ PI(G) then
8:         return ⟨w, Δλ⟩                ▷ Worst delay is updated
9:     else
10:        for all e ← INCOMING-EDGES(v) do
11:            s ← SOURCE-VERTEX(e)      ▷ Get source vertex
12:            d_e ← DELAY(e)
13:            d_s^out ← d_v^out + d_e
14:            ⟨w, Δλ⟩ ← PROCESS-VERTEX(G, s, w*, d_s^out)
15:            if w > w* then
16:                ⟨w*, Δλ*⟩ ← ⟨w, Δλ⟩
17:            end if
18:        end for
19:        return ⟨w*, Δλ*⟩
20:    end if
21: end function
```

and containing the trail can never be larger than the largest delay found so far, w^*. Therefore it is useless to further explore the fanin cone of v and it can be pruned. In essence this allows us to avoid analyzing all paths that originate within that cone, thus impliciting pruning many paths at once. It should be noted that this pruning is only valid for a specific trail. If v is subsequently visited through another trail its fanin cone may not be pruned (at that point a new condition is verified, corresponding the the trail used). If a given current vertex is not pruned, all its fanin vertices are subsequently visited. When the current vertex is a primary input, the trail is a complete path, and d_v^{out} is its affine delay function. If the worst delay of this path is larger than the worst delay w^* found so far, then w^* is updated. The procedure is complete when every path of the circuit is either explicitly visited or pruned.

The proposed algorithm iterates through every primary output and then chooses the corner that produces the worst delay among all primary outputs. An alternative technique is to connect all the primary outputs to a super-sink vertex through edges of delay 0, and just analyze the super-sink as the only primary output. In every step of the algorithm (refer to the pseudo-code at the top of the page), the function PROCESS-VERTEX processes the current vertex, v. It either prunes the fanin cone of v and returns or invokes itself over each of its fanin vertices. For a given current vertex, v, it starts by adding d_v^{in} and d_v^{out} to obtain d_v^{path}, that is an upper bound on the delay of any complete path that contains v and the trail. As we have mentioned d_v^{in} was previously computed and annotated in v. d_v^{out} can be computed by adding the delay of all the edges in the trail. This value is computed in the calling instance of PROCESS-VERTEX and passed as an argument. As d_v^{in} and d_v^{out}, d_v^{path} is an affine function of the parameter variations. The worst value for d_v^{path}, w, and the corresponding corner, $\Delta\lambda$, are subsequently computed using Eqns. (3) and (4). If $w \leq w^*$, that means that the worst-delay path cannot contain the trail, because w is an upper bound on the delay of any path containing the trail. Therefore we stop the traversal at this vertex, which corresponds to prune its fanin cone. If $w > w*$, and v is a primary input, it means the trail is a complete path and its worst delay is larger than the largest known delay computed so far, and therefore the latter should be updated. If v is not a primary input then the delay estimate is just a conservative bound, and therefore it cannot be used to update the largest know delay. We proceed until all the paths in the circuit are either explicitly explored or pruned. At the end, the largest known delay w^* and the corresponding parameter variation settings, $\Delta\lambda^*$, are the worst delay of the circuit and the worst-delay corner, respectively.



v	trail	d_v^{path}	w	$\Delta\lambda$	w^*	$\Delta\lambda^*$
g	$\emptyset$	$5 + 3\Delta\lambda_1 + 2\Delta\lambda_2$	10	$\langle 1, 1\rangle$	0	$\emptyset$
e	$\langle e, g\rangle$	$5 + 2\Delta\lambda_2$	7	$\langle X, 1\rangle$	0	$\emptyset$
a	$\langle a, e\rangle, \langle e, g\rangle$	$5 + 2\Delta\lambda_2$	7	$\langle X, 1\rangle$	7	$\langle X, 1\rangle$
b	$\langle b, e\rangle, \langle e, g\rangle$	$5 - \Delta\lambda_1 + \Delta\lambda_2$	6	$\langle 0, 1\rangle$	7	$\langle X, 1\rangle$
f	$\langle f, g\rangle$	$3 + 3\Delta\lambda_1 - 2\Delta\lambda_2$	6	$\langle 1, 0\rangle$	7	$\langle X, 1\rangle$

Figure 2: Example of path space search.

Figure 2 illustrates the execution of the algorithm for a small timing graph. It should be noticed that w^* is only updated when vertex a is analyzed because only then the trail is a complete path, and therefore d_v^{path} is the exact delay of that path, and not an upper bound. Further, the fanin cone of f is not analyzed because $w \leq w^*$. This corresponds to pruning a portion of the path space, namely paths $\{\langle c, f\rangle, \langle f, g\rangle\}$ and $\{\langle d, f\rangle, \langle f, g\rangle\}$.

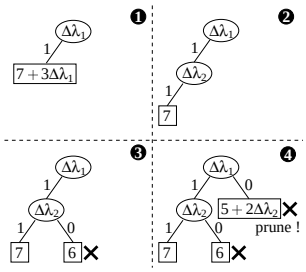
4.3 Parameter Space Search

Similar branch-and-bound techniques can be employed when the search is conducted in the parameter space. By analyzing the worst delay obtained for specific corners of the parameter space it is possible to effectively prune regions of such space, e.g. exclude other corners, without having to explicitly compute their associated circuit delay.

We assume corners to be either completely or partially specified. *Completely* specified corners have a value assigned to each element of the parameter variation vector. For such corners the delay of a given circuit can be exactly computed. In *partially* specified corners, the value of at least one of the elements of the parameter variation vector is unknown. For such corners only an upper bound on the delay of a given circuit can be computed, assuming that all the unknown elements of the parameter variation vector assume their worst values. A partially specified corner implicitly represents a group of completely specified corners. For example, $\langle 1, 0, X\rangle$ implicitly represents $\langle 1, 0, 0\rangle$ and $\langle 1, 0, 1\rangle$.

The underlying principle of this algorithm is that if the delay upper bound, w, produced by a given completely or partially specified corner, $\Delta\lambda$, is not better than the delay, w^*, produced by some completely specified corner already analyzed, $\Delta\lambda^*$, then the corresponding corners cannot produce a delay that is larger than w^* and should therefore be pruned. No advantage is obtained when the pruning is performed on a completely specified corner. However, when a partially specified corner is pruned, all the completely specified corners that it implicitly represents are also pruned and will not be explored any further.

The algorithm explores the parameter space by either analyzing or pruning every corner. When a corner is analyzed a delay estimate of the circuit is computed. In order to keep track of all the partially or completely specified corners that have already been either analyzed or pruned, we use a *de-*



$\Delta\lambda_1$	$\Delta\lambda_2$	delay
0	X	$5+2\Delta\lambda_2$
1	X	$6+2\Delta\lambda_2$
X	0	$5+3\Delta\lambda_1$
X	1	$7+3\Delta\lambda_1$
0	0	5
0	1	7
1	0	6
1	1	7

X = unspecified

Figure 3: Example of parameter space search.

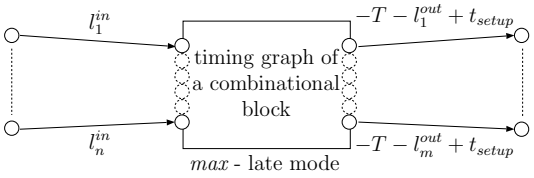


Figure 4: Timing graph to model setup constraints.

cision tree. Each node in the decision tree represents one, parameter variation, and can have at most a left and right child, which are the subtrees where the parameter variation assumes each of its extreme values. The leaves are delay estimates. Figure 3 illustrates the evolution of the decision tree during the execution of the algorithm for the timing graph in Figure 2. Note that in step (4) corner $\langle 0, X\rangle$ is pruned because its worst delay is 7 (for $\Delta\lambda_2 = 1$), which does not represent an improvement over the worst delay produced by the corner $\langle 1, 1\rangle$, previously analyzed.

5. WORST-SLACK CORNER

In this section we generalize the methodology for computing the WDC proposed in Section 4, to enable the computation of worst-slack corners. We present this generalization by illustrating the computation of worst-slack corners induced by setup and hold constraints in sequential circuits. Slacks induced by other constraints can be handled similarly.

Sequential circuits consist of combinational blocks interleaved by registers, usually implemented with flip-flops. Typically they are composed of several stages, where a register captures data from the primary outputs of a combinational block and injects it into the primary inputs of the combinational block in the next stage. Register operation is synchronized by clock signals generated by one or multiple clock sources. Clock signals that reach distinct flip-flops (sinks in the clock tree) are delayed by a certain *clock latency*.

Within a clock period T, we assume that data is injected into a combinational block by a register of n flip-flops with parametric clock latencies $l_1^{in}(\Delta\lambda), \ldots, l_n^{in}(\Delta\lambda)$, respectively, and captured by a register of m flip-flops with parametric clock latencies $l_1^{out}(\Delta\lambda), \ldots, l_m^{out}(\Delta\lambda)$, respectively. If the clock network is a tree, which is a common situation, then large portions of the net are shared among multiple paths. In this case, it is feasible to use a very accurate method (even perform electrical level simulation), to compute good estimates of the clock latencies, using a path-based approach in a very efficient way.

5.1 Setup Time and Late Mode

Proper operation of a flip-flop requires that the input data line must be stable for a specific period of time before the capturing clock edge. This period of time is designated by *setup time*, and we will represent it by t_{setup}.

Let us consider the case where a flip-flop, with clock latency l_i^{in}, connected to the i-th primary input of the combinational block, is injecting data, and another flip-flop, with clock latency l_j^{out}, connected to the j-th primary output of the combinational block, is capturing the result, Assuming the delay in the combinational block, from the i-th primary input to the j-th primary output in *late* mode to be $d_{i,j}^{late}$, then the setup time in the capturing flip-flop is observed only if the following condition holds,

$$l_i^{in} + d_{i,j}^{late} \leq T + l_j^{out} - t_{setup} \tag{8}$$

This condition must hold for every $\langle i, j\rangle$ input/output flip-flop pair. For a given output flip-flop j this set of constraints can be compactly written as

$$\max_{i=1,\ldots,n}(l_i^{in} + d_{i,j}^{late}) \leq T + l_j^{out} - t_{setup} \tag{9}$$

This expression induces a slack, s_j^{setup}, defined as,

$$s_j^{setup} = T + l_j^{out} - t_{setup} - \max_{i=1,\ldots,n}(l_i^{in} + d_{i,j}^{late}) \tag{10}$$

that is non-negative when the conditions are met and negative otherwise. The worst-slack corner for s_j^{setup} is the corner where its value is minimized, formally

$$\min_{\Delta\lambda}(s_j^{setup}) = -\max_{\Delta\lambda}(-s_j^{setup}) \tag{11}$$

Ignoring the sign and expanding s_j^{setup} we obtain

$$\max_{\Delta\lambda}(-s_j^{setup}) = \max_{\Delta\lambda}\left[\max_{i=1,\ldots,n}(l_i^{in} + d_{i,j}^{late}) - T - l_j^{out} + t_{setup}\right] \tag{12}$$

The corner, $\Delta\lambda^*$, that maximizes the value of $-s_j^{setup}$ among all outputs $j = 1, \ldots, m$ is given by

$$\max_{\Delta\lambda}\left\{\max_{j=1,\ldots,m}\left[\max_{i=1,\ldots,n}(l_i^{in} + d_{i,j}^{late}) - T - l_j^{out} + t_{setup}\right]\right\} \tag{13}$$

Comparing Eqns. (13) and (5) we can easily conclude that the worst setup slack corner problem can be cast to the WDC problem, if the original timing graph of the combinational block is modified as illustrated in Figure 4.

5.2 Hold Time and Early Mode

For a correct operation of a flip-flop, the input data line must be stable for a certain period of time after the capturing clock edge. This period of time is designated by *hold time*, and we will represent it by t_{hold}. Assuming the same connectivity as before, and the delay in *early* mode to be $d_{i,j}^{early}$, the hold time in the capturing flip-flop is observed only if the following condition holds,

$$l_i^{in} + d_{i,j}^{early} \geq l_j^{out} + t_{hold} \tag{14}$$

Following the same steps as before, we obtain the worst-slack corner for s_j^{hold},

$$\min_{\Delta\lambda}(s_j^{hold}) = \min_{\Delta\lambda}\left[\min_{i=1,\ldots,n}(l_i^{in} + d_{i,j}^{early}) - l_j^{out} - t_{hold}\right] \tag{15}$$

The corner, $\Delta\lambda^*$, that minimizes the value of s_j^{hold} among all outputs $j = 1, \ldots, m$ is given by

$$\min_{\Delta\lambda}\left\{\min_{j=1,\ldots,m}\left[\min_{i=1,\ldots,n}(l_i^{in} + d_{i,j}^{early}) - l_j^{out} - t_{hold}\right]\right\} \tag{16}$$

As before, by comparing Eqns. (16) and (6) we can easily conclude that the worst hold slack corner problem can be cast to the WDC problem, formulated in Section 3, if the original timing graph of the combinational block is modified as illustrated in Figure 5.

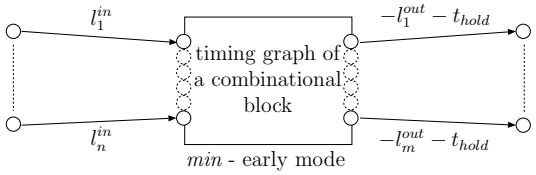


Figure 5: Timing graph to model hold constraints.

5.3 Multi-Cycle Paths

Multi-cycle paths are paths between registers where the output register captures the result more than one clock cycle after the input register has injected the data. For a multi-cycle path with cycle $c \in \mathbb{N}$ $(c > 1)$, Eqs. (8) and (14) can be rewritten to,

$$l_i^{in} + d_{i,j}^{late} \leq c \times T + l_j^{out} - t_{setup} \quad (17)$$

$$l_i^{in} + d_{i,j}^{early} \geq (c-1) \times T + l_j^{out} + t_{hold} \quad (18)$$

The cycle c is usually a user-specified timing constraint.

6. BOUND COMPUTATION

The computation of arrival time and delay upper/lower bounds is a pervasive task in the algorithms presented in previous sections. Their performance is highly dependent on the *tightness* of those bounds. Tighter bounds are more expensive to compute, but potentially allow for larger regions of the search space to be pruned, and therefore may have a significant impact in performance. The bound computation consists in a min/max operation between two or more parametric formulas. Given two parametric formulas a and b, the cheapest upper bound on their max, $c \geq \max[a, b]$, can be computed by picking each coefficient to be the max of the coefficients of a and b. Formally,

$$c_i = \max[a_i, b_i] \ , \ \ i = 0, 1, \ldots, p \quad (19)$$

This bound is very cheap to compute, but it is also loose. The tightest upper bound on the max, with only one bounding function c, can be computed by solving the LP,

$$\begin{array}{ll} \min & \epsilon \\ \text{s.t.} & \epsilon \geq c(\Delta\lambda^{(q)}) - \max[a(\Delta\lambda^{(q)}), b(\Delta\lambda^{(q)})],\ q = 1, \ldots, 2^p \\ & c(\Delta\lambda^{(q)}) \geq \max[a(\Delta\lambda^{(q)}), b(\Delta\lambda^{(q)})] \\ & 0 \leq \Delta\lambda_l \leq 1,\ l = 1, \ldots, p \end{array} \quad (20)$$

where $\Delta\lambda^{(q)}$ is the q-th process corner. This bound is the tightest (for one plane), but it is expensive to compute.

7. EXPERIMENTAL RESULTS

A realistic circuit block was synthesized and mapped to an industrial 90nm technology. As process parameters, we considered the widths and thicknesses of the six metal layers needed to route the block. During parasitic extraction of the design, we computed the nominal values and sensitivities of each parasitic element (resistors and grounded capacitors), relative to each one of the 12 parameters. From the circuit block we extracted 3 combinational circuits that we used as benchmarks. Table 1 presents useful information about the timing graph of each circuit, including the number of process parameters considered.

Table 2 presents the CPU time and the search size for the execution of the exhaustive (Exh) and branch-and-bound versions of the WDC computation by searching in the path and parameter spaces. The path space branch-and-bound versions were executed using both loose (BnB-L) and tight (BnB-T) bounds, as detailed in Section 6. Table 3 presents the CPU time and search size for the execution of the exhaustive (Exh) and branch-and-bound (BnB-L) versions of the worst-slack corner computation for hold constraints.

By analyzing the experimental results in Table 2 it is easy to conclude that the branch-and-bound technique with loose bounds is very effective, since it reduces the CPU times and search size by several orders of magnitude, both when searching the parameter space and the path space. We can also conclude that the parameter space search is significantly more expensive than path space search. Additionally, as expected, when tighter bounds are used the amount of search is slightly reduced, since more pruning should occur, which indicates potential for some moderate improvement, if tighter, but still cheap, bounds can be computed. Since the tight bounds are expensive to compute, in this case the additional pruning is not noticeable in the CPU time. Observing the results in Table 3 we can conclude that, as expected, the branch-and-bound techniques are also effective in computation of the worst-slack corner.

Name	#Vertex	#Edge	#PI	#PO	#Par
mult	2507	3324	20	19	12
add	679	890	41	22	12
share	375	493	26	13	12

Table 1: Benchmark information.

	Name	Parameter		Path		
		Exh	B-n-B	Exh	BnB-L	BnB-T
CPU Time (s)	mult	356.52	10.94	2.68	0.02	141.20
	add	27.54	0.2	0.01	<0.01	22.21
	shared	9.52	0.05	0.01	<0.01	11.69
Search Size	mult	4096	125	3249498	1623	1170
	add	4096	27	9144	595	466
	shared	4096	19	3846	52	52

Table 2: Worst-delay corner computation.

	Name	Path	
		Exh	BnB-L
CPU Time (s)	mult	3.08	0.01
	add	0.01	<0.01
	shared	0.01	<0.01
Search Size	mult	3803713	29
	add	11552	56
	shared	4859	120

Table 3: Worst-slack corner computation (hold).

8. CONCLUSIONS

This paper proposes an efficient, branch-and-bound based, automated methodology for computing the exact worst-delay process corners of a digital integrated circuit, given a linear parametric characterization of the gate and interconnect delays. Experimental evidence shows that the proposed approach is particularly effective, leading to reductions in CPU time up to several orders of magnitude, when computing circuit timing while accounting for parameter variability.

ACKNOWLEDGMENTS

The authors want to thank Andreas Kuehlmann, Christoph Albrecht, Vinod Kariat and Igor Keller for discussions.

9. REFERENCES

[1] S. Bhardwaj, S. B. K. Vrudhula, and D. Blaauw. Tau: Timing analysis under uncertainty. In *Proceedings of ICCAD*, pages 615–620, San Jose, CA, November 2003.

[2] S. Boyd and L. Vandenberghe. *Convex Optimization.* Cambridge University Press, 2004.

[3] L. G. e Silva, Z. Zhu, J. Phillips, and L. M. Silveira. Variation-Aware, Library Compatible Delay Modeling Strategy. In *Proceedings of the IFIP VLSI-SoC Conference*, Nice, France, October 2006.

[4] J.-J. Liou, K.-T. Cheng, S. Kundu, and A. Krstic. Fast Statistical Timing Analysis by Probabilistic Event Propagation. In *Proceedings of DAC*, pages 661–666, Las Vegas, NV, June 2001.

[5] F. N. Najm. On the Need for Statistical Timing Analysis. In *Proceedings of DAC*, pages 764–765, Anaheim, CA, June 2005.

[6] S. Onaissi and F. N. Najm. A Linear-Time Approach for Static Timing Analysis Covering All Process Corners. In *Proceedings of ICCAD*, San Jose, CA, November 2006.

[7] L. Scheffer. Explicit Computation of Performance as a Function of Process Variation. In *International Workshop on Timing Issues in the Specification and Synthesis of Digital Systems*, Monterey, CA, December 2002.

[8] J. Stolfi and L. H. de Figueiredo. Self-Validated Numerical Methods and Applications. In *Operations Research*, July 1997.

[9] C. Visweswariah, K. Ravindran, K. Kalafala, S. G. Walker, and S. Narayan. First-order incremental block-based statistical timing analysis. In *Proceedings of DAC*, pages 331–336, San Diego, CA, June 2004.