Efficient Block-Based Parameterized Timing Analysis Covering All Potentially Critical Paths

Khaled R. Heloue, Sari Onaissi, and Farid N. Najm, *Fellow, IEEE*

***Abstract*—In order for the results of timing analysis to be useful, they must provide insight and guidance on how the circuit may be improved so as to fix any reported timing problems. A limitation of many recent variability-aware timing analysis techniques is that, while they report delay distributions, or verify multiple corners, they do not provide the required guidance for re-design. We propose an efficient block-based parameterized timing analysis technique that can accurately capture circuit delay at every point in the parameter space, by reporting all paths that can become critical. Using an efficient pruning algorithm, only those *potentially critical* paths are carried forward, while all other paths are discarded during propagation. This allows one to examine local robustness to parameters in different regions of the parameter space, not by considering differential sensitivity at a point (that would be useless in this context) but by knowledge of the paths that can become critical at nearby points in parameter space. We give a formal definition of this problem and propose a technique for solving it, which improves on the state of the art, both in terms of theoretical computational complexity and in terms of runtime on various test circuits.**

***Index Terms*—Hyperplane, parameterized timing analysis, piece-wise planar (PWP), PVT variations, required arrival times.**

I. Introduction

SIGNAL and clock path delays in integrated circuits are subject to variations arising from many sources, including (manufacturing) process variations, (supply/ground) voltage variations, and temperature variations. These are collectively referred to as PVT variations. During design, one accounts for the delay variability by either "padding" the path delays with a *timing margin* so that the chip would yield well at all process corners in spite of the variations (application-specific integrated circuits approach), or by "binning" the resulting chips at different frequencies (microprocessors). While this is not a new problem, the scale of the problem has increased recently, because: 1) an increasing number of circuit parameters have significant variability, causing an increase in the number of corners; and 2) within-die variations are becoming more significant, and they cannot be handled by the traditional corner-based approach. The variables or parameters under study are of two types: many transistor and metal line parameters are directly related to underlying statistical process variables, so they may be modeled as *random variables*, with certain distributions; on the other hand, the supply/ground voltage and temperature are not random, and must be modeled as simply unknown or *uncertain variables*, within known bounds.

Manuscript received April 12, 2011; revised July 14, 2011; accepted September 6, 2011. Date of current version March 21, 2012. This paper was recommended by Associate Editor S. Vrudhula.

K. R. Heloue is with the Technology Group, AMD, Markham, ON L3T7X6, Canada (e-mail: khaled@eecg.utoronto.ca).

S. Onaissi is with the Design and Technology Solutions Division, Intel, Santa Clara, CA 95052 USA (e-mail: sari@eecg.utoronto.ca).

F. N. Najm is with the Department of Electrical and Computer Engineering, University of Toronto, Toronto, ON M5S3G4, Canada (e-mail: f.najm@utoronto.ca).

Color versions of one or more of the figures in this paper are available online at http://ieeexplore.ieee.org.

Digital Object Identifier 10.1109/TCAD.2011.2175392

Given the two types of variables under study, two types of solution techniques have emerged: statistical static timing analysis (SSTA) and multicorner static timing analysis (MCSTA). SSTA models parameters as random variables, assuming that their distributions and correlations are known *a priori* [1]–[3], and provides the distribution of circuit delay, from which the timing yield can be estimated. On the other hand, MCSTA models the PVT parameters as uncertain variables, within given bounds, and attempts to verify the timing at all corners in a single timing run [4]. All these techniques consider the circuit delay to be dependent on a number of PVT parameters, be they random or uncertain. Therefore, one can describe the required overall solution to this problem as parameterized static timing analysis (PSTA).

The motivation for this paper is the simple notion that in order for the results of timing analysis to be useful they must provide guidance on how the circuit may be improved so as to fix any reported timing problems. To understand the need for PSTA in general, consider the simple case where delay is linear in the variational (PVT) parameters. In a circuit, the delay of any input-output path becomes a linear expression in terms of the parameters, or what we refer to as a *hyperplane*. At the nominal PVT point, the hyperplane corresponding to the path with the largest delay (under nominal conditions) is *dominant* (over all others). As we move around in PVT space, some other path may become critical, and, correspondingly, another hyperplane may become dominant. Overall, across the whole PVT space, the total circuit delay follows some piece-wise planar (PWP) surface. This surface is defined by all the hyperplanes that can become dominant at *some* point in PVT space. We refer to these hyperplanes as *potentially dominant* and to their corresponding paths as *potentially critical*.

Suppose we are at some operating point in PVT space, and we are interested in the *robustness* of the circuit at that point. In other words, we are interested in the impact of variations on overall circuit delay around that point. What information would be useful to the designer in this case?

One could consider providing the *sensitivity* of delay, at that point, to the various PVT parameters, such as by means of the partial derivatives of delay to each of the parameters. However, because of the PWP nature of the delay surface, such point metrics are actually useless. One may find the derivatives to have low values at that point, yet one may be very close to a "break point" in the surface where another hyperplane with much larger sensitivities suddenly becomes dominant. Instead, one must be able to quickly discover what paths (i.e., hyperplanes) become dominant in a certain neighborhood around the point of interest. Given a list of problematic paths in the neighborhood, when working on fixing some path, one avoids being "blind-sighted" to the criticality of other paths. Thus, in order for the results of timing analysis to be useful, we believe that the whole PWP surface is required. It is not enough to give the user the worst-case corner; that does not provide a full picture of what needs to be fixed. Also, simply providing the timing yield, as is done in SSTA, or simply providing a list of a large number of paths, with a failure probability for each, does not give sufficient insight into what paths need to be fixed around the operating point. Instead, a PWP surface (for the total circuit delay) allows one to examine the local neighborhood in order to see which parameters and paths may be problematic (so that one can focus on them as part of redesign). It should be mentioned that the "broken" nature of the delay surface is not due to the linearity assumption. Instead, it is actually due to the *max* function that is implicit in the problem of timing verification of setup constraints (a similarly broken delay surface results from the *min* function in a similar problem of verifying hold constraints). If one assumes a nonlinear, say polynomial, delay dependence, one simply ends up with a piece-wise polynomial surface, which presents the same sort of problems.

In order to faithfully represent the PWP surface for the total circuit delay, we must include (during propagation in the timing graph) all the hyperplanes that can become dominant *somewhere* in PVT space. Simply carrying along *all* paths can be problematic due to possible path count explosion; hence, an efficient *pruning strategy* is needed, whereby redundant paths that cannot become dominant *anywhere* in PVT space are identified and *pruned* during the propagation. This problem was studied in [5], where an exact pruning algorithm and a sufficient condition for pruning were proposed, and where it was found that indeed the number of potentially dominant paths is manageable and does not explode. In that work, the exact algorithm (as we will see) has time complexity $\mathcal{O}(p^2n^2)$, where p is the number of PVT parameters and n is the number of hyperplanes to be pruned, and the sufficient condition is $\mathcal{O}(pn^2)$. In this paper, we propose: 1) a more efficient exact solution to the pruning problem that takes $\mathcal{O}(p^2mn)$ time, where m is the number of potentially dominant hyperplanes at the circuit outputs; and 2) a sufficient condition for pruning that is $\mathcal{O}(pn)$. We will see that the resulting improvements in runtime can be significant for *hard* circuits.

The rest of this paper is organized as follows. In Section II, we review some basic terminology and formally describe the pruning problem. We also describe the pruning techniques presented in [5], and assess the complexity of their exact pruning algorithm. In Section III, we transform the pruning problem from the parameterized timing domain to the domain of computational geometry, and show how it relates to two standard problems in that field. In Section IV, we present our exact pruning algorithm and sufficient condition for pruning, and study the complexity of these pruning strategies. We provide test results and comparisons to previous work in Section V. After that, Section VI introduces the concept of partial critical surfaces, and Section VII gives an extension of our method where these are propagated in the timing graph instead of complete critical surfaces. Finally, we give our concluding remarks in Section VIII.

A preliminary version of this paper appeared in [6], which presented a sufficient condition method and an exact method for finding all potentially critical paths, or the critical surfaces, at the outputs of a circuit. In this paper, we present an extension that allows one to find only the potentially critical paths, which violate timing constraints, instead of all potentially critical paths, as given in Sections VI and VII. The hyperplane delays of such paths form what we refer to as partial critical surfaces, and these are cheaper to find than the full critical surfaces. This provides designers with a timing analysis method that is faster than the state-of-the-art method of [6], but which is guaranteed to find only potentially critical paths that violate timing constraints. Because timing analysis is preformed repeatedly at various steps of the design flow, the runtime gains achieved by this extension can result in a large overall speedup. For example, circuit timing/power optimization is typically performed by iteratively introducing changes to the circuit and then running static timing analysis. In this case, one would be able to use the method as presented in [6] sparingly and to rely more heavily on the extended method presented here. Because of the runtime gains that this would provide, the extension given in Sections VI and VII makes this paper more practical and applicable in industry.

In addition to this extension, a different set of circuits was used to generate the results of this paper than in [6]. This set combines some of the circuits of the ISCAS-85 benchmark suite used in [6] with other circuits from the more recent ITC-99 benchmark circuits. These test circuits were mapped using a different cell library and more heavily optimized using commercial tools than were those in [6]. As a result, the numbers of potentially critical paths (hyperplanes) reported for the ISCAS-85 circuits are less than those seen for the same circuits in [6]. This and other improvements in the code also led to improved runtimes over those reported in [6].

II. Background

In this section, we first review some basic terminology covering timing modeling and propagation. Then, we describe the problem formulated by the authors of [5] and briefly review their approach.

A. Modeling and Propagation

In block-based timing analysis, timing quantities are propagated in the timing graph in topological order, through a sequence of basic operations, such as *add* operations on input arrival times and arc delays, and *max* operations on the timing quantities resulting from those additions. In this way, the output arrival time is determined and is then propagated to subsequent stages. This is shown in Fig. 1, which shows an AND gate and its corresponding timing graph. The nets A, B, and C are represented as nodes in the timing graph whereas the timing arcs between the inputs of the gate and its output are represented by edges with delays D_1 and D_2. Here, the arrival time D_C at the output of the AND gate is computed as the *max* of the sums of arrival times and their corresponding edge delays. In other words

$$D_C = \max\left(D_A + D_1, D_B + D_2\right) \tag{1}$$

where D_A and D_B are the signal arrival times at the nodes A and B, and D_1 and D_2 are the timing arc delays. This can be easily generalized to gates with more than two inputs.

Since variability in the process and environmental (PVT) parameters affects transistor performance, gate delays should be represented in such a way to highlight their dependence on these underlying parameters. First-order linear delay models have been extensively used in the literature, and they generally capture this dependence well. In this paper, we assume that gate delay is a linear function of process and environmental parameters, such as channel length L, threshold voltage V_t, supply voltage V_{dd}, and temperature T. These parameters are assumed to vary in specified ranges; however, without loss of generality, we can easily normalize these ranges to $[-1, 1]$, similarly to what was done in [4]. Hence, gate delay D can be expressed as follows:

$$D = d_o + \sum_{i=1}^{p} d_i X_i \quad -1 \leq X_i \leq 1 \quad \forall i \tag{2}$$

where d_o is the nominal delay, X_is are the normalized PVT parameters, and d_is are the delay sensitivities to these parameters. Since D is a linear function of p parameters, it is referred to as a delay *hyperplane*. The linear dependence of delays on process parameters is not too strong an assumption, and it has been widely adopted in the context of SSTA (e.g., in [2]). Even if the dependence of gate delay on process parameters is not strictly linear, one can still apply our method by first constructing a linear expression that is an upper bound on whatever nonlinear surface one may have for describing the true dependence of delay on these parameters, and then use that linear expression in place of the true delay in our approach. The variables X_i in (2) can correspond to any meaningful process parameters that one cares about. Some may represent die-to-die variations while others might represent within-die variations. In this paper, we will simply refer to the X_is as process parameters, without regard to exactly what type of parameters they are. However, in our results we only include die-to-die variations. With-in die variations would be handled by including a separate variable for each gate/region. This may result in longer delay expressions (and a runtime penalty), and the accuracy of the results depends on how these variables are chosen by the user. More work is required to fully apply our approach in that context, and this remains a possible future application of our work.

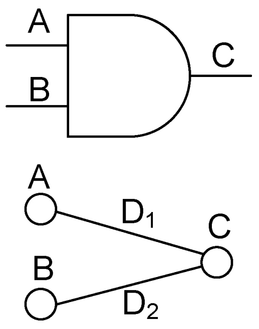

Fig. 1. Propagation for a single gate.

The delay of a path is simply the *sum* of arc delays of all gates on that path. Since arc delays are expressed as hyperplanes, so will be the path delay; in the rest of this paper, when we refer to the delay of a path, it is understood that we mean *path delay hyperplane*. Although this is true for a path, the arrival time at a node, which is the *max* of all path delay hyperplanes in the fan-in cone of that node, is not necessarily a hyperplane. This is shown in Fig. 2, where four paths, P_1–P_4, converge at a node. The arrival time, A, at that node is given by

$$A = \max(P_1, P_2, P_3, P_4). \tag{3}$$

Shown as the broken dashed line, A is a PWP surface because either P_1, P_2, or P_3 can become the maximum (or dominant) hyperplane, depending on which region of the parameter space is under consideration. Note that P_4 is always *covered* by another hyperplane, and therefore does not show up in the PWP surface. Paths, such as P_1–P_3, which can become dominant are referred to as *potentially critical* or *nonredundant* paths, whereas paths, such as P_4, which cannot become critical are referred to as *redundant* or *prunable* paths. We will formally define these terms in the next section. Ideally, during analysis, only those potentially critical paths (or nonredundant hyperplanes) must be propagated to subsequent stages, while all other hyperplanes must be discarded or pruned.

B. Pruning Problem

Let D_j be the delay hyperplane of path j in a set of n paths converging on a node, so that D_j is given by

$$D_j = a_{oj} + \sum_{i=1}^{p} a_{ij} X_i \quad j = 1, \ldots, n. \tag{4}$$

The hyperplane D_j is said to be redundant or prunable if and only if

$$\max(D_1, \ldots, D_n) = \max(D_1, \ldots, D_{j-1}, D_{j+1}, \ldots, D_n). \quad \forall X_i \tag{5}$$

In this case, no matter where we are in the parameter space, D_j will never show up as the maximum hyperplane, as other hyperplanes will be dominating it. An example of this is path P_4 in Fig. 2; such a redundant hyperplane can be pruned from the set without affecting the shape of the PWP surface representing the max. On the other hand, if (5) is not satisfied, then D_j is a nonredundant hyperplane and must be kept in the

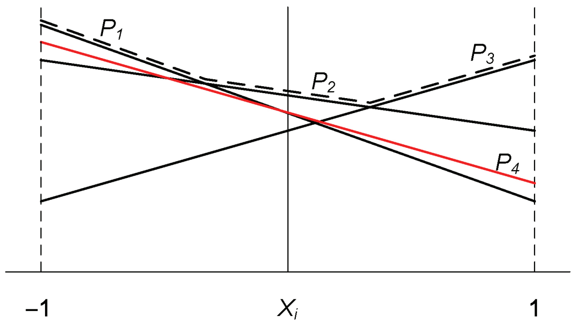

Fig. 2. MAX of path delay hyperplanes.

set. An example of this are paths P_1–P_3 that show up in the PWP surface.

Formally, the pruning problem can be stated as follows. Given a set $\mathcal{P}$ of n hyperplanes D_j, find the set $\mathcal{Q} \subseteq \mathcal{P}$, such that $\mathcal{Q}$ is an irreducible set of m nonredundant hyperplanes $\tilde{D}_j$, where $m \leq n$, and such that

$$\max(D_1, \ldots, D_n) = \max(\tilde{D}_1, \ldots, \tilde{D}_m) \quad \forall X_i. \tag{6}$$

Only those m nonredundant hyperplanes are needed to describe the shape of the PWP surface defined by the max. This pruning problem was studied by the authors of [5], who proposed two techniques for pruning. We now review these techniques and describe some of their limitations.

1) *Pairwise Pruning:* The first technique is based on pairwise comparisons between hyperplanes, to check if any hyperplane can prune another hyperplane, as follows. Let D_1 and D_2 be two hyperplanes

$$D_1 = a_{o1} + \sum_{i=1}^{p} a_{i1} X_i \tag{7}$$

$$D_2 = a_{o2} + \sum_{i=1}^{p} a_{i2} X_i. \tag{8}$$

If $D_1 - D_2 \leq 0$ for all values of X_i, then D_1 is pruned by D_2, denoted by $D_1 \prec D_2$. Since $-1 \leq X_i \leq 1$, then $D_1 \prec D_2$ if and only if

$$a_{o1} - a_{o2} + \sum_{i=1}^{p} |a_{i1} - a_{i2}| \leq 0 \tag{9}$$

which can be easily checked.

The pairwise pruning procedure [5] is shown in Algorithm 1, where we have preserved the same flow as in [5] for clarity. It has two nested loops that cover all pairs of hyperplanes, checking if $D_j \prec D_i$. Note that this algorithm is only a *sufficient condition* for pruning and is not an exact solution for the pruning problem. In fact, the resulting set $\mathcal{Q}$ is not necessarily an irreducible set. Going back to Fig. 2, PAIRWISE will fail to identify P_4 as a redundant hyperplane since $P_1, \cdots, P_4$ are pairwise nonprunable. In addition, the complexity of PAIRWISE is $\mathcal{O}(pn^2)$, where p is the number of PVT parameters and n is the number of hyperplanes. This quadratic complexity can be problematic, particularly if a large number of redundant hyperplanes, which are identified as nonredundant by PAIRWISE, are propagated to subsequent stages, potentially causing a blowup in the number of hyperplanes, as reported by [5] on one of the test circuits.

Algorithm 1 PAIRWISE

```
Input: $\mathcal{P} = \{D_1, \ldots, D_n\}$;
Output: $\mathcal{Q} \supseteq \{D_1, \ldots, D_m\}$;
1: Mark all hyperplanes in $\mathcal{P}$ as nonredundant;
2: for $i = 1 : n$ do
3:     if ($D_i$ is marked redundant) then
4:         continue;
5:     end if
6:     for $j = 1 : n$ do
7:         if ($D_j$ is marked redundant) then
8:             continue;
9:         end if
10:        if ($D_j \prec D_i$) then
11:            Mark $D_j$ as redundant;
12:        end if
13:    end for
14: end for
15: Add all nonredundant hyperplanes to $\mathcal{Q}$;
```

Algorithm 2 FEASCHK

```
Input: $\mathcal{P} = \{D_1, \ldots, D_n\}$;
Output: $\mathcal{Q} = \{D_1, \ldots, D_m\}$;
1: Mark all hyperplanes in $\mathcal{P}$ as nonredundant;
2: for $j = 1 : n$ do
3: Formulate (10) for $D_j$ excluding redundant hyperplanes;
4:     if (feasible) then
5:         Add $D_j$ to $\mathcal{Q}$;
6:     else
7:         Mark $D_j$ as redundant;
8:     end if
9: end for
```

2) *Feasibility Check:* The second pruning technique is a *necessary and sufficient* condition for pruning. It is therefore an exact solution for the pruning problem, which guarantees that the resulting set $\mathcal{Q}$ is an irreducible set of nonredundant hyperplanes. The idea is to perform a *feasibility check* for every hyperplane D_j by searching for a point in the space of X_is where D_j is dominant over *all* other hyperplanes. If this is feasible, then D_j is nonredundant, otherwise, D_j is redundant and can be pruned from the set. Thus, D_j is nonredundant if and only if the following system of inequalities has a feasible solution:

$$\begin{aligned} D_j \geq D_k \quad & k = 1, \ldots, n, \; k \neq j \\ -1 \leq X_i \leq 1 \quad & i = 1, \ldots, p. \end{aligned} \tag{10}$$

Algorithm 2 describes FEASCHK, where a feasibility check is formulated for every hyperplane in the starting set $\mathcal{P}$ (line 6). If there is a feasible solution, then the hyperplane is nonredundant and is added to $\mathcal{Q}$. Otherwise, it is marked as redundant and is pruned from the set.

Note that the feasibility check in (10) consists of solving a linear program (LP) with p variables and $(n + p)$ constraints, which has a complexity of $\mathcal{O}\left(p^2(n + p)\right)$, if an interior-point-based LP solver is used [7]. Therefore, the complexity of FEASCHK, which requires n feasibility checks to determine the irreducible set of nonredundant hyperplanes, is $\mathcal{O}\left(p^2(n+p)n\right)$, which is $\mathcal{O}(p^2n^2)$ if $p \leq n$ that is usually the case. Given that this pruning algorithm would potentially be applied at every node in the timing graph, its $\mathcal{O}(n^2)$ behavior in the number of hyperplanes can be expensive.

In the following sections, we present a more efficient method for solving the pruning problem. By transforming this problem into a standard problem in *computational geometry*, we present an exact pruning algorithm that is $\mathcal{O}(p^2mn)$, where n is the number of hyperplanes in the initial set $\mathcal{P}$, and m is the number of nonredundant hyperplanes in the final irreducible set $\mathcal{Q}$. We also propose a sufficient condition for pruning that can be used as a pre-processing step, and which is $\mathcal{O}(pn)$.

III. Problem Transformation

In this section, we show how we map our parameterized timing pruning problem into a standard problem in computational geometry.

A. From Computational Geometry

The field of computational geometry deals with the study of algorithms to solve problems stated in terms of geometry. Typical problems include Convex Hull, Vertex/Facet enumeration, and Voronoi diagrams, to name a few [8]. We have identified two standard problems that can be related to the pruning problem: enumeration of extreme points of a convex hull and its equivalent (dual) problem of minimal representation of a polytope. We first review these problems and show how the pruning problem can be transformed into a standard problem.

1) *Extreme Points Enumeration:* To understand this problem, let us start by defining the following terms.

Definition 1 (Convex Hull): The convex hull of a set P of n points, denoted as $conv(P)$, is the smallest convex set that contains these points.

Definition 2 (Extreme Points): Given a set P of n points in d dimensions, the minimal subset E of P for which $conv(P) = conv(E)$ is called the set of extreme points. In other words, if point $e \in E$, then $conv(P \setminus \{e\}) \neq conv(P)$.

The extreme points enumeration problem can be stated as follows. Given a set P of n points, determine the minimal subset E of m extreme points, where $m \leq n$. This is shown graphically in Fig. 3(a), where the shaded region is the convex hull and points 1–4 are the set of extreme points. Note that points 5 and 6 do not contribute to the convex hull and can thus be removed.

2) *Minimal Polytope Representation:* We begin by defining some terms that will help us introduce this standard problem.

Definition 3 (Hyperplane): It is the set $\{x \mid a^T x = b\}$, where $a \in \mathbb{R}^d$, $a \neq 0$ and $b \in \mathbb{R}$. It is the solution set of a nontrivial linear equation among the components of x. A hyperplane divides $\mathbb{R}^d$ into two half-spaces.

Definition 4 (Half-Space): It is the set $\{x \mid a^T x \leq b\}$, where $a \in \mathbb{R}^d$, $a \neq 0$, and $b \in \mathbb{R}$. It is the solution set of one nontrivial linear inequality.

Definition 5 (Polyhedron/Polytope): A polyhedron is the set $P \subseteq \mathbb{R}^d$, such that

$$P = \{x \mid a_j^T x \leq b_j, \quad j = 1, \ldots, n\}. \tag{11}$$

It is, therefore, the intersection of a finite number of half-spaces. A *bounded* polyhedron is called a polytope. A polyhedron/polytope can be written in matrix form as follows:

$$P = \{x \mid Ax \leq b\} \tag{12}$$

where A is an $n \times d$ matrix, and $b \in \mathbb{R}^n$. Note that A is not necessarily the minimal representation of P.

Definition 6 (Supporting Hyperplane): If one of the two closed half-spaces of a hyperplane h contains a polytope P, then h is called a supporting hyperplane of P. Note that every row in the matrix representation of the polytope P corresponds to a supporting hyperplane.

For example, the shaded region in Fig. 3(b) is a polytope defined as the intersection of six half-spaces, each bounded by a hyperplane, and all six hyperplanes are *supporting hyperplanes*.

Definition 7 (Bounding Hyperplane): A hyperplane that is spanned by its intersection with a polytope P is called a bounding hyperplane of P. Those rows in the matrix representation of P, which can be satisfied with equality for some values of x, correspond to bounding hyperplanes of P.

For example, in Fig. 3(b), only hyperplanes 1–4 are *bounding hyperplanes*; they appear at the boundary of the polytope.

With the above definitions, the problem of minimal polytope representation can be stated as follows. Given a polytope P with n supporting hyperplanes, find all m bounding hyperplanes of the polytope, where $m \leq n$. This will correspond to the minimal representation of P. In other words, if P is defined as $Ax \leq b$, where A is an $n \times d$ matrix, and $b \in \mathbb{R}^n$, find a reduced $\tilde{A}$ and $\tilde{b}$ such that

$$P = \{x \mid Ax \leq b\} = \{x \mid \tilde{A}x \leq \tilde{b}\} \tag{13}$$

where $\tilde{A}$ and $\tilde{b}$ are the rows of A and b that correspond to the m bounding hyperplanes of P. Referring again to the example in Fig. 3(b), if hyperplanes 5 and 6 were removed, it would not affect the shape of the polytope.

The above two problems have obvious similarities; in fact, these two problems are equivalent, as one is the dual of the other. This can be explained by the the *point-hyperplane* duality in computational geometry [8]. Point-hyperplane duality is a common transformation whereby a point p at distance r from the origin O is associated with the hyperplane normal to Op at distance $1/r$ from the origin. Under this transformation, extreme points enumeration and minimal polytope representation are two equivalent problems. This is shown in Fig. 3, where the extreme points 1–4 of the convex hull are transformed to the bounding hyperplanes 1–4 of the polytope; also, points 5 and 6 on the interior of the convex hull are transformed to hyperplanes 5 and 6, which do not appear in the minimal polytope representation. Therefore, an algorithm that can solve one problem efficiently can also be used to solve the other problem, and vice versa. We have identified an efficient algorithm in [9], which solves extreme point enumeration.

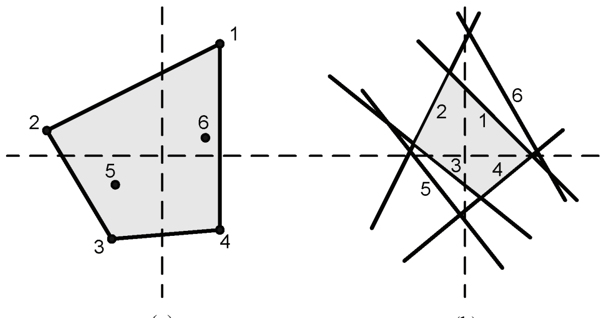

Fig. 3. (a) Extreme points of convex hull. (b) Minimal polytope representation (dual problem).

The same algorithm can be used to solve the dual problem of minimal polytope representation. In the next section, we show how the pruning problem, defined in Section II-B, can be transformed to a minimal polytope representation problem. Once this is established, we can adapt the algorithm in [9] to solve the pruning problem efficiently.

B. To Parameterized Timing

Recall the pruning problem, where given a set of n delay hyperplanes D_j, we want to determine every hyperplane that can become the maximum hyperplane, for some setting of the PVT parameters. These hyperplanes are referred to as nonredundant, whereas other hyperplanes that cannot become dominant are referred to as redundant hyperplanes, and should be pruned. Let $D_{\max}$ be the PWP maximum of all n hyperplanes, i.e., $D_{\max} = \max(D_1, \cdots, D_n)$, $\forall X_i$, such as the broken line in Fig. 2. As a result, the following condition holds:

$$D_{\max} \geq D_j = a_{oj} + \sum_{i=1}^{p} a_{ij} X_i \quad -1 \leq X_i \leq 1 \quad \forall i \ \forall j. \tag{14}$$

Note that if D_j is a nonredundant hyperplane, then the above inequality will be satisfied with equality, i.e., $D_{\max} = D_j$, when D_j becomes the maximum hyperplane for some setting of X_is. Otherwise, if D_j is redundant, then $D_{\max} > D_j$ for all X_is.

By rearranging (14) so as to include $D_{\max}$ in the parameters, we get the following:

$$\sum_{i=1}^{p} a_{ij} X_i - D_{\max} \leq -a_{oj} \quad j = 1, \ldots, n. \tag{15}$$

Let $x = [X_1 \ X_2 \ \cdots \ X_p \ D_{\max}]^T$, $h_j = [a_{1j} \ a_{2j} \ \cdots \ a_{pj} \ -1]^T$, and $b_j = -a_{oj}$. Then, we can write the above inequality as

$$h_j^T x \leq b_j \quad j = 1, \ldots, n. \tag{16}$$

Finally, if $b = [b_1 \ b_2 \ \cdots \ b_n]^T$ and $H = [h_1 \ h_2 \ \cdots \ h_n]^T$. Then, we can write the above inequalities in a matrix form

$$Hx \leq b \quad -1 \leq X_i \leq 1. \tag{17}$$

This defines a polytope $\mathcal{H}$ in $p+1$ dimensions, where p is the number of PVT parameters. Now if we were to find the minimal representation of $\mathcal{H}$, this would result in determining all the rows that correspond to bounding hyperplanes, that is, the rows that can be satisfied with equality, as explained in Definition 7. If row j is a bounding hyperplane, then $h_j^T x = b_j$ is satisfied for some parameter setting. By rearranging this equality in terms of $D_{\max}$, we get $D_{\max} = D_j$, which is the condition for which a delay hyperplane is nonredundant in the pruning problem, as observed above. Therefore, determining the minimal representation of $\mathcal{H}$ would actually solve the pruning problem and determine the set of nonredundant delay hyperplanes.

IV. Pruning Algorithm

In this section, we present two pruning algorithms: 1) an exact solution to the pruning problem, which is a modified version of the algorithm in [9], and 2) a sufficient condition for pruning that is linear in the number of hyperplanes, and which can be used to speed up the pruning algorithm by reducing the number of calls of the exact algorithm during propagation in the timing graph.

A. Exact Algorithm

1) *Description:* In the minimal polytope representation problem, one needs to identify which rows of the defining polytope matrix correspond to bounding hyperplanes. Let $\mathcal{H}$ be a polytope defined by the system of inequalities $Hx \leq b$, and let $h_j^T x \leq b_j$ be a row in that system. In order to test whether $h_j^T x \leq b_j$ corresponds to a bounding hyperplane, we need to check if $h_j^T x = b_j$ is satisfied for some value of x; this can be tested using the following LP:

$$\begin{aligned} \text{maximize} \quad & v = h_j^T x \\ \text{suchthat} \quad & Hx \leq b. \end{aligned} \tag{18}$$

If the solution is $v^* < b_j$, then the hyperplane $h_j^T x = b_j$ is not a bounding hyperplane and can be removed (pruned) from the system; this means that other inequalities are acting in such a way that $h_j^T x \leq b_j$ is never "pushed to its boundary." Otherwise, if the solution is $v^* = b_j$, then $h_j^T x = b_j$ is a bounding hyperplane of the polytope and should be kept in the system.

The LP in (18) is formulated in Procedure 1, Check_Redund(). This procedure takes as inputs a set of delay hyperplanes $\mathcal{B}$, and a hyperplane $D \in \mathcal{B}$ that we are trying to prune. The delay hyperplanes are first transformed from the parameterized timing domain to the computational geometry domain, where a polytope $Hx \leq b$ is created. Then, $h_j^T x \leq b_j$ is checked to see if it corresponds to a bounding hyperplane of the polytope by formulating the LP in (18). If so, then pruned=FALSE, and D is nonredundant, otherwise, pruned=TRUE and D is redundant. The procedure also returns x^*, which is a *solution witness* generated by the LP solver, i.e., it is the value of x at which the LP maximum v^* is found. Recall that the complexity of an LP is linear in the number of constraints [7]. Therefore, the complexity of Check_Redund() is linear in the size of $\mathcal{B}$; specifically, it is $\mathcal{O}(p^2|\mathcal{B}|)$.

Algorithm 3 describes the exact pruning algorithm PRUNE, which uses Check_Redund(). PRUNE takes a set of delay hyperplanes $\mathcal{P}$ and determines the set $\mathcal{Q} \subseteq \mathcal{P}$ of nonredundant

Procedure 1 Check_Redund($\mathcal{D}$, $\mathcal{B}$)

```
Inputs: Hyperplane $\mathcal{D}$, and a set of hyperplanes $\mathcal{B}$ including $\mathcal{D}$;
Outputs: pruned = {TRUE, FALSE}, $x^*$ solution witness;
1: $D \Rightarrow h_j^T x \leq b_j$ and $\mathcal{B} \Rightarrow Hx \leq b$; {transform inputs from
   delay domain to polytope domain as described in
   section III-B}
2: Formulate the LP in (18) and get $x^*$ as solution witness;
3: if ($h_j^T x^* = b_j$ then
4:     pruned = FALSE;
5: else
6:     pruned = TRUE;
7: end if
8: return (pruned, $x^*$)
```

Algorithm 3 PRUNE

```
Input: Set of hyperplanes $\mathcal{P} = \{D_1, \ldots, D_n\}$ of size $n$;
Output: Set of all nonredundant hyperplanes $\mathcal{Q}$ of size $m \leq n$;
1: $\mathcal{Q}$ = Get_Initial_NR($\mathcal{P}$);
2: $\mathcal{P}' = \mathcal{P} \backslash \mathcal{Q}$;
3: repeat
4:     Let $D$ be the next hyperplane in $\mathcal{P}'$;
5:     Remove $\mathcal{D}, \mathcal{P}' = \mathcal{P}' \backslash \{\mathcal{D}\}$;
6:     [pruned, $x^*$] = Check_Redund($D$, $\mathcal{Q} \cup \{D\}$);
       {run a small LP}
7:     if (pruned = TRUE) then
8:         $D$ is redundant and is not added to $\mathcal{Q}$;
9:     else
10:        [pruned, $x^*$] = Check_Redund ($D$, $\mathcal{Q} \cup \mathcal{P}' \cup \{D\}$);
           {run a large LP}
11:        if (pruned = FALSE) then
12:            $D$ is nonredundant;
13:            Add it to set, $\mathcal{Q}, = \mathcal{Q} \cup \{D\}$;
14:        else
15:            $D$ is redundant and is not added to $\mathcal{Q}$;
16:            Use witness $x^*$ to get a set $\mathcal{W}$ of nonredundant
               hyperplanes containing $x^*$;
17:            Add $\mathcal{W}$ to set, $\mathcal{Q} = \mathcal{Q} \cup \mathcal{W}$;
18:        end if
19:    end if
20: until $\mathcal{P}' = \{\}$
```

hyperplanes. The algorithm starts by determining a small subset of nonredundant hyperplanes by calling a procedure Get_Initial_NR(), shown in Procedure 2. Get_Initial_NR() probes the delay hyperplanes at a predefined set of points in the parameter space X_i, in order to determine which delay hyperplane is maximum at every point. Those hyperplanes that show up as maximum hyperplanes are nonredundant, and are therefore added to the initial set. In addition to the nominal probing point, $X_i = 0\ \forall i$, $2p$ probes are chosen such that $X_j = \pm 1$, $X_i = 0\ \forall i \neq j$, $j = 1, \ldots, p$, which makes Get_Initial_NR() linear in the number of hyperplanes and the number of probes; specifically, it is $\mathcal{O}(pn)$.

Procedure 2 Get_Initial_NR ($\mathcal{P}$)

```
Input: Set of hyperplanes $\mathcal{P} = \{D_1, \ldots, D_n\}$;
Output: Subset $\mathcal{Q}$ of nonredundant hyperplanes;
1: $\mathcal{Q} = \{\}$;
2: Find $D_j$ with maximum nominal delay $a_{oj}$;
3: $\mathcal{Q} = \mathcal{Q} \cup \{D_j\}$;
4: Set $X_i$ to 0, $\forall_i$;
5: for i = 1:p do
6:     Set $X_i$ to 1;
7:     Find $D_j$ with maximum value at $X_i$;
8:     $\mathcal{Q} = \mathcal{Q} \cup \{D_j\}$;
9:     Set $X_i$ to $-1$;
10:    Find $D_j$ with maximum value at $X_i$;
11:    $\mathcal{Q} = \mathcal{Q} \cup \{D_j\}$;
11:    Reset $X_i$ to 0;
12: end for
```

Once this initial set of nonredundant hyperplanes is determined, PRUNE creates the set of remaining hyperplanes $\mathcal{P}'$ (line 2), and starts a loop until $\mathcal{P}'$ is empty (line 20). In every run of the loop, a hyperplane D is removed from $\mathcal{P}'$, and is first checked for redundancy against the set $\mathcal{Q}$ of nonredundant hyperplanes that were discovered so far, by calling Check_Redund() (line 6). If $\mathcal{Q}$ prunes D, then D is definitely pruned by the bigger set $\mathcal{P}$, and is therefore discarded as a redundant hyperplane. Otherwise, if D is found to be nonredundant against $\mathcal{Q}$, then we cannot claim that it is nonredundant in $\mathcal{P}$. Hence, Check_Redund() is called again (line 10) where D is checked against the bigger set $\mathcal{Q} \cup \mathcal{P}' \cup \{D\}$. If D could not be pruned (line 11), then D is a nonredundant hyperplane and is added to $\mathcal{Q}$. Otherwise, D is pruned and discarded as a redundant hyperplane. Recall that Check_Redund() formulates the LP in (18) and returns a solution witness x^*. Although D is identified as redundant (line 15), the solution witness x^* can be used to check which constraints of the LP were satisfied with equality at x^*; those satisfied would correspond to bounding hyperplanes of the polytope. Hence a set $\mathcal{W}$ of nonredundant delay hyperplanes can be identified (line 16), which are added to $\mathcal{Q}$. The authors of [9] proved that at least one new nonredundant hyperplane is discovered in this step. The reader is referred to [9] for more details about the proof of correctness of this algorithm.

2) *Complexity:* The execution time of PRUNE is dominated by the time to solve the LPs formulated in Check_Redund() at lines 6 and 10. Line 6 is called for all hyperplanes in $\mathcal{P}'$, which is $\mathcal{O}(n)$ times. The LP in line 6 has at most p variables and $\mathcal{O}(m)$ constraints, because m is the largest possible size of $\mathcal{Q}$. So, in total, this would take $\mathcal{O}(np^2m)$. As for the second LP formulated by Check_Redund() in line 10, notice that every time it is called, a new nonredundant hyperplane is discovered, either explicitly (as in lines 11 and 12), or through the use of the solution witness x^* (line 16). Therefore, this LP, which has at most p variables and n constraints, is solved at most m times, which is the total number of nonredundant hyperplanes so that its complexity $\mathcal{O}(mp^2n)$. And since Get_Initial_NR() is $\mathcal{O}(pn)$, then the overall complexity of PRUNE is $\mathcal{O}(p^2mn)$, which is

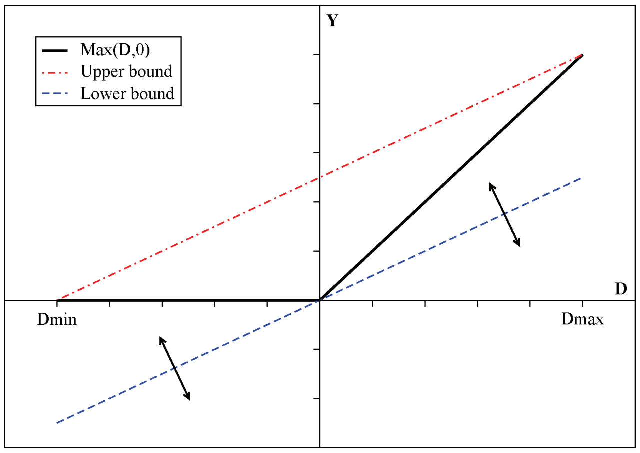

Fig. 4. Bounding the max operation.

an improvement over the $\mathcal{O}(p^2n^2)$ approach of [5], particularly when many hyperplanes are redundant, i.e., when $m \ll n$.

B. Sufficient Condition

Applying the exact algorithm at every node in the timing graph can be expensive. Because we are only interested in the nonredundant hyperplanes at the primary outputs, it makes sense to use a *faster* sufficient condition for pruning at the internal nodes, provided that the number of hyperplanes remains under control. Once the primary outputs are reached, the exact algorithm is applied to determine the nonredundant hyperplanes, which correspond to the potentially critical paths in the circuit. We propose a sufficient condition for pruning based on the following idea. Recall that when a set of hyperplanes $\{D_1, \ldots, D_k\}$ prunes a hyperplane D, the following condition from (5) is satisfied:

$$\max(D_1, \ldots, D_k, D) = \max(D_1, \ldots, D_k), \quad \forall X_i \tag{19}$$

which can be written as

$$\max(D_1, \ldots, D_k) \geq D, \quad \forall X_i \tag{20}$$

and which can be checked using a LP.

Now assume one can find efficiently a hyperplane D_{lb}, which acts as a lower bound on $\max(D_1, \ldots, D_k)$. Then a sufficient condition for pruning D would be to check if D_{lb} prunes D. If so, then $\max(D_1, \ldots, D_k)$ would also prune D, because

$$\max(D_1, \ldots, D_k) \geq D_{lb} \geq D \quad \forall X_i. \tag{21}$$

We show next, based on the work in [10], how we can determine a lower bound on the maximum of a set of hyperplanes.

1) *Finding a Lower Bound:* Let A and B be two delay hyperplanes. Let $C = \max(A, B)$ be the maximum of A and B, and assume that either hyperplane can become dominant. We are interested in finding a hyperplane, C_{lb}, that acts as a lower bound on C. It turns out to be useful to explain the lower bound by first describing a useful upper bound hyperplane C_{ub} on C, as follows. We can write C as follows:

$$C = \max(A, B) = B + \max([A - B], 0) \tag{22}$$
$$= B + \max(D, 0) = B + Y \tag{23}$$

where $D = A - B$ and $Y = \max(D, 0)$. Note that the difference D is also a hyperplane, $D = d_o + \sum d_i X_i$, and assume that $D_{\max}$ and $D_{\min}$ are the maximum and minimum values of D over the space of variation, which can be determined easily. Notice that $D_{\max} \geq 0$ and $D_{\min} \leq 0$ since either A or B can become dominant.

Fig. 4 shows a broken solid line representing a plot of $Y = \max(D, 0)$ between $D_{\min}$ and $D_{\max}$, the extreme values of D. We are interested in finding a linear function of D that is guaranteed to upper bound Y; unlike Y this linear function of D would also be a hyperplane. The dashed-dotted line represents an affine function of D which upper bounds Y and is exact at $D_{\max}$ and $D_{\min}$. The equation for Y_{ub}, the upper bound on Y, can be expressed as follows:

$$Y_{ub} = \frac{D_{\max}}{D_{\max} - D_{\min}}(D - D_{\min}). \tag{24}$$

By replacing Y with Y_{ub} in (22), we get an upper bound C_{ub} as follows:

$$\begin{aligned} C_{ub} &= B + Y_{ub} = B + \frac{D_{\max}}{D_{\max} - D_{\min}}([A - B] - D_{\min}) \\ &= \left(\frac{D_{\max}}{D_{\max} - D_{\min}}\right) A - \left(\frac{D_{\min}}{D_{\max} - D_{\min}}\right) B \\ &- \frac{D_{\max} \cdot D_{\min}}{D_{\max} - D_{\min}}. \end{aligned} \tag{25}$$

Note that, unlike C, C_{ub} is a hyperplane since it is a linear combination of A and B. To gain a more intuitive understanding of the above relationship, let us define the following terms.

1) $S = D_{\max} - D_{\min}$ to be the "spread" of D.
2) $S_A = D_{\max}$ to be the "strength" of A, i.e., the region where A dominates B $(D \geq 0)$.
3) $S_B = -D_{\min} = |D_{\min}|$ to be the "strength" of B, i.e., the region where B dominates A $(D \leq 0)$.
4) $\alpha = \frac{S_A}{S}$ is the fraction of space where A dominates B.
5) $(1 - \alpha) = 1 - \frac{S_A}{S} = \frac{S_B}{S}$ is the fraction of space where B dominates A.

Then, using the above notations, we can rewrite C_{ub} as follows:

$$C_{ub} = \alpha A + (1 - \alpha)B + \alpha(1 - \alpha) \cdot S \tag{26}$$

where A and B are both weighted by their "extent of dominance," so to speak, and the last term accounts for the region where both A and B are dominant, hence the product of α and $(1 - \alpha)$.

Using the same analysis, we would like to find a lower bound on Y in order to find a lower bound on $C = \max(A, B)$. Looking back at Fig. 4, it is easy to see that any function in the form $Y_{lb} = aD$, where $0 \leq a \leq 1$, is a valid lower bound on $Y = \max(D, 0)$, and, unlike Y, is also a hyperplane since D is a hyperplane. In practice, we have found that limiting the choice of Y_{lb} to one of three functions depending on the

Algorithm 4 PRUNE_LB

```
Input: Set of hyperplanes $\mathcal{P} = \{D_1 \ldots, D_n\}$ of size $n$;
Output: Reduced set $\mathcal{Q} \subseteq P$;
1: $\mathcal{Q}$ = Get_Initial_NR($\mathcal{P}$);
2: Find a lower bound $D_{lb}$ on the maximum of all
   hyperplanes in $\mathcal{Q}$;
3: $\mathcal{P}' = \mathcal{P} \backslash \mathcal{Q}$;
4: repeat
5:    Choose an arbitrary $D \in \mathcal{P}'$ and remove it from $\mathcal{P}'$;
6:    Check if $D_{lb}$ prunes $D$, i.e. $D \prec D_{ib}$;
7:    if ($D \prec D_{lb}$) then
8:       $D$ is redundant in $\mathcal{P}$ {sufficient condition is able to
         prune};
9:    else
10:      Add $D$ to $\mathcal{Q}$ {sufficient condition fails, and $D$ is
         not pruned};
11:   end if
12: until $\mathcal{P}' = \{\}$
```

values of $D_{\min}$ and $D_{\max}$ is sufficient; Y_{lb} is set to be one of the following:

$$Y_{lb} = \begin{cases} D, & \text{if} |\text{D}_{\max}| \gg |\text{D}_{\min}| \\ 0, & \text{if} |\text{D}_{\max}| \ll |\text{D}_{\min}| \\ \left(\frac{D_{\max}}{D_{\max} - D_{\min}}\right) D, & \text{otherwise} \end{cases} \quad (27)$$

where the slope of Y_{lb} in the third case is equal to that of the upper bound Y_{ub}. Note that $\gg$ means "much larger than," and it was set to be at least four times.

Replacing each case of the above in (22) gives us C_{lb}, the lower bound on C

$$C_{lb} = \begin{cases} A, & \text{if} |\text{D}_{\max}| \gg |\text{D}_{\min}| \\ B, & \text{if} |\text{D}_{\max}| \ll |\text{D}_{\min}| \\ \alpha A + (1 - \alpha) B, & \text{otherwise} \end{cases} \quad (28)$$

where α is as defined in (26). Therefore, C_{lb} is a hyperplane since it is a linear combination of A and B in either of the three cases.

Although the above analysis is restricted to finding a lower bound on the maximum of two hyperplanes, it can be recursively applied to find a lower bound on the maximum of $n \geq 2$ hyperplanes. It is easy to show that the complexity of doing this for n hyperplanes is $\mathcal{O}(pn)$.

2) *Algorithm Description:* Algorithm 4 describes our lower bound based sufficient condition for pruning, PRUNE_LB. It takes as input a set $\mathcal{P}$ of n hyperplanes and returns a reduced set $\mathcal{Q} \subseteq \mathcal{P}$. Similarly to PRUNE, PRUNE_LB starts by determining an initial set of nonredundant hyperplanes by calling Get_Initial_NR(), which takes $\mathcal{O}(pn)$ time. Next, a lower bound D_{lb} on the maximum of all hyperplanes in this set is determined as shown in the previous section. This takes $\mathcal{O}(p^2)$ time since the size of the initial set is $\mathcal{O}(p)$. Then, for every hyperplane D in the remaining set $\mathcal{P}'$, the lower bound is used to test whether D is prunable or not. If so, then D is redundant in $\mathcal{P}$, otherwise, D is added to $\mathcal{Q}$; the cost of this loop is $\mathcal{O}(np)$. Thus, the runtime of PRUNE_LB is $\mathcal{O}(p(p+n))$, which in practice is really $\mathcal{O}(pn)$ because one expects that it is always the case that $p < n$. This represents an improvement over the $\mathcal{O}(pn^2)$ sufficient condition of [5].

V. Results

In order to verify the accuracy and speed of our pruning techniques, we have tested this approach on a number of circuits from the ISCAS-85 and ITC-99 benchmark suites, mapped to a commercial 90 nm library. The timing engine was implemented in C++, with an interface to the commercial optimization package MOSEK [11], which was used to solve the LPs in PRUNE and FEASCHK algorithms. In our tests, the cell library was not characterized for sensitivities to any specific process parameters. Instead, and in order to allow us to test the approach under some extreme conditions, we have assumed that cell delay depends on a set of ten arbitrary parameters that are normalized to vary in $[-1, 1]$ as per PSTA timing models. In addition, the delay sensitivities to these parameters were generated randomly such that every cell exhibits a total of $\pm 20\%$ deviation in its nominal delay as a result of parameter variability. As to the signs of these sensitivities, they were set at random, again in order to better test the limits of our approach (as the sensitivity signs are made less correlated, one would expect to see more nonredundant hyperplanes).

We test our approach using the following flow: run PRUNE_LB on every node in the timing graph and then apply PRUNE at the primary output to determine the exact number of nonredundant hyperplanes. For comparison, we also test the equivalent flow from [5]: run PAIRWISE on all nodes and then apply FEASCHK at the primary output. Table I shows the results, where we report the number of hyperplanes reported at the primary output by the sufficient condition (PRUNE_LB and PAIRWISE), the number of nonredundant hyperplanes found at the primary outputs after exact pruning (PRUNE and FEASCHK), and the total runtimes. For example, for circuit c432, the number of hyperplanes propagated by PRUNE_LB to the primary output is 122, from which 93 hyperplanes are found to be nonredundant by PRUNE. The overall runtime is found to be 0.23 s.

We can draw several conclusions from Table I. First, it is clear that the proposed approach is practical and offers the hope that the timing of the circuit across the *whole* PVT space *can* be provided for use by downstream tools. Second, note that what determines whether a circuit is *harder* or *easier* to analyze is not necessarily the number of gates in the design, but the number of hyperplanes that are nonredundant, i.e., the number of potentially critical paths; this depends on circuit topology. While c7552 is larger than c6288, we find that the latter takes more time to analyze as the resulting number of nonredundant hyperplanes is much larger. Notice that even under the extreme sensitivity settings, described above, most circuits have a reasonably small number of nonredundant hyperplanes at their outputs, which is in-line with the observations in [5] where the number of potentially critical paths was shown to be manageable for most circuits. For easy circuits, the performance of our method is comparable to [5], while for harder circuits, such as c6288 and I99C1, our approach becomes faster.

TABLE I

SUMMARY OF HYPERPLANES AT PRIMARY OUTPUT AND RUNTIMES FOR: 1) PRUNE_LB + PRUNE, AND 2) PAIRWISE + FEASCHK

Circuit	PAIRWISE Result	FEASCHK Result	Runtime (s)	PRUNE_LB Result	PRUNE Result	Runtime (s)
*c*432	116	93	0.24	122	93	0.23
*c*2670	86	81	0.34	91	81	0.29
*c*5315	526	441	1.3	535	441	1.29
*c*6288	5238	2353	194.56	15 404	2353	108.77
*c*7552	319	241	0.93	299	241	0.89
*b*01	13	11	0.05	13	11	0.04
*b*05	665	456	2.63	671	456	2
*b*14	2003	1703	6.19	1962	1703	6.23
*b*17	28 346	20 662	111.48	32 355	20 662	95.88
*I*99*C*1	4450	1045	118.79	9426	1045	59.41

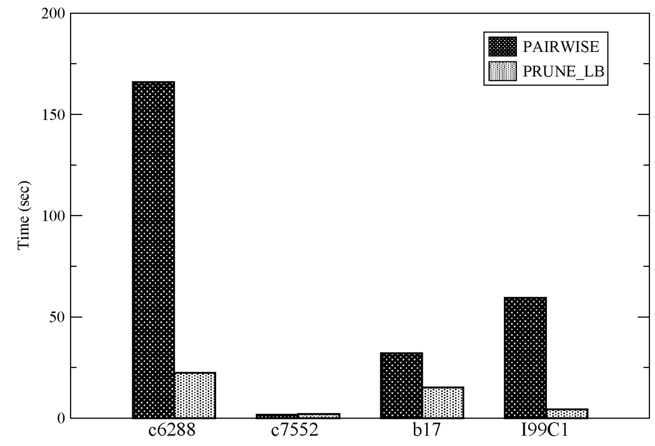

Fig. 5. Comparison of PRUNE_LB and PAIRWISE algorithms.

Another metric to look at is the quality of pruning by the sufficient condition PRUNE_LB, and a comparison of that to PAIRWISE of [5]. As explained earlier, both algorithms are applied on every node in the test circuits until the primary outputs are reached. Table I shows a comparison of the two pruning techniques in terms of how many hyperplanes they report at the primary outputs, as shown in the second and fifth columns. For example, PAIRWISE and PRUNE_LB report 4450 and 9426 nonredundant hyperplanes at the primary output of circuit I99C1, respectively, while the exact number of nonredundant hyperplanes is 1045. Notice that the PAIRWISE algorithm typically yields results that are closer to the exact solution. However, this advantage of PAIRWISE comes at a runtime disadvantage when compared to PRUNE_LB, as can be seen for our larger circuits shown in Fig. 5. This is particularly true for c6288 and I99C1, which are *harder* circuits compared to the rest of the test circuits. For example, although the number of hyperplanes reported by PRUNE_LB for circuit c6288 is almost three times larger than that reported by PAIRWISE, the speed up is 7.5×. On the other hand, the performance of the two methods, in terms of both runtime and quality of pruning, is comparable for other cases where the number of hyperplanes seen at the output is smaller (e.g., as shown for c7552).

VI. Partial Critical Surfaces

Finding the complete critical PWP surface at an output node of the timing graph allows one to study this surface at any point in the PVT space. However, designers would typically be interested in looking at areas of this surface where timing failures occur or are within some margin of occurring. This can be used to achieve runtime improvements by finding partial critical surfaces that are accurate in all areas where the complete critical surface is within some margin of violating timing constraints. Such a partial surface can be found by propagating only potentially critical paths that are also *potentially failing*. Here, we give a formal definition of potentially failing paths and present an overview of how they can be used to improve runtimes.

A failing path is defined as a path that fails a setup or a hold timing constraint of the circuit. Consider the hyperplane delay D_j of a path terminating at an output node z. Assume that the launch register for hyperplane D_j has a clock-to-q time τ_{cq}, and that its capture register has a required setup time τ_s, and hold time, τ_h. If the clock signal arrival times at the input and output registers are a_{in} and a_{out}, respectively, then the path delay D_j has to satisfy

$$\begin{aligned} a_{\text{in}} + \tau_{cq} + D_j + \tau_s - a_{\text{out}} &\leq T \\ a_{\text{in}} + \tau_{cq} + D_j - \tau_h - a_{\text{out}} &\geq 0 \end{aligned} \tag{29}$$

where T is the required clock period for this circuit. Assuming that τ_{cq}, τ_s, τ_h, a_{in}, and a_{out} are also hyperplanes, we can write (29) as

$$R'_z \leq D_j \leq R_z \tag{30}$$

where R'_z and R_z are the hold and setup required arrival times, respectively. These required arrival times are hyperplanes in the process parameters X_j, $1 \leq j \leq p$ because they are found by taking the sums and differences of such hyperplanes. We say that the path corresponding to D_j is potentially failing if the constraints in (30) are violated for some process setting. That is, the path corresponding to D_j is a potentially failing path if, and only if

$$\begin{aligned} \min_X (R_z - D_j) &< 0 \\ \max_X (R'_z - D_j) &> 0. \end{aligned} \tag{31}$$

Since both $R_z - D_j$ and $R'_z - D_j$ are hyperplanes, determining their minimum and maximum values over the process space is a simple operation. Therefore, given R'_z and R_z, checking if D_j is a potentially failing hyperplane is straightforward.

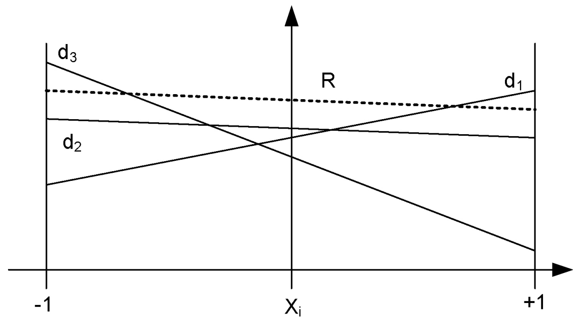

Fig. 6. Potentially failing paths.

It is important to note that potentially critical paths are not always potentially failing. For example, consider the set of potentially critical hyperplane delays $\{d_1, d_2, d_3\}$ in Fig. 6, which also shows the setup required arrival time R. Note that although d_2 is potentially critical it does not fail the setup timing constraint. On the other hand, d_1 and d_3 are potentially failing and together they form a partial surface of potentially critical paths that are also potentially failing. This partial surface is accurate in areas of the PVT space where timing failures occur; however, its accuracy is not guaranteed in "safe" areas. Because we are mainly interested in identifying and fixing timing failures, this represents an opportunity to reduce the computational load of our method. That can be done by propagating partial surfaces that consist of *failing* critical paths only through the timing graph, instead of complete critical surfaces. In this way, we can reduce the number of hyperplanes carried forward and the sizes of LPs required in the exact pruning methods. This can be used to fix all timing failures, and after that, one can perform a timing run that propagates complete critical surfaces to compute accurate robustness measures or to perform further optimizations.

VII. Partial Surface Propagation

Finding partial critical surfaces can be done as follows. First, required arrival times are propagated backward in the timing graph to find required arrival times at all internal nodes of the timing graph. Then, the same forward propagation used for complete critical surfaces is applied, albeit with an additional processing step. At each node, all propagated hyperplanes are first compared to the required arrival time to determine the potentially failing paths. All "safe" paths, i.e., paths that pass the required arrival time are removed and our earlier pruning methods are then applied on the remaining potentially failing paths to find the partial critical surface of these. This reduces the number of candidate paths at each circuit node and allows us to find critical delay surfaces faster. However, these surfaces will include only potentially failing paths.

A. Propagating Required Arrival Times

Typically, required arrival times will be available at the outputs of a timing graph. However, it is also possible to perform a backward propagation that will give required arrival times at all the internal nodes of the timing graph. We will

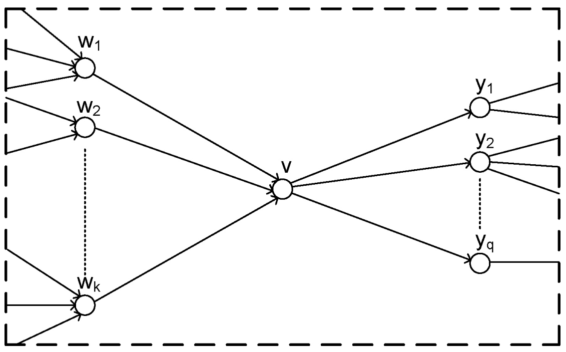

Fig. 7. Cross-section of a timing graph showing an arbitrary node.

restrict our discussion to required arrival times for satisfying setup timing constraints, i.e., longest path delay constraints. However, with minor changes, all arguments are applicable to required arrival times for hold timing constraints.

The required arrival time S_v at node n, shown in Fig. 7, can be written in terms of the required arrival times S_{y_q} of its output nodes y_i, $1 \le i \le q$, and the delays d_{vy_i}, $1 \le i \le q$, as follows:

$$S_v = \min\left(S_{y_1} - d_{vy_1}^{\max}, \ldots, S_{y_q} - d_{vy_q}^{\max}\right). \tag{32}$$

This means that the required arrival time at the internal nodes of a timing graph can be found through a backward propagation of required arrival times starting at the outputs of the timing graph. Required arrival times at the outputs of the timing graph will be hyperplanes, and as a result, internal required arrival times will be the minimum surfaces of a set of hyperplanes. In general, these minimum surfaces will be PWP surfaces. However, finding these surfaces would be too expensive and would offset any runtime gains we are hoping to achieve. Instead, we find conservative *hyperplane* estimates of required arrival times that are much cheaper to compute and propagate. Because these hyperplane required arrival times are conservative, they can be safely used to prune signal arrival time hyperplanes in the forward propagation. That is, paths delay hyperplanes that satisfy the conservative required arrival times are guaranteed to satisfy the "exact" required arrival times and can be pruned. The authors of [4] presented a method max_hp where given a set of hyperplanes, a conservative hyperplane estimate of their maximum can be found in time linear in the number of hyperplanes. The time complexity of max_hp is $\mathcal{O}(np)$ where n is the number of hyperplanes and p is the number of process parameters. This method can be easily adapted to write a method min_hp, which finds a conservative hyperplane minimum of a set of hyperplanes as follows:

$$d_{\min} = min\_hp\left(d_1, \ldots, d_q\right) = -max\_hp\left(-d_1, \ldots, -d_q\right). \tag{33}$$

Our backward propagation uses this method to find hyperplane required arrival times at internal nodes as follows. For a node v shown in Fig. 7, hyperplane required signal arrival times $\hat{S}_{y_i}$ will be available at each of its output nodes y_i. Then, given the hyperplane delays $d_{vy_i}^{\max}$ of the edges between v and

TABLE II
SUMMARY OF HYPERPLANES AT PRIMARY OUTPUT AND RUNTIMES FOR CRITICAL AND CRITICAL_RAT

Circuit	CRITICAL Result	Runtime (s)	CRITICAL_RAT $M = 0\%$	Runtime s	CRITICAL_RAT $M = 5\%$	Runtime (s)	CRITICAL_RAT $M = 10\%$	Runtime (s)
*c*432	93	0.23	32	0.12	49	0.15	64	0.19
*c*2670	81	0.29	3	0.12	3	0.12	6	0.14
*c*5315	441	1.29	33	0.32	58	0.4	95	0.51
*c*6288	2353	108.77	252	14.15	564	34.23	905	55.82
*c*7522	241	0.89	14	0.37	22	0.4	37	0.44
*b*01	11	0.04	2	0.01	3	0.03	3	0.03
*b*05	456	2	80	0.57	133	0.89	214	1.27
*b*14	1703	6.23	12	1.61	27	1.84	285	2.47
*b*17	20 662	95.88	65	9.83	288	10.92	999	14.65
*I*99*C*1	1045	59.41	283	14.57	471	36.27	476	41.45

its output nodes, the hyperplane required arrival time $\hat{S}_v$ can be found by writing

$$\hat{S}_v = min\_hp\left(\hat{S}_{y_1} - d_{vy_1}^{\max}, \hat{S}_{y_2} - d_{vy_2}^{\max}, \ldots, \hat{S}_{y_q} - d_{vy_q}^{\max}\right). \tag{34}$$

This method is applied in a backward traversal that visits all the nodes of the timing graph in reverse topological order until the inputs are reached and all internal required arrival times are found. However, from (32) we can see that this requires the availability of maximum edge delays. These delays are functions of the incoming signal transition times, and are known exactly only when these transition times are available. This means that in order to find required arrival times at internal nodes, one has to first find edge delays using a forward propagation.

B. Finding Edge Delays

Consider the node v, shown in Fig. 7. In order to compute the maximum edge delays $d_{vy_i}^{\max}$, $1 \leq i \leq q$, it is required to have the maximum signal transition time $t_v^{\max}$ at node n. In this paper, we use a forward propagation of maximum signal transition times to find an upper bound on the signal slew of each node in the timing graph. The nodes are visited in topological order, and at each node v, this transition time can be computed using the maximum signal transition times of its input nodes as follows. Let each of the nodes w_i, $1 \leq i \leq k$, have a maximum signal transition time $t_{w_i}^{\max}$. The output slew $t_{w_i n}$ of each of the edges $w_i v$, $1 \leq i \leq k$, is a function of its input slew and its output capacitance. Therefore, the maximum output slew $t_{w_i n}^{\max}$ of each of the edges $w_i n$ can be written as $t_{w_i v}(t_{w_i}^{\max})$ and an upper bound $t_v^{\max}$ on t_v can be found as

$$t_v^{\max} = \max(t_{w_1 v}^{\max}, \ldots, t_{w_k v}^{\max}). \tag{35}$$

This bound can then be similarly propagated to y_i, $1 \leq i \leq q$. After this forward propagation of maximum slews in the timing graph is done, it becomes possible to use these slew values to compute conservative hyperplane delays for all timing graph edges. These edge delays can now be used to find conservative internal required arrival times through the backward propagation described earlier.

C. Modified Propagation

After computing hyperlane required arrival times at all the internal nodes of the timing graph using the forward slew propagation and the backward propagation of required arrival times, it becomes possible to propagate partial critical surfaces through the timing graph. This is done by propagating hyperplanes as in the case of complete critical surfaces; however, this propagation includes an additional pruning step at each node. Given the required arrival time $\hat{S}_v$ at a given node v, each hyperplane D_j in the propagated set of arrival time hyperplanes D_v is pruned if it satisfies the following constraint:

$$\min_X(\hat{S}_v - D_j) > 0. \tag{36}$$

Because the internal required arrival time $\hat{S}_v$ is conservative, all such hyperplanes in D will satisfy required arrival times at the outputs and are pruned to get the set D'. After that, we apply any of our pruning algorithms (PRUNE or PRUNE_LB) to D' to find and propagate the critical surface of this set.

D. Results

We test our approach using the following flow, which we refer to as CRITICAL_RAT: find internal required arrival times at all internal nodes of the circuit, and then run a forward propagation where at every node of the timing graph required arrival time pruning is used to prune "safe" hyperplanes and PRUNE_LB on the remaining hyperplanes to find partial critical surfaces. In addition, for the primary outputs of the timing graph, PRUNE is applied after PRUNE_LB to find exact partial surfaces for these nodes. We compare this flow to our standard flow that applies PRUNE_LB for all nodes and PRUNE for outputs without using required arrival time pruning. This flow is referred to as CRITICAL. Note that in CRITICAL_RAT one could manipulate the required arrival times at the output nodes before these are propagated backward to make internal required arrival times more stringent. This means that potentially critical hyperplanes that are "close" to failing are also propagated forward, and that the partial surfaces propagated forward become accurate where timing failures occur or are "close" to occurring. This allows designers to account for any unmodeled physical or dynamic effects that might exist on silicon. So, in our tests, we ran CRITICAL_RAT with output required arrival times reduced by a margin M of 0%, 5%, and 10%. Table II shows the results, where we show the number of nonredundant hyperplanes reported at the primary outputs by CRITICAL and CRITICAL_RAT, in addition to the total runtimes of each. The results for CRITICAL_RAT are presented for the three different margins applied to required arrival times at the outputs.

Let us first consider the case of $M = 0\%$ in Table II. It is clear that CRITICAL_RAT results in a much smaller number of nonredundant hyperplanes at the outputs. This is the result of pruning hyperplanes that satisfy internal required arrival times, which also results in less hyperplanes carried forward and in much smaller runtimes. The circuits that show the greatest speedup are those where a large number of hyperplanes were being propagated forward. In particular, we see that circuits $c6288$, $b17$, and $I99C1$ now have a much smaller number of hyperplanes at the outputs and that the speedup achieved is very large. On the other hand, for circuits where the number of nonredundant hyperplanes was small to begin with, such as $c432$ and $b01$, the speedup is not as large. Now, let us consider what happens when we start using a margin with the output required arrival times (i.e., making them more stringent before backward propagation). In the case of $M = 5\%$ we start to see that the number of nonredundant hyperplanes and runtimes start to increase. When we get to $M = 10\%$, we see further increases in these; however, the runtime gain for all the circuits is still significant. Moreover, this partial loss of runtime gains allows us to focus timing failures and identifies paths that are close to failing, thus requiring attention.

VIII. Conclusion

In this paper, we proposed an efficient block-based parameterized timing analysis technique that can accurately capture circuit delay at every point in the parameter space, by reporting all paths that can become critical. Using efficient pruning algorithms, only those potentially critical paths are carried forward, while all other paths are pruned during propagation. After giving a formal definition of this problem, we proposed: 1) an exact algorithm for pruning; and 2) a fast sufficient condition for pruning, that improve on the state of the art, both in terms of theoretical computational complexity and in terms of runtime on various test circuits. In addition, we presented a method where internal required arrival times are used to prune "safe" *potentially critical* paths, thus achieving further speedup for our method.

References

[1] H. Chang and S. S. Sapatnekar, "Statistical timing analysis considering spatial correlations using a single PERT-like traversal," in *Proc. IEEE/ACM Int. Conf. Comput.-Aided Des.*, Nov. 2003, pp. 621–625.

[2] C. Visweswariah, K. Ravindran, K. Kalafala, S. Walker, and S. Narayan, "First-order incremental block-based statistical timing analysis," in *Proc. Des. Autom. Conf.*, Jun. 2004, pp. 331–336.

[3] A. Agarwal, D. Blaauw, and V. Zolotov, "Statistical timing analysis for intra-die process variations with spatial correlations," in *Proc. IEEE/ACM Int. Conf. Comput.-Aided Des.*, Nov. 2003, pp. 900–907.

[4] S. Onaissi and F. N. Najm, "A linear-time approach for static timing analysis covering all process corners," *IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst.*, vol. 27, no. 7, pp. 1291–1304, Jul. 2008.

[5] S. V. Kumar, C. V. Kashyap, and S. S. Sapatnekar, "A framework for block-based timing sensitivity analysis," in *Proc. Des. Autom. Conf.*, Jun. 2008, pp. 688–693.

[6] K. R. Heloue, S. Onaissi, and F. N. Najm, "Efficient block-based parameterized timing analysis covering all potentially critical paths," in *Proc. ICCAD*, Nov. 2008, pp. 173–180.

[7] S. P. Boyd and L. Vandenberghe, *Convex Optimization*. Cambridge, U.K.: Cambridge University Press, 2004.

[8] F. P. Preparata and M. I. Shamos, *Computational Geometry: An Introduction*. Berlin, Germany: Springer, 1985.

[9] T. Ottmann, S. Schuierer, and S. Soundaralakshmi, "Enumerating extreme points in higher dimensions," *Nordic J. Comput.*, vol. 8, no. 2, pp. 179–192, 2001.

[10] K. R. Heloue and F. N. Najm, "Parameterized timing analysis with general delay models and arbitrary variation sources," in *Proc. Des. Autom. Conf.*, Jun. 2008, pp. 403–408.

[11] MOSEK. [Online]. Available: http://www.mosek.com/

Khaled R. Heloue received the B.E. degree in computer and communications engineering from the American University of Beirut, Beirut, Lebanon, in 2003, and the M.A.Sc. and Ph.D. degrees in electrical and computer engineering from the University of Toronto, Toronto, ON, Canada, in 2005 and 2010, respectively.

In 2008, he interned with the Strategic CAD Labs, Intel Corporation, Hillsboro, OR. He is currently a Member of Technical Staff with the Technology Group, AMD, Markham, ON, where he is developing design guidelines and methodologies to cover the impact of variability on timing. His current research interests include computer-aided design for very large scale integration, with a focus on performance analysis in the presence of process and environmental variability.

Sari Onaissi received the B.E. degree in computer and communications engineering from the American University of Beirut, Beirut, Lebanon, in 2005, and the M.A.Sc. and Ph.D. degrees in electrical and computer engineering from the University of Toronto, Toronto, ON, Canada, in 2007 and 2011, respectively.

In 2009, he interned with the PrimeTime Team, Synopsys, Inc., Mountain View, CA. He is currently with the Design and Technology Solutions Division, Intel, Santa Clara, CA.

Farid N. Najm (S'85–M'89–SM'96–F'03) received the B.E. degree in electrical engineering from the American University of Beirut, Beirut, Lebanon, in 1983, and the Ph.D. degree in electrical and computer engineering (ECE) from the University of Illinois at Urbana-Champaign (UIUC), Urbana, in 1989.

From 1989 to 1992, he was with Texas Instruments, Dallas, TX. He then joined the ECE Department, UIUC, as an Assistant Professor and became an Associate Professor in 1997. In 1999, he joined the Department of ECE, University of Toronto, Toronto, ON, Canada. He is currently a Professor and Chair with the Department of ECE, University of Toronto. In 2010, he authored the book *Circuit Simulation* (Wiley, New York). His current research interests include computer-aided design for very large scale integration, with an emphasis on circuit level issues related to power, timing, variability, and reliability.

Dr. Najm is a Fellow of the Canadian Academy of Engineering. He was an Associate Editor for the IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems from 2001 to 2009. He has received the IEEE Transactions on Computer-Aided Design Best Paper Award, the NSF Research Initiation Award, the NSF CAREER Award, and was an Associate Editor for the IEEE Transactions on Very Large Scale Integration Systems from 1997 to 2002. He serves on the Executive Committee of the International Symposium on Low-Power Electronics and Design (ISLPED). He has served on the technical committees of various conferences, including ICCAD, DAC, CICC, ISQED, and ISLPED.