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

Khaled R. Heloue
Department of ECE
University of Toronto
Toronto, Ontario, Canada
khaled@eecg.utoronto.ca

Sari Onaissi
Department of ECE
University of Toronto
Toronto, Ontario, Canada
sari@eecg.utoronto.ca

Farid N. Najm
Department of ECE
University of Toronto
Toronto, Ontario, Canada
f.najm@utoronto.ca

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 (which 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 that improves on the state of the art, both in terms of theoretical computational complexity and in terms of run time on various test circuits.

1. 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 (ASICs 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 *i)* an increasing number of circuit parameters have significant variability, causing an increase in the number of corners, and *ii)* 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.

Given the two types of variables under study, two types of solution techniques have emerged: statistical static timing analysis (SSTA) and multi-corner static timing analysis (MCSTA). SSTA models parameters as random variables, assuming that their distributions and correlations are known a priori [1, 2, 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 work 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 which 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 for 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

*This work was supported in part by Intel Corp.

978-1-4244-2820-5/08/$25.00 ©2008 IEEE

which is implicit in the problem of timing verification of setup constraints (a similarly broken delay surface results from the *min* function, in the similar problem of verifying hold constraints). If one assumes a non-linear, 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 run-time can be significant for *hard* circuits.

The rest of the paper is organized as follows. In section 2, 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 3, 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 4, we present our exact pruning algorithm and our sufficient condition for pruning, and study the complexity of these pruning strategies. We provide test results and comparisons to previous work in section 5, and conclude in section 6.

2. 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.

2.1 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, where the arrival time C at the output of the AND gate is computed as the *max* of the sums of arrival times and arc delays at the inputs of the gate. In other words:

$$C = \max\left(A + D_1, B + D_2\right) \tag{1}$$

where A and B are input arrival times, and D_1 and D_2 are 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 well this dependence. In this work, 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:

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

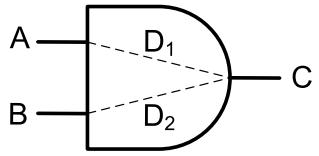

Figure 1: Propagation for a single gate

where d_o is the nominal delay, X_i's are the normalized PVT parameters, and d_i's are the delay sensitivities to these parameters. Since D is a linear function of p parameters, then it is referred to as a delay *hyperplane*.

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 the 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 piecewise-planar (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 *non-redundant* 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 non-redundant hyperplanes) must be propagated to subsequent stages, while all other hyperplanes must be discarded or pruned.

2.2 The 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 piecewise-planar surface representing the max. On the other hand, if (5) is not satisfied, then D_j is a non-redundant hyperplane and must be kept in the set. An example of this are paths $P_1 - P_3$ which 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 non-redundant hyperplanes $\tilde{D}_j$, where $m \le n$, 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 non-redundant hyperplanes are needed to describe the shape of the PWP surface defined by the max. This pruning

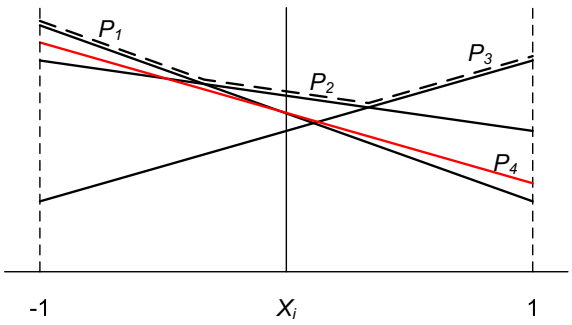

Figure 2: MAX of path delay hyperplanes

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.

2.2.1 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:

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

$$D_2 = a_{o2} + \sum_{i=1}^{p} a_{i2} X_i \quad (8)$$

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:

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

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 non-prunable. 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 non-redundant by PAIRWISE, are propagated to subsequent stages, potentially causing a blow-up in the number of hyperplanes, as reported by [5] on one of the test circuits.

2.2.2 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 non-redundant hyperplanes. The idea is to perform a *feasibility check* for every hyperplane D_j by searching for a point in the space of X_i's where D_j is dominant over *all* other hyperplanes. If this is feasible, then D_j is non-redundant, otherwise, D_j is redundant and can be pruned from the set. Thus, D_j is non-redundant if and only if the following system of inequalities has a feasible solution:

$$D_j \geq D_k, \quad k = 1, \ldots, n, \; k \neq j \quad (10)$$
$$-1 \leq X_i \leq 1, \quad i = 1, \ldots, p$$

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 non-redundant and is added to $\mathcal{Q}$. Otherwise, it is marked as redundant and is pruned from the set.

Algorithm 1 PAIRWISE

Input: $\mathcal{P} = \{D_1, \ldots, D_n\}$;
Output: $\mathcal{Q} \supseteq \{\tilde{D}_1, \ldots, \tilde{D}_m\}$;
1: Mark all hyperplanes in $\mathcal{P}$ as non-redundant;
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 non-redundant hyperplanes to $\mathcal{Q}$;

Algorithm 2 FEASCHK

Input: $\mathcal{P} = \{D_1, \ldots, D_n\}$;
Output: $\mathcal{Q} = \{\tilde{D}_1, \ldots, \tilde{D}_m\}$;
1: Mark all hyperplanes in $\mathcal{P}$ as non-redundant;
2: **for** $j = 1 : n$ **do**
3: **if** (D_j is marked redundant) **then**
4: continue;
5: **end if**
6: Formulate (10) for D_j excluding redundant hyperplanes;
7: **if** (feasible) **then**
8: Add D_j to $\mathcal{Q}$;
9: **else**
10: Mark D_j as redundant;
11: **end if**
12: **end for**

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 [6]. Therefore, the complexity of FEASCHK, which requires n feasibility checks to determine the irreducible set of non-redundant hyperplanes is $\mathcal{O}\left(p^2(n+p)n\right)$, which is $\mathcal{O}(p^2n^2)$ if $p \leq n$, which 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 which is $\mathcal{O}(p^2mn)$, where n is the number of hyperplanes in the initial set $\mathcal{P}$, and m is the number of non-redundant 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 $\mathcal{O}(pn)$.

3. PROBLEM TRANSFORMATION

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

3.1 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 [7]. 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.

3.1.1 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. 3a where the shaded region is the convex hull, and points 1 through 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.

3.1.2 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. 3b 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. 3b, only hyperplanes 1 through 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. 3b, if hyperplanes 5 and 6 were removed, it would not affect the shape of the polytope.

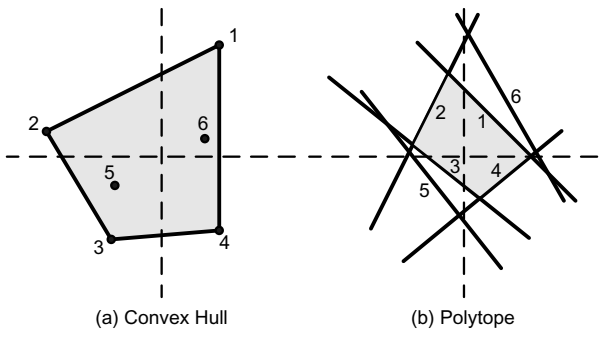

Figure 3: (a) Extreme Points of Convex Hull (b) Minimal Polytope Representation (dual problem)

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 [7]. 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 to 4 of the convex hull are transformed to the bounding hyperplanes 1 to 4 of the polytope; also, the 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 [8], which solves extreme points enumeration. 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 2.2, can be transformed to minimal polytope representation problem. Once this is established, we can adapt the algorithm in [8] to solve the pruning problem efficiently.

3.2 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 non-redundant, 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:

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

Note that if D_j is a non-redundant 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_i's. Otherwise, if D_j is redundant, then $D_{\max} > D_j$ for all X_i's.

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 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 the 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 non-redundant 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 non-redundant delay hyperplanes.

4. 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 [8], 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.

4.1 Exact Algorithm

4.1.1 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:

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

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(), below. This procedure takes as inputs a set of delay hyperplanes $\mathcal{B}$, and a hyperplane $D \in \mathcal{B}$ that we're 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 non-redundant, 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 [6]. Therefore, the complexity of Check_Redund() is linear in the size of $\mathcal{B}$; specifically, it is $\mathcal{O}(p^2|\mathcal{B}|)$.

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

Inputs: Hyperplane D, and a set of hyperplanes $\mathcal{B}$ including 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 3.2}
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 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 non-redundant hyperplanes. The algorithm starts by determining a small subset of non-redundant 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 non-redundant, 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 $\mathcal{O}(pn)$.

Algorithm 3 PRUNE

Input: Set of hyperplanes $\mathcal{P} = \{D_1, \ldots, D_n\}$ of size n;
Output: Set of all non-redundant hyperplanes $\mathcal{Q}$ of size $m \leq n$;
1: $\mathcal{Q}$ = Get_Initial_NR($\mathcal{P}$);
2: $\mathcal{P}' = \mathcal{P} \setminus \mathcal{Q}$;
3: **repeat**
4: Let D be the next hyperplane in $\mathcal{P}'$;
5: Remove D, $\mathcal{P}' = \mathcal{P}' \setminus \{D\}$;
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 non-redundant;
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 non-redundant hyperplanes containing x^*;
17: Add $\mathcal{W}$ to set, $\mathcal{Q} = \mathcal{Q} \cup \mathcal{W}$;
18: **end if**
19: **end if**
20: **until** $\mathcal{P}' = \{\}$

Once this initial set of non-redundant 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 non-redundant 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 non-redundant against $\mathcal{Q}$, then we cannot claim that it is non-redundant in $\mathcal{P}$. Hence, Check_Redund() is called again (line 10) where D is checked against the bigger set $\mathcal{P} \cup \mathcal{P}'$. If D could not be pruned (line 11), then D is a non-redundant hyperplane and is added to $\mathcal{Q}$. Otherwise, D is pruned and dis-

carded 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 non-redundant delay hyperplanes can be identified (line 16), which are added to $\mathcal{Q}$. The authors of [8] prove that at least 1 new non-redundant hyperplane is discovered in this step. The reader is referred to [8] for more details about the proof of correctness of this algorithm.

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

Input: Set of hyperplanes $\mathcal{P} = \{D_1, \ldots, D_n\}$;
Output: Subset $\mathcal{Q}$ of non-redundant 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: $\mathcal{Q} = \mathcal{Q} \cup \{D_j\}$;
12: Reset X_i to 0;
13: **end for**

4.1.2 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 non-redundant hyperplane is discovered, either explicitly (as in lines 11-12), 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 non-redundant 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 an improvement over the $\mathcal{O}(p^2n^2)$ approach of [5], particularly when many hyperplanes are redundant, i.e. when $m \ll n$.

4.2 Sufficient Condition

Applying the exact algorithm at every node in the timing graph can be expensive. Because we are only interested in the non-redundant 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 non-redundant 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 an 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:

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

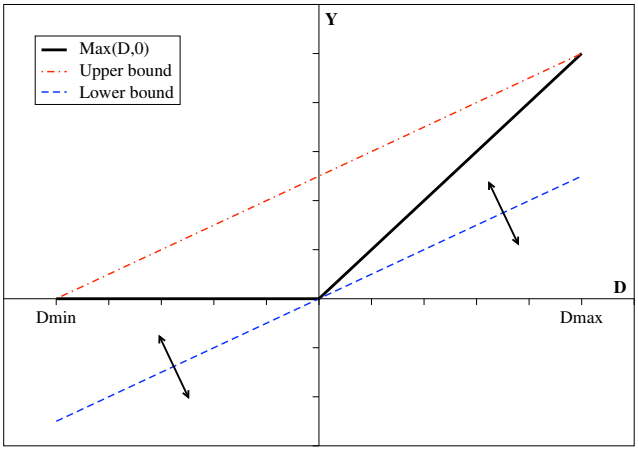

Figure 4: Bounding the max operation

We show next, based on our recent work in [9], how we can determine a lower bound on the maximum of a set of hyperplanes.

4.2.1 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:

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

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 (23), we get an upper bound C_{ub} as:

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

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:

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

- $(1-\alpha) = 1 - \frac{S_A}{S} = \frac{S_B}{S}$ 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 values of $D_{\min}$ and $D_{\max}$ is sufficient; Y_{lb} is set to be one of the following:

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

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 4 times.

Replacing each case of the above in (23) gives us C_{lb}, the lower bound on C:

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

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 $\mathcal{O}(pn)$.

4.2.2 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 non-redundant 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's always the case that $p < n$. This represents an improvement over the $\mathcal{O}(pn^2)$ sufficient condition of [5].

Algorithm 4 PRUNE_LB

Input: Set of hyperplanes $\mathcal{P} = \{D_1, \ldots, D_n\}$ of size n;
Output: Reduced set $\mathcal{Q} \subseteq \mathcal{P}$;
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: $\mathcal{P}' = \mathcal{P} \setminus \mathcal{Q}$;
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. $D \prec D_{lb}$;
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** $\mathcal{P}' = \{\}$

5. 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 benchmark suite, mapped to a commercial 90nm library. The timing engine was implemented in C++, with an interface to the commercial optimization package MOSEK [10], 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 10 arbitrary parameters that are normalized to vary in $[-1, 1]$. 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 non-redundant hyperplanes).

Two of the ISCAS-85 circuits were excluded from the analysis, for the following reasons. Modern circuits are not very deep, i.e., they do not have a large number of logic levels between registers. This is a result of high clock frequencies and heavy pipelining. Thus, the number of input-output paths is not as large as it was in older circuits that may have had 40-50 levels of logic. Given this, and given our extreme settings of the sensitivities, we have encountered unrealistically large dominant hyperplane counts in c1355 and c6288, and we exclude these circuits from the results. Given more realistic (less extreme) settings of the sensitivities, even these circuits would be manageable. Indeed, for c1355, if the sensitivity signs are all set to be the same, then the analysis completes easily with 174 dominant hyperplanes at the circuit primary outputs. For c6288, if the range of variations is reduced to 5% and the sensitivities are made equal, then again the analysis completes quickly, with only 68 non-redundant planes at the output. In any case, we now present the results, under the extreme settings for sensitivity, on all the other circuits.

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 non-redundant 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 1 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 non-redundant 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 1481, from which only 639 hyperplanes are found to be non-redundant by PRUNE. The overall runtime is found to be 128.49sec.

We can draw several conclusions from Table 1. 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. Secondly, note that what determines whether a circuit is *harder* or *easier* to analyze is not the number of gates in the design, but the number of hyperplanes that are non-redundant, i.e., the number of potentially critical paths; this depends on circuit topology. While c7552 is much larger than c432, we find that the latter takes more time to analyze as the resulting number of non-redundant hyperplanes is much larger. Notice that even under the extreme sensitivity settings, described above, most circuits have a reasonably small number of non-redundant 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

Table 1: Summary of hyperplanes at primary output and Run-times for (1) PRUNE_LB + PRUNE and (2) PAIRWISE + FEASCHK

ISCAS-85 Circuit	PRUNE_LB Result	PRUNE Result	Run-time (sec)	PAIRWISE Result	FEASCHK Result	Run-time (sec)
c432	1481	639	128.49	1114	639	338.25
c499	1918	520	156.09	1539	520	229.75
c880	16	12	0.62	16	12	0.57
c1908	63	37	2.09	46	37	2.24
c2670	556	126	18.32	270	126	18.85
c3540	60	30	2.80	56	30	3.55
c5315	18	12	2.22	12	12	2.12
c7552	14	9	3.67	11	9	4.57

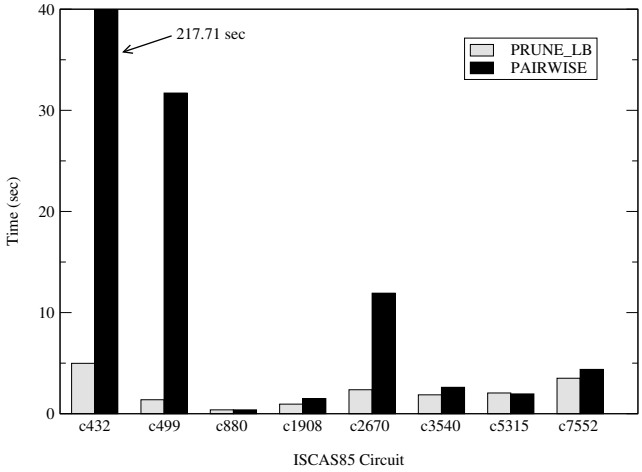

Figure 5: Comparison of PRUNE_LB and PAIRWISE Algorithms

to [5], while for harder circuits, such as c432 and c499, our approach becomes faster.

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 1 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 column. For example, PRUNE_LB and PAIRWISE report 1481 and 1114 non-redundant hyperplanes at the primary output of circuit c432, respectively, while the exact number of non-redundant hyperplanes is 639. 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 in Fig. 5. This is particularly true for c432, c499, and c2670, 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 c432 is slightly larger than that reported by PAIRWISE, the actual speed up is 44×. Also notice that the performance of the two methods, in terms of both runtime and quality of pruning, is comparable for all other *easy* cases where the number of hyperplanes seen at the output is smaller.

Finally, we draw the reader's attention is to circuit c2670 in Table 1, where the total run-time for both flows is comparable. This is due to the fact that PRUNE_LB predicted 556 hyperplanes to be reduced by PRUNE, whereas PAIRWISE predicted only 270 hyperplanes to be reduced by FEASCHK. In a sense, the speed up achieved by our sufficient condition, shown in Fig 5, is lost since our exact algorithm had to reduce a larger set of hyperplanes. In this case, an alternative approach may be to apply PRUNE_LB on every node, then to apply PAIRWISE for additional pruning at the primary output before calling PRUNE to determine the exact number of non-redundant hyperplanes.

6. CONCLUSION

In this work, we have 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 have 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 run time on various test circuits. Our work has also established a link between the pruning problem in the parameterized timing domain, and two standard problems in computational geometry.

7. REFERENCES

[1] H. Chang and S. S. Sapatnekar. Statistical timing analysis considering spatial correlations using a single PERT-like traversal. In *IEEE/ACM International Conference on Computer-Aided Design*, pages 621–625, San Jose, CA, November 9–13 2003.

[2] C. Visweswariah, K. Ravindran, K. Kalafala, S. Walker, and S. Narayan. First-order incremental block-based statistical timing analysis. In *Design Automation Conference*, pages 331–336, San Diego, CA, June 7–11 2004.

[3] A. Agarwal, D. Blaauw, and V. Zolotov. Statistical timing analysis for intra-die process variations with spatial correlations. In *IEEE/ACM International Conference on Computer-Aided Design*, pages 900–907, San Jose, CA, November 9–13 2003.

[4] S. Onaissi and F. N. Najm. A linear-time approach for static timing analysis covering all process corners. In *IEEE/ACM International Conference on Computer-Aided Design*, pages 217–224, San Jose, CA, November 5–9 2006.

[5] S. V. Kumar, C. V. Kashyap, and S. S. Sapatnekar. A framework for block-based timing sensitivity analysis. In *Design Automation Conference*, pages 688–693, Anaheim, CA, June 8–13 2008.

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

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

[8] T. Ottmann, S. Schuierer, and S. Soundaralakshmi. Enumerating Extreme Points in Higher Dimensions. *Nordic Journal of Computing*, 8(2):179–192, 2001.

[9] K. R. Heloue and F. N. Najm. Parameterized timing analysis with general delay models and arbitrary variation sources. In *Design Automation Conference*, pages 403–408, Anaheim, CA, June 8–13 2008.

[10] MOSEK - http://www.mosek.com/.