# PSTA-based Branch and Bound Approach to the Silicon Speedpath Isolation Problem *

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

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

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

## ABSTRACT

The lack of good "correlation" between pre-silicon simulated delays and measured delays on silicon (silicon data) has spurred efforts on so-called silicon debug. The identification of speed-limiting paths, or simply *speedpaths*, in silicon debug is a crucial step, required for both "fixing" failing paths and for accurate learning from silicon data. We propose using characterized, pre-silicon, variational timing models to identify speedpaths that can best explain the observed delays from silicon measurements. Delays of all logic paths are written as affine functions of process parameters, called hyperplanes, and a branch and bound approach is then applied to find the "best" path combinations. Our method has been tested on a set of ISCAS-89 circuits and the results show that it accurately identifies the speedpaths in most cases, and that this is achieved in a very efficient manner.

## 1. INTRODUCTION

"How good are our flows in really predicting silicon?" is a key question that was raised by the authors of [1], hoping to spur more research on learning from silicon data in order to achieve better silicon-to-model correlation. Indeed, this question touches on the widely accepted reality, that silicon speed-limiting paths, or simply silicon *speedpaths*, are not always accurately predicted by existing timing flows [2]. This can be attributed to a number of effects, including process, design, or environmental effects, that are either not fully understood or too difficult to model in modern designs.

Silicon debug, whereby silicon information is obtained particularly in the form of silicon delay measurements and/or actual silicon speedpaths, is an integral part of the design cycle and a crucial step in achieving design closure. In [3], the authors give an overview of the silicon debug process and the many steps involved in it. Typically, designers go through several "revisions" of silicon to further optimize and push the performance of a given design. These silicon iterations are referred to as *silicon steppings*. One obvious objective is to identify speedpaths so they can be "fixed" before the next silicon stepping. In order to do that, functional tests are repeatedly performed while continuously increasing clock frequency so as to capture the point of failure. In this way, engineers are able to identify a speed-limiting "logic block". Using logic and timing simulations, they further attempt to spatially isolate the speedpath that led to the failure. Another, more difficult, objective is to perform *causality analysis* on the silicon speedpaths in order to identify the root causes of failure and possibly predict additional speedpaths. The latter issue is mainly concerned with reconciling models and silicon, as it focuses on how the silicon data, that is, the found speedpaths and their actual measured silicon delays, can be used to tune and improve the EDA tools and flows, particularly the timing flows.

In the past few years, the above problem has been tackled from different angles. In [4], silicon measurements are used to come up with a "criticality ranking" of paths. The authors do not try to find the speedpath corresponding to each measurement, but rather to specify a list of critical paths. In [1], the authors give an overview of *speedpath isolation* techniques. These involve functional and timing analysis to determine the exact speedpath out of a set of candidate paths. The authors also present a method to identify the root causes which make these paths speed-limiting. As mentioned earlier, the main purpose of this is to utilize the silicon data to improve the EDA tools and models and is done as follows.After isolating the speedpaths, simulation is used to determine the sensitivities of these paths to a set of chosen parameters. Based on the magnitudes of the sensitivities, the possible root causes of the timing failure can then be ranked. The delays of the paths from silicon are not used in the analysis, and the authors do not try to reconcile actual measurements with the models. On the other hand, the authors of [2] employ a machine-learning approach that uses a small number of identified speedpaths to predict a larger set of paths that are potentially speed-limiting. They do this without identifying the root causes, which they say saves time and reduces the time to market. Other works, e.g. [5] use silicon data to find exact gate delays for a given chip which can be useful in tuning the timing models.

In order to operate correctly, all these approaches either *i)* assume that the exact speedpath corresponding to a delay measurement is available [2, 5], or *ii)* assume that it can be *isolated* by running logic and detailed timing analyses [1]. Hence, being able to isolate speedpaths or to associate silicon delay measurements to actual speedpaths is clearly a *cornerstone* in all these techniques. As a result, solving the silicon speedpath isolation problem is crucial to achieving all the other benefits that one could obtain from having silicon information.

Speedpath isolation is by no means a straightforward approach. As mentioned in [1], even with the most accurate logical and timing analyses, it may still not be possible to isolate the speedpath whose delay is the measured silicon delay. For example, if several candidate speedpaths have similar delays, and in the presence of dynamic effects and/or process variations, the authors in [1] acknowledge that their path isolation technique may fail to determine the correct speedpath. In such cases, direct probing techniques, such as laser probing [6], may be used to determine the actual speedpath, although they are usually expensive and undesirable to use.

In this paper, we propose a technique for silicon speedpath isolation. Our approach first considers, for every silicon delay measurement, a corresponding set of potential candidate speed-

*This work was supported in part by Intel Corporation.


*ICCAD'09,* November 2–5, 2009, San Jose, California, USA.

paths. Parameterized static timing analysis (PSTA) is then used to obtain a variational model for every candidate path. These variational models are then combined across the different sets of candidate speedpaths to create a cost function that weighs the different combinations of paths, with a *combination* being a collection of paths, each selected from a different set of candidate speedpaths. Using this PSTA-based cost function, we construct a branch and bound search tree and determine the top combinations of candidate speedpaths that are most likely to have resulted in the measured silicon delays. Our experiments show that the branch and bound approach to speedpath isolation is both efficient and accurate, even when silicon exhibits relatively large deviations from the pre-silicon PSTA models.

It is important to note up-front that we did not, in fact, have access to silicon measurements for purpose of this project. Instead, we have emulated the potential errors, variations, and unmodeled effects of silicon by software. In other words, we artificially perturb our timing models to create an alternate version of a circuit that is supposed to represent the true circuit on silicon, and we used the delay of that circuit as a "silicon measurement".

# 2. BACKGROUND

## 2.1 Silicon Debug and Speedpath Isolation

The authors of [1] give an overview of the process of silicon debug and how speedpaths are typically specified when silicon delay measurments are taken. In the process of silicon debug, input patterns are applied to the chip and the clock period is reduced until a failure is observed at one of the scan elements (latches or registers). When such a failure is observed, it is not clear which input vector is the one that caused it. Therefore the test is repeated for the clock period just larger than that at which the failure occured, only this time the clock is reduced for only one of the input vectors. This test is repeated until the vector responsible for the failure is identified. At this stage it becomes possible to run a functional simulation of the circuit to determine logic paths that are sensitized by this input vector. After determining this set of sensitizable paths the testers can further reduce this set by performing more accurate and detailed timing simulations. However, if two or more of the sensitizable paths have similar delays, and in the presence of process variability and other dynamic effects, such an analysis fails to determine the path responsible for the delay [1].

PSTA models provide an opportunity to use variability information, although not necessarily complete, to further reduce the number of paths with potentially speed-limiting behaviour. This is because one can make use of commonalities/differences in the variational models to favor/dismiss certain combinations of paths. This will be explained in detail in the next section.

## 2.2 PSTA Timing Models

All logic cell and interconnect delays are modeled as linear (strictly speaking, *affine*) functions of normalized process parameters, whose values vary between $-1$ and $+1$, as in [7, 8, 9]. Hence the delay of a logic cell or of an interconnect $RC$-tree can be written as an affine function of these process parameters. Because the delays of individual paths, in both clock networks and functional blocks, are sums of gate and interconnect delays, they also become such affine functions of the process parameters. Let the number of process parameters under consideration be $p$. Thus, a timing quantity $t$, representing a timing arc, interconnect, or path delay, is written as follows:

$$t = \bar{t} + \sum_{j=1}^{p} \delta_j X_j \tag{1}$$

where $X = (X_1, \ldots, X_p)$ is the set of normalized process parameters, $\bar{t}$ is the *nominal value* of $t$, and $\delta_j$ is its *sensitivity* to process parameter $X_j$. Such an affine function represents a *hyperplane* in $(p+1)$-dimensional space, and will often be referred to as, simply, a hyperplane.

However, *characterized* or *pre-silicon* timing models of logic cells, interconnect, and consequently logic paths are necessarily approximate and will deviate from silicon by some error. This can be attributed to various process, design and environmental effects. For one, it will always be the case that timing models will not account for certain physical effects that might be computationally too costly to model or to include in the timing flow. Also, some physical effects or logic cells might be mis-modeled, therefore resulting in errors. Furthermore, dynamic effects such as multiple input switching (MIS), cross-capacitive noise, and supply voltage droop are difficult to model and generally cause silicon delays to deviate from the predicted delay models. Therefore we write *actual* or *post-silicon* timing quantities representing delays of timing arcs, interconnect, and paths as:

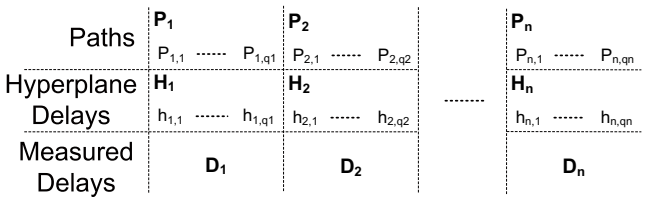


**Figure 1: Problem Setup**

$$t = \bar{t} + \sum_{j=1}^{p} \delta_j X_j + e \tag{2}$$

where the *silicon error* $e$ is a theoretically uncertain and unbounded term, which models the inaccuracies inherent in (1) relative to silicon. However, typically a lot of engineering effort goes into creating delay models, and although we consider these models to be approximate, they are still regarded as *good* approximations of silicon delays. As such, the effect of the error term introduced in (2) is in practice a relatively modest deviation of silicon delay around (1) rather than an unbounded effect that completely distorts the timing model.

# 3. OVERVIEW

## 3.1 Problem Definition

For a given sample chip $C$ at a particular silicon stepping, a set of delay measurements are obtained for each of the scan latches $\{L_1, \ldots, L_n\}$. Let the set of these delay measurements be $D = \{D_1, D_2, \ldots, D_n\}$, where $D_i$ is the *measured delay* at the scan latch $L_i$. As mentioned earlier, even with logic and detailed timing analysis, it may not always be possible to isolate or identify correctly *which* path in the fan-in cone of $L_i$ is responsible for the measured delay $D_i$. It is however possible to identify a "bin" of *candidate paths*, where one can be certain that *the true* speedpath, that is, the path that resulted in the measured delay $D_i$, is one of those candidate paths. Therefore, as inputs to our problem, we can associate with every scan latch $L_i$ and consequently every delay measurement $D_i$, a unique bin of $q_i$ potential candidate speedpaths $P_i = \{P_{i,1}, \ldots, P_{i,q_i}\}$.

In this paper, we propose a novel CAD solution to the *path isolation* problem, where the goal is to "filter" the bins of candidate paths and determine, by leveraging PSTA models, the true path that corresponds to the measured delay in every bin. In other words, we will use PSTA information about candidate paths to determine the most likely combination of "true" speedpaths $P^* = \{P_{1,t_1}, \ldots, P_{n,t_n}\}$, where $P_{i,t_i} \in P_i$ is the speedpath whose delay is the measured delay $D_i$, $1 \leq i \leq n$. We show next how this can be done.

In accordance with our timing model, every path $P_{i,k}$ has a hyperplane delay $h_{i,k}$. Therefore, every delay measurement $D_i$ is also associated with a bin $H_i = \{h_{i,1}, \ldots, h_{i,q_i}\}$ of hyperplane delays, corresponding to the bin of candidate paths $P_i$, as shown in Fig. 1. Each hyperplane delay includes a silicon error factor representing possible distortion to the model as seen in (2), and

therefore a candidate hyperplane $h_{i,k}$ is written as:

$$h_{i,k} = \bar{h}_{i,k} + \sum_{j=1}^{p} \delta_j^{(i,k)} X_j + e_{i,k} \quad (3)$$

As an illustration, suppose that we have measured a set of three delays $\{D_1, D_2, D_3\}$ corresponding to three hyperplane bins $H_1 = \{h_{1,1}, h_{1,2}\}$, $H_2 = \{h_{2,1}, h_{2,2}\}$, and $H_3 = \{h_{3,1}, h_{3,2}, h_{3,3}\}$. The graphs in Fig. 2 show possible characterized hyperplane delays of the paths in these bins along with the measured delay for each bin assuming a single variable $X_1$. Note that each of the hyperplanes deviates from silicon by an unknown value. Thus looking at the first bin in this graph, it is not possible to tell whether the measured delay $D_1$ results from $h_{1,1}$ or $h_{1,2}$. If the process point, here the value of $X_1$, *and the error terms were known*, then determining which path resulted in $D_1$ could be done in a straighforward way. We would simply find which hyperplane is the one that achieves the measured delay for the given process setting and silicon errors. However, for a chip such as our sample chip $C$, the particular process point $X^{(C)} = (X_1^{(C)}, \ldots, X_p^{(C)})$ is typically not known, and of course neither are any of the error terms $e_{i,k_i}$, $1 \le i \le n$, $1 \le k_i \le q_i$. This uncertainty prevents the identification of the *exact hyperplane* $h_{i,t_i} \in H_i$ responsible for each of the $n$ measured delays $D_i$, at the scan elements $L_i$, $1 \le i \le n$.

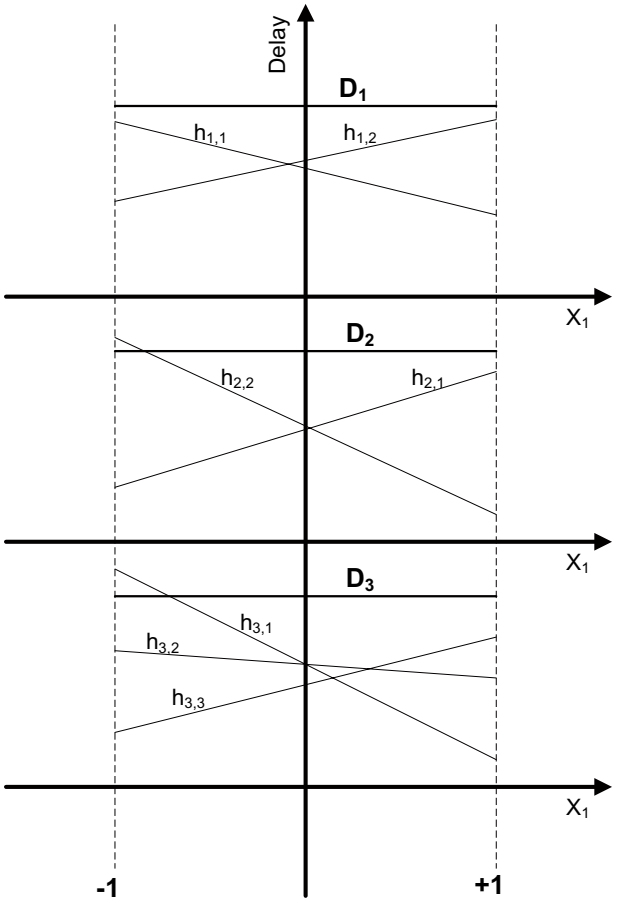


**Figure 2: Utilizing PSTA Information for Path Isolation**

In our analysis, the only available information is the silicon measured delays and the characterized hyperplane delays of the candidate speedpaths corresponding to those measured delays. Although characterized delay models are incomplete, as explained in Section 2.2, they still provide a valuable opportunity to make better informed decisions about which paths are responsible for measured delays. For example by looking at the second bin in Fig. 2, we can predict that hyperplane $h_{2,2}$ is more likely to be the speedpath in $H_2$ than $h_{2,1}$ since the measured delay can be achieved with *less distortion* to the hyperplane model. This prediction places the process parameter $X_1$ close to $-1$, and as a result, we can use this information in the first bin, and conclude that $h_{1,1}$ is the more probable speedath in $H_1$. This shows how we can combine information across different bins to *home in* on the most likely speedpaths.

Moving back to the general problem, let a *combination* or *selection* of paths be any set of $n$ paths such that each of the paths belongs to one of the $n$ path bins, e.g. $\{P_{1,k_1}, \ldots, P_{n,k_n}\}$ where $P_{i,k_i} \in P_i$. Recall that we would like to leverage the PSTA information to determine the most likely combination of true speedpaths $P^* = \{P_{1,t_1}, \ldots, P_{n,t_n}\}$, where $P_{i,t_i} \in P_i$ is the speedpath whose delay is the measured delay $D_i$. Now consider the arbitrary selection of paths $\{P_{1,k_1}, \ldots, P_{n,k_n}\}$. Given the process setting $X^{(C)} = (X_1^{(C)}, \ldots, X_p^{(C)})$ of the chip $C$, it is always possible to find an assignment $\{e_{1,k_1}, \ldots, e_{n,k_n}\}$ of errors, such that the delays of the paths in our arbitrary set become equal to the measured delays. Thus, with the introduction of the silicon error term in (2), it becomes possible for any such combination to be considered the "true" selection of speedpaths. As a result, we are not able to definitively exclude any combination from the possibility of being the true combination. Therefore, it becomes clear that we need to define some *ranking metric* to determine which combinations are most likely to be the true combination. In this work, we show how to determine this list of top $m$ combinations, without requiring any knowledge of the exact process point or silicon error values.

## 3.2 A Cost Function

Recall that the silicon error terms of candidate hyperplanes are deviations around the timing model, and therefore, for the "true" combination of speedpaths, these are expected to be small. Based on this premise, we define a *cost function* that determines the likelihood of any given combination of paths $\{P_{1,k_1}, \ldots, P_{n,k_n}\}$ to be the selection of "true" speedpaths. The cost $v$ is defined as the optimized objective function of the following quadratic program (QP):

$$\text{minimize: } v = \sum_{i=1}^{n} e_{i,k_i}^2$$

Subject to :

$$\begin{aligned} \bar{h}_{1,k_1} + \sum_{j=1}^{p} \delta_j^{(1,k_1)} X_j + e_{1,k_1} &= D_1 \\ \bar{h}_{2,k_2} + \sum_{j=1}^{p} \delta_j^{(2,k_2)} X_j + e_{2,k_2} &= D_2 \\ &\vdots \\ \bar{h}_{n,k_n} + \sum_{j=1}^{p} \delta_j^{(n,k_n)} X_j + e_{n,k_n} &= D_n \end{aligned} \quad (4)$$

This cost function evaluates the likelihood of a combination being the selection of "true" speedpaths by minimizing the norm of errors required to match each hyperplane with the measured delay of its bin at a **common** process setting. Consider the combination $S_1 = \{h_{1,1}, h_{2,2}, h_{3,1}\}$ in our example in Fig. 2. A visual inspection of Fig. 2 shows that by incurring a small error on each of these hyperplanes, we can make them intersect with their respective measured delays and that this can be done at a common process setting $X_1$ that is close to $-1$. Thus, the cost of such a combination as computed by (4) would be small. In fact, for a particular hyperplane combination, its cost is mainly affected by two factors. The first is how "far" each individual hyperplane is from its measured delay. Consider the combination $S_2 = \{h_{1,2}, h_{2,1}, h_{3,3}\}$.

Each of these hyperplanes is relatively "farther" away from its corresponding measured delay than the hyperplanes in $S_1$, and applying (4) on this combination will show that it not a likely combination. The second factor that affects the cost of a combination is whether "closeness" of candidate hyperplanes to measured delays is achieved for some common process setting, i.e. whether the paths of a combination are "compatible" with each other. Consider the combination $S_3 = \{h_{1,2}, h_{2,2}, h_{3,1}\}$, which only slightly varies from $S_1$. We see that taken individually, these hyperplanes are not far from their respective measured delays. However these can only be made equal to their measured delays at a low cost for very different process points, therefore the cost of $S_3$ will be much higher than that of $S_1$. This shows how our cost function can make use of the PSTA models and their interactions to favor or dismiss different combinations.

Using (4), the costs of all combinations of paths can be computed to determine the $m$ combinations most likely to be the true one. However, this would require running $(q_1 \times q_2 \times \ldots \times q_n)$ QP's. This exhaustive approach would be very expensive as the number of possibilities is exponential in the number of bins $n$. Therefore, in our approach we employ a branch and bound technique that allows us to search for the best $m$ combinations without having to list all possible solutions. Before discussing our method in detail, we will first give an overview of branch and bound and explain how this method can be applied to our problem.

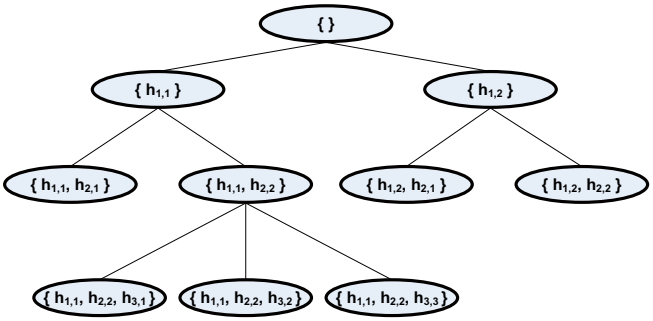


**Figure 3: A Search Tree**

# 4. A BRANCH AND BOUND FRAMEWORK

Branch and bound is a term used to describe a generic algorithm that is widely used to solve combinatorial optimization problems where the enumeration of all possible solutions is too expensive. Branch and bound works by building a search tree that explores the complete search space without having to visit every possible solution individually. Each node of this search tree represents a subset of the space of all possible solutions, and by branching a node, i.e. creating its children, this subset is further divided into a set of, typically mutually exclusive, smaller subsets. The root of the tree represents the space of all possible solutions. Branching occurs starting from the root creating nodes that represent subdivisions of the solution space, until the leaves, which represent individual or specific solutions, are reached. It is possible to construct a "full" tree, where the leaves enumerate all possible solutions, and where the optimal solution would be determined by finding the leaf with the minimum cost. However, even without considering the cost of building the search tree, such an approach would be just as expensive as the exhaustive enumeration of all possible solutions. Therefore, it is clear that in order for branch and bound to provide a runtime improvement over exhaustive enumeration, it would have to avoid building the complete search tree.

Such an improvement is only possible if one uses a *cost bounding function*. This is a function which gives a lower bound on the costs of all the descendants of a node. For an internal node, the knowledge of such a bound, combined with the availability of an *existing solution* with a cost that is lower than this bound, allows us to *prune* this node. That is, we can exclude the subtree rooted at this node from the search since all its leaves are guaranteed to have costs worse than that of the existing solution and will not be optimal. Obviously, the quality of the cost bounding function has a direct effect on the runtime improvment offered by branch and bound. If this function gives very conservative bounds, then few existing solutions will have lower costs. Similarly, the performance of branch and bound is dependent on the quality of existing or initial solutions. A "good" initial solution will have a lower cost, and thus it is likely to result in more pruned nodes.

In addition to the bounding function and initial solutions, other factors can also affect the performance of branch and bound. One such factor is the division strategy used to determine how the subspace represented by a node is divided among its children. For example, what part of the complete search space does each child of the root represent and how is each of these children further divided to create their child nodes. Another is the branching strategy adopted; for example would the nodes be created in a breadth first search (BFS) manner, a depth first search (DFS) manner, or some mixture of both. In our path isolation problem, we are not only interested in finding a single optimal solution, but rather the top $m$ optimal path combinations. Thus although our proposed method uses the concepts presented here, it also builds on them to achieve this aim. In what follows we present the main components that make up the branch and bound framework of our approach. We first discuss how a branch and bound search tree can be organized in order to represent the search space of all path combinations. After that, we present the cost bounding function used for node pruning. Then, we discuss some other choices for the rest of the factors that influence branch and bound performance such as branching strategies and finding initial solutions.

## 4.1 A Search Tree for Path Isolation

Consider the tree in Fig. 3, which corresponds to the hyperplane bins shown in Fig. 2 and presented in Section 3.1. In this example, an internal node with a *partial combination* such as $\{h_{1,1}, h_{2,2}\}$ represents all *complete combinations*, or simply combinations, which include $\{h_{1,1}, h_{2,2}\}$ as a subset. In our approach, the branch and bound search tree is such that at each level the search space is further subdivided by fixing a hyperplane from a specific bin. For instance, at level 2, the search space of each of the nodes of level 1 is divided into two search spaces, each restricted to one of the two hyperplanes that make up $H_2$. Thus in our division strategy, the division of a search space into subspaces at level $l$ is such that each of these only contains combinations that include a particular hyperplane from $H_l$.

## 4.2 A Cost Bounding Function

For our problem, the cost of a path combination can be found by applying (4). In order to apply a branch and bound solution we need a cost bounding function that can provide a lower bound on the costs of the descendants of an internal node. For an internal node with a partial combination $\{h_{1,k_1}, h_{2,k_2}, \ldots, h_{l,k_l}\}$, $l < n$, such a bound on the cost of all complete combinations which include this partial combination can be computed as follows:

$$\text{minimize: } v = \sum_{i=1}^{l} e_{i,k_i}^2$$

Subject to :

$$\begin{aligned}
\bar{h}_{1,k_1} + \sum_{j=1}^{p} \delta_j^{(1,k_1)} X_j + e_{1,k_1} &= D_1 \\
\bar{h}_{2,k_2} + \sum_{j=1}^{p} \delta_j^{(2,k_2)} X_j + e_{2,k_2} &= D_2 \\
&\vdots \\
\bar{h}_{l,k_l} + \sum_{j=1}^{p} \delta_j^{(l,k_l)} X_j + e_{l,k_l} &= D_l
\end{aligned} \tag{5}$$

For a complete combination which has $\{h_{1,k_1}, h_{2,k_2}, \ldots, h_{l,k_l}\}$ as a subset, the optimization in (5) is in effect an underconstrained version of (4). Therefore, it can be directly seen that (5) does provide a lower bound on the cost of the descendants of an internal node, and therefore it can be used as a cost bounding function. Note that applying (5) on a leaf node, i.e. a node at level $n$, gives the exact cost of that leaf. Thus (5) can be thought of as a generalization of (4) that can be applied to both internal nodes and leafs. For an iternal node, (5) gives a lower bound on the costs of the descendants of this node, whereas for a leaf it returns the exact cost. Therefore in our discussion the term *node cost* refers to both the exact cost, if the node is a leaf, and to the lower bound on the cost of the descendants of the node, if the node is an internal node.

## 4.3 Other Factors

So far we have explained how a basic branch and bound approach can be applied to our problem, and we have presented our cost bounding function. The order in which bins are considered has a direct effect on the performance of a branch and bound solution of path isolation, where this is considered to be part of the division strategy. We found that sorting the bins in non-decreasing order of number of hyperplanes produces better runtimes than a non-increasing or a random order. As for the branching strategy, we adopt a BFS strategy in the presentation of our work, although it is possible to use our proposed solution in conjuction with other branching strategies.

# 5. PROPOSED METHOD

Our approach consists of two distinct stages. The first stage builds a search tree using a greedy strategy to find a set of $m$ initial combinations. The optimality of these combinations is not guaranteed and the objective here is for these to be "good" initial solutions to be used in the branch and bound approach. The second stage of our approach uses the tree and the $m$ initial combinations to come up with the optimal top $m$ combinations. This stage tries to verify the ranking of the first stage, and if need be, "fixes" it to find the top $m$ combinations using a branch and bound strategy. In this section, a detailed explanation of both stages is presented.

## 5.1 Greedy Search

The first step in our branch and bound method is to find a set of $m$ initial combinations using a greedy search approach. In this step, a search tree is built such that at each level, the algorithm only keeps nodes with the lowest costs. This is done by pruning a pre-determined percentage of nodes at each level, where the nodes pruned are those with the highest costs. This greedy strategy is adopted in the hope that it will find $m$ leaves with relatively "low" costs. The pseudo-code of this stage is shown in the *GreedySearch* algorithm in Procedure 1 and is explained in detail below. However, we first present the scheme used to compute the number of nodes to prune at each level when executing this algorithm.

### 5.1.1 Computing Number to Prune

Recall that $q_1, \ldots, q_n$ represent the number of hyperplanes per bin. We would like to come up with a pruning scheme that *i)* results in at least $m$ leaves, and *ii)* is such that the precentage of nodes to be pruned increases with the level of the tree. The first requirement stems from the objective of the greedy search stage, which aims at coming up with $m$ "good" combinations but whose optimality is not guaranteed. After that, the second stage verifies and, if need be, corrects the results of this stage to guarantee optimality. On the other hand, the second requirement is set to improve the quality of the results of the first stage. This is because the costs of nodes deeper in the tree become more representative of the costs of their descendant leaves, and thus with our heuristic strategy it is less "risky" to prune nodes with the worst costs at deeper levels. One pruning strategy which can satisfy these requirements is to define at every level $l$, $1 \leq l \leq n$, the ratio of pruned nodes $b_l$ as:

$$b_l = 1 - r^{l-1} \tag{6}$$

where $r \leq 1$. This pruning ratio satisfies the second criterion that the percentage of pruned nodes increases with the level $l$. It still remains to satisfy the first criterion of ending up with at least $m$ leaves. This can be achieved by assigning an appropriate value to $r$ as follows. Let $T$ be the total number of possible path combinations, thus we have:

$$T = q_1 \times q_2 \times \ldots \times q_n \tag{7}$$

Since we are pruning a ratio of $b_l$ nodes at any given level $l$, the ratio of nodes preserved at each level is $r^{l-1} = 1 - b_l$. Thus at level $l$, the number of preserved nodes $T_l$ can be written as:

$$T_l = (1)(q_1) \times (r)(q_2) \times \ldots \times (r^{l-1})(q_l) \tag{8}$$

Thus $T_n$, the number of leaves preserved, can be written as:

$$T_n = (1 \times r \times \ldots \times r^{n-1}) \times T \tag{9}$$

Since we require the number of leaves to be $m$, we can write:

$$m = T_n = \left(r^{\frac{n(n-1)}{2}}\right) \times T \tag{10}$$

Therefore $r$ can be computed as:

$$r = \left(\frac{m}{T}\right)^{\left(\frac{2}{n(n-1)}\right)} \tag{11}$$

Thus, in order to compute the number of nodes to be preserved at any level $l$, we can compute $T_l$ as shown in (8). In case the value of $T_l$ does not turn out to be an integer then we simply take the ceiling of its value to be the number of nodes preserved.

**Procedure 1** GreedySearch($m$, $H$)

**Inputs:** `m` ← the number of desired combinations
`H` ← the set of `n` hyperplane bins
**Outputs:** `SearchTree`

```
1: Create the root of SearchTree;
2: Create first level of SearchTree;
3: Set CurrentLevel = 1;
4: while (CurrentLevel ≤ n) do
5:    Compute T_CurrentLevel, the number of nodes to preserve
      at CurrentLevel;
6:    Make T_CurrentLevel nodes with lowest costs live;
7:    Prune all other nodes at CurrentLevel;
8:    if ( CurrentLevel < n ) then
9:       Create children of live nodes at CurrentLevel, and com-
         pute their costs;
10:   end if
11:   CurrentLevel = CurrentLevel + 1;
12: end while
13: return SearchTree;
```

### 5.1.2 Proposed Algorithm

The *GreedySearch* algorithm employs the pruning strategy proposed above to construct a BFS tree that is used to come up with $m$ "good"initial combinations. This algorithm requires as inputs the number of required top combinations $m$, and the set of bins of hyperplanes $H$. As for the output of this procedure, *GreedySearch* returns *SeacrchTree*, the BFS tree it constructs.

Algorithm 1 first creates the root of the search tree and then adds the first level of the tree (lines 1–2). After that (lines 4–11), *GreedySearch* builds *SearchTree* in a BFS manner where at each level it preserves a number of nodes found using (8). For each level $CurrentLevel$, $1 \leq CurrentLevel \leq n$, the number of nodes to be preserved $T_{CurrentLevel}$, is first computed (line 5). Then, that many nodes with the lowest costs are kept, and the rest of the nodes at this level are pruned (line 6–7). After that, the children of all live nodes at this level are created and their costs are computed, unless these are at level $n$, i.e., unless these nodes are complete combinations or leaves (lines 8–9). Thus, by the time the loop (lines 4–12) exits at the last level with $CurrentLevel = n$, we would have preserved (at least) $T_{CurrentLevel} = T_n = m$ leaves as shown in (10). These leaves will be used as the $m$ initial combinations and will be returned along with *SearchTree* (line 13).

## 5.2 Optimizing the Top Candidates

The greedy search is followed by a verification step that provides the top $m$ **optimal** path combinations. This step considers the list of the top $m$ **known** combinations from the search tree of the greedy search step. It checks whether this list is optimal, and if not then it "revises" the combinations in the list in order to ensure optimality of the results. The psuedo-code of this stage is shown in the *GetTopCombinations* procedure shown in Procedure 2 and is explained in detail below.

As is the case in Procedure 1, Procedure 2 requires as inputs the number of required top combinations $m$, and the set of bins of hyperplanes $H$. In addition, it also requires the tree *SearchTree* constructed by *GreedySearch*. Procedure 2 returns the list of top $m$ **optimal** path combinations *TopLeaves*.

**Procedure 2** GetTopCombinations($m$, $H$, *SearchTree*)

```
Inputs: m ← the number of desired combinations
        H ← the set of n hyperplane bins
        SearchTree ← GreedySearch(m, H);
Outputs: TopLeaves ← array of m top combinations (leaves);
 1: TopLeaves = sorted top m leaves in SearchTree;
 2: CurrentRank = 1;
 3: while (CurrentRank ≤ m) do
 4:    CurrentCandidate = TopLeaves[CurrentRank];
 5:    CurrentLevel = 1;
 6:    while (CurrentLevel < n) do
 7:       for all CurrentNode ∈ Childless Nodes at CurrentLevel
          do
 8:          if (cost(CurrentNode) ≤ cost(CurrentCandidate))
             then
 9:             Make CurrentNode live;
10:             Create children of CurrentNode;
11:          else
12:             Make CurrentNode pruned;
13:          end if
14:       end for
15:       CurrentLevel = CurrentLevel + 1;
16:    end while
17:    for all CurrentLeaf ∈ Leaves(SearchTree) do
18:       if ((CurrentLeaf is not Don't Touch) and
          (cost(CurrentLeaf) ≤ cost(CurrentCandidate)))
          then
19:          CurrentCandidate = CurrentLeaf;
20:       end if
21:    end for
22:    TopLeaves[CurrentRank] = CurrentCandidate;
23:    Set CurrentCandidate to Don't Touch;
24:    CurrentRank = CurrentRank + 1;
25: end while
26: return TopLeaves;
```

Procedure 2 begins by storing the sorted list of the top $m$ **known** combinations in *TopLeaves* (line 1). After that, the ranked combinations in *TopLeaves* are adjusted so that they are guaranteed to be optimal. The algorithm iterates over the $m$ positions in *TopLeaves* (line 3–25) where for each rank *CurrentRank*, the combination is verified, and possibly "fixed", as follows. Let us consider the case when $CurrentRank = 1$, i.e., the case when Procedure 2 is finding the most favorable combination. On line 4, *CurrentCandidate* is set to the first combination currently stored in *TopLeaves*, which is the top combination found by *GreedySearch*. After that, a BFS traversal of *SearchTree* is carried out, where at each level the cost of each **childless** node is compared to the cost of *CurrentCandidate* (line 5–16). If this cost is less than that of *CurrentCandidate* then the node is made live and its children are created. On the other hand, if this cost is greater than of *CurrentCandidate*, then this node is pruned. Note that two types of **childless** nodes are considered here. The first type is that of nodes pruned by Procedure 1 according to the pruning heuristic used in our greedy search. Thus, in effect, Procedure 2 is checking the "legality" of these prunings. If *GreedySearch* had pruned a node, whose cost is less than that of any known leaf then this node should be "unpruned". The other type of nodes being considered are descendants of "unpruned" nodes, produced through the expansion of the tree that takes place on lines 9–10 when Procedure 2 determines that a node should be made live and creates its children. After the traversal of the tree is performed, the tree becomes such that the cost of any pruned node is guaranteed to be greater than the cost of *CurrentCandidate*. Therefore, the leaves of the tree must include any candidate combinations whose cost is less than that of *CurrentCandidate*, if any. Therefore, the leaves of the tree are re-examined on lines 17–21 to see if any of the leaves provide a better path combination than the one in *CurrentCandidate*. In addition to comparing the costs of the leaf *CurrentLeaf* and *CurrentCandidate*, Procedure 2 checks if *CurrentLeaf* is flagged as "Don't Touch" (line 18). The significance of this will be explained shortly, however for the case of $CurrentRank = 1$ no leaves are marked as such, and this loop simply finds the leaf with the lowest cost. After that, *TopLeaves*[*CurrentRank*], the combination ranked first, is updated to the optimal combination found after examining the leaves of the search tree.

The last step in *GetTopCombinations* is to flag the found path combination, or the leaf, as "Don't Touch" on line 23. This means that when Procedure 2 is finding the combination ranked second, the combination marked first will be flagged as "Don't Touch" and thus will not be considered as a candidate for this position. Similarly, when the algorithm is determining the $i^{th}$ ranked combination, it has already determined the combinations up to the $(i-1)^{th}$ rank, and has flagged these as "Don't Touch". Thus, the algorithm always finds the best "unranked" combination, and then flags it as "Don't Touch" and proceeds to find the next best combination.

# 6. EXPERIMENTAL SETUP

Our approach was tested on a selected set of circuits from the ISCAS-89 benchmark suite. These circuits were first synthesized and mapped to a 90nm CMOS library. They were then placed and routed, and the HSPICE netlist of each circuit was extracted. Nominal delays and slews were characterized for all cells in the library, and a set of 10 parameters $X_1, \ldots, X_{10}$ was selected to model process variations. The ranges of those parameters were chosen such that their combined effect on the delay or slew of a cell or an interconnect resistance or capacitance value is 20%. The sensitivities of gate delays and interconnect *RC*-trees to these process parameters were randomly generated.

For each of our test circuits, we designated a set of 15 registers

**Table 1: Ranks of True Combinations**

| ISCAS-89 Circuit | Mis-modeled Cell | | | Capacitive-coupling | | | Voltage Droop |
|---|---|---|---|---|---|---|---|
| | **Case** 1 Rank/Cut-off | **Case** 2 Rank/Cut-off | **Case** 3 Rank/Cut-off | **Varying** 20% Rank/Cut-off | **Varying** 30% Rank/Cut-off | **Varying** 40% Rank/Cut-off | 15% **Droop Effect** Rank/Cut-off |
| **s400** | 1 / 1 | 1 / 1 | 1 / 1 | 1 / 1 | 1 / 1 | 1 / 1 | 1 / 1 |
| **s5378** | 1 / 4 | 1 / 4 | 1 / 4 | 1 / 2 | 1 / 1 | 1 / 4 | 1 / 8 |
| **s9234** | 1 / 1 | 1 / 1 | 1 / 2 | 1 / 2 | 1 / 1 | 1 / 1 | 2 / 2 |
| **s15850** | 1 / 2 | 1 / 1 | 2 / 2 | 1 / 2 | 5 / 5 | 5 / 6 | 3 / 3 |
| **s38584** | 5 / 5 | 1 / 2 | 1 / 1 | 1 / 2 | 1 / 1 | 9 / 4 | 5 / 2 |

as the circuit's scan elements. For each test circuit, we generate "silicon" delay measurements at the inputs of its designated scan registers, and then we determine the characterized hyperplane delays of all candidate paths for each measurement. A C++ implementation of Procedure 1 and Procedure 2 is then applied on the delay measurements and the hyperplane bins to determine the combinations that are most likely to be "true".

## 6.1 Simulating Silicon

Our circuits were not implemented on silicon; instead, we use our extracted netlists in order to create a "silicon netlist". For a given circuit, this "silicon netlist" is derived from our extracted netlist as follows. First, a random process setting is chosen for the circuit. We then introduce errors to the delays of certain circuit elements in the "silicon netlist" in order to simulate the impact of some "silicon effect". In particular, for each circuit we produce "silicon netlists" that simulate three different "silicon effects": mis-modeled logic cells, capacitive-coupling, and voltage droop. Each of these effects is produced differently, and in what follows we present the details involved in reproducing these effects.

### *6.1.1 Mis-modeled Logic Cells*

Usually a mis-modeled logic cell is discovered if it is used frequently, since designers would catch on to the discrepancies between expected and silicon delays. However, problems generally arise when some less commonly utilized cell is mis-modeled due to characterization errors or an outdated model. In our tests, we simulate the effect of such a mis-modeled cell by choosing a specific cell and increasing the delays of all the instances of this cell in a given "silicon netlist" by 20%.

### *6.1.2 Capacitive-coupling*

Typically, capacitive-coupling between interconnect lines that are close to each other leads to a degradation in signal quality and higher delays. We recreate this effect by introducing a penalty of up to 5% on the delays of a certain number of randomly chosen interconnect RC-trees of a "silicon netlist".

### *6.1.3 Voltage Droop*

Voltage droop refers to a reduction in the supply voltage of a logic cell, which leads to an increase in the delay of this cell. This effect is produced by choosing logic cells in a "silicon netlist" randomly, and increasing their delays by up to 15%. Although the cells are chosen randomly, we make sure that enough cells are chosen so as to affect the "worst-case" delays at the inputs of all scan registers.

## 6.2 Measured and Hyperplane Delays

The availability of the "silicon netlist", where all delays are exact, allows us to find the "worst-case" delay at the input of each of the scan registers using static timing analysis. Therefore in our approach we take these "worst-case" delays to be our measured delays, and for each such delay we record the path that results in it. It then becomes possible to find the input vector or pattern that can sensitize it.

Recall that each "measured delay" is associated with a set of candidate speedpaths that is found using logic and detailed timing simulations. For our tests, the bin of candidate paths of a delay measurement, taken at the input of a scan register, is considered to be that of all paths that terminate at that input. That is, we do not try to trim the list of candidate paths by performing logic simulations. This will leads to more challenging cases to test our method on. However, after listing the candidate hyperplane delays of a given delay measurement we do remove those with very "low" hyperplane delays from the list. This is somewhat equivalent to performing timing simulations and removing paths that are impossible to be speed-limiting. This is done in order to avoid any artificial improvement of the quality of the results of our method. That is, keeping such paths in the lists of candidate paths would lead to cases were the number of combinations is large but where many of these combinations can be easily removed by our branch and bound approach.

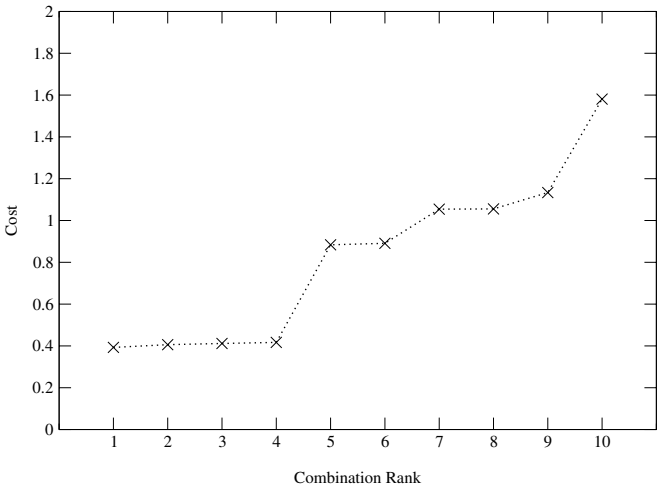


**Figure 4: Cost Vs. Combination Rank**

# 7. RESULTS

Table 1, shows results for all three types of test cases. For each test case, 15 (number of scan registers) delay measurements are taken and are fed, along with the bins of characterized hyperplanes of the candidate speedpaths, into Procedure 1 and Procedure 2 to identify the combinations most likely to be the "true" combination. Tests were run for a required number of combinations $m = 10$. For each test, two figures are reported. The first is "Rank", which here is the rank of the "true" combination in the top 10 combinations our method identified. The second figure, referred to as "Cut-off", describes another aspect of our results which is explained in detail below.

## 7.1 Predicting the True Combination

In each of our tests, we use a "silicon netlist" to measure de-

lays, and in each case record the "true" combination of paths that resulted in the delays. The recorded combination can then compared to the top combinations found using our method to check how well our approach ranks it. Consider the results for "mis-modeled cell" tests in Table 1. For each circuit, we ran tests for three different mis-modeled cells. Notice that in most cases, our method correctly identifies the "true" combination, and even when it doesn't, the rank of the "true" combination is always high. The next set of tests is that of "capacitive-coupling" tests. In this type of test, we provide three tests cases where in each case a different percentage of interconnect trees are affected by capacitive-coupling. Note, that for the case where only 20% of interconnect structures are affected, our method always finds the "true" combination. Our method still finds the "true" combination in most of the cases where 30% and 40% of the interconnect trees are affected. However, as we introduce more silicon errors to the circuit, the rank of the "true" combination falls sometimes. As for the "voltage droop" tests, we notice that in these cases the "true" combination is ranked very high, but not as high as in the other types of tests. This could be explained by the considerable delay penalty of 15% and the relatively large number of cells affected.

## 7.2 Cut-off Results

Consider the graph in Fig. 4 showing, for one of our test cases, the top 10 paths combinations and their respective costs found using our cost function in (4). This test was run on circuit s5378 where a capacitive-coupling delay penalty of up to 5% was imposed on 40% of interconnect trees. Although the top combination found by our method in this case was the "true" combination, it is clear from the graph that the top 4 combinations stand out for their low costs and that a significant jump in cost exists between the costs of the $4^{th}$ and the $5^{th}$ combinations. The "cut-off" numbers reported in Table 1, refer to such a rise in the cost of the combinations found. Thus in the case of the combinations seen in Fig. 4, the $4^{th}$ combination is the "cut-off". The value of this parameter allows engineers to evaluate the reliability of results reported by our method. Thus a "cut-off" of 1 means that a single combination stands out with a much lower cost than all other combinations. Therefore, it is less likely that another combination is the "true" combination and in this case the confidence in the results increases. On the other hand, a "cut-off" of 4 as seen in Fig. 4 means that it might be risky to completely ignore the possibility that the "true" combination is any one of these four combinations rather than the first one. However, it also turns out that, typically, all the combinations before the "cut-off" share a lot of their paths. For example, our top 4 combinations from Fig. 4 share 13 paths out of the 15 each is made up of. This leads to the conclusion that the 13 shared paths can be considered to be "true" with a high degree of confidence. Also, the only possibilities to be considered for the remaining two delay measurements are those seen in the top 4 combinations. Therefore, the "cut-off" value also allows the reduction of the search space for engineers and allows them to do a minimal amount of probing or process parameter measurement to try to determine the exact paths with higher certainty.

Now cosider the "cut-off" values reported in Table 1. In many cases we find that the rank of the "true" combination was 1 and that this combination was also the "cut-off". This means that engineers can proceed with the results with a high level of confidence. However, in some other cases we find that the "cut-off" can increase to 8. Nevertheless, in almost all of the tests, except for two, we always find that the rank of the "true" combination is within the "cut-off". Given the similarities between all the combinations within the "cut-off", this means that most of the paths of the "true" combination have been identified.

## 7.3 Runtimes

Table 2 shows the number of possible combinations for each circuit and the average runtimes for running our method on each of the circuits. Note that the runtimes presented are "processor times" and not "wall clock times". Due to the large number of possible combinations for these circuits, we do not present any runtimes of the exhaustive approach, however the runtimes of our method show that our branch and bound approach is quite efficient. Also note that the runtimes presented are not only dependent on the number of combinations. This can be attributed to the multitude of factors that affect the performace of any branch and bound technique.

**Table 2: Average Runtimes**

| ISCAS-89 Circuit | Number of Possible Combinations | Average Runtime |
|---|---|---|
| **s400** | 2,488,320 | 4.01 sec |
| **s5378** | 67,032 | 17.28 sec |
| **s9234** | 105,984,000 | 24.4 sec |
| **s15850** | 8,553,600 | 18.8 sec |
| **s38584** | 7,464,960 | 13.36 sec |

# 8. CONCLUSION

In this paper, we presented a new technique for silicon speedpath isolation, whereby the (pre-silicon) characterized variational models are used in order to identify the speedpaths resulting in silicon measured delays. We showed that the problem can be solved using a branch and bound approach that makes use of PSTA hyperplane delays and silicon delay measurements, in order to find the path combinations that are most likely to be the true speedpaths. Our results showed that these true speedpaths were accurately identified in most cases, and that this was achieved with efficient runtimes. This precise identification of speed-limiting paths will not only enable engineers to "fix" these failed paths, but more importantly, it will pave the way for accurate learning from silicon data in order to better "tune" the pre-silicon models.

# 9. REFERENCES


[1] K. Killpack, C. Kashyap, and E. Chiprout. Silicon speedpath measurement and feedback into EDA flows. In *DAC*, pages 390–395, 2007.

[2] P. Bastani, K. Killpack, L.-C. Wang, and E. Chiprout. Speedpath prediction based on learning from a small set of examples. In *DAC*, pages 217–222, 2008.

[3] D. Josephson and B. Gottlieb. The crazy mixed up world of silicon debug. In *Custom Integrated Circuits Conference, 2004. Proceedings of the IEEE 2004*, pages 665–670, 2004.

[4] L. Lee, L.-C. Wang, P. Parvathala, and T. M. Mak. On silicon-based speed path identification. In *IEEE VLSI Test Symposium*, pages 35–41, 2005.

[5] F. Koushanfar, P. Boufounos, and D. Shamsi. Post-silicon timing characterization by compressed sensing. In *International Conference on Computer Aided Design (ICCAD)*, pages 185–189, November 2008.

[6] M. Paniccia, T. Eiles, V.R.M. Rao, and W.M. Yee. Novel optical probing technique for flip chip packagedmicroprocessors. In *Test Conference, 1998. Proceedings., International*, pages 740–747, 1998.

[7] S. Onaissi and F.N. Najm. A linear-time approach for static timing analysis covering all process corners. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 27(7):1291–1304, 2008.

[8] K. R. Heloue, S. Onaissi, and F. N. Najm. Efficient block-based parameterized timing analysis covering all potentially critical paths. In *ICCAD*, pages 173–180, November 2008.

[9] 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, 2008.