# A Framework for Block-Based Timing Sensitivity Analysis

Sanjay V. Kumar
University of Minnesota
Minneapolis MN 55455

Chandramouli V. Kashyap
Intel Corporation
Hillsboro OR 97123

Sachin S. Sapatnekar
University of Minnesota
Minneapolis MN 55455

**ABSTRACT**

Since process and environmental variations can no longer be ignored in high-performance microprocessor designs, it is necessary to develop techniques for computing the sensitivities of the timing slacks to parameter variations. This additional slack information enables designers to examine paths that have large sensitivities to various parameters: such paths are not robust, even though they may have large nominal slacks and may hence be ignored in traditional timing analysis. We present a framework for block-based timing analysis, where the parameters are specified as ranges – rather than statistical distributions which are hard to know in practice. We show that our approach – which scales well with the number of processors – is accurate at all values of the parameters within the specified bounds, and not just at the worst-case corner. This allows the designers to quantify the robustness of the design at any design point. We validate our approach on circuit blocks extracted from a commercial 45nm microprocessor.

## Categories and Subject Descriptors
B.8.2 Performance Analysis and Design Aids

## General Terms
Algorithms, Performance, Design

## Keywords
Variations, Arrival times, Slacks, Pruning, Reordering

## 1. Introduction

Microprocessors are designed under the nominal or typical conditions where the process parameters, such as channel length ($L_e$), threshold voltage ($V_t$) etc, which affect the transistor drive strengths, and environmental parameters, such as supply voltage ($V_{dd}$), are assumed to be at fixed values. Unlike ASICs, which are designed under worst case assumptions, microprocessor designers have relied upon at-speed testing of manufactured parts to grade parts by frequency, with higher frequency parts selling at higher prices. However, due to increasing levels of parameter variations, as well as aggressive design styles striving for the best performance at the lowest power, designing at the nominal point causes surprises in silicon [2]. Often paths with large positive slacks turn out to be speed limiting in silicon. Therefore, from a designer's perspective, ordering paths by nominal slack, as is customary, does not provide a complete prioritization of paths to work on. For example, a representative slack[1] distribution of paths in a modern microprocessor is shown in Fig 1(a). Due to power performance tradeoffs, a steep timing "wall" is created, where a large number of paths have the same slack. When the first silicon arrives, it may so happen that the drive strength of certain kinds of devices – for example, low power devices – turns out to be lower than was assumed during design. As a result, paths that are more susceptible to variations in this device type are likely to show up as speed limiting in silicon, necessitating costly design re-spins.

This work was supported in part by the SRC under contract 2007-TJ-1572.

In the above example, however, if in addition to nominal slacks the slack *sensitivity* of paths to the drive strength of the low power devices were available, it would have been possible to fix paths that are very sensitive to the drive current variations of this particular device before tape-out. Here, by slack sensitivity we mean the change in slack for a given change in a parameter. For example, the slack sensitivity distribution of the same set of paths, when the drive current of all low-power devices is weaker by 20% is shown in Fig 1(b). Note that the steep timing wall now has a finite slope when viewed from a sensitivity perspective – paths that appeared equivalent in terms of slack appear different in terms of sensitivities. Thus, by taking into account the nominal slack and the sensitivity of the slack to parameters in conjunction with the amount of variations in the parameters – often specified as a range, rather than a statistical distribution – a more effective prioritization of paths to work on can be provided to the designer. A case for sensitivity analysis was also made in [4]. However, the paper does not provide any algorithmic details of how sensitivities can be propagated in a block-based manner in a non-statistical setting.

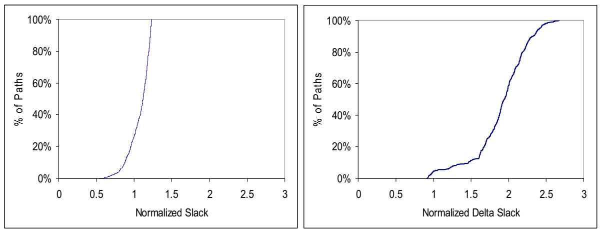

*Fig 1: cdf of top 1000 paths of a block showing: (a) nominal slack[1] (b) slack sensitivity when $I_d$ of all low power devices is 20% weaker*

In this paper, we propose a block-based algorithmic framework for solving the following problem: *Given a set of parameters and their ranges (bounds), compute accurate arrival times (slacks) for all settings of the parameters within the specified ranges in a single timing run.* This allows us to compute the arrival time (slack) sensitivity at any given point – within the range of variations – by simply querying for timing information in the neighborhood of the point. For instance, referring to Fig 1(b), this framework allows us to compute the change in slack – slack sensitivity – for any variation in drive current within the 20% bound.

It is generally accepted that block-based techniques have certain advantages over path-based methods – fast run times, incrementality and timing aware optimization, etc., – and is the method of choice in industrial timing analyzers. We distinguish our approach from block-based SSTA [1], [6] which assumes that the distributions and their correlations are known a priori. This is usually not the case, but the bounds or ranges of parameters are easier to obtain. Further, certain parameters such as $V_{dd}$ and Miller Coupling Factors (MCF) are not statistical in nature, and are therefore naturally described in terms of ranges.

[1] Slack values are normalized with respect to the FO4 delay of an inverter, throughout the paper.

Recently, a block-based static timing algorithm that also works with parameter ranges was presented in [3]. The primary goal of that work was to preserve the accuracy at only the worst-case corner. To achieve this goal, the arrival time sensitivities to the parameters were adjusted during the output arrival time computation. As a result, the arrival times at non-worst case settings were not accurate (we show this in Table 7 of Section 4). Our goal in this work is different: we wish to compute accurate timing at all points, not just the worst case point. This enables us to compute the sensitivity of a timing parameter by evaluating the timing at two different points in the parameter space and computing the change in the timing quantity.

This paper is organized in four sections as follows. In the next section we establish some notation and background for our method, with comments on a couple of papers relevant to our work. In Section 3, we introduce the various algorithms for propagating the arrival times through the circuit. We present experimental results in Section 4 based on a 45nm commercial microprocessor design.

## 2. Preliminaries

The timing graphs for an inverter and a two-input NAND gate are shown in Fig 2. The label on the edge is the delay of the input-output transition of the gate. We also associate the notion of an arrival time at every node in the graph. For a single input gate, such as an inverter, as shown in Fig 2(a), the output arrival time $A_2$ is given by:

$$A_2 = A_1 + d_{12} \tag{1}$$

where $A_1$ is the input arrival time and $d_{12}$ is delay of the arc between 1 and 2. For a two input NAND gate shown in Fig 2(b), the output arrival time $A_3$ is given by:

$$A_3 = \max(A_1 + d_{13}, A_2 + d_{23}) \tag{2}$$

which generalizes in an obvious way for gates with more than two inputs. A timing path is a sequence of nodes such that a delay arc exists between two successive nodes in the sequence. The first node in the sequence is the input node of the path and the last node in the sequence is the output node of the path.

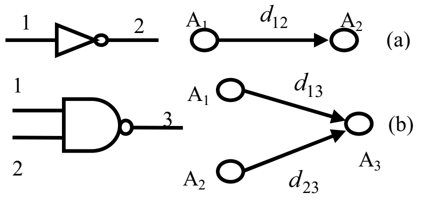

*Fig 2: Timing graph of (a) inverter, and (b) NAND gate*

Suppose the delay of a gate depends on $n$ parameters. Assuming a first order variation model, the delay can be written as:

$$d = \overline{d} + \sum_{i=1}^{n} a_i X_i \tag{3}$$

where $a_i$ are the delay sensitivities and $-1 \le X_i \le 1$ with $i$=1,...,$n$. We refer to (3) as the delay *hyperplane*. We assume that the physical parameters such as L, $V_{dd}$, etc. have been transformed into the abstract parameters $X_i$ by means of the affine transformation as described in [3],[5]. We use **X** to denote the set of points in the hypercube defined by $-1 \le X_i \le 1$. Since the arrival time at the output node of a timing path is simply the summation of delays along the path, we express the arrival time $A$ at the output node, as the arrival time hyperplane:

$$A = \overline{A} + \sum_{i=1}^{n} b_i X_i \tag{4}$$

where $\overline{A}$ is the nominal arrival time. Thus, the arrival time at the output node of a path has a simple representation that faithfully captures the sensitivity of the path to the parameters, given by the $b_i$ terms in (4).

However, such a simple linear representation of arrival times is not helpful for computing sensitivities when many paths converge on a node. Consider the scenario depicted in Fig 3(a) which shows four different paths with different arrival times (denoted as $A_1$, $A_2$, $A_3$, and $A_4$ respectively), and different sensitivities to some parameter $X_i$. If these four paths converge at the same node, the arrival time at that node is given by the maximum of the arrival time of the four paths (generalized from (2)) and unlike (4), the arrival time is a nonlinear function of the parameters. The maximum arrival time at the nominal value of $X_i$ is given by path 2 (hyperplane $A_2$). However, if the parameter changes by -0.5, path 1 is the dominant path whereas if the parameter changes by +0.7, path 3 is dominant. Therefore, we need a representation of an arrival time that is faithful to the fact that different paths dominate for different settings of the parameters, and as a result gives the correct arrival time for any setting of the parameters.

One representation of the arrival times in the presence of a max function is to use a *bounding* hyperplane as shown by the dotted line in Fig 3(b) (in one dimension it is a line) [3]. The authors propose an algorithm that provides a tight upper bound on the worst case arrival time value at the node. The advantage of this method is that the representation remains linear and canonical, and that it ensures that the worst case delay of the circuit is an upper bound on the true delay of the circuit. However, as mentioned in the introduction, in microprocessor design we are interested in the sensitivities around a design point rather than the worst case delay. As the figure shows, the bounding hyperplane is significantly inaccurate at non-worst case settings of the parameters, particularly around the nominal value. Further, no information as to which path is dominant and under what conditions is provided. Additionally, since the method artificially raises the delay hyperplanes during the MAX computation, the sensitivity of the delay with respect to the parameters is not preserved.

However, if we relax the requirement that we propagate only a single arrival time hyperplane, then a *piecewise-planar* representation of the MAX function is possible (see Fig 3(c)). We allow a *set* of hyperplanes such that each can be a maximum hyperplane for some setting of the parameters within the allowed ranges. Intuitively, each hyperplane represents a path up to that node in the timing graph.

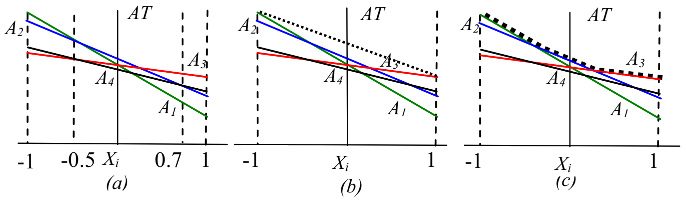

*Fig 3(a) MAX arrival time (AT) of 4 paths using: (b) a bounding hyperplane using [3], and (c) an exact piece-wise planar representation*

More formally, for a set of $m$ arrival time hyperplanes given by:

$$A_j = \overline{A_j} + \sum_{i=1}^{n} b_{ji} X_i, j = 1, \cdots, m \tag{5}$$

We say that a hyperplane $A_j$ is *prunable* if and only if:

$$\max_{i=1}^{m} A_i = \max\ (A_1, \cdots, A_{j-1}, A_{j+1}, \cdots, A_m) \tag{6}$$

For example, in Fig 3(c), $A_4$ can be pruned because it does not contribute to the MAX function (shown by the dotted line). A set of hyperplanes is called *irreducible* if no hyperplane in the set is prunable. Applying this definition to the four paths in Fig 3(a), we see that the three hyperplanes $\{A_1, A_2, A_3\}$ form an irreducible set and the MAX is shown by the dotted line in Fig 3(c) which forms a piece-wise linear representation (piece-wise planar in higher dimensions). While this raises the possibility of exponential blowup in the number of hyperplanes under the worst case, since every path (there are exponential number of paths in the worst case) could become critical at some setting of the parameters, we show in Section 4 that this is not the case on practical industrial designs. In the next section, we also describe an algorithm that trades accuracy for run-time and avoids the potentially exponential run-times.

We mention in passing that recently in [5], a branch and bound method was proposed to compute the exact worst case path delay using the method of [3] to provide the bounds needed to prune the search space. While the method could be adapted to compute the circuit delay at any settings of the parameters, it involves searching through the path space any time the parameter setting changes. Further, the run time depends on the quality of the upper bounds. As shown in Fig 3(b), the upper bound can be very loose at non-worst case settings of the parameters, particularly at the nominal corner. Finally, this method does not explicitly provide us with a technique for determining under what conditions a particular path could become the most critical, something that is useful for the designers to know.

## 3. Propagation of Arrival Times

In this section, we describe techniques to propagate and prune the arrival times on the timing graph. The basic operations of static timing are the SUM and the MAX. Considering the inverter in Fig 2(a), the SUM operation is defined as:

$$\mathbf{A_2} = \{A_j + d_{12} \mid A_j \in \mathbf{A_1}\} \qquad (7)$$

where $\mathbf{A_1}$ is the set of input hyperplanes and $\mathbf{A_2}$ is the set of output hyperplanes. In the rest of this section, we focus our attention on the more complex MAX operation[2].

Given a set of arrival times at the inputs of a gate, the arrival time at the output of the gate is the union of the sets of arrival times at the inputs. We refer to the set of hyperplanes at the output node as $\mathbf{U}=\{A_1,..,Am\}$. We need to perform a pruning operation on $\mathbf{U}$ to determine $\mathbf{A} \subseteq \mathbf{U}$ (ideally $\mathbf{A}$ would be irreducible). Such a pruning operation is necessary in order to ensure that the number of hyperplanes on every node does not increase exponentially as we perform a forward propagation along the timing graph.

### 3.1 Pairwise Pruning Algorithm

We first describe a simple algorithm that compares the arrival times in a pairwise manner. Given two hyperplanes $A_1$ and $A_2$, we write $A_2 \prec A_1$ if $A_1 - A_2 \geq 0$ for all values of $X$ in $\mathbf{X}$. That is (using (4)):

$$\left(\overline{A_1} - \overline{A_2}\right) + \left(\sum_{i=1}^{n}(b_{1i} - b_{2i})X_i\right) \geq 0 \qquad (8)$$

Since the parameters have all been normalized to lie between -1 and +1, (8) is always true if:

$$\left(\overline{A_1} - \overline{A_2}\right) \geq \left(\sum_{i=1}^{n}(|\, b_{1i} - b_{2i}\,|)\right) \qquad (9)$$

Note that given two hyperplanes $A_1$ and $A_2$, we have one of:

1. $A_1 \prec A_2$ in which case we prune $A_1$,
2. $A_2 \prec A_1$ in which case we prune $A_2$,
3. neither $A_1 \prec A_2$ nor $A_2 \prec A_1$ and we keep them both.

[2] The algorithms described in the paper can be adapted for the MIN operation in a straightforward manner.

Given a set of hyperplanes $\mathbf{U}=\{A_1,..,A_m\}$ the pruning algorithm is outlined in Fig 4. The PAIRWISE algorithm begins with the set $\mathbf{U}$, and initially assumes that all the hyperplanes in $\mathbf{U}$ are non-prunable. A hyperplane in $\mathbf{U}$ is compared against all other non-prunable hyperplanes in $\mathbf{U}$ and if it is not prunable, it is added to the set $\mathbf{A}$. The overall run-time for this operation is O ($m^2$).

```
PAIRWISE(In:U;Out:A)
//U={A1,…,Am}
A={};
Mark all hyperplanes non-prunable;
for i=1:m
  if (Ai is marked pruned) continue;
  for j=1:m
    if (Aj is marked pruned) continue;
    if( $A_j \prec A_i$ ) mark Aj pruned;
  end  //for j=1:m
end    //for i=1:m
Add all non-prunable hyperplanes to A
```

*Fig 4: PAIRWISE pruning algorithm*

### 3.2 Necessary and Sufficient Condition for Pruning

The PAIRWISE algorithm does not guarantee that the set $\mathbf{A}$ is irreducible. This can be illustrated by Fig 3 where the PAIRWISE algorithm does not mark $A_4$ as non-prunable since (9) does not hold for any hyperplane that is compared with $A_4$. Therefore, the condition described in (9) to prune a hyperplane is sufficient but not necessary. In order to determine if a hyperplane in $\mathbf{U}$ is prunable or not, it must be simultaneously compared with *all* other non-prunable hyperplanes in $\mathbf{U}$. Thus, to determine if a hyperplane $A_j$ in $\mathbf{U}$ can be pruned, the following condition must be satisfied:

$$\begin{gathered} A_j \geq A_k \forall k = 1,...,m, k \neq j \\ -1 \leq X_i \leq 1, i = 1,\cdots,n \\ \textit{has no feasible solution} \end{gathered} \qquad (10)$$

(10) is a *necessary* and *sufficient* condition to determine if a hyperplane can be pruned. The FEASCHK algorithm that performs pruning based on (10) is shown in Fig 5. Since feasibility checking is done by all LP solvers, we use the commercial optimization package CPLEX [6] which performs feasibility check efficiently. We note that the algorithm is inherently parallelizable since the feasibility check for each hyperplane can be performed in parallel, if a multi-processor machine were available. Also, unlike the method in [5], FEASCHK can be easily adapted to find a point in $\mathbf{X}$ where a given hyperplane (path) is non-prunable (critical).

```
FEASCHK (In:U;Out:A)
//U={A1,…,Am}
A={};
Mark all hyperplanes non-prunable;
for j=1:m
  if (Aj is marked pruned) continue;
  formulate (10), check for feasibility;
  // only include Ak not marked pruned
  if(solution to (10) is feasible)
      A= A U Aj;
  else
      mark Aj as pruned
end    //for j=1:m
```

*Fig 5: FEASCHK pruning algorithm*

While the feasibility checking can be done in time polynomial in the size of (10), the theoretically exponential number of hyperplanes that are possible at a node makes this exponential.

However, as we show in the next section, FEASCHK is in practice efficient on realistic circuits.

In order to optimize the run-time spent in determining the set of hyperplanes to propagate, FEASCHK algorithm can be applied selectively. If the number of hyperplanes on the node exceeds a certain user-specified threshold, we apply FEASCHK; else the PAIRWISE algorithm is used to prune the hyperplanes. This implies that some redundant hyperplanes that can be pruned are carried forward, until the threshold is reached. This algorithm is denoted as PAIRWISE_FEASCHK_THRESH.

## 3.3 Exploring Run-time Accuracy Trade-offs

While the algorithms described in the previous subsections are exact, the run-time depends on the nature of the logic cone and the number of parameters considered. In the worst-case, the run-times could be exponential since exponential number of hyperplanes may be carried. We now describe two methods which trade-off accuracy for runtime.

### *3.3.1 A Soft-Pruning Strategy*

As explained in the previous section, the PAIRWISE pruning strategy can often be inefficient, leading to an exponential blow-up of the number of non-prunable hyperplanes, if most of them cannot be pruned using (8). Instead, if we relax (8) as follows, then additional pruning can be achieved:

$$\left(\overline{A_1}-\overline{A_2}\right)+\left(\sum_{i=1}^{n}\left(|\,b_{1i}-b_{2i}\,|\right)\right)\geq \varepsilon,\varepsilon<0 \qquad (11)$$

Intuitively, this implies that we mark $A_2$ as prunable even if it can exceed $A_1$ by a small amount ε. In order to account for the fact that pruning $A_2$ may lead to an inaccurate timing estimate at some setting of the parameters, we *raise* hyperplane $A_1$ by increasing its nominal arrival time $\overline{A}_1$, by the minimum amount required for (8) to be satisfied. This not only allows us to prune $A_2$ but may also allow several other planes to be pruned by the raised hyperplane of $A_1$, thereby considerably decreasing the number of hyperplanes that need to be propagated.

There is a tradeoff between the number of hyperplanes pruned and the pessimism in the actual arrival time numbers due to raising some of the hyperplanes, based on the value of ε. However, in practice, a small value of ε, (-0.5% of $\overline{A}1$ based on our experiments) provides a considerable speedup without significantly over-estimating the arrival times. In our implementation, we use this idea of a *soft threshold* for pairwise-pruning our hyperplanes as a preprocessing stage to reduce the cardinality of **U**, before applying FEASCHK. Each hyperplane is allowed to be raised at most once during PAIRWISE pruning if (11) is true but (8) is still false. This algorithm is referred to as PAIRWISE_SOFT_PRUNE_FEASCHK and the results are shown in Section 4.2.

### *3.3.2 Shrinking Hypercube Method*

We now describe a method that allows accuracy to be traded-off for run-time by limiting the number of hyperplanes carried. This idea is explained in Fig. 6 where four hyperplanes in an irreducible set **A** are shown. $X_i$ is in [-1, 1] as before. However, if $X_i$ is restricted to lie in [-0.5, 0.5], $A_1$ and $A_4$ are prunable as shown in (b), while at the nominal value of $X_i$ (interval size is zero) $A_1$, $A_3$, and $A_4$ are all prunable. Thus, by reducing the size of the range of parameters, fewer hyperplanes can be propagated.

Hyperplanes that are not prunable outside the range are replaced with a bounding hyperplane using the method of [3]. While this method is pessimistic outside the reduced range of $X_i$, it is faster since fewer hyperplanes are propagated. Further, by preserving accuracy within the reduced range which is centered on the nominal point, the arrival times around the nominal are still calculated accurately.

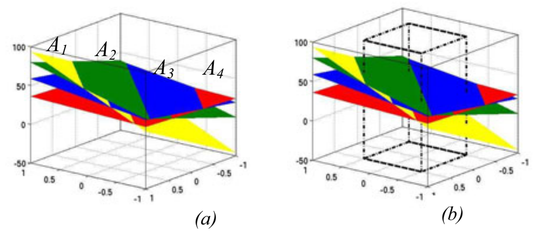

*Fig 6: (a) Four non-prunable hyperplanes ($A_1$, $A_2$, $A_3$, $A_4$) (b) Shrunk hypercube for the four planes in (a), (given by $-0.5 \le X_i \le 0.5$) such that only $A_2$ and $A_3$ are non-prunable*

More formally, for every node we have a triple consisting of the non-prunable hyperplanes, the hypercube where the hyperplanes are non-prunable, and a bounding hyperplane which is an upper bound of all the hyperplanes at that node. Consider an *m*-input gate: At the $i^{th}$ input we have the triple: <$\mathbf{A}_i$, $\mathbf{X}_{ai}$, $B_i$> where $\mathbf{A}_i$ is an irreducible set, $\mathbf{X}_{ai}$ is the set of points in the reduced hypercube given by $-a_i \le X_j \le a_i$ $(0 \le a_i \le 1,\ j=1,\ldots,n)$ and $B_i$ is the bounding hyperplane. To compute the triple at the output, we start with the initial triple <**U**, $\mathbf{X}_0$, $\mathbf{U}_b$> where $\mathbf{U} = \mathbf{A}_1 \cup \cdots \mathbf{A}_m$, $\mathbf{X_0}$ is the smallest hypercube from the inputs, and is given by $\mathbf{X}_0 = \mathbf{X}_{a1} \cap \cdots \mathbf{X}_{am}$, and $\mathbf{U}_b = \{B_1, \cdots, B_m\}$. We prune **U** such that the number of non-prunable hyperplanes is less than the user specified threshold. We do this by iteratively shrinking $\mathbf{X_0}$ if necessary (by some delta), as shown in the algorithm SHRINK_HYPERCUBE in Fig 7. We also compute the bounding hyperplane $B$ on $\mathbf{U_b}$, using [3].

```
SHRINK_HYPERCUBE(In:U,X0,Ub,N;Out: A,X,B))
//U={A1,…,Am}, Ub={B1,…,Bm}
//N = maximum number of non-prunable
  hyperplanes allowed
//X0 = Initial hypercube
Apply FEASCHK algorithm with bounds on X from
   X0, to obtain the irreducible set of A.
while (size(A) > N)
  Shrink hypercube X0 by delta
  Apply FEASCHK with new bounds on X to
  obtain the new irreducible set of A.
end
Compute bounding hyperplane B on Ub
```

*Fig 7: SHRINK_HYPERCUBE algorithm*

The SHRINK_HYPERCUBE method is equivalent to FEASCHK when **X** is [-1,1], and reduces to the method in [3] if $N$ = 1. This algorithm can be extended to shrink each dimension by different amounts and to also use binary search in the *while* loop in Fig 7.

## 4. Results

In this section, we present the simulation results obtained on a 45nm based commercial microprocessor design. Global variations in four different parameters types, namely supply voltage ($V_{dd}$), Miller Coupling Factor (MCF), channel length of NMOS transistors ($L_n$), and channel length of PMOS transistors ($L_p$), are considered. $L_n$ and $L_p$ are each divided into two different types, based on whether the device is nominal or low power, and further into three types based on layout dependent information. Each of the individual L parameters is assumed to vary independently of each other, thereby resulting in 14 different parameters (12 for L, MCF, and $V_{dd}$). The ranges of these parameters are shown in Table 1.

*Table 1: Range of variations for parameters*

| Parameter (total of 14) | Range of Variations |
|---|---|
| $V_{dd}$[3] | 0 to -18% |
| $L_n$ and $L_p$ (12 different types) | ±10% |
| MCF | ±33% |

The bounds on these parameters are provided as an input to the timing engine. We found that the delay is linear in the parameter variations within these ranges. The library characterization flow has been enhanced to compute the delay sensitivities on all timing arcs with respect to each of the above parameters as a function of input slopes and output loads. The pruning algorithms described in Section 3 are applied on four different design blocks. Table 2 presents information about the benchmark circuits. The timing engine is implemented in C++, with an interface to CPLEX [7], to perform FEASCHK pruning. The arrival times are computed for RISE and FALL transitions at the MAX and MIN modes as is typical in a static timing tool.

*Table 2: Benchmark information*

| | Block1 | Block2 | Block3 | Block4 |
|---|---|---|---|---|
| # of registers | 623 | 1086 | 2510 | 1021 |
| # of nodes | 21425 | 22384 | 40972 | 50599 |
| # of timing arcs | 14143 | 10044 | 16879 | 46647 |

## 4.1 Run-time Comparisons

The run-times for performing a forward propagation on the timing graph computing the set of irreducible arrival time hyperplanes on every node are shown in Table 3. The run-time numbers are relative to the timing run where a single parameterized hyperplane (the hyperplane with the largest (smallest) arrival time at the nominal point for MAX (MIN) analysis) is propagated. The results indicate significant difference between the run-times on Block1 versus the other blocks. As shown in Fig 8, this is because there are a large number of reconvergent paths in Block1 and consequently a larger fraction of nodes that contain 100 or more hyperplanes. It is also interesting to see that, for Block1, PAIRWISE takes an order of magnitude more runtime than FEASCHK although the complexity of PAIRWISE is less than that of FEASCHK, which can be explained as follows.

*Table 3: Run-times (relative to nominal) with 14 parameters*

| Method | Block1 | Block2 | Block3 | Block4 |
|---|---|---|---|---|
| PAIRWISE | - [4] | 1.2x | 1.15x | 1.69x |
| FEASCHK | 14.11x | 1.67x | 1.40x | 1.74x |
| PAIRWISE_FEASCHK_ THRESHOLD (N=50) | 14.44x | 1.2x | 1.16x | 1.76x |

Since PAIRWISE is a sufficient but not a necessary condition for pruning, it carries forward a significant number of prunable hyperplanes, which has a cascading effect as the hyperplanes are propagated through the circuit. FEASCHK on the other hand does more work to find truly prunable hyperplanes at every node and the number of hyperplanes it carries forward is therefore significantly reduced. For example, there were 93 nodes that had more than 1000 hyperplanes with PAIRWISE whereas there were only 15 such nodes with FEASCHK.

To summarize, our experiments on the four blocks indicate that PAIRWISE_FEASCHK_THRESH provides significantly better run-time performance over the PAIRWISE method for circuits with large number of hyperplanes (as seen in Block 1). At the same time it performs better than FEASCHK on Blocks 2-4. *Thus, the runtime of these methods is very dependent on the topology of the circuits.* Since Block1 has a large number of equally critical paths, we focus on that block in the rest of the section.

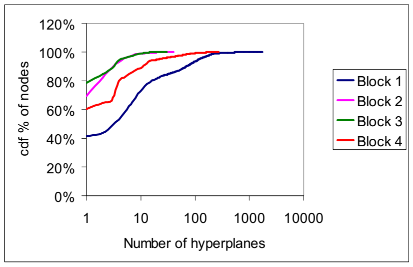

*Fig 8: cdf of the number of hyperplanes on the four blocks for MAX operations, performed using FEASCHK*

## 4.2 Approximate Methods

In order to explore run-time accuracy trade-offs, the SHRINK_HYPERCUBE method, described in Section 3.3.2 is applied on Block 1 for different values of N, where N denotes the maximum number of hyperplanes that can be propagated at every node. The run-times and the smallest size of the hypercube, computed across the inputs of the 623 sequential gates in the design, for the case of 14 parameters, are shown in Table 4.

*Table 4: SHRINK_HYPERCUBE method on Block 1*

| Number of hyperplanes allowed | Run-time relative to FEASCHK | Size of hypercube |
|---|---|---|
| 50 | 0.69x | 0.25 |
| 100 | 0.74x | 0.25 |
| 200 | 0.83x | 0.50 |
| 400 | 0.93x | 0.75 |
| 800 | 0.99x | 0.75 |

A step-size of 0.25 is used to shrink the hypercube in each iteration of the while loop in Fig 7. The run-times are compared with respect to the FEASCHK run-time in Table 3. The results indicate a good trade-off between the run-times, the size of the hypercube (denoted in Section 3.3.2 by $a$, where $-a \le X_j \le a, j = 1, \cdots, n$), and the maximum number of hyperplanes allowed (N).

*Table 5: Distribution of the hypercube size in Block 1*

| Size of Hypercube | No. of cones | Cumulative % |
|---|---|---|
| 0.25 | 2 | 1.19% |
| 0.5 | 2 | 2.38% |
| 0.75 | 51 | 3.28% |
| 1 | 1619 | 100.00% |

A cdf of the size of the hypercube for each of the timing cones in Block 1 is shown in Table 5, for the case where N was set to 100, in the SHRINK_HYPERCUBE algorithm. The results indicate that more than 95% of the timing cones have a hypercube of size 1, implying less than 100 hyperplanes on them, and hence the arrival times computed on all these cones are exact, for any setting.

To further explore run-time accuracy trade-offs, the PAIRWISE_SOFT_PRUNE_FEASCHK algorithm, explained in Section 3.3.1 was applied on Block 1. The results are compared

[3] The normalized parameter for $V_{dd}$ is assumed to vary from [-1,0] with -1 representing the case where $V_{dd}$ is at -18%. The pruning algorithms are modified to handle this special case, accordingly.

[4] This run did not finish due to insufficient memory.

with FEASCHK algorithm in Table 6. The table shows a reduction in the maximum number of hyperplanes on a node by a factor of three, when compared with FEASCHK. Accordingly, a 33% speedup over FEASCHK is obtained at the expense of a small overestimation (maximum of 1.6%) in the nominal arrival times.

*Table 6:PAIRWISE_SOFT_PRUNE_FEASCHK on Block 1*

| | FEASCHK | PAIRWISE_SOFT_ PRUNE_FEASCHK |
|---|---|---|
| Run-time | 14.11x | 9.58x |
| Number of hyperplanes on the largest cone | 948 | 382 |

## 4.3 Slack Computation

We briefly explain how we compute the slacks at the inputs of all registers in the blocks. Consider the cone shown in *Fig 9*. In our framework, the arrival times at the data and clock inputs of the sampling register are irreducible sets of hyperplanes denoted as $\mathbf{A_d}$ and $\mathbf{A_c}$, respectively. The required arrival time at the data input of the sampling register is given by:

$$\mathbf{R_d} = \{A_{j_c} + T - S \mid A_{jc} \in \mathbf{A_c}\} \qquad (12)$$

where $T$ is the cycle time and $S$ is the setup time. We do not consider setup time variations in this work. On all our benchmarks, the cardinality of $\mathbf{A_c}$ was one since there was no fanin in the clock network. The margin at the data input of the register is given by:

$$\mathbf{M_d} = \{R_{id} - A_{jd} \mid A_{jd} \in \mathbf{A_d}, R_{id} \in \mathbf{R_d}\} \qquad (13)$$

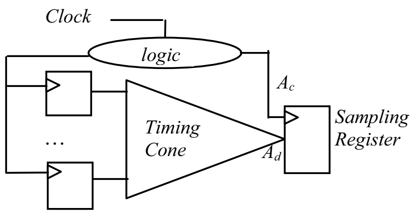

*Fig 9: A timing cone*

Thus, (13) can be computed in $O(|\mathbf{A_d}|.|\mathbf{R_d}|)$ time. $\mathbf{M_d}$ may be pruned further using the techniques of Section 3.

In order to evaluate the sensitivity of the slack of the various paths to parameter variations, we first compute the set of *irreducible slack hyperplanes* at the data input of each of the registers on Block 1, for the case of 14 parameters. We now consider the cone with the highest number of irreducible slack hyperplanes on Block 1 (consisting of 948 hyperplanes). Table 7 shows the slacks (computed as a minimum of the margins of the 948 irreducible hyperplanes) at different settings of the parameters, (none of which are worst case): 1) nominal (*Nominal*), 2) all devices 5% faster (*Fast L*), 3) low $V_{dd}$, high MCF (*Low $V_{dd}$, High MCF*), 4) certain layout type devices being 5% slower (*Slow Layout*) 5) low power devices being 5% slower (*Slow Low Power*) and 6) all parameters at their worst values (*Worst Case*). The slack at each of these settings is significantly different from the nominal slack, demonstrating that different paths have different sensitivities and the ability of our method to predict that.

We also compute the upper bound on the arrival times (AT) at each of these settings using the upper bounding hyperplane method in [3] in order to determine the extent of pessimism induced by using such an upper bounding method, and the results are shown in the last column in Table 7. Expectedly, at the worst case corner setting, the arrival time computed using [3] is exact, and there is no overestimation, whereas at other settings of the parameters, particularly at the nominal, the arrival times computed using [3] are higher by as much as 20-30%.

*Table 7: Slacks at different settings of the parameters*

| Setting | Normalized Slack | Delta Slack w.r.t. Nom. | AT Overestimation using [3] |
|---|---|---|---|
| Nominal | -0.16 | - | 24.42% |
| Fast L | +0.64 | +0.80 | 33.21% |
| Low $V_{dd}$, High MCF | -1.13 | -0.98 | 20.23% |
| Slow Layout | -0.74 | -0.58 | 17.47% |
| Slow Low Power | -1.45 | -1.29 | 17.57% |
| Worst Case | -5.28 | -5.12 | 0.00% |

Fig 10 shows the plot of the nominal slacks versus the new slack for the 948 hyperplanes at a different setting **X**. This new setting is computed such that the path marked with a P in the figure, which has a large nominal slack, becomes the most timing critical at that setting. The setting corresponded to a 9% droop in $V_{dd}$, worst case MCF, certain layout transistor types being fast, others being slow. The set of paths that are encircled are the most sensitive when the parameters are at this particular setting. Note that this information is not obtained in current timing flows based on nominal slacks. In this case, the path marked with P would not have been considered critical. However, in our flow we can compute the slack at any setting of the parameters, thus enabling a what-if analysis.

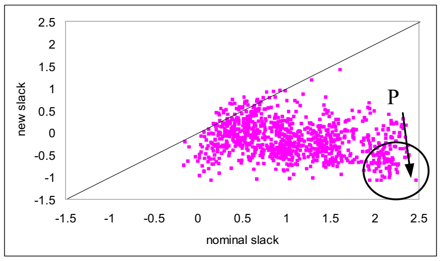

*Fig 10: Slacks at a different setting of* **X** *s.t. the path (marked* P *in the figure) with a large nominal slack becomes the most timing- critical*

## CONCLUSION

We present a block-based framework for computing the arrival times and slacks at all settings of the parameters, where only ranges on the parameter variations are known. We describe various pruning techniques in this paradigm. Results on industrial circuits show the viability of our approach.

## ACKNOWLEDGMENTS

We thank Prof. Farid Najm (Univ. of Toronto, Canada) and Drs. Noel Menezes and Rafael Rios (Intel Corporation) for helpful discussions on the approach.

## REFERENCES

[1] H. Chang and S. Sapatnekar, "Statistical Timing Analysis under Spatial Correlations," *TCAD*, 24(9):1467-1482, 2005.

[2] K. Killpack, C. V. Kashyap, and E. Chiprout, "Silicon Speedpath Measurement and Feedback into EDA Flows," *DAC*, pp. 390-395, 2007.

[3] S. Onaissi, and F. N. Najm, "A Linear-Time Approach for Static Timing Analysis Covering All Process Corners," *ICCAD*, pp. 217-224, 2006.

[4] L. Scheffer, "Why are Timing Estimates so Uncertain? What could we do about this?" *TAU* workshop, 2006.

[5] L. G. Silva, M. Silveira, and J. R. Phillips, "Efficient Computation of the Worst-Delay Corner," *DATE*, pp. 1-6, 2007.

[6] C. Visweswariah *et al* , "First Order Incremental Block Based Statistical Timing Analysis" *DAC*, pp. 331-336, 2004.

[7] CPLEX - http://www.ilog.com/products/cplex/index.cfm