An Exact Framework For Post-Layout Timing Correction

Hirendu Vaishnav and Massoud Pedram

CENG Technical Report 94-36

Department of Electrical Engineering - Systems
University of Southern California
Los Angeles, California 90089-2562
(213)740-4458

September 1994

An Exact Framework for Post-Layout Timing Correction*

Hirendu Vaishnav and Massoud Pedram
Department of Electrical Engineering-Systems
University of Southern California
Los Angeles, CA 90089

September 23, 1994

Abstract

The circuit designers are often faced with the problem of circuits which do not meet the timing constraints after the final stage of physical design. To avoid expensive reiterations of the synthesis and/or physical design cycle, it is imperative to identify a mechanism which corrects the timing behavior of the circuit with minimal perturbation to the existing layout. This report proposes a mechanism which identifies a minimal set of cells in the timing critical circuit (i.e., part of the circuit which has negative slack) after placement or routing such that speeding up these points result in simultaneous speedup of the entire circuit. Furthermore, exact relationship between these cells is also characterized based on the relative slacks and dependencies between the cells. This allows us to speedup the circuit simultaneously with a minimum increase in area. Once these cells are identified, transforms that speedup these cells can be applied to satisfy the timing constraints of the circuit. In this report, a simple transform that attempts to identify the best implementation of the cell, is used as a proof of concept.

*This report is being submitted at the end of Summer Internship (28th May - 12th Aug. 1994) at IBM-Hudson Valley Research Center, East Fishkill, NY.

Acknowledgements

This work was performed under the supervision of Dr. Jiten Apte at IBM, East Fishkill. I greatly appreciate supervision and guidance provided by Dr. Apte during my stay at IBM. I would also like to thank Dr. Deepak Sherlekar, Dr. Rajesh Gupta, Dr. Vasant Rao, Girish Rao and Paramesh Santanam for helpful discussions and other assistance.

Contents

1 Introduction and Motivation

The circuit designers are often faced with the problem of circuits which do not meet the timing constraints after the final stage of physical design. Traditionally, designers resort to tedious and time consuming manual timing correction. Such timing correction may be far from optimal in terms of the area penalty suffered. This area increase may also perturb the original circuit significantly requiring, in the worst case, expensive reiterations of synthesis and physical design stages. To avoid such expensive reiterations, it is imperative that efficient and accurate mechanisms be developed to achieve the required timing correction. In this report, we present a mechanism which identifies a minimal number of points to which a speedup transform needs to be applied to guarantee the circuit speedup and characterize exactly the effect of these transforms on the entire circuit. This mechanism is then applied to the simple transform of cell implementation switching, i.e., discrete gate sizing.

Prior work in post-layout timing correction has been relatively sparse. Previous work on timing correction has mostly concentrated on logic restructuring of NAND and NOR structures during logic synthesis. A speed up mechanism based on selective collapsing and re-decomposition of 2-input NAND decomposed networks was proposed by Singh et al [6]. Berman et al [1] have proposed restructuring mechanisms using the concept of frontier motion. However, both these approaches minimize estimated delay during logic synthesis and hence their applicability is limited due to inaccuracy of delay estimation during logic synthesis. Recently, Kannan et. al [5] have addressed the problem of post-layout timing correction. They propose a simple heuristic of identifying the most critical path and then identifying a single cell on that path to perform either a fanout optimization or a cell implementation switching. This mechanism acts only locally and hence does not take into account global factors like sharing of cells between multiple critical paths etc.. With this mechanism it is likely that a large number of cell implementation switches be performed (usually to larger implementations), while a similar effect could have been achieved with a smaller number of cell implementation switches.

As shown in the flow graph in Figure 1, our approach handles the problem of timing correction in a very global manner. First, a critical circuit graph consisting only of negative slack edges is extracted from the original graph. A vertex separator of minimum cardinality on this graph is then identified which corresponds to minimum number of points on the network, speeding up which will guarantee timing improvement of the entire circuit. Next, relative criticality and dependencies between the vertices belonging to the minimum vertex separator is exactly characterized using the concept of Area-SpeedUp curves. Each vertex belonging to the vertex separator set can now be used as a seed to grow into regions to which a set of transforms speeding up the circuit can be applied with minimum area increase. In general, the regions grown from the vertices belonging to the vertex separator set may include any logic in the fanin cone of the vertex, and the transform applied to these regions could be as complex as a complete resynthesis and replacement[1]. However, in this report we perform only discrete gate sizing on the vertices belonging to vertex separator.

The rest of the report is organized as follows. Section 2 describes the mechanism used to identify a minimum cardinality vertex separator from the critical graph. Section 3 introduces the concept of *Area-Slack curves* (ASCurves) and corresponding definitions. Section 4 explains our mechanism to perform optimal discrete gate sizing of the cells belonging to the cut. In the same section, we present how ASCurves can be used to perform an exact cell implementation switching when the cells belonging to vertex separator are independent. Conclusions and future work are discussed in section 6.

2 Identifying Minimum Number of Points to Speed-Up

Given a routed and/or placed circuit, the corresponding *original circuit graph* is denoted by $G_o(V_o, E_o)$ where each vertex corresponds to a port on a cell in the original circuit and each edge corresponds to a net segment in the original circuit. The slack S_v at vertex $v \in V_o$ is calculated as $S_v = R_v - A_v$ where R_v is the required arrival time at vertex v and A_v is the arrival time at vertex v. We define *edge slack* S_e for an edge $e = uv$ as $S_e = R_v - D_e - A_u$ where D_e is the delay through the edge. The following lemmas guarantee continuity of all negative slack edges.

Lemma 2.1 *If for a vertex v in the network, E_I is the set of all input edges and E_O is the set of all output edges, then:*

1. $S_v = min_{\{e \in E_I\}} S_e$
2. $S_v = min_{\{e \in E_O\}} S_e$

Proof Since $A_v = max_{\{e \in E_I, e=uv\}} A_u + D_e$,

$$\begin{aligned} S_v &= R_v - max_{\{e \in E_I, e=uv\}} A_u + D_e \\ &= min_{\{e \in E_I, e=uv\}} R_v - A_u - D_e \\ &= min_{\{e \in E_I\}} S_e \end{aligned}$$

Likewise, $R_v = min_{\{e \in E_O, e=vu\}} R_u - D_e$. Hence,

$$\begin{aligned} S_v &= min_{\{e \in E_O, e=vu\}} R_u - D_e - A_v \\ &= min_{\{e \in E_O, e=vu\}} R_u - A_v - D_e \\ &= min_{\{e \in E_O\}} S_e \end{aligned}$$

[1]Please refer to [7] for more details.

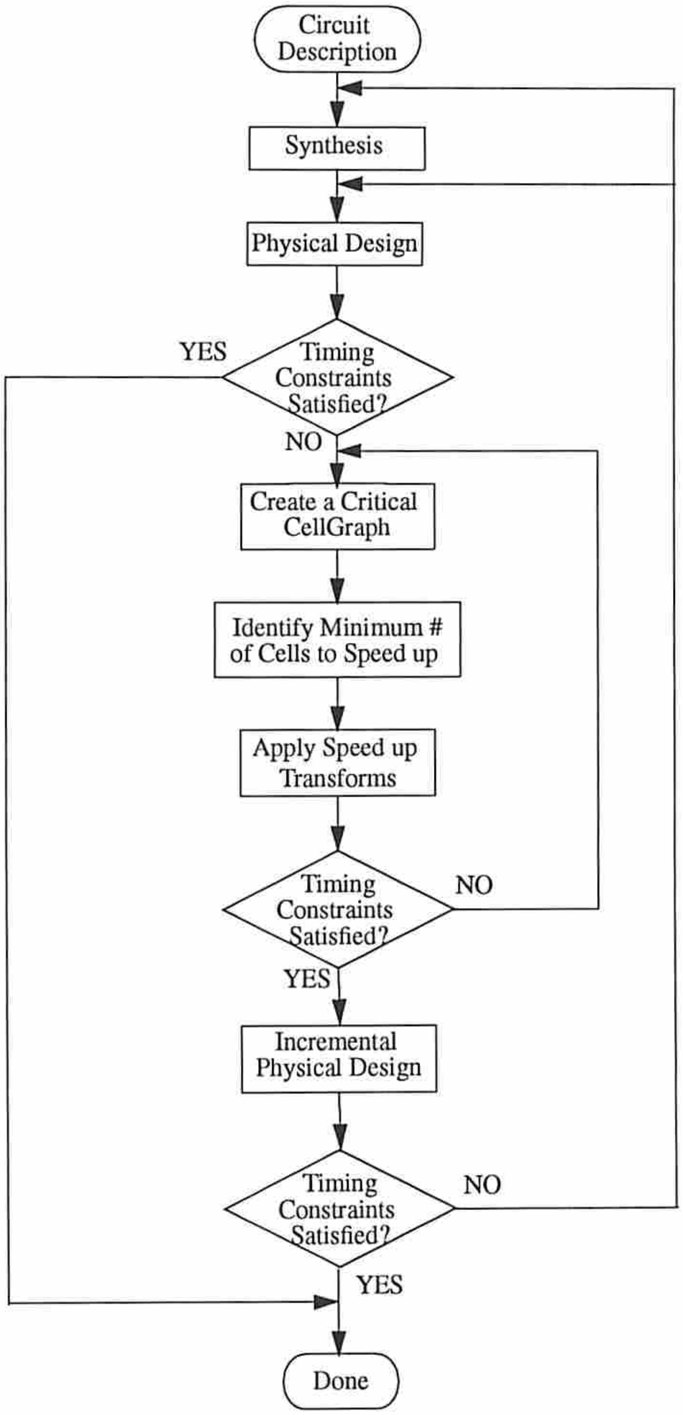

Circuit Description
Synthesis
Physical Design
Timing Constraints Satisfied?
YES
NO
Create a Critical CellGraph
Identify Minimum # of Cells to Speed up
Apply Speed up Transforms
Timing Constraints Satisfied?
NO
YES
Incremental Physical Design
Timing Constraints Satisfied?
NO
YES
Done

Figure 1: Flow Chart for Post-Placement Timing Correction

■

Corollary 2.2 *If for a vertex v in the network, E_I is the set of all input edges and E_O is the set of all output edges, then for each negative slack edge $e \in E_I$ ($e \in E_O$) there is an output (input) edge e' such that $S_e > S_{e'}$.*

Corollary 2.3 *Slack at a vertex is equal to the worst input edge slack and the worst output edge slack.*

Likewise, we define the *path slack* of a path p connecting u and v to be $S_p = R_v - D_p - A_u$ where D_p is the delay through the path. The results shown above for edges can also be extended to paths as given below. The proof for the following lemma is similar to the above proof and hence is omitted.

Lemma 2.4 *If for a vertex v in the network, P_I is the set of all paths that terminate in v and P_O is the set of all paths that originate in v, then:*

1. $S_v = min_{\{p \in P_I\}} S_p$
2. $S_v = min_{\{p \in P_O\}} S_p$

Corollary 2.5 *For a vertex v in the network, for each negative slack path $p \in P_I$ ($p \in P_O$) there is a path $p' \in P_O$ ($p' \in P_I$) such that $S_p > S_{p'}$.*

Corollary 2.6 *Slack at a vertex is equal to the worst input path slack and the worst output path slack.*

A *critical graph* $G_c(V_c, E_c)$ is extracted from G_o by collapsing all ports belonging to a cell into a single vertex and then retaining only the negative slack edges and the corresponding negative slack cell vertices. During the construction of this critical graph a dummy source node and a dummy sink node are introduced. An edge is introduced from the dummy source to each primary input and to each latch clock input in the critical graph. Likewise, an edge is introduced from each primary output and from each latch data input to the dummy sink. Next, all the vertices corresponding to PIs, POs and latch data inputs in the critical graph are collapsed, i.e., for each of these vertices, all inputs to the vertex are directly connected to all outputs of the vertex, thereby allowing removal of these vertices as shown in Figure 2. This critical graph captures all critical paths in the original circuit – namely, PI to latch paths, latch to latch paths, and latch to PO paths – as paths from the dummy source to the dummy sink. Hence,

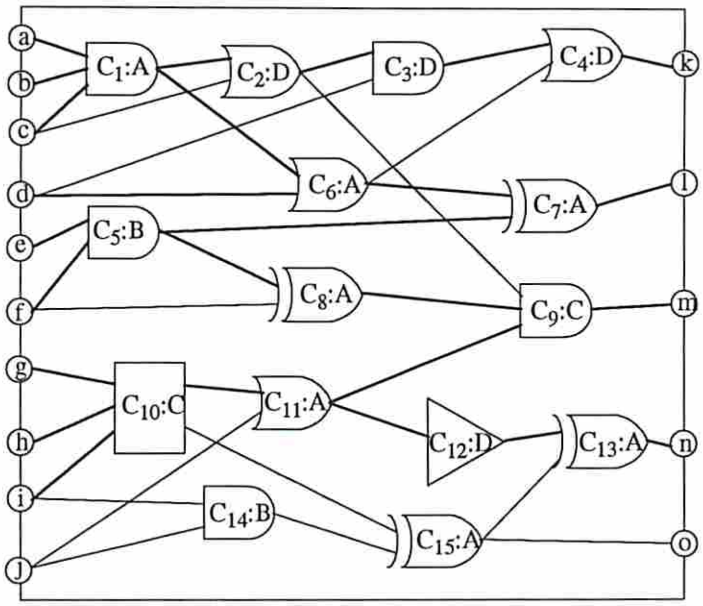
$V_c = \{v | v \in S_v < 0 \;\; or \;\; v = s \;\; or \;\; v = t\}$ where s is the dummy source and t is the dummy sink, and $E_c = \{e | e \in E_o \;\; and \;\; S_e < 0\}$ where S_e is the edge slack. During the construction of this graph, no edge which introduces a self-loop, i.e., a gate output which feeds back to the same cell (this can happen when we have disjoint multiple gates on the same cell), is added. Also, no parallel edges are introduced in this critical graph. If a *maximal path* is defined to be a path which is not a sub-path of a another path, the following theorem can be proposed for the critical graph. The proof follows from lemma 2.1 and corollaries 2.2 and 2.3.

Theorem 2.7 *If the combinational part of original circuit was acyclic, the critical graph is a connected DAG and each maximal path in this graph originates in dummy source and terminates in dummy sink.*

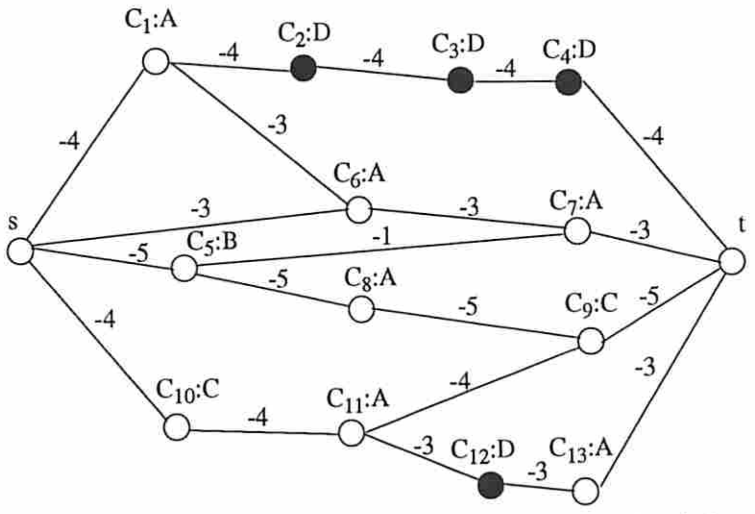
A *vertex separator* of s and t in the critical graph is a set of vertices, removal of which disconnects s and t. Let us denote such a vertex separator as V_s. As a consequence of theorem 2.7, speeding up the cells corresponding to vertices belonging to V_s by Δ results in overall circuit speedup of at least Δ under static timing analysis. Identifying a minimum cardinality vertex separator corresponds to minimum number of such cells whose speedup guarantees overall circuit speedup. More specifically, if the slack for each such cell could be improved to zero or to a positive value, the original circuit will no longer violate any timing constraints.

The minimum cardinality vertex separator problem can be converted to a max-flow/min-cut problem using a simple graph transformation proposed by Dantzig and Fulkerson [2]. According to this transformation, a *cell graph* $G_g(V_g, E_g)$ is constructed from the critical graph. A cell graph is a graph in which each cell is represented by two vertices connected by an edge. This is exemplified in Figure 3. Each cell c in the original critical graph is divided into two vertices, a *cell input vertex* c_i corresponding to the input vertex for cell c, and a *cell output vertex* c_o corresponding to the output vertex for cell c. All the inputs to c in critical graph are connected to c_i in the cell graph and all the outputs of c are connected from c_o in the cell graph. An edge is introduced between c_i and c_o as shown in figure. Thus, each edge in cell graph is either a cell edge (i.e., an edge connecting a cell input vertex to the corresponding cell output vertex) or a net edge. Next, all the cell edges are assigned an edge capacity of 1, while all net edges are assigned an edge capacity of ∞. It has been shown by Dantzig and Fulkerson that a maximum flow on this graph corresponds to maximum number of vertex disjoint paths from s to t, and hence, to the minimum cardinality of a vertex separator of s and t.

Since the max-flow through this graph remains invariant even when the net edges have capacity 1, the graph can be converted to a $0-1$ graph of *type 2* [4] before applying a max-flow algorithm. A graph is called a $0-1$ graph if flow through an edge can either be 0 or 1. A $0-1$ graph is of type 2 if each vertex in the graph either has one incoming edge

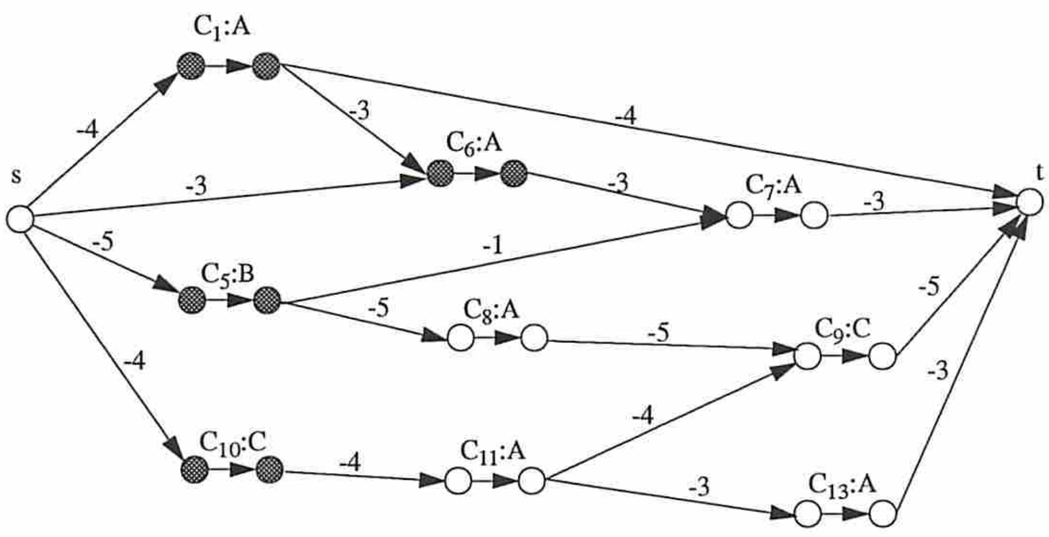


Original Circuit. Thick lines indicate negative slack edges. Each cell has 4 implementations (A, B, C, and D). Each cell above is identified by "name:implementation" pair.



The corresponding Critical Graph. Edge weights represent edge slacks. Darkened vertices have maximal implementation

Figure 2: The original circuit and the critical graph.



The shaded vertices correspond to the cells on the min-cut.

Figure 3: Cell Graph.

or one outgoing edge (except for dummy source and dummy sink which also have no incoming edges and no outgoing edges, respectively). For such graphs, Dinic's algorithm [3] can identify the max-flow in time complexity $O(\sqrt{|V_g|} \cdot |E_g|)$.

Once the max-flow is identified, the set of cells belonging to the minimum cardinality vertex separator is identified as follows. The capacity of each net edge is reset to ∞. The graph is traversed from s using a breadth first search. After the BFS, all the cells whose cell input vertex is visited during the traversal but the cell output vertex is not visited, form the minimum cardinality vertex separator. From now on, we refer to these set of cells as the min-cut of the cell graph.

Clearly, there can be many cuts with the same cardinality in the cell graph. We propose a mechanism which identifies alternative cuts based on relative speed up potentials of the cells on the cut in section 4.

3 Area-Speedup Curves

Once the cells which need to be speeded up are identified as described in the previous section, the simple approach of speeding them up maximally may be applied. For discrete gate sizing problem, this corresponds to choosing the fastest implementations for cells belonging to the cut. Let us denote by Δ_i the amount of speedup obtained at cell $c_i \in V_s$ where $1 \leq i \leq |V_s|$. Selecting fastest implementations will guarantee a circuit speedup of $min_i \Delta_i$. However, with this mechanism some cells are likely to be speeded up redundantly, resulting in a non-optimal area increase to achieve the required speedup. In this section, we propose the concept of *Area-Speedup Curves* (ASCurves) which enumerate and characterize exactly the set of area irredundant implementations to achieve different amounts of circuit speedup. Thus, apart from providing an area irredundant set of cell implementations, ASCurves also enumerate cell implementations for all possible speedup values, allowing a choice of any cell implementations including the implementations corresponding to the best speedup.

Before describing the details of enumerating all possible speedup values, we provide definition and a brief description of the ASCurves. For a circuit with multiple possible implementations, it is likely that the speedup at the output of a cell be different for different implementations of the cell itself or any cell in the fanin cone of the cell. Each such speedup has a corresponding area cost associated with it. We refer to this pair of area cost and the corresponding speedup (i.e., slack_for_current_implementation - slack_for_original_implementation) as an *Area-Speedup Point* (ASPoint). An ASPoint P_i with area A_i and speedup S_i is denoted as $P_i = \{A_i, S_i, I_i\}$ where I_i contains information about corresponding implementation. That is, for an ASPoint corresponding to a cell implementation B, $I_i = B$ (We denote cell implementations as A, B, C, D etc. in order of increasing size, i.e., $A_A \leq A_B \leq A_C \leq A_D$.).

For discrete gate sizing, we allow I_i of an ASPoint to contain information about more than one cell implementation. Specifically, I_i is an array of the size of the min-cut, with implementation of jth min-cut cell stored in $I_i[j], 1 \leq j \leq |V_s|$. An invalid/undetermined implementation of jth cell in the cut is specified by $I_i[j] = NULL$. For such points, $A_i = \sum_j A_{I_i[j]}, 1 \leq j \leq |V_s|$ when $A_{NULL} = 0$. We call such ASPoints as *CutASPoints* to signify that these ASPoints can characterize implementations of the whole cut.

Since we are looking at different implementations of cells either in the fanin cone of the cell or the cell itself only, use of arrival times instead of slacks and speedup values is also appropriate. However, using slacks instead of arrival time to capture delay information allows us to capture relative criticality of the cells as explained in the next section.

Definition 3.1 *Two ASPoints or CutASPoints* $P_i = \{A_i, S_i, I_i\}$ *and* $P_j = \{A_j, S_j, I_j\}$ *are* non-inferior *with respect to each other if* $A_i < A_j$ *and* $S_i < S_j$ *or if* $A_i > A_j$ *and* $S_i > S_j$.

Clearly, an ASPoint and a CutASPoint has useful practical information only when the *reference* slack value corresponding to the original implementation (with respect to which the speedups were derived) is also known. This slack value is incorporated in the AS-Curves and CutASCurves. An ASCurve consists of a set of non-inferior ASPoints along with the reference slack. Thus, ASCurve $C = \{\{P_i | 1 \leq i \leq |C|\}, S\}$ where S is the

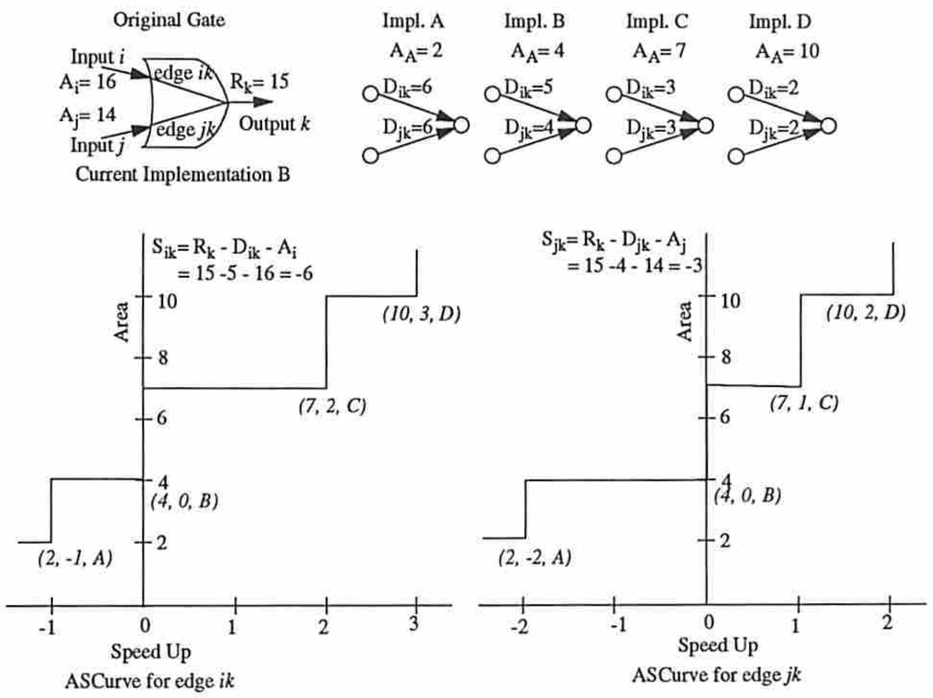


Figure 4: Generating ASCurves for a cell.

reference slack. Likewise, CutASCurve is a set of non-inferior CutASPoints with the corresponding slack value. ASCurve corresponding to an edge e is denoted by C_e. An example of ASCurve is shown in Figure 4. Before describing some operations on AS-Curves and CutASCurves, we present a few definitions/operations on the ASPoints and CutASPoints.

Definition 3.2 *Two ASPoints* $P_i = \{A_i, S_i, I_i\}$ *and* $P_j = \{A_j, S_j, I_j\}$ *are* compatible *iff both ASPoints are for the same cell and* $I_i = I_j$.

Definition 3.3 *Two CutASPoints* $P_i = \{A_i, S_i, I_i\}$ *and* $P_j = \{A_j, S_j, I_j\}$ *are* compatible *iff* $\forall k, 1 \leq k \leq |V_s|; I_i[k] = NULL$ *or* $I_j[k] = NULL$ *or* $I_i[k] = I_j[k]$.

Definition 3.4 *A* union *of implementation* I_i *with implementation* I_j *of for the same min-cut cell (denoted* $I_i \cup I_j$*) generates a new implementation* I_o *such that* $I_o = I_j$ *if* $I_i = I_j$ *and* $I_o = NULL$*, otherwise.*

Definition 3.5 *A* union *of implementation array* I_i *with implementation* I_j *of* l*th min-cut cell (denoted* $I_i \cup I_j$*) generates a new implementation array* I_o *such that* $I_o[l] = I_j$ *if* $I_i[l] = NULL$ *and* $I_o[l] = I_i[l]$*, otherwise.*

Definition 3.6 *A* union *of implementation arrays I_i and I_j (denoted $I_i \cup I_j$) generates a new implementation array I_o such that $I_o[l] = I_j[l]$ if $I_i[l] = NULL$ and $I_o[l] = I_i[l]$, otherwise $\forall k, 1 \leq k \leq |V_s|$.*

The time-complexity for the union and compatibility check involving implementation arrays is $O(|V_s|)$.

Definition 3.7 *Given an ASPoint $P_i = \{A_i, S_i, I_i\}$ with corresponding edge slack $Slack_i$ and another ASPoint $P_j = \{A_j, S_j, I_j\}$ with corresponding edge slack $Slack_j$, an* intersection *of P_i and P_j (denoted $P_i \cap P_j$) results in an ASPoint P_o derived as follows:*

- *If P_i and P_j are compatible: $P_o = \{A_o, S_o, I_o\}$ where $I_o = I_i$ (since $I_i = I_j$); $A_o = A_i$ (since $A_i = A_j$); $S_o = min(S_i + Slack_i, S_j + Slack_j)$.*
- *If P_i and P_j are incompatible: $P_o = NULL$.*

Definition 3.8 *Given a CutASPoint $P_i = \{A_i, S_i, I_i\}$ with corresponding edge slack $Slack_i$ and another CutASPoint $P_j = \{A_j, S_j, I_j\}$ with corresponding edge slack $Slack_j$, an* intersection *of P_i and P_j (denoted $P_i \cap P_j$) results in a CutASPoint P_o derived as follows:*

- *If P_i and P_j are compatible: $P_o = \{A_o, S_o, I_o\}$ where $I_o = I_i \cup I_j$; $A_o = \sum_j A_{I_o[j]}, 1 \leq j \leq |V_s|$; $S_o = min(S_i + Slack_i, S_j + Slack_j)$.*
- *If P_i and P_j are incompatible: $P_o = NULL$.*

The time complexity of intersection of ASPoints is $O(1)$. However, the time complexity of intersection of CutASPoints is $O(|V_s|)$.

Definition 3.9 *Given an ASPoint or a CutASPoint $P_i = \{A_i, S_i, I_i\}$ with corresponding edge slack $Slack_i$ and another ASPoint or CutASPoint $P_j = \{A_j, S_j, I_j\}$ with corresponding edge slack $Slack_j$, an* union *of P_i and P_j (denoted $P_i \cup P_j$) is a CutASPoint P_o derived as follows:*

- *If P_i and P_j are compatible: $P_o = \{A_o, S_o, I_o\}$ where $I_o = I_i \cup I_j$; $A_o = \sum_j A_{I_o[j]}, 1 \leq j \leq |V_s|$; $S_o = S_i + S_j$*
- *If P_i and P_j are incompatible: $P_o = NULL$*

The time complexity of union of CutASPoints is $O(|V_s|)$. In all other cases, the complexity of union is $O(1)$.

For discrete gate sizing, if the arrival times at the inputs to a cell are fixed, (i.e., the ASCurve at the input consists of a single ASPoint) the ASCurve of an internal edge (an edge connecting an input pin of the cell to an output pin of the cell) is derived as follows. For each internal edge of the cell, cell area and the corresponding speedup is computed for each cell implementation. Under a simple mechanism, the speedup for each implementation is calculated based on the intrinsic delay, output net delay (to account for the change in the drive of the cell), and the input net delay (to account for the change in the input capacitance). However, with a more accurate mechanism, incremental analysis of delays and exact slacks can be performed using a static timing analyzer. Such static analyzers also account for the change in the slew rate of the signals. It was observed from our experiments that using such a timing analyzer did not result in much slower run-times. Once the slacks corresponding to different implementations are computed, an ASCurve is created for the edge using the non-inferior ASPoints. This is shown in Figure 4.

It is likely that some cell in the input cone or the output cone may change implementation, thereby invalidating all the slack computations at the current edge. However, as will be described in section 4.4, we take care of this by characterizing exact dependencies between cells belonging to min-cut.

However, when more than one implementation is available for some cells in the input cone of the cell input, (i.e., the area-slack curve at the input has more than one ASPoint) the ASCurve of an internal edge needs to be derived using a convolution of the ASCurve of input edge with the ASCurve of the internal edge (this is also referred to as the OR-merging of ASCurves). The same is also applicable to CutASCurves.

Definition 3.10 *An* OR-Merge *of an input edge e' ASCurve (or CutASCurve) $C_{e'} = \{\{P'_i | 1 \leq i \leq |C_{e'}|\}, S'\}$ and the output edge e'' ASCurve (or CutASCurve) $C_{e''} = \{\{P''_j | 1 \leq j \leq |C_{e''}|\}, S''\}$ is a CutASCurve $C = \{\{P | P = P'_i \cup P''_j \forall (P'_i, P''_j) \in C' \times C''$ and P is non-inferior with respect to all other points$\}, S''\}$.*

As can be seen from the above, in the worst case, $|C| = |C_{e'}| \cdot |C_{e''}|$. In general, however, due to the requirement of non-inferiority the size of resultant CutASCurve is significantly less. The time complexity of OR-Merge is $O(|C_{e'}||C_{e''}||V_s|)$ when a CutASCurve is being merged with another and is $O(|C_{e'}||C_{e''}|)$ otherwise. As a matter of fact, when one of the curves being merged is an ASCurve, the complexity will be further reduced by noticing that the number of points on the ASCurve will be upper bounded by maximum number of implementation available for a cell. If we denote this number by c, and assume that $C_{e'}$ is ASCurve, the complexity of OR-Merge is $O(c|C_{e''}|)$. Usually, c is 3-4 even for large industrial libraries.

The worst case size of any CutASCurve can be estimated using c to be $O(c^{|V_s|})$ However, on average the size is much smaller. Also, the size can be easily restricted to be linear in terms of the timing delay (or minimum slack) by discretizing the delay values on the CutASCurve.

A consolidated ASCurve corresponding to all input edges of a pin can be generated from the ASCurves of the corresponding input edges by performing an AND-Merge sequentially on the ASCurves of these internal edges. Again, this is also applicable to CutASCurves.

Definition 3.11 *An*
AND-Merge *of an edge e' ASCurve (or CutASCurve) $C_{e'} = \{\{P'_i | 1 \leq i \leq |C_{e'}|\}, S'\}$ with another edge e'' ASCurve (or CutASCurve) $C_{e''} = \{\{P''_j | 1 \leq j \leq |C_{e''}|\}, S''\}$ is an ASCurve (or CutASCurve) $C = \{\{P | P = P'_i \cap P''_j \forall (P'_i, P''_j) \in C' \times C''$ and P is non-inferior with respect to all other ASPoints$\}, min(S', S'')\}$.*

In the above, though it may seem that in the worst case, $|C| = |C_{e'}| \cdot |C_{e''}|$, in reality, due to the non-inferiority requirement of the curves the worst case size is $|C| = |C_{e'}| + |C_{e''}|$. A simple implementation of AND-Merge will first merge the two sorted list of ASPoints (or CutASPoints) into a single sorted list in time linear in $|C|$ and then for each ASPoint (or CutASPoint), starting with the first element on the list, perform an union with the next higher compatible ASPoint (or CutASPoint) from the other curve. Thus, the timing complexity for this is $O(|C_{e'}| + |C_{e''}|)$. Hence, the timing complexity of AND-Merge, when a CutASCurve is being merged with another CutASCurve is $O(c^{|V_s|}|V_s|)$. When an ASCurve is being merged with an ASCurve, the timing complexity is $O(c)$. When an ASCurve is being merged with a CutASCurve, the timing complexity is $O(c^{|V_s|})$.

4 Optimal Discrete Gate Sizing for the Cut

Once the min-cut is identified, a *dependency graph* $G_d(V_d, E_d)$ is created from the original graph G_o and the min-cut V_s. Each vertex in V_d corresponds to a port on a cell belonging to V_s. Apart from that, a dummy source s and a dummy sink t are added to V_d. Each edge in E_d is either an internal edge or a dependency edge as described next.

If an input port and an output port belonging to the same cell are connected in G_o, they are also connected in G_d. At this point, the dependency graph (excluding s and t) is a bipartite graph with one partition consisting of input ports of the cut and other partition corresponding to the output ports of the cut. Next, timing dependency between the vertices of V_d is identified and an edge is introduced for each such dependency. Dependency edges are either between an output port of a cell to an input port of another cell, or between the dummy source s and an input port of the cut, or between an output port of the cut and the dummy sink t. Each such edge corresponds to a negative slack path between the two vertices which does not contain any port of a cell of belonging to the min-cut. We call such paths which do not pass through a cell in the cut as *cut independent* paths. Hence, in the dependency graph each edge corresponds to a cut independent path. An example of dependency graph is shown in Figure 5. The following theorem guarantees an edge between at least one input port of each cell on the min-cut and the dummy source, and an edge between at least one output port of each cell on the min-cut and the dummy sink.

Lemma 4.1 *1. For at least one input port corresponding to each output port on the min-cut, a negative slack cut independent path exists from the dummy source.*

2. For at least one output port corresponding to each input port on the min-cut, a negative slack cut independent path exists to the dummy sink.

Proof If for a cell on the min-cut, no such path exist, then there is no path connecting s in the cell graph to the corresponding cell without passing through one or more cells in the min-cut. However, such a cell could be dropped from the cut, improving the cardinality of the cut but still keeping s disconnected from t. This lead to a contradiction to the fact that original cut was a min-cut. Proof for the second part is similar. ■

4.1 Independent Min-cut Cells

When the cells on the min-cut are *timing independent* of each other, i.e., speeding up any cell on the min-cut has no impact on the speeding up of any other cell on the min-cut, optimal gate sizing of the cells on the min-cut can be performed very efficiently. A sufficient condition for the cells on the min-cut to be timing independent of each other is that no cell on the min-cut is in the fanin cone of another cell on the min-cut, i.e., they are logically independent. Such a situation is shown in Figure 6. However, based the relative slack values on any path from s to the cell input vertices, cells on the min-cut can be timing independent even when they are logically dependent on each other. An example of this is shown in Figure 5. As a matter of fact, the following theorem formally characterizes the conditions in which the cells belonging to the cut will be timing independent of each other.

Theorem 4.2 *The cells belonging to the min-cut are timing independent iff:*

1. *For each input port on the min-cut, one worst slack path from s is cut independent.*
2. *For each output port on the min-cut, one worst slack path to t is cut independent.*

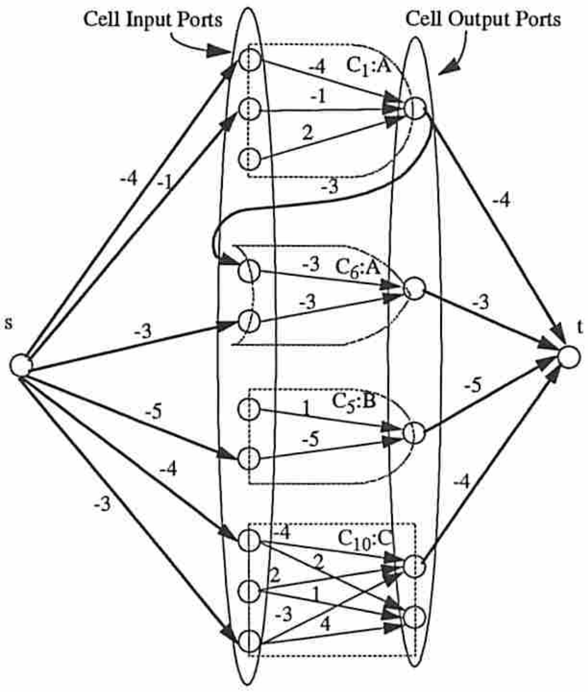


Figure 5: Dependency Graph corresponding to circuit in Figure 2 and 3. Dark edges are the net edges.

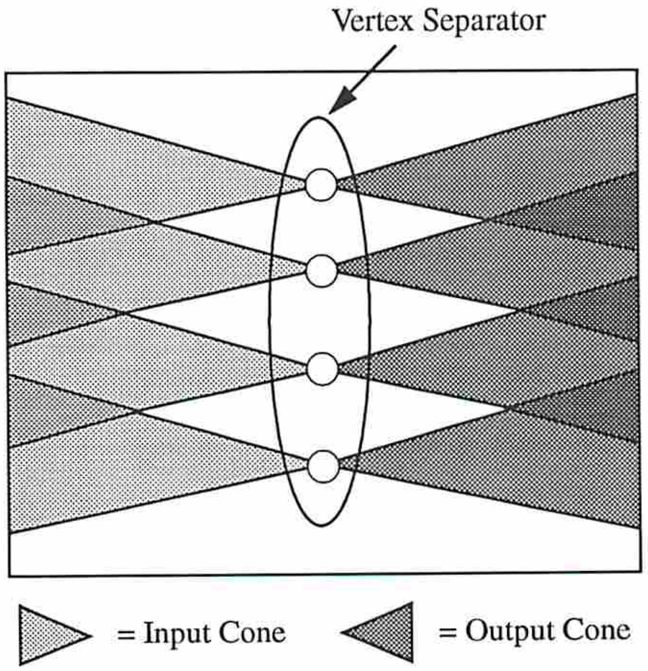


Figure 6: Logically independent cells on the cut.

Proof

$\Rightarrow$: For an input port, the arrival time is determined by the worst slack path from s. If such a path is cut independent, speeding up any cell on the min-cut will not change the arrival time at the port. Since the input ports are feeding out only to the output ports on the same cell, the arrival time at the output ports will also remain invariant to the speed up of any cell. Conversely, the required time at the output ports will be determined by the worst slack path to t. Again, if for all output ports, these paths are cut independent, the required times at all ports will remain invariant of speed up of other cells. Hence, if the above two conditions are satisfied, the cells belonging to the cut are timing independent.

$\Leftarrow$: Here, we will show that if either condition 1, or condition 2 is not satisfied, the cells belonging to the min-cut will not be timing independent. Suppose each worst slack path from s to an input port p is cut dependent. There are two cases:

1. If all paths connecting s with the input port are cut dependent, the arrival time will be strictly dependent on the speed up of those cells, implying that the cells are not timing independent.
2. If not all paths connecting s to the input port are cut dependent, let us assume that the cut independent path with worst path slack has slack S. Let $\Delta = S - S_p$. In this case, the arrival time at p will be dependent on the cell implementation of the cells belonging to cut dependent paths until a speed up of Δ is obtained on each of these cut dependent path. Hence, the cells are not timing independent.

Thus, if each worst slack path from s to an input port p is cut dependent, then the cells on the min-cut are not timing independent. Similarly, it can be shown that if each worst slack path from an output port p to t is cut dependent, then the cells on the min-cut are not timing independent.

■

In this case, we can ignore the dependency edges between the ports in the dependency graph[2]. Thus, when the cells on the cut are timing independent, the dependency graph can be reduced to a graph in which the cells on the min-cut form a bipartite subgraph and the s connects directly to some of the input ports and each output port is connected to t.

After the construction of the dependency graph, optimal implementation of the cells on the cut is performed as follows. The corresponding algorithm is as given in section

[2]It should be noted the above is valid only when the cells are being speeded up. When the cells are being slowed down as well, different analysis needs to be performed. I leave this as future work.

5. First, for each edge on the bipartite subgraph we create an ASCurve corresponding to different speed up values for different implementation of the cell. Given internal edge ASCurves for all internal edges of an output port, an ASCurve for the output edge (connecting to t) can be computed by performing a sequential AND-Merge on the internal edges as in routine CreateOutputASCurve.

Next, we need to merge all the ASCurves on the input edges to t into a unified CutASCurve. Each positive speed up point on this CutASCurve will correspond to a speed up of the entire circuit by the same amount, and the corresponding area gives the area required for the speed up. The difference between this area and the previous area corresponds to the area penalty for this speed up. It is also likely that the cells belonging to the cut are multi-output cells. In that case, more than one edge connecting to t corresponds to the same cell. Our mechanism of CutASCurve inherently handles the possibility of conflicting cell implementation choices as described next.

As shown in routine CreateCutASCurve, each ASCurve of the input edge is merged into the current CutASCurve. During the merge, as per definition of AND-Merge, we are merging ASPoints on the ASCurves with CutASPoints on the current CutASCurve. According to the definition of union on an ASPoint and CutASPoint, the points will be merged only if they are compatible. Thus, all CutASPoints on the resulting CutASCurve correspond to a feasible cut implementation.

Once the CutASCurve is identified, based on some criteria, a CutASPoint is chosen and the corresponding cell implementations are chosen for the cells on the min-cut. In general, the criteria is to choose the slowest positive slack implementation, or in absence of a positive slack implementation choose the fastest negative slack implementation. If the slack could not be reduced to zero or to a positive value, the process can be repeated after collapsing the cells which have no speed up potential.

The proof of the following theorem follows from the construction of the critical graph, dependency graph, definitions of ASCurve, AND-Merge and algorithms/routines proposed above.

Theorem 4.3 *When the cells on the min-cut are logically independent, the algorithm DoOptimalGateSizing performs a discrete gate sizing on the cells belonging to the cut optimally.*

Since the mechanism above identifies exact implementations of the cells on the min-cut to achieve a specific amount of speed up it will identify and slow down the cells which are redundantly faster. This implies that the algorithm also recovers area, further justifying the claim that the area perturbation will be minimal.

In the case when the cells are not logically independent but are timing independent, the above is true only if no cell on the cut was redundantly faster, i.e., if there is a chance

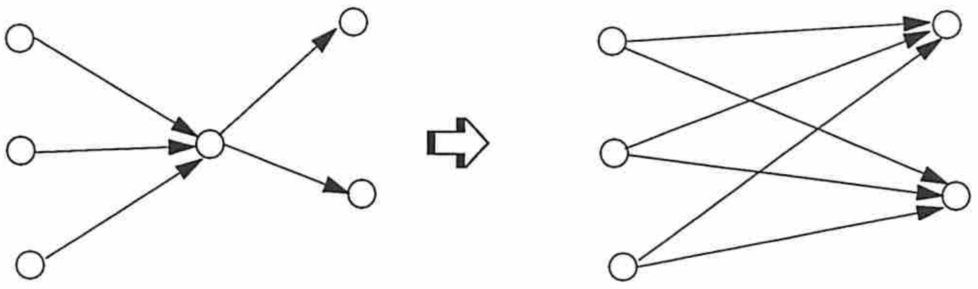


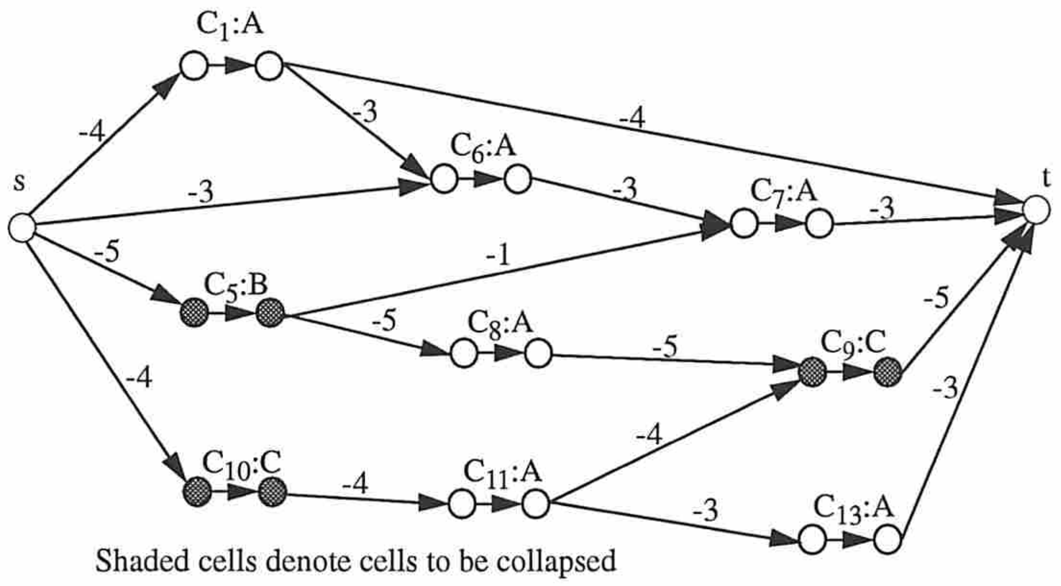
Figure 7: Collapsing a cell with insufficient speed up potential.

that some cell on the cut get slowed down. In that case, the conditions proposed above do not hold and more precise analysis taking into account the dependencies between the cells needs to be carried out.

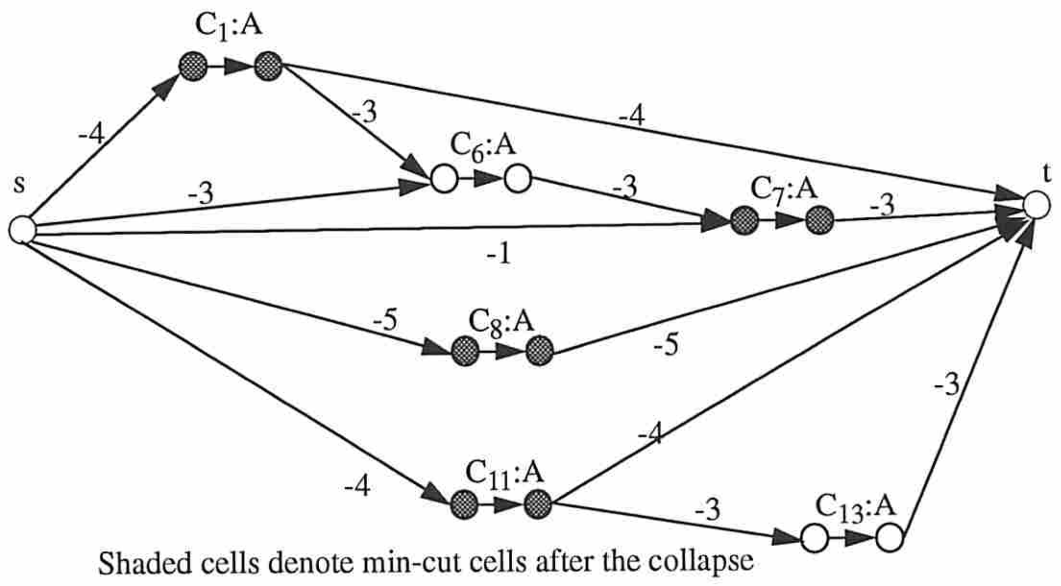
4.2 Handling Multiple Minimum Cardinality Cuts

The min-cut algorithm proposed in section 2 identifies only the first minimum cardinality cut. Speeding up such a min-cut might be better as these cells are closer to PIs and latch outputs and hence have a greater output cone of influence. This is particularly appropriate for discrete gate sizing as the only optimization done is on the cells belonging to the min-cut. When the min-cut cells are used for identifying the fanin regions of these cells on which more complex speedup transforms are applied, it might be actually better to identify a min-cut at some greater depth (however, a min-cut closer to POs might result in larger fanin cones, resulting in greater work in identifying the regions for resynthesis). In this case, cells can be collapsed selectively in the cell graph to make sure that they never occur in the min-cut. This is shown in Figure 7. Also, though our mechanism is exact after a min-cut is identified, the mechanism of identifying the min-cut is a heuristic. It is quite likely that some other cut of the same or higher cardinality might produce better speed ups at lower area cost. It is also likely that following a cell belonging to the cut, there may be a sequence of singly-linked cells as shown in Figure 9. The mechanism by which alternative min-cuts/cells be identified is as described below.

1. Multiple cuts with minimum cardinality needs to be distinguished by the speed up potentials of the cells on the cut. By requiring that each cell on the critical graph has a specific amount of speed up potential (e.g., Δ) and collapsing the remaining cells, we guarantee that the cut which will be identified guarantees at least a speed up of Δ, avoiding all the cuts of the same or less cardinality which had a speed up of less than Δ. Thus, this mechanism even allows us to choose a cut which has greater cardinality than the min-cut. This is also useful for discrete gate sizing to avoid selecting the gates in the min-cut which already have their



A) CellGraph before the collapse



B) CellGraph after the collapse

Figure 8: An illustration of collapsing cells which do not meet minimum speed up requirement.

fastest implementation. This is illustrated in Figure 8, where we require that each cell should have at least three faster implementations.

2. In case of sequential cells following the min-cut, it is better to consider all the sequential cells as a single cell as it results in greater speed up potential for the min-cut. Hence, all the sequential cells are *merged* into a single cell edge in the graph as shown in Figure 9. The corresponding edges in the port graph is also shown in the figure. In this case, CutASCurves are created by OR-Merging the internal edges which are sequential.

An algorithm which starts with maximum possible speed up (i.e., speed up corresponding to the worst negative slack) as the required minimum speed up potential, and then reduces it until there is a min-cut with the required speed up potential, can be used to incorporate the above mechanism.

4.3 Timing Complexity of OptimalTimingCorrection

As described earlier, the timing complexity of identifying the min-cut is $O(\sqrt{V_g}|E_g|)$. Once the dependency graph is identified the complexity of generating ASCurves for internal edges and AND-Merging them or each output is $O(c)$. Thus, the complexity of creating an AND-Merged ASCurve for all output ports, assuming that each cell has at most o outputs, is $O(co|V_s|)$. The complexity of AND-Merging the ASCurves to the final CutASCurve, sequentially is $O(c^{|V_s|}o|V_s|)$. However, due to the fact that a lot of inferior points are dropped from the CutASCurve, on average the run time is significantly less than this[3]. Also, as mentioned in section 3, when delay values are discretized, size of CutASCurve can be reduced from $O(c^{|V_s|})$ to linear in the worst negative slack reducing the timing complexity from $O(c^{|V_s|}o|V_s|)$ to $O(worst\_slack \cdot o \cdot |V_s|)$.

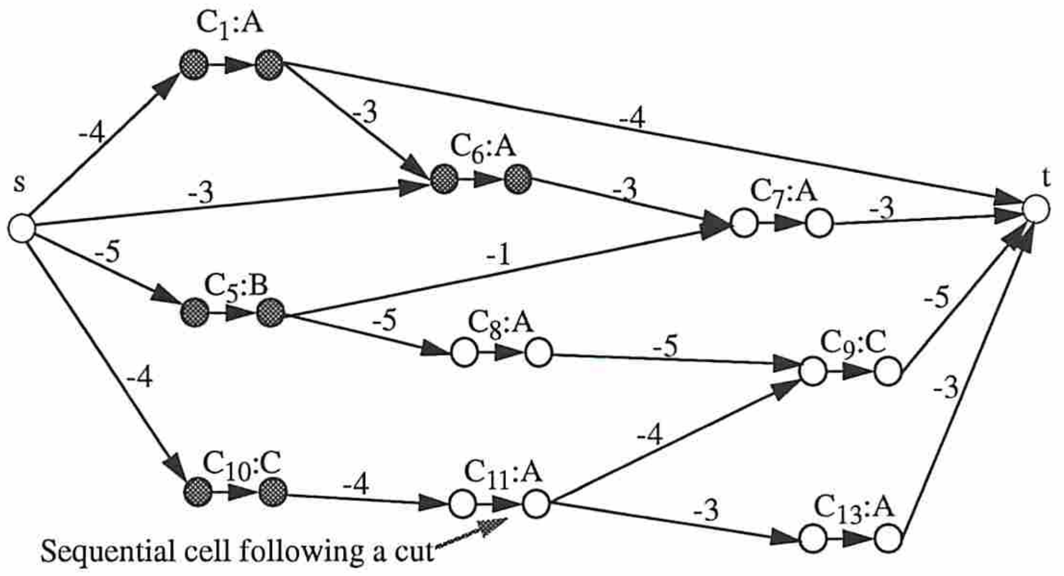
Considering c, o and *worst_slack* to be constants, overall time complexity of Optimal-TimingCorrection is $O(\sqrt{V_g}|E_g| + c^{|V_s|}|V_s|)$ or $O(\sqrt{V_g}|E_g| + |V_s|) = O(\sqrt{V_g}|E_g|)$ when the delay values are discretized.

4.4 Dependent Min-cut Cells

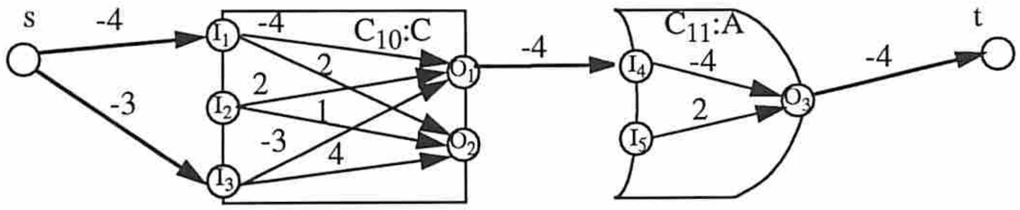
When the cells on the min-cut are dependent on each other, exact dependencies need to be identified between the cells on the cut[4]. The only effect of dependencies is that speeding up a cell on the cut might make some other cell less critical and hence reduce the speed up required for that cell. Thus, assuming that the cells are timing independent, when they are actually timing dependent, might imply that we might speed up some cell more than required, resulting in an increased area penalty. Thus, assuming timing

[3]Even otherwise it seems that it can be shown that the complexity in the worst case is pseudo-polynomial in $|V_s|$. I will address this in future.

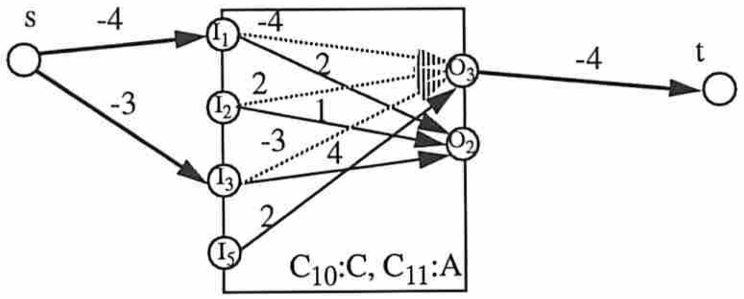
[4]Again, this will be addressed in future.



A) Original cell graph



B) Sequential cells before the merge



┈┈┈⦀┈ denotes edges containing CutASCurve for Cell implementations of C_{10} and C_{11}

━━► denotes edges containing ASCurve for Cell implementations of C_{10}

──► denotes edge containing ASCurve for Cell implementations of C_{11}

C) Corresponding part of dependency graph after the merge

Figure 9: An illustration of merging of sequential cells following a cut.

independence still guarantees the speed up corresponding to the CutASPoint speed up. However, it may result in a greater speed up or it may use extra area to achieve the speed up.

5 Implementation

The initial implementation of the algorithm is as given in this section[5]. The main algorithm *OptimalTimingCorrection* is given below.

```
Algorithm 1 OptimalTimingCorrection ($G_o$)
$G_o(V_o, E_o)$ is the original circuit graph
begin
1   while worstSlackIsNegative($G_o$) do
2     $G_g$ = CreateCellGraph($G_o$);
3     ForEachCell $v \in G_g$ do
4       if hasSpeedUpPotential($v$) continue;
5       else collapse($v$);
6     if areConnectedDirectly($s, t$) exit;
7     $V_s$ = IdentifyMinimumVertexSeparator($G_o, G_g$)
8     DoOptimalGateSizing($G_o, V_s$);
end
```

Method *worstSlackIsNegative* does a static timing analysis of the circuit and sets the arrival time, required time and slack values at each port in the circuit. It returns 1 if the worst negative slack at a primary output or a data input to a latch is negative, and 0 otherwise. *CreateCellGraph* creates a cell graph from the circuit as shown in algorithm 2. The method *hasSpeedUpPotential* checks the library for alternative implementation of the cell. If no faster implementation is available, it returns a 0, otherwise it returns a 1. Alternatively, some other criteria of speed up potential can be used to guarantee a minimal speed up potential. Also, when the transform being applied is that of re-synthesis/re-placement a mechanism based on circuit structure/placement could be used to identify speed up potential of the fanin cone of the cell. If the cell does not satisfy speed up potential required, line 5 of the algorithm collapses it as shown in Figure 7. After collapsing all the cells which do not meet the minimum speed up potential criteria, if the dummy source in the cell graph directly connected to the dummy sink, the circuit

[5]However, it should be noted that further modification/improvement in the implementation will be made in future.

can not be speeded up further. In that case, as shown in line 6, the algorithm terminates. Routine *IdentifyMinimumVertexSeparator* identifies the minimum cardinality vertex separator as shown in algorithm 3. Finally, *DoOptimalGateSizing* performs the actual gate sizing on the vertex separator as explained in algorithm 4.

```
Algorithm 2 CreateCellGraph ($G_o$)
$G_o(V_o, E_o)$ is the original circuit graph
Returns a cell graph $G_g(V_g, E_g)$
begin
1   $V_g$ = NULL; $E_g$ = NULL;
2   ForEachCell $v \in G_o$ do
3     CellPresentInCellGraph = 0;
4     ForEachPort $p$ on $v$ do
5       if sourceCondIsSatisfied($p$)
6         ForEachSinkPort $p_s$ of $p$
7           if sinkCondIsSatisfied($p_s$)
8             CellPresentInCellGraph = 1;
9             if isDataInputToLatch($p_s$)
10              connect($c_o$, $t$);
11            else
12              connect($c_o$, inputCellVertexOf($p_s$);
13      else if negativeSlackClockInputOnLatch($p$)
14        CellPresentInCellGraph = 1;
15        connect($s$, $c_i$);
16    if CellPresentInCellGraph
17      connect($c_i$, $c_o$);
18  ForEachPIPort $p$ on $G_o$ do
19    if sourceCondIsSatisfied($p$)
20      ForEachSinkPort $p_s$ of $p$
21        if sinkCondIsSatisfied($p_s$)
22          connect($s$, inputCellVertexOf($p_s$);
23  return($G_g(V_g, E_g)$);
end
```

The above algorithm generates a cell graph from the original circuit[6]. First, for each port on each cell of the graph it is checked if it is a valid source port. Method *sourceCondIsSatisfied* checks to see if the port is either a negative slack output port or a negative

[6]Though the concept of critical graph is useful in explaining approach minimum cardinality vertex separator and its relevance, it is possible to bypass generating critical graph as done here.

slack output corresponding to a bi-directional port. The corresponding negative slack input ports or negative slack inputs corresponding to bi-directional ports are identified and the corresponding edges are introduced in lines 6-12. If the sink port turns out to be a data input port on a latch, an edge is introduced to the dummy sink instead as shown in line 10. Also, lines 13-15 connect the input ports directly from the dummy source if the source port is a clock input on the cell. It should be noted that according to theorem 2.1 and corollaries 2.2 and 2.3, each negative slack port should be connected in cell graph. Hence, as shown in line 16-17, if the any of the input port and output port is connected, an internal edge is introduced for the cell. Finally, as shown in lines 18-22, for each negative slack primary input port, the dummy source is connected to the corresponding negative slack sink port.

```
Algorithm 3 IdentifyMinimumVertexSeparator $(G_o, G_g)$
$G_o(V_o, E_o)$ is the original circuit graph
$G_g(V_g, E_g)$ is corresponding cell graph
Returns a vertex separator set $V_s$
begin
1   flowPresent = 1; totalFlow = 0;
2   while flowPresent do
3     flowPresent = 0;
4     MarkAllEdgesUnUsed($E_g$);
5     UnsetDepthLevels($V_g$);
6     DoBFSAndSetDepthLevels($s$);
7     if wasSinkReachedInLevelizedDFSSearch($s$)
8       flowPresent = 1; totalFlow++;
9   MarkAllForwardEdgesUnBlocked($E_g$)
10  UnsetDepthLevels($V_g$);
11  DoBFSAndSetDepthLevels($s$);
12  $V_s$ = NULL;
13  ForEachVertex $v \in V_o$
14    if depthIsSet($v_i$)
15      if depthIsUnSet($v_o$)
16        $V_s = V_s \cup v$;
17  return($V_s$);
end
```

The algorithm *IdentifyMinimumVertexSeparator* identifies the min-cut and corresponding minimum cardinality vertex separator as shown above. The *while* loop in lines 2-8, iteratively finds a path in the levelized graph. The number of iterations of this loop corresponds to the maximum flow through the corresponding network. The routine

DoBFSAndSetDepthLevels essentially generates a levelized graph by setting as depth of each node, the shortest distance from the dummy source to the node. Routine *wasSinkReachedInLevelizedDFSSearch*, performs a DFS on the levelized graph while not traversing edges which were blocked during the previous traversal of the graph. If a path to the dummy sink was found, it returns a 1; else it returns a 0. For each such path, the edges are also marked as blocked to avoid reuse of the edges during further calls to this routine. In essence, routine *MarkAllForwardEdgesUnBlocked* makes all the forward edges have a capacity of ∞ by making them unblocked. After doing another BFS (line 11), lines 12-16 then identify the corresponding minimum cardinality vertex separator.

Once the minimum cardinality vertex separator is identified, the following algorithm performs optimal gate sizing on the cells belonging to the vertex separator. Details of the implementation of the algorithm (and the other two algorithms) are explained on section 4 and hence are not provided.

```
**Algorithm** 4 DoOptimalGateSizing $(G_o, V_s)$
*$G_o(V_o, E_o)$ is the original circuit graph*
*$V_s$ is the minimum cardinality vertex separator*
**begin**
1  $G_d(V_d, E_d)$ = CreateDependencyGraph$(G_o, V_s)$;
2  **ForEachInternalEdge** $e \in E_d$ **do**
3    $C_e$ = CreateASCurve$(e)$;
4  **ForEachOutputPort** $v \in V_d$ **do**
5    $C_{e=vt}$ = CreateOutputASCurve$(v, G_d)$;
6  $C$ = CreateCutASCurve$(t, G_d)$;
7  ChooseCutImplementation$(C)$;
**end**
```

CreateDependencyGraph creates a dependency graph from the vertex separator as explained in section 4. Routine *CreateASCurve* creates ASCurves for the internal edge as shown in Figure 4. Lines 4-5 create ASCurve for each output edge on an output port as explained in section 4 and in routine *CreateOutputASCurve*. Routine *CreateCutASCurve* creates the CutASCurve corresponding to all the cells on the min-cut. Finally, *ChooseCutImplementation*, chooses the cell implementations of the cells based on the given cost criteria.

```
Algorithm 5 CreateOutputASCurve ($v, G_d$)
  $v$ is an output port in the dependency graph
  $G_d(V_d, E_d)$ is dependency graph
  Returns the output ASCurve $C_o$
begin
1   $C_o$ = NULL; $E_I$ = set of input edges of $v$
2   ForEachInputEdge $e \in E_I$ do
3     if ($C_o$ = NULL) $C_o = C_e$;
4     else $C_o$ = AND-Merge($C_o, C_e$);
5   return ($C_o$);
end
```

Algorithm *CreateOutputASCurve* sequentially AND-Merges the internal edge ASCurves to create the output edge ASCurve as explained in section 4 and 3. Likewise, the following algorithm *CreateCutASCurve* sequentially AND-Merges the output edge ASCurves to create the overall CutASCurve.

```
Algorithm 6 CreateCutASCurve ($v, G_d$)
  $v$ is the sink in the dependency graph
  $G_d(V_d, E_d)$ is dependency graph
  Returns the CutASCurve $C_o$
begin
1   $C_o$ = NULL; $E_I$ = set of input edges of $t$
2   ForEachInputEdge $e \in E_I$ do
3     if ($C_o$ = NULL) $C_o = C_e$;
4     else $C_o$ = AND-Merge($C_o, C_e$);
5   return ($C_o$);
end
```

6 Concluding Remarks and Future Work

In this report, an approach to perform post-layout timing correction was proposed. The approach is based on identifying a minimum vertex separator on the critical graph of the network and then speeding up the cells on the corresponding min-cut to achieve the timing correction. This approach was then applied to perform an optimal discrete gate sizing for cells belonging to the min-cut. When the cells are timing independent, the algorithm was shown to be optimal. The algorithm was implemented in IBM CAD-tool *Hierarchical Design Planner* and looks promising in terms of efficiency and practical value.

The future work involves improving the implementation, generating experimental results and extending it to the case when the cells are not timing independent. If the results are promising, the approach will be extended to post-layout timing correction using other transforms, namely, that of post-placement selective resynthesis and re-placement.

References

[1] C. L. Berman, D. J. Hathaway, A. S. LaPaugh, and L. H. Trevillyan. Efficient techniques for timing correction. In *Proceedings of the 27th Design Automation Conference*, pages 415–419, June 1990.

[2] G. B. Dantzig and D. R. Fulkerson. On the max-flow min-cut theorem of networks. In *Linear Inequalities and Related Systems. Anals of Math. Study 38*, pages 215–221. Princeton University Press, 1956.

[3] E. A. Dinic. Algorithm for solution of a problem of maximum flow in a network with power estimation. *Soviet Math. Dokl.*, 11:1277–1280, 1970.

[4] Shimon Even. *Graph Algorithms.* Computer Science press, Rockville, Maryland, first edition, 1979.

[5] L. N. Kannan, P. R. Suaris, and Hong-Gee Fang. A methodology and algorithms for post-placement delay optimization. In *Proceedings of the 31th Design Automation Conference*, pages 327–332, June 1994.

[6] K. J. Singh, A. Wang, R. K. Brayton, and A. Sangiovanni-Vincentelli. Timing optimization of combinational logic. In *Proceedings of the IEEE International Conference on Computer Aided Design*, pages 282–285, November 1988.

[7] H. Vaishnav, J. Apte, and D. Sherlekar. An integrated framework for layout-driven resynthesis. IBM Invention Disclosure: FI894-0230, July 12, 1994.