An Algorithm for Incremental Timing Analysis

Jin-fuw Lee, and Donald T. Tang

IBM Thomas J. Watson Research Center,
Yorktown Heights, NY 10598

Abstract- In recent years, many new algorithms have been proposed for performing a complete timing analysis of sequential logic circuits. In this paper, we present an incremental timing analysis algorithm. When an incremental design change is made on the logic network, this algorithm will identify the portion of design for which the timing is affected, and quickly derive the new arrival times and slacks. A fast incremental timing analysis is desirable for users doing interactive logic design. It is particularly important for a logic synthesis program, which needs to evaluate the circuit delays under many logic modifications.

1. Introduction

Designs using level-sensitive latches have become fairly popular lately. A significant advantage of such a design style is that the cycle stealing across the latches is allowed and the clock cycle time can be made smaller than the longest combinational logic delay. A general formulation of timing constraints for both edge-triggered flip-flops and level-sensitive latches was presented by Sakallah, Mudge, and Olukotun in [1]. These timing constraints are used for a pattern-independent timing analysis of logic circuits [1]. Szymanski and Shenoy in [2] developed a timing verification algorithm through an elegant analysis of timing constraints. There are several other algorithms [3-4] for the pattern-independent timing analysis in the literature. All these algorithms are based on the latch graph and have been used for performing the timing analysis of complete logic designs.

A sequential logic circuit consists of a combinational logic network, a set of memory elements (level-sensitive latches or flip-flops) and a set of primary inputs (PIs) and outputs (POs). The timing constraint set, G, for such a circuit may be abstracted into the form of a ***latch graph***, L. A node on the latch graph represents a PI, a PO, or a memory element, while an edge represents the longest and shortest combinational delay. Each memory element is controlled by a clock waveform, which is characterized by a clock cycle time, , a setup time, S_i, a hold time, H_i, a clock opening time B_i, and a clock closing time F_i. An example of a sequential circuit and the corresponding latch graph is shown in Fig. 1. The latch graph is extracted by running the longest and shortest path algorithm through combinational logic N times, each with one memory element or a PI as the source node. The complexity of the latch graph extraction is $O(N \times |G|)$, which, for a large circuit, usually takes more CPU time than that of timing analysis itself, $O(m \times |L|)$, where m, the number of iterations, is typically much less than N. Therefore, as a first step for achieving fast timing analysis, we need to avoid the expensive overhead of the latch graph extraction.

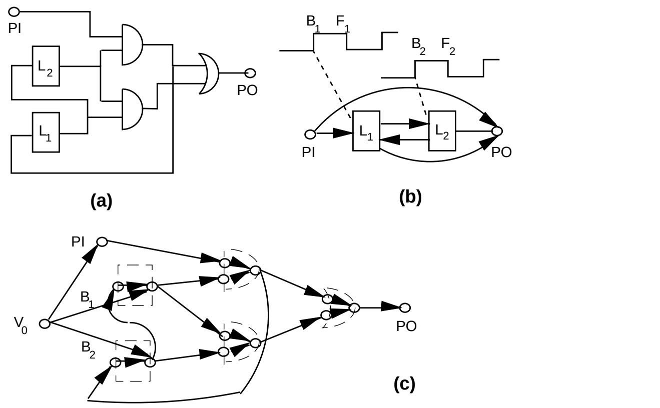

Figure 1. A simple example. (a)A sequential logic circuit. (b)The latch graph. (c)The timing constraint graph.

A direct approach was proposed in [5] to apply timing analysis on the full timing constraint graph $G = (V,E)$, which is defined as follows: Each node V_i in G represents either a PI, a PO or a pin on the logic gate, while each edge (V_i, V_j) represents the delay $\Lambda_{i,j}$ between a pair of pins. In G, a global source node V_0 is added to represent the time origin, and an edge is inserted from V_0 to each PI node with user-asserted late arrival time as edge weight. This way, all the signal paths originated from PIs may be extended to the common source node. To account for the signal paths originated from memory elements, an edge is also inserted from V_0 to the output pin of each memory element with clock opening-edge arrival time B_i as edge weight. See Fig. 1(c). It was shown in [5] that the late-mode worst-case arrival time A_i at node V_i is equal to the longest path length from V_0 to V_i in G. Therefore, the longest path algorithms, such as the Bellman-Ford method, may be used to solve the late-mode timing problem, [5, 6]. In the general case where G may contain some feedback loops, the longest path algorithm takes a number of iterations to converge with a complexity $O(m \times |G|)$, where m, the number of iterations is bounded by N and typically less than 10. For a chip with 50,000 gates, the longest path algorithm on the full timing constraint graph typically takes about one minute of CPU time on a 50 MIP machine, while the latch graph based algorithm may take up to 10 times of CPU time. When we apply these two approaches to the incremental design environment as shown in Fig. 2(a) and 2(b), the difference becomes even wider. Suppose that n design modifications are successively tried. The difference in CPU times between two methods would be n times

32nd ACM/IEEE Design Automation Conference ®
Permission to copy without fee all or part of this material is granted, provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission. © 1995 ACM 0-89791-756-1/95/0006 $3.50

www.manaraa.com

bigger. Since the latch graph must be regenerated for each incremental logic change, and it is difficult to improve the extraction time of the latch graph incrementally, we elect to develop our incremental timing analysis algorithm by modifying the direct approach. See Fig. 2(c).

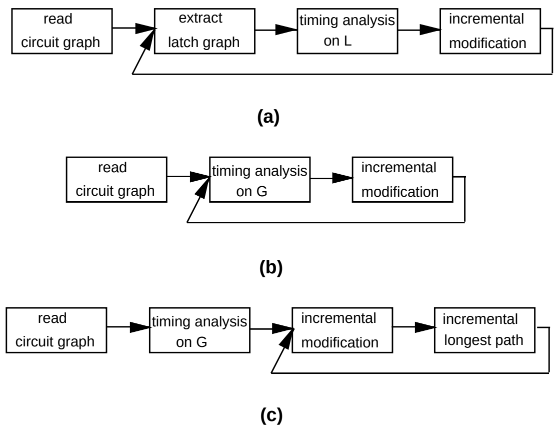

Figure 2. Timing analysis methods for incremental changes. (a)Timing analysis based on latch graph. (b)Timing analysis based on full timing graph. (c)Incremental timing analysis.

The full timing constraint graph is also an ideal medium for capturing the incremental changes made at gate levels. The logic design is often modified either manually or with an optimization program to meet some delay or power requirements, using techniques such as power-up, power-down, re-placement, re-route, or re-synthesis. Let G' be the timing constraint graph after some modifications are made on edge weights. Let $\Lambda_{i,j}$ and $\Lambda'_{i,j}$ be respectively the delay weight of edge (V_i, V_j) in G and G'. The difference between two graphs is captured by those edges with weights changed: $G' - G = \{(V_i, V_j) \mid \Lambda'_{i,j} \neq \Lambda_{i,j}\}$. To simplify discussions, let us include in G and G' some edges with weight $-\infty$ to represent non-existing edges. This enables one to use an identical set of edges for G and G', and to consider the ***deletion*** and ***insertion*** of edges as special cases of weight changes: The deletion of an edge is modelled as a change of edge weight from $\Lambda_{i,j}$ to $-\infty$, while the insertion of an edge is modelled as a change of edge weight from $-\infty$ to $\Lambda'_{i,j}$. For an incremental change in the logic design involving a small number of edges, it is very desirable to have a fast method to find out the corresponding changes in timing. Since the computation time of the longest path algorithm on G' is proportional to $|G'|$, it may take more than 1 hour CPU time for a large VLSI chip with millions of transistors. This is too costly for chip designs which need frequent incremental modifications. In this paper, we propose an incremental longest path algorithm which is very efficient, since it generally retains and utilizes the timing information as much as possible to minimize the amount of computation.

Let us briefly review the single-source longest path problem [5-7]. Given a node V_i, there may be many paths from V_0 to this node. For each such path p, the path length $L(p)$ is defined as the sum of edge weights along p. The longest path length A_i is $\max_p\{L(p)\}$, where the maximum is taken over the set of all paths from V_0 to V_i. It may be shown that if G does not contain any positive loop, the set of longest path lengths $\{A_i\}$ exists for all nodes, and it is the minimum solution which satisfies all the constraints $\{A_i - A_j \geq \Lambda_{i,j} \mid (V_i, V_j) \in G\}$. In many longest path algorithms [5, 6], a ***dominance graph*** $T = \{(t_i, V_i)\}$ is also built, where t_i is the dominant predecessor of V_i which updates A_i in the path searching process. If G does not contain any positive loop, T is the ***longest path tree***. If G contains positive loops, T will also contain some loops, all of which have positive gains. Note that the inclusion of non-existing edges, i.e. edges with weight $-\infty$, does not change the longest path lengths in G.

2. The incremental longest path problem

Problem: Given the longest path solution $\{A_i\}$ to G and the incremental change $G' - G$, find the new longest path lengths $\{A'_i\}$ in G'.

The edges in $G' - G$ may be classified into two kinds:

1. Edges with positive changes: Edge weights in G are increased. This may happen, when some circuit in the design is replaced by a lower power version. An insertion of a new edge can be modelled as an increase of edge weight from $-\infty$ to the new delay value.
2. Edges with negative changes: Edge weights in G are decreased. This may happen, when some circuit in the design is replaced by a higher power version. A deletion of an edge can be modelled as a decrease of edge weight from its previous value to $-\infty$.

Let $E^{(+)}$ and $E^{(-)}$ represent respectively the sets of edges with positive and negative changes: $G' - G = E^{(+)} \quad E^{(-)}$.

Definition 1: Let $e_{i,j} = (V_i, V_j)$ be an edge in the set $G' - G$, and $C(e_{i,j})$ be the set of nodes inside the ***fan-out cone*** from node V_j. Then the fan-out cone from the set of modified edges is defined as the union: $C = \quad \{C(e) \mid e \in G' - G\}$. For an example, see Fig. 3.

Definition 2: The ***cone of change*** C_C is defined as the set of those nodes in which the new arrival time is different from the old one: $C_C = \{V_i \mid A_i \neq A'_i\}$.

Lemma 1

$A_i = A'_i$ for nodes $V_i \notin C$. That is, C_C is a subset of C.

Proof: There can not be any path from $e_{i,j}$ to those nodes in $V - C(e_{i,j})$ by the definition of a fan-out cone. Therefore, the signal arrival times at these nodes will not be affected by the change of weight on $e_{i,j}$. When there are two or more edges modified, A_i may change only for those nodes in the union of fan-out cones, C.
Q.E.D.

Since $|C|$ is less than $|V|$, the computation time can be saved by restricting the application of the longest path algorithm to C, instead of the full graph G'. This leads to the following simple incremental longest path algorithm:

Algorithm ILP1(G, G')

```
1.  Apply a depth-first search from edges in $G' - G$ to generate
    the fan-out cone, $C$.
2.  For each node $V_i$ in C,
    a.  If all the predecessors of $V_i$ are inside the cone $C$, initialize $A_i$ to $-\infty$.
    b.  Otherwise, initialize $A_i$ to $\max_j\{A_j + \Lambda_{j,i} \mid V_j \notin C\}$
```

المنارة للاستشارات
www.manaraa.com

3. Apply the longest path algorithm on the nodes inside C.

Lemma 2 (Monotonicity)

Let A_i and A'_i represent respectively the longest path lengths from V_0 to V_i in G and G'.

1. If $\Lambda_{i,j} \le \Lambda'_{i,j}$ for every edge, then $A_i \le A'_i$.
2. If $\Lambda_{i,j} \ge \Lambda'_{i,j}$ for every edge, then $A_i \ge A'_i$.

Proof: Let $L(p)$ and $L'(p)$ be the path lengths of p in G and G' respectively. For the first case, $L(p) \le L'(p)$ for every path p, and hence $A_i \le A'_i$. The second case can be proved in the similar way. **Q.E.D.**

3. A new incremental longest path method

According to Lemma 1, C_C is a subset of C. Some nodes inside the cone C may be dominated by the signal arrival times at nodes outside C and therefore may not be affected by the change in $G' - G$, such as node V_{14} in Fig. 3. This motivates us to construct new algorithms by restricting the path search within C_C in order to further cut down the computation complexity.

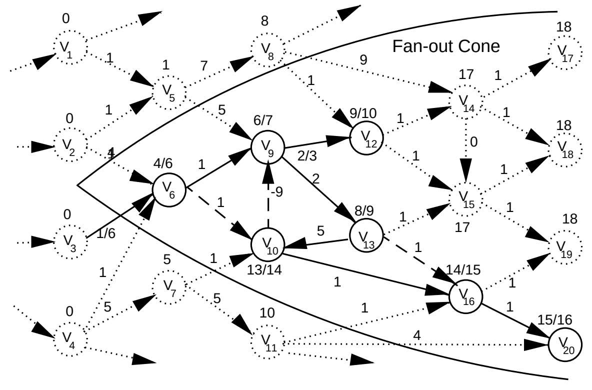

Figure 3 : Numbers associated with edges are weights, while numbers associated with nodes are longest path lengths (labels). The change in edge weights and node labels are shown as two numbers separated by / .

3.1. The case with positive changes only.

This is the case with $E^{(+)} \ne$ NULL, and $E^{(-)}$ = NULL. In such a situation, $\Lambda'_{i,j} > \Lambda_{i,j}$ on edges of $E^{(+)}$, and thus according to Lemma 2, A_i will be a lower bound for A'_i. Therefore, when searching for new longest paths in G', we may ignore those paths which have path lengths less than or equal to A_i. Edges $e_{i,j} = (V_i, V_j)$ in $E^{(+)}$ can be divided into two cases:

1. Case $A_i + \Lambda'_{i,j} \le A_j$: The new constraint $\Lambda'_{i,j}$ on $e_{i,j}$ is satisfied.
2. Case $A_i + \Lambda'_{i,j} > A_j$: The new constraint $\Lambda'_{i,j}$ on $e_{i,j}$ is violated. Such edges will be used to drive the search for new longest paths inside C_C and will be called ***driving*** edges.

Since weight changes on edges belonging to Case 1 will not affect A_i, these edges may be dropped from the set $E^{(+)}$. A queue $Q^{(0)}$, constructed from the fan-in nodes of remaining driving edges, will then be used to guide a ***dynamic breadth-first search*** in C_C for new arrival times. In order to make a breadth-first traversal on a graph which may contain loops, we need to create directions for edges encountered. This is done by assigning a breadth-first search number, $bfs(V_j)$, to each node V_j according to the order in which it is added to the queue. Forward-directed edges are edges (V_i, V_j) with $bfs(V_i) < bfs(V_j)$, while backward-directed edges are edges (V_i, V_j) with $bfs(V_i) > bfs(V_j)$. It is clear that forward-directed edges form a directed acyclic graph. The breadth-first traversal is performed on the forward-directed edges to update A_i. When a backward-directed edge is encountered, its fan-in node may be added to the output queue $Q^{(1)}$. When the forward traversal is completed, the breadth-first traversal in the reverse direction is started with $Q^{(1)}$. This process takes a few number (typically less than 10) of iterations to converge, if G' does not contain positive loops. If loops appear on the dominance graph T, then they all must have positive gains, and need to be reported as timing violations.

Algorithm DrivePositive$(E^{(+)})$

```
1. Set the iteration counter $m$=0.. Generate the queue $Q^{(0)} = \{V_i \mid e_{i,j} = (V_i, V_j) \in E^{(+)}$ and $e_{i,j}$ is a driving edge$\}$,
2. Repeat the following:
   a. For each node $V_i$ in queue $Q^{(m)}$, set $bfs(V_i)$ respectively to from 1 to $|Q^{(m)}|$.
   b. Set the output queue $Q^{(m+1)}$ to NULL.
   c. Pop the top node $V_i$ out of $Q^{(m)}$. For each fan-out edge ($V_i$, $V_j$), do
      1) Case $bfs(V_i) < bfs(V_j)$: if $A_i + \Lambda'_{i,j} > A_j$, then
         a) Set $A_j$ to $A_i + \Lambda'_{i,j}$, and the dominance predecessor pointer $t_j$ to $V_i$.
         b) If $V_j$ is not in queue $Q^{(m)}$, add $V_i$ to the bottom of $Q^{(m)}$. Increase $|Q^{(m)}|$ by 1 and set $bfs(V_j)$ to $|Q^{(m)}|$.
      2) Case $bfs(V_i) > bfs(V_j)$: if $V_i$ is not in $Q^{(m+1)}$, add $V_i$ to the top of $Q^{(m+1)}$.
   d. If $m \ge 10$, search loops in the dominance graph $T$. If loops are found, report positive loops, and exit.
   e. Increase $m$ by 1.
3. Stop when $Q^{(m)}$ is empty.
```

We shall illustrate the above algorithm with the example in Fig. 3. This graph contains a loop $V_9V_{13}V_{10}V_9$. There are two edges in $E^{(+)}$: the weight on edge (V_3, V_6) is increased from 1 to 6, and the weight on edge (V_9, V_{12}) is increased from 2 to 3. Since the old longest path lengths to V_3, V_6, V_9, and V_{12} are respectively 0, 4, 6, and 9, constraint 6 on edge (V_3, V_6) is violated, while constraint 3 on edge (V_9, V_{12}) is satisfied. So $Q^{(0)}$ is set to $\{V_3\}$. In the first breadth-first search, we traverse nodes V_6, V_9, V_{12} and V_{13}, and update their labels A_i respectively to 6, 7, 10, and 9. Since a backward-directed edge (V_{13}, V_{10}) is encountered, $Q^{(1)}$ is set to $\{V_{13}\}$. This leads to A_{10} =14 during the second iteration, and $Q^{(2)}$ is set to $\{V_{10}\}$. During the third iteration, we traverse nodes V_{16} and V_{20}, and update their labels to 15 and 16. Now the algorithm converges, since $Q^{(3)}$ becomes empty. So $C_C = \{V_6, V_9, V_{10}, V_{12}, V_{13}, V_{16}, V_{20}\}$, and $|C_C|$ = 7 is less than $|C|$ = 12.

3.2. The case with negative changes only.

This is the case with $E^{(-)} \ne$ NULL, and $E^{(+)}$ = NULL. Here we would like to utilize the dominance graph T to speed up the search of the new longest path length A'_i.

www.manaraa.com

Definition 3: Let $e_{i,j} = (V_i, V_j)$ be an edge in $E^{(-)}$ T, and $C_D(e_{i,j})$ be the ***dominance fan-out cone*** consisting of nodes to each of which there is a directed path in T from $e_{i,j}$. Then the dominance fan-out cone from $E^{(-)}$ is defined as the union: $C_D = \{C_D(e) \mid e \in E^{(-)}\ \ T\}$. For an example, see Fig. 4.

Lemma 3

For the case $E^{(+)}$ = NULL, we have $A_i = A'_i$ for nodes $V_i \notin C_D$. In other words, C_C is a subset of C_D.

Proof: Since $\Lambda'_{i,j} < \Lambda_{i,j}$, A_i becomes an upper bound to A'_i according to Lemma 2. Let the longest path from V_0 to V_i in G be p_i, i.e., $A_i = L(p_i)$. If $V_i \notin C_D$, then p_i does not encounter any edge from $E^{(-)}$, and A_i, being the path length of p_i in G', is also a lower bound to A'_i. **Q.E.D.**

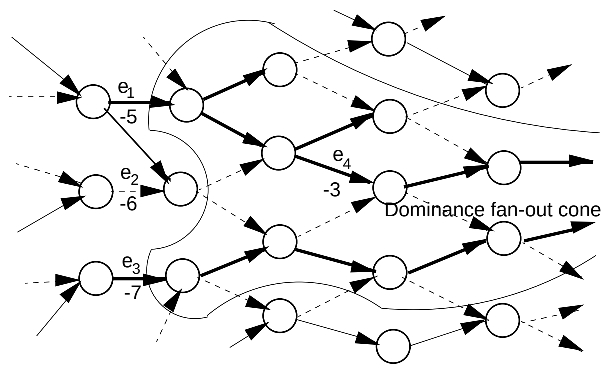

Figure 4. The edges modified are labelled with the weight changes. T is shown by edges in solid lines, while C_D is covered by edges in heavy solid lines. e_2 is not on T, and will not affect the timing.

Thus we need only to concentrate our efforts in finding the new longest path leading to nodes in C_D. In the following discussion, we shall use p_i to represent the old longest path from V_0 to V_i in G. For a node $V_i \in C_D$, $A_i = L(p_i)$ is no longer the path length of p_i in G', because p_i must encounter edges from $E^{(-)}$. For example, in Fig. 4, the path lengths to nodes in $C_D(e_1) - C_D(e_4)$ are reduced by 5, while the path lengths to nodes in $C_D(e_4)$ are reduced by 5+3.

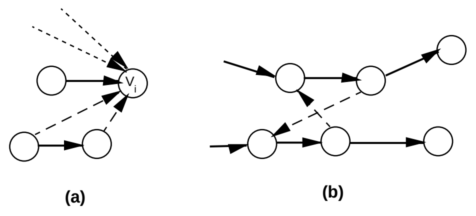

Figure 5. Dotted, solid, and dashed lines represent respectively side, tree, and cross edges. (a)Fan-in edges of node V_i. (b)Loops formed from cross and tree edges.

To calculate the new longest path length to a node $V_i \in C_D$, we observe that the last edge on such a path must be a fan-in edge of V_i. The fan-in edges of V_i fall into the following three types as illustrated in Fig. 5(a):

1. ***Side edges*** ($V_j \notin C_D$): The fan-in nodes of these edges are outside the cone C_D.
2. ***Tree edges*** ($V_j \in C_D$ and $(V_j, V_i) \in T$): These edges are on the dominance tree T.
3. ***Cross edges*** ($V_j \in C_D$ and $(V_j, V_i) \notin T$): These are edges between nodes inside the cone C_D, but not part of T.

For a side edge (V_j, V_i), $A_j = A'_j = L(p_j)$ according to Lemma 3, and $A_j + \Lambda_{j,i}$ is the length for the path consisting of p_j and edge (V_j, V_i). For a tree edge (V_j, V_i), the path length of p_i in G' is the sum of weights along p_i, which may be evaluated by a breadth-first traversal of nodes in C_D along edges in T. For a cross edge (V_j, V_i), a path length to V_i from this edge can not be directly derived from A_j, since A_j may not be equal to any path length in G'. For the moment, let us ignore cross edges, and define A''_i as the maximum of new path lengths among paths leading to node V_i through side and tree edges. Then for nodes inside C_D, A''_i, being the length of a path in G', is a lower bound of A'_i.

If p_i contains several edges from $E^{(-)}$ T, then the path length of p_i will be affected by the weight changes on all these edges, for example, the fan-out node of e_4 in Fig. 4. To facilitate a systematical calculation of new path lengths A''_i, a breadth-first ordering of edges in $E^{(-)}$ T is established first. That is, if there is a directed path in T from edge $e_{i,j}$ to edge $e_{k,l}$, then edge $e_{i,j}$ is placed before edge $e_{l,k}$. C_D and the ordering are constructed in Step 1 of Algorithm *DriveNegative*.

To calculate $\{A''_i\}$, we take edges from $E^{(-)}$ T according to the sort order, and make a breadth-first traversal of their descendent nodes along T. For each node encountered, its fan-in edges are examined. The path lengths resulting from side and tree edges are calculated, and A''_i is set to the maximum of these lengths in Step 2 of Algorithm *DriveNegative*.

Another reason for dropping cross edges during the derivation of $\{A''_i\}$ is that these cross edges may form loops with tree edges, see Fig. 5(b), and the breadth-first traversal method will break down on these loops. Now, after obtaining A''_i, we need to check these cross edges (V_j, V_i) to see whether the corresponding constraint, $A''_j + \Lambda'_{j,i} \le A''_i$ are violated, and collect those edges with violated constraints into a set $E^{(v)}$. If $E^{(v)}$ is empty, then $\{A''_i\}$ is indeed the longest path length set in G'. If $E^{(v)}$ is not empty, then define a new graph G'' as follows:

$$\Lambda''_{j,i} = \left\{ \begin{array}{ll} -\infty & \text{for edges } (V_j, V_i) \in E^{(v)} \\ \Lambda'_{j,i} & \text{for edges } (V_j, V_i) \notin E^{(v)} \end{array} \right\}$$

Then clearly constraints on all edges of G'' are satisfied, and $\{A''_i\}$ is the longest path length set in G''. The derivation of $E^{(v)}$ is done in Steps 3-4 of Algorithm *DriveNegative*.

Since $\Lambda''_{i,j} \le \Lambda'_{i,j}$, $E^{(v)}$ is a set of positive driving edges and Algorithm *DrivePositive* may be used to complete the derivation of $\{A'_i\}$, the longest path length set in G'. Note that this *DrivePositive* will only change A_i for nodes inside C_D.

Algorithm DriveNegative$(E^{(-)}, E^{(v)})$

```
1.  For each node $V_j$ in $\{V_j \mid (V_i, V_j) \in E^{(-)}\ \ T\}$, do
    a.  Create a queue $Q = \{V_j\}$, and mark $V_j$ as cone-member of $C_D$.
    b.  While $Q$ is not empty, do
        1)  Pop the top node $V_l$ out of $Q$.
        2)  For each edge $e_{l,k} = (V_l, V_k)$ in $T$, do
```

www.manaraa.com

```
            a)  If $(V_l, V_k) \in E^{(-)}$   $T$, place $e_{l,k}$ after $e_{i,j}$ in the sort order.
            b)  Else if $V_k$ is not marked as cone-member, do so and add $V_k$ to the bottom of $Q$.
2.  For each edge $(V_i, V_j)$ in $E^{(-)}$   $T$ do,
      If $V_j$ is not marked as visited, then
    a.  Create a queue $Q = \{V_j\}$.
    b.  While $Q$ is not empty, do
        1)  Pop the top node $V_l$ out of $Q$, and mark it as visited.
        2)  Find the dominance predecessor, $t_l = V_{k0}$, and set $A''$ to $A_{k0} + \Lambda_{k0,l}$.
        3)  For each side edge $\{e_{k,l} = (V_k, V_l) \mid V_k \notin C_D\}$ do,
              If $A_k + \Lambda_{k,l} > A''$, set $A''$ to $A_k + \Lambda_{k,l}$, and $t_l$ to $A_k$.
        4)  If $A'' = A_l$, then continue.
        5)  Set $A_l$ to $A''$.
        6)  For each fan-out edge $\{(V_l, V_m) \in T\}$, if $V_m$ is not marked as visited, add $V_m$ to $Q$.
3.  Set $E^{(v)}$ = NULL.
4.  For each node $V_i \in C_D$, do
      For each fan-out edge, $e_{i,j} = (V_i, V_j)$, do
      If $V_j \in C_D$, $t_j \neq V_i$, and $A_j - A_i < \Lambda'_{i,j}$, then add $e_{i,j}$ to $E^{(v)}$.
```

It can be shown that the computation complexity of Algorithm *DriveNegative* is $O(|C_D|)$. We shall illustrate the algorithm with the example in Fig. 3. Let us reverse the changes: the weight on edge (V_3, V_6) is decreased from 6 to 1, and the weight on edge (V_9, V_{12}) is decreased from 3 to 2. These are two edges in $E^{(-)}$. The edges in solid lines show the longest path tree T as it fans out from $E^{(-)}$. Clearly $E^{(-)}$ $T = E^{(-)}$. During Step 1 of the algorithm, the cone members of C_D are derived as those nodes circled by solid lines, and the sorting on $E^{(-)}$ is done with edge (V_3, V_6) followed by edge (V_9, V_{12}). During Step 2 of the algorithm, we traverse through nodes in cone C_D, and use side edges(dotted lines) and tree edges(solid lines) to update node labels. The labels on V_6, V_9, V_{10}, V_{12}, V_{13} V_{16}, and V_{20} are changed back respectively to 4, 6, 13, 9, 8, 14, and 15. During Step 3-4 of the algorithm, cross edges(dashed lines) are checked, and no driving edge is found. So $E^{(v)}$ = NULL, and we have derived the solution.

3.3. The general case

This is the case with $E^{(+)} \neq$ NULL, and $E^{(-)} \neq$ NULL. For this general case, Lemma 3 is revised in the following form:

Lemma 4

Let C_D be the dominance fan-out cone from $E^{(-)}$ T. Then we have $A_i \leq A'_i$ for nodes $V_i \notin C_D$.

Proof: For those nodes V_i outside the C_D of $E^{(-)}$, the longest path p_i leading to node V_i in G does not encounter any edge from $E^{(-)}$. Hence A_i, being the path length of p_i in G, can not be greater than the new path length of p_i in G', and is a lower bound for A'_i, the longest path length in G'. **Q.E.D.**

However, for those nodes V_i inside C_D, A_i may not be a lower bound for A'_i. Algorithm *DriveNegative* may be used to generate lower bounds for these nodes, and a set of driving edges, $E^{(v)}$. Then merge edges from $E^{(v)}$ with those from $E^{(+)}$, and use Algorithm *DrivePositive* to derive $\{A'_i\}$. If no loop is found in T, then we reach our final solution A'_i. On the other hand, if loops are found in T, then they must all be loops with positive gains which will be reported as timing violations. In such cases, in order to find meaningful arrival times and slacks, we need to modify the graph G' to remove these loop violations. This may be accomplished by either deleting an edge on the loop, or decreasing the weight of one edge such that the loop gain becomes non-positive. For example, a latch on the loop can be set in the test mode with the corresponding edge for the internal delay removed. The edges deleted or the edges with weights reduced then contribute to $E^{(-)}$, the set of edges with negative change in the modified graph. We need to make another round of iteration to find the timing for the modified graph. This process may be continued until all positive loops are broken.

```
Algorithm ILP2($E^{(+)}$, $E^{(-)}$)
 Repeat
1.  DriveNegative($E^{(-)}$, $E^{(v)}$).
2.  Set $E^{(-)}$=NULL, and merge $E^{(+)}$ and $E^{(v)}$ into $E^{(+)}$   $E^{(v)}$.
3.  DrivePositive ($E^{(+)}$   $E^{(v)}$).
4.  For each loop found in $T$, break an edge, and collect the edge into a new set $E^{(-)}$.
 Until $E^{(-)}$ is empty.
```

4. Results and Discussion

We have implemented these algorithms into CYCLOPSS [5], and run them through ISCAS'89 benchmark circuits and one moderately large industrial chip example. The chip example contains 50,000 gates, and has a cycle time of   = 7.0 ns. For the ISCAS'89 circuits, we adopted the transformed version [6] in which a complementary two-phase clocking scheme is employed to control level-sensitive latches. For the cycle time, we use   = 1.2 $_{\min}$, where $_{\min}$ is the minimum cycle time for the circuit. Our experiments start with a full timing analysis using both the latch-graph(L) based algorithm, and the full-graph(G) longest path algorithm. Each analysis consists of two runs, a forward run through the timing constraint graph (L or G) to derive the arrival time, A_i, and a backward run through the graph to derive the ***required arrival time***, R_i. The slack is calculated as $R_i - A_i$. The characteristic values and CPU times on a 40 MIP machine for seven ISCAS'89 circuits and the chip example are listed in Table 1.

Each circuit is then subject to about 60 consecutive incremental changes, among which, half contain negative change in edge weights, and the other half contain positive change in edge weights. Each incremental change involves the weight modification of all the fan-out edges from a randomly selected set of nodes. (The size, K, of this node set ranges from 1, 10, 100, 1,000, 10,000, to 100,000 nodes.) If the node picked for the incremental change is the source pin of a net, this corresponds to a change of the source-to-sink delay of the net. If the node picked is the input pin of a gate, this corresponds to a change of internal gate delay of the pin. In both cases, the weight change is selected with a random number generator which has a mean value 0 and a variance 0.2 . After each change, both *ILP*1 and *ILP*2 are used to incrementally update arrival times and slacks. Fig. 6 shows the plot of the CPU times of the two algorithms versus K using the logarithmic scales in both axes for the circuit, s35932. In this plot, points marked with '.' and 'o' are respectively the CPU times of *ILP*1 for positive and negative weight changes, while points marked with '+' and 'x' are respectively the CPU times of *ILP*2 for positive and negative weight changes. More experimental data for the CPU running times of Algorithm *ILP*1 and

المنارة للاستشارات
www.manaraa.com

*ILP*2 are respectively presented in Table 2 and Table 3. Each entry in these tables shows the average CPU times of incremental timing runs for a sample of 10 different circuit modifications.

circuit name	*gate count*	*latch count*	*cycle* min	*cycle*	*node count*	*edge count*	*T*1 sec	*T*2 sec
s27	26	6	8	9.6	78	86	0.02	0.00
s1423	1462	148	80	96.0	3982	4962	6.3	1.21
s5378	5916	358	32.7	39.2	14866	17662	8.4	3.24
s9234	11650	456	76	91.2	28130	32840	23.6	7.64
s13207	17240	1338	92	110.4	41212	47578	29.4	12.87
s35932	35586	3456	54	64.8	96290	120628	41.4	24.04
s38584	41410	2904	70	84.0	110406	137388	85.4	34.16
chip	50236	13131	–	7.0	249267	318202	721.5	74.96

Table 1: Characteristic values, and CPU times for full timing analysis. Column *T*1 is from a latch-graph based algorithm. Column *T*2 is from a full-graph longest path algorithm.

Algorithm *ILP*1 runs only moderately faster than the full timing algorithm with a speed-up ranging from a few percent to a factor of 5, See Table 2. Therefore, *ILP*1, based on the concept of the simple fan-out cone, is not a very powerful algorithm. On the other hands, Algorithm *ILP*2 runs significantly faster than *ILP*1. For both circuit s35932 and the chip example, the ratios of CPU times for a full timing analysis (T_2) to that of *ILP*2 are about 10,000 times for K=1, 1,000 times for K=10, 100 times for K=100, 10 times for K=1,000, and a few times for K=10,000. From Table 3, we noticed that for small incremental changes involving $K \le 10$ nodes, the CPU times of *ILP*2, being in the order of hundredth of seconds, seem less sensitive to the sizes of the circuits. This corresponds to a speed-up of more than three orders of magnitude relative to the full timing analysis algorithm for large circuits. Even for incremental changes involving as large as K=100 to 1,000 nodes, the speed-up relative to the full timing analysis algorithm is still as high as 10 to 100. Therefore, Algorithm *ILP*2 can be used effectively under the interactive environment, in which designers need to make frequent design changes and quickly find the timing change. The dramatic speed of Algorithm *ILP*2 also makes it an ideal timing tool for coupling to a logic synthesis program, since, with such a fast incremental timer, the synthesis program can afford to evaluate a tremendous number of circuit modifications before converging to the final circuit implementation.

We would like to point out that our algorithms may also be used to solve the incremental shortest path problem. This can be achieved by making the following transformations in graphs G and G': $\Lambda_{i,j} \rightarrow -\Lambda_{i,j}$, $\Lambda'_{i,j} \rightarrow -\Lambda'_{i,j}$, $A_i \rightarrow -A_i$, and $A'_i \rightarrow -A'_i$. The shortest path problem corresponds to the early-mode timing problem under the conservative constraints [1,2]

Acknowledgment

We would like to thank K. Belkhale of IBM Fishkill for helpful discussions and suggestions.

References

1. K. A. Sakallah, T. N. Mudge, and O. A. Olukotun, "Check Tc and min Tc: Timing verification and optimal clocking of synchronous digital circuits," Proc. ICCAD, pp. 552-555, Nov 1990.
2. T. G. Szymanski, and N. Shenoy, "Verifying clock schedules," proc. ICCAD, pp. 124-131, Nov. 1992
3. R. S. Tsay and Ichiang Lin, "A system timing verifier for multiple-phase level-sensitive clock design," Research Report RC 17272, IBM Yorktown, 1991.
4. T. M. Burks, K. A. Sakallash, T. N. Mudge, "Identification of critical paths in circuits with level-sensitive latches," Proc. ICCAD, pp. 137-141, Nov. 1992.
5. J. F. Lee, D. T. Tang and C. K. Wong, "A timing analysis algorithm for circuits with level-sensitive latches," to be published in Proc. ICCAD, Nov. 1994.
6. T. G. Szymanski, "Computing optimal clock schedules," Proc. 29 th Design Automation conference, pp. 399-404, 1992.
7. E. L. Lawler, "Combinational Optimization: Networks, and Matroids," Holt, Rinehart and Winston 1976.

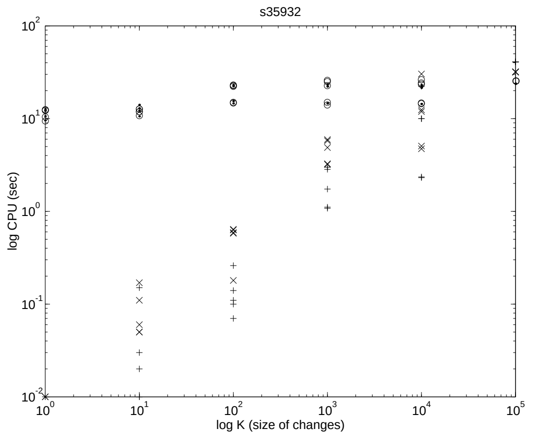

Figure 6. CPU times of incremental algorithms for s35932.

circuit name	$K = 1$ sec	$K = 10$ sec	$K = 10^2$ sec	$K = 10^3$ sec	$K = 10^4$ sec	$K = 10^5$ sec	*T*2 sec
s27	0.002	0.000	–	–	–	–	0.00
s1423	0.112	0.195	0.208	0.205	–	–	1.21
s5378	0.891	1.160	1.493	1.559	1.789	–	3.24
s9234	1.875	5.333	6.608	6.903	7.526	–	7.64
s13207	3.821	7.412	10.067	11.582	12.005	–	12.87
s35932	11.418	12.217	19.562	19.954	19.860	11.52	24.04
s38584	22.976	26.251	29.098	29.093	29.683	24.48	34.16
chip	12.759	21.547	23.072	27.159	34.803	48.30	74.96

Table 2: Average CPU times for running Algorithm *ILP*1.

circuit name	$K = 1$ sec	$K = 10$ sec	$K = 10^2$ sec	$K = 10^3$ sec	$K = 10^4$ sec	$K = 10^5$ sec	*T*2 sec
s27	0.001	0.000	–	–	–	–	0.00
s1423	0.001	0.002	0.073	0.279	–	–	1.21
s5378	0.002	0.022	0.174	1.028	2.205	–	3.24
s9234	0.005	0.094	0.540	3.219	6.829	–	7.64
s13207	0.008	0.031	0.277	2.568	6.872	–	12.87
s35932	0.003	0.066	0.330	3.283	11.14	26.41	24.04
s38584	0.003	0.020	0.388	2.201	10.97	28.45	34.16
chip	0.001	0.028	0.308	2.798	14.61	42.88	74.96

Table 3: Average CPU times for running Algorithm *ILP*2.

www.manaraa.com