

# UI-Timer: An Ultra-Fast Clock Network Pessimism Removal Algorithm

Tsung-Wei Huang\*, Pei-Ci Wu†, and Martin D. F. Wong‡

\*twh760812@gmail.com, †peiciwu@gmail.com, ‡mdfwong@illinois.edu

Department of Electrical and Computer Engineering, University of Illinois at Urbana-Champaign, IL, USA

**Abstract**—The recent TAU computer-aided design (CAD) contest has aimed to seek novel ideas for accurate and fast clock network pessimism removal (CNPR). Unnecessary pessimism forces the static-timing analysis (STA) tool to report worse violation than the true timing properties owned by physical circuits, thereby misleading signoff timing into a lower clock frequency at which circuits can operate than actual silicon implementations. Therefore, we introduce in this paper UI-Timer, a powerful CNPR algorithm which achieves exact accuracy and ultra-fast runtime. Unlike existing approaches which are dominated by explicit path search, UI-Timer proves that by implicit path representation the amount of search effort can be significantly reduced. Our timer is superior in both space and time saving, from which memory storage and important timing quantities are available in constant space and constant time per path during the search. Experimental results on industrial benchmarks released from TAU 2014 CAD contest have justified that UI-Timer achieved the best result in terms of accuracy and runtime over all participating timers.

## I. INTRODUCTION

The lack of accurate and fast algorithms for clock network pessimism removal (CNPR) has been recently pointed out as a major weakness of existing static-timing analysis (STA) tools [10]. Conventional STA tools rely on conservative dual-mode operations to estimate early-late and late-early path slacks [7]. This mechanism, however, imposes unnecessary pessimism due to the consideration of delay variation along common segments of clock paths, as illustrated in Figure 1. Unnecessary pessimism may lead to tests being marked as failing whereas in actuality they should be passing. Thus designers and optimization tools might be misled into an over-pessimistic timing report. Therefore, the goal of this paper is to identify and eliminate unwanted pessimism during STA so as to prevent true timing properties of circuits from being skewed.

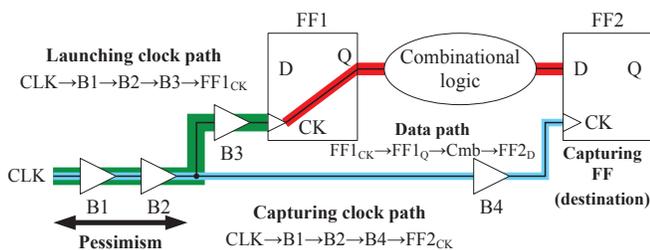


Figure 1. Clock network pessimism incurs in the common path between the launching clock path and the capturing clock path.

The importance and impact of CNPR are demonstrated in Figure 2. It is observed that the number of failing tests was reduced from 642 to less than half after the pessimism was removed. Unwanted pessimism might force designers and optimization tools to waste a significant yet unnecessary amount of efforts on fixing paths that meet the intended clock frequency. Such a problem becomes even critical when design comes to deep submicron era where data paths are shorter, clocks are faster, and clock networks are longer to accommodate larger and complex chips. Moreover, without pessimism removal designers

and CAD tools are no longer guaranteed to support legal turnaround for timing-specific improvements, which dramatically degrades the productivity. At worst, signoff timing analyzer gives rise to the issue of “leaving performance on the table” and concludes a lower frequency at which the circuits can operate than their actual silicon implementations [14].

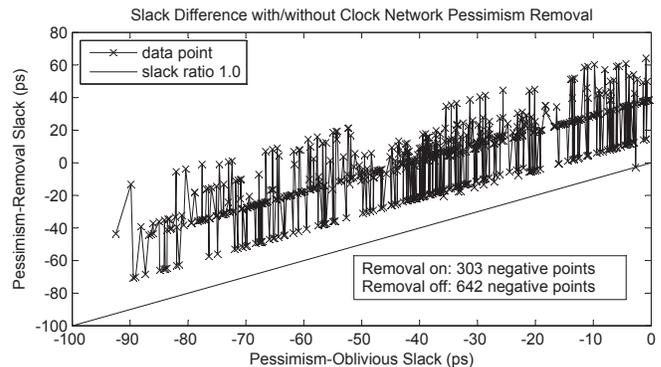


Figure 2. Impact on clock network pessimism from a circuit in [1].

State-of-the-art CNPR algorithms are dominated by straightforward path-based methodology [6], [9], [12], [14]. Critical paths are identified without considering the pessimism first. Then for each path the common segment is found by a simple walk through the corresponding launching clock path and capturing clock path. Finally, slack of each path is adjusted by the amount of pessimism on the common segment. The real challenge is the amount of pessimism that needs to be removed is path-specific. The most critical path prior to pessimism removal is not necessarily reflective of the true counterpart (see the data-point line in Figure 2), revealing a potential drawback that path-based methodology has the worst-case performance of exhaustive search space in peeling the true critical paths. Accordingly, prior works are usually too slow to handle complex designs and unable to always identify the true critical path exactly.

In this paper we introduce UI-Timer, a powerful CNPR algorithm which achieves exact accuracy, ultra-fast runtime, and low memory requirement. Our contributions are summarized as follows: 1) We introduce a theoretical framework that maps the CNPR problem to a graph search formulation. The mapping allows the true critical path to be directly identified through our search space, rather than the time-consuming yet commonly-applied strategy which interleaves the search between slack computation and pessimism retrieval. 2) Unlike predominant explicit path search, we represent the path implicitly using two efficient and compact data structures, namely suffix tree and prefix tree, and yield a significant saving in both search space and search time. 3) The effectiveness and efficiency of our timer have been verified by TAU 2014 CAD contest [1], [10]. Comparatively, UI-Timer confers a high degree of differential over all participating timers in terms of accuracy and runtime.

## II. STATIC TIMING ANALYSIS

STA is a method of verifying expected timing characteristics of a circuit. The dual-mode or early-late timing model is the most popular convention because it accounts for various within-chip variations such as temperature fluctuations and voltage drops [7]. The earliest and latest timing instants that a signal reaches are quantified as earliest and latest *arrival time* ( $at$ ), while the limits imposed on a circuit node for proper logic operations are quantified as earliest and latest *required arrival time* ( $rat$ ). The verification of timing at a circuit node is determined by the largest difference or *worst slack* between the required arrival time and signal arrival time. In this paper, we focus on two primary types of timing verification – *hold test* and *setup test* for a specified flip-flop (FF). Considering a hold test or setup test  $t$ , the following equations are applied for STA [1].

$$rat_t^{early} = at_o^{late} + T_{hold}, \quad rat_t^{late} = at_o^{early} + T_{clk} - T_{setup} \quad (1)$$

$$slack_{worst}^{hold} = at_d^{early} - rat_t^{early}, \quad slack_{worst}^{setup} = rat_t^{late} - at_d^{late} \quad (2)$$

Notice that  $T_{clk}$  is the clock period,  $T_{hold}$  and  $T_{setup}$  are values of hold and setup constraints, and  $o$  and  $d$  are respectively the clock pin and the data pin of the testing FF. In general, the best-case fast condition is critical for hold test and the worst-case slow condition is critical for setup test. For a data path feeding the testing FF, a positive slack means the required arrival time is satisfied and a negative slack means the required arrival time is in a violation.

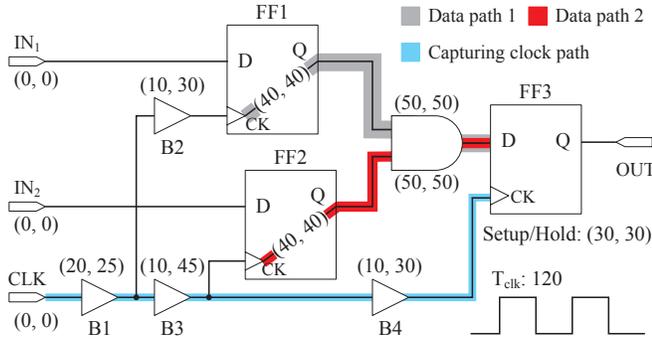


Figure 3. An example of sequential circuit network.

Consider a sample circuit in Figure 3, where two data paths feed a common FF. Numbers enclosed within parentheses denote the earliest and latest delay of a circuit node. Assuming all wire delays and arrival times of primary inputs are zero, we perform the setup test on FF3. The latest required arrival time of FF3 is obtained by subtracting the values of clock period plus the earliest arrival time at the clock pin of FF3 from the value of setup constraint, which is equal to  $(120 + (20 + 10 + 10)) - 30 = 130$ . The respective latest arrival times of data path 1 and data path 2 at the data pin of FF3 are  $25 + 30 + 40 + 50 = 145$  and  $25 + 45 + 40 + 50 = 160$ . Using equation (2), the setup slacks of data path 1 and data path 2 are  $130 - 145 = -15$  (failing) and  $130 - 160 = -30$  (failing), respectively.

## III. CLOCK NETWORK PESSIMISM REMOVAL

The dual-mode split-timing analysis has greatly enabled timers to effectively account for any within-chip variation effects. However, the dual-mode analysis inherently embeds unnecessary pessimism, which results in an over-conservative design. Take the slack of data path 1 in Figure 3 for example. The pessimism arises with buffer B1 since it was accounted for both earliest and latest delays at the same

time which is physically impossible. In general, the pessimism of two circuit nodes appears in the common path from the clock source to the the closest point to which the two nodes converge through upstream traversal. Such point is also referred to as *clock reconverging node*. The true timing without pessimism can be obtained by adding the final slack to a credit which is defined as follows [1]:

$$credit_{u,v}^{hold} = at_{cp}^{late} - at_{cp}^{early} \quad (3)$$

$$credit_{u,v}^{setup} = at_{cp}^{late} - at_{cp}^{early} - (at_r^{late} - at_r^{early}) \quad (4)$$

Notice that  $r$  is the clock source and  $cp$  is the clock reconverging node of nodes  $u$  and  $v$ . Since setup test compares the data point against the clock point in the subsequent clock cycle, the credit rules out the arrival time at the clock source [1]. The slack prior to common-path-pessimism removal (CPPR) is referred to as *pre-CPPR slack* or *post-CPPR slack* otherwise. For the same instance in Figure 3, the credits of data path 1 and data path 2 for setup test are respectively 5 and 40, which in turn tell their true slacks being  $-15 + 5 = -10$  (failing) and  $-30 + 40 = 10$  (passing). A key observation here is that the most critical pre-CPPR slack (data path 2) is not necessarily reflective of the true critical path (data path 1). Analyzing the single-most critical path during CPPR is obviously insufficient. In practice, reporting a number of ordered critical paths for a given test rather than merely the single-most critical one is relatively necessary and important.

## IV. PROBLEM FORMULATION

The circuit network is input as a directed-acyclic graph (DAG)  $G = \{V, E\}$ .  $V$  is the node set with  $n$  nodes which specify pins of circuit elements (e.g., primary IO, logic gates, FFs, etc.).  $E$  is the edge set with  $m$  edges which specify pin-to-pin connections. Each primary input, i.e., the node with zero indegree, is assigned by an earliest arrival time and a latest arrival time. Each edge  $e$  or  $e_{u \rightarrow v}$  is directed from its tail node  $u$  to head node  $v$  and is associated with a dual tuple of earliest delay  $delay_e^{early}$  and latest delay  $delay_e^{late}$ . A path is an ordered sequence of nodes  $\langle v_1, v_2, \dots, v_n \rangle$  or edges  $\langle e_1, e_2, \dots, e_n \rangle$  and the path delay is the sum of delays through all edges. In this paper, we are in particular emphasizing on the data path, which is defined as a path from the clock source pin of an FF to the data pin of another FF. The arrival time of a data path is the sum of its path delay and arrival time from where this data path originates. The clock tree is a subgraph of  $G$  which distributes the clock signal with clock period  $T_{clk}$  from the tree root  $r$  to all the sequential elements that need it. A test is defined with respect to an FF as either a hold check or setup check to verify the timing relationship between the clock pin and the data pin of the FF, so that the hold requirement  $T_{hold}$  or setup requirement  $T_{setup}$  is met. We refer to the testing FF as *destination FF* and those FFs having data paths feeding the destination FF as *source FFs*. Using the above knowledge, the CNPR problem is formulated as follows:

**Objective:** Given a circuit network  $G$  and a hold or setup test  $t$  as well as a positive integer  $k$ , the goal is to identify the top  $k$  critical paths (i.e., data paths that are failing for the test) from source FFs to the destination FF in ascending order of post-CPPR slack.

## V. ALGORITHM

The overall algorithm of UI-Timer is presented in Algorithm 1. It consists of two stages: *lookup table preprocessing* and *pessimism-free path search*. The goal of the first stage is to tabulate the common path information for quick lookup of credit, while the goal in the

second stage is to identify the top- $k$  critical paths in a pessimism-free graph derived from a given test. We shall detail in this section each stage in bottom-up fashion.

---

**Algorithm 1:** UI-Timer( $t, k$ )

---

**Input:** test  $t$ , path count  $k$

**Output:** solution set  $\Psi$  of the top- $k$  critical paths

- 1 BuildCreditLookupTable();
  - 2  $G_p \leftarrow$  pessimism-free graph for the test  $t$ ;
  - 3  $\Psi \leftarrow$  GetCriticalPath( $G_p.source, G_p.destination, k$ );
  - 4 **return**  $\Psi$ ;
- 

*A. Lookup Table Preprocessing*

In graph theory, the clock reconverging node of two nodes in the clock tree is equivalent to the lowest common ancestor (LCA) of the two nodes. The arrival time information of each node in the clock tree can be precomputed and therefore the credit of two nodes can be obtained immediately once their LCA is known. Many state-of-the-art LCA algorithms have been invented over the last decades. The table-lookup algorithm by [5] is employed as our LCA engine due to its simplicity and efficiency. For a given clock tree, we build three tables as follows:

- The Euler table  $E$  records the identifiers of nodes in the Euler tour of the clock tree;  $E[i]$  is the identifier of  $i^{th}$  visited node.
- The level table  $L$  records the levels of nodes visited in the Euler tour;  $L[i]$  is the level of node  $E[i]$ .
- The occurrence table  $H[v]$  records the index of the first occurrence of node  $v$  in array  $E$ .

As a result, the LCA of a node pair  $(u, v)$  is the node situated on the smallest level between the first occurrence of  $u$  the and first occurrence of  $v$ . We have the following lemma:

**Lemma 1:** Denoting the index of the node with the smallest level between the index  $a$  and  $b$  in the level table  $L$  as  $MinL(a, b)$ , the LCA of a given node pair  $(u, v)$  is  $E[MinL(H[u], H[v])]$ .

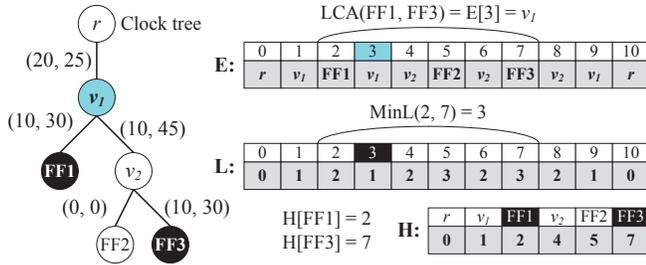


Figure 4. Derived tabular fields from the clock tree in Figure 3.

Take the LCA of FF1 and FF3 in Figure 4 for example. The occurrence indices of FF1 and FF3 in Euler tour are 2 and 7, respectively. Referring to the indices between 2 and 7 in the level table, the node with the lowest level is situated in the third position of the Euler table. Hence, the LCA of FF1 and FF3 is  $v_1$ . It is obvious the operations taken on occurrence table and Euler table can be done in constant time. Finding the position of an element with the minimum value between two specified indices in level table (i.e., the value returned by function  $MinL(a, b)$  for a given index pair  $a$  and  $b$ ) is the major task. We adopt the sparse-table solution whereby a two-dimensional (2D) table  $M[i][j]$  is used to store the index of the

minimum value in the level table starting at  $i$  having length  $2^j$  [5]. This concept is visualized in Figure 5.

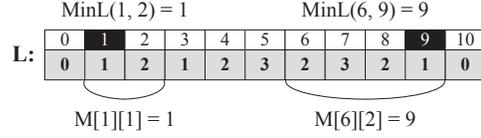


Figure 5. Range minimum query to the level table from Figure 4.

Figure 5 indicates that the optimal substructure of  $M[i][j]$  is the minimum value between the first and second halves of the interval with  $2^{j-1}$  length each. Hence, the table  $M$  can be fulfilled using dynamic programming with the following recurrence:

$$M[i][j] = \begin{cases} i, & \text{base case } j = 0 \\ M[i][j-1], & \text{if } L[M[i][j-1]] \leq L[M[i+2^{j-1}][j-1]] \\ M[i+2^{j-1}][j-1], & \text{otherwise} \end{cases}$$

Provided the table  $M$  has been processed, the value of  $MinL(a, b)$  can be computed by selecting two blocks that entirely cover the interval between  $a$  and  $b$  and returning the minimum between them. Let  $c$  be  $\lfloor \log(b-a+1) \rfloor$  and assume  $b > a$ , the following formula is used for computing the value of  $MinL(a, b)$ :

$$MinL(a, b) = \begin{cases} M[a][c], & \text{if } L[M[a][c]] \leq L[M[b-2^c+1][c]] \\ M[b-2^c+1][c], & \text{otherwise} \end{cases}$$

The procedure of building tables  $E, L, H$ , and  $M$  is presented in Algorithm 2. Tables  $E, L$ , and  $H$  can be built using depth-first search starting at the root of the clock tree (line 1), while table  $M$  is fulfilled via bottom-up dynamic programming (line 2:16). Using these tables as infrastructure, the credit of two given nodes in the clock tree can be retrieved in constant time by Algorithm 3. The LCA of the two given nodes is found first (line 1:12). Then for the hold test, the credit is returned as the difference between the latest arrival time and the earliest arrival time at the LCA (line 14:15). For the setup test which performs timing check in the subsequent clock cycle, the credit excludes the arrival time at the clock source (line 16:18). We conclude the lookup table preprocessing by theorem 1.

---

**Algorithm 2:** BuildCreditLookupTable( $G$ )

---

**Input:** circuit network  $G$

- 1 Build tables  $E, L, H$  via Euler tour starting at the root  $r$  of clock tree;
  - 2  $size_1 \leftarrow L.size$ ;
  - 3  $size_2 \leftarrow \lfloor \log(L.size) \rfloor$ ;
  - 4 Create a 2D table  $M$  with size  $size_1 \times (size_2 + 1)$ ;
  - 5 **for**  $i \leftarrow 0$  **to**  $size_1 - 1$  **do**
  - 6 |  $M[i][0] \leftarrow i$ ;
  - 7 **end**
  - 8 **for**  $j \leftarrow 1$  **to**  $size_2 - 1$  **do**
  - 9 | **for**  $i \leftarrow 0$  **to**  $size_1 - 2^j$  **do**
  - 10 | | **if**  $L[M[i][j-1]] < L[M[i+2^{j-1}][j-1]]$  **then**
  - 11 | | |  $M[i][j] \leftarrow M[i][j-1]$ ;
  - 12 | | **else**
  - 13 | | |  $M[i][j] \leftarrow M[i+2^{j-1}][j-1]$ ;
  - 14 | | **end**
  - 15 | **end**
  - 16 **end**
- 

**Theorem 1:** UI-Timer builds lookup tables  $E, L, H$ , and  $M$  in  $O(n \log n)$  space and  $O(n \log n + m)$  time. Using these lookup tables,

---

**Algorithm 3:** GetCredit( $u, v$ )
 

---

**Input:** nodes  $u$  and  $v$

```

1 if  $u$  or  $v$  is not a node of the clock tree then
2   return 0;
3 end
4 if  $H[u] > H[v]$  then
5   swap( $u, v$ )
6 end
7  $c \leftarrow \lfloor \log(H[u] - H[v] + 1) \rfloor$ ;
8 if  $L[M[H[u]][c]] < L[M[H[v] - 2^c + 1][c]]$  then
9    $lca \leftarrow E[M[H[u]][c]]$ ;
10 else
11    $lca \leftarrow E[M[H[v] - 2^c + 1][c]]$ ;
12 end
13 if hold test then
14   return  $at_{lca}^{late} - at_{lca}^{early}$ ;
15 else
16    $r \leftarrow$  root of the clock tree;
17   return  $at_{lca}^{late} - at_{lca}^{early} - (at_r^{late} - at_r^{early})$ ;
18 end

```

---

the credit of two given nodes in the clock tree can be retrieved in  $O(1)$  time.

### B. Formulation of Pessimism-Free Graph

In the course of hold or setup check, the required arrival time of the destination FF and the amount of pessimism between each source FF and the destination FF remain fixed regardless of which data path is being considered. Precisely speaking, the way data paths passing through plays the most vital role in determining the final slack values. In order to facilitate the path search without interleaving between slack computation and pessimism retrieval, we construct a pessimism-free graph  $G_p = \{V_p, E_p\}$  for a given test  $t$  as follows:

**Rule #1:** We designate the data pin  $d$  of the destination FF and artificially create a source node  $s$  and connect it to the clock pin  $i$  of each source FF. Denoting the set of artificial edges as  $E_s$ , we have  $V_p = V \cup \{s\}$  and  $E_p = E \cup E_s$ .

**Rule #2:** We associate 1) *offset weight* with each artificial edge and 2) *delay weight* with each ordinary circuit connection as follows:

- $\forall e_{s \rightarrow i} \in E_s, w_{e_{s \rightarrow i}}^{hold} = credit_{i,d}^{hold} - rat_t^{early} + at_i^{early}$ .
- $\forall e_{s \rightarrow i} \in E_s, w_{e_{s \rightarrow i}}^{setup} = credit_{i,d}^{setup} + rat_t^{late} - at_i^{late}$ .
- $\forall e \in E, w_e^{hold} = delay_e^{early}$ .
- $\forall e \in E, w_e^{setup} = -delay_e^{late}$ .

An example of pessimism-free graph is shown in Figure 6. The intuition is to separate out the constant portion of the post-CPPR slack by an artificial edge such that the search procedure can focus on the rest portion which is totally depending on the way data paths passing through. It is clear that the cost of any source-destination path (i.e., sum of all edge weights) in the pessimism-free graph is equivalent to post-CPPR slack of the corresponding data path which is obtained by removing the artificial edge. This crucial fact is highlighted in the following theorem:

**Theorem 2:** The cost of each source-destination path in the pessimism-free graph  $G_p$  is equal to the post-CPPR slack of the corresponding data path.

On the basis of theorem 2, the problem of identifying the top- $k$  critical paths for a given test is similar to the path ranking

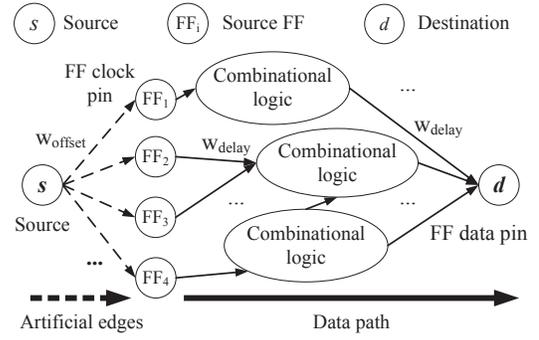


Figure 6. Derivation of pessimism-free graph from a given test.

problem applied to the pessimism-free graph. A number of state-of-the-art algorithms for path ranking have been proposed over the past years [3], [8], [11], [13]. The best time complexity acquired to date is  $O(m + n \log n + k)$  from the well-know Eppstein's algorithm [8]. However, it relies on sophisticated implementation of heap tree which results in little practical interests. Moreover, most existing approaches are developed for general graphs and lack a compact and efficient specialization to certain graphs such as the directed-acyclic circuit network. We shall discuss in the following sections the key contribution of UI-Timer in resolving these deficiencies.

### C. Implicit Representation of Data Path

Although explicit path representation is the major pursuit of existing approaches, the inherent restriction makes it difficult to devise efficient algorithms with satisfactory space and time complexities [9], [12]. UI-Timer performs implicit path representation instead, yielding significant improvements on memory usage and runtime performance. While the spirit is similar to [8], our algorithm differs in exploring a more compact and efficient way to implicit path search and explicit path recovery. We introduce the following definitions:

**Definition 1 – Suffix Tree:** Given a pessimism-free graph, the suffix tree refers to the successor order obtained from the shortest path tree  $T_d$  rooted at the destination node.

**Definition 2 – Prefix Tree:** The prefix tree is a tree order of non-suffix-tree edges such that each node *implicitly* represents a path with prefix from its parent path deviated on the corresponding edge and suffix followed from the suffix tree. The root which is artificially associated with a null edge refers to the shortest path in  $T_d$ . Table I lists the data field to which we apply for each node.

TABLE I DATA FIELD OF A PREFIX TREE NODE	
Member	Definition
$p$	pointer to the parent node
$e$	deviation edge
$w$	cumulative deviation cost
$c$	credit for pessimism removal
Constructor	PrefixNode( $p, e, w, c$ )

An example is illustrated in Figure 7. The suffix tree is depicted with bold edges and numbers on nodes denote the shortest distance to the destination node. Dashed edges denote artificial connections from the source node. The shortest path is  $\langle e_3, e_8, e_{12}, e_{15} \rangle$  which is implicitly represented by the root of prefix tree. The prefix tree node

marked by “ $e_{11}$ ” implicitly represents the path with prefix  $\langle e_3, e_8 \rangle$  from its parent path deviated on “ $e_{11}$ ” and suffix  $\langle e_{14} \rangle$  following from the suffix tree. As a result, explicit path recovery can be realized in a recursive manner as presented in Algorithm 4.

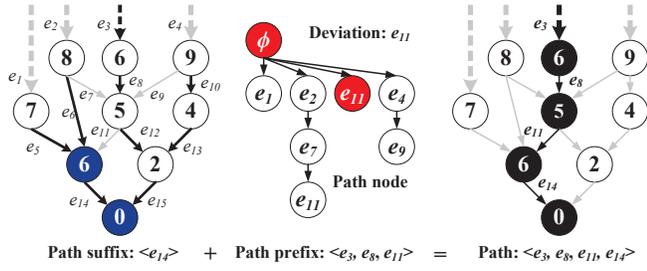


Figure 7. Implicit path representation using suffix tree and prefix tree.

---

**Algorithm 4:** RecoverDataPath( $pxf, end$ )

---

**Input:** prefix-tree node pointer  $pxf$ , node  $end$

```

1 beg ← head[pxf.e];
2 if pxf.p ≠ NIL then
3   RecoverDataPath(pxf.p, tail[pxf.e]);
4 end
5 while beg ≠ end do
6   Record the path trace through pin “beg”;
7   beg ← successor[beg]
8 end
9 Record the path trace through pin “end”;

```

---



---

**Algorithm 5:** Slack( $pxf, s, r$ )

---

**Input:** prefix-tree node pointer  $pxf$ , source node  $s$ , CPPR flag  $r$   
**Output:** post-CPPR slack for true flag  $r$  or pre-CPPR slack otherwise

```

1 if r = true then
2   return pxf.w + dis[s];
3 end
4 return pxf.w + dis[s] - pxf.c;

```

---

In order to retrieve the path cost, we keep track of the deviation cost of each edge  $e$ , which is defined as follows [8]:

$$dvi[e] = dis[head[e]] - dis[tail[e]] + weight[e] \quad (5)$$

Notice that  $dis[v]$  denotes the shortest distance from node  $v$  to the destination node. Intuitively, deviation cost is a non-negative quantity that measures the distance loss by being deviated from  $e$  instead of taking the ordinary shortest path to destination. Therefore for each node in the prefix tree, the corresponding path cost (i.e., post-CPPR slack) is equal to the summation of its cumulative deviation cost and the cost of shortest path in  $T_d$ . Algorithm 5 realizes this process. We conclude the conceptual construction so far by the following two important lemmas.

**Lemma 2:** *UI-Timer deals with the implicit representation of each data path in  $O(1)$  space and time complexities.*

**Lemma 3:** *The cumulative deviation cost of each node in the prefix tree is greater than or equal to that of its parent node.*

Above lemmas are two obvious byproducts of our prefix tree definition. Lemma 1 tells that UI-Timer stores each data path in constant space and records or queries important information such as credit and slack in constant time. While lemma 2 is true due to

the monotonicity, we shall demonstrate in the next section its strength and simplicity in pruning the search space.

**D. Generation of Top-k Critical Paths**

We begin by presenting a key subroutine of our path generating procedure – *Spur*, which is described in Algorithm 6. In a rough view, *Spur* describes the way UI-Timer expands its search space for discovering critical paths. After a path  $p_i$  is selected as the  $i$ -th critical path, each node along the path  $p_i$  is viewed as a deviation node to spur a new set of path candidates (line 2:14). Any duplicate path should be ruled out from the candidate set (line 1 and line 5:7) and each newly spurred path is parented to the path  $p_i$  in the prefix tree (line 8). Having a path candidate with non-negative post-CPPR slack, the following search space can be pruned and is exempted from the queuing operation (line 9:11). This simple yet effective prune strategy is a natural result of lemma 2 due to the monotonic growth of path cost along with our search expansion.

---

**Algorithm 6:** Spur( $pxf, s, d, Q$ )

---

**Input:** prefix-tree node pointer  $pxf$ , source node  $s$ , destination node  $d$ , priority queue  $Q$

```

1 u ← head[pxf.e];
2 while u ≠ d do
3   for e ∈ fanout(u) do
4     v ← head[e];
5     if v = successor[u] or v is unreachable then
6       continue;
7     end
8     pxf_new ← new PrefixNode(pxf, e, pxf.w + dvi[e], pxf.c);
9     if Slack(pxf_new, s, true) < 0 then
10      Q.enqueue(pxf_new);
11    end
12  end
13 u ← successor[u];
14 end

```

---

**Lemma 4:** *The procedure Spur is compact, meaning every path candidate is generated uniquely.*

*Proof:* Suppose there is at least a pair of duplicate path candidates  $p_1$  and  $p_2$ , which are implicitly represented by  $\xi_1$  and  $\xi_2$  the sets of deviation edges. Since  $p_1$  and  $p_2$  are identical,  $\xi_1$  and  $\xi_2$  must be identical as well. If both  $\xi_1$  and  $\xi_2$  contain only one edge, the respective prefix tree nodes must be parented to the same node, which is invalid due to the filtering statement in line 5:7. If both  $\xi_1$  and  $\xi_2$  contain multiple edges, there exists at least two distinct permutations in the prefix tree that represent the same path. However, this will result in a cyclic connection of edges which violates the graph property of the circuit network. Therefore by contradiction the procedure *Spur* is compact. ■

**Lemma 5:** *The procedure Spur takes  $O(n + m \log k)$  time complexity.*

*Proof:* The entire procedure takes up to  $n$  phases on scanning a given path and spurs at most  $m$  new path candidates. We maintain only the top- $k$  critical candidates ever seen such that the maximum number of items in the priority queue at any time will not exceed  $k$ . This can be achieved in  $O(m \log k)$  time using a min-max priority queue [4]. Therefore the total complexity is  $O(n + m \log k)$ . ■

Using Algorithms 4–6 as primitive, the top- $k$  critical paths can be identified using Algorithm 7. Prior to the search, we construct the suffix tree by finding the shortest path tree rooted at the destination node  $d$  in the pessimism-free graph (line 1). Then each of the most critical paths from source FFs to the destination FF is viewed as an

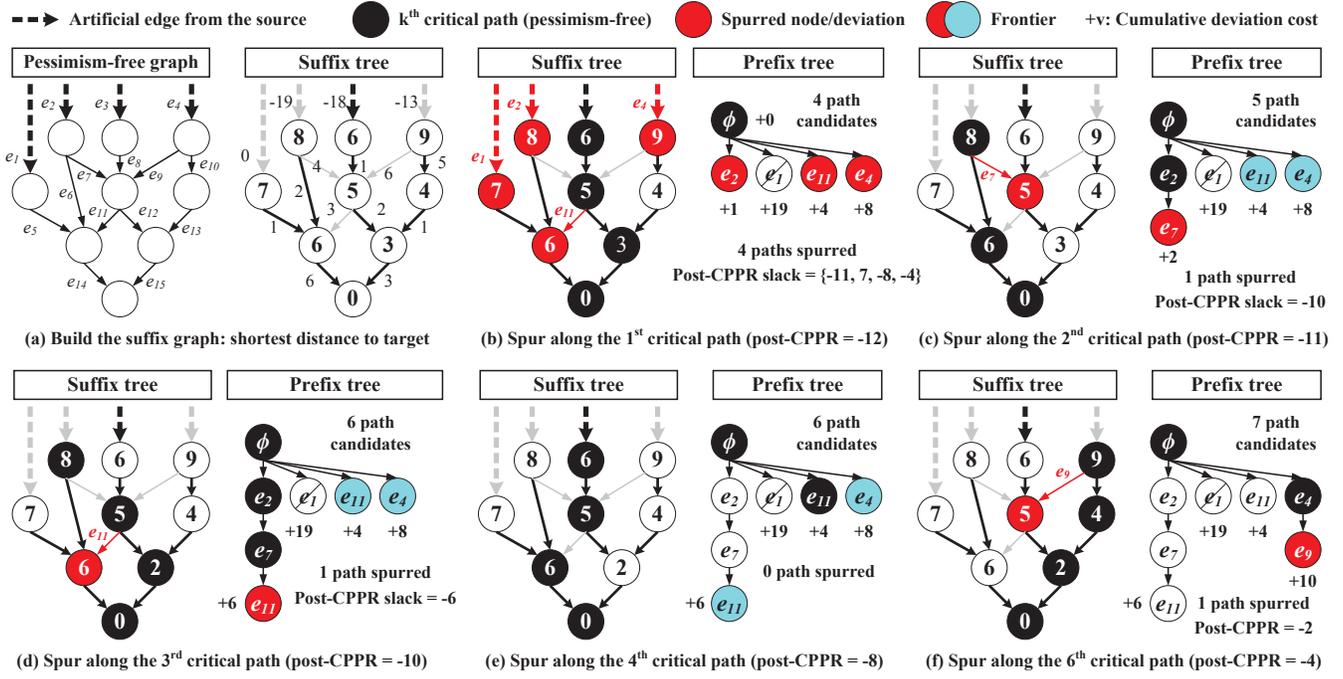


Figure 8. Exemplification of UI-Timer. (a) UI-Timer builds a suffix tree in the initial iteration by finding the shortest path tree rooted at the target node. (b) During the first search iteration, four paths are spurred from the most critical path  $\langle e_3, e_8, e_{12}, e_{15} \rangle$ . (c) During the second search iteration, one path is spurred from the second critical path  $\langle e_2, e_6, e_{14} \rangle$ . (d) During the third search iteration, one path is spurred from the third critical path  $\langle e_2, e_7, e_{12}, e_{15} \rangle$ . (e) No path is generated from the fourth and fifth search iterations. (f) During the sixth search iteration, one path is spurred from the sixth critical path  $\langle e_4, e_{10}, e_{13}, e_{15} \rangle$ .

#### Algorithm 7: GetCriticalPath( $s, d, k$ )

**Input:** source node  $s$ , destination node  $d$ , path count  $k$   
**Output:** solution set  $\Psi$  of critical paths

- 1 Build the suffix tree by finding the shortest path tree rooted at  $d$ ;
- 2 Initialize a priority queue  $Q$  keyed on cumulative deviation cost;
- 3  $\Psi \leftarrow \phi$ ;
- 4  $num\_path \leftarrow 0$ ;
- 5 **for**  $e \in fanout(s)$  **do**
- 6      $credit \leftarrow GetCredit(head[e], d)$ ;
- 7      $px \leftarrow new PrefixNode(NIL, e, dev[e], credit)$ ;
- 8     **if**  $Slack(px, s, true)$  **is failing then**
- 9          $Q.enqueue(px)$ ;
- 10    **end**
- 11 **end**
- 12 **while**  $Q$  is not empty **do**
- 13      $px\_new \leftarrow Q.dequeue()$ ;
- 14      $num\_path \leftarrow num\_path + 1$ ;
- 15      $\Psi \leftarrow \Psi \cup RecoverDataPath(px, d)$ ;
- 16     **if**  $num\_path \geq k$  **then**
- 17         **break**;
- 18     **end**
- 19      $Spur(px, s, d, Q)$ ;
- 20 **end**
- 21 **return**  $\Psi$ ;

initial path candidate (line 5:11). The major search loop (line 12:20) iteratively looks for a path with lowest cumulative deviation cost from the path candidate set and performs spurring operation on it. Iteration ends when we have extracted  $k$  paths (line 16:18) or no more steps can be proceeded. Finally, we draw the following two theorems.

**Theorem 3:** *UI-Timer is complete, meaning that it can exactly identify the top- $k$  critical paths for each hold test or setup test without clock network pessimism.*

*Proof:* Proving the completeness of UI-Timer is equivalent to showing that the major search framework of UI-Timer is exactly identical to a typical graph search problem [13]. The search space or search tree of UI-Timer grows equivalently with the prefix tree, in which each state represents a path implicitly. *Spur* is responsible for neighboring expansion, iteratively including a set of new deviation edges as tree leaves or search frontiers. Since by definition all paths can be viewed as being deviated from the shortest path, the initial state is equivalent to the root of the prefix tree. Using a priority queue, the items or paths extracted are in the order of criticality. ■

**Theorem 4:** *UI-Timer solves each hold test or setup test in space complexity  $O(n \log n + m + k)$  and time complexity  $O(n \log n + kn + km \log k)$ .*

*Proof:* The space complexity of UI-Timer involves  $O(n + m)$  for storing the circuit graph,  $O(n \log n)$  for lookup table, and  $O(n)$  for the suffix tree as well as  $O(k)$  for the prefix tree. As a result, the total space requirement is  $O(n \log n + n + k)$ . On the other hand, it takes up to  $k$  iterations on calling the procedure *Spur* in order to discover the top- $k$  critical paths. Recalling that the lookup table is built in time  $O(n \log n)$  and the suffix tree can be constructed in time  $O(n + m)$  using topological relaxation, the time complexity of UI-Timer is thus  $O(n \log n + kn + km \log k)$ . ■

An exemplification is given in Figure 8. (a) illustrates a suffix tree derived by computing the shortest path tree rooted at the destination node from a given pessimism-free graph. (b) shows a total of four paths are spurred from the current-most critical path  $p_1 = \langle e_3, e_8, e_{12}, e_{15} \rangle$  in the first search iteration. For instance, the path with deviation edge  $e_{11}$  has cumulative cost equal to  $0 + (6 - 5 + 3) = 4$ . The corresponding explicit path recovery is  $\langle e_3, e_8, e_{11}, e_{14} \rangle$  as a result of combining the prefix of  $p_1$  ending

at the tail of  $e_{11}$  and the suffix from the suffix tree beginning at the head of  $e_{11}$ . On the other hand, the path with deviation edge  $e_1$  has deviation cost equal to  $0 + (7 - (-12) + 0) = 19$  which in turns tells the value of its post-CPPR slack being  $-12 + 19 = 7$ . Since the post-CPPR slack has been positive already, by lemma 3 the following search space can be pruned (node marked with a slash “/”). Accordingly in the end of this iteration, only three of the four spurred paths are explored as search frontiers from the parent path  $p_1$ . (c)–(f) repeat the same procedure except no more paths are spurred from the fourth and fifth search iterations.

### E. Parallel Implementation for Multiple Tests

The generic framework of UI-Timer is developed on the basis of one test at one time. In other words, each test is treated as an independent input without dependency on the others. For applications where multiple tests are designated, a readily available parallel extension can be carried out by evoking multiple threads with each operating on one test. With the shared lookup table and the circuit graph, we impose the least memory requirement by maintaining only private information about the suffix tree and the prefix tree for each thread. A number of tests with up to the maximum number of threads supported by the machine can be simultaneously processed. The multi-threaded implementation is presented in Algorithm 8.

---

#### Algorithm 8: UI-TimerParallel( $\hat{t}, k$ )

---

**Input:** test vector  $\hat{t}$ , path count  $k$   
**Output:** solution vector  $\hat{\Psi}$  of the top- $k$  critical paths for each test

- 1 BuildCreditLookupTable();
- 2 **Parallel for** index  $i$  in range( $\hat{t}$ ) **do**
- 3      $G_p^i \leftarrow$  pessimism-free graph for the test  $\hat{t}[i]$ ;
- 4      $\hat{\Psi}[i] \leftarrow$  GetCriticalPath( $G_p^i$ .source,  $G_p^i$ .destination,  $k$ );
- 5 **end**
- 6 **return**  $\hat{\Psi}$ ;

---

## VI. EXPERIMENTAL RESULT

UI-Timer is implemented in C++ language on a 2.67GHz 64-bit Linux machine with 8GB memory. The application programming interface (API) provided by OpenMP 3.1 is used for our multi-thread parallelization [2]. Our machine can execute a maximum of four threads concurrently. Experiments are undertaken on a set of circuit benchmarks released from TAU 2014 CAD contests [10]. Figure 9 illustrates the impact of CNPR on hold and setup test slacks for circuits *des\_perf* and *vga\_lcd*. As contest rules, we run for each circuit benchmark the timer setting the path count  $k$  from 1 to 20 on all setup and hold tests and collect averaged quantities such as runtime and accuracy for comparison. The accuracy is measured by the error rate of mismatched paths to a golden reference offered by an industrial timer [1], [10]. Table II lists the benchmark statistics and overall performance of UI-Timer comparing with top-ranked timers, “LightSpeed” and “iTimerC,” from TAU 2014 CAD contest [1]. For fair comparison, all timers are run with four threads.

We begin by comparing UI-Timer with LightSpeed. The strength of UI-Timer is clearly demonstrated in the accuracy value. Our timer achieves exact accuracy yet LightSpeed suffers from many path mismatches. The highest error rate is observed in the smallest design *s27*. Unfortunately, we are unable to report experimental data of *ac97\_ctrl*, *Combo5*, *Combo6*, and *Combo7*, because LightSpeed encounters execution faults. Although LightSpeed is faster in some cases, the performance margin of LightSpeed reaches up to  $\times 141$  worse than UI-Timer in circuit *tv80* (i.e., 32.38 vs 0.23) while the

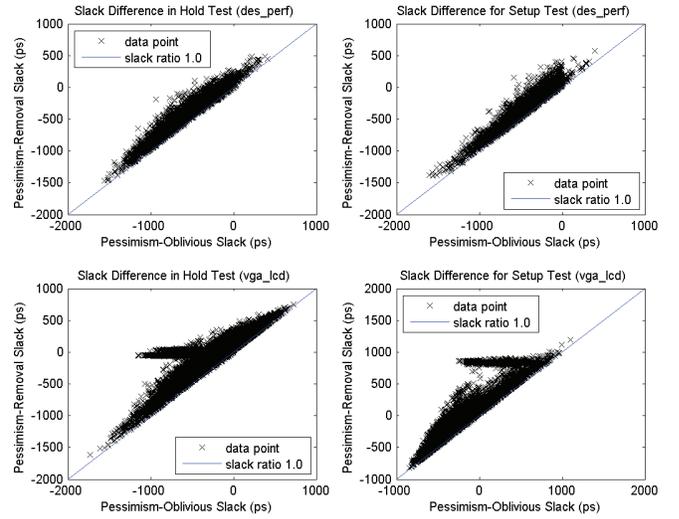


Figure 9. Impact of CNPR on test slacks for circuits *des\_perf* and *vga\_lcd*.

counterpart of UI-Timer is more comparable by at most  $\times 9$  slower in *vga\_lcd* (i.e., 16.78 vs 159.15). In this concern, the solution quality of UI-Timer is more stable and reliable, especially for high-frequency designs where accuracy is the major concern of timing-specific optimizations. Next we compare UI-Timer with iTimerC. In general, UI-Timer outperforms iTimerC across nearly all circuit benchmarks in terms of runtime. Although iTimerC acquires better accuracy than LightSpeed, its runtime performance is not remarkable especially in larger designs such as *Combo5*, *Combo6* and *Combo7*. The largest difference is observed in circuit *tv80* where UI-Timer reaches the goal by  $\times 101$  faster than iTimerC (i.e., 23.13 vs 0.23). Similar trend can be also found in other cases.

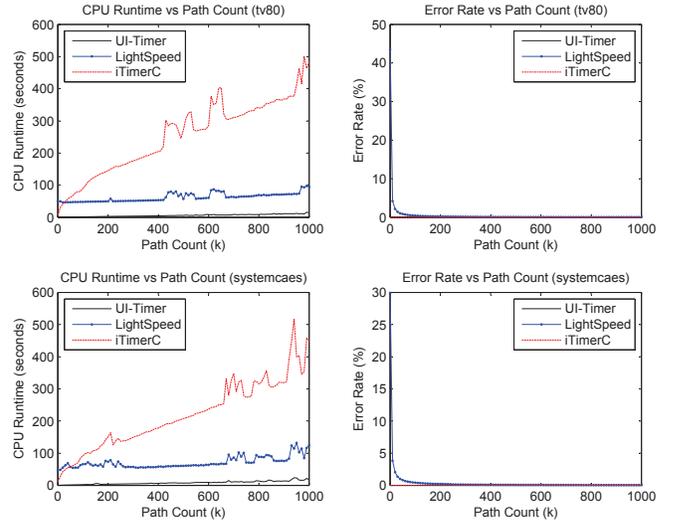


Figure 10. Performance characterization of UI-Timer, LightSpeed, and iTimerC for circuits *tv80* and *systemcaes*.

Finally we investigate the scalability of UI-Timer by varying the input parameter of the path count from 1 to 1000. The performance comparing UI-Timer with LightSpeed and iTimerC on two example circuits, *tv80* and *systemcaes*, is characterized in Figure 10. We see all runs are accomplished instantaneously by UI-Timer and the

TABLE II  
PERFORMANCE COMPARISON BETWEEN UI-TIMER AND TOP-RANKED TIMERS LIGHTSPEED AND iTIMER C FROM TAU 2014 CAD CONTEST [1].

Circuit	V	E	C	I	O	# Tests	# Paths	LightSpeed			iTimerC		UI-Timer	
								AER	MER	CPU	AER	CPU	AER	CPU
s27	109	112	6	6	1	6	9	9.97	50.00	0.20	0	0.40	0	0.20
s344	574	658	16	11	11	30	71	0	0	0.22	0	0.53	0	0.22
s349	598	682	16	11	11	30	71	0	0	0.25	0	0.53	0	0.22
s386	570	701	7	9	7	12	27	0	0	0.20	0	0.49	0	0.20
s400	708	813	22	5	6	42	77	0	0	0.23	0	0.56	0	0.21
s510	891	1091	7	21	7	12	99	0	0	0.18	0	0.40	0	0.18
s526	933	1097	22	5	6	42	44	0	0	0.25	0	0.56	0	0.22
s1196	1928	2400	19	16	14	36	478	0	0	0.25	0	0.59	0	0.22
s1494	2334	2961	7	10	19	12	105	0	0	0.25	0	0.58	0	0.21
systemcdes	10826	13327	1967	132	65	380	41436	6.79	32.89	2.27	0	3.62	0	0.14
wb_dma	14647	17428	5218	217	215	1374	158	7.46	39.30	0.23	0	0.90	0	0.28
tv80	18080	23710	3608	14	32	838	19227963	8.20	43.49	32.38	0	23.13	0	0.23
systemcaes	23909	29673	6643	260	129	2500	13069928	6.53	29.92	33.23	0	22.44	0	0.62
mem_ctrl	36493	45090	10638	115	152	3754	62938	5.41	24.73	0.65	0	3.71	0	0.83
ac97_ctrl	49276	55712	22223	84	48	9370	148	-	-	-	0	2.95	0	1.31
usb_func	53745	66183	17665	128	121	4392	129854	6.43	37.87	0.94	0	5.64	0	1.41
pci_bridge32	70051	78282	33474	162	207	16450	17296	5.04	25.49	2.27	0	14.49	0	4.71
aes_core	68327	86758	5289	260	129	2528	21064	6.72	31.70	0.68	0	4.46	0	0.96
des_perf	330538	404257	88751	235	64	19764	1682	4.60	11.89	3.37	0	18.37	0	19.24
vga_lcd	449651	525615	172065	89	109	50182	5281	7.94	43.21	16.78	0	119.24	0	159.15
Combo2	260636	284091	171529	170	218	29574	62938	4.70	24.07	9.19	0	49.00	0	56.12
Combo3	181831	284091	73784	353	215	8294	129854	6.71	35.14	3.39	0	20.30	0	11.35
Combo4	778638	866099	469516	260	169	53520	19227963	7.93	42.13	205.69	0	557.81	0	333.04
Combo5	2051804	2228611	1456195	432	164	79050	19227963	-	-	-	N/A	> 3 hrs	0	1225.50
Combo6	3577926	3843033	2659426	486	174	128266	19227963	-	-	-	N/A	> 3 hrs	0	3544.04
Combo7	2817561	3011233	2136913	459	148	109568	19227963	-	-	-	N/A	> 3 hrs	0	2485.81

|V|: size of node set. |E|: size of edge set. |C|: size of clock tree. |I|: # of primary inputs. |O|: # of primary outputs. # Tests: # of setup tests and hold tests. # Paths: max # of data paths per test. AER/MER: avg/max error rate of mismatched paths (%). CPU: avg program runtime (seconds). -: unexpected program fault.

runtime gap to the other timers becomes clear as path count grows. With regard to accuracy, our timer is always exact and confers a fundamental difference to LightSpeed which sacrifices accuracy for unpronounced speedup. To sum up in precise, these results have justified the practical viability of UI-Timer.

## VII. CONCLUSION

In this paper we have presented UI-Timer, an exact and ultra-fast algorithm for handling the CNPR problem during static timing analysis. Unlike existing approaches which frequently use exhaustive path search with case-by-case heuristics, our timer maps the CNPR problem to a graph-theoretic formulation and applies an efficient search routine using a highly compact and efficient data structure to obtain an exact solution. UI-Timer has several merits such as simplicity, coding ease, and most importantly the theoretically-proven completeness and optimality. These advantages confer UI-Timer a high degree of differential over existing methods. Comparatively, experimental results have demonstrated the superior performance of UI-Timer in terms of accuracy and runtime over top-ranked timers from TAU 2014 CAD contest.

## ACKNOWLEDGEMENT

This work was partially supported by the National Science Foundation under Grant CCF-1320585. The authors acknowledge Y.-M. Yang, Y.-W. Chang, and I. H.-R. Jiang from team iTimerC and M. S. S. Kumar and N. Sireesh from team LightSpeed for sharing their binaries in TAU 2014 CAD contest.

## REFERENCES

[1] TAU 2014 Contest: Pessimism Removal of Timing Analysis, <http://sites.google.com/site/tacontest2014>

[2] OpenMP: Parallel Programming API, <http://openmp.org/wp/>

[3] H. Aljazzar and S. Leue, "K\*: A Heuristic Search Algorithm for Finding the K Shortest Paths," *Artificial Intelligence*, vol. 175, no. 18, pp. 2129–2154, 2011.

[4] M. Atkinson, J. Sack, N. Santoro, and T. Strothotte, "Min-max heaps and generalized priority queues," *Commun. ACM*, vol. 29, no. 10, pp. 996–1000, 1986.

[5] M. A. Bender and M. F. Colton, "The LCA problem revisited," *Proc. 4th Latin American Symposium on Theoretical Informatics*, vol. 1776, pp. 88–94. Springer, 2000.

[6] Sarvesh Bhardwaj, Khalid Rahmat, and Kayhan Kucukcakar, "Clock-Reconvergence Pessimism Removal in Hierarchical Static Timing Analysis," US patent 8434040, 2013.

[7] J. Bhasker and R. Chadha, "Static Timing Analysis for Nanometer Designs: A Practical Approach," *Springer*, 2009.

[8] D. Eppstein, "Finding the k shortest paths," *Proc. IEEE FOCS*, pp. 154–165, 1994.

[9] D. Hathaway, J. P. Alvarez, and K. P. Belkale, "Network Timing Analysis Method which Eliminates Timing Variations between Signals Traversing a Common Circuit Path," US patent 5636372, 1997.

[10] J. Hu, D. Sinha, and I. Keller, "TAU 2014 Contest on Removing Common Path Pessimism during Timing Analysis," *Proc. ACM ISPD*, pp. 153–160, 2014.

[11] E. Q. V. Martins and M. M. B. Pascoal, "A New Implementation of Yen's Ranking Loopless Paths Algorithm," *A quarterly Journal of Operation Research*, vol. 1, no. 2, 2003.

[12] A. K. Ravi, "Common Clock Path Pessimism Analysis for Circuit Designs using Clock Tree Networks," US patent 7926019, 2011.

[13] J. Y. Yen, "Finding the k Shortest Loopless Paths in a Network," *Manage. Sci.*, vol. 17 no. 11, pp. 712–716, 1971.

[14] J. Zejda and P. Frain, "General Framework for Removal of Clock Network Pessimism," *Proc. IEEE/ACM ICCAD*, pp. 632–639, 2002.