UI-Timer 1.0: An Ultra-Fast Path-Based Timing Analysis Algorithm for CPPR

Tsung-Wei Huang and Martin D. F. Wong

***Abstract*—The recent TAU computer-aided design (CAD) contest has aimed to seek novel ideas for accurate and fast common path pessimism removal (CPPR). 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 1.0, a powerful CPPR algorithm which achieves high accuracy and ultra-fast runtime. Unlike existing approaches which are dominated by explicit path search, UI-Timer 1.0 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 1.0 achieved the best result in terms of accuracy and runtime over existing CPPR algorithms.**

***Index Terms*—Static timing analysis (STA), common path pessimism removal (CPPR)**

I. Introduction

THE lack of accurate and fast algorithms for common path pessimism removal (CPPR) has been recently pointed out as a major weakness of existing static-timing analysis (STA) tools [3]. Conventional STA tools rely on conservative dual-mode operations to estimate early-late and late-early path slacks [4]. This mechanism, however, imposes unnecessary pessimism due to the consideration of delay variation along common segments of clock paths, as illustrated in Figure 1. This is because signal cannot simultaneously experience early-mode and late-mode operations along the physically common segment of the data path and clock path in the clock network. Unnecessary pessimism may lead to timing tests (e.g., setup check, hold check, etc.) being marked as failing whereas in reality 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.

The importance and impact of CPPR are demonstrated in Figure 2. It is observed that the number of failing tests

Preliminary version of this paper is presented at the 2014 IEEE/ACM International Conference on Computer-Aided Design (ICCAD'14), San Jose, CA, November 2014 [1], [2].

Tsung-Wei Huang and Martin D. F. Wong are with the Department of Electrical and Computer Engineering, University of Illinois at Urbana-Champaign (UIUC), IL, USA (email: twh760812@gmail.com; mdfwong@illinois.edu);

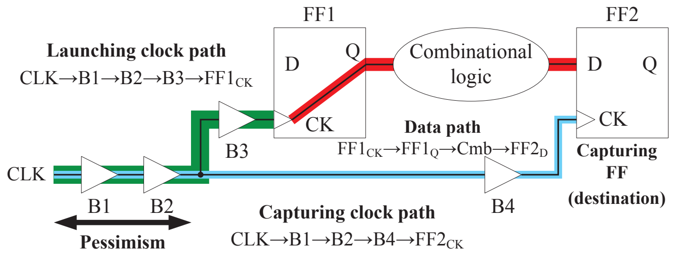


Figure 1. Common path pessimism incurs in the common path between the launching clock path and the capturing clock path.

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 [5].

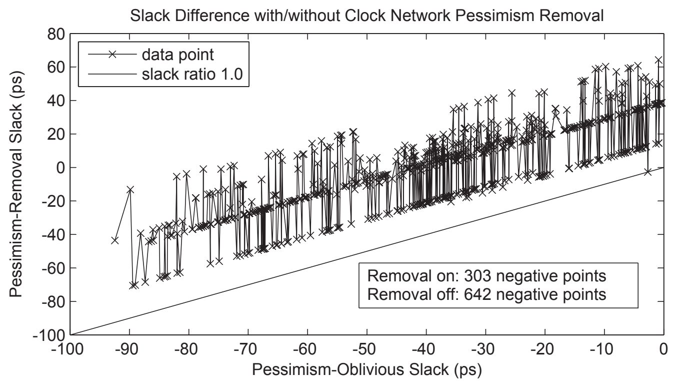


Figure 2. Impact on common path pessimism from a circuit in [6].

State-of-the-art CPPR algorithms are dominated by straightforward path-based methodology [7], [8], [9]. 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 line plot in Figure 2), revealing a potential drawback that path-based methodology has the worst-case performance of exhaustive search space in peeling out the true critical paths. Accordingly, prior works are usually too slow to handle complex designs and unable to always identify the true critical path accurately [6].

In this paper we introduce UI-Timer 1.0, a powerful CPPR algorithm which achieves high accuracy, ultra-fast runtime, and low memory requirement. Our contributions are summarized as follows: 1) We introduce a theoretical framework that maps the CPPR 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 [6]. Comparatively, UI-Timer 1.0 confers promissing results over existing timers in terms of accuracy and runtime. The source code of our timer has been released to the public domain [10], which can be an indicator assisting researchers in discovering and optimizing the performance bottleneck of their tools.

The rest of the paper is organized as follows: In Sections II–III we discuss the preliminary and background of STA and CPPR. Prior works are briefed in Section IV. In Section V, we formally formulate the problem of CPPR and define terminologies. In Section VI, we present the algorithm of UI-Timer 1.0, followed by practical applications and technical details in Sections VII–VIII. The experimental results are demonstrated in Section IX. Finally, we draw the conclusion and future works in Section X.

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 provides both lowerbound and upperbound quantities to accounts for various on-chip variations (OVC) such as process parameter, e.g., transistor width, voltage drops, and temperature fluctuations [4]. In contrast to statistical STA (SSTA) where process variations are modeled as random variables, the early-late timing model has deterministic behaviors and thus enables lower computational complexity for timing propagation. 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 data point at a flip-flop (FF). The hold test and setup test are two safe timing guard that constrain the earliest required arrival time and the latest required arrival time for a data point, respectively. Considering a timing test t, the following equations are applied for STA [6].

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

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

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.

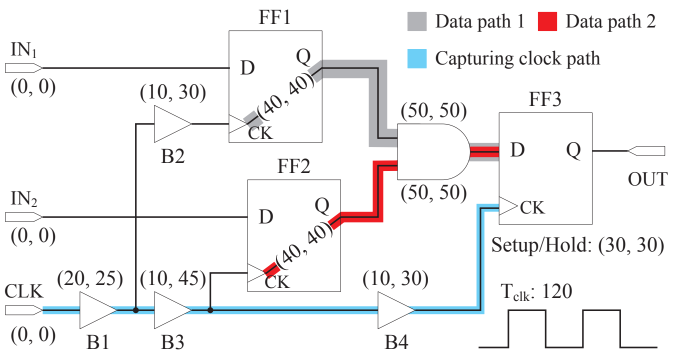


Figure 3. An example of sequential circuit network.

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. Common-Path-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 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 [6]:

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

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

$$slack_{post-CPPR}^{setup} = slack_{pre-CPPR}^{setup} + credit_{u,v}^{setup} \tag{5}$$

$$slack_{post-CPPR}^{hold} = slack_{pre-CPPR}^{hold} + credit_{u,v}^{hold} \tag{6}$$

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 [6]. The slack prior to common path pessimism removal (CPPR) is referred to as *pre-CPPR slack* and *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. Prior Works

Removing pessimism from the design during timing analysis is integral to meeting chip timing, area, and power targets. To this end, existing STA tools continue to invest heavily in research and development on this topic and explore new ideas and concepts to improve CPPR runtime and memory usage [11]. Predominant approach relies on identifying a set of critical paths without CPPR first. Then the CPPR credit of each of these paths are discovered through the traversal on the clock network, after which the true slack can be retrieved [8], [9]. Based on this framework, straightforward heuristics such as dominator grouping for clock reconverging nodes [5], hierarchical timing analysis [7], branch-and-bound pruning [12], [13], and CPPR credit caching [14] are proposed to either shrink the solution space or reduce the computational complexity. However, these works suffer from a common drawback of exhaustive search space. In spite of fine-tuned heuristics, the resulting performance is always case-by-case and has no guaranteed characteristics of polynomial space and time complexity.

V. 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 $e_{u\to v}$ 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 $\langle v_1, v_2, \cdots, v_n\rangle$ or edges $\langle e_1, e_2, \cdots, e_n\rangle$ 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 CPPR 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.*

VI. Algorithm

The overall algorithm of UI-Timer 1.0 is presented in Algorithm 1. It consists of 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_1.0(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 $\Psi$;
```

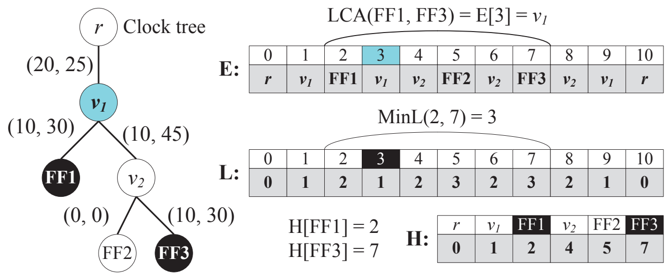
A. Lookup Table Preprocessing

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 [15] 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($H[u]$, $H[v]$)].*



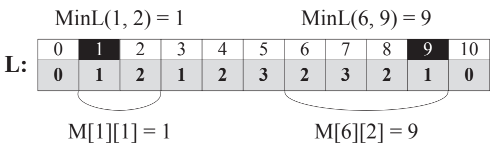
E:	0	1	2	3	4	5	6	7	8	9	10
	r	v1	FF1	v1	v2	FF2	v2	FF3	v2	v1	r

L:	0	1	2	3	4	5	6	7	8	9	10
	0	1	2	1	2	3	2	3	2	1	0

H:	r	v1	FF1	v2	FF2	FF3
	0	1	2	4	5	7

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 [15]. This concept is visualized in Figure 5.



L:	0	1	2	3	4	5	6	7	8	9	10
	0	1	2	1	2	3	2	3	2	1	0

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:

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

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):

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

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 1.0 builds lookup tables E, L, H, and M in O(nlogn) space and O(nlogn + m) time. Using these lookup tables, 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

Algorithm 3: GetCredit(u, v)

```
Input: nodes $u$ and $v$
if $u$ or $v$ is not a node of the clock tree then
    return 0;
end
if $H[u] > H[v]$ then
    swap(u, v)
end
$c \leftarrow \lfloor log(H[u] - H[v] + 1) \rfloor$ ;
if $L[M[H[u]][c]] < L[M[H[v] - 2^c + 1][c]]$ then
    $lca \leftarrow E[M[H[u]][c]]$;
else
    $lca \leftarrow E[M[H[v] - 2^c + 1][c]]$;
end
if hold test then
    return $at_{lca}^{late} - at_{lca}^{early}$;
else
    $r \leftarrow$ root of the clock tree;
    return $at_{lca}^{late} - at_{lca}^{early} - (at_r^{late} - at_r^{early})$;
end
```

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 the destination node 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 $V_p = V \bigcup \{s\}$ and $E_p = E \bigcup E_s$.

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

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

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.*

Proof: The cost of a source-destination path can be written as the delay of the corresponding data path p from the source FF i to the destination FF d plus the offset weight associated with the edge $e_{s \to i}$. The path cost for hold test is $credit_{i,d}^{hold} - rat_t^{early} + at_i^{early} + \sum_{e \in p} delay_e^{early}$ and $credit_{i,d}^{setup} + rat_t^{late} - at_i^{late} - \sum_{e \in p} delay_p^{late}$ for setup test. It is clear that by definition the cost is just the post-CPPR slack of a given path in either hold test or setup test. ■

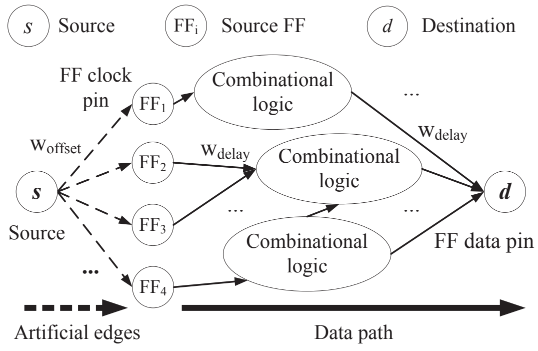


Figure 6. Derivation of pessimism-free graph from a given test.

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 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 [16], [17], [18], [19], [20]. The best time complexity acquired to date is $O(m + nlogn + k)$ from the well-know Eppstein's algorithm [17]. 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 1.0 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 [8], [9]. UI-Timer 1.0 performs implicit path representation instead, yielding significant improvements on memory usage and runtime performance. While the spirit is similar to [17], our algorithm differs in exploring a more compact and efficient way to implicit path search and explicit path recovery. We introduce the following definitions:

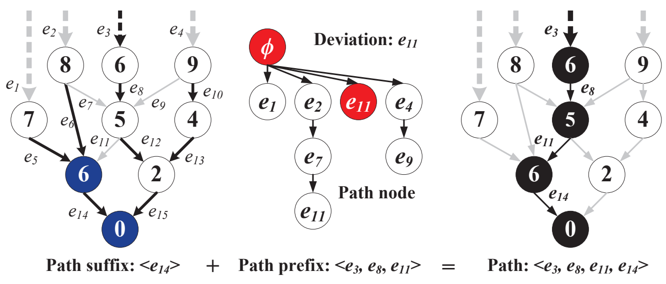
Definition 1 – Suffix Tree: Given a pessimism-free graph, the suffix tree refers to the successor order obtained from the shortest path tree T_d rooted at the destination node.

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.



Deviation: e_{11}
Path node
Path suffix: $<e_{14}>$ + Path prefix: $<e_3, e_8, e_{11}>$ = Path: $<e_3, e_8, e_{11}, e_{14}>$

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

```
Algorithm 4: RecoverDataPath(pfx, end)
  Input: prefix-tree node pointer pfx, node end
1 beg ← head[pfx.e];
2 if pfx.p ≠ NIL then
3     RecoverDataPath(pfx.p, tail[pfx.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(pfx, s, r)
  Input: prefix-tree node pointer pfx, 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 pfx.w + dis[s];
3 end
4 return pfx.w + dis[s] - pfx.c;
```

In order to retrieve the path cost, we keep track of the deviation cost of each edge e, which is defined as follows [17]:

$$dvi[e] = dis[head[e]] - dis[tail[e]] + weight[e] \quad (7)$$

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 1.0 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 2 tells that UI-Timer 1.0 stores each data path in constant space and records or queries important information such as credit and slack in constant time. While lemma 3 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 1.0 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 3 due to the monotonic growth of path cost along with our search expansion.

```
Algorithm 6: Spur(pfx, s, d, Q)
  Input: prefix-tree node pointer pfx, source node s, destination node d,
         priority queue Q
1  u ← head[pfx.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          pfx_new ← new PrefixNode(pfx, e, pfx.w + dvi[e], pfx.c);
9          if Slack(pfx_new, s, true) < 0 then
10             Q.enque(pfx_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

ξ_1 and ξ_2 the sets of deviation edges. Since p_1 and p_2 are identical, ξ_1 and ξ_2 must be identical as well. If both ξ_1 and ξ_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 ξ_1 and ξ_2 contain multiple edges, there exists at least two distinct permutations in the prefix tree that represent the same path. However, this will results 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 + mlogk) 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(mlogk)* time using a min-max priority queue [21]. Therefore the total complexity is *O(n + mlogk)*. ■

Algorithm 7: GetCriticalPath(*s*, *d*, *k*)

```
Input: source node s, destination node d, path count k
Output: solution set $\Psi$ of the top-k critical paths
Build the suffix tree by finding the shortest path tree rooted at d;
Initialize a priority queue Q keyed on cumulative deviation cost;
$\Psi \leftarrow \phi$ ;
num_path $\leftarrow$ 0;
for $e \in$ fanout(s) do
    credit $\leftarrow$ GetCredit(head[e], d);
    pfx $\leftarrow$ new PrefixNode(NIL, e, dvi[e], credit);
    if Slack(pfx, s, true) $< 0$ then
        Q.enque(pfx);
    end
end
while Q is not empty do
    pfx_new $\leftarrow$ Q.deque();
    num_path $\leftarrow$ num_path + 1;
    $\Psi \leftarrow \Psi \bigcup$ RecoverDataPath(pfx, d);
    if num_path $\geq k$ then
        break;
    end
    Spur(pfx, s, d, Q);
end
return $\Psi$;
```

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 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 1.0 is complete, meaning that it can exactly identify the top-k critical paths for each hold test or setup test without common path pessimism.*

Proof: Proving the completeness of UI-Timer 1.0 is equivalent to showing that the major search framework of UI-Timer 1.0 is exactly identical to a typical graph search problem [20]. The search space or search tree of UI-Timer 1.0 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. ■

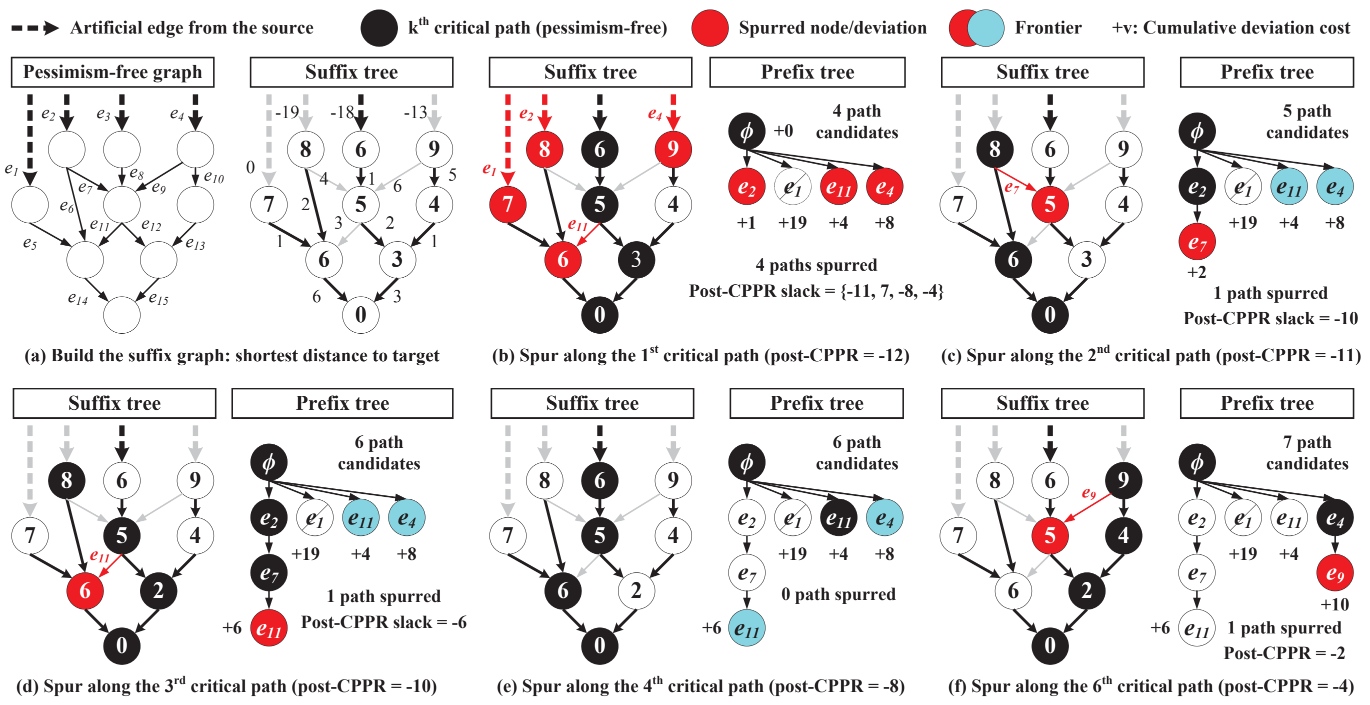
Theorem 4: *UI-Timer 1.0 solves each hold test or setup test in space complexity O(nlogn + m + k) and time complexity O(nlogn + kn + kmlogk).*

Proof: The space complexity of UI-Timer 1.0 involves *O(n + m)* for storing the circuit graph, *O(nlogn)* 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(nlogn + 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(nlogn)* and the suffix tree can be constructed in time *O(n + m)* using topological relaxation, the time complexity of UI-Timer 1.0 is thus *O(nlogn + kn + kmlogk)*. ■

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.

VII. Application to Multiple Tests

The architecture of UI-Timer 1.0 is developed on the basis of one test at one time. That is, each test is regarded as an independent input and has no dependence on each other. For applications where multiple tests are designated, a readily available parallel framework can be carried out by forking multiple threads with each operating on a subset of tests. 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. One multi-threaded application is presented in



Artificial edge from the source
kth critical path (pessimism-free)
Spurred node/deviation
Frontier
+v: Cumulative deviation cost
Pessimism-free graph
Suffix tree
Prefix tree
(a) Build the suffix graph: shortest distance to target
(b) Spur along the 1st critical path (post-CPPR = -12)
4 path candidates
4 paths spurred
Post-CPPR slack = {-11, 7, -8, -4}
(c) Spur along the 2nd critical path (post-CPPR = -11)
5 path candidates
1 path spurred
Post-CPPR slack = -10
(d) Spur along the 3rd critical path (post-CPPR = -10)
6 path candidates
1 path spurred
Post-CPPR slack = -6
(e) Spur along the 4th critical path (post-CPPR = -8)
6 path candidates
0 path spurred
(f) Spur along the 6th critical path (post-CPPR = -4)
7 path candidates
1 path spurred
Post-CPPR = -2

Figure 8. Exemplification of UI-Timer 1.0. (a) UI-Timer 1.0 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 forth 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 8, in which we sweep the test test and report the top-k critical paths for each test.

```
Algorithm 8: SweepReport($\hat{t}$, $k$)
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 $\hat{\Psi}$;
```

```
Algorithm 9: GetCriticalTest($\hat{t}$, $k$)
Input: test vector $\hat{t}$, test count $k$
Output: the set $\hat{\Omega}$ of the top-$k$ critical tests
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     $p \leftarrow$ GetCriticalPath($G_p^i$.source, $G_p^i$.destination, 1);
5     $t$.criticality $\leftarrow$ $p$.slack;
6 end
7 sort $\hat{t}$ according to criticality;
8 $\hat{\Omega} \leftarrow$ top-$k$ tests in $\hat{t}$;
9 return $\hat{\Omega}$;
```

As opposed to the sweep report in Algorithm 8, block report is another common application where probing the top-k critical paths across all timing tests is the main goal. We refer the criticality of a test to the slack value of the top most critical path extracted from this test. It is intuitive by set property that the top-k critical paths must exist in the path set generated from the top-k critical tests. Therefore, we first develop Algorithm 9 to peel the top-k critical tests out of a given test set. Algorithm 9 sweeps the test set and finds the most critical path for each test (line 1:4). The post-CPPR slack value of each path is used as the criticality of the corresponding test (line 5). A sorting procedure is then followed so as to peel out the top-k critical tests (line 7:9).

Using Algorithm 9, the function of block report for the globally top-k critical paths is constructed in Algorithm 10. We first apply Algorithm 9 to peel out the top-k critical tests (line 1). Since it has been shown that the globally top-k critical paths must be investigated from these tests, we iteratively extract the top-k critical paths from each of the top-k critical tests (line 3:10). An efficient min-max priority queue [21] is employed to dynamically maintain the solution paths (line 2) and prune unnecessary search (line 4:6).

Theorem 5: *The function SweepReport in Algorithm 8 takes O(nlogn + $|\hat{t}|$(kn + kmlogk) / C) time complexity, where $\hat{t}$ is the input test vector and C is the number of available cores or threads.*

Proof: Algorithm 8 exerts the core procedure of UI-Timer 1.0 on a given test vector $\hat{t}$. A sequential version hence takes *O(nlogn + $|\hat{t}|$(kn + kmlogk))* time complexity. Notice that the lookup tables for CPPR credit only needs one-time building, which takes *O(nlogn)* time complexity. Running Algorithm 8 in a machine with C cores or C threads supports a parallel reduction by up to a factor of C. Therefore, the runtime

Algorithm 10: BlockReport($\widehat{t}$, k)

```
Input: test vector $\widehat{t}$, path count $k$
Output: the set $\widehat{\Psi}$ of the globally top-$k$ critical paths across $\widehat{t}$
1  $\widehat{\Omega} \leftarrow$ GetCriticalTest($\widehat{t}$, $k$);
2  $Q \leftarrow$ priority queue keyed on slack values;
3  for $t \in \widehat{\Omega}$ do
4      if $Q.size = k$ and $t.criticality \geq Q.top\_max$ then
5          break;
6      end
7      $G_p^t \leftarrow$ pessimism-free graph for the test $t$;
8      $Q \leftarrow Q \cup$ GetCriticalPath($G_p^t.source$, $G_p^t.destination$, $k$);
9      $Q.maintain\_top\_k\_min(k)$;
10 end
11 $\widehat{\Psi} \leftarrow$ paths from the priority queue $Q$;
12 return $\widehat{\Psi}$;
```

complexity of sweep report is *O(nlogn +* $|\widehat{t}|$*(kn + kmlogk) / C).* ■

Theorem 6: *The function GetCriticalTest in Algorithm 9 takes O(nlogn + (n + m) / C +* $|\widehat{t}|log|\widehat{t}|$ *+ k) time complexity, where* $\widehat{t}$ *is the input test vector and C is the number of available cores or threads.*

Proof: The first section (before sorting) of Algorithm 9 is nearly the same as Algorithm 8, except that only the single most critical paths is generated. Therefore, the time complexity is *O(nlogn +* $|\widehat{t}|$*(n + m) / C).* Afterwards, sorting the test vector $\widehat{t}$ takes $O(|\widehat{t}|log|\widehat{t}|)$ time complexity and outputting the top-k critical tests takes linear time complexity $O(k)$. Hence, the entire runtime complexity of Algorithm 9 is *O(nlogn + (n + m) / C +* $|\widehat{t}|log|\widehat{t}|$ *+ k).* ■

Theorem 7: *The function BlockReport in Algorithm 10 takes O(nlogn + (n + m) / C +* $|\widehat{t}|log|\widehat{t}|$ *+* k^2n *+* k^2mlogk*) time complexity, where* $\widehat{t}$ *is the input test vector and C is the number of available cores or threads.*

Proof: Algorithm 10 first calls Algorithm 9 to obtain the top-k critical tests from a given test vector $\widehat{t}$, which takes *O(nlogn + (n + m) / C +* $|\widehat{t}|log|\widehat{t}|$ *+ k)* time complexity. Generating the globally top-k critical paths involves k iterations calling Algorithm 7. Besides, each iteration requires k logarithmic operations in order to maintain the top-k critical paths in the priority queue. The time complexity of each iteration is thus *O(kn + kmlogm + klogk).* As a result, the total time complexity of block report is *O(nlogn + (n + m) / C +* $|\widehat{t}|log|\widehat{t}|$ *+* k^2n *+* k^2mlogk*).* ■

VIII. Implementation and Technical Details

In this section, we highlight two implementation techniques that are practical for the improvement of runtime performance, despite not reducing the theoretical bound. It is observed from the program profiler that the majority of the runtime is spent on the construction of suffix tree, which is equivalent to finding the shortest path tree in the pessimism-free graph. The shortest path routines such as storage initialization, distance relaxation, and fanin/fanout scanning typically exhibit wild and deep swing in the search space and consume a huge amount of CPU instructions. The problem becomes even critical when multiple tests are taken into account. To remedy this problem, two verified trials are worth delivering.

A. Memory Pool for Efficient Storage Initialization

Constructing the suffix tree is equivalent to discovering the shortest path tree rooted at the target node of the pessimism-free graph. A generic framework of any shortest path algorithms requires two data arrays, *distance* and *successor*, for storing the distance labels and shortest path tree connection, respectively [22]. Before the relaxation on distance labels takes effect, programmer should clear the two arrays by assigning an infinite value to every distance entry and a nil value to every successor entry. Nonetheless, real applications come with multiple tests. This linear procedure will be repeated for each test and the accumulative runtime becomes non-negligible. Furthermore, in most cases each test involves only a small portion of the entire circuit graph in labeling process. It is desirable to clear those entries ever participating in the previous search. To this end, we preallocate a memory pool for *distance* and *successor* arrays and clear their memory values in the very beginning. We also keep track of those entries whose values were ever modified in the course of shortest path routines and clear these entries by the end of function return. As a consequence, the computational effort on storage initialization can be minimized.

B. Redundant Search Space Pruning

Reducing the size of suffix tree is another effective way to decrease the runtime, and it can be beneficial for the later search on prefix paths. Since we consider only violating points, any suffix paths discovered so far with positive value can be discarded so as to prune the subsequent search space. In the course of shortest path search, the worst timing quantities at a given pin (which can be precomputed) provide a lower bound and a upper bound on the minimum hold and maximum setup path slack that are reachable from this pin. An A*-like pruning strategy can thus be employed, as presented in Algorithm 11. Notice that without loss of generality one can replace the cutoff value with any user-specified slack threshold and this has no impact on the overall correctness subject to a proper implementation of shortest path algorithms.

Algorithm 11: is_prunable(m, p, dis)

```
Input: test type $m$, a pin $p$, a distance array $dis$
Output: true if $p$ is prunable from the suffix tree or false otherwise
1  if $m =$ HOLD then
2      if $dis[p] + at_p^{early} \geq cutoff$ then
3          return true;
4      end
5  end
6  if $dis[p] - at_p^{late} \geq cutoff$ then
7      return true;
8  end
9  return false;
```

Lemma 6: *The pruning strategy in Algorithm 11 is correct, meaning that the derived suffix tree contains no path suffix of which having slack value larger than the given cutoff value.*

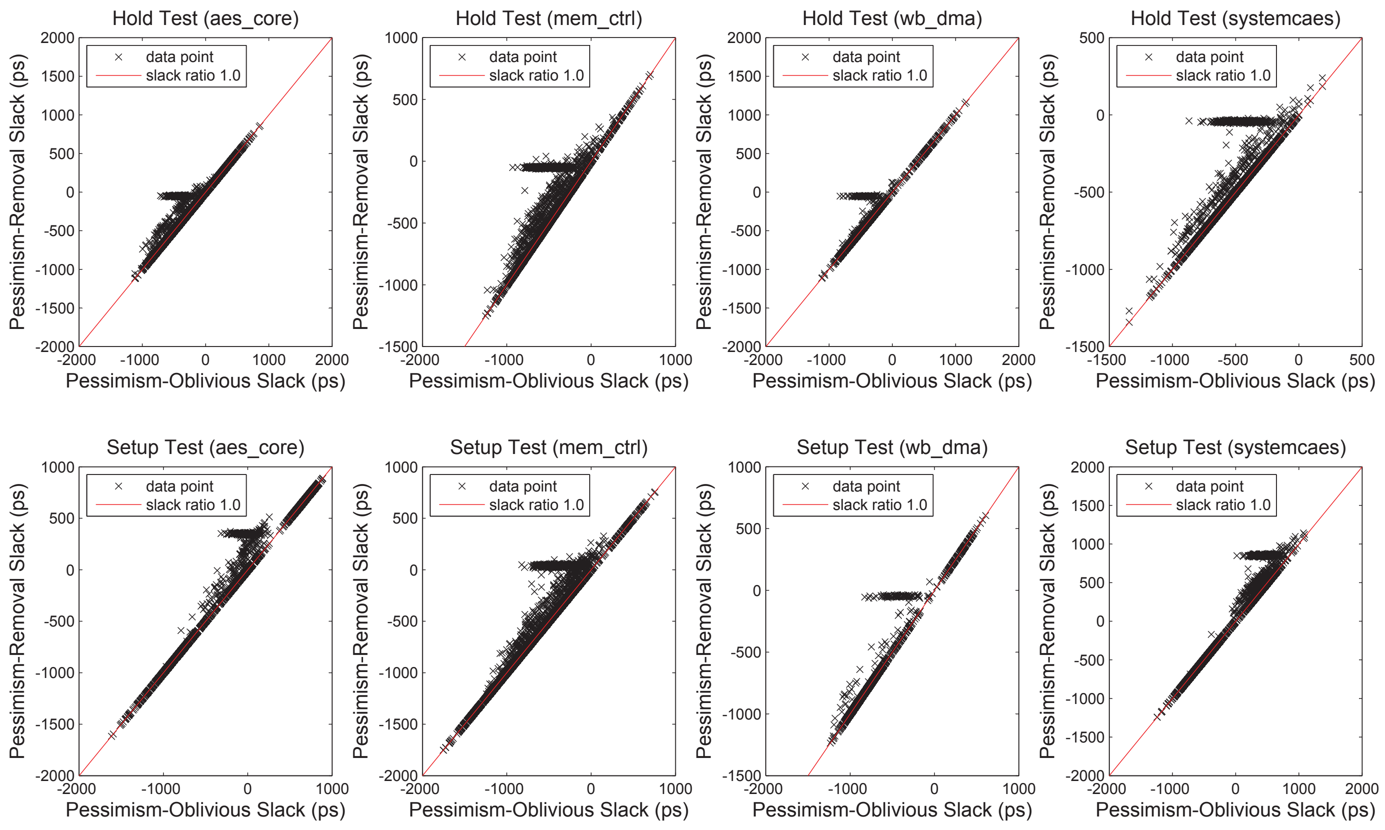


Figure 9. Impact of CPPR on hold and setup time slacks for circuits aes_core, mem_ctrl, wb_dma, and systemcaes. Data points are sampled based on the worst pre-CPPR slack value of each test.

We have proved that the cost of any source-destination path in the pessimism-free graph is identical to the slack value of the corresponding data path. In hold time test, the distance value of a pin p, denoted as $dis[p]$, represents the potential slack value discovered so far from the destination. The earliest arrival time at this pin, denoted as at_p^{early}, is the minimum delay that will be added for any complete data paths suffixed at the pin p. That is, the slack values of such paths are lower-bounded by $dis[p] + at_p^{early}$ and any search points exceeding the cutoff values can be pruned. The proof for the setup time test can be drawn in a similar way.

IX. Experimental Results

UI-Timer 1.0 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 [23]. 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 [3]. The benchmarks are modified from well-known industrial circuits (e.g., s27, s510, systemcdes, wb_dma, pci_bridge32, vga_lcd, etc.) that have been released to the public domain for research purpose. Statistics of these circuits are summarized in Table II. All benchmarks are associated with multiple tests. The three largest circuits, Combo5, Combo6, and Combo7, have million-scale graph data. For example, the circuit Combo6 has 3577926 pins and 3843033 edges.

A. *Effectiveness of CPPR*

Figure 9 depicts the impact of CPPR on hold and setup test slacks for circuits des_perf and vga_lcd. The horizontal and vertical axes in the plots denote the pre-CPPR slack and the post-CPPR slacks, respectively. Each plot is attached a reference line with slope 1.0 indicating the identical slacks. It is observed that each post-CPPR slack is at least the pre-CPPR slack value and most post-CPPR slack values are improved. The plots indicate the effectiveness of CPPR during design closure from designers' perspective. The synthesis and optimization tools can focus their efforts on true timing-critical paths and optimize these paths only by the amount necessary to meet the target clock frequency of the chip.

B. *Comparison with TAU 2014 CAD Contest Entries*

We first compare UI-Timer 1.0 with the final entries in TAU 2015 CAD contest. Adhering to contest rules, we ran the timer for each circuit benchmark with different path counts k from 1 to 20 across all setup and hold tests and collected averaged quantities on runtime and accuracy for comparison. The accuracy is measured by the percentage of mismatched paths to a golden reference generated by an industrial timer [6], [3]. Table II lists the overall performance of UI-Timer 1.0 in comparison to the top-3 timers, "Timer-1st", "Timer-2nd", and "Timer-3rd", for short, from TAU 2014 CAD contest [6]. For fair comparison, all timers are run in the same environment with four threads.

TABLE II
COMPARISON BETWEEN UI-TIMER 1.0 AND THE TOP-3 WINNERS, TIMER-1ST, TIMER-2ND, AND TIMER-3RD FROM TAU 2014 CAD CONTEST [6].

Circuit	$\|V\|$	$\|E\|$	$\|C\|$	# Tests	# Paths	Timer-2nd			Timer-3rd		Timer-1st		UI-Timer 1.0	
						AER	MER	CPU	AER	CPU	AER	CPU	AER	CPU
s27	109	112	6	6	9	9.97	50.00	0.20	0	0.40	0	0.20	0	0.01
s344	574	658	16	11	11	0	0	0.22	0	0.53	0	0.22	0	0.02
s349	598	682	16	11	11	0	0	0.25	0	0.53	0	0.22	0	0.02
s386	570	701	7	9	7	0	0	0.20	0	0.49	0	0.20	0	0.02
s400	708	813	22	5	6	0	0	0.23	0	0.56	0	0.21	0	0.02
s510	891	1091	7	21	7	0	0	0.18	0	0.40	0	0.18	0	0.01
s526	933	1097	22	5	6	0	0	0.25	0	0.56	0	0.22	0	0.02
s1196	1928	2400	19	16	14	0	0	0.25	0	0.59	0	0.22	0	0.01
s1494	2334	2961	7	10	19	0	0	0.25	0	0.58	0	0.21	0	0.02
systemcdes	10826	13327	1967	380	41436	6.79	32.89	2.27	0	3.62	0	0.14	0	0.09
wb_dma	14647	17428	5218	1374	158	7.46	39.30	0.23	0	0.90	0	0.28	0	0.19
tv80	18080	23710	3608	838	19227963	8.20	43.49	32.38	0	23.13	0	0.23	0	0.23
systemcaes	23909	29673	6643	2500	13069928	6.53	29.92	33.23	0	22.44	0	0.62	0	0.37
mem_ctrl	36493	45090	10638	3754	62938	5.41	24.73	0.65	0	3.71	0	0.83	0	0.52
ac97_ctrl	49276	55712	22223	9370	148	-	-	-	0	2.95	0	1.31	0	0.69
usb_funct	53745	66183	17665	4392	129854	6.43	37.87	0.94	0	5.64	0	1.41	0	0.78
pci_bridge32	70051	78282	33474	16450	17296	5.04	25.49	2.27	0	14.49	0	4.71	0	2.91
aes_core	68327	86758	5289	2528	21064	6.72	31.70	0.68	0	4.46	0	0.96	0	0.62
des_perf	330538	404257	88751	19764	1682	4.60	11.89	3.37	0	18.37	0	19.24	0	6.25
vga_lcd	449651	525615	172065	50182	5281	7.94	43.21	16.78	0	119.24	0	159.15	0	30.19
Combo2	260636	284091	171529	29574	62938	4.70	24.07	9.19	0	49.00	0	56.12	0	13.67
Combo3	181831	284091	73784	8294	129854	6.71	35.14	3.39	0	20.30	0	11.35	0	4.53
Combo4	778638	866099	469516	53520	19227963	7.93	42.13	205.69	0	557.81	0	333.04	0	78.10
Combo5	2051804	2228611	1456195	79050	19227963	-	-	-	N/A	> 3 hrs	0	1225.50	0	226.47
Combo6	3577926	3843033	2659426	128266	19227963	-	-	-	N/A	> 3 hrs	0	3544.04	0	544.36
Combo7	2817561	3011233	2136913	109568	19227963	-	-	-	N/A	> 3 hrs	0	2485.81	0	464.68

$|V|$: size of node set. $|E|$: size of edge set. $|C|$: size of clock tree. # 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.

We begin by comparing UI-Timer 1.0 with Timer-2nd. The strength of UI-Timer 1.0 is clearly demonstrated in the accuracy value. Our timer achieves exact accuracy yet Timer-2nd 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 Timer-2nd encounters execution faults. It is expected that Timer-2nd is faster in some cases as they sacrifice the accuracy for speed. However, the performance margin of Timer-2nd can be up to $\times$141.78 worse than UI-Timer 1.0 in circuit tv80 (i.e., 32.38 vs 0.23) while the counterpart of UI-Timer 1.0 is more competitive by at most $\times$1.85 slower in des_perf (i.e., 3.37 vs 6.25). As a result, the solution quality of UI-Timer 1.0 is more stable and reliable, especially for high-frequency designs where accuracy is the top priority of timing-specific optimizations.

Next we compare UI-Timer 1.0 with Timer-3rd and Timer-1st. In general, full accuracy scores are observed for all timers, while UI-Timer 1.0 reaches the goal far faster than the others. It can be seen that Timer-3rd suffers from significant runtime overhead across nearly all benchmarks and fails to accomplish the three largest designs, Combo5, Combo6, and Combo7, within 3 hours. Compared to Timer-1st, the first-place winner in TAU 2014 CAD Contest, our Timer achieves fairly remarkable speedup across all benchmarks. For example, our timer reaches the goal by $\times$22.0, $\times$5.3, and $\times$6.5 faster than Timer-1st in circuits s1196, vga_lcd, and Combo6, respectively. Similar trend can be found in other cases as well. The speedup curve becomes more pronounced for large circuits. In terms of memory profiling, we did not see too much difference between UI-Timer 1.0 and other entires. All computations are able to fit into the main memory with less than 1GB.

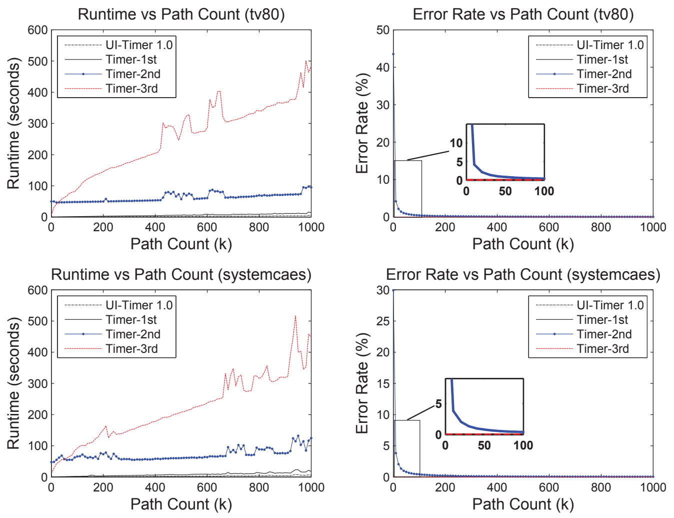


Figure 10. Performance characterization of UI-Timer 1.0, Timer-1st, Timer-2nd, and Timer-3rd for circuits tv80 and systemcaes.

We investigate the scalability of UI-Timer 1.0 by varying the input parameter, the path count k, from 1 to 1000. The performance comparing UI-Timer 1.0 with the top-3 entires, Timer-1st, Timer-2nd, and Timer-3rd on two example circuits, tv80 and systemcaes, is characterized in Figure 10. We see all runs are accomplished instantaneously by UI-Timer 1.0 and the runtime gap to the other timers becomes clear as path count grows. Take the point of 980 paths for example. UI-Timer 1.0 consumes only 3.41 seconds while the runtime values for Timer-1st, Timer-2nd, and Timer-3rd are 10.38 seconds, 93.25 seconds, and 500.26 seconds, respectively. With regard to accuracy, our timer is always exact and confers a fundamental difference to Timer-2nd which sacrifices accuracy for speedup.

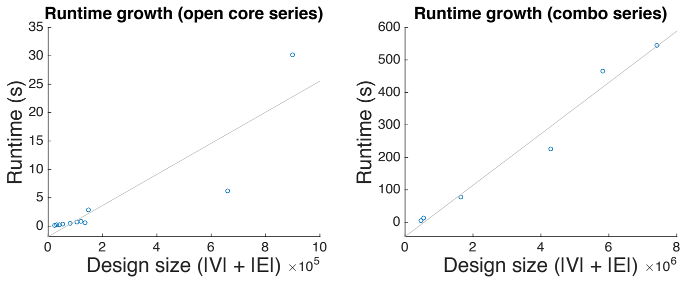


Figure 11. Scatter plot on runtime growth and design size for UI-Timer 1.0.

Finally we give a scatter plot showing the runtime growth of UI-Timer 1.0 versus the design size in Figure 11. The measurement is taken over the open core series (systemcdes, wb_dma, etc.) and the combo series (Combo2, Combo3, etc.). We approximate the design size using discrete quantity on the total number of nodes and edges in the circuit graph. It is convinced by the least square reference line that the runtime of UI-Timer 1.0 grows linearly with respect to the increase of design size. One can indirectly infer the amount of runtime needed for larger designs.

C. Comparison with the State-of-the-Art Timer

We have seen the superior performance of UI-Timer 1.0 in comparison to the top-ranked timers in TAU 2014 timing analysis contest. Ever since the contest was concluded, a few following works demonstrating promising results have been published in recent years [14], [12], [13]. We are particularly interested in the comparison with the timer, "iTimerC" [13], as it presented significant improvement to the contest winners. We observed both timers, iTimerC and UI-Timer 1.0, performed very well and achieved close results based on TAU 2014 contest environment. In order to discover the performance margin, we enhance the difficulty and the scale of this experiment on the six largest benchmarks, Combo2–Combo7. Each timer is requested to peel out the top-50 critical tests and report the top-2000 critical paths for each of the tests. In other words, evaluation is undertaken under an extreme condition in which reporting a high number of critical paths over a subset of critical tests is the goal.

The performance comparison between UI-Timer 1.0 and iTimerC [13] is presented in table III. It can be seen that UI-Timer 1.0 achieves highly scalable and reliable performance when the design size and query difficulty scale up. The higher runtime in setup test is expected because most critical paths come from the violation of setup constraint. Our runtime is superior in almost all testcases. We have observed significant runtime speedup to iTimerC by more than an order of magnitude for million-scale graphs, Combo5, Combo6, and

TABLE III
COMPARISON BETWEEN UI-TIMER 1.0 AND ITIMERC [13].

Circuit	Type	iTimerC [13]		UI-Timer 1.0	
		AER	CPU	AER	CPU
Combo2	hold	0	4.20	0	2.77
Combo2	setup	0	12.94	0	11.35
Combo3	hold	0	3.98	0	1.39
Combo3	setup	0	10.08	0	8.16
Combo4	hold	0	14.09	0	14.38
Combo4	setup	0	73.91	0	24.21
Combo5	hold	0	1334.24	0	47.20
Combo5	setup	unknown	> 1 hr	0	59.01
Combo6	hold	unknown	> 1 hr	0	130.60
Combo6	setup	unknown	> 1 hr	0	127.59
Combo7	hold	unknown	> 1 hr	0	88.91
Combo7	setup	unknown	> 1 hr	0	110.90

AER: avg error rate of mismatched paths (%). CPU: runtime (s).

Combo7. Considering the hold tests in Combo5, UI-Timer 1.0 requires only 47.20 seconds which is $\times 28.27$ faster than that by iTimerC. For the rest of million-scale graphs, our timer is able to analyze the timing by less than 3 minutes, whereas iTimerC cannot finish the program within 1 hour. These results have justified the practical viability of our timer.

D. Search Space Pruning through Slack Cutoff

Due to the high complexity of CPPR, modern industrial timers, in practice, apply various cutoff slack strategies to prune the search space. For example, the number of CPPR branching points can be controlled by some tolerance or threshold values so as to reduce the runtime and memory. As aforementioned, one important feature of UI-Timer 1.0 is the ease to control the slack margin, which has the potential to affect the number of paths generated during CPPR. By default, UI-Timer 1.0 reports negative slack and such cutoff value can be easily tuned since every path is 1) implicitly represented in constant time and space, and 2) generated in increasing order of post-CPPR slack values.

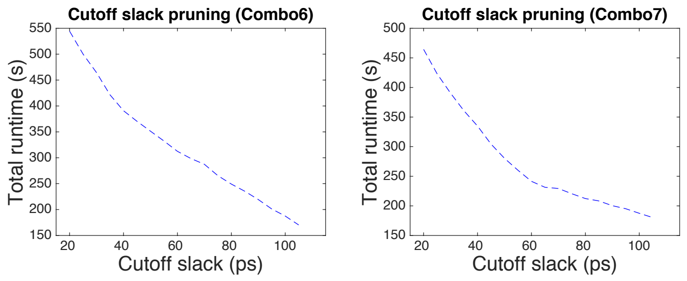


Figure 12. Runtime reduction curve under different slack cutoff values.

The runtime reduction under different cutoff slack values is plotted in Figure 12. We run experiments with five cutoff slack values, 20 ps, 40 ps, 60 ps, 80 ps, and 100 ps on the two largest benchmarks, Combo6 and Combo7. It is expected that the runtime decreases as the cutoff slack values increase. The higher the cutoff slack value is, the less the search space is spanned by path ranking. In spite of higher pessimism (less CPPR credit), the curve can be an useful indicator in striking a balance between program runtime and pessimism margin.

E. Extension to Distributed Computing

We have performed an extra evaluation on a distributed system running the three largest cases, Combo5, Combo6, and Combo7, in order to further demonstrate the scalability of our program. UI-Timer 1.0 is advantageous in handling every timing test independently. In distributed environment, multiple tests can be evenly partitioned into groups with respect to the number of cores. Each group is then assigned to one computing node and is analyzed by the timer independently. The application programming interface (API) provided by OpenMPI 1.6.5 is used as our message passing interface for distributed computing [24]. The evaluation is taken on a computer cluster having over 500 compute nodes with each configured with 16 Intel E5-2670 2.60GHz cores and 128GB RAM. The network infrastructure is 384-port Mellanox MSX6518-NR FDR InfiniBand for high speed cluster interconnect [25].

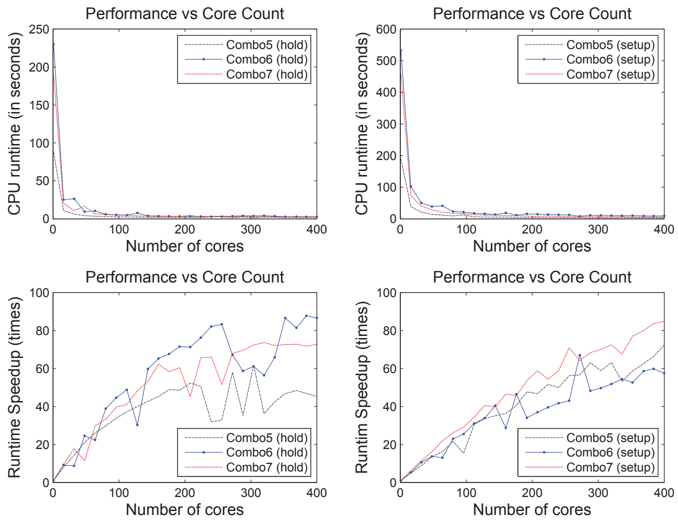


Figure 13. Runtime and speedup curves of hold tests and setup tests for benchmarks Combo5, Combo6, and Combo7 on a distributed system.

We begin by demonstrating the runtime performance versus the number of cores that is invoked for running our program. The core count is varied from 1 to 400 and the runtime is measured by a synchronized moment at which all process cores complete their jobs (i.e., reading the file, passing message, and handling all algorithmic procedures). The performance is interpreted in terms of the runtime and its relative speedup to a baseline which was run in single-core execution. Figure 13 shows the performance plot of this evaluation. It can be clearly seen that the runtime is reduced drastically as the number of cores increases. For example, the setup tests of Combo6 are accomplished by less than 1 minute with 16 cores, obtaining $\times 5.23$ speedup to the single-core execution (266.29 vs 50.95). Similar speedup curve is also present in other testcases. In a single minute, hold tests and setup tests of all testcases are solvable using only 16 cores.

X. Conclusion and Future Works

In this paper we have presented UI-Timer 1.0, an exact and ultra-fast algorithm for handling the CPPR problem during static timing analysis. Unlike existing approaches which frequently use exhaustive path search with case-by-case heuristics, our timer maps the CPPR 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. We have highlighted important features of UI-Timer 1.0 such as simplicity, coding ease, and most importantly the theoretically-proven completeness and optimality. Comparatively, experimental results have demonstrated the superior performance of UI-Timer 1.0 in terms of accuracy and runtime over existing timers.

Future works shall focus on fast incremental timing analysis with CPPR [26]. Various stages of the design flow such as logic synthesis, placement, routing, physical synthesis, and optimization facilitate a need for incremental timing analysis. The performance of incremental timing with CPPR plays a key role in the success of timing optimizations. Due to the path-specific property of CPPR, CPPR-aware incremental timing has emerged as one of the major challenges in existing timing analysis tools [10]. A high-quality CPPR-aware incremental timer is definitely advantageous to speed up the timing closure. Distributed timing analysis is also of our interests. As we move to many-core era, an effective distributed timing algorithm is important to speed up the timing closure [27], [28].

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 for sharing their binary iTimerC and M. S. S. Kumar and N. Sireesh for sharing their binary LightSpeed.

References

[1] T.-W. Huang, P.-C. Wu, and M. D. F. Wong, "Ui-timer: An ultra-fast clock network pessimism removal algorithm," in *Proc. IEEE/ACM ICCAD*, 2014, pp. 758–765.
[2] T.-W. Huang, P.-C. Wu, and M. D. F. Wong, "Fast path-based timing analysis for cppr," in *Proc. IEEE/ACM ICCAD*, 2014, pp. 596–599.
[3] J. Hu, D. Sinha, and I. Keller, "Tau 2014 contest on removing common path pessimism during timing analysis," in *Proc. ACM ISPD*, 2014, pp. 153–160.
[4] J. Bhasker and R. Chadha, *Static Timing Analysis for Nanometer Designs: A Practical Approach*. Springer, 2009.
[5] J. Zejda and P. Frain, "General framework for removal of clock network pessimism," in *Proc. IEEE/ACM ICCAD*, 2002, pp. 632–639.
[6] "Tau 2014 contest: Pessimism removal of timing analysis," in *http://sites.google.com/site/taucontest2014*.
[7] S. Bhardwaj, K. Rahmat, and K. Kucukcaka, "Clock-reconvergence pessimism removal in hierarchical static timing analysis," in *US patent*, 2013.
[8] D. Hathaway, J. P. Alvarez, and K. P. Belkbale, "Network timing analysis method which eliminates timing variations between signals traversing a common circuit path," in *US patent 5636372*, 1997.
[9] A. K. Ravi, "Common clock path pessimism analysis for circuit designs using clock tree networks," in *US patent 7926019*, 2011.
[10] "Incremental timing analysis and incremental cppr," in *http://sites.google.com/site/taucontest2015*.
[11] V. Garg, "Common path pessimism removal: An industry perspective," in *Proc. IEEE/ACM ICCAD*, 2014, pp. 592–595.
[12] C.-H. Tsai and W.-J. Mak, "A fast parallel approach for common path pessimism removal," in *Proc. IEEE/ACM ASPDAC*, 2015, pp. 372–377.
[13] Y.-M. Yang, Y.-W. Chang, and I. H.-R. Jiang, "itimerc: Common path pessimism removal using effective reduction methods," in *Prof. IEEE/ACM ICCAD*, 2014, pp. 600–605.

[14] C. Kalonakis, C. Antoniadis, P. Giannakou, D. Dioudis, G. Pinitas, and G. Stamoulis, "Tktimer: Fast and accurate clock network pessimism removal," in *Proc. IEEE/ACM ICCAD*, 2014, pp. 606–610.

[15] M. A. Bender and M. F. Colton, "The lca problem revisited," in *Proc. 4th Latin American Symposium on Theoretical Informatics*, 2000, pp. 88–94.

[16] H. Aljazzar and S. Leue, "K*: A heuristic search algorithm for finding the k shortest paths," in *Artificial Intelligence*, 2011, pp. 2129–2154.

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

[18] E. Q. V. Martins and M. M. B. Pascoal, "A new implementation of yen's ranking loopless paths algorithm," in *A Quaterly Journal of Operation Research*, 2003.

[19] W. Qiu and D. M. H. Walker, "An efficient algorithm for finding the k longest testable paths through each gate in a combinational circuit," in *Proc. IEEE ITC*, 2003, pp. 592–601.

[20] J. Y. Yen, "Finding the k shortest loopless paths in a network," *Manage. Sci.*, vol. 17, no. 11, pp. 712–716, 1971.

[21] M. Atkinson, J. Sack, N. Santoro, and T. Strothotte, "Min-max heaps and generalized priority queue," in *Commun. ACM*, 1986, pp. 996–1000.

[22] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Chapter 24: Single-source Shortest Paths, Introduction to Algorithm*. MIT Press, 2009.

[23] "Openmp: Parallel programming api," in *http://www.openmp.org*.

[24] "Openmpi: Open-source high-performance computing," in *http://www.open-mpi.org*.

[25] "Illinois campus cluster," in *https://campuscluster.illinois.edu*.

[26] T.-W. Huang and M. D. F. Wong, "Opentimer: A high-performance timing analysis tool," in *Proc. IEEE/ACM ICCAD*, 2015, pp. 895–902.

[27] T.-W. Huang and M. D. F. Wong, "Accelerated path-based timing analysis with mapreduce," in *Proc. ACM ISPD*, 2015, pp. 103–110.

[28] T.-W. Huang and M. D. F. Wong, "On fast timing closure: Speeding up incremental path-based timing analysis with mapreduce," in *Proc. IEEE/ACM SLIP*, 2015, pp. 1–6.

PLACE PHOTO HERE

Tsung-Wei Huang received the B.S. and M.S. degrees from the Department of Computer Science, National Cheng Kung University (NCKU), Tainan, Taiwan, in 2010 and 2011, respectively. He is currently a PhD candidate in the Department of Electrical and Computer Engineering, University of Illinois at Urbana-Champaign (UIUC). His current research interests focus on distributed computing and parallel timing analysis applications.

Tsung-Wei was the recipient of several awards including the 1st place in ACM/SIGDA 2010 Student Research Competition (SRC), the 2nd place in ACM 2011 Student Research Competition Grand Final across all disciplines, the 1st place in TAU 2014 Timing Analysis Contest on CPPR, the 2nd place in TAU 2015 Timing Analysis on Incremental Timing and Incremental CPPR, the 2nd place in ACM/SIGDA 2014 CADathlon Programming contest, A. Richard Newton Young Student Fellow Award in 2014 ACM/IEEE Design Automation Conference (DAC), and 2015 Rambus Outstanding Computer Engineering Research Fellowship in UIUC.

PLACE PHOTO HERE

Martin D. F. Wong received his B.S. degree in Mathematics from the University of Toronto and M.S. degree in Mathematics from the University of Illinois at Urbana-Champaign (UIUC). He obtained his Ph.D. degree in Computer Science from UIUC in 1987. From 1987 to 2002, he was a faculty member in Computer Science at the University of Texas at Austin. He returned to UIUC in 2002 where he is currently the Executive Associate Dean for the College of Engineering and the Edward C. Jordan Professor in Electrical and Computer Engineering.

He has published over 400 technical papers and graduated more than 45 Ph.D. students in the area of Electronic Design Automation (EDA). He has won a few best paper awards for his works in EDA and has served on many technical program committees of leading EDA conferences. He has also served on the editorial boards of IEEE Transactions on Computers, IEEE Transactions on Computer-Aided Design (TCAD), and ACM Transactions on Design Automation of Electronic Systems (TODAES). He is a Fellow of IEEE.