A Provably Good and Practically Efficient Algorithm for Common Path Pessimism Removal in Large Designs

Zizheng Guo
CECA, CS Department
Peking University
Beijing, China
gzz@pku.edu.cn

Tsung-Wei Huang
ECE Department
University of Utah
Salt Lake City, USA
tsung-wei.huang@utah.edu

Yibo Lin*
CECA, CS Department
Peking University
Beijing, China
yibolin@pku.edu.cn

***Abstract*—Common path pessimism removal (CPPR) is imperative for eliminating redundant pessimism during static timing analysis (STA). However, turning on CPPR can significantly increase the analysis runtime by 10–100× in large designs. Recent years have seen much research on improving the algorithmic efficiencies of CPPR, but most are architecturally constrained by either the speed-accuracy trade-off or design-specific pruning heuristics. In this paper, we introduce a novel CPPR algorithm that is provably good and practically efficient. We have evaluated our algorithm on large industrial designs and demonstrated promising performance over the current state-of-the-art. As an example, our algorithm outperforms the baseline by 36–135× faster when generating the top-10K post-CPPR critical paths on a million-gate design. At the extreme, our algorithm with one core is even 4–16× faster than the baseline with 8 cores.**

I. Introduction

Static timing analysis (STA) is a pivotal step in the overall design flow [1]. The predominant approach creates early and late bounds on each signal delay. However, this early/late timing split causes the analysis to be artificially pessimistic due to analyzing only the worst-case scenarios. Unnecessary pessimism will lead tests to be marked failing whereas in actuality they should be passing. Designers and optimization tools might be misled into an over-pessimistic timing report, leading to unnecessary increases in design turnaround time and cost. To this end, *common path pessimism removal* (CPPR) is imperative for eliminating redundant pessimism during STA. Figure 1 gives an example. Prior to CPPR, data path 2 can be more critical than data path 1, but the result may change after CPPR because the common path pessimism 2 is larger than pessimism 1. However, this process of pessimism removal is extremely time-consuming due to the *path-by-path* timing anaylsis across all flip-flop (FF) pairs. According to [2], generating a complete timing report with CPPR incurs 10-100× more runtime and memory.

The recent years have seen several research work and algorithms to reduce the long runtimes of CPPR. For instance, the TAU community has organized contests to seek new ideas for accurate and fast CPPR algorithms [3], [4]. iTimerC [5] employs a branch-and-bound technique to prune the search space of path generation. HappyTimer [6] designs a block-based algorithm with an alternative delay metric to remove pessimism during the timing update. OpenTimer [2] introduces a dual data structure to remove path pessimism and parallelize the process across independent FFs. There is also research on improving memory consumption of CPPR [7], [8]. Other approaches such as tag-based updates and modified delay models have been applied in commercial tools [4]. A fundamental challenge is that existing algorithms encounter large time and space complexities proportional to the product of FF count and the graph size, because they may end up enumerating all possible FF pairs for CPPR. As a result, even introducing speed-accuracy trade-off or design-specific pruning heuristics cannot guarantee consistent, decent performance in large designs.

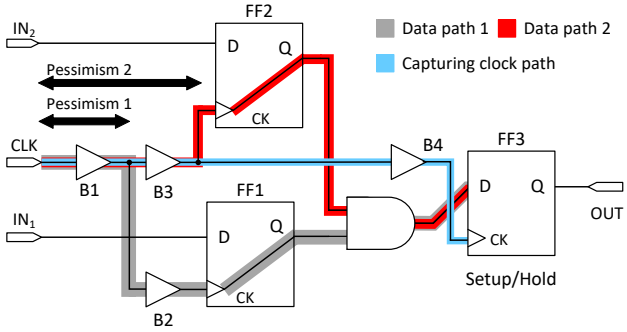

Fig. 1: An example of CPPR impact. Before CPPR, data path 2 can be more critical than data path 1. However, the post-CPPR slack of data path 2 can become less critical than data path 1, as pessimism 2 is larger than pessimism 1.

In this paper, we introduce a novel provably good and practically efficient CPPR algorithm for analyzing large designs. We summarize three technical contributions of our work: (1) First, instead of enumerating all possible FF pairs, we identify lowest common ancestors (LCA) that incur common path pessimism on data paths, and process the FF pairs in different LCA depth groups. Then, we design an efficient distance tuple structure to deal with depth constraints, largely reducing the search space of CPPR. (2) Second, we prove that the time complexity of our algorithm is irrelevant to the number of FFs, but the depth of the clock tree which is typically smaller by orders of magnitude. (3) Third, our algorithm is highly parallelizable across the depths of the clock tree. This organization largely facilitates the adoption of multithreading to gain further speed-up through manycore parallelism.

We have evaluated our algorithm on large industrial designs of millions of gates and compared our performance with three state-of-the-art CPPR algorithms [2], [6], [5]. Our algorithm significantly outperforms the baseline algorithms in runtime. For instance, when generating one post-CPPR critical path, we are 3–23× faster. The difference becomes even remarkable at a large path count. When generating the top-10K post-CPPR critical paths, our algorithm is 36–135× faster than the others. At the extreme, our algorithm of one core (single-threaded) is even 4–16× faster than the baseline of eight cores where performance scalability stagnates. The following section titles are self-explanatory.

II. Preliminaries

In STA, a circuit is represented as a directed acyclic graph (DAG), where nodes denote pins and edges denote interconnections between pins. FFs are driven by a clock source through the clock tree. Each edge has an early and a late bounds on the signal delay. A data path

*Corresponding author

978-1-6654-3274-0/21/$31.00

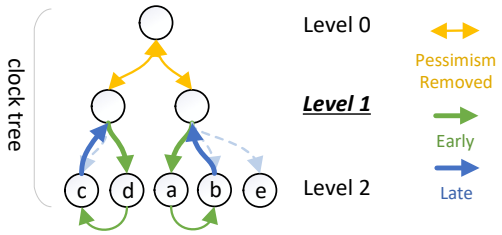

Fig. 2: When fixing the depth of LCA to be 1, we only care about these launching-capturing FF pairs: (c,d), (d,c), (a,b), (a,e), (b,a), (b,e), (e,a), (e,b). We can then tell precisely for each launching FF the pessimism it will introduce when paired with any other capturing FF, namely the edges above Level 1 colored yellow in the figure.

starts from a launching FF or a primary input and ends at a capturing FF. The delay of a path is the sum of the edge delays along the path. We consider the setup and hold timing constraints [3], [4]:

Definition 1. We denote o_i, d_i as the respective clock pin and the data pin of FF i. For a path p from o_1 to d_2, the setup slack and the hold slack of p are defined as follows:

$$\begin{aligned} slack^{\text{setup}}(p) &= rat^{\text{late}}(o_2) - at^{\text{late}}(o_2) \\ &= at^{\text{early}}(o_2) + T_{\text{clk}} - T_{\text{setup}} - at^{\text{late}}(o_1) - delay^{\text{late}}(p), \\ slack^{\text{hold}}(p) &= at^{\text{early}}(o_2) - rat^{\text{early}}(o_2) \\ &= at^{\text{early}}(o_1) + delay^{\text{early}}(p) - at^{\text{late}}(o_2) - T_{\text{hold}}. \end{aligned} \tag{1}$$

The definition of slack assumes a worst case of edge delays for each test. However, it introduces unnecessary pessimism because there can be a common segment between two clock paths (see Figure 1). To remove the pessimism, we add a credit to the slack as follows:

Definition 2. We define CPPR credit on clock tree node u as $credit(u) = at^{\text{late}}(u) - at^{\text{early}}(u)$. The credit for a path with launching FF u and capturing FF v is thus $credit(LCA(u,v))$.

Then, the post-CPPR slack for a path p can be written as [3], [4],

$$slack^{\text{setup/hold}}_{\text{CPPR}}(p) = slack^{\text{setup/hold}}(p) + credit(LCA(u,v)), \tag{2}$$

where $u = p.lauFF, v = p.capFF$ and $slack^{\text{setup/hold}}(p)$ is the pre-CPPR slack. With the above definitions, we formulate the common path pessimism removal problem as follows.

CPPR problem formulation [4]: *Given a circuit graph with updated delay values, timing constraints, and a number k, report the top-k post-CPPR critical paths.*

The key challenge of CPPR is that the credit is *path-specific* and it depends on the launching and capturing FFs. Different paths have different credits to add to the slack, even if they share the same launching or capturing FFs. Most previous work enumerate all FF pairs to find post-CPPR critical paths ended at a target capturing FF and then reduce the result to a top-k set [5], [6], [2]. However, the main drawback is that these algorithms may end up enumerating all FF pairs in the worst case, requiring long analysis runtimes to complete CPPR.

III. Algorithms

We propose a new CPPR algorithm to overcome the runtime challenges of CPPR by enumerating the LCA depths of launching FFs and capturing FFs, instead of a large amount of FF pairs. Figure 2 illustrates our motivation. By fixing a depth and then looking for all possible FF pairs pertaining to this LCA depth, we are able to precisely remove the pessimism and directly get the global top-k post-CPPR critical paths.

TABLE I: Notations of our Algorithms

Notation	Description
D	The number of clock tree levels.
$at^{\text{early/late}}(u)$	Early/late arrival time for clock tree node u.
$delay^{\text{early/late}}(u,v)$	Early/late delay for edge $u \to v$.
$credit(u)$	CPPR credit on clock tree node u.
$depth(u)$	The depth of clock tree node u.
$p.lauFF/capFF$	The launching/capturing FF node of path p.
$f_d(u)$	The ancestor of node u on clock tree with depth d.
$LCA(u,v)$	The lowest common ancestor of u and v.
$slack^{\text{setup/hold}}(p)$	The pre-CPPR slack of path p.
$slack^{\text{setup/hold}}(p,d)$	The slack eliminating the pessimism above level d.
$slack^{\text{setup/hold}}_{\text{CPPR}}(p)$	The post-CPPR slack of path p.
$P^{\text{setup/hold}}_{d}(k)$	Top-k path candidates at level d.
$P^{\text{setup/hold}}_{*}(k)$	Top-k self-loop path candidates.
$P^{\text{setup/hold}}_{\text{PI}}(k)$	Top-k primary input path candidates.
$P^{\text{setup/hold}}_{\text{CPPR}}(k)$	Top-k paths ranked by post-CPPR slack.

A. Definitions and Notations

Definition 3. We break the clock tree into levels at different depths d and define d-Pessimism Removed slack of path p as the pre-CPPR slack of path p eliminating the pessimism above level d, precisely $slack^{\text{setup/hold}}(p,d) = slack^{\text{setup/hold}}(p) + credit(f_d(p.lauFF))$.

Apparently, we have $slack^{\text{setup/hold}}(p,0) = slack^{\text{setup/hold}}(p)$. We rewrite Equation (2) as:

$$\begin{aligned} slack^{\text{setup/hold}}_{\text{CPPR}}(p) &= slack^{\text{setup/hold}}(p,0) + credit(LCA(u,v)) \\ &= slack^{\text{setup/hold}}(p, depth(LCA(u,v))), \end{aligned} \tag{3}$$

where $u = p.lauFF, v = p.capFF$.

Definition 4. We define the set of setup/hold path candidates at level d as setup/hold critical paths p that satisfy these two constraints: 1) $p.lauFF \neq p.capFF$; 2) $depth(LCA(p.lauFF, p.capFF)) \leq d$.

We define the top-k path candidates at level d as $P^{\text{setup/hold}}_{d}(k)$, which are the top-$k$ among the set of setup/hold path candidates at level d, ranked by $slack^{\text{setup/hold}}(p,d)$.

Note that the second constraint requires $depth \leq d$ instead of $depth = d$. This is important as it makes the fast retrieval of $P^{\text{setup/hold}}_{d}(k)$ possible. This definition covers all top-k post-CPPR paths satisfying p satisfying $p.lauFF \neq p.capFF$ (see Lemma 1).

As Definition 4 does not cover paths that have $p.lauFF = p.capFF$, we define another type of path candidates as follows:

Definition 5. We define self-loop paths as paths that satisfy $p.lauFF = p.capFF$. We define top-k self-loop path candidates as $P^{\text{setup/hold}}_{*}(k)$, which are the top-$k$ among all setup/hold critical paths ranked by $slack^{\text{setup/hold}}(p, depth(p.lauFF))$.

Note that Definition 5 considers both self-loop paths and non-self-loop paths and ranks them by $slack^{\text{setup/hold}}(p, depth(p.lauFF))$. We shall show (in Lemma 2) that this definition covers all global top-k post-CPPR paths that are self-loop paths. The above definitions are for paths that originate from a FF. We also consider paths that originate from a primary input pin:

Definition 6. We define top-k primary input path candidates as $P^{\text{setup/hold}}_{\text{PI}}(k)$, which are the top-$k$ among all setup/hold critical paths that originate from a primary input, ranked by their slacks. Paths that originate from primary inputs do not have pessimism to remove.

B. The Overall Algorithm

The overall algorithm is presented in Algorithm 1. The algorithm consists of two stages: *path candidates generation* and *top paths selection*. We generate path candidates based on enumeration of the depth of LCA between launching FF and capturing FF (line 2), self-loop path candidates (line 3) and primary input path candidates (line

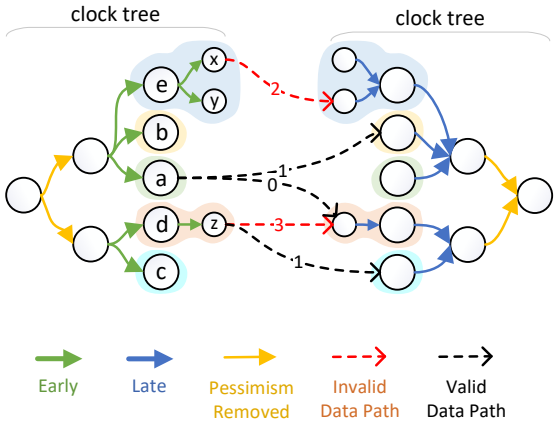

Fig. 3: Example of node grouping with $d = 1$ for hold check. In this case, nodes are grouped using $f_2(u)$, forming 5 different groups, e.g., node x's group is e, node a's group is a, etc. We disallow data paths that connect the same group, i.e., that have $f_{d+1}(p.lauFF) = f_{d+1}(p.capFF)$. Invalid data paths are marked in red. All data paths are labeled with their LCA depths. Each valid data path p satisfies $p.lauFF \neq p.capFF$ and LCA depth $\leq d$.

4). A total of up to $k(D+2)$ path candidates are generated. After that, we select the top-k of all path candidates with smallest post-CPPR slack values (line 6), and output them. We elaborate on the subroutines in more detail and prove the correctness in the following subsections.

Algorithm 1: `getPostCPPRPaths`(k, *mode*=setup/hold)

1 **for** d = *0, 1, 2, ..., $D-1$* **do**
2 $\quad P_d^{mode}(k) \leftarrow$`getPathsAtLCALevel`$(d, k, mode)$;
3 $P_*^{mode}(k) \leftarrow$`getPathsFromSelfLoops`$(k, mode)$;
4 $P_{\text{PI}}^{mode}(k) \leftarrow$`getPathsFromPIs`$(k, mode)$;
5 $paths \leftarrow [P_0^{mode}(k), ..., P_{D-1}^{mode}(k), P_*^{mode}(k), P_{\text{PI}}^{mode}(k)]$;
6 **return** $P_{\text{CPPR}}^{mode}(k) =$`selectTopPaths`$(paths, k)$;

C. Generation of the Top-1 Path

We first propose an efficient algorithm to generate path candidates for $k = 1$, including top-1 path candidates at each level (Definition 4), top-1 self-loop path candidate (Definition 5) and top-1 primary input path candidate (Definition 6). This algorithm will generalize to our top-k case. After generating all top-1 path candidates, we can reduce them to the global top-1 path using `selectTopPaths`.

We introduce a *node grouping* technique to find path candidates at different levels (Definition 4). In Figure 3, we demonstrate how node grouping helps us filter out paths that are not path candidates. When generating path candidates at level d, we group each node u satisfying $depth(u) > d$ by $f_{d+1}(u)$. Intuitively, we cut the tree between level d and level $d+1$, and the tree below level $d+1$ breaks into pieces which are formed as groups. The path constraints in Definition 4 are equivalent to finding paths that connect two different groups, i.e. $f_{d+1}(p.lauFF) \neq f_{d+1}(p.capFF)$.

Algorithm 2 generates top-1 path candidates at level d (Definition 4) with *node grouping*. The notations are summarized in Table II. We traverse the circuit graph to compute the earliest (latest) arrival time tuples of each pin for hold (setup) constraint. We keep two arrival time tuples, $at(u)$ and $at'(u)$, for each pin u. The $at'(u)$ serves as a fallback for $at(u)$ when $at(u)$ is unavailable due to the node grouping requirement that the capturing FF must have a different group index than the launching FF.

TABLE II: Arrival time tuples on pin u for Algorithm 2.

Name	Member	Description
$at(u)$	*time*	Earliest arrival time
	from	The previous node of the earliest path
	groupid	The group index of the first node of that path
$at'(u)$	*time*	Second earliest arrival time with a different groupid
	from	The previous node of that path
	groupid	The group index of the first node of that path

Above is for hold check. Replace *earliest* with *latest* for setup check.

Algorithm 2: `getPathsAtLCALevel`(d, $k = 1$, *mode*)

1 **for** *FF clock pin u with $depth(u) > d$* **do**
2 $\quad v \leftarrow$ the Q-pin of u;
3 $\quad$ **if** *mode = setup* **then**
4 $\quad\quad Q\_at \leftarrow at^{\text{late}}(u) + delay^{\text{late}}(u, v) - credit(f_d(u))$;
5 $\quad$ **else**
6 $\quad\quad Q\_at \leftarrow at^{\text{early}}(u) + delay^{\text{early}}(u, v) + credit(f_d(u))$;
7 $\quad$ Update $at(v)$ and $at'(v)$ with $time = Q\_at$, $from = u$, $groupid = f_{d+1}(u)$;
8 **for** *Circuit pin u in topological order* **do**
9 $\quad$ **for** *Edge $u \rightarrow v$* **do**
10 $\quad\quad$ **if** *mode = setup* **then** $d \leftarrow delay^{\text{late}}(u, v)$;
11 $\quad\quad$ **else** $d \leftarrow delay^{\text{early}}(u, v)$;
12 $\quad\quad$ Update $at(v)$ and $at'(v)$ with $time = at(u).time + d$, $from = u$, $groupid = at(u).groupid$;
13 $\quad\quad$ Update $at(v)$ and $at'(v)$ with $time = at'(u).time + d$, $from = u$, $groupid = at'(u).groupid$;
14 **for** *FF clock pin u with $depth(u) > d$* **do**
15 $\quad v \leftarrow$ the D-pin of u;
16 $\quad T_{\text{setup/hold}} \leftarrow$ the setup/hold constraint value;
17 $\quad$ **if** $at(v).groupid = f_{d+1}(u)$ **then** $D\_at \leftarrow at'(v).time$;
18 $\quad$ **else** $D\_at \leftarrow at(v).time$;
19 $\quad$ **if** *mode = setup* **then**
20 $\quad\quad T_{\text{clk}} \leftarrow$ clock period;
21 $\quad\quad slack \leftarrow at^{\text{early}}(u) + T_{\text{clk}} - T_{\text{setup}} - D\_at$;
22 $\quad$ **else** $slack \leftarrow D\_at - (at^{\text{late}}(u) + T_{\text{hold}})$;
23 $\quad$ Obtain one path with slack = *slack*;
24 **return** path with smallest slack;

First, we initialize the arrival time for Q-pins of FFs in the arrival time arrays (lines 1-7). We offset the arrival time of Q-pins by $credit(f_d(u))$ (lines 4 and 6), because we are interested in $slack^{\text{setup/hold}}(p, d)$, as Definition 4 required. Then, we propagate the arrival time tuples through a topological order of the pins in the graph (lines 8-13). After that, we compute slacks on each D-pin of FF (lines 14-23). For a FF with clock pin u and D-pin v, we are interested in paths that end at v and start at a Q-pin of another FF, whose clock pins reside in a different group than u. We find the best of such path using $at(v)$ and $at'(v)$ in lines 17-18. Specifically, if $at(v)$ is a path that originates from a different group, we accept it; if not, we accept the fallback, i.e., $at'(v)$. Finally, we select the path with smallest $slack^{\text{setup/hold}}(p, d)$. This slack value is computed in line 21 and 22, derived from Equation (1), with $D\_at = Q\_at(p.lauFF) + delay(p)$.

Algorithm 3 finds self-loop path candidates (Definition 5). As Definition 5 does not limit the range of paths as Definition 4 does, the algorithm is a simplified version of Algorithm 2, where we do not maintain group indices or fallbacks for arrival time tuples. First, we initialize the arrival time for Q-pins (lines 1-7). For self-loop path candidates, we need to rank paths by $slack^{\text{setup/hold}}(p, depth(p.lauFF))$, so we offset the arrival time of Q-pins by $credit(u)$. Then, we do arrival time propagation (lines 8-12), slack computation (lines 13-21), and finally select the path with smallest slack.

Algorithm 3: `getPathsFromSelfLoops`($k = 1$, *mode*)

1 **for** *FF clock pin* u **do**
2 $v \leftarrow$ the Q-pin of u;
3 **if** *mode* = *setup* **then**
4 $Q\_at \leftarrow at^{\text{late}}(u) + delay^{\text{late}}(u, v) - credit(u)$;
5 **else**
6 $Q\_at \leftarrow at^{\text{early}}(u) + delay^{\text{early}}(u, v) + credit(u)$;
7 Update $at(v)$ with $time = Q\_at$, $from = u$;
8 **for** *Circuit pin* u *in topological order* **do**
9 **for** *Edge* $u \to v$ **do**
10 **if** *mode* = *setup* **then** $d \leftarrow delay^{\text{late}}(u, v)$;
11 **else** $d \leftarrow delay^{\text{early}}(u, v)$;
12 Update $at(v)$ with $time = at(u).time + d$, $from = u$;
13 **for** *FF clock pin* u **do**
14 $v \leftarrow$ the D-pin of u;
15 $T_{\text{setup/hold}} \leftarrow$ the setup/hold constraint value;
16 **if** *mode* = *setup* **then**
17 $T_{\text{clk}} \leftarrow$ clock period;
18 $slack \leftarrow at^{\text{early}}(u) + T_{\text{clk}} - T_{\text{setup}} - at(v).time$;
19 **else**
20 $slack \leftarrow at(v).time - (at^{\text{late}}(u) + T_{\text{hold}})$;
21 Obtain one path with slack = $slack$;
22 **return** path with smallest slack;

Algorithm 4: `getPathsFromPIs`($k = 1$, *mode*)

1 **for** *Primary input pin* u **do**
2 $PI\_at \leftarrow$ the early/late arrival time of u for *mode*=hold/setup;
3 Update $at(u)$ and $at'(u)$ with $time = PI\_at$, $from$ =N/A, $groupid$ =N/A;
4 Propagate $at(u)$ for circuit pin u in topological order, same as Algorithm 3 line 8-12;
5 Obtain paths at FF clock pins, same as Algorithm 3 line 13-21;
6 **return** path with smallest slack;

Algorithm 4 finds primary input path candidates (Definition 6). This algorithm is similar to Algorithm 3, except that we initialize the arrival time of primary inputs in lines 1-3 rather than the arrival time of Q-pins. There are no common paths in primary input path candidates, so this time we do not offset the arrival time.

D. Generation of Top-k Paths

We now present our algorithm for generating the top-k path candidates where $k > 1$. We extend our algorithm for $k = 1$ to support generating k path candidates. We represent a path implicitly using a list of deviation edges, and generate paths progressively from previous paths, inspired by [2]. We demonstrate the idea of deviation edges in Figure 4. Adding a deviation edge to a path will increase its slack, and we compute the amount of increase using fallbacks provided by our arrival time tuples. For brevity, we define:

$$at_{\text{auto}}(u, gid) = \begin{cases} at(u), & at(u).groupid \neq gid, \\ at'(u), & at(u).groupid = gid. \end{cases}$$

The algorithm for generating top-k path candidates at level d is presented in Algorithm 5. First, the arrival time arrays $at(u)$ and $at'(u)$ are computed in the same way as Algorithm 2. Then, paths with the smallest slack on each capturing FF are pushed into a min-max-heap (lines 3-7), with computed slacks the same as Algorithm 2. After that, we repeatedly pop a path with minimal slack from the min-max-heap, output it, and then push all its deviations into heap

Algorithm 5: `getPathsAtLCALevel`(d, k, *mode*)

1 Compute and propagate arrival time tuples, same as Algorithm 2 lines 1-13;
2 $H \leftarrow$ new Min-Max-Heap of paths ranked by $p.slack$;
3 **for** *FF clock pin* u *with* $depth(u) > d$ **do**
4 $v \leftarrow$ the D-pin of u;
5 $constraint \leftarrow$ the setup/hold constraint value;
6 Compute smallest slack at v, same as Algorithm 2 lines 17-22;
7 Push one path p into H with $p.slack = slack$, $p.groupid = f_d(u)$, $p.pos = v$, $p.devlist = []$;
8 **for** i = *1 to* k **do**
9 $p \leftarrow$ pop path with smallest slack from H;
10 Output path p as i-th smallest slack path candidate;
11 $u \leftarrow p.pos$;
12 **while** u *is not a clock tree node* **do**
13 $from \leftarrow at_{\text{auto}}(u, p.groupid).from$;
14 **for** *edge* $w \to u$ *where* $w \neq from$ **do**
15 **if** *mode* = *setup* **then**
16 $cost \leftarrow at_{\text{auto}}(u, p.groupid).time - at_{\text{auto}}(w, p.groupid).time - delay^{\text{late}}(w, u)$;
17 **else**
18 $cost \leftarrow at_{\text{auto}}(w, p.groupid).time + delay^{\text{early}}(w, u) - at_{\text{auto}}(u, p.groupid).time$;
19 Push one path p' into H with $p'.slack = p.slack + cost$, $p'.groupid = p.groupid$, $p'.pos = w$, $p'.devlist = p.devlist + [w \to u]$;
20 $u \leftarrow from$;

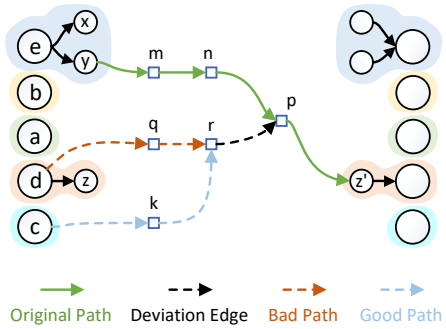

Fig. 4: Illustration of deviation edge and its effect. Assume the shortest path to z' is $CLK \to y, y \to m, m \to n, n \to p, p \to z'$. Deviation happens when we choose to go to p from another direction r, and the deviation edge is $r \to p$ that replaces $n \to p$ in the original path. In the example, we can go from a launching FF to r by two paths. When node grouping is used, we do not consider the one tagged "bad path" because it originates from d which is in the same group as the capturing FF z'.

again (lines 8-20). We enumerate deviations by traversing backwards on the path (the loop at line 12), and enumerate all incoming edges for nodes on the path (the loop at line 14). For each deviation edge, we compute its *cost* by equations at line 16 and 18. This cost is always non-negative, because we are deviating from a more pessimistic path to a less pessimistic one by introducing a suboptimal edge. The resulting deviated path is pushed back to the heap and the loop continues.

The algorithm for generating top-k self-loop path candidates and top-k primary input path candidates is similar to Algorithm 5, except that we do not add constraints to the group of nodes. Specifically, we replace the occurrence of $at_{\text{auto}}(u, gid)$ by $at(u)$ and discard gid. All

Algorithm 6: `selectTopPaths`($paths$, k)

1 $[P_0^{mode}(k), ..., P_{D-1}^{mode}(k), P_*^{mode}(k), P_{\text{PI}}^{mode}(k)] \leftarrow paths$;
2 $H \leftarrow$ new Min-Max-Heap of paths ranked by $p.slack$;
3 **for** d = *0, 1, 2, ...,* $D-1$ **do**
4 **for** *path* p *in* $P_d^{mode}(k)$ **do**
5 **if** $depth(LCA(p.launchFF, p.captureFF)) = d$ **then**
6 Push p into H;
7 **for** *path* p *in* $P_*^{mode}(k)$ **do**
8 **if** $p.launchFF = p.captureFF$ **then**
9 Push p into H;
10 **for** *path* p *in* $P_{PI}^{mode}(k)$ **do**
11 Push p into H;
12 **return** top-k paths in H;

other code for maintaining the heap and generating deviated paths is the same.

After getting all path candidates, we reduce them to the global top-k paths using Algorithm 6. We get paths with LCA depth d from $P_d^{mode}(k)$, self-loop paths from $P_*^{mode}(k)$, and primary-input paths from $P_{\text{PI}}^{mode}(k)$. We discard other path candidates that are not used (lines 5 and 8). We push the paths into a heap and finally extract the top-k among them.

E. Parallelization

The main Algorithm 1 calls procedures `getPathsAtLCALevel`, `getPathsFromSelfLoops`, and `getPathsFromPIs` for a total of $D+2$ times. Each time we perform an iteration on the graph. The iterations are independent of each other and hence we can perform parallel iterations with T threads. The `selectTopPaths` procedure can run iteratively, in which each thread locks and updates the global heap once it finishes one call. The majority of runtime lies in the calls to `getPathsAtLCALevel`, `getPathsFromSelfLoops`, and `getPathsFromPIs`, while the runtime of iterative top path selection is negligible. Thus, our algorithm is highly parellelizable.

F. Correctness and Complexity

Due to page limit, we only show our main theorems. We recommend readers to check our lemmas and detailed proof at this link[1].

For correctness, we have the following theorem:

Theorem. *With all the path candidates,* `selectTopPaths` *(Algorithm 6) correctly selects and returns global top-k paths ranked by their post-CPPR slacks.*

For time and space complexity, we have the following theorem:

Theorem. *Algorithm 1 runs in $O(nD)$ time complexity for $k = 1$, and $O(nDk \log k)$ for $k > 1$. The space complexity is $O(T(n+k)+kp)$ for T threads, and $p < n$ denotes the average length of critical paths.*

IV. Experimental Results

We implemented our algorithm in C++ and evaluated the performance of our algorithm on a 64-bit Linux machine with 40 cores Intel Xeon CPU at 2.20 GHz and 960 GB memory. We conducted experiments on large industrial designs from TAU contests [3], [4], and their statistics are shown in Table III. The levels of the clock trees are about 100 in all benchmarks, 300–1700× smaller than the number of FFs.

We compare our approach with three state-of-the-art timers: an open-source tool (OpenTimer [2]), and the TAU 2014 contest winners (HappyTimer [6] and iTimerC [5]). Since HappyTimer and iTimerC are not open-source, we acquired their executables directly from the authors. We do not compare with commercial tools because of different application scopes. Our scope targets a standalone research environment, but commercial tools need to deal with many other components in the closure flow even though many of them may not be directly related to CPPR. It is very difficult to come up with a fair comparison for the CPPR problem itself. Indeed, OpenTimer [2] has reported significant speedup over commercial tools. Table IV lists the overall performance comparison. We measure the runtime and memory consumption on computing the global top-k post-CPPR slacks on the designs listed in Table III where k=1, 100, 10K, for both setup and hold tests. We tested our timer for both 1 and 8 threads, as it starts saturating at 8 threads. OpenTimer and iTimerC are tested using 8 threads. HappyTimer is tested using 1 thread because it does not support multi-threading. We did not include accuracy metrics because our proposed algorithm generates full accuracy results.

TABLE III: Benchmark statistics.

Benchmark	#Edges	#FFs	D	#FFs/D	FF connectivity
vga_lcdv2	449651	25091	56	448.05	28.55
Combo4v2	778638	26760	82	326.34	37.93
Combo5v2	2051804	39525	91	434.34	22.34
Combo6v2	3577926	64133	101	634.98	37.11
Combo7v2	2817561	54784	96	570.67	32.81
netcard_iccad	3999174	97831	75	1304.41	196.42
leon2_iccad	4328255	149381	85	1757.42	1245.44
leon3mp_iccad	3376832	108839	75	1451.19	489.06

Our timer is faster than all baseline timers by at least 2.41×. The largest speedup of our timer with 8 threads is 96.28× compared to OpenTimer, 217.51× compared to HappyTimer, and 87.46× compared to iTimerC. Our timer with a single thread can achieve up to 89.01× speedup (`Combo4v2`, k=10K) compared to HappyTimer. The average speedup ratios (baseline over ours), for k=1 are 22.69, 20.83, and 3.28 compared to OpenTimer, HappyTimer, and iTimerC respectively. The ratios for k=10K are 51.80, 135.21, and 36.47, respectively. The large runtime gains come from the fundamental difference of the time complexity. All baselines can end up with enumerating all pairs of FFs (#FFs in Table III), while our algorithm depends only on the depth of the clock tree (D in Table III), which is 300-1700× smaller. These results demonstrate the effectiveness and efficiencies of our algorithm to reduce the long runtimes of CPPR.

For memory consumption, OpenTimer uses up to 22.57× more memory than our timer. Although we use more memory than OpenTimer when $k \leq 100$, our timer with 1 thread already outperforms OpenTimer up to 33.37× (`leon2`, k=10K) with very little memory overhead.

HappyTimer and iTimerC adopt design-specific pruning heuristics and achieve good performance on designs (e.g., `vga_lcdv2`, `leon3mp`) with small k, but they do not scale well to large k. For example, HappyTimer leverages the sparsity of the connection between launching and capturing FFs for pruning, but such an assumption fails at designs with high "FF connectivity" (defined as the average number of capturing FFs that can be reached from each launching FF). As a result, it becomes extremely slow and memory-intensive on large designs such as `leon2` in Table III.

Figure 5 draws the runtime and memory consumption versus k, the number of post-CPPR critical paths requested. Our algorithm runs very fast for all number of paths, while the runtime of iTimerC rises rapidly when k increases from 1K to 10K. Meanwhile, our algorithm has a steady memory consumption regardless of k, while the memory usage of OpenTimer and iTimerC explode when k is large. We attribute the decent scalability over k to the elimination of FF enumeration and the progressive path generation.

[1]https://guozz.cn/publication/cpprdac-21/proof.pdf

TABLE IV: Performance comparison between OpenTimer (8 threads), HappyTimer (1 thread), iTimerC (8 threads) and ours (both 1 thread and 8 threads are tested) to find the top-k post-CPPR critical paths on large circuit designs.

Benchmark	k	OpenTimer 8 Threads		HappyTimer 1 Thread		iTimerC 8 Threads		Ours 8 Threads		Ours 1 Thread		OpenTimer 8 Threads		HappyTimer 1 Thread		iTimerC 8 Threads		Ours 8 Threads		Ours 1 Thread	
		RT	Mem	RT	Mem	RT	Mem	RT	Mem	RT	Mem	RTR	MemR	RTR	MemR	RTR	MemR	RTR	MemR	RTR	MemR
vga_lcdv2	1	18.05	0.25	13.12	1.74	9.85	0.84	3.90	0.37	7.98	0.17	4.63	0.67	3.36	4.66	2.53	2.24	1.00	1.00	2.05	0.46
	100	18.19	0.26	29.10	6.20	10.30	0.88	3.51	0.37	7.89	0.17	5.18	0.69	8.29	16.62	2.93	2.35	1.00	1.00	2.25	0.46
	10K	64.42	2.32	186.93	7.35	70.12	2.84	4.03	0.39	10.53	0.18	15.99	5.94	46.38	18.79	17.40	7.27	1.00	1.00	2.61	0.46
Combo4v2	1	37.72	0.46	31.89	2.20	25.61	2.12	6.89	0.66	14.62	0.31	5.47	0.70	4.63	3.35	3.72	3.23	1.00	1.00	2.12	0.47
	100	36.76	0.49	50.73	8.75	64.69	2.19	6.30	0.66	14.76	0.31	5.83	0.74	8.05	13.34	10.27	3.33	1.00	1.00	2.34	0.47
	10K	380.93	15.40	1583.48	52.84	636.72	10.08	7.28	0.68	17.79	0.32	52.33	22.57	217.51	77.47	87.46	14.78	1.00	1.00	2.44	0.47
Combo5v2	1	190.98	1.21	61.99	3.02	77.05	7.97	19.48	1.72	41.46	0.81	9.80	0.70	3.18	1.75	3.96	4.64	1.00	1.00	2.13	0.47
	100	191.61	1.25	94.32	10.68	85.42	8.03	18.85	1.72	41.00	0.81	10.16	0.73	5.00	6.21	4.53	4.67	1.00	1.00	2.18	0.47
	10K	831.84	27.34	2639.89	55.96	1009.92	15.60	20.25	1.75	50.38	0.82	41.08	15.66	130.36	32.06	49.87	8.93	1.00	1.00	2.49	0.47
Combo6v2	1	568.08	2.18	185.91	5.94	140.57	14.92	33.70	2.97	67.08	1.38	16.86	0.73	5.52	2.00	4.17	5.03	1.00	1.00	1.99	0.47
	100	567.58	2.22	251.80	22.35	142.00	14.97	32.85	2.97	70.87	1.38	17.28	0.75	7.67	7.53	4.32	5.04	1.00	1.00	2.16	0.47
	10K	1333.36	31.27	3037.02	59.56	961.51	26.04	33.85	2.98	77.89	1.39	39.39	10.48	89.72	19.96	28.41	8.73	1.00	1.00	2.30	0.47
Combo7v2	1	376.45	1.73	136.06	5.05	91.34	12.17	25.15	2.34	53.70	1.09	14.97	0.74	5.41	2.16	3.63	5.19	1.00	1.00	2.14	0.47
	100	382.07	1.78	186.31	18.35	122.26	12.25	24.03	2.34	51.55	1.09	15.90	0.76	7.75	7.83	5.09	5.23	1.00	1.00	2.15	0.47
	10K	1556.68	45.97	4992.00	108.33	1493.34	25.15	25.99	2.36	56.41	1.10	59.90	19.46	192.07	45.85	57.46	10.64	1.00	1.00	2.17	0.47
netcard	1	976.41	1.79	445.09	50.81	110.60	7.40	39.70	3.23	114.21	1.45	24.59	0.55	11.21	15.74	2.79	2.29	1.00	1.00	2.88	0.45
	100	977.27	1.80	985.09	313.78	94.55	7.43	39.80	3.23	115.25	1.45	24.55	0.56	24.75	97.20	2.38	2.30	1.00	1.00	2.90	0.45
	10K	2749.46	74.06	MLE	MLE	582.47	17.42	41.59	3.25	116.67	1.46	66.11	22.82	MLE	MLE	14.01	5.37	1.00	1.00	2.81	0.45
leon2	1	3131.08	2.12	4377.00	446.80	104.61	8.81	43.41	3.55	129.78	1.75	72.13	0.60	100.83	126.03	2.41	2.49	1.00	1.00	2.99	0.49
	100	3131.73	2.13	MLE	MLE	124.62	8.84	45.19	3.55	129.78	1.75	69.30	0.60	MLE	MLE	2.76	2.49	1.00	1.00	2.87	0.49
	10K	4320.00	41.81	MLE	MLE	1177.35	45.30	44.87	3.56	129.47	1.75	96.28	11.74	MLE	MLE	26.24	12.72	1.00	1.00	2.89	0.49
leon3mp	1	1017.85	1.63	1001.66	117.08	94.45	6.55	30.81	2.74	79.17	1.24	33.04	0.60	32.51	42.76	3.07	2.39	1.00	1.00	2.57	0.45
	100	1013.06	1.63	MLE	MLE	93.76	6.57	30.27	2.74	81.37	1.24	33.47	0.60	MLE	MLE	3.10	2.40	1.00	1.00	2.69	0.45
	10K	1348.55	12.48	MLE	MLE	340.72	12.82	31.11	2.75	86.76	1.25	43.35	4.53	MLE	MLE	10.95	4.65	1.00	1.00	2.79	0.45
Avg. Ratio	1	-										22.69	0.66	20.83	24.81	3.28	3.44	1.00	1.00	2.36	0.47
	100	-										22.71	0.68	10.25†	24.79†	4.42	3.48	1.00	1.00	2.44	0.47
	10K	-										51.80	14.15	135.21†	38.83†	36.47	9.14	1.00	1.00	2.56	0.47

RT: Runtime in seconds. **RTR**: Runtime ratio. **Mem**: Memory in GB. **MemR**: Memory ratio. **MLE**: Memory limit exceeded (> 960 GB)
† Average ratios for HappyTimer are inaccurate as failure cases of HappyTimer on large designs are not included.

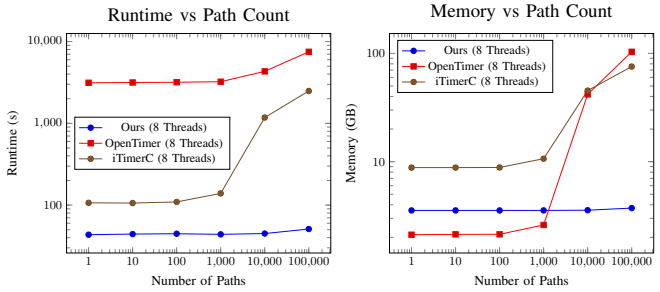

Fig. 5: Runtime and memory values at different numbers of post-CPPR paths (i.e., k) on leon2.

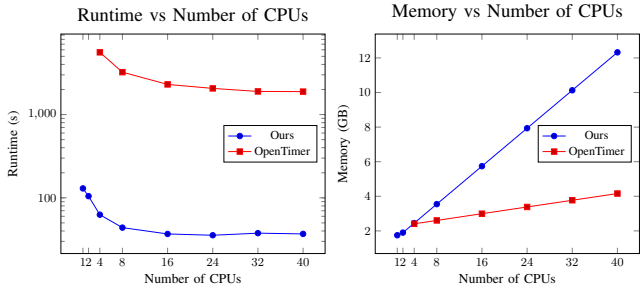

Fig. 6: Runtime and memory values at different numbers of threads for k=1000 on `leon2`. OpenTimer with 1 and 2 threads failed to finish within 3 hours, so we skip those two points.

Figure 6 draws the runtime and memory consumption at different numbers of threads. We do not show iTimerC because its binary is hardcoded for 8 threads. We observe that our algorithm is scalable to different numbers of threads, even though it uses more memory than OpenTimer at a larger thread count. The reason is that we have a slightly larger constant behind our space complexity, in which we use extra arrival time tuples to keep track of paths. However, we believe this is acceptable, given the significant speedup (4–96×) over OpenTimer across all thread numbers we have tested.

V. Conclusion

In this paper, we have proposed a novel provably good and practically efficient CPPR algorithm. Instead of enumerating all the FF pairs, we process the FF pairs in groups of LCA depths to address their common path pessimism in the clock tree, and introduce efficient data structures to reduce the search space for finding post-CPPR paths. We prove the algorithm has a time complexity proportional to the depth of the clock tree, rather than the number of FFs which is typically larger by orders of magnitude. Our algorithm is highly parallelizable. By performing parallel iterations over different, independent LCA depths, our algorithm has achieved 3-23× speedup on generating one post-CPPR critical path, and 36-135× speedup on generating 10K post-CPPR critical paths over the state-of-the-art CPPR algorithms. We plan to extend our algorithm to a GPU target in the future.

Acknowledge

This work was supported in part by the National Science Foundation of China (Grant No. 62034007 and No. 62004006) and Zhejiang Provincial Key R&D program (Grant No. 2020C01052).

References

[1] J. Bhasker and R. Chadha, *Static timing analysis for nanometer designs: A practical approach*. Springer Science & Business Media, 2009.
[2] T. Huang, G. Guo, C. Lin, and M. D. F. Wong, "OpenTimer v2: A New Parallel Incremental Timing Analysis Engine," *IEEE TCAD*, pp. 1–1, 2020.
[3] J. Hu, D. Sinha, and I. Keller, "TAU 2014 contest on removing common path pessimism during timing analysis," in *Proc. ISPD*, 2014, pp. 153–160.
[4] J. Hu, G. Schaeffer, and V. Garg, "TAU 2015 contest on incremental timing analysis," in *Proc. ICCAD*. IEEE, 2015, pp. 882–889.
[5] P.-Y. Lee, I. H.-R. Jiang, C.-R. Li, W.-L. Chiu, and Y.-M. Yang, "iTimerC 2.0: Fast incremental timing and cppr analysis," in *Proc. ICCAD*. IEEE, 2015, pp. 890–894.
[6] B. Jin, G. Luo, and W. Zhang, "A fast and accurate approach for common path pessimism removal in static timing analysis," in *Proc. ISCAS*. IEEE, 2016, pp. 2623–2626.
[7] C. Peddawad, A. Goel, B. Dheeraj, and N. Chandrachoodan, "iitrace: A memory efficient engine for fast incremental timing analysis and clock pessimism removal," in *Proc. ICCAD*, 2015, pp. 903–909.
[8] T. Chung-Hao and M. Wai-Kei, "A fast parallel approach for common path pessimism removal," in *Proc. ASPDAC*, 2015, pp. 372–377.