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

Zizheng Guo, Mingwei Yang, Tsung-Wei Huang *Member, IEEE*, Yibo Lin *Member, IEEE*

***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. Our algorithm also outperforms the commercial STA engine PrimeTime up to 26.99× faster. By exploiting parallelism within the circuit graph, we can reduce the memory consumption of our algorithm by 30%, with only 3% runtime increase.**

***Index Terms*—static timing analysis, STA, CPPR**

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 because we need to analyze timing *path by path* 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 [7], [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 [8], [9]. 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.

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, both within the circuit graph and across the depths of the clock tree. The organization of parallelism across the clock tree depths largely facilitates the adoption of multithreading to gain further speed-up through multicore parallelism. However, the drawback of this parallelism is that its memory consumption is proportional to the number of threads, because each thread works on a duplicate of the circuit graph. By introducing parallelism within the circuit graph, we can reduce the memory consumption of our algorithm on multi-core CPUs without trading in

The preliminary version has been presented at the Design Automation Conference (DAC) in 2021. This work was supported in part by the National Science Foundation of China (Grant No. 62034007 and No. 62004006), National Science Foundation of the US (CCF-2126672), and Zhejiang Provincial Key R&D program (Grant No. 2020C01052).

Z. Guo, M. Yang, and Y. Lin are with the Center for Energy-Efficient Computing and Applications (CECA), School of Electronics Engineering and Computer Science, Peking University, Beijing, China. Corresponding author: Yibo Lin (yibolin@pku.edu.cn).

T.-W. Huang is with The Department of Electrical and Computer Engineering, The University of Utah, Salt Lake City, Utah, USA.

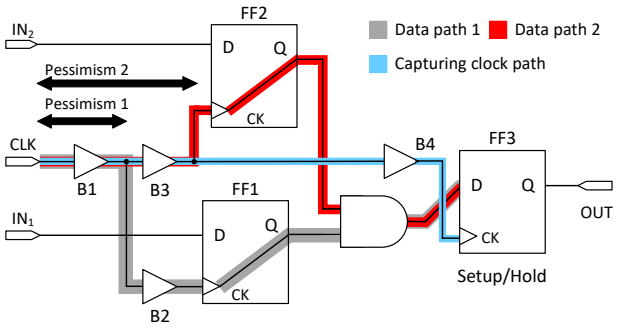


Fig. 1: An example of CPPR impact. Before CPPR, data path 2 can be more critical than data path 1. However, the post-CPPR slack of data path 2 can become less critical than data path 1, as pessimism 2 is larger than pessimism 1.

much runtime.

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 and memory. 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, using only 2.5%–10.9% of their memory. At the extreme, our algorithm of one core (single-threaded parallelism) is even 4–16× faster than the baseline of eight cores where performance scalability stagnates. By exploiting parallelism both within the circuit graph and across the depths of the clock tree, we can halve our memory consumption while still being 29–104× faster in generating the top-10K paths. We also compared our performance with the commercial STA engine PrimeTime, where our algorithm is 18.51× faster on generating top-1 critical path, and 26.99× faster on generating top-10K critical paths.

The rest of the paper is organized as follows. Section II describes the background and motivation; Section III explains the detailed implementation; Section IV demonstrates the results; Section V concludes the paper.

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 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:

$$\begin{aligned} slack^{\text{setup}}(p) &= rat^{\text{late}}(d_2) - at^{\text{late}}(d_2) \\ &= at^{\text{early}}(o_2) + T_{\text{clk}} - T_{\text{setup}} - at^{\text{late}}(o_1) - delay^{\text{late}}(p), \\ slack^{\text{hold}}(p) &= at^{\text{early}}(d_2) - rat^{\text{early}}(d_2) \\ &= at^{\text{early}}(o_1) + delay^{\text{early}}(p) - at^{\text{late}}(o_2) - T_{\text{hold}}. \end{aligned} \tag{1}$$

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 $credit(u) = at^{\text{late}}(u) - at^{\text{early}}(u)$. Each FF corresponds to a clock tree leaf node. 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 might 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.

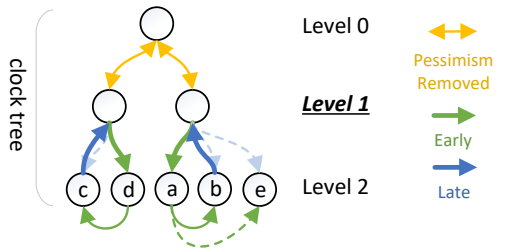


Fig. 2: When fixing the depth of LCA to be 1, we only care about these launching-capturing FF pairs: (c, d), (d, c), (a, b), (a, e), (b, a), (b, e), (e, a), (e, b). We can then tell precisely for each launching FF the pessimism it will introduce when paired with any other capturing FF, namely the edges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

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

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 up to level d.
$slack^{\text{setup/hold}}_{\text{CPPR}}(p)$	The post-CPPR slack of path p.
$P_d^{\text{setup/hold}}(k)$	Top-k path candidates at level d.
$P_*^{\text{setup/hold}}(k)$	Top-k self-loop path candidates.
$P_{\text{PI}}^{\text{setup/hold}}(k)$	Top-k primary input path candidates.
$P_{\text{CPPR}}^{\text{setup/hold}}(k)$	Top-k paths ranked by post-CPPR slack.

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.

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 (i.e. up to) 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 $P_d^{\text{setup/hold}}(k)$, 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 $P_d^{\text{setup/hold}}(k)$ possible. This definition covers all top-k post-CPPR paths 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 in Definition 5, a self-loop path candidate is not necessarily a self-loop path, as we consider both self-loop paths and non-self-loop paths and rank them by $slack^{\text{setup/hold}}(p, depth(p.lauFF))$. We shall show (in Lemma 2) that this definition still covers all self-loop paths present in the global top-k post-CPPR 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 $P_{\text{PI}}^{\text{setup/hold}}(k)$, 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 4). A total of up to $k(D + 2)$ path candidates are generated, of which kD are path candidates at each level, k are self-loop path candidates, and another k are primary input path candidates. 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     P_d^mode(k) ←getPathsAtLCALevel(d, k, mode);
3 P_*^mode(k) ←getPathsFromSelfLoops(k, mode);
4 P_PI^mode(k) ←getPathsFromPIs(k, mode);
5 paths ← [P_0^mode(k), ..., P_{D−1}^mode(k), P_*^mode(k), P_PI^mode(k)];
6 return P_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

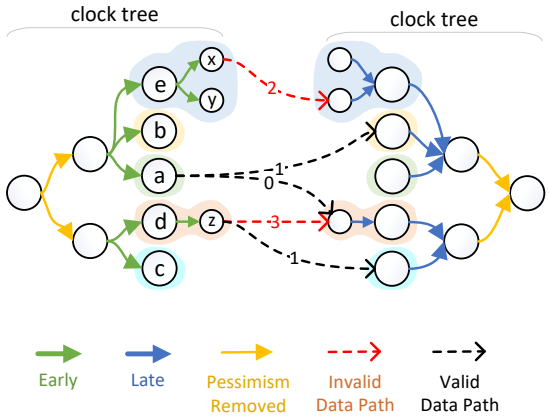


Fig. 3: Example of node grouping with $d = 1$ for hold check. In this case, nodes are grouped using $f_2(u)$, forming 5 different groups, e.g., node x's group is e, node a's group is a, etc. We disallow data paths that connect the same group, i.e., that have $f_{d+1}(p.lauFF) = f_{d+1}(p.capFF)$. Invalid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

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

The above is for hold check. For setup check, replace 'earliest' with 'latest'.

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.

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 14-24). 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 23, 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

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

```
1  for FF clock pin u with depth(u) > d do
2      v ← the Q-pin of u;
3      if mode = setup then
4          Q_at ← at^late(u) + delay^late(u, v) − credit(f_d(u));
5      else
6          Q_at ← at^early(u) + delay^early(u, v) + credit(f_d(u));
7      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      for Edge u → v do
10         if mode = setup then d ← delay^late(u, v) ;
11         else d ← delay^early(u, v) ;
12         Update at(v) and at'(v) with
             time = at(u).time + d, from = u,
             groupid = at(u).groupid;
13         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     v ← the D-pin of u;
16     T_setup/hold ← the setup/hold constraint value;
17     if at(v).groupid = f_{d+1}(u) then D_at ← at'(v).time
         ;
18     else D_at ← at(v).time ;
19     if mode = setup then
20         T_clk ← clock period;
21         slack ← at^early(u) + T_clk − T_setup − D_at;
22     else
23         slack ← D_at − (at^late(u) + T_hold);
24     Obtain one path with slack = slack;
25 return path with smallest slack;
```

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 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 [10], [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:

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

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* $u \rightarrow v$ **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;

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 [11] (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 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 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.

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 17-23;
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* $w \rightarrow u$ *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 $p'.slack = p.slack + cost$, $p'.groupid = p.groupid$, $p'.pos = w$, $p'.devlist = p.devlist + [w \rightarrow u]$;
20 $u \leftarrow from$;

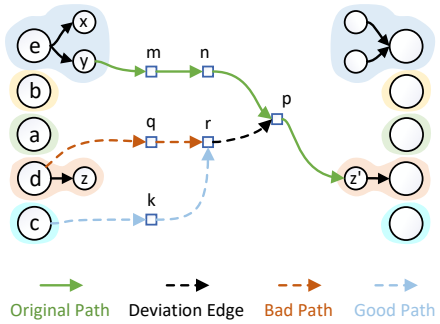


Fig. 4: Illustration of deviation edge and its effect. Assume the shortest path to z' is $CLK \rightarrow y, y \rightarrow m, m \rightarrow n, n \rightarrow p, p \rightarrow z'$. Deviation happens when we choose to go to p from another direction r, and the deviation edge is $r \rightarrow p$ that replaces $n \rightarrow p$ 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'.

E. Parallelization

We now introduce two parallelization strategies of our algorithm that exploit different types of parallelism of the CPPR problem.

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

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

In parallelization strategy #1 (Section III-E1), we exploit parallelism across different iterations (i.e. different clock tree depths). This is our default parallel strategy which demonstrates the best runtime performance on multi-core CPUs. However, its memory footprint is proportional to the number of the threads used to run the algorithm, because each thread works on a duplicate of the circuit graph.

In parallelization strategy #2 (Section III-E2), we overcome the memory issue of parallelization strategy #1 by seeking parallelism within the circuit graph to be processed in each iteration. By parallelizing within the circuit graph, all threads work on the same circuit graph structure in memory. The memory consumption is reduced as a result of the shared circuit graph structure by multiple threads. However, this parallelization strategy can introduce small runtime overhead, because of the following reasons:

1) Tasks on graph nodes enumerate the input edges and compute the arrival time tuples. This workload is small, which makes it hard to fully utilize the CPU core.
2) The maximum number of tasks in parallel is limited by the number of nodes in each level. For the levels at the rear of the circuit, the number of nodes can be smaller than the number of threads.
3) Scheduling and sychronization for parallelization across circuit graph nodes is more complex than parallelization across iterations, which introduces a larger runtime overhead.

In Section III-E3, we combine the parallelization strategies #1 and #2 by allocating threads among the two kinds of parallelism. In this way, we can balance the runtime and memory of our algorithm.

1) Parallelization Strategy #1: Exploit Parallelism across Clock Tree Depths: In this parallelization strategy, each thread computes the path candidates from one iteration (i.e. one clock tree depth), as demonstrated in Figure 5. 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, with the iterations 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.

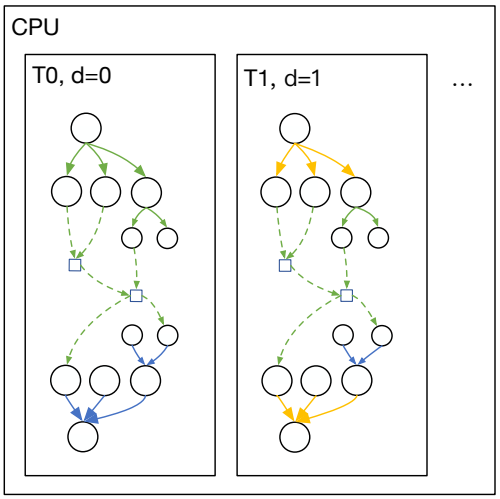


Fig. 5: Parallelization across clock tree depth; i.e., parallelize the algorithm by putting different iterations onto different threads.

The majority of runtime lies in the calls to `getPathsAtLCALevel`, `getPathsFromSelfLoops`, and `getPathsFromPIs`, while the runtime of iterative top path selection is negligible. As a result, this strategy maximizes the CPU parallelism.

2) Parallelization Strategy #2: Exploit Parallelism within the Circuit Graph: In this parallelization strategy, each thread computes the arrival time tuple for a single clock tree node, as demonstrated in Figure 6. For each clock tree depth, we initialize and propagate the arrival time tuples along the clock tree and circuit DAG. The propagation on nodes can be regarded as tasks with dependencies. We can put different tasks onto different threads, provided that the dependencies between tasks are not violated. This can be addressed by either *levelizing* the task graph or using a *dynamic scheduled* parallel programming framework like Taskflow [12], [**?**]. We describe briefly the idea of levelization as follows.

We build the levelization of a DAG iteratively, by maintaining a set of nodes called frontiers, denoted as F. The initial frontiers are nodes that do not have input edges. We iteratively discover the next frontiers F' from the current frontiers F, by deleting all output edges from F and collect the nodes that lose all input edges afterwards. The resulting levels are the frontiers in each iteration. It is guaranteed that nodes within the same level do not have mutual dependency, and they only depend on nodes from previous frontiers. Thus, we process levels one by one, and perform tasks within the same level in parallel.

3) Hybrid Parallelization: We combine the advantages of the above two parallelization strategies by parallelizing both across clock tree depths and within a circuit graph, as illustrated in Figure 7. We assign a group of threads to each iteration (Threads 0, 1 in Figure 7 for iteration $d = 0$, Threads 2, 3 for iteration $d = 1$), and different iterations are computed in parallel by different groups of threads. Within each iteration, threads within the group propagate arrival time tuples on the circuit graph in parallel.

Let T_w denote the number of threads within a group (i.e.,

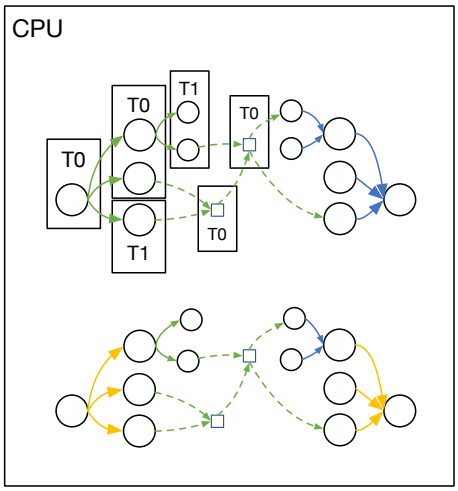


Fig. 6: Parallization within circuit graph; i.e., parallelize the algorithm by distributing nodes to different threads while preserving their dependency (through levelization or dynamic scheduling), and processing the iterations one by one.

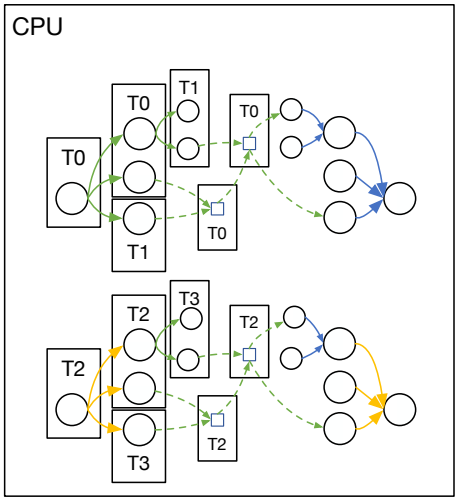


Fig. 7: Parallelize the algorithm by assigning each clock tree depth a group of threads, each working on a batch of nodes. This makes use of both inter-depth and intra-depth parallelism.

the number of threads to compute each iteration), and T_g denote the number of groups (i.e., the number of concurrent iterations). There are a total of $T = T_w \times T_g$ threads, working on only T_g circuit graph instances in memory. When $T_w = 1$, we only parallelize our algorithm across clock tree depths. Similarly, when $T_g = 1$, we only parallelize our algorithm within the circuit graph. Larger T_g introduces higher parallel scalability across independent iterations, and larger T_w reduces overall memory footprint, because we do not need to replicate the clock graph when increasing it. By adjusting T_w and T_g, we can balance the runtime and memory of our algorithm, and get the best performance within the memory budget of independent situations.

F. Correctness and Complexity

The correctness of our algorithm is based on Lemmas 1-3. We show in these lemmas that the global top-k post-CPPR critical paths are covered by the three types of path candidates (See Definitions 4-6 for the three types).

Lemma 1. *For any path $p \in P^{mode}_{CPPR}(k)$ with $p.lauFF \neq p.capFF$ and $depth(LCA(p.lauFF, p.capFF)) = d$, we have $p \in P^{mode}_{d}(k)$.*

This lemma is derived from the fact that we rank path candidates in $P^{mode}_{d}(k)$ by optimistic slack values. Paths with $depth(LCA) < d$ are ranked with slacks larger than their post-CPPR slacks. Paths with $depth(LCA) = d$ are ranked with exact post-CPPR slacks, and thus they will be top-k in $P^{mode}_{d}(k)$ as long as they are global top-k. The detailed proof is presented below.

Proof. By contradiction. Suppose $p \notin P^{mode}_{d}(k)$. Then for any $q \in P^{mode}_{d}(k)$, we have

$$slack^{mode}(q, d) \leq slack^{mode}(p, d).$$

For path p, because $depth(LCA(p.lauFF, p.capFF)) = d$, we have

$$slack^{mode}_{\text{CPPR}}(p) = slack^{mode}(p, d).$$

For any $q \in P^{mode}_{d}(k)$, because $depth(LCA(q.lauFF, q.capFF)) \leq d$, we have

$$slack^{mode}_{\text{CPPR}}(q) \leq slack^{mode}(q, d).$$

Combining the equations above, we get

$$slack^{mode}_{\text{CPPR}}(q) \leq slack^{mode}(q, d) \leq slack^{mode}(p, d) = slack^{mode}_{\text{CPPR}}(p).$$

That means every path $q \in P^{mode}_{d}(k)$ has smaller post-CPPR slack than p. There are a total of k paths in $P^{mode}_{d}(k)$. Thus, p cannot be ranked top-k in $P^{mode}_{\text{CPPR}}(k)$, which is a contradiction. □

Lemma 2. *For any path $p \in P^{mode}_{CPPR}(k)$ with $p.lauFF = p.capFF$, we have $p \in P^{mode}_{*}(k)$.*

Proof. By contradiction. Suppose $p \notin P^{mode}_{*}(k)$. Then for any $q \in P^{mode}_{*}(k)$, we have

$$slack^{mode}(q, depth(q.lauFF)) \leq slack^{mode}(p, depth(p.lauFF)).$$

Whether or not q is a self-loop path, there must be $depth(q.lauFF) \geq depth(LCA(q.lauFF, q.capFF))$, and thus we have

$$slack^{mode}_{\text{CPPR}}(q) \leq slack^{mode}(q, depth(q.lauFF)).$$

On the other side, p is a self-loop path by our assumption, and thus

$$slack^{mode}_{\text{CPPR}}(p) = slack^{mode}(p, depth(p.lauFF)).$$

Combining the equations above, we get

$$\begin{aligned} slack^{mode}_{\text{CPPR}}(q) &\leq slack^{mode}(q, depth(q.lauFF)) \\ &\leq slack^{mode}(p, depth(p.lauFF)) \\ &= slack^{mode}_{\text{CPPR}}(p). \end{aligned}$$

That means every path $q \in P^{mode}_{*}(k)$ has smaller post-CPPR slack than p. There are a total of k paths in $P^{mode}_{*}(k)$. Thus, p cannot be ranked top-k in $P^{mode}_{\text{CPPR}}(k)$, which is a contradiction. □

Lemma 3. *For any path $p \in P_{CPPR}^{mode}(k)$ that originates from a primary input rather than a launching FF, we have $p \in P_{PI}^{mode}(k)$.*

Proof. This one is apparent because every path $q \in P_{\text{PI}}^{mode}(k)$ originates from primary inputs and they are sorted by their post-CPPR slacks. □

The three lemmas draw the following correctness theorem, which implies that we can obtain top-k post-CPPR paths from the path candidates:

Theorem 1. *With all the path candidates, `selectTopPaths` (Algorithm 6) correctly selects and returns global top-k paths ranked by their post-CPPR slacks.*

Proof. Because of Lemma 1, 2 and 3, we have encountered every path that has the potential to become one of the global top-k paths within the execution of Algorithm 6. By filtering out paths that we do not want, every path appears at most once. Thus, the global top-k paths can be obtained by selecting the top-k of all kinds of path candidates. □

For the correctness of our algorithms for finding path candidates (Algorithm 5), we have the following lemma which states the property of arrival time tuples.

Lemma 4. *For any circuit pin v whose input edges are $u_1 \to v, u_2 \to v, ..., u_k \to v$, by the definition of $at(v)$ and $at'(v)$, we have*

$$at(v).time = \min_{1 \le i \le k} at(u_i).time + delay^{early}(u_i, v),$$
$$at'(v).time = \min_{1 \le i \le k} at_{auto}(u_i, at(v).groupid).time + delay^{early}(u_i, v).$$

This ensures that we can correctly propagate the two sets of arrival time tuples using tuples on previous pins. The above statements are for hold check. For setup check, one needs to replace $\min$ by $\max$, and early by late.

Proof. The first equation is obvious according to the optimal substructure of shortest path on DAG. For the second one, the right-hand side (RHS) gives a valid solution to the problem defined by left-hand side (LHS), so we must have LHS≤RHS. We assume LHS<RHS and prove by contradiction. Suppose the shortest path given by LHS is from u_i. Then the arrival time of the path at u_i must be smaller than both $at(u_i).time$ and $at'(u_i).time$, and that contradicts the optimality of them. □

This lemma draws the following correctness theorem for our path candidates finding algorithm.

Theorem 2. *Procedure `getPathsAtLCALevel` (Algorithm 5) correctly computes $P_d^{mode}(k)$.*

Proof. According to the way the algorithm assigns the arrival time of launching FFs and capturing FFs, $slack(p, d)$ is added to the slack of path p in both modes. For every circuit pin v, the algorithm maintains two sets of arrival time tuples, $at(v)$ and $at'(v)$, the latter of which serves as a fallback for the former. From these two sets of arrival time, according to Lemma 4, we can always find the shortest path (for hold check, longest path for setup check) to v subject to any node grouping contraint. Thus, the algorithm computes top-1 path candidate at level d correctly.

For the correctness of top-k path finding, we represent each path as a list of deviations from a shortest path. Since by definition all paths can be regarded as a list of deviations from a shortest path, it suffices to show that we find paths in ascending order of their slacks. In other words, each deviation introduces a non-negative increase on the slack of a path (the $cost$ in Algorithm 5 line 16, 18). For any circuit pin u and group index gid,

$$cost^{\text{hold}} = at_{\text{auto}}(w, gid).time + delay^{\text{early}}(w, u) - at_{\text{auto}}(u, gid).time$$

for hold check, and

$$cost^{\text{setup}} = at_{\text{auto}}(u, gid).time - delay^{\text{late}}(w, u) - at_{\text{auto}}(w, gid).time$$

for setup check are non-negative. This is obvious according to Lemma 4 and the definition of at_{auto}. □

Finally, we conclude with the following theorem of overall correctness.

Theorem 3. *The overall Algorithm 1 outputs $P_{CPPR}^{mode}(k)$ correctly.*

Proof. The correctness of procedures `getPathsFromSelfLoops` and `getPathsFromPIs` can be proved similarly to Theorem 2. The correctness follows from Theorem 1. □

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

Theorem 4. *For $k = 1$, Algorithm 1 runs in $O(nD)$ time complexity.*

Proof. The algorithm calls `getPathsAtLCALevel` D times, `getPathsFromSelfLoops` once, and `getPathsFromPIs` once. Each of them consists of a single forward propagation and constant times of enumeration on FFs. Thus, they run in $O(n)$ time. For $k = 1$, the `selectTopPaths` procedure just selects the path with the smallest slack from at most $D + 2$ paths, so it runs in $O(D)$ time. As a result, the overall algorithm runs in $O(nD)$. □

Theorem 5. *For $k > 1$, Algorithm 1 runs in $O(nDk \log k)$ time complexity.*

Proof. In procedures `getPathsAtLCALevel`, `getPathsFromSelfLoops`, and `getPathsFromPIs`, the propagation of arrival time takes $O(n)$. After that, we pop paths from a min-heap for k times. Each time one path is popped from the heap, and we scan for all its deviations and push them into the heap. The count of deviations from a single path cannot exceed the size of the graph, which is n, and thus there are $O(nk)$ heap operations. By using a min-max heap, we are able to limit the size of the heap and always keep the smallest k paths in the heap. In this way,

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

each heap operation takes $O(\log k)$. As a result, each of the three procedures runs in $O(nk \log k)$.

The `selectTopPaths` procedure selects the top-k paths from at most $k(D+2)$ paths, which can be done in $O(kD \log k)$. We conclude that the overall algorithm runs in $O(nDk \log k)$. □

Theorem 6. *The algorithm runs with space complexity $O(T_g(n+k)+kp)$, where T_g denotes the number of thread groups working on independent iterations, and $p < n$ denotes the average length of critical paths.*

Proof. For every call to `getPathsAtLCALevel`, `getPathsFromSelfLoops`, and `getPathsFromPIs`, we need $O(n)$ of memory to store arrival time tuples for circuit pins. We represent each path as a list of deviation edges. Because deviation edges are added one by one, we do not need to store all of them on a single path. Instead, we arrange them in a prefix tree [2], where each path is denoted by a node and each deviation edge is denoted by an edge, and thus we need $O(k)$ memory to store all the paths. Each thread group has its own dedicated memory for working on a call. Thus, the overall memory complexity is $O(T_g(n+k))$. The additional $O(kp)$ of memory is the size of the resulting global top-k 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 also compare our approach with the commercial STA engine PrimeTime.

A. Parallelization Strategy #1

We start by setting $T_w = 1$, i.e. no parallelization within the circuit graph. As a result, $T = T_g$, which means that all threads work on independent iterations. This is our default parallelization strategy which demonstrates the best runtime performance on multi-core CPUs. 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.

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.

Our algorithm with parallelization strategy #1 has a good memory performance when generating a large number of critical paths. For example, we reduce the memory consumption for k=10K by 14.15×, 38.83×, and 9.14× compared to OpenTimer, HappyTimer and iTimerC respectively. 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. As we will show later, by exploiting parallelism within the circuit graph (i.e. setting $T_w > 1$), we can get comparable runtime while largely reduce memory consumption.

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.

B. Parallelization Strategy #2 and Hybrid

We now test the performance when incorporating parallelization strategies within the circuit graph. To conduct a fair comparison, we fix the total number of threads $T = T_w \times T_g = 8$, and test the runtime and memory performance of different (T_w, T_g) pairs. Table V lists the overall performance comparison.

By increasing T_w and decreasing T_g, the runtime becomes longer due to less parallel iterations and more parallelism inside the circuit graph. However, the memory consumption

TABLE IV: Performance comparison between OpenTimer (8 threads), HappyTimer (1 thread), iTimerC (8 threads) and ours [13] (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 [13] 8 Threads		Ours [13] 1 Thread		OpenTimer 8 Threads		HappyTimer 1 Thread		iTimerC 8 Threads		Ours [13] 8 Threads		Ours [13] 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.

TABLE V: Runtime and memory comparison of our algorithm running with different (T_w, T_g) pairs, and OpenTimer with 8 threads. All pairs satisfy $T = T_w \times T_g = 8$. The pair $T_w = 1, T_g = 8$ is the same as the 8-thread strategy [13] in Table IV.

Benchmark	k	OpenTimer 8 Threads		Ours T_w=8, T_g=1		Ours T_w=4, T_g=2		Ours T_w=2, T_g=4		Ours [13] T_w=1, T_g=8		OpenTimer 8 Threads		Ours T_w=8, T_g=1		Ours T_w=4, T_g=2		Ours T_w=2, T_g=4		Ours [13] T_w=1, T_g=8	
		RT	Mem	RT	Mem	RT	Mem	RT	Mem	RT	Mem	RT	Mem	RTR	MemR	RTR	MemR	RTR	MemR	RTR	MemR
vga_lcdv2	1	18.05	0.25	5.99	0.17	4.59	0.20	4.09	0.26	3.90	0.37	4.63	0.68	1.54	0.46	1.18	0.54	1.05	0.70	1.00	1.00
	100	18.19	0.26	6.45	0.17	4.76	0.20	3.67	0.26	3.51	0.37	5.18	0.70	1.84	0.46	1.36	0.54	1.05	0.70	1.00	1.00
	10K	64.42	2.32	8.51	0.18	5.82	0.21	4.71	0.27	4.03	0.39	15.99	5.95	2.11	0.46	1.44	0.54	1.17	0.69	1.00	1.00
Combo4v2	1	37.72	0.46	12.80	0.31	8.98	0.36	7.47	0.46	6.89	0.66	5.47	0.70	1.86	0.47	1.30	0.55	1.08	0.70	1.00	1.00
	100	36.76	0.49	12.56	0.31	8.52	0.36	7.55	0.46	6.30	0.66	5.83	0.74	1.99	0.47	1.35	0.55	1.20	0.70	1.00	1.00
	10K	380.93	15.40	16.88	0.32	10.74	0.37	8.55	0.48	7.28	0.68	52.33	22.65	2.32	0.47	1.48	0.54	1.17	0.71	1.00	1.00
Combo5v2	1	190.98	1.21	27.32	0.81	22.28	0.94	20.23	1.20	19.48	1.72	9.80	0.70	1.40	0.47	1.14	0.55	1.04	0.70	1.00	1.00
	100	191.61	1.25	27.64	0.81	22.41	0.94	20.03	1.20	18.85	1.72	10.16	0.73	1.47	0.47	1.19	0.55	1.06	0.70	1.00	1.00
	10K	831.84	27.34	32.77	0.82	24.28	0.95	21.25	1.22	20.25	1.75	41.08	15.62	1.62	0.47	1.20	0.54	1.05	0.70	1.00	1.00
Combo6v2	1	568.08	2.18	45.63	1.38	37.29	1.61	34.91	2.06	33.70	2.97	16.86	0.73	1.35	0.46	1.11	0.54	1.04	0.69	1.00	1.00
	100	567.58	2.22	45.31	1.38	37.74	1.61	34.62	2.06	32.85	2.97	17.28	0.75	1.38	0.46	1.15	0.54	1.05	0.69	1.00	1.00
	10K	1333.36	31.27	51.10	1.39	40.22	1.62	36.00	2.07	33.85	2.98	39.39	10.49	1.51	0.47	1.19	0.54	1.06	0.69	1.00	1.00
Combo7v2	1	376.45	1.73	36.45	1.09	29.31	1.27	26.77	1.63	25.15	2.34	14.97	0.74	1.45	0.47	1.17	0.54	1.06	0.70	1.00	1.00
	100	382.07	1.78	36.01	1.09	29.62	1.27	27.04	1.63	24.03	2.34	15.90	0.76	1.50	0.47	1.23	0.54	1.13	0.70	1.00	1.00
	10K	1556.68	45.97	42.02	1.10	31.98	1.28	28.18	1.64	25.99	2.36	59.90	19.48	1.62	0.47	1.23	0.54	1.08	0.69	1.00	1.00
netcard	1	976.41	1.79	48.44	1.46	42.13	1.71	39.24	2.22	39.70	3.23	24.59	0.55	1.22	0.45	1.06	0.53	0.99	0.69	1.00	1.00
	100	977.27	1.80	49.99	1.46	42.13	1.71	39.30	2.22	39.80	3.23	24.55	0.56	1.26	0.45	1.06	0.53	0.99	0.69	1.00	1.00
	10K	2749.46	74.06	54.98	1.46	45.44	1.72	40.85	2.23	41.59	3.25	66.11	22.79	1.32	0.45	1.09	0.53	0.98	0.69	1.00	1.00
leon2	1	3131.08	2.12	58.78	1.75	48.25	1.90	42.43	2.45	43.41	3.55	72.13	0.60	1.35	0.49	1.11	0.54	0.98	0.69	1.00	1.00
	100	3131.73	2.13	58.89	1.75	47.76	1.90	43.84	2.45	45.19	3.55	69.30	0.60	1.30	0.49	1.06	0.54	0.97	0.69	1.00	1.00
	10K	4320.00	41.81	64.99	1.75	51.35	1.91	44.78	2.47	44.87	3.56	96.28	11.74	1.45	0.49	1.14	0.54	1.00	0.69	1.00	1.00
leon3mp	1	1017.85	1.63	42.50	1.24	33.89	1.46	31.02	1.88	30.81	2.74	33.04	0.59	1.38	0.45	1.10	0.53	1.01	0.69	1.00	1.00
	100	1013.06	1.63	42.17	1.24	34.40	1.46	31.11	1.88	30.27	2.74	33.47	0.59	1.39	0.45	1.14	0.53	1.03	0.69	1.00	1.00
	10K	1348.55	12.48	47.95	1.25	36.89	1.47	32.52	1.90	31.11	2.75	43.35	4.54	1.54	0.45	1.19	0.53	1.05	0.69	1.00	1.00
Avg. Ratio	1	-										22.69	0.66	1.44	0.47	1.15	0.54	1.03	0.69	1.00	1.00
	100	-										22.71	0.68	1.52	0.47	1.19	0.54	1.06	0.69	1.00	1.00
	10K	-										51.80	14.16	1.69	0.47	1.24	0.54	1.07	0.69	1.00	1.00

RT: Runtime in seconds. **RTR**: Runtime ratio. **Mem**: Memory in GB. **MemR**: Memory ratio. T_g**,** T_w follows our definition in Sect. III.

is reduced because we store less circuit graph instances in memory. For example, strategy $T_w = 2, T_g = 4$ only increases the average runtime by 3%, 6%, and 7% for k=1, 100, 10K respectively, but reduces 30% of the memory consumption in all cases we have tested. In this strategy, we have comparable memory consumption compared to OpenTimer even at small ks. At large ks, we reduce the memory usage by at most 33.02× (`netcard`, k=10K).

Increasing T_w generally reduces the memory usage at the cost of increased runtime. Strategy $T_w = 4, T_g = 2$ increases the average runtime by 15%, 19% and 25% for k=1, 100, and 10K, respectively, but only uses 0.54× memory of our parallelization strategy #1. Despite this runtime overhead, our algorithm is still 41.77×, 109.04×, and 29.41× faster than OpenTimer, HappyTimer, and iTimerC, respectively, on average in generating the top-10K paths (computed by dividing the average runtime ratio of the baselines by the average runtime ratio of this strategy. Specifically, $41.77 = 51.80/1.24$, $109.04 = 135.21/1.24$, and $29.41 = 36.47/1.24$.). Strategy $T_w = 8, T_g = 1$ has the smallest memory consumption, which is nearly the same as our 1-thread version. By parallelizing within the circuit graph, it accelerates the 1-thread version by 1.63×, 1.60×, and 1.51× for k=1, 100, and 10K respectively. However, it introduces 44%–69% runtime overhead compared to parallelizing across iterations because of limited parallelism within the circuit graph.

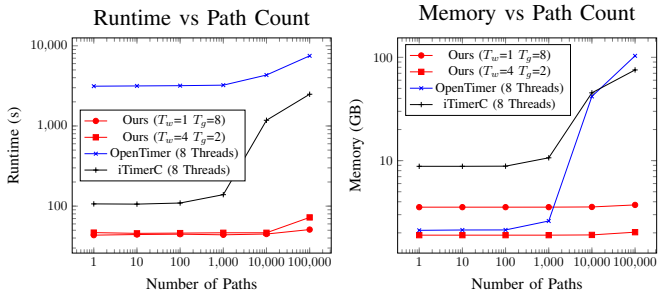


Fig. 8: Runtime and memory values at different numbers of post-CPPR paths (i.e., k) on `leon2`.

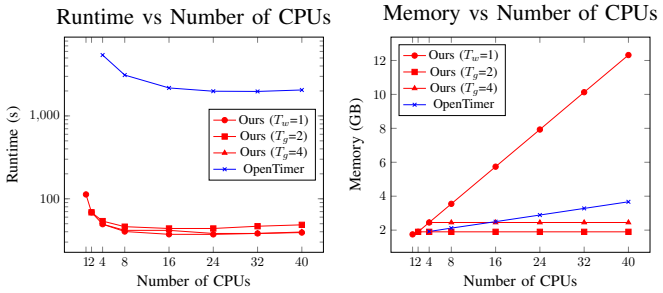


Fig. 9: Runtime and memory values at different numbers of threads for k=1 on `leon2`. OpenTimer with 1 and 2 threads failed to finish within 3 hours, so we skip those two points.

C. Performance at Different Path Counts

Figure 8 draws the runtime and memory consumption versus k, the number of post-CPPR critical paths requested. We tested two strategies of our algorithms, one with $T_w = 1, T_g = 8$ [13] and another with $T_w = 4, T_g = 2$. 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. For our algorithm, the memory consumption of strategy $T_w = 4, T_g = 2$ is smaller compared to strategy $T_w = 1, T_g = 8$ across all numbers of paths, and there is a remarkable gap. Meanwhile, these two strategies have similar runtime except for k =100K. This shows that additional parallelization within circuit graph gives smaller memory consumption only at the cost of a slight runtime overhead.

D. Performance at Different Thread Counts

Figure 9 and Figure 10 draw the runtime and memory consumption at different numbers of threads. To show the runtime and memory feature of our algorithm under different parallelization strategies, we have tested three different ways of thread allocation between T_g and T_w. For the first strategy, we fix $T_w = 1$ (which is the same strategy as [13]) to demonstrate the parallelism across clock tree depths. For the other two strategies, we fix T_g to 2 and 4 respectively to demonstrate the parallelism within the circuit graph. We do not show iTimerC because its binary is hardcoded for 8

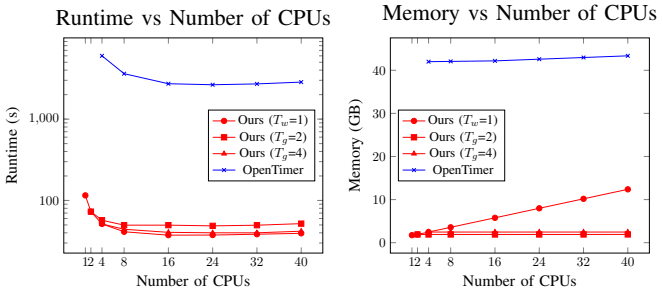


Fig. 10: Runtime and memory values at different numbers of threads for k=10K on `leon2`. OpenTimer with 1 and 2 threads failed to finish within 3 hours, so we skip those two points.

threads. The result shows that our algorithm has an outstanding performance on runtime. For k=10K, our algorithm uses only a small portion of the memory than OpenTimer, while being significantly faster. We also observe that our algorithm is scalable to different numbers of threads in all of the three strategies. For $T_w = 1$ and $k = 1$, it uses more memory than OpenTimer at a larger thread count. The reason is that we parallelize our algorithm across clock tree depths (setting $T = T_g$), thus replicating the circuit graph in many threads. This is the same way OpenTimer adopted to leverage multi-core CPU power, while we have a slightly larger constant behind our space complexity, in which we use extra arrival time tuples to keep track of paths. However, by fixing $T_g = 2$ or 4 instead of T_w, we achieve both low runtime and low memory than OpenTimer, even when $k = 1$. Furthermore, by fixing T_g instead of T_w, our algorithm uses a fixed amount of memory regardless of the number of threads used.

E. Comparison with Commercial Tool

In this section, we provide a detailed comparison between our proposed algorithm and Synopsys PrimeTime. We conduct the test in this section on a platform with 12 cores Intel Xeon CPU at 2.60 GHz and 64 GB memory. We have to use this platform instead of the one used in previous sections, because our PrimeTime 2018.3 license is tied to this hardware. Notice that the comparison between our algorithm and PrimeTime may be unfair 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. Despite this, we have made several efforts to make this comparison as fair as possible:

1) We write a simple program to transform the data format in TAU contests (i.e. delay-annotated timing graph) into format that PrimeTime can read directly (i.e. verilog source code, cell library, and design constraints). This helps us run PrimeTime on the same set of benchmarks as we use in the previous sections.
2) We enable CPPR in PrimeTime reports by setting `remove_clock_reconvergence_pessimism` to `true`, and `crpr_threshold_ps` to 0. With these

TABLE VI: Runtime and memory comparison with the commercial STA engine PrimeTime.

Benchmark	k	PrimeTime 8 Threads		Ours 8 Threads		PrimeTime/Ours Ratio	
		RT	Mem	RT	Mem	RTR	MemR
vga_lcdv2	1	44	1511.72	2.82	377.92	15.72	4.00
	100	46	1509.97	2.87	378.99	16.01	3.98
	10K	80	1689.95	3.15	397.80	25.50	4.25
Combo4v2	1	79	1696.06	4.91	671.24	16.10	2.53
	100	80	1692.12	4.96	671.95	16.20	2.52
	10K	238	2588.26	5.51	699.75	43.13	3.70
Combo5v2	1	196	2139.38	13.93	1750.21	14.07	1.22
	100	197	2139.38	14.61	1749.76	13.48	1.22
	10K	279	2888.70	14.50	1775.71	19.24	1.63
Combo6v2	1	358	2959.11	22.57	3043.11	15.85	0.97
	100	358	2959.20	24.02	3042.95	14.90	0.97
	10K	412	3054.88	23.91	3059.71	17.25	1.00
Combo7v2	1	282	2571.66	18.86	2403.86	14.97	1.07
	100	288	2577.47	18.27	2404.62	15.74	1.07
	10K	402	3445.32	19.16	2424.03	20.96	1.42
netcard	1	412	3516.32	26.08	3281.10	15.81	1.07
	100	407	3558.82	25.42	3282.06	16.01	1.08
	10K	478	4190.51	25.91	3300.40	18.45	1.27
leon2	1	889	4892.45	29.14	3590.14	30.51	1.36
	100	888	4911.28	29.09	3593.77	30.54	1.37
	10K	1148	5615.83	29.71	3610.15	38.63	1.56
leon3mp	1	533	3197.52	21.23	2808.18	25.09	1.14
	100	531	3199.67	21.11	2808.02	25.17	1.14
	10K	731	4240.16	22.35	2824.72	32.72	1.50
Avg. Ratio	1	-				18.52	1.67
	100	-				18.51	1.67
	10K	-				26.99	2.04

RT: Runtime in seconds. **RTR**: Runtime ratio.
Mem: Memory in MB. **MemR**: Memory ratio.

settings, PrimeTime will rank paths based on precise post-CPPR slacks without speed-accuracy tradeoff.

3) We provide delay annotations directly to PrimeTime (in SDF format) to bypass its built-in delay modeling of cell arcs and net arcs. In this way, PrimeTime concentrates on solving the CPPR path extraction problem.

Table VI shows the overall runtime and memory results. We run our proposed algorithm with parallel strategy #1 (i.e. $T_w = 1, T_g = 8$). Notice that we have re-run the experiments of our algorithm on the new platform for a fair comparison with PrimeTime. We observe an average runtime speedup of 18.52×, 18.51×, and 26.99× for k=1,100,10K respectively. Across all benchmarks we have tested, we are at least 13.48× faster than PrimeTime. Our algorithm also has efficient memory performance, using only 60% the memory for k=1,100, and 50% the memory for k=10K on average. A notable case with largest runtime speedup is `Combo4v2` with k=10K. In this case, we are 43.13× faster, while only using 27% of the memory compared to PrimeTime. The runtime and memory gap becomes larger with larger k. These results demonstrate the advantage of our proposed algorithm compared to an industry-standard commercial STA engine.

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, and we can balance the runtime and memory consumption by changing the allocation of threads. By performing parallel iterations over different, independent LCA depths and the nodes in the same level when propagating the arrival time tuples, 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 [14], [15], [16] in our future work, and integrate our timer into timing-driven design optimization tasks such as placement [17], [18], [19] and routing. Meanwhile, incremental CPPR analysis remains a challenging problem due to lack of pruning techniques for post-CPPR path-based analysis, which we plan to investigate in the future.

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] T.-W. Huang and M. D. Wong, "UI-timer 1.0: An ultrafast path-based timing analysis algorithm for CPPR," *IEEE TCAD*, vol. 35, no. 11, pp. 1862–1875, 2016.

[8] 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.

[9] T. Chung-Hao and M. Wai-Kei, "A fast parallel approach for common path pessimism removal," in *Proc. ASPDAC*, 2015, pp. 372–377.

[10] D. Eppstein, "Finding the k shortest paths," *SIAM J. Comput.*, vol. 28, no. 2, p. 652–673, 1999.

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

[12] T. W. Huang, Y. Lin, C. X. Lin, G. Guo, and M. D. F. Wong, "Cpp-taskflow: A general-purpose parallel task programming system at scale," *IEEE TCAD*, pp. 1–1, 2020.

[13] Z. Guo, T.-W. Huang, and Y. Lin, "A provably good and practically efficient algorithm for common path pessimism removal in large designs," in *Proc. DAC*. ACM, 2021.

[14] G. Guo, T.-W. Huang, Y. Lin, and M. Wong, "Gpu-accelerated critical path generation with path constraints," in *Proc. ICCAD*. ACM, 2021.

[15] Z. Guo, T.-W. Huang, and Y. Lin, "Gpu-accelerated static timing analysis," in *Proc. ICCAD*. ACM, 2020.

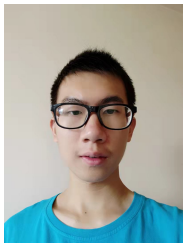
[16] ——, "Heterocppr: Accelerating common path pessimism removal with heterogeneous cpu-gpu parallelism," in *Proc. ICCAD*. ACM, 2021.

[17] Y. Meng, W. Li, Y. Lin, and D. Z. Pan, "elfplace: Electrostatics-based placement for large-scale heterogeneous fpgas," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2021.

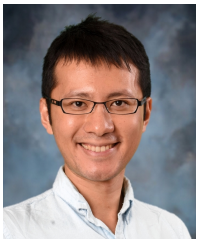
[18] Y. Lin, Z. Jiang, J. Gu, W. Li, S. Dhar, H. Ren, B. Khailany, and D. Z. Pan, "DREAMPlace: Deep learning toolkit-enabled gpu acceleration for modern vlsi placement," *IEEE TCAD*, 2020.

[19] Y. Lin, W. Li, J. Gu, H. Ren, B. Khailany, and D. Z. Pan, "ABCD-Place: Accelerated batch-based concurrent detailed placement on multi-threaded cpus and gpus," *IEEE TCAD*, 2020.

Zizheng Guo is currently pursuing his undergraduate degree in computer science at Peking University, with the Center for Energy-Efficient Computing and Applications. His research interests include data structures, algorithm design and GPU acceleration for combinatorial optimization problems. He is currently working on static timing analysis in VLSI CAD. He has received National Scholarship in 2020 and 2021, Peking University Exceptional Award for Academic Innovation in 2020, and POSCO Asia Fellowship in 2019.



Mingwei Yang is currently an undergraduate student in the Computer Science Department associated with the Center for Energy-Efficient Computing and Applications at Peking University, China. His research interests include algorithm design and GPU acceleration.



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 obtained his Ph.D. degree in the Electrical and Computer Engineering (ECE) Department at the University of Illinois at Urbana-Champaign (UIUC). He is currently an assistant professor in the ECE department at the University of Utah. Dr. Huang has been building software systems for parallel computing and timing analysis. His PhD thesis won the prestigious 2019 ACM SIGDA Outstanding PhD Dissertation Award for his contributions to distributed and parallel VLSI timing analysis.

Yibo Lin (S'16–M'19) received the B.S. degree in microelectronics from Shanghai Jiaotong University in 2013, and his Ph.D. degree from the Electrical and Computer Engineering Department of the University of Texas at Austin in 2018. He is current an assistant professor in the Computer Science Department associated with the Center for Energy-Efficient Computing and Applications at Peking University, China. His research interests include physical design, machine learning applications, GPU acceleration, and hardware security.