An Efficient Critical Path Generation Algorithm Considering Extensive Path Constraints

Guannan Guo*, Tsung-Wei Huang†, Chun-Xun Lin*, and Martin Wong*‡
*Department of Electrical and Computer Engineering, University of Illinois at Urbana-Champaign, IL, USA
†Department of Electrical and Computer Engineering, University of Utah, Salt Lake City, UT, USA
‡Department of Computer Science and Engineering, The Chinese University of Hong Kong, Shatin, NT, Hong Kong

***Abstract*—In this paper, we introduce a fast and efficient critical path generation algorithm considering extensive path constraints on a Static Timing Analysis (STA) graph. Critical path generation is a key routine in the inner loop of path-based analysis and timing-driven synthesis flows. Our algorithm can report arbitrary numbers of critical paths on a logic cone constrained by a sequence of from/through/to pins under different min/max modes and rise/fall transitions. Our algorithm is general, efficient, and provably good. Experimental results have showed that our algorithm produces reports that matches a golden reference generated by an industrial signoff timer. Our results also correlate to a commercial timer yet achieving more than an order of magnitude speed-up.**

I. Introduction

Static timing analysis (STA) plays an important role in many timing-driven optimization flows including logic synthesis, placement, and routing [1]. Optimization algorithms call a timer in their inner loops to evaluate the impact of a design change or transform on the timing profile. One of such key routines is the `report_timing` command, which reports a set of critical paths subject to a sequence of path constraints. It is well known during the process of generating path reports, the core timing model is produced only once and the `report_timing` call happens subsequently and sequentially to meet users need [2]. This process becomes important especially on advanced technologies, where timing signoff has shifted from graph-based analysis (GBA) to path-based analysis (PBA) to reduce the pessimism [3]. As a result, the recent TAU contest was seeking novel ideas to tackle the problem of efficient generation of timing reports from an STA graph [2].

However, generating timing reports is a computationally intensive step and is inherently complex since there are extensive constraints and dependencies of one timing path on another. For example, users may request a path to go through certain pins, registers, and logic cones; they may also specify the checks to occur at a particular timing split (min/max) or transition (rise/fall). Figure 1 gives an example command that requests the top-2 critical paths through pins Inst1/Zn and Inst3/Zn in order. A typical user flow in an optimization loop has thousands of such commands each with different path numbers and constraints. It is imperative for the timing tool to report paths efficiently as this will enable faster design closure and reasonable turnaround time [3].

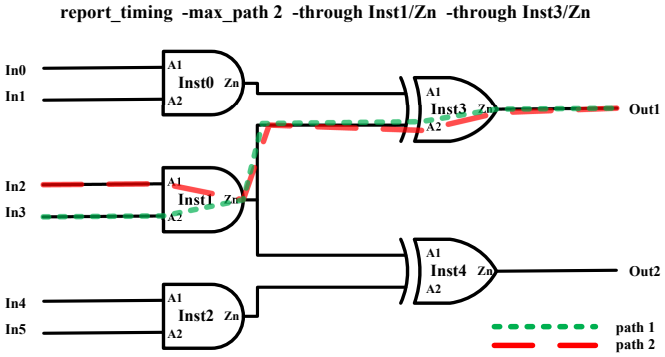

Fig. 1: An example of a `report_timing` query.

As a consequence, we introduce in this paper an efficient timing critical path generation algorithm on an STA graph with updated arrival times and required arrival times. Specifically, we propose efficient data structures, algorithms, and parallelization strategies to tackle the `report_timing` command that incorporates extensive path constraints. We summarize our contributions as follows:

- **A general critical path generation algorithm**. Our algorithm can handle extensive and important path constraints such as through pins, min/max splits, rise/fall transitions, and arbitrary path count limits. These realistic constraints span a vast majority of spectrum in practical usage. Our algorithm is general and flexible, adding only linear time and space complexity to an existing timer infrastructure. The result can benefit designs of PBA flows and timing-driving optimizations where critical path generation plays a key role in the inner loop.
- **A graph-theoretic framework**. Our algorithm maps the path generation problem into a graph formulation. We introduced an effective node pruning and edge filtering strategy to remove unwanted paths. Our solution is *exact* and does not compromise any accuracy with speed.
- **Parallelization benefits**. Our algorithm is highly parallelizable. The proposed path search strategy can be easily partitioned to a set of independent work across effective endpoints, allowing us to take advantage of multithreading to speed up the search process with strong scalability.

We have integrated our algorithm in an open-source STA tool, OpenTimer [4], and evaluated our algorithms on the TAU

978-1-7281-1085-1/20/$31.00 ©2020 IEEE

2018 Timing Analysis Contest benchmarks with a golden reference generated by an industrial standard timer [2]. Our path report can exactly match the reference report according to the contest evaluation metrics. We have also compared our results with a commercial timer, OpenSTA (developed by Parallax Software Inc [5], [6]), and demonstrated a strong correlation in path slacks. In terms of performance, our algorithm can achieve more than an order of magnitude speed-up on large designs. We believe the proposed algorithm can help accelerate the design closure in timing-driven optimization flows.

II. Timing Report Generation

It is well known to those who understand STA that core timing graph model is generated only once and timing report generation happens subsequently and sequentially as there are many different forms of reports that users need [2]. In this section, we discuss the typical STA timing graph model and highlight existing works.

A. STA Timing Graph Model

In STA, the circuit design is modeled as a *directed acyclic graph* (DAG) $G = \{V, E\}$, where a vertex represents a pin in the design and an edge denotes a timing arc between two vertices. Each pin $p \in V$ is annotated with two possible transition values, rise (v_p^{rise}) or fall (v_p^{fall}). Each edge $e_{u \to v} \in E$ is a directed arc from vertex u to v. A path is an ordered sequence of vertices $\langle v_1, v_2, \cdots, v_n \rangle$ or a sequence of edges $\langle e_1, e_2, \cdots, e_m \rangle$. *Slack* measures the amount of timing violation. It is defined as the difference between signal arrival time and required arrival time. Critical paths are paths with worst negative slacks. Given an STA graph, one of the main goals of STA is to report the top-k critical paths under user-specified path constraints, or the so-called `report_timing` command [1], [2].

B. Related Works

Generating the top-k critical paths is an essential routine in many STA tools [4], [7], [8], [9]. Developers often report a set of critical paths according to their slack values and measure the timing properties of each path. The algorithm to rank k critical paths, especially when k is more than one, under extensive path constraints such as which path goes through which pins under which transitions and so on, is an extremely difficult challenge. As a result, the recent TAU Timing Analysis Contest settled down on this topic looking for novel ideas and algorithms for path generation [2]. [9] proposed a k-shortest path algorithm that targeted on common path pessimism removal (CPPR) with polynomial time and space complexity. Yet, it did not consider any path constraints. [8] proposed a branch-and-bound algorithm to explore critical paths under given timing exceptions. Depending on the initial points, the algorithm may consume an exponential growth of search space before reaching an effective pruning threshold. The dependency on pruning points also makes it difficult to take advantage of multithreading. Inspired by the 2018 TAU contest environment, [10] proposed a cache-based framework to capture the similarity between path constraints of successive queries. However, the algorithm is not exact and needs to trade in accuracy for speed. Neither can the cache scheme produce consistent speed-up at all situations.

C. k-Critical Path Generation Algorithm

The state-of-the-art algorithm is a variant of the top-k shortest path algorithm using implicit path representation and fast path recovery strategy to generate timing critical paths from an STA graph [9]. Given an integer k, a destination node d, a set of source nodes $S = \{s_1, s_2, s_3, \cdots\}$ and their corresponding distance offset, the algorithm outputs the top-k shortest paths from a source in S to d. It uses a suffix and a prefix tree data structures to represent a path implicitly. A *suffix tree* is defined as the shortest path tree $T_d = \{V_T, E_T\}$ rooted at d. A *prefix tree* is constructed with edges not in the suffix tree, which are called deviation edges. Each node in the prefix tree represents a deviation edge $e^{pre} \in E/E_T$. By backtracing to the root of prefix tree, a set of deviation edges can be collected. A full path enumeration can be recovered by complementing this set with edges in the suffix tree. A single node in the prefix tree implicitly represents a full path in G. The prefix tree will maintain a min-heap for all the leaf nodes. Each time a leaf node l_e is popped from the heap, new leaf nodes on l_e will be spawned. By traversing the path recovered from l_e, all the deviation edges along this path will be the new children of l_e. Given that $|V| = n, |E| = m$, the time complexity of this algorithm is $O(nlogn + kn + kmlogk)$ and space complexity is $O(nlogn + m + k)$. More algorithm details can be referred to [9]. While we shall develop our algorithm on top of this tree structure, the proposed concept is applicable to other frameworks that operate on an STA graph.

III. The Proposed Algorithm

An overview of our algorithm is shown in Figure 2. It consists of multiple steps: (1) We use OpenTimer to construct the STA graph given standard input files including *Verilog* (gate-level netlist of the design), *liberty* (gate delay and tests specifications), *spef* (parasitic RC), and simple *SDC* or *Timing* (design constraints) . We then call `update_timing` to perform graph-based analysis. (2) Queries are processed one at a time. A `PathBuilder` class is constructed to hold information from each `report_timing` query. (3) The topological sorting algorithm is launched from the last pin in the query. A node to topological rank mapping is built up. (4) The pruning step collects nodes in subgraph restricted by the query. In the meantime, we complete a node to level mapping. (5) Suffix and prefix trees mentioned in Section II-C are constructed based on the list of post-pruning nodes. Both the rank and level mappings are used to filter out arcs that violate the query constraints.

Details of graph-based analysis completed by OpenTimer are explained in [4]. Our algorithm executes the requested path-based analysis. Starting from the second step, for each query in .ops file, we parse it and store its arguments into a

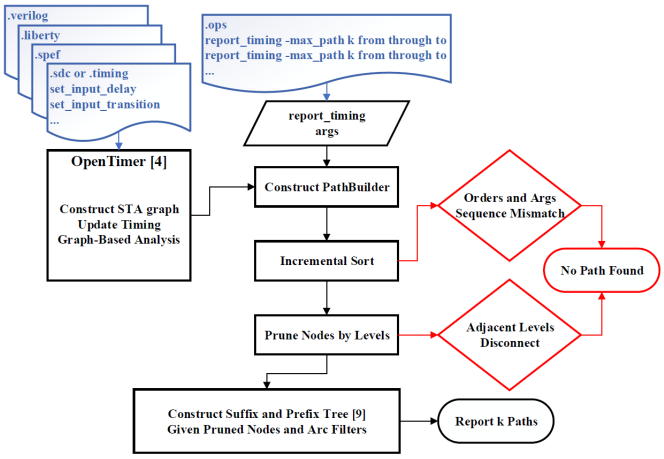

Fig. 2: Overview of our algorithm.

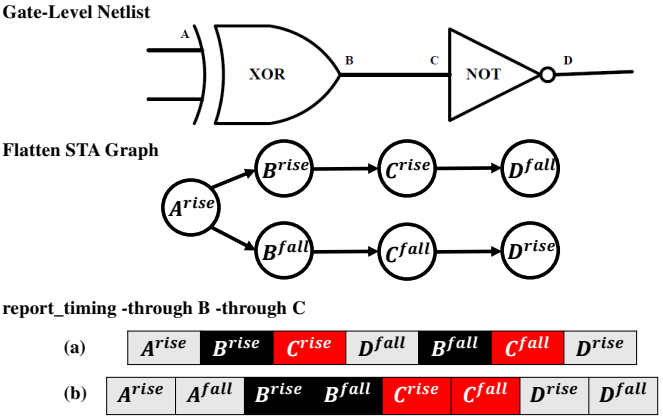

Fig. 3: (a) Pin B and pin C have interleaving ranks. (b) Ranks of pin B are disjoint from ranks of pin C.

`PathBuilder` class. It contains the pin sequence information denoted as $\texttt{Thru} = \langle p_1, p_2, \cdots p_{last} \rangle$. We use $p_i.\texttt{tran}$ to represent the required transition on pin p_i. This field can be empty if user does not require a specific transition.

A. Incremental Topological Ranking

This step preprocesses the STA subgraph and establishes a node to topological rank mapping while it preserves the circuit property. Let G^* be the fan-in subgraph of last pin p_{last}. We perform topological sort algorithm on p_{last} with the following **rank assigning rule**:

$$rank(v_p^{fall}) = rank(v_p^{rise}) + 1, \forall p$$

Two nodes corresponding to different transitions of the same pin are assigned with adjacent ranks. The topological ranks obtained under this rule is correct because there should be no path between v_p^{rise} and v_p^{fall} in an STA graph generated from a valid circuit design. Figure 3 illustrates the sorting results (a) without rank assigning rule and (b) with rank assigning rule enforced. In this way, ranks of required pins in `Thru` are sufficient to partition the vertex space of G^*, which enable us to filter fan-in and fan-out arcs without enumerating all possible transitions over required pins. Otherwise, the enumeration is exponential to the number of pins in `Thru`. Besides, subsequent sorted ranks from later queries are based on offset from previous ranks, which makes this step incremental.

B. Partition and Node Pruning

Main objectives of this step are: (1) collect nodes from subgraph G^* for suffix tree construction; (2) build vertex to level mapping for arc filtering. The order of required pins in `Thru` list should agree with their ranks from low to high after the sort. The increasing order of these ranks should be unique otherwise a cycle exists in DAG or no path exists between some required pins. For each pair of adjacent pins $\langle p_i, p_{i+1} \rangle$ in `Thru`, node set of p_i is denoted as *floor* F_i and node set of p_{i+1} denoted as *ceiling* C_i. By this definition, the current *ceiling* is the next *floor* $C_i = F_{i+1}$. Elements of each set is dependent on specified transition on associated pin.

$$\forall i \geq 1, F_i = C_{i-1} = \begin{cases} \{v_{p_i}^T\} & \text{if } p_i.\texttt{tran} = T \\ \{v_{p_i}^{rise}, v_{p_i}^{fall}\} & \text{otherwise} \end{cases}$$

Denote the intersected subgraph of fan-out cone of F_i and fan-in cone of C_i as level $i, \forall i \geq 1$. The level 0 is formed by the first *ceiling* C_0 or F_1 along with source nodes of the timing graph. The level index i can be used for constant time look-up for *floor* F_i and *ceiling* C_i. Every node in the level i has rank value bounded by ranks of F_i and C_i. Nodes not belonging to any level should be eliminated or pruned from the search space. To facilitate this, we define the following **arc filtering rule**: For each node u in level i,

- For each fan-in arc $e_{u \leftarrow v}$, node v either is a node in the *floor* $v \in F_i$ or satisfies $rank(v) > rank(v_{p_i}^{fall})$. Fan-in arcs of $u \in F_i$ are asserted as arcs of previous *ceiling* $u \in C_{i-1}$.
- For each fan-out arc $e_{u \rightarrow v}$, node v either is a node in the *ceiling* $v \in C_i$ or satisfies $rank(v) < rank(v_{p_{i+1}}^{rise})$. Fan-out arcs of $u \in C_i$ are asserted as arcs of next *floor* $u \in F_{i+1}$.

Algorithm 1: node_pruning

```
/* list of post-pruning nodes                    */
Output: Nodes
Result: node to level mapping, endpoints in tail
Nodes ← empty list;
for level ← 0 to Thru.size do
    if level > 0 then
        Forward filtered BFS from floor to ceiling:
        1: mark all visited pins
    end
    Backward filtered DFS from ceiling to floor:
    1. append marked nodes to the list Nodes
    2. all nodes appended have their index mapped to level
end
Forwards BFS from last required pin:
1. mark visited pins as in the tail area
2. collect encountered endpoints
return Nodes
```

We perform forward filtered BFS from each *floor* and backward filtered DFS from associated *ceiling* to further con-

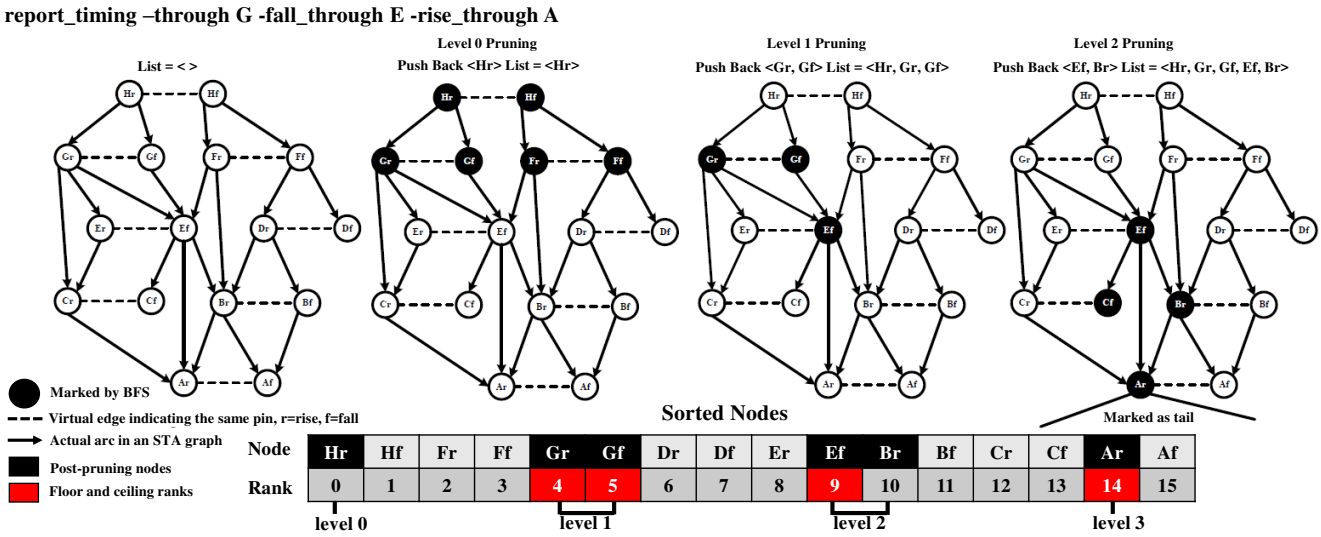

Fig. 4: Exemplification of node pruning. The leftmost graph represents the STA subgraph G_A^* that is reachable to pin A. Nodes in the table are arranged in increasing ranks. For each level, forward BFS (except level 0) and backward recursive DFS are performed to prune nodes in individual level. The search space of these methods are constrained by arc filtering (rank comparison). A final BFS is performed from A^{rise} to mark the unconstrained tail section in case pin A is not a datapath endpoint.

dense the node space and obtain a list of post-pruning nodes. Algorithm 1 outlines the pruning method. A list of timing graph nodes is initialized (line 2). Only nodes connected to both *floor* and *ceiling* of each level are appended to the list (line 3:11). BFS (line 5) and DFS (line 8) are both under the filtering rule described in this section. The entire output list obeys topological order because the list is built with increasing level and DFS preserves orders in each level. In the meanwhile, the node to level mapping is constructed (line 10) so that the level index can be looked up in constant time. Figure 4 illustrates an exemplification of this pruning process on a sorted subgraph.

C. Arc Filtering

The previous step reduces search space of nodes during suffix tree construction. The big picture of this stage is to refine search space for edges. For each endpoint marked in the tail area, we perform top-k critical paths generation algorithm described in Section II-C. The construction of single destination suffix tree (shortest path tree) consists of two steps, topological sort and distances relaxation. Previous step has already collected a list of post-pruning nodes in sorted order up to the last pin in `Thru` sequence. The suffix tree's constructor will fetch this list and append it with searched list in the tail area. In the distance relaxation step, whenever a fan-in arc is traversed, the filtering rule in Section III-B is applied. Since the node to level mapping is built, we can acquire ranks of *floor* and *ceiling* in constant time. Similarly, we apply the filtering rule when the prefix tree searches fan-out arcs for deviation edges. Therefore, constant overhead will be added upon traversal for each arc.

Theorem *Our top-k critical path generation algorithm under general user constraints has time complexity $O(nlogn+kn+kmlogk)$ and space complexity $O(nlogn+m+k)$.*

Proof: The preprocessing steps in Section III-A and III-B takes linear time with respect to the graph size $O(m+n)$. Filtering arc in Section III-C costs $O(1)$ for each arc. Both mappings occupy linear space $O(m+n)$ as well. Therefore, the overall time and space complexity remain $O(nlogn+kn+kmlogk)$ and $O(nlogn+m+k)$ as [9]. In other words, our algorithm adds only linear complexity to an existing framework.

IV. Multithreading

Our algorithm is advantageous in effective separation of path generation from each endpoint. Multiple datapath endpoints may reside in the marked tail section. Top-k critical paths can be generated independently from different endpoints. We apply Cpp-taskflow [11], a modern C++ task-based parallel programming model, to implement this parallelization. Per-endpoint report generation is encapsulated in a task. All tasks merge their k worst paths into a single min-max heap that tracks the globally k worst paths. The critical paths remain in the heap are reported as the final result. A key difference between our task-based method from existing loop-based strategies is dynamic load balancing. By using task parallelism, we can flow our computations naturally with the timing graph structure, rather than level-by-level synchronization [4].

V. Experimental Results

We conducted all experiments on a 3.20 GHz 4-core 64-bit Linux machine with 32 GB memory. We built our algorithm on top of OpenTimer [4] and compiled the entire program in g++-7.3.0 with optimization flag `O2` and C++17 standards `-std=c++17` enabled. We started with the experiments on

the TAU 2018 benchmarks (leon3mp and vga_lcd) with a golden reference generated by a commercial signoff tool to verify the correctness of our algorithm. However, the contest problem is a simplified version of our goal. It assumes each query to have (1) a fixed startpoint (2) a fixed endpoint (3) a fixed transition for each through pin, and (4) a single critical path. To demonstrate the capability of our algorithm, we conducted another experiment that modified leon3mp and vga_lcd query files (.ops) to include more realistic and complicated constraints such as arbitrary through pins and path count limit. Besides, we supplied our experiments with two more large combinational circuits c5315 and c7552 from the TAU 2015 benchmarks. In both experiments, we consider the open-source tool OpenSTA by Parallax Software Inc as baseline [5], [6]. We do not compare with other proprietary timers of closed source that prevent us from analyzing the algorithms. OpenSTA is a mature commercial STA tool with a rich set of features. Over the past years, OpenSTA has conducted numerous correlation experiments with many industrial standard signoff timers. We are interested in the performance and slack correlation between our algorithm and OpenSTA. Through tracing the source of OpenSTA, we found OpenSTA adopted a branch-and-bound algorithm similar to [8] in dealing with path constraints during the path generation. The code also embedded a number of heuristics to prune the search space. For fair comparison, we measured runtime only on the report_timing command. In order to highlight the effectiveness of our algorithms, we tested that both tools produced similar runtime performance without path constraints.

A. Results on TAU 2018 Contest Benchmarks

TABLE I: Comparison between our algorithm and OpenSTA on TAU 2018 Timing Analysis Contest Benchmarks [2]

Circuit	#Gates	#Arcs	#Q/#M	$\mathbf{cpu_{ours}}$	$\mathbf{cpu_{osta}}$
vga_lcd	139K	757K	46352/46352	0.25	148
leon3mp	1.25M	6.28M	49987/49987	2.48	980

#Q: number of `report_timing` queries
#M: number of paths matching the golden reference
cpu: runtime in minutes

Results of our algorithm on the TAU 2018 benchmarks are shown in Table I. Keep in mind the contest simplified the problem and each query requested only one path ($k = 1$). We used **#M** to indicate the number of matched paths from our report to the golden reference. Two paths are matched if and only if they have the same sequence of pins and transitions. As shown in Table I, our algorithm can produce correct path reports that match the golden reference in 100%. We can accomplish runtime within 3 minutes for both benchmarks. The contest winner also achieved similar values. We omit the comparison with the contest top performers because they were tied to the simplified problem and could not handle generic constraints. We are also interested in the performance difference between our algorithm and OpenSTA. As shown in last two columns of Table I, our algorithm is more than an order of magnitude faster than OpenSTA. We can reach the goal by up to 592× and 395× faster for vga_lcd and leon3mp, respectively. This is because our algorithm can precisely prune large search space using the circuit graph property, instead of branching multiple search paths to find the pruning threshold.

B. Results on Complex Benchmarks

In this experiment, we consider more realistic and complex query patterns. We modified the query file (.ops) of the TAU 2018 contest benchmarks. Each query in the original vga_lcd benchmark defines a subgraph that contains only a limited number of paths. We removed the `-from` pin and `-to` pin restrictions to enlarge the number of paths in the subgraph to make the search space much more complex. Combinational circuits c5315 and c7752 are benchmarks from the TAU 2015 contest. We randomly generated a set of `report_timing` queries for each design. Each query is in the most general format, since it requests higher path count limit without any restrictions on pin transitions, path startpoint, and endpoint. Table II shows the statistics of each benchmark.

We evaluated the correlation between our algorithm and OpenSTA as follows: (1) We obtained the slack histogram by accumulating path numbers across different ranges of slack values (see Figure 5). (2) We extracted the slack vectors from both path reports and computed the correlation coefficient. As shown in Figure 5 and Table II, our algorithm has a strong linear correlation with OpenSTA. It can be observed in top row of Figure 5 both our algorithm and OpenSTA produced exactly the same output. The correlation coefficient of both circuits equals one. The purpose of positive slack in leon3mp is to examine the generality of algorithms under arbitrary slack ranges. The bottom half of the figure shows a bit shifted slack values, which is caused by the different numerical methods on interconnect delay model between OpenTimer and OpenSTA. However, both path traces match exactly. In terms of performance, our algorithm is faster than OpenSTA in all benchmarks. The speed-up values over OpenSTA are 10.41×, 69.96×, 5.52×, and 7.84× repectively.

C. Scalability with Multithreading

In this section, we evaluated the multithreading performance of our algorithm on the modified vga_lcd benchmark. There are multiple endpoints involved in each query in this benchmark. We ran OpenSTA on the same testcase. Figure 6 demonstrates that our algorithm efficiently parallelizes independent workload across all endpoints. In general, both algorithms scale with increasing number of CPU cores. However, the performance margin between ours and OpenSTA is considerably large at all CPU numbers. For example, under 1, 2, 3, and 4 CPU cores, OpenSTA took 37.33, 28.77, 24, and 22.26 minutes to finish; ours are only 3.5, 3, 2.85, and 2.75 minutes. A primary reason to this is our task-based parallelization method with dynamic load balancing which avoids excessive waste of thread resources especially for large timing graphs.

TABLE II: Benchmark statistics and performance comparison between the proposed algorithm and OpenSTA

Circuit	#Pins	#Gates	#Arcs	#Queries	k	$\mathbf{cpu_{ours}}$	$\mathbf{cpu_{osta}}$	CC Mean	CC STD
vga_lcd	397809	139529	756631	4000	16	217.59	2266.06	1.0000	3.2150×10^{-7}
leon3mp	3376832	1247725	6277562	4000	8	58.30	4078.43	1.0000	1.6921×10^{-5}
c5315	3245	918	6201	6782	32	18.55	102.47	0.9852	2.3886×10^{-2}
c7552	3802	1147	7133	4401	32	9.27	72.71	0.9863	2.6426×10^{-2}

CC: correlation coefficient k: number of paths generated **cpu**: runtime in seconds

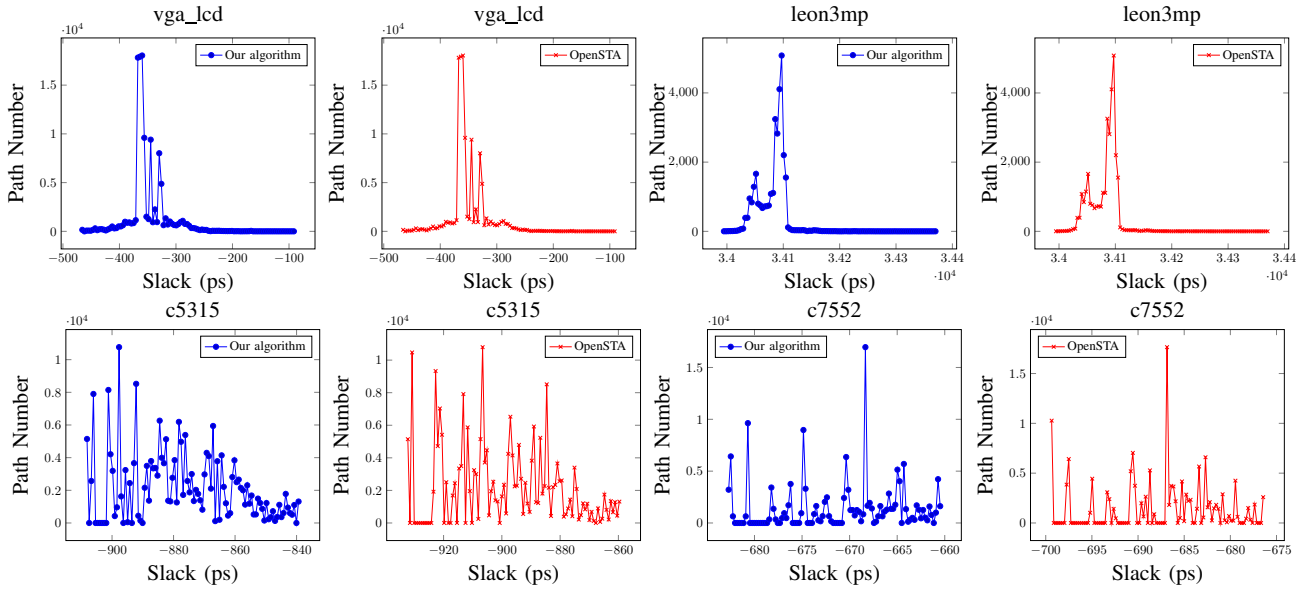

Fig. 5: Comparison of slack distribution between our algorithm and OpenSTA on vga_lcd, leon3mp, c5315, and c7552.

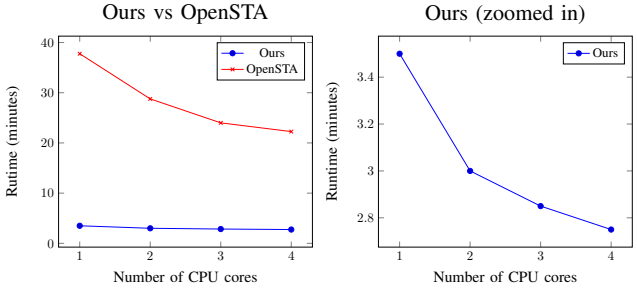

Fig. 6: Multithreading performance on vga_lcd.

VI. Conclusion

In this paper, we have introduced an efficient timing critical path generation algorithm on an STA graph. Our algorithm can quickly search the critical paths under extensive path constraints to meet users need. We have integrated our algorithm into an open-source static timing analysis tool OpenTimer and evaluated it on the 2018 TAU contest benchmarks. Our results match the golden reference provided by the contest. Compared with the open-source commercial tool OpenSTA, our results showed a strong correlation in path slacks, yet achieving more than an order of magnitude speed-up.

VII. Acknowledgement

This work is partially supported by NSF Grant CCF-1718883.

References

[1] J. Bhasker *et al.*, *Static Timing Analysis for Nanometer Designs: A Practical Approach*. Springer, 2009.
[2] "TAU 2018 Contest," https://sites.google.com/view/taucontest2018/home.
[3] J. Hu *et al.*, "TAU 2015 contest on incremental timing analysis," in *IEEE/ACM ICCAD*, 2015, pp. 895–902.
[4] T.-W. Huang and M. D. F. Wong, "OpenTimer: A high-performance timing analysis tool," in *IEEE/ACM ICCAD*, 2015, pp. 895–902.
[5] "Parallax Software," http://www.parallaxsw.com/.
[6] "OpenSTA," https://github.com/abk-openroad/OpenSTA.
[7] Y.-M. Yang, Y.-W. Chang, and I. H.-R. Jiang, "iTimerC: Common path pessimism removal using effective reduction methods," in *IEEE/ACM ICCAD*, Nov 2014, pp. 600–605.
[8] P.-Y. Lee, I. H.-R. Jiang, and T.-C. Chen, "Fastpass: Fast timing path search for generalized timing exception handling," in *IEEE/ACM ASP-DAC*, 2018, pp. 172–177.
[9] T.-W. Huang and M. D. F. Wong, "UI-Timer 1.0: An ultrafast path-based timing analysis algorithm for cppr," *IEEE TCAD*, vol. 35, no. 11, pp. 1862–1875, Nov 2016.
[10] K.-M. Lai, T.-W. Huang, and T.-Y. Ho, "A general cache framework for efficient generation of timing critical paths," in *IEEE/ACM DAC*, 2019.
[11] T.-W. Huang, C.-X. Lin, G. Guo, and M. D. F. Wong, "Cpp-Taskflow: Fast task-based parallel programming using modern C++," in *IEEE IPDPS*, 2019.