Incremental Critical Path Generation for Dynamic Graphs

Che Chang
University of Wisconsin, Madison
Madison, WI, USA
che.chang@wisc.edu

Cheng-Hsiang Chiu
University of Wisconsin, Madison
Madison, WI, USA
chenghsiang.chiu@wisc.edu

Boyang Zhang
University of Wisconsin, Madison
Madison, WI, USA
bzhang523@wisc.edu

Tsung-Wei Huang
University of Wisconsin, Madison
Madison, WI, USA
tsung-wei.huang@wisc.edu

***Abstract*—Incremental Critical Path Generation (CPG) is crucial for static timing analysis (STA) applications to incrementally analyze critical paths and validate timing constraints. However, current state-of-the-art incremental CPG algorithms can only handle static graphs where the graph topology does not change. To solve this problem, we introduce an efficient incremental CPG algorithm that can handle dynamic graphs. Compared to existing methods, our algorithm can identify more scenarios to efficiently reuse paths from the previous CPG while discarding unnecessary ones after the graph topology is changed. Our algorithm is up to 8.2× faster than a state-of-the-art timer when generating two million paths on a large design.**

I. Introduction

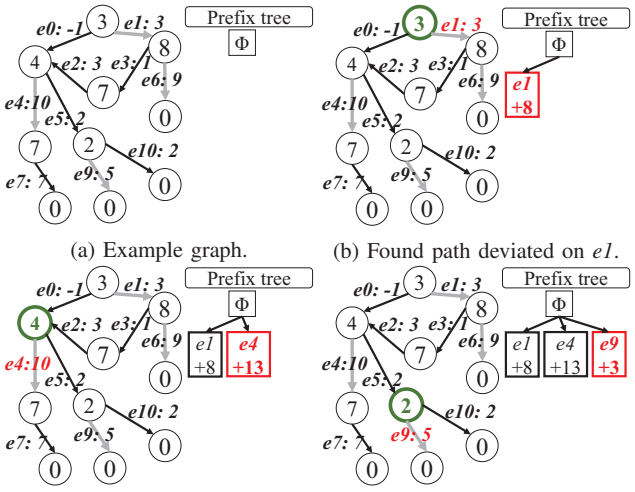
Critical Path Generation (CPG) is an important step for static timing analysis (STA) applications to validate timing constraints. For example, timers rely on CPG to perform common path pessimism removal. Among various CPG algorithms [1]–[21], the state-of-the-art OpenTimer [1] has introduced a fast implicit path representation algorithm for CPG. However, OpenTimer suffers from the lack of incrementality, which is the ability to quickly update critical paths after the circuit is incrementally modified. Incremental CPG plays an important role in many optimization flows, such as timing-driven placement [22] and gate sizing [23].

Recently, Chang et al. introduced the first incremental CPG algorithm called *Ink* [24] atop OpenTimer. Ink partially reuses the path results from the previous CPG query and eliminates unnecessary path computations. However, Ink is limited to *static graph*, where the graph topology does not change. Static graph is particularly suitable for certain timing-driven applications, such as gate sizing, that only modify vertex or edge attributes rather than the graph topology. That is, Ink cannot be used for generic timing-driven applications that analyze critical paths on dynamic graphs.

To overcome this problem, we propose an efficient incremental CPG algorithm that can handle dynamic graphs. Compared to Ink, our algorithm can identify more scenarios to efficiently reuse paths from the previous CPG while discarding unnecessary ones after the graph topology is changed. Therefore, our algorithm can be applied to various timing-driven applications that run incremental CPG on both static and dynamic graphs.

We evaluate the performance of our algorithm on real circuit benchmarks generated by OpenTimer [1]. Compared to OpenTimer's CPG algorithm, our algorithm is up to 8.2× faster when generating two million paths on a large design.

II. Background



(a) Example graph. (b) Found path deviated on *e1*.

(c) Found path deviated on *e4*. (d) Found path deviated on *e9*.

Fig. 1: Example prefix tree expansion for CPG.

OpenTimer's [1] CPG algorithm has two stages: suffix tree construction and prefix tree expansion. The suffix tree is a shortest path tree rooted at the destination vertices, which can be built with topological relaxations. The second stage is to explore paths that deviate from the suffix tree by performing prefix tree expansion. To be clear, to expand the prefix tree means to expand the critical path search space by finding the children nodes for a certain prefix tree node.

Figure 1 illustrates OpenTimer's CPG algorithm on an example graph. As shown in Figure 1(a), each edge is associated

2159-3477/24/$31.00 ©2024 IEEE
DOI 10.1109/ISVLSI61997.2024.00150

with a weight. Black edges represent the suffix tree, which are the edges on the shortest path to the destination vertices. Gray edges are the edges that do not belong to the suffix tree, and we call them the *non-suffix tree edges*. The numbers on the vertices represent the shortest distance from that vertex to its destination vertex. The prefix tree now contains an initial node Φ, which implicitly represents the shortest path. Figure 1(b) shows that we visit the source vertex (marked in green). We explore one new path by deviating on the non-suffix tree edge *e1*. We create a new prefix tree node (red box) associated with *e1*. We also need to determine the "deviation cost" of this new prefix tree node. The deviation cost represents the distance loss by deviating on a given edge. For example, in Figure 1(b), the red node has a deviation cost of 8 because by choosing *e1*, the distance to the destination vertices becomes 8 (shortest distance from *tail*[*e1*] to the destination vertices) + 3 (weight of *e1*) = 11, which is 8 units more than the shortest distance. Figure 1(c) shows that we traverse along the shortest path to the next vertex (marked in green) and explore one new path by deviating on the non-suffix tree edge *e4*. The deviation cost is 13. Figure 1(d) shows that we traverse along the shortest path to the next vertex (marked in green) and explore one new path by deviating on the non-suffix tree edge *e9*. The deviation cost is 3. In this example, we discovered three new paths, and we can rank them in the correct order with their deviation costs. OpenTimer uses a priority queue to ensure we always expand from the prefix tree node with the smallest deviation cost.

We plan to open-source the project to benefit the community [25] and leverage task-parallel computing libraries [26]–[40] to further improve the overall performance.

III. Algorithm

When a circuit topology is modified, updating the suffix tree is equivalent to relaxing the shortest path distance values of affected nodes, which can be done using the algorithm proposed by Ink [24]. As a result, we focus on incrementally updating the prefix tree between two different graphs. In addition to the theorem proposed by Ink, we further identify a key property of the prefix tree, that is, *no matter how the circuit graph changes, we only need to consider two scenarios in which a prefix tree node associated with edge e will become invalid*:

- Scenario 1: *e* becomes a suffix tree edge.
- Scenario 2: *e* is removed by the user.

These two scenarios identify which nodes we need to remove, which in turn identify the reusable nodes. Figure 2 illustrates the two scenarios. Figure 2(a) shows an example partial graph and its corresponding prefix tree. *P* represents an arbitrary parent node. Figure 2(b) shows if *e3* belongs to the suffix tree after some graph updates, we should remove the node associated with *e3*, because it will no longer appear in the prefix tree. Since the edge connecting vertex *C* and *D* has become a non-suffix tree edge, any nodes discovered after *e3* (e.g. *e4*) should also be removed because the edges that they are associated with are no longer reachable during the expansion. In this case, the nodes associated with *e0*, *e1*, *e2* are reusable, and we can prune them from the search space for the subsequent prefix tree expansion. We only need to partially expand the subsequent prefix tree, which effectively reduces runtime. Figure 2(c) shows if *e3* is removed by the user after some graph updates, we should only remove the node associated with *e3*. We can keep the node associated with *e4* since the edge connecting *C* and *D* is still a suffix tree edge. This indicates that *e4* is still reachable during the expansion. In this case, the nodes associated with *e0*, *e1*, *e2*, and *e4* are reusable, and we can prune them from the search space for the subsequent prefix tree expansion.

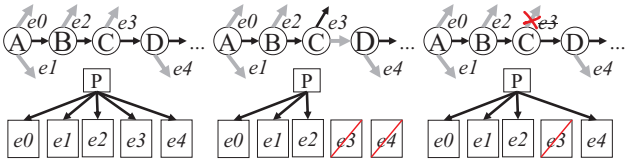
Algorithm 1 describes the proposed incremental prefix tree expansion algorithm for dynamic graphs. We initialize an array *R* to record the nodes to re-expand (line 2). We use BFS to traverse the prefix tree (lines 3 and 14:16) from the previous CPG query because we need to visit the nodes in the order in which they are discovered. We pop *node* from the queue (line 4), if *node* belongs to the suffix tree (*Scenario 1*), it disappears from the prefix tree, so we mark it and its children as removed (line 5:6). *node*'s right siblings are discovered after *node* and are no longer reusable, so we remove them as well (line 6). If *node*.edge is removed by the user (*Scenario 2*), then *node*'s children should also be removed (line 7:8). If none of the above occurs, then *node* is reusable, we update its deviation cost (line 10) and prune *node* from *node*.parent's search space to avoid generating duplicated nodes (line 11). We record *node*.parent in *R* to incrementally expand later (line 12:13).

Algorithm 1: IncPfxt(*P*)

```
   Input: prefix tree P, suffix tree S
   Output: array of nodes to re-expand R
1  Q ← initialize a queue with the root of P;
2  R ← ϕ;
3  while Q is not empty
4      node ← Q.pop();
5      if node.edge belongs to S then
6          ▷ Scenario 1: mark node, node.children,
           node.right_siblings as removed;
7      else if node.edge is removed by user then
8          ▷ Scenario 2: mark node and node.children as
           removed;
9      else
10         update node.deviation_cost;
11         prune node from node.parent's search space;
12         if node.parent ∉ R then
13             R ← R ∪ node.parent;
14     Foreach c ∈ node.children
15         if c is not marked as removed then
16             Q.push(c);
17 return R;
```

IV. Experimental Results

We implemented our algorithm in C++ and compiled it with GCC 11.4.0 on a 4.8-GHz 64-bit Linux machine of an



(a) A prefix tree. (b) Scenario 1. (c) Scenario 2.

Fig. 2: Scenarios that invalidate prefix tree nodes.

Intel Core i5-13500 Processor. We enable the optimization flag `-O3` and C++17 standard `-std=c++17`. We evaluate the performance of our algorithm with large circuit benchmarks generated by OpenTimer [1]. An incremental iteration is a circuit modifier (insertion or deletion of edges and vertices) followed by a CPG query that reports the top-k critical paths. All data is an average of 50 incremental iterations. We do not compare accuracy because our proposed algorithm can generate the exact path slacks as OpenTimer.

A. Overall Performance

Table I compares the runtime of our incremental CPG algorithm and OpenTimer [1]'s full CPG algorithm. We follow the experimental setting of Ink [24] but change its circuit modifiers to handle dynamic graphs through random insertion and deletion of edges and vertices. Full CPG (denoted as "OT") refers to the update that re-runs the whole CPG without incrementality, while incremental CPG (denoted as "Ours") refers to our proposed algorithm. As shown in Table I, our algorithm outperforms full CPG in all benchmarks. For example, our algorithm is 8.2× faster in netcard.

TABLE I: Overall performance comparison between full and incremental CPG on dynamic graphs (*OT*: full CPG, *Ours*: incremental CPG). Runtime is measured at milliseconds.

Circuit	$\|V\|$	$\|E\|$	#Paths	OT	Ours	Speedup
wb_dma	13K	16K	40K	6.5	2	3.2×
ac97_ctrl	42K	53K	100K	16.7	6.3	2.6×
aes_core	66K	86K	180K	71.9	35	2×
des_perf	303K	387K	500K	344	211	1.6×
vga_lcd	397K	498K	1M	1090	230	4.8×
netcard	3.9M	4.9M	2M	4763	581	8.2×
leon2	4.3M	5.2M	2M	7102	1123	6.3×

Figure 3 plots the runtime distribution of full CPG and our proposed algorithm across 50 incremental iterations. Regardless of the runtime variation, we see a consistent gap between full CPG and our algorithm. Taking netcard for example, the runtime gap between full CPG and our algorithms is roughly 4000 ms at the 12^{th} iteration. This is because our proposed algorithm partially reuses paths from the previous incremental iteration, we only need to update the slacks of these paths instead of recomputing them, which greatly reduces runtime.

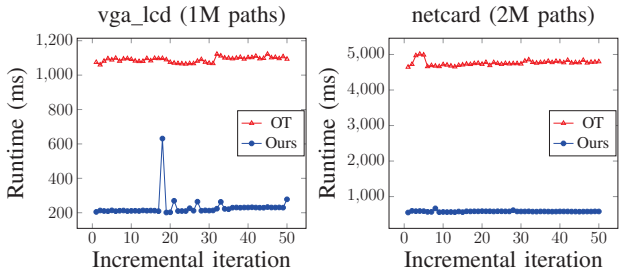


Fig. 3: Runtime distribution across 50 incremental iterations for vga_lcd and netcard.

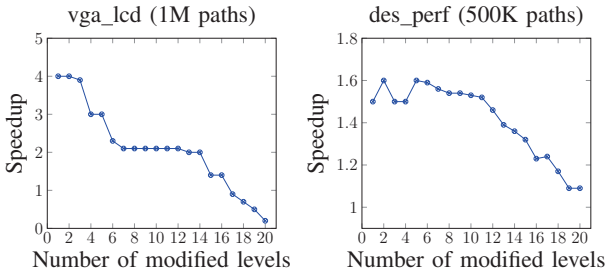


Fig. 4: Speedup vs. number of modified levels for vga_lcd and des_perf.

B. Performance at Different Incrementalities

Figure 4 plots the speedup of our proposed algorithm over full CPG at different numbers of modified levels. In this experiment, we perform BFS and levelize the circuit graphs to investigate how the number of modified levels affects the performance of our algorithm. We modify the levels in a bottom-up fashion. For example, 20 modified levels means we modified the last 20 levels of the graph. We do not consider modifying the levels top-down because it may trigger our proposed algorithm to discard all the paths from the previous CPG query, which is equivalent to full CPG. This experiment is important because the speedup of our proposed algorithm over full CPG is closely tied to how much the most critical path is impacted by the circuit modifiers. The experiments in section IV-A may not fully capture this relationship. Modifying the circuit graph in levels ensures that we always apply the circuit modifiers to the most critical path.

The speedup drops as we increase the modified levels. Taking des_perf for example, the speedup is 1.6× at two levels and 1.2× at 17 levels. This is because as we increase the modified levels, we force our proposed algorithm to discard more prefix tree nodes from the previous CPG query and fewer reusable nodes are left. This trend is particularly evident in vga_lcd, since vga_lcd's connectivity is large. Modifying one level causes our proposed algorithm to discard significantly more nodes than des_perf. For example, the speedup drops to less than 1× at 17 levels in vga_lcd. At this point, our proposed algorithm has no benefit over full CPG.

V. Conclusion

We have introduced an incremental CPG algorithm for dynamic graphs. Compared to Ink [24], our algorithm can identify more scenarios to efficiently reuse paths from the previous CPG while discarding unnecessary ones after the graph topology is changed. Our algorithm is up to 8.2× faster than a state-of-the-art timer when generating two million paths on a large design.

Acknowledgment

This project is supported by NSF grants 2235276, 2349144, 2349143, 2349582, and 2349141.

References

[1] T.-W. Huang, G. Guo, C.-X. Lin, and M. D. F. Wong, "OpenTimer v2: A New Parallel Incremental Timing Analysis Engine," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, 2021.

[2] B. Jin, G. Luo, and W. Zhang, "A fast and accurate approach for common path pessimism removal in static timing analysis," in *IEEE ISCAS*, 2016, pp. 2623–2626.

[3] T.-W. Huang and M. Wong, "OpenTimer: A High-Performance Timing Analysis Tool," in *IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, 2015.

[4] ——, "UI-Timer 1.0: An Ultra-Fast Path-Based Timing Analysis Algorithm for CPPR," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, 2016.

[5] G. Guo, T.-W. Huang, Y. Lin, and M. Wong, "GPU-accelerated Path-based Timing Analysis," in *IEEE/ACM Design Automation Conference (DAC)*, 2021.

[6] G. Guo, T.-W. Huang, Y. Lin, Z. Guo, S. Yellapragada, and M. D. F. Wong, "A gpu-accelerated framework for path-based timing analysis," *IEEE Transactions on Computer-aided Design of Integrated Circuits and Systems (TCAD)*, 2023.

[7] T.-W. Huang, P.-C. Wu, and M. D. F. Wong, "Ui-timer: An ultra-fast clock network pessimism removal algorithm," in *IEEE/ACM ICCAD*, 2014.

[8] G. Guo, T.-W. Huang, Y. Lin, and M. Wong, "GPU-accelerated Critical Path Generation with Path Constraints," in *IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, 2021.

[9] Z. Guo, T.-W. Huang, and Y. Lin, "A Provably Good and Practically Efficient Algorithm for Common Path Pessimism Removal in Large Designs," in *IEEE/ACM International Conference on Computer-aided Design (ICCAD)*, 2020.

[10] ——, "Accelerating Static Timing Analysis using CPU-GPU Heterogeneous Parallelism," *IEEE Transactions on Computer-aided Design of Integrated Circuits and Systems (TCAD)*, 2023.

[11] ——, "HeteroCPPR: Accelerating Common Path Pessimism Removal with Heterogeneous CPU-GPU Parallelism," in *IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, 2021.

[12] G. Guo, T.-W. Huang, and M. D. F. Wong, "Fast STA Graph Partitioning Framework for Multi-GPU Acceleration," in *IEEE/ACM Design, Automation and Test in Europe Conference (DATE)*, 2023.

[13] Z. Guo, T.-W. Huang, J. Zhou, C. Zhuo, Y. Lin, R. Wang, and R. Huang, "Heterogeneous Static Timing Analysis with Advanced Delay Calculator," in *IEEE/ACM Design, Automation and Test in Europe Conference (DATE)*, 2024.

[14] G. Guo, T.-W. Huang, C.-X. Lin, and M. Wong, "An Efficient Critical Path Generation Algorithm Considering Extensive Path Constraints," in *ACM/IEEE Design Automation Conference (DAC)*, 2020.

[15] Z. Guo, T.-W. Huang, and Y. Lin, "A Provably Good and Practically Efficient Algorithm for Common Path Pessimism Removal in Large Designs," in *IEEE/ACM Design Automation Conference (DAC)*, 2021.

[16] T.-W. Huang, C.-X. Lin, , and M. Wong, "Distributed Timing Analysis at Scale," in *ACM/IEEE Design Automation Conference (DAC)*, 2019.

[17] T.-W. Huang, M. Wong, D. Sinha, K. Kalafala, and N. Venkateswaran, "A Distributed Timing Analysis Framework for Large Designs," in *IEEE/ACM Design Automation Conference (DAC)*, 2016.

[18] T.-W. Huang, "A General-purpose Parallel and Heterogeneous Task Programming System for VLSI CAD," in *IEEE/ACM International Conference on Computer-aided Design (ICCAD)*, 2020.

[19] T.-W. Huang, P.-C. Wu, and M. Wong, "Fast Path-Based Timing Analysis for CPPR," in *IEEE/ACM ICCAD*, 2014.

[20] T.-W. Huang and L. Hwang, "Task-parallel Programming with Constrained Parallelism," in *IEEE High-Performance Extreme Computing Conference (HPEC)*, 2022.

[21] T.-W. Huang, "Enhancing the Performance Portability of Heterogeneous Circuit Analysis Programs," in *IEEE High-Performance Extreme Computing Conference (HPEC)*, 2022.

[22] M.-C. Kim, J. Hu, J. Li, and N. Viswanathan, "Iccad-2015 cad contest in incremental timing-driven placement and benchmark suite," in *IEEE/ACM ICCAD*, 2015.

[23] D. Mangiras, D. Chinnery, and G. Dimitrakopoulos, "Task-based parallel programming for gate sizing," *IEEE TCAD*, 2023.

[24] C. Chang, T.-W. Huang, D.-L. Lin, G. Guo, and S. Lin, "Ink: Efficient Incremental k-Critical Path Generation," in *ACM/IEEE DAC*, 2024.

[25] T.-W. Huang, C.-X. Lin, G. Guo, and M. Wong, "Essential Building Blocks for Creating an Open-source EDA Project," in *ACM/IEEE Design Automation Conference (DAC)*, 2019.

[26] C.-H. Chiu, D.-L. Lin, and T.-W. Huang, "Programming Dynamic Task Parallelism for Heterogeneous EDA Algorithms," in *IEEE/ACM International Conference on Computer-aided Design (ICCAD)*, 2023.

[27] D.-L. Lin, T.-W. Huang, J. S. Miguel, and U. Ogras, "TaroRTL: Accelerating RTL Simulation using Coroutine-based Heterogeneous Task Graph Scheduling," in *International European Conference on Parallel and Distributed Computing (Euro-Par)*, 2024.

[28] T.-W. Huang, D.-L. Lin, C.-X. Lin, and Y. Lin, "Taskflow: A Lightweight Parallel and Heterogeneous Task Graph Computing System," *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, 2022.

[29] T.-W. Huang, C.-X. Lin, G. Guo, and M. Wong, "Cpp-Taskflow: Fast Task-based Parallel Programming using Modern C++," in *IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2019.

[30] T.-W. Huang, Y. Lin, C.-X. Lin, G. Guo, and M. Wong, "Taskflow: A General-purpose Parallel Task Programming System at Scale," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, 2021.

[31] C.-X. Lin, T.-W. Huang, G. Guo, and M. Wong, "A Modern C++ Parallel Task Programming Library," in *ACM Multimedia Conference (MM)*, 2019.

[32] ——, "An Efficient and Composable Parallel Task Programming Library," in *IEEE High-performance and Extreme Computing Conference (HPEC)*, 2019.

[33] C.-X. Lin, T.-W. Huang, and M. Wong, "An Efficient Work-Stealing Scheduler for Task Dependency Graph," in *IEEE International Conference on Parallel and Distributed Systems (ICPADS)*, 2020.

[34] B. Zhang, D.-L. Lin, C. Chang, C.-H. Chiu, B. Wang, W. L. Lee, C.-C. Chang, D. Fang, and T.-W. Huang, "G-PASTA: GPU Accelerated Partitioning Algorithm for Static Timing Analysis," in *ACM/IEEE DAC*, 2024.

[35] W. L. Lee, D.-L. Lin, T.-W. Huang, S. Jiang, T.-Y. Ho, Y. Lin, and B. Yu, "G-kway: Multilevel GPU-Accelerated k-way Graph Partitioner," in *ACM/IEEE Design Automation Conference (DAC)*, 2024.

[36] T.-W. Huang, B. Zhang, D.-L. Lin, and C.-H. Chiu, "Parallel and Heterogeneous Timing Analysis: Partition, Algorithm, and System," in *ACM International Symposium on Physical Design (ISPD)*, 2024.

[37] C.-H. Chiu and T.-W. Huang, "Efficient Timing Propagation with Simultaneous Structural and Pipeline Parallelisms," in *ACM/IEEE Design Automation Conference (DAC)*, 2022.

[38] ——, "Composing Pipeline Parallelism using Control Taskflow Graph," in *ACM International Symposium on High-Performance Parallel and Distributed Computing (HPDC)*, 2022.

[39] T.-W. Huang and Y. Lin, "Concurrent CPU-GPU Task Programming using Modern C++," in *IEEE International Workshop on High-level Parallel Programming Models and Supportive Environments (HIPS)*, 2022.

[40] D.-L. Lin and T.-W. Huang, "Efficient GPU Computation using Task Graph Parallelism," in *European Conference on Parallel and Distributed Computing (Euro-Par)*, 2021.