Ink: Efficient Incremental k-Critical Path Generation

Che Chang
University of Wisconsin at Madison
Madison, Wisconsin, USA

Tsung-Wei Huang
University of Wisconsin at Madison
Madison, Wisconsin, USA

Guannan Guo
University of Illinois Urbana-Champaign
Urbana-Champaign, Illinois, USA

Shiju Lin
The Chinese University of Hong Kong
Hong Kong, China

ABSTRACT

Critical Path Generation (CPG) is crucial for static timing analysis (STA) applications to validate timing constraints. Recent years have witnessed CPG algorithms that can rank k critical paths efficiently and accurately. However, they all suffer from the lack of *incrementality*, which is the ability to quickly update critical paths after the circuit is incrementally modified. To solve this problem, we introduce Ink, an efficient incremental CPG algorithm. Inspired by the large path trace similarity between adjacent CPG queries, Ink identifies a set of paths to reuse for the next query and effectively prunes the path search space. We have demonstrated the promising performance of Ink on large circuit benchmarks. Ink is up to 22.4× faster and consumes up to 31% less memory than a state-of-the-art timer when generating one million paths on a large design.

1 INTRODUCTION

[1] Critical Path Generation (CPG) is a key routine in static timing analysis (STA) applications. For example, a practical timer counts on CPG to perform path-based analysis (PBA), such as common path pessimism removal (CPPR) and advanced on-chip variation (AOCV) update, for removing unwanted pessimism [1]. As the design complexity continues to grow, CPG runtime can become a significant bottleneck in many STA engines [3]. To alleviate this problem, academia has introduced various CPG algorithms that can rank k critical paths efficiently. For example, iTimerC introduces a branch-and-bound technique to prune redundant path traversals [6]; iitRace introduces a pin coloring scheme to perform efficient path reduction [9]; OpenTimer introduces a fast implicit path representation algorithm using suffix tree and prefix tree [4].

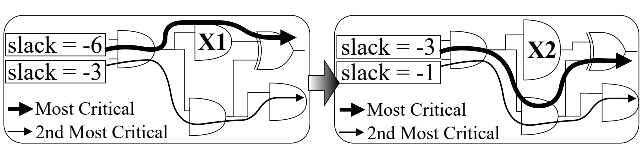


Figure 1: Illustration of CPG (k = 2) for a gate sizing operation (X1→X2). The second most critical path trace is unaffected.

Although existing CPG algorithms have demonstrated efficiency and accuracy [3], they all suffer from the lack of *incrementality*, which is the ability to quickly update critical paths after the circuit is incrementally modified. Incrementality plays an important role in many optimization flows, such as timing-driven placement [5] and gate sizing [8]. Figure 1 shows two critical paths before and after a gate sizing operation that incrementally modifies the circuit. Despite different slack values, critical path traces exhibit a large similarity between the two CPG queries (e.g., the second most critical path trace does not change). In fact, according to [3], the overlap ratio of path traces between adjacent incremental timing iterations can go up to 90%. This implies that many path results computed in the previous CPG query are highly reusable for the next CPG query. Without incrementality, CPG algorithms will waste substantial time and memory on recomputing the same paths.

However, designing a fast incremental CPG algorithm is very challenging because we need to efficiently identify which paths to keep and reuse for the next CPG query after the circuit is modified. When those paths are identified, we need to effectively prune them from the search space to avoid duplicated paths. To overcome these challenges, we introduce Ink, an efficient incremental CPG algorithm. Ink is inspired by the implicit path representation algorithm of OpenTimer [4] (suffix and prefix trees), but redesigns its core search routine to efficiently support incrementality. We summarize three technical contributions of Ink as follows:

- We design a fast incremental suffix tree update algorithm that minimally identifies the affected subgraph of the suffix tree and performs only the necessary updates on shortest path values.
- We design a fast incremental prefix tree expansion algorithm that identifies a set of paths to reuse for the next CPG query. After identifying these paths, we can effectively prune the path search space.
- We give rigorous analysis to justify the correctness and complexity of the proposed algorithms.

We evaluate Ink's performance on real circuit benchmarks generated by a state-of-the-art timer, OpenTimer [4]. Compared to OpenTimer's CPG algorithm [4], Ink is up to 22.4× faster and consumes up to 31% less memory when generating one million critical paths on a large design. We plan to make Ink open-source to benefit timing optimization-related research.

2 BACKGROUND

2.1 Incremental Critical Path Generation

The circuit network is input as a directed-acyclic graph $G = \{V, E\}$. V is a set of n vertices that represent pins of circuit components (e.g., logic gates, flip-flops, etc.). E is a set of m edges that represent pin-to-pin connections. Each edge e is directed from its head vertex u to tail vertex v and is associated with a delay w_e. A path is an ordered sequence of edges $\langle e_1, e_2, ..., e_i \rangle$. The path delay is the summation of delays through all edges of that path. A circuit modifier is an operation that modifies the circuit to perform timing-driven optimization.

[1]This is a work-in-progress paper just for the disclosure purpose for our LBR submission.

In this paper, we target the circuit modifier that only alters the edge weights of the graph, which is a specific yet widely used scenario. For example, many gate sizing and timing-driven placement applications call an incremental timer after resizing a gate and moving a cell, where the circuit topology is not changed [2, 7]. Also, based on our experience, the algorithm to handle incrementality between different topologies will likely be very different from Ink, which deserves further research work.

Given a circuit graph G and a positive integer k, a CPG query reports the top-k critical paths in ascending order of path slack (or path delay depending on how the graph is formulated [4]). An incremental iteration is defined as at least one circuit modifier followed by one CPG query.

2.2 Implicit Path Representation

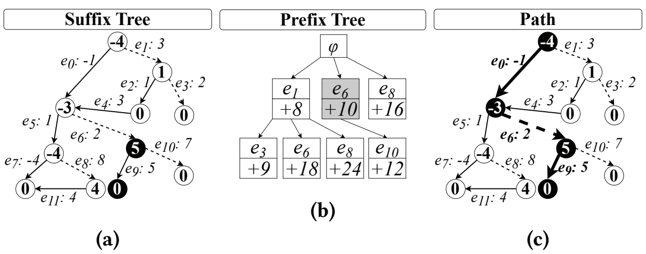


Figure 2: Implicit path representation using suffix tree and prefix tree. Suffix $\langle e_9 \rangle$ + Prefix $\langle e_0, e_6 \rangle$ = Path $\langle e_0, e_6, e_9 \rangle$.

Although there are many CPG algorithms [4, 6, 9], we adopt the implicit path representation algorithm proposed by OpenTimer [4], which outperforms existing algorithms in space and time complexity. As shown in Figure 2, OpenTimer represents critical paths using two complementary data structures, *suffix tree* and *prefix tree*. A suffix tree is a shortest path tree rooted at the destination vertices, constructed with topological relaxations. Figure 2(a) shows an example graph and its suffix tree. Solid edges denote the suffix tree, and dashed edges denote non-suffix tree edges. Numbers on the vertices denote the shortest distance to their destination vertices.

A prefix tree is a tree order of non-suffix tree edges. Each prefix tree node implicitly represents a path deviated from its parent path. The prefix tree root refers to the shortest path in the suffix tree. Figure 2(b) shows an example. The prefix tree root φ implicitly represents the shortest path $\langle e_0, e_5, e_7 \rangle$ in the suffix tree. The prefix tree node marked by "e_6" (colored in gray) implicitly represents the path with prefix $\langle e_0 \rangle$ from its parent path deviated on e_6 and followed by suffix $\langle e_9 \rangle$ from the suffix tree. Figure 2(c) illustrates this path as bold edges $\langle e_0, e_6, e_9 \rangle$. To retrieve the path delay, we record the "deviation cost" of each non-suffix tree edge e: $dvi[e] = dis[tail[e]] + weight[e] - dis[head[e]]$, where $dis[v]$ denotes the shortest distance from vertex v to its destination vertex. Intuitively, deviation cost measures the distance loss by deviating on edge e instead of taking the ordinary shortest path to the destination vertex. For example, in Figure 2(a), e_6 has a deviation cost of $dis[tail[e_6]] + weight[e_6] - dis[head[e_6]] = 10$, which means by deviating on e_6, we get a path that is 10 longer than the shortest path from $head[e_6]$ to its destination vertex. To conclude, Table 1 lists the data fields to which we apply for each prefix tree node [4].

Constructor	PfxtNode(p, e, w)	RespurListItem(pfx, pes)
Members	p: parent node e: deviation edge w: cumulative $dvi[e]$	pfx: prefix tree node pes: pruned edges for pfx

Table 1: Data fields of a prefix tree node (PfxtNode) and a re-spur list item (RespurListItem).

3 INK: INCREMENTAL k-CRITICAL PATH GENERATION

Ink has two stages, *incremental suffix tree update* and *incremental prefix tree expansion*, to perform incremental CPG.

3.1 Incremental Suffix Tree Update

The goal of incremental suffix tree update is to perform only necessary topological relaxations on the affected subgraph of the suffix tree, as opposed to the complete bottom-up topological relaxations in OpenTimer [4]. Algorithm 1 presents the incremental suffix tree update algorithm. After collecting an array of head vertices M from user-modified edges, we perform DFS on M to identify the affected vertices V in reversed topological order (line 2). We record the affected prefix tree nodes for the second stage (line 5:6) and perform edge relaxations on the fanouts of each vertex in V (line 7).

Following the suffix tree example in Figure 2(a), Figure 3(a) shows that we modify the weights of e_1, e_3, e_6, and e_{10}. Figure 3(b) shows that after performing DFS on the head vertices of the modified edges, we identify five affected vertices (marked in gray). We then perform edge relaxations on the fanouts of these five vertices. For example, as shown in Figure 3(b), we perform edge relaxations on e_5 ($dis[tail[e_5]] + weight[e_5] = -3$) and e_6 ($dis[tail[e_6]] + weight[e_6] = -4$). Since $-3 > -4$, -4 becomes $head[e_6]$'s new shortest distance to its destination vertex. $tail[e_6]$ is the new successor of $head[e_6]$. Lemma 1 concludes Algorithm 1.

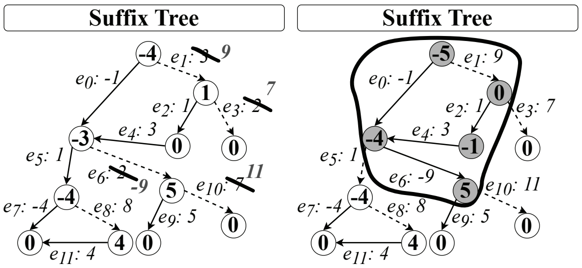
Lemma 1. *Algorithm 1 takes $O(n + km)$ time complexity.*

PROOF. In addition to the suffix tree update time $O(n + m)$ using topological relaxation, each of the m edges needs up to k iterations to collect its dependent prefix tree nodes. Therefore, the total time complexity is $O(n + km)$. □

Algorithm 1: IncSfxt(M)

Input: array of head vertices of user-modified edges M
Global: array of affected prefix tree nodes P

```
1 P ← ϕ;
2 V ← DFS on M to identify affected vertices in reversed
    topological order;
3 Foreach u ∈ V
4     Foreach e ∈ fanout(u)
5         Foreach n ∈ dependent_pfxt_nodes(e)
6             P ← P ∪ n;
7         Relax(u, tail[e], weight[e]);
```



(a) Edge weight modifications **(b) Updated subgraph**

Figure 3: Illustration of Algorithm 1. We only perform topological relaxations on the fanouts of the gray vertices in (b).

3.2 Incremental Prefix Tree Expansion

After updating the suffix tree, the next step is to explore paths that deviate from the suffix tree by expanding the prefix tree. To be clear, “expand” means to generate the children nodes for a certain prefix tree node by finding non-suffix tree edges to deviate on. When a timing-driven application queries k critical paths (potentially very large k), expanding the prefix tree becomes very expensive if not done incrementally. However, incremental prefix tree expansion has two major challenges: 1) we need to know which prefix tree nodes are reusable after applying the circuit modifiers and 2) after identifying these nodes, we need to prune them from the search space for the next query to avoid generating duplicated nodes. To overcome challenge 1, we introduce a theorem that serves as the cornerstone of our incremental prefix tree expansion algorithm:

Theorem 1. *Given a prefix tree node p and p's children C, and each child $c_i \in C$ is associated with an edge e_i, where i represents the order in which c_i is discovered. $\forall i, j \in \mathbb{Z}_{>0}$, if $i < j$ and e_j becomes a suffix tree edge after the circuit is changed, then c_i remains p's child.*

Proof. Assume c_i is not p's child, we examine two cases: 1) if e_i and e_j have the same head vertex v, e_i must be a suffix tree edge, which contradicts the fact that e_j is the only suffix edge among v's fanouts. 2) if e_i and e_j have different head vertices, since c_j is discovered later than c_i, c_i is not affected. Thus, by contradiction Theorem 1 is correct. □

Intuitively, Theorem 1 states that if c_j is associated with a suffix tree edge after the circuit is changed (meaning that c_j will disappear from the prefix tree in the next CPG query), we can reuse c_j's left siblings because they are discovered before c_j and removing c_j does not affect them. We only need to update these siblings' cumulative deviation costs. Since Theorem 1 applies to every level of the prefix tree, we can maximize the number of reusable nodes and reduce memory reallocation overhead. To overcome challenge 2, we maintain a “re-spur list” that records which nodes need re-expansion. For each of these nodes, to avoid generating duplicated children nodes, we also record which edges to skip during re-expansion. Table 1 lists the data field to which we apply for each re-spur list item. *pes* records what edges we should skip when generating the children nodes for *pfx*.

Algorithm 2 describes a key subroutine of Ink, MarkPfxtNodes. The goal of Algorithm 2 is to categorize the prefix tree nodes into reusable and removed nodes by applying Theorem 1. We update

Algorithm 2: MarkPfxtNodes(P, Q)

Input: array of affected prefix tree nodes P, queue Q
Output: re-spur list R

1 Sort P in ascending order of level;
2 R, $pes \leftarrow \phi$;
3 **Foreach** $p \in P$
4   **if** p is updated **or** p is removed **then**
5   **continue**;
6   **Foreach** $s \in siblings(p)$
7   Q.push(s);
8   **while** Q is not empty
9     $n \leftarrow Q$.pop();
10   **if** $n.parent \in$ a re-spur list item **then**
11   mark n as removed;
12   **if** n is not removed **then**
13   **if** $tail[n.e] = successor[head[n.e]]$ **then**
14   mark n as removed;
15   **if** $n.parent \notin$ a re-spur list item **then**
16           $r \leftarrow$ new RespurListItem($n.parent$, pes);
17           $R \leftarrow R \cup r$;
18   clear pes;
19   **else**
20   update $n.w$ and mark n as updated;
21         $pes \leftarrow pes \cup n.e$;
22   **Foreach** $c \in n.children$
23   Q.push(c);
24   **if** n is removed **then**
25   mark c as removed;
26 **return** R;

the cumulative deviation costs of the reusable nodes and mark others for lazy removal. Note that Algorithm 2 only prepares Ink for incremental prefix tree expansion by generating the re-spur list; the actual expansion happens in Algorithm 4. To ensure top-down traversal of the affected prefix tree nodes, we sort the array of affected prefix tree nodes P in ascending order of level (line 1). We initialize a re-spur list R and a set of pruned edges *pes* (line 2). For each node in P, if unmarked (line 4:5), we push its siblings to a queue Q to perform BFS (line 6:7). This is because Theorem 1 requires us to visit these nodes in the same order as they are discovered. We pop a node n from Q (line 9). If n's parent is already in the re-spur list (line 10), implying that a left sibling of n is marked as removed, we mark n as removed too (line 11), since n is discovered later than this sibling. If n is unmarked (line 12), we check if $n.e$ is a suffix tree edge (line 13). If so, n disappears from the prefix tree, and we mark n as removed (line 14). We create a re-spur list item (line 16:17), indicating that n's parent will later expand but skip *pes*. Otherwise, we update n's cumulative deviation cost and add n's edge to its parent's *pes* (line 20:21). We finally enqueue n's children for the later BFS iterations (line 22:25).

Continuing from the updated suffix tree in Figure 3(b), Figure 4 illustrates Algorithm 2. We denote a prefix tree node associated with e_i and cumulative deviation cost w as PfxtNode(e_i, w). For simplicity, we leave out the *parent node* member mentioned in

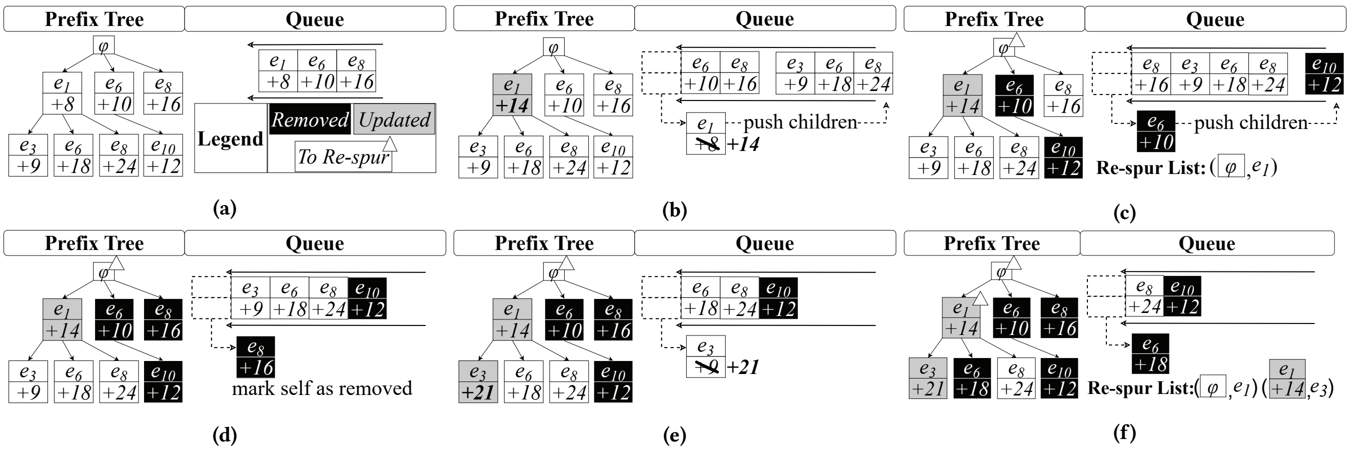


Figure 4: Illustration of Algorithm 2 (continuation of Figure 3(b)), a key subroutine of the proposed incremental prefix tree expansion algorithm. (a) Prefix tree and a queue that has PfxtNode(e_1, 8) and its siblings. (b) e_1 is still a non-suffix tree edge, so we update PfxtNode(e_1, 8)'s cumulative deviation cost to 14. (c) e_6 is now a suffix tree edge, so we mark PfxtNode(e_6, 10) and its children as removed. We then create a re-spur list item indicating that φ will skip e_1 during re-expansion. (d) e_8 is discovered later than e_6, so we mark PfxtNode(e_8, 16) and its children as removed. (e) Similar to (b), we update PfxtNode(e_3, 9)'s cumulative deviation cost to 21. (f) Similar to (c), we mark PfxtNode(e_6, 18) as removed. We then create a re-spur list item indicating that PfxtNode(e_1, 14) will skip e_3 during re-expansion.

Table 1, since it is already illustrated. Figure 4(a) illustrates the prefix tree for four paths and a queue containing PfxtNode(e_1, 8) and its siblings. Note that a four-critical path query may generate more than four nodes [4], so we see eight nodes in Figure 4(a). Figure 4(b) illustrates that we pop PfxtNode(e_1, 8) from the queue. Since e_1 is still a non-suffix tree edge, PfxtNode(e_1, 8) remains φ's child. We update PfxtNode(e_1, 8)'s cumulative deviation cost to $0{+}9{-}(-5) = 14$ using the shortest path values in Figure 3(b). We also push PfxtNode(e_1, 14)'s children to the queue. Figure 4(c) illustrates that we pop PfxtNode(e_6, 10) from the queue. Since e_6 is now a suffix tree edge, PfxtNode(e_6, 10) should be removed. We create a re-spur list item indicating that PfxtNode(e_6, 10)'s parent φ will skip e_1 during re-expansion. We should remove PfxtNode(e_6, 10)'s children as well, and we push them to the queue. Figure 4(d) illustrates that we pop PfxtNode(e_8, 16) from the queue. PfxtNode(e_8, 16)'s parent φ belongs to a re-spur list item, indicating that one of PfxtNode(e_8, 16)'s left siblings is removed. Since PfxtNode(e_8, 16) is discovered later than this removed sibling, we remove PfxtNode(e_8, 16) and its children. Figure 4(e)–(f) repeat the same procedure and finally produce two re-spur list items. Lemma 2 concludes Algorithm 2.

Lemma 2. *Algorithm 2 takes $O(k \log k)$ time complexity.*

Proof. We sort the affected prefix tree nodes at the beginning, which takes $O(k \log k)$ time complexity. It takes $O(k)$ time complexity to visit each node exactly once during the marking process. The total time complexity is $O(k \log k)$. □

Algorithm 3 describes another subroutine, which redesigns the Spur algorithm in [4] to support incrementality. Algorithm 3 expands the prefix tree from a given prefix tree node. Our algorithm includes a set of pruned edges *pes* as input, which allows us to minimally expand the prefix tree from a given node by pruning *pes* during expansion (lines 1 and 5). Lemma 3 concludes Algorithm 3.

Algorithm 3: SpurPruned(*pfx*, *d*, $\hat{Q}$, *pes*)

Input: a prefix tree node *pfx*, destination vertex *d*, priority queue $\hat{Q}$, a set of pruned edges *pes*

```
1  mark all edges in pes as pruned in the given graph;
2  u ← tail[pfx.e];
3  while u ≠ d
4  |   Foreach e ∈ fanout(u)
5  |   |   if tail[e] = successor[u] or e is pruned then
6  |   |   |   continue;
7  |   |   pfx_new ← new PfxtNode(pfx, e, pfx.w + dvi[e]);
8  |   |   Q̂.enqueue(pfx_new);
9  |   u ← successor[u];
10 unmark all edges in pes in the given graph;
```

Lemma 3. *Algorithm 3 takes $O(n + m \log k + k)$ time complexity.*

Proof. In addition to the Spur algorithm [4], which takes $O(n + m \log k)$ time complexity, we scan the pruned edges (lines 1 and 10), which takes $O(k)$ time complexity. Therefore, the total time complexity is $O(n + m \log k + k)$. □

Using Algorithms 2–3 as primitives, Algorithm 4 describes the incremental prefix tree expansion algorithm. The goal of Algorithm 4 is to retrieve the top-k critical paths in ascending order of path delay by incrementally expanding the prefix tree. Since we are retrieving paths incrementally, we transfer the essential information from the previous CPG query, including a priority queue $\hat{Q}$ of nodes keyed on their cumulative deviation costs (line 1) and the dequeued nodes Λ (line 2). We initialize the solution path set and a queue Q (line 3). We generate a re-spur list R using Algorithm 2 (line 4). Since Algorithm 2 invalidates $\hat{Q}$'s heap property, we heapify $\hat{Q}$ (line

5). With R, we can reuse updated nodes from the previous CPG and minimally expand the prefix tree (line 6:7). In OpenTimer [4], this critical path retrieval procedure always satisfies the condition where the nodes in Λ have cumulative deviation costs no more than the minimum cumulative deviation cost in $\hat{Q}$. However, Algorithm 2 may cause Λ to violate this condition. To solve this, we recover unremoved paths from Λ and record the maximum cumulative deviations cost *max_dc* in Λ (line 8); we also expand any leaf nodes in Λ, because they may have undiscovered children. If in the path search loop (line 9:18), we see a node that has a cumulative deviation cost less than *max_dc* (line 16), meaning the above condition is still violated, we continue executing the loop. The path search loop iteratively dequeues a node *pfx* (line 10), recovers the path (line 14:15), and then expands the search space for *pfx* (line 18) until we retrieved enough paths and the above condition is fulfilled. Combining Lemma 2–3, we draw the following theorem.

Theorem 2. *Algorithm 4 takes $O(n + m + k)$ space complexity and $O(kn + km \log k + k^2)$ time complexity.*

Proof. The space complexity of Algorithm 4 involves $O(n + m)$ for storing the circuit graph, $O(n)$ for the suffix tree, $O(k)$ for the prefix tree, and $O(k)$ for the re-spur list. Hence, the total space complexity is $O(n + m + k)$. We perform Algorithm 3 up to k iterations to obtain the top-k critical paths. Therefore, the total time complexity is $O(kn + km \log k + k^2)$. □

Algorithm 4: IncPfxt(d, k, P)

Input: destination vertex d, path count k, affected prefix tree nodes P
Output: solution set Ψ of critical paths

```
1  Q̂ ← transfer priority queue of nodes from the previous CPG;
2  Λ ← transfer dequeued nodes from the previous CPG;
3  Ψ, Q ← ϕ
4  R ← MarkPfxtNodes(P, Q);
5  Q̂.heapify();
6  Foreach r ∈ R
7      SpurPruned(r.pfx, d, Q̂, r.pes);
8  num_paths, max_dc, Ψ ← recover paths from nodes that are
     unremoved in Λ and record max cumulative deviation cost;
9  while Q̂ is not empty
10     pfx ← Q̂.dequeue();
11     if pfx is removed then
12         continue;
13     num_paths ← num_paths + 1;
14     path ← recover path from pfx;
15     Ψ ← Ψ ∪ path;
16     if pfx.w ≥ max_dc and num_paths ≥ k then
17         break;
18     SpurPruned(pfx, d, Q̂, ϕ);
19 return Ψ;
```

4 EXPERIMENTAL RESULTS

We implemented Ink in C++ and compiled it with GCC 11.4.0 on a 4.8-GHz 64-bit Linux machine of an Intel Core i5-13500 Processor. We enable the optimization flag `-O3` and C++17 standard `-std=c++17`. We select seven large circuits generated by OpenTimer [4] to evaluate Ink's performance.

4.1 Overall Performance Comparison

Table 2 compares the suffix tree update runtime, prefix tree expansion runtime, total runtime, and memory usage between full CPG and incremental CPG (Ink) on seven circuits. For each circuit, we measure the performance of Ink by taking the average of 100 incremental iterations that simulate a gate-sizing optimization algorithm [7] developed atop OpenTimer [4]. For wb_dma, tv80, ac97_ctrl, aes_core, and des_perf, we use their maximum path counts for each CPG call. For vga_lcd and netcard, whose maximum path counts are enormous, we use sufficiently large path counts (one million and five million) for each CPG call. Each incremental iteration randomly resizes a gate to alter the edge weight of the circuit graph and issue a CPG call to trigger a timing update. Full CPG refers to the update that re-runs the whole CPG without incrementality, which is how OpenTimer [4] deals with circuit graph updates, while incremental CPG refers to the proposed method. As shown in Table 2, Ink outperforms full CPG in all circuits. Since Ink partially reuses the previous CPG results, it is faster and uses less memory than full CPG. For example, Ink is 22.4× faster and uses 31% less memory in vga_lcd.

Figure 5 plots the runtime distribution of full CPG and Ink across 50 incremental iterations. Depending on the circuit modifier, the runtime per incremental iteration can vary. Regardless of the variation, we see a consistent runtime gap between full CPG and Ink. Taking netcard as an example, Ink is 8.3× faster than full CPG at the 22^{nd} incremental iteration.

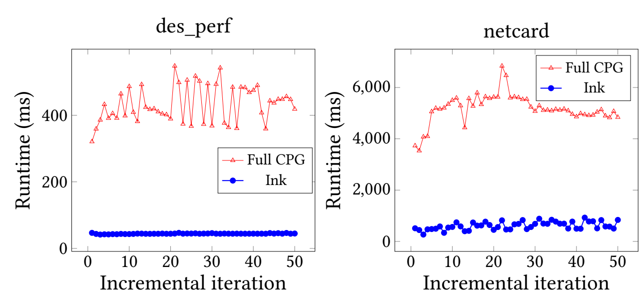


Figure 5: Runtime distribution of full CPG and Ink across 50 incremental iterations for des_perf and netcard.

4.2 Performance at Different Path Counts

Figure 6 demonstrates the speedup of Ink over full CPG at different path counts for des_perf and netcard. As we increase the path count, the speedup of Ink first decreases and then increases after a certain path count. For example, in des_perf, the speedup decreases from over 3× to less than 2× between one path and 100K paths, and then the speedup increases after 100K paths. This is because when the path count is small, Algorithm 1 is the major contributor to

Table 2: Overall performance comparison between full CPG (OpenTimer [4]) and incremental CPG (Ink).

				Full CPG (OpenTimer [4])				Incremental CPG (Ink)			
Circuit	$\|V\|$	$\|E\|$	Path count (K)	Sfxt (ms)	Pfxt (ms)	Total (ms)	Mem (MB)	Sfxt (ms)	Pfxt (ms)	Total (ms)	Mem (MB)
wb_dma	12602	8184	32	1.3	3.9	5.2	23.1	0.4 (3.3×)	0.6 (6.5×)	1 (**5.2**×)	17.8 (-23%)
tv80	16681	11364	45	2	6.3	8.3	30.4	0.5 (4×)	1.1 (5.7×)	1.6 (**5.2**×)	22.6 (-26%)
ac97_ctrl	40210	25803	103	7	19.4	26.4	64.3	1.7 (4.1×)	3 (6.5×)	4.7 (**5.6**×)	47.2 (-27%)
aes_core	66221	43022	172	13.2	56.1	69.3	104.7	3.3 (4×)	6 (9.4×)	9.3 (**7.5**×)	75.9 (-28%)
des_perf	295808	189276	757	82.1	260.3	342.4	447.1	13.4 (6.1×)	30.8 (8.5×)	44.2 (**7.7**×)	320.8 (-28%)
vga_lcd	397806	473772	1000	99.6	712	811.6	778.7	5.7 (17.5×)	30.5 (23.3×)	36.2 (**22.4**×)	538.5 (-31%)
netcard	3901343	2402788	5000	1612.4	3012.1	4624.5	4308.9	440.2 (3.7×)	209.5 (14.4×)	649.7 (**7.1**×)	3466.2 (-20%)

Sfxt: suffix tree update runtime   Pfxt: prefix tree expansion runtime

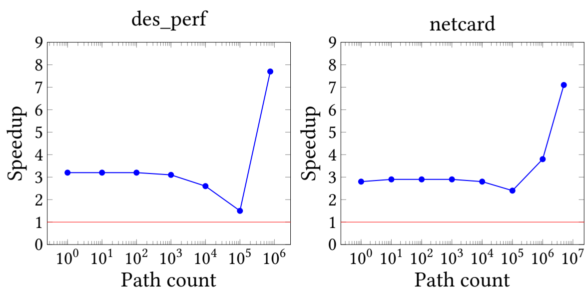


Figure 6: Speedup vs path count for des_perf and netcard.

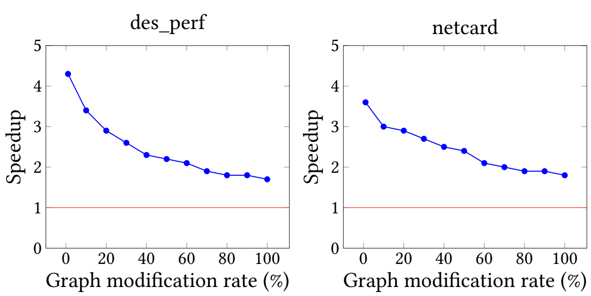


Figure 7: Speedup vs incrementality for des_perf and netcard.

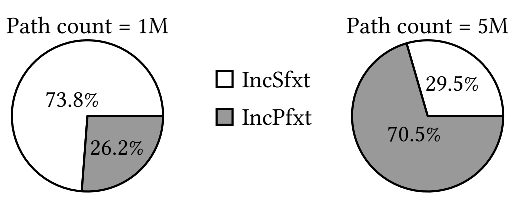


Figure 8: Speedup breakdown of Algorithm 1 (IncSfxt) and Algorithm 4 (IncPfxt) at different path counts.

Ink's overall speedup. As we increase the path count, prefix tree expansion starts to dominate the performance, but the path count is not large enough for Algorithm 4 to become effective; thus, Ink's overall speedup decreases. As we further increase the path count, Algorithm 4 exhibits a large speedup over full prefix tree expansion; thus, Ink's overall speedup increases.

4.3 Performance at Different Incrementalities

Figure 7 demonstrates the speedup of Ink over full CPG at different graph modification rates for des_perf and netcard. As we increase the graph modification rate, the speedup drops accordingly. For example, Ink's speedup drops from 3.6× to 1.8× in netcard. This is because the higher the graph modification rate, the more nodes that Ink needs to visit in Algorithm 2. Ink is most effective at a low graph modification rate. For example, Ink is over 4× faster in des_perf at 1% graph modification rate. This emphasizes Ink's benefit because realistically one incremental iteration involves only modifying far less than 1% of the gates in the circuit. On the contrary, Ink is still faster at 100% graph modification rate. For example, Ink is almost 2× faster in netcard at 100% graph modification rate. This is because even if the whole circuit is updated, it is very likely that many critical path traces remain the same. Ink only needs to update the path delays, which largely reduces memory reallocation overhead.

4.4 Speedup Breakdown of IncSfxt and IncPfxt

Figure 8 demonstrates the speedup breakdown of Algorithm 1 (IncSfxt) and Algorithm 4 (IncPfxt) for netcard. As we increase the path count from one million to five million, the speedup of Algorithm 4 becomes more remarkable. For example, the speedup of Algorithm 4 increases from 26.2% to 70.5%. This is because the efficiency of Algorithm 1 is constrained by the size of the affected subgraph of the suffix tree. On the contrary, the larger the path count, the more paths we can reuse, which increases the benefit of Algorithm 4.

5 CONCLUSION

In this paper, we have introduced Ink, an efficient incremental k-critical path generation algorithm. We have presented an incremental suffix tree update algorithm that minimally identifies the affected subgraph of the suffix tree and performs only necessary updates on the shortest path values. We have presented an incremental prefix tree expansion algorithm that identifies reusable paths from the previous CPG query and effectively prunes the path search space. Compared to a state-of-the-art timer, Ink is up to 22.4× faster and consumes up to 31% less memory when generating one million critical paths on a large design. We plan to extend Ink to support circuit modifiers that change the circuit topology in the future.

REFERENCES

[1] Jayaram Bhasker and Rakesh Chadha. 2009. *Static Timing Analysis for Nanometer Designs: A Practical Approach.* Springer.

[2] Zizheng Guo and Yibo Lin. 2022. Differentiable-Timing-Driven Global Placement. In *ACM/IEEE DAC*.

[3] Jin Hu, Greg Schaeffer, and Vibhor Garg. 2015. TAU 2015 contest on incremental timing analysis. In *ACM/IEEE ICCAD*.

[4] Tsung-Wei Huang, Guannan Guo, Chun-Xun Lin, and Martin D. F. Wong. 2021. OpenTimer v2: A New Parallel Incremental Timing Analysis Engine. *IEEE TCAD* (2021).

[5] Myung-Chul Kim, Jin Hu, Jiajia Li, and Natarajan Viswanathan. 2015. ICCAD-2015 CAD contest in incremental timing-driven placement and benchmark suite. In *IEEE/ACM ICCAD*.

[6] Pei-Yu Lee, Iris Hui-Ru Jiang, Cheng-Ruei Li, Wei-Lun Chiu, and Yu-Ming Yang. 2015. iTimerC 2.0: Fast incremental timing and CPPR analysis. In *ACM/IEEE ICCAD*.

[7] Dimitrios Mangiras, David Chinnery, and Giorgos Dimitrakopoulos. 2023. Task-Based Parallel Programming for Gate Sizing. *IEEE TCAD* (2023).

[8] Muhammet Mustafa Ozdal, Chirayu Amin, Andrey Ayupov, Steven Burns, Gustavo Wilke, and Cheng Zhuo. 2012. The ISPD-2012 Discrete Cell Sizing Contest and Benchmark Suite. In *ACM ISPD*.

[9] Chaitanya Peddawad, Aman Goel, Dheeraj B, and Nitin Chandrachoodan. 2015. iitRACE: A memory efficient engine for fast incremental timing analysis and clock pessimism removal. In *ACM/IEEE ICCAD*.